Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A

Document number	DDI0616
Document quality	EAC
Document version	B.a
Document confidentiality	Non-confidential
Document build information	5084c744 Tuesday, 22 A ust 20 40

This document is now RETIRED.
The Arm Architecture Reference Manual for A-profile architecture (DDI0487) is the definitive reference for this architecture specification.

Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A

Release information

Date	Version	Changes
2023/Aug/18	B.a	• Second release, including SME2 and EAC maintenance updates to SME.
2022/Feb/07	A.a	• First release.

Non-Confidential Proprietary Notice

Product Status

The information in this document is final; that is, it is for a developed product.

The information in this Manual is at EAC quality, which means that:

- All features of the specification are described in the manual.
- Information can be used for software and hardware development.

Contents

Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A

Chapter E4 Glossary terms

Preface

About this supplement

I_RFSSZ This supplement is the *Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A*.

I_LZCVC This supplement describes the changes and additions introduced by SME to the Armv9-A architecture.

I_FMHRZ This supplement also describes the changes and additions introduced by *The Scalable Matrix Extension version 2* (SME2) to the Armv9-A architecture.

I_HMDXX In this supplement, unless stated otherwise, when SME is used, the behavi[illegible]pplies to SME2.

I_CDMPR For SME, this supplement is to be read with the following documents:

- *Arm® Architecture Reference Manual for A-profile architecture* [1]
- *Arm® Architecture Registers, for A-profile architecture* [2]
- *Arm® A64 Instruction Set Architecture, for A-profile architecture* [3]

Together, the supplement and these documents provide a full description of the Armv9-A Scalable Matrix Extension, and the Armv9-A Scalable Matrix Extension version 2.

This supplement is organized into parts:

- SME Application Level Programmers' Model

  Describes how the PE at an application level is altered by the implementation of SME.

- SME System Level Programmers' Model

  Describes how the PE at system level is altered by the implementation of SME.

- SME Instruction Set

  Describes the extensions made for SME to the A64 instruction set.

- Appendices

  Provides reference information relating to the SME. This includes summarized information about the instruction set, implementation defined pseudocode and System register data, and a glossary that defines terms used in this document that have a specialized meaning.

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists where appropriate.

`monospace`

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Blue text

Indicates a link. This can be:

- A cross-reference to another location within the document
- A URL, for example http://developer.arm.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Asterisks in instruction mnemonics

Some behavior descriptions in this manual apply to a group of similar instructions that start with the same characters. In these situations, an * might be inserted at the end of a series of characters as a wildcard.

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font.

Rules-based writing

This specification consists of a set of individual *content items*. A content item is classified as one of the following:

- Declaration
- Rule
- Goal
- Information

Declarations and Rules are normative statements. An implementation that is compliant with this specification must conform to all Declarations and Rules in this specification that apply to the implementation.

Declarations and Rules must not be read in isolation. Where a particular feature is specified by multiple Declarations and Rules, these are generally grouped into sections and subsections that provide context. Where appropriate, these sections begin with a short introduction.

Arm strongly recommends that implementers read *all* chapters and sections of this document to ensure that an implementation is compliant.

Content items other than Declarations and Rules are informative statements. These are provided as an aid to understanding this specification.

Content item identifiers

A content item may have an associated identifier which is unique among content items in this specification.

After this specification reaches Beta status, a given content item has the same identifier across subsequent versions of the specification.

Content item rendering

In this document, a content item is rendered with a token of the following format in the left margin: L_{iiiii}

- L is a label that indicates the content class of the content item.
- $iiiii$ is the identifier of the content item.

Content item classes

Declaration

A Declaration is a statement that does one or more of the following:

- Introduces a concept
- Introduces a term
- Describes the structure of data
- Describes the encoding of data

A Declaration does not describe behavior.

A Declaration is rendered with the label *D*.

Rule

A Rule is a statement that describes the behavior of a compliant implementation.

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label *R*.

Goal

A Goal is a statement about the purpose of a set of rules.

A Goal explains why a particular feature has been included in the specification.

A Goal is comparable to a "business requirement" or an "emergent property."

A Goal is intended to be upheld by the logical conjunction of a set of rules.

A Goal is rendered with the label *G*.

Information

An Information statement provides information and guidance and an aid to understanding the specification.

An Information statement is rendered with the label *I*.

Additional reading

This section lists publications by Arm and by third parties.

See Arm Developer (https://developer.arm.com) for access to Arm documentation.

[1] *Arm® Architecture Reference Manual for A-profile architecture*. (ARM DDI 0487) Arm Ltd.

[2] *Arm® Architecture Registers, for A-profile architecture*. (ARM DDI 0601) Arm Ltd.

[3] *Arm® A64 Instruction Set Architecture, for A-profile architecture*. (ARM DDI 0602) Arm Ltd.

[4] *Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture*. (ARM DDI 0598) Arm Ltd.

[5] *Arm® Architecture Reference Manual Supplement, The Transactional Memory Extension (TME), for A-profile architecture*. (ARM DDI 0617) Arm Ltd.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have any comments or queries about our documentation, create a ticket at https://support.developer.arm.com.

As part of the ticket, include:

- The title, (Arm® Architecture Reference Manual Supplement, The Scalable Matrix Extension (SME), for Armv9-A).
- The number, (DDI0616 B.a).
- The section name to which your comments refer.
- The page number(s) to which your comments refer.
- The rule identifier(s) to which your comments refer, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Inclusive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact terms@arm.com.

Part A
Introduction

Chapter A1
SME Introduction

A1.1 About the Scalable Matrix Extension

I_YVCPK The *Scalable Matrix Extension* (SME) defines:

- Architectural state capable of holding two-dimensional matrix tiles.
- A Streaming SVE processing mode, which supports execution of SVE2 instructions with a vector length that matches the tile width.
- Instructions that accumulate the outer product of two vectors into a tile.
- Load, store, and move instructions that transfer a vector to or from a tile row or column.

The extension also defines System registers and fields that identify the presence and capabilities of SME, and enable and control its behavior at each Exception level.

I_PYQMB The *Scalable Matrix Extension version 2* (SME2) extends the SME architecture to increase the number of applications that can benefit from the computational efficiency of SME, beyond its initial focus on outer products and matrix-matrix multiplication.

SME2 adds data processing instructions with multi-vector operands and a multi-vector predication mechanism. These include:

- Multi-vector multiply-accumulate instructions, that read SVE Z vectors and accumulate into *ZA array vectors* to permit reuse of the SME outer product hardware for vector operations, including widening multiplies that accumulate into more vectors than they read.
- Multi-vector load, store, move, permute, and convert instructions, that read and write multiple SVE Z vectors to preprocess inputs and post-process outputs of the multi-vector multiply-accumulate instructions.
- An alternative predication mechanism to the SVE predication mechanism, to control operations performed on multiple vector registers.

SME2 also adds:

- A Range Prefetch hint instruction to prepare the memory system to prefetch and retain a set of strided address ranges in the most appropriate cache levels.
- Compressed neural network capability using dedicated lookup table instructions and outer product instructions that support binary neural networks.
- A 512-bit architectural register, *ZT0*, to support the lookup table feature.

I_{SQCGB} Unless otherwise specified by this document, the behaviors of instructions and architectural state when the PE is in *Streaming SVE mode* are as described in *Arm® Architecture Reference Manual for A-profile architecture* [1].

Chapter A2
Architecture Features and Extensions

A2.1 Extensions and features defined by SME

SME inherits the rules for architectural features and extensions from *Arm® Architecture Reference Manual for A-profile architecture* [1]. This specification describes changes to those rules, and defines any features added by SME.

R_{PDXHJ} SME is represented by the feature FEAT_SME.

R_{QFSVK} FEAT_SME is an OPTIONAL extension from Armv9.2.

I_{VQCZZ} The following list summarizes the OPTIONAL SME features:

- FEAT_SME_FA64, support the full A64 instruction set in *Streaming SVE mode*.
- FEAT_SME_F64F64, Double-precision floating-point outer product instructions.
- FEAT_SME_I16I64, 16-bit to 64-bit integer widening outer product instructions.
- FEAT_EBF16, support for Extended BFloat16 mode.

R_{KXCXC} FEAT_SME_FA64 requires FEAT_SVE2.

R_{KQDDR} SME2 represents a version of the SME architecture that implements FEAT_SME2.

R_{SZZTV} FEAT_SME2 is an OPTIONAL extension from Armv9.2.

R_{BCCQL} FEAT_SME2 requires FEAT_SME.

A2.2 Changes to existing features and extension requirements

R_{DSHWS} If SME is implemented, the following features are also implemented:

- FEAT_HCX.
- FEAT_FGT.
- FEAT_FCMA.
- FEAT_FP16.
- FEAT_FHM.
- FEAT_BF16.

Part B
SMP Application Level Programmers' Model

Chapter B1
Application processing modes

B1.1 Overview

SME extends the AArch64 application level programmers' model with added processing modes and related instructions, architectural state, and registers:

- The PSTATE.SM control to enable an execution mode, known as *Streaming SVE mode*.
- The PSTATE.ZA control to enable access to *ZA storage*, and to the *ZT0 register* when SME2 is implemented. The Special-purpose register, SVCR, which provides read/write access to PSTATE.SM and PSTATE.ZA from any Exception level.
- The SMSTART and SMSTOP instructions, aliases of the MSR (immediate) instruction, that can set or clear PSTATE.SM, PSTATE.ZA, or both PSTATE.SM and PSTATE.ZA from any Exception level.
- A software thread ID register to manage per-thread SME context, TPIDR2_EL0.

B1.2 Process state

D_{XDPXS} A PE that implements SME has a *Streaming SVE mode*.

D_{JYVLM} Streaming SVE register state is the vector registers *Z0-Z31* and predicate registers *P0-P15* that can be accessed by SME, SVE, Advanced SIMD, and floating-point instructions when the PE is in *Streaming SVE mode*.

D_{DMZFR} Streaming SVE register state includes the SVE *FFR* predicate register if FEAT_SME_FA64 is implemented and enabled at the current Exception level.

I_{XXKGV} If SME is implemented, a PE has the following additional architectural state:

- Streaming SVE register state.
- *ZA* storage.
- When SME2 is implemented, the *ZT0* register.

I_{ZTTNW} A PE enters *Streaming SVE mode* to access Streaming SVE vector and predicate register state.

I_{NXQFB} If SME is implemented, this does not imply that FEAT_SVE and FEAT_SVE2 are implemented by the PE when it is not in *Streaming SVE mode*.

I_{RNTPX} When the PE is in *Streaming SVE mode*, a different set of vector lengths might be available for SVE instructions, as specified in C1.1.3 *Vector lengths*.

I_{TDSPN} When the PE is in *Streaming SVE mode*, the performance characteristics of some instructions might be significantly reduced, as specified in E1.3 *Reduced performance in Streaming SVE mode*.

I_{NWFQX} SME extends a PE's *Process state* or PSTATE with the PSTATE.SM and PSTATE.ZA fields. The PSTATE fields can be modified by the SMSTART and SMSTOP instructions, and can also be read and written using the SVCR register.

I_{DHSSW} The PSTATE.SM field controls the mode of *Streaming SVE mode*.

I_{NVWRT} The PSTATE.ZA field controls all of the following:

- Access to *ZA* storage.
- Access to the *ZT0* register, when SME2 is implemented.

I_{DVPDL} The SMSTART instruction does either or both of the following:

- Enters *Streaming SVE mode*.
- Enables the *ZA* storage, and when SME2 is implemented enables the *ZT0* register.

I_{QQZTL} The SMSTOP instruction does either or both of the following:

- Exits *Streaming SVE mode*.
- Disables the *ZA* storage, and when SME2 is implemented disables the *ZT0* register.

I_{NKJKL} After entering *Streaming SVE mode*, subsequent SMSTART and SMSTOP instructions might be used to enable and disable the *ZA* storage, and the *ZT0* register when SME2 is implemented, for different phases of execution within *Streaming SVE mode*, before using a final SMSTOP instruction to exit *Streaming SVE mode*.

D_{WHXDZ} SME and SME2 instructions are the instructions defined by the SME architecture in Chapter D1 *SME instructions*.

D_{NHNFF} A *legal* instruction is an implemented instruction that can be executed by a PE when PSTATE.SM and PSTATE.ZA are in the required state, unless its execution at the current Exception level is prevented by a configurable trap or enable.

D_{HZFSG} An *illegal* instruction is an implemented instruction whose attempted execution by a PE when PSTATE.SM and PSTATE.ZA are not in the required state causes an SME illegal instruction exception to be taken, unless its execution at the current Exception level is prevented by a higher-priority configurable trap or enable.

See also:

- MSR (immediate).
- SMSTART.

- SMSTOP.
- C1.1.3 *Vector lengths*.
- C1.2.3 *Validity of SME and SVE state*.
- C1.3.4 *ESR_EL1, ESR_EL2, and ESR_EL3*.
- C1.4.8 *SVCR*.
- Chapter E1 *Instructions affected by SME*.

B1.2.1 PSTATE.SM

I_{YYQJK} The value of PSTATE.SM can be changed by executing the MSR instructions that access the SVCR. For more information, see B1.2.3 *Changing PSTATE.SM and PSTATE.ZA*.

D_{PRGHY} The PE is in *Streaming SVE mode* when the Effective value of PSTATE.SM is 1.

R_{VJZBC} When the PE is in *Streaming SVE mode*:

- Streaming SVE register state is valid.
- SME and SME2 instructions that access the Streaming SVE register state are *legal*.
- SME and SME2 instructions that do not access the *ZA* storage or *ZT0* register are *legal*.
- SME and SME2 instructions that access the *ZA* storage or *ZT0* register are *legal* if *ZA* storage is enabled.
- *Legal* instructions that access SVE or SIMD&FP registers access the Streaming SVE register state.

I_{YDRPH} The SVE *FFR* predicate register is not architecturally visible when the PE is in *Streaming SVE mode* if FEAT_SME_FA64 is not implemented or not enabled at the current Exception level.

R_{CKSBS} When the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:

- Most Advanced SIMD instructions are *illegal*, as described in E1.1.1 *Illegal Advanced SIMD instructions*.
- Some SVE and SVE2 instructions are *illegal*, as described in E1.1.2 *Illegal SVE instructions*.
- Most other instructions implemented by the PE, including scalar floating-point instructions, remain *legal*.

D_{DVDTY} The PE is not in *Streaming SVE mode* when the Effective value of PSTATE.SM is 0.

R_{XBBFD} When the PE is not in *Streaming SVE mode*:

- Streaming SVE register state is not valid.
- SME and SME2 instructions that access the Streaming SVE register state are *illegal*.
- SME LDR (vector), STR (vector), and ZERO (tile) instructions that access the *ZA* storage are *legal* if *ZA* is enabled, and all other instructions that access the *ZA* storage are *illegal*.
  SME2 LDR (ZT0), STR (ZT0), and ZERO (ZT0) instructions that access the *ZT0* register are *legal* if *ZA* is enabled, and all other instructions that access the *ZT0* register are *illegal*.
- The MSR and MRS instructions that directly access the SME SVCR register are *legal*.
- Instructions which access SVE or SIMD&FP registers access the Non-streaming SVE or SIMD&FP register state.
- All other instructions implemented by the PE are *legal*.

R_{RSWFQ} When the Effective value of PSTATE.SM is changed by any means from 0 to 1, an entry to *Streaming SVE mode* is performed, and each implemented bit of SVE registers *Z0-Z31*, *P0-P15*, and *FFR* in the new mode is set to zero.

R_{KFRQZ} When the Effective value of PSTATE.SM is changed by any means from 1 to 0, an exit from *Streaming SVE mode* is performed, and each implemented bit of SVE registers *Z0-Z31*, *P0-P15*, and *FFR* in the new mode is set to zero.

R_{MHTLZ} When the Effective value of PSTATE.SM is changed by any means from 0 to 1, or from 1 to 0, the FPSR is set to the value 0x0000_0000_0800_009f, in which all of the cumulative status bits are set to 1.

I_{YTZVD} Statements which refer to the value of the SVE vector registers, *Z0-Z31*, implicitly also refer to the lower bits of those registers accessed by the SIMD&FP register names *V0-V31*, *Q0-Q31*, *D0-D31*, *S0-S31*, *H0-H31*, and *B0-B31*.

See also:

- C1.1.2 *Traps and exceptions*.
- C1.4.8 *SVCR*.

B1.2.2 PSTATE.ZA

I_{GJZLD} The value of PSTATE.ZA can be changed by executing the MSR instructions that access the SVCR. For more information, see B1.2.3 *Changing PSTATE.SM and PSTATE.ZA*.

D_{HBFWD} The following are enabled when PSTATE.ZA is 1:

- The *ZA* storage.
- When SME2 is implemented, the *ZT0* register.

R_{SFWMY} When *ZA* storage is enabled:

- The contents of *ZA* storage, and the *ZT0* register when SME2 is implemented, are valid and are retained by hardware irrespective of whether the PE is in *Streaming SVE mode*.
- SME and SME2 instructions that access the *ZA* storage or the *ZT0* register are *legal* and can be executed, unless execution is prevented by some other trap or exception.

D_{VLMFC} The following are disabled when PSTATE.ZA is 0:

- The *ZA* storage.
- When SME2 is implemented, the *ZT0* register.

R_{JHMYL} When *ZA* storage is disabled:

- The contents of *ZA* storage, and the *ZT0* register when SME2 is implemented, are not valid.
- SME and SME2 instructions that access the *ZA* storage or the *ZT0* register are *illegal*.
- There is no effect on other instructions implemented by the PE.

R_{YRZRM} When PSTATE.ZA is changed by this means from 0 to 1, all implemented bits of the *ZA* storage, and the *ZT0* register when SME2 is implemented, are set to zero.

I_{LRDZR} When PSTATE.ZA is changed from 1 to 0, there is no architecturally defined effect on the *ZA* storage, and the *ZT0* register when SME2 is implemented, because the contents of *ZA* storage and the *ZT0* register cannot be observed when PSTATE.ZA is 0.

I_{QWCJS} When PSTATE.ZA is changed from 0 to 1, or 1 to 0, there is no effect on the SVE vector and predicate registers and the FPSR. PSTATE.SM is not changed.

See also:

- B2.6 *SME2 Lookup table*.
- C1.1.2 *Traps and exceptions*.
- C1.4.8 *SVCR*.

B1.2.3 Changing PSTATE.SM and PSTATE.ZA

D_{QRSXV} The following MSR (immediate) instructions are provided to independently set or clear PSTATE.SM, PSTATE.ZA, or both PSTATE.SM and PSTATE.ZA respectively:

- `MSR SVCRSM, #<imm1>.`
- `MSR SVCRZA, #<imm1>.`
- `MSR SVCRSMZA, #<imm1>.`

R_{MPQWY} `MSR SVCRSM`, `MSR SVCRZA`, and `MSR SVCRSMZA` instructions are permitted to be executed from any Exception level.

D_{YGDXX} The `SMSTART` instruction is the preferred alias of the following instructions:

- `MSR SVCRSM, #1.`

- `MSR SVCRZA, #1`.
- `MSR SVCRSMZA, #1`.

D_{DZTDH} The `SMSTOP` instruction is the preferred alias of the following instructions:

- `MSR SVCRSM, #0`.
- `MSR SVCRZA, #0`.
- `MSR SVCRSMZA, #0`.

I_{SZYBC} Access to `SVCR` through the `MRS` and `MSR` (register) instructions might be used where a calling convention or ABI requires saving, restoring, or testing of `PSTATE.SM` and `PSTATE.ZA`, and are permitted to be executed from any Exception level. However, the `MSR` (immediate) instructions might be higher performance than the `MSR` (register) instruction, so the `MSR` (immediate) instructions are preferred for explicit changes to `PSTATE.SM` and `PSTATE.ZA`.

I_{HNNJR} The PE might consume less power when `PSTATE.SM` is 0 and `PSTATE.ZA` is 0.

See also:

- MSR (immediate).
- SMSTART.
- SMSTOP.
- C1.4.8 *SVCR*.

B1.2.4 TPIDR2_EL0

D_{FJMMT} If SME is implemented, the register `TPIDR2_EL0` is added.

I_{SLNHN} The Software Thread ID Register #2 provides additional thread identifying information that can be read and written from all Exception levels.

I_{QPMJN} This register is reserved for use by the ABI to manage per-thread SME context.

See also:

- TPIDR2_EL0.

Chapter B2
Architectural state

B2.1 Architectural state summary

D_{XJCGQ} The *Effective Streaming SVE vector length* (SVL) is a power of two in the range 128 to 2048 bits inclusive.

I_{NBPPM} When the PE is in *Streaming SVE mode*, the *Effective SVE vector length* (VL) is equal to SVL.

This might be different from the value of VL when the PE is not in *Streaming SVE mode*, as described in C1.1.3 *Vector lengths*.

D_{JBVYJ} In a vector, SVL bits:

- SVL_B is the number of 8-bit elements.
- SVL_H is the number of 16-bit elements.
- SVL_S is the number of 32-bit elements.
- SVL_D is the number of 64-bit elements.
- SVL_Q is the number of 128-bit elements.

SVL [bits]	SVL_B	SVL_H	SVL_S	SVL_D	SVL_Q
128	16	8	4	2	1
256	32	16	8	4	2
512	64	32	16	8	4
1024	128	64	32	16	8
2048	256	128	64	32	16

See also:

- Chapter B1 *Application processing modes*.
- C1.1.3 *Vector lengths*.

B2.2 SME ZA storage

D_{SSXPL} The *ZA storage* is an architectural register state consisting of a two-dimensional *ZA array* of [$SVL_B \times SVL_B$] bytes.

B2.2.1 ZA array vector access

R_{FFWNB} The *ZA* array can be accessed as vectors of SVL bits.

D_{PPPCM} An untyped vector access to the *ZA* array is represented by ZA[*N*], where *N* is in the range 0 to SVL_B-1 inclusive.

D_{DTVZN} In SME `LDR` (vector) and `STR` (vector) instructions, an untyped *ZA array vector* is selected by the sum of a 32-bit general-purpose vector select register W*v* and an immediate vector select offset *offs*, modulo SVL_B.

D_{YXHFR} The preferred disassembly for an untyped *ZA* array vector is ZA[W*v*, *offs*], where *offs* is an immediate in the range 0-15 inclusive.

D_{CRJPC} The *ZA* array can be accessed as vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements.

D_{WMVZT} An elementwise vector access to the *ZA* array is indicated by appending a vector index "[*N*]" to the *ZA* array name and element size qualifier, where *N* is in the range 0 to SVL_B-1 inclusive, as follows:

- An 8-bit element vector access to the *ZA* array is represented by ZA.B[*N*].
- A 16-bit element vector access to the *ZA* array is represented by ZA.H[*N*].
- A 32-bit element vector access to the *ZA* array is represented by ZA.S[*N*].
- A 64-bit element vector access to the *ZA* array is represented by ZA.D[*N*].
- A 128-bit element vector access to the *ZA* array is represented by ZA.Q[*N*].

B2.2.2 ZA tile access

D_{VSVMX} A *ZA tile* is a square, two-dimensional sub-array of elements within the *ZA* array.

I_{WLRTV} Depending on the element size with which it is accessed, the ZA array is treated as containing one or more *ZA* tiles, as described in the following sections.

D_{DWMYT} A *ZA* tile is indicated by appending the tile number to the *ZA* name.

D_{ZGBHT} A *ZA tile slice* is a one-dimensional set of horizontally or vertically contiguous elements within a *ZA* tile.

R_{PZNWB} A vector access to a tile reads or writes a *ZA* tile slice.

I_{NFXHH} A *ZA* tile can be accessed as vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements.

I_{YZDBS} A *ZA* tile can be accessed as horizontal slices of SVL bits.

R_{GPVSZ} A *ZA* tile is accessed as horizontal slices if the V field in the accessing instruction opcode is 0.

D_{TRHTX} An access to horizontal tile slices is indicated by an "H" suffix on the *ZA* tile name.

I_{HBYTT} A *ZA* tile can be accessed as vertical slices of SVL bits.

R_{GPPPK} A *ZA* tile is accessed as vertical slices if the V field in the accessing instruction opcode is 1.

D_{WSBVG} An access to vertical tile slices is indicated by a "V" suffix on the *ZA* tile name.

R_{TWWTL} In SME instructions, the tile slice is selected by the sum of a 32-bit general-purpose slice index register W*s* and an immediate slice index offset *offs*, modulo the number of slices in the named tile.

B2.2.3 Accessing an 8-bit element ZA tile

D_{HMSNH} An 8-bit element *ZA* tile is indicated by a ".B" qualifier following the tile name.

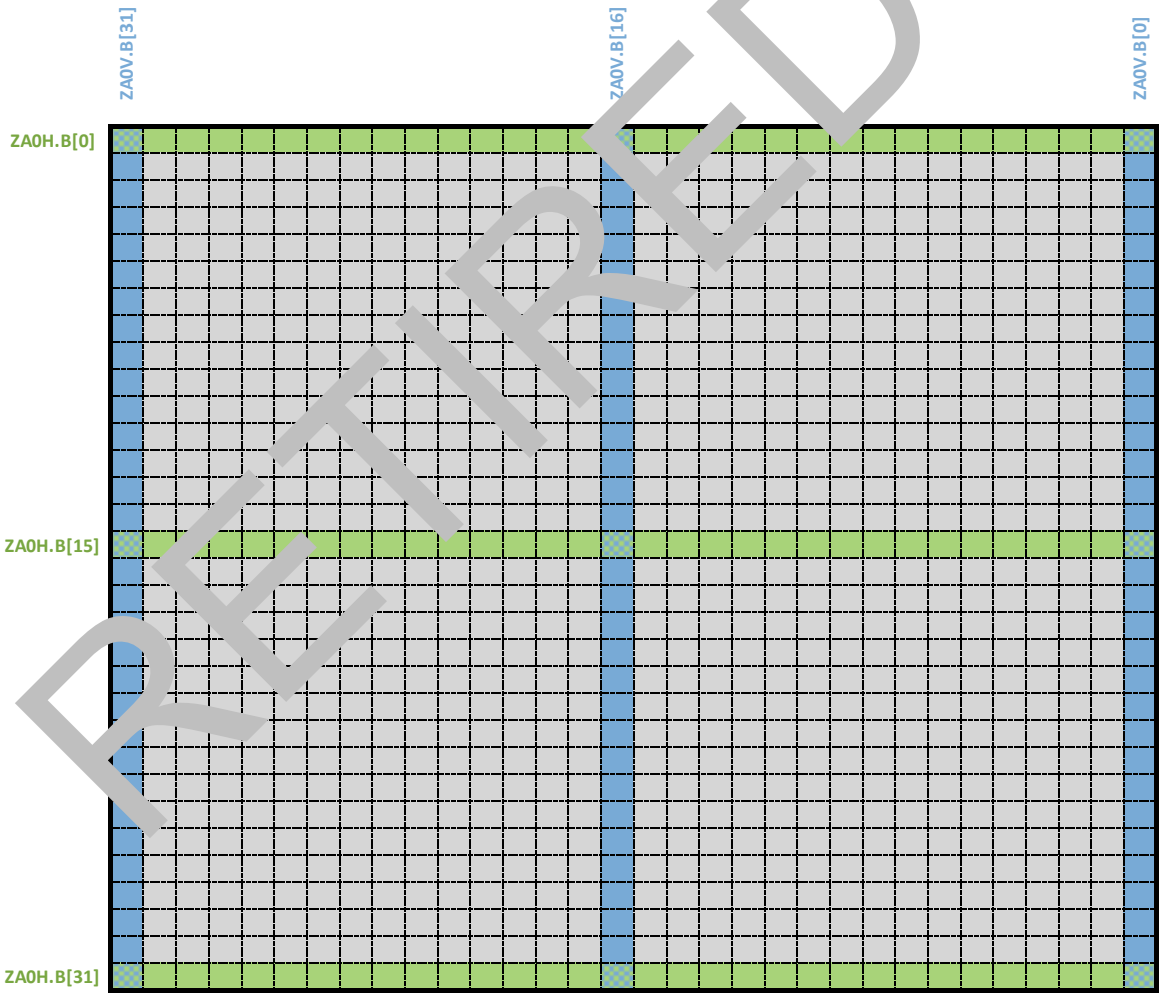
D_{NLCNH} There is a single tile named ZA0.B which consists of [$SVL_B \times SVL_B$] 8-bit elements and occupies all of the *ZA* storage.

R_{NBSMJ} An access to a horizontal or vertical 8-bit element ZA tile slice reads or writes SVL_B 8-bit elements.

D_{NMHLM} An access to a horizontal or vertical 8-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *N* is in the range 0 to SVL_B-1 inclusive:

- ZA0H.B[*N*] indicates a horizontal 8-bit element *ZA* tile slice selection.
- ZA0V.B[*N*] indicates a vertical 8-bit element *ZA* tile slice selection.

I_{JVTNY} Horizontal and vertical ZA0.B slice accesses are illustrated in the followi m for SVL of 256 bits:



R_{DCSDX} An access to the horizontal slice ZA0H.B[*N*] reads or writes the SVL_B bytes in *ZA* array vector ZA.B[*N*].

R_{FHYSQ} An access to the vertical slice ZA0V.B[*N*] reads or writes the 8-bit element [*N*] within each horizontal slice of ZA0.B.

D_{CDDVV} The preferred disassembly is:

- ZA0H.B[W*s*, *offs*], for a horizontal 8-bit element *ZA* tile slice selection.

- ZA0V.B[W*s*, *offs*], for a vertical 8-bit element *ZA* tile slice selection.

Where *offs* is an immediate in the range 0-15 inclusive.

B2.2.4 Accessing a 16-bit element ZA tile

D_{LNXPD} A 16-bit element *ZA* tile is indicated by a ".H" qualifier following the tile name.

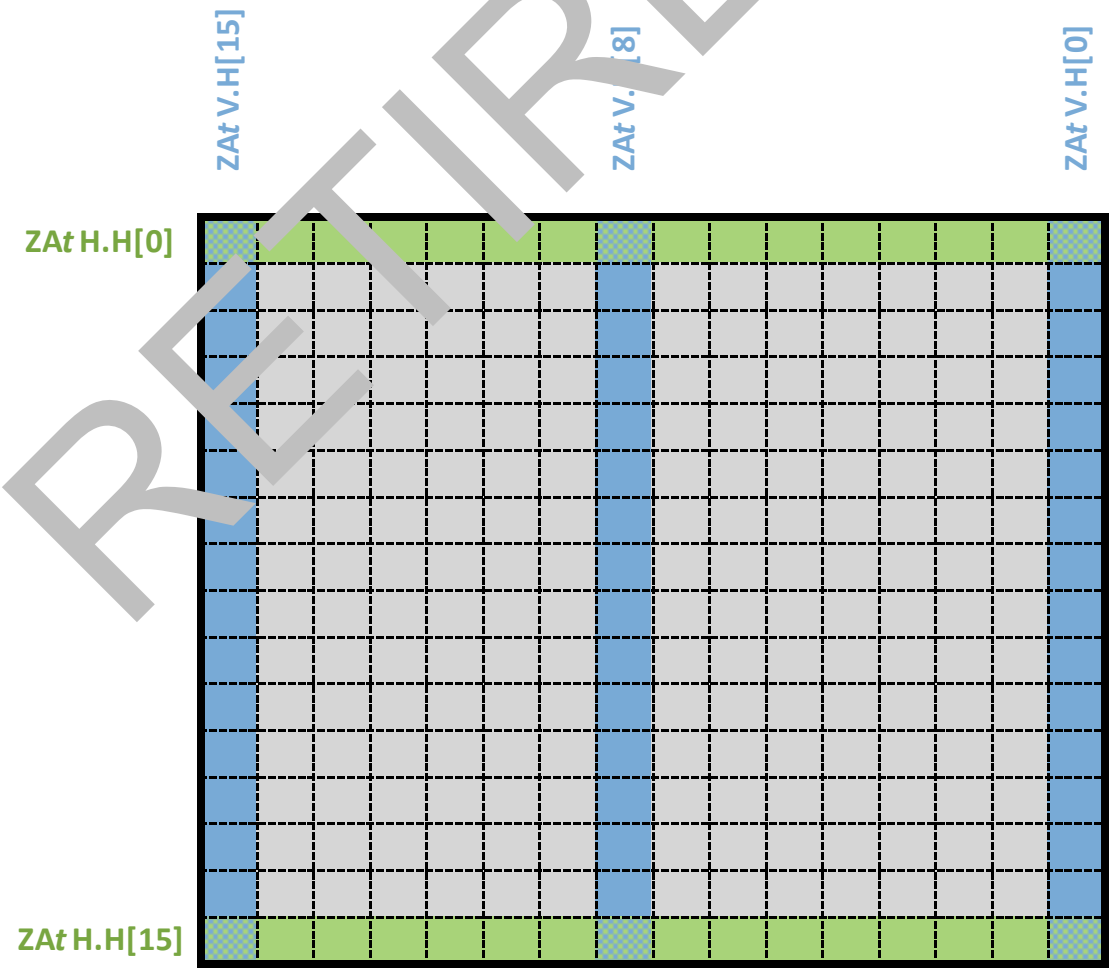
D_{GWZDM} There are two tiles named ZA0.H and ZA1.H. Each tile consists of [$SVL_H \times SVL_H$] 16-bit elements, and occupies half of the *ZA* storage.

R_{NMGXG} An access to a horizontal or vertical 16-bit element ZA tile slice reads or writes SVL_H 16-bit elements.

D_{DHKMC} An access to a horizontal or vertical 16-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is 0 or 1, and *N* is in the range 0 to SVL_H-1 inclusive:

- ZA*t*H.H[*N*] indicates a horizontal 16-bit element *ZA* tile slice selection.
- ZA*t*V.H[*N*] indicates a vertical 16-bit element *ZA* tile slice selection.

I_{ZSWJW} Horizontal and vertical ZA*t*.H slice accesses, where *t* is 0 or 1, are illustrated in the following diagram for SVL of 256 bits:



R_{BTLQC} An access to the horizontal slice ZA*t*H.H[*N*] reads or writes the SVL_H 16-bit elements in *ZA* array vector ZA.H[*t* + 2 * *N*].

R_{NGJBJ} An access to the vertical slice ZA*t*V.H[*N*] reads or writes the 16-bit element [*N*] within each horizontal slice of ZA*t*.H.

D_{RHQJT} The preferred disassembly is as follows:

- ZA*t*H.H[W*s*, *offs*], for a horizontal 16-bit element *ZA* tile slice selection.
- ZA*t*V.H[W*s*, *offs*], for a vertical 16-bit element *ZA* tile slice selection.

Where *t* is 0 or 1, and *offs* is an immediate in the range 0-7 inclusive.

B2.2.5 Accessing a 32-bit element ZA tile

D_{HBKZV} A 32-bit element *ZA* tile is indicated by a ".S" qualifier following the tile name.

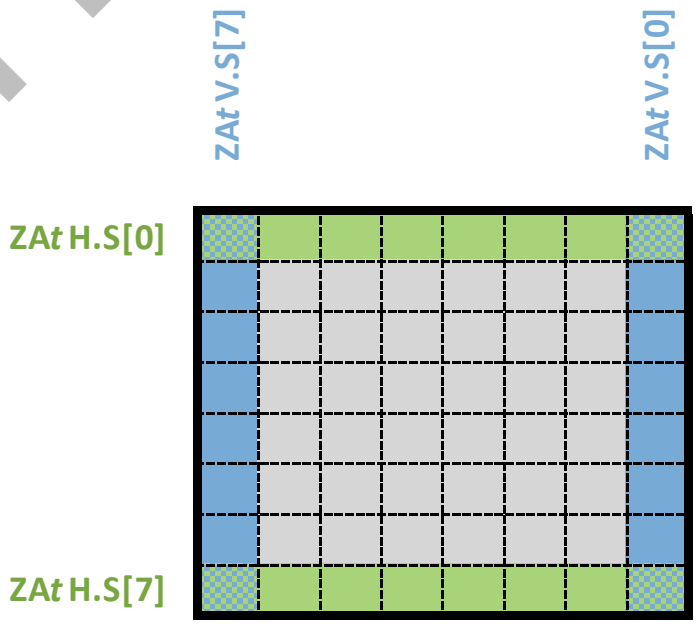
D_{RDRRT} There are four tiles named ZA0.S, ZA1.S, ZA2.S, and ZA3.S. Each tile consists of [SVL × SVL_S] 32-bit elements, and occupies a quarter of the *ZA* storage.

R_{XFPPL} An access to a horizontal or vertical 32-bit element ZA tile slice reads or writes SVL_S 32-bit elements.

D_{JFPSJ} An access to a horizontal or vertical 32-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *N* is 0, 1, 2, or 3, and *N* is in the range 0 to SVL_S-1 inclusive:

- ZA*t*H.S[*N*] indicates a horizontal 32-bit element ZA tile slice selection.
- ZA*t*V.S[*N*] indicates a vertical 32-bit element ZA tile slice selection.

I_{SZXZR} Horizontal and vertical ZA*t*.S slice accesses where *N* is 0, 1, 2, or 3, are illustrated in the following diagram for SVL of 256 bits:



R_{JBJZY} An access to the horizontal slice ZA*t*H.S[*N*] reads or writes the SVL_S 32-bit elements in *ZA* array vector ZA.S[*t* + 4 * *N*].

R_{GBYSJ}

An access to the vertical slice ZA*t*V.S[*N*] reads or writes the 32-bit element [*N*] within each horizontal slice of ZA*t*.S.

D_{LQLJH} The preferred disassembly is:

- ZA*t*H.S[W*s*, *offs*], for a horizontal 32-bit element *ZA* tile slice selection.
- ZA*t*V.S[W*s*, *offs*], for a vertical 32-bit element *ZA* tile slice selection.

Where *t* is 0, 1, 2, or 3, and *offs* is 0, 1, 2, or 3.

B2.2.6 Accessing a 64-bit element ZA tile

D_{TWMMM} A 64-bit element *ZA* tile is indicated by a ".D" qualifier following the tile name.

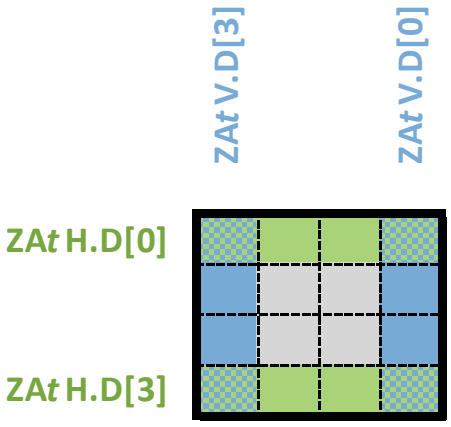
D_{THPSD} There are eight tiles named ZA0.D, ZA1.D, ZA2.D, ZA3.D, ZA4.D, ZA5.D, ZA6.D and ZA7.D. Each tile consists of [$SVL_D \times SVL_D$] 64-bit elements, and occupies an eighth of the *ZA* storage.

R_{ZXYBQ} An access to a horizontal or vertical 64-bit element ZA tile slice reads or writes SVL_D 64-bit elements.

D_{DCXSX} An access to a horizontal or vertical 64-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is in the range 0-7 inclusive, and *N* is in the range 0 to SVL_D-1 inclusive:

- ZA*t*H.D[*N*] indicates a horizontal 64-bit element *ZA* tile slice selection.
- ZA*t*V.D[*N*] indicates a vertical 64-bit element *ZA* tile slice selection.

I_{LGJZC} Horizontal and vertical ZA*t*.D slice accesses, where *t* is in the range 0-7 inclusive, are illustrated in the following diagram for SVL of 256 bits:



R_{CVVJK} An access to the horizontal slice ZA*t*H.D[*N*] reads or writes the SVL_D 64-bit elements in *ZA* array vector ZA.D[*t* + 8 * *N*].

R_{JYQKK} An access to the vertical slice ZA*t*V.D[*N*] reads or writes the 64-bit element [*N*] within each horizontal slice of ZA*t*.D.

D_{MQQPX} The preferred disassembly is:

- ZA*t*H.D[W*s*, *offs*], for a horizontal 64-bit element *ZA* tile slice selection.
- ZA*t*V.D[W*s*, *offs*], for a vertical 64-bit element *ZA* tile slice selection.

Where *t* is in the range 0-7 inclusive, and *offs* is 0 or 1.

B2.2.7 Accessing a 128-bit element ZA tile

D_{GZDSH} A 128-bit element *ZA* tile is indicated by a ".Q" qualifier following the tile name.

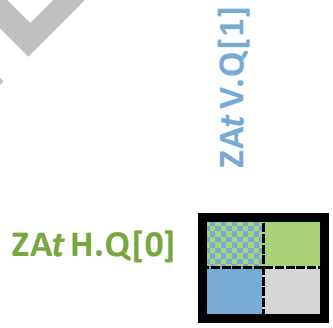
D_{RPMJL} There are 16 tiles named ZA0.Q, ZA1.Q, ZA2.Q, ZA3.Q, ZA4.Q, ZA5.Q, ZA6.Q, ZA7.Q, ZA8.Q, ZA9.Q, ZA10.Q, ZA11.Q, ZA12.Q, ZA13.Q, ZA14.Q, and ZA15.Q. Each tile consists of [$SVL_Q \times SVL_Q$] 128-bit elements, and occupies 1/16 of the *ZA* storage.

R_{QGHPF} An access to a horizontal or vertical 128-bit element ZA tile slice reads or writes SVL_Q 128-bit elements.

D_{RLQKW} An access to a horizontal or vertical 128-bit element ZA tile slice is indicated by appending a slice index "[*N*]" to the tile name, direction suffix, and qualifier. For example, where *t* is in the range 0-15 inclusive, and *N* is in the range 0 to SVL_Q-1 inclusive:

- ZA*t*H.Q[*N*] indicates a horizontal 128-bit element *ZA* tile slice selection.
- ZA*t*V.Q[*N*] indicates a vertical 128-bit element *ZA* tile slice selection.

I_{YQPWS} Horizontal and vertical ZA*t*.Q slice accesses, where *t* is in the range 0-15 inclusive, are illustrated in the following diagram for SVL of 256 bits:



R_{PJTQJ} An access to the horizontal slice ZA*t*H.Q[*N*] reads or writes the SVL_Q 128-bit elements in *ZA* array vector ZA.Q[*t* + 16 * *N*].

R_{TRJFZ} An access to the vertical slice ZA*t*V.Q[*N*] reads or writes the 128-bit element [*N*] within each horizontal slice of ZA*t*.Q.

D_{VCLJP} The preferred disassembly is:

- ZA*t*H.Q[W*s*, 0], for a horizontal 128-bit element *ZA* tile slice selection.
- ZA*t*V.Q[W*s*, 0], for a vertical 128-bit element *ZA* tile slice selection.

Where *t* is in the range 0-15 inclusive, and the slice index offset is always zero.

B2.3 ZA storage layout

B2.3.1 ZA array vector and tile slice mappings

I_{PYTLW} Each horizontal tile slice corresponds to one *ZA* array vector.

The horizontal slice mappings for all tile sizes are illustrated by this table:

ZA Array Vector	8-bit element Tile Horizontal Slice	16-bit element Tile Horizontal Slice	32-bit element Tile Horizontal Slice	64-bit element Tile Horizontal Slice	128-bit element Tile Horizontal Slice
ZA[0]	ZA0H.B[0]	ZA0H.H[0]	ZA0H.S[0]	ZA0H.D[0]	ZA0H.Q[0]
ZA[1]	ZA0H.B[1]	ZA1H.H[0]	ZA1H.S[0]	ZA1H.D[0]	ZA1H.Q[0]
ZA[2]	ZA0H.B[2]	ZA0H.H[1]	ZA2H.S[0]	ZA2H.D[0]	ZA2H.Q[0]
ZA[3]	ZA0H.B[3]	ZA1H.H[1]	ZA3H.S[0]	ZA3H.D[0]	ZA3H.Q[0]
ZA[4]	ZA0H.B[4]	ZA0H.H[2]	ZA0H.S[1]	ZA4H.D[0]	ZA4H.Q[0]
ZA[5]	ZA0H.B[5]	ZA1H.H[2]	ZA1H.S[1]	ZA5H.D[0]	ZA5H.Q[0]
ZA[6]	ZA0H.B[6]	ZA0H.H[3]	ZA2H.S[1]	ZA6H.D[0]	ZA6H.Q[0]
ZA[7]	ZA0H.B[7]	ZA1H.H[3]	ZA3H.S[1]	ZA7H.D[0]	ZA7H.Q[0]
ZA[8]	ZA0H.B[8]	ZA0H.H[4]	ZA0H.S[2]	ZA0H.D[1]	ZA8H.Q[0]
ZA[9]	ZA0H.B[9]	ZA1H.H[4]	ZA1H.S[2]	ZA1H.D[1]	ZA9H.Q[0]
ZA[10]	ZA0H.B[10]	ZA0H.H[5]	ZA2H.S[2]	ZA2H.D[1]	ZA10H.Q[0]
ZA[11]	ZA0H.B[11]	ZA1H.H[5]	ZA3H.S[2]	ZA3H.D[1]	ZA11H.Q[0]
ZA[12]	ZA0H.B[12]	ZA0H.H[6]	ZA0H.S[3]	ZA4H.D[1]	ZA12H.Q[0]
ZA[13]	ZA0H.B[13]	ZA1H.H[6]	ZA1H.S[3]	ZA5H.D[1]	ZA13H.Q[0]
ZA[14]	ZA0H.B[14]	ZA0H.H[7]	ZA2H.S[3]	ZA6H.D[1]	ZA14H.Q[0]
ZA[15]	ZA0H.B[15]	ZA1H.H[7]	ZA3H.S[3]	ZA7H.D[1]	ZA15H.Q[0]
if applicable ZA[16] to ZA[SVL_B-1]	…	…	…	…	…

B2.3.2 Tile mappings

I_{YVYJP} The smallest *ZA* tile granule is the 128-bit element tile. When the *ZA* storage is viewed as an array of tiles, the larger 64-bit, 32-bit, 16-bit, and 8-bit element tiles overlap multiple 128-bit element tiles as follows:

Tile	Overlaps
ZA0.B	ZA0.Q, ZA1.Q, ZA2.Q, ZA3.Q, ZA4.Q, ZA5.Q, ZA6.Q, ZA7.Q, ZA8.Q, ZA9.Q, ZA10.Q, ZA11.Q, ZA12.Q, ZA13.Q, ZA14.Q, ZA15.Q
ZA0.H	ZA0.Q, ZA2.Q, ZA4.Q, ZA6.Q, ZA8.Q, ZA10.Q, ZA12.Q, ZA14.Q
ZA1.H	ZA1.Q, ZA3.Q, ZA5.Q, ZA7.Q, ZA9.Q, ZA11.Q, ZA13.Q, ZA15.Q

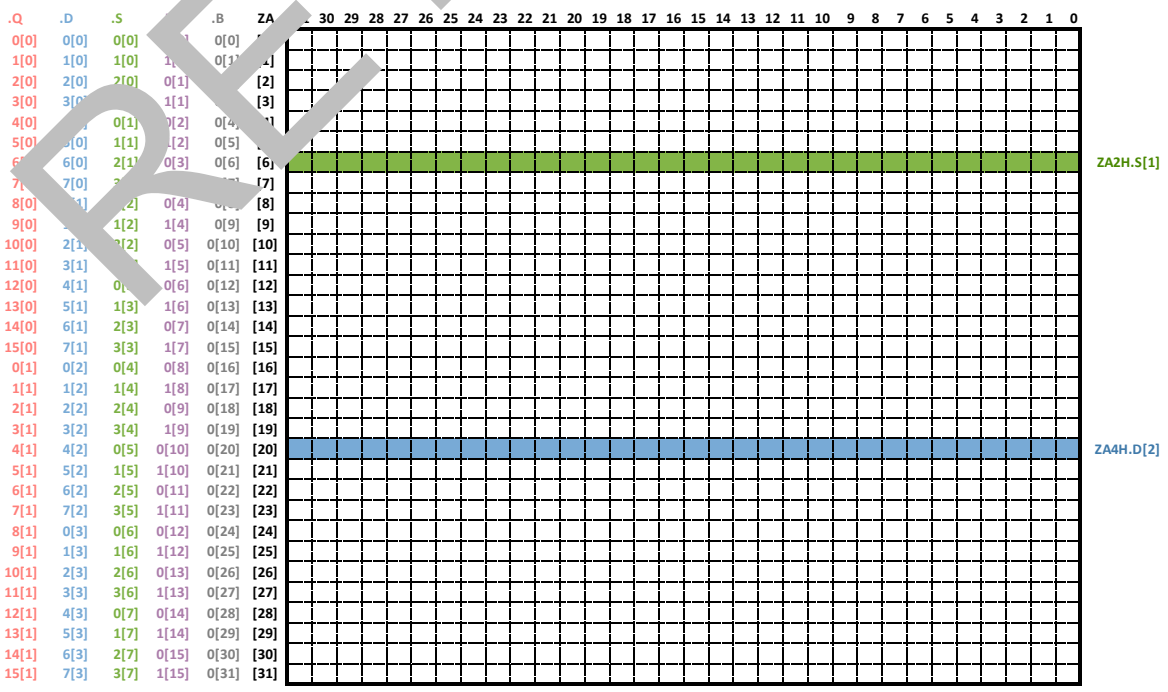
Tile	Overlaps
ZA0.S	ZA0.Q, ZA4.Q, ZA8.Q, ZA12.Q
ZA1.S	ZA1.Q, ZA5.Q, ZA9.Q, ZA13.Q
ZA2.S	ZA2.Q, ZA6.Q, ZA10.Q, ZA14.Q
ZA3.S	ZA3.Q, ZA7.Q, ZA11.Q, ZA15.Q
ZA0.D	ZA0.Q, ZA8.Q
ZA1.D	ZA1.Q, ZA9.Q
ZA2.D	ZA2.Q, ZA10.Q
ZA3.D	ZA3.Q, ZA11.Q
ZA4.D	ZA4.Q, ZA12.Q
ZA5.D	ZA5.Q, ZA13.Q
ZA6.D	ZA6.Q, ZA14.Q
ZA7.D	ZA7.Q, ZA15.Q

I_{WGZBT} The architecture permits concurrent use of different element size tiles.

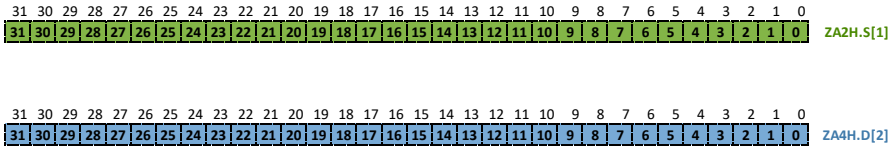
B2.3.3 Horizontal tile slice mappings

I_{NJJXW} The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for a 32-bit element and 64-bit element horizontal tile slice.

Each small numbered square represents 32 bits.



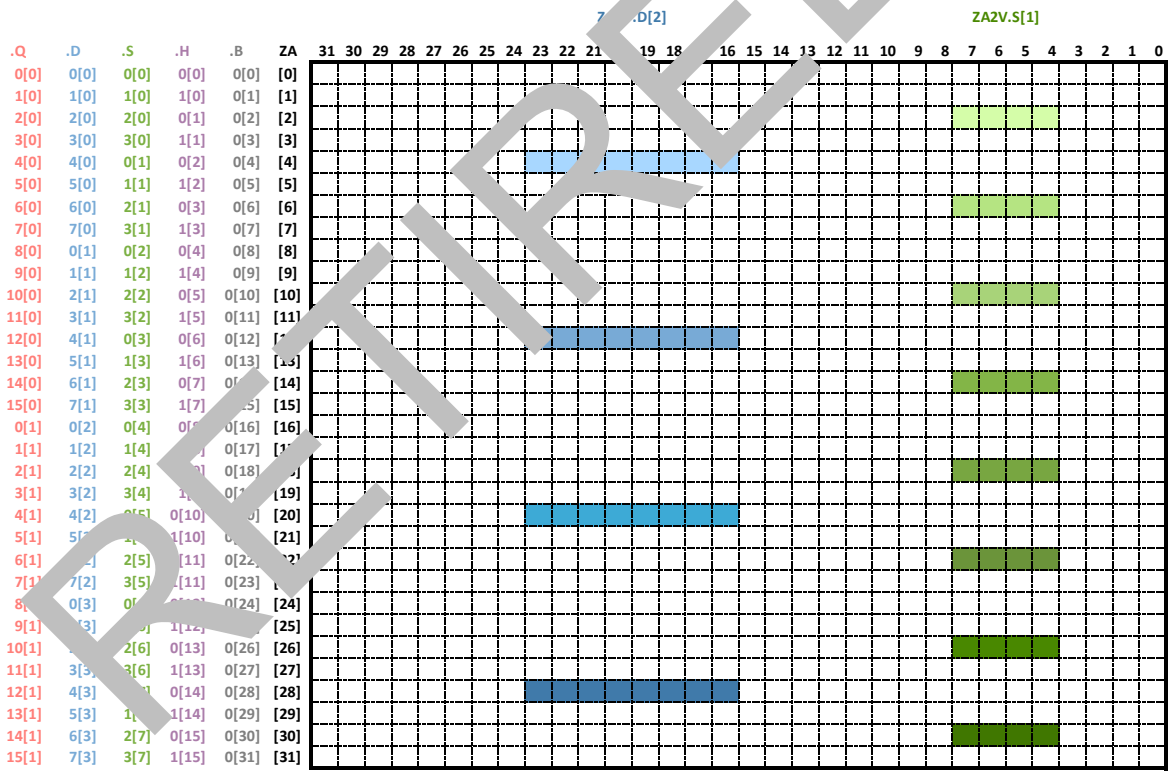
An SME vector load, store, or move instruction that accesses horizontal tile slices ZA2H.S[1] or ZA4H.D[2] treats the slices as vectors with the following layout:



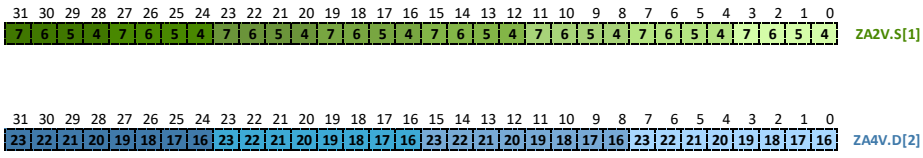
B2.3.4 Vertical tile slice mappings

I_TNCCV The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for 32-bit element and 64-bit element vertical tile slice.

Each small numbered square represents 8 bits.



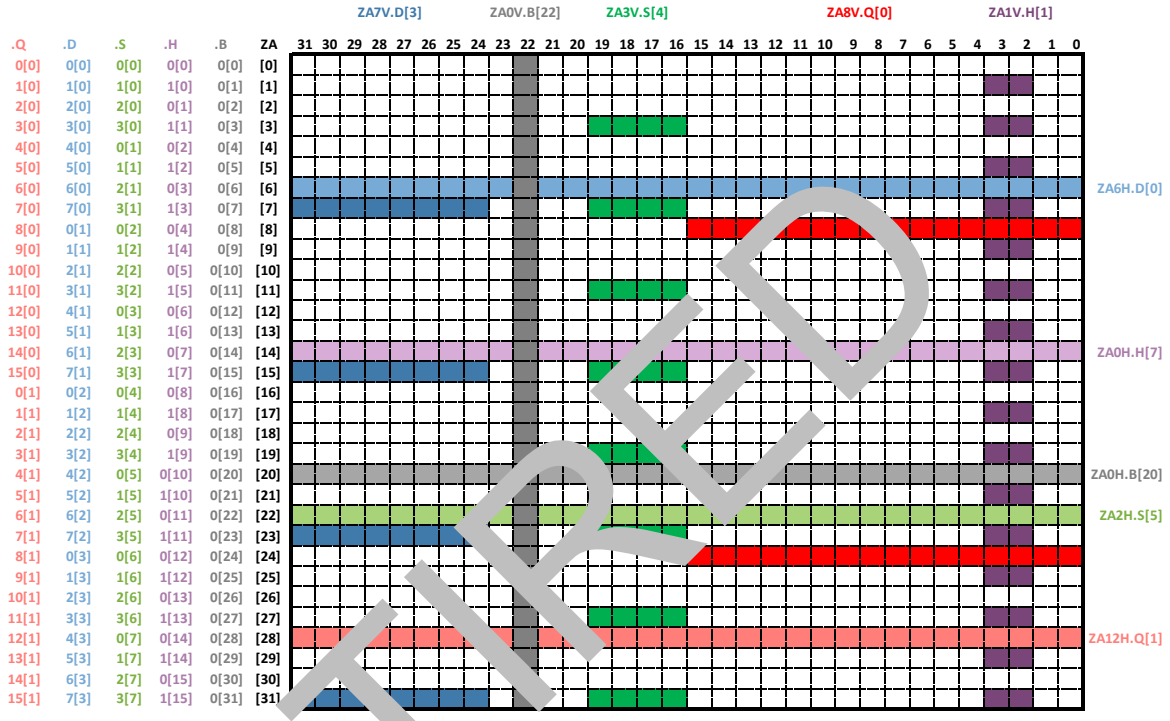
An SME vector load, store, or move instruction which accesses vertical tile slices ZA2V.S[1] or ZA4V.D[2] treats the slices as vectors with the following layout:



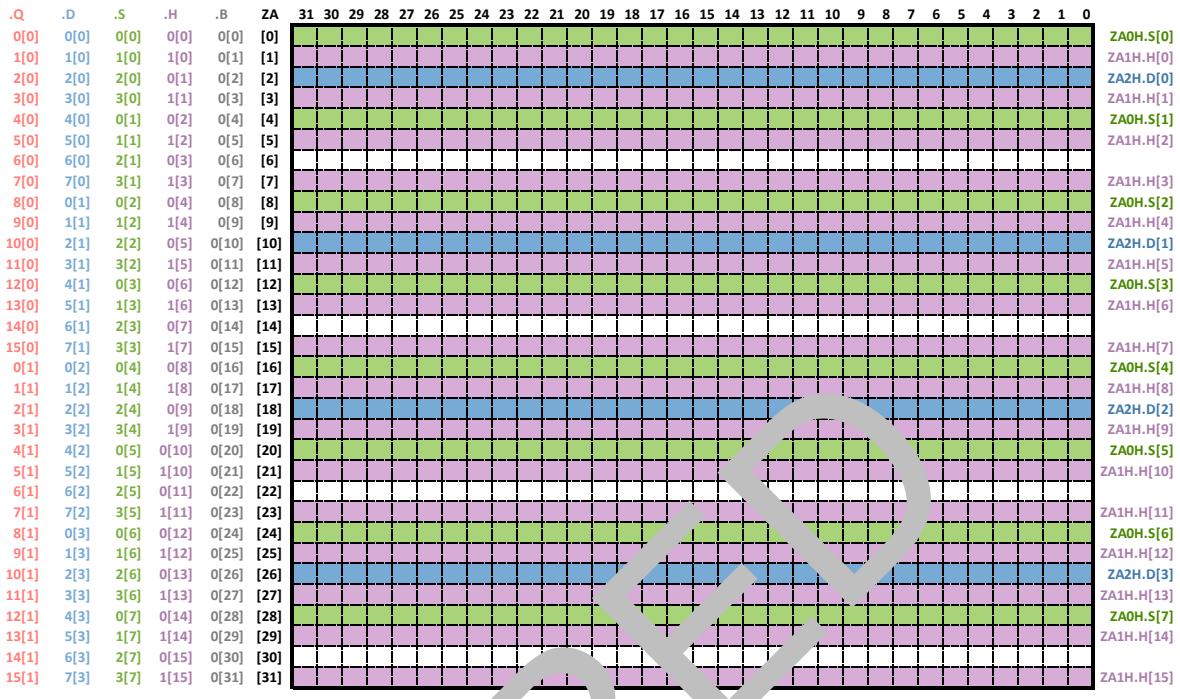
B2.3.5 Mixed horizontal and vertical tile slice mappings

I_CGXPJ The following diagram illustrates the *ZA* storage mapping for SVL of 256 bits, for various element size tiles, horizontal tile slices, and vertical tile slices.

Each small square represents 8 bits.



I_HVFMB It is possible to simultaneously use non-overlapping *ZA* array vectors within tiles of differing element sizes. For example, tiles ZA1.H, ZA2.S, and ZA2.D have no *ZA* array vectors in common, as illustrated in the following diagram for SVL of 256 bits:



I_WDMCK It is possible to access overlapping ZA array vectors within tiles of differing element sizes. For example, tiles ZA0.H, ZA2.S, and ZA6.D have common ZA array vectors.

B2.4 SME2 Multi-vector operands

R_{KLRKJ} Multi-vector operands allow certain SME2 instructions to access source and destination operands which each consist of one of the following:

- A group of two or four SVE *Z* vector registers.
- A group of two or four *ZA* tile slices.
- A group of two, four, eight, or sixteen *ZA* array vectors.

B2.4.1 Z multi-vector operands

D_{PSTFY} A multi-vector operand consisting of two or four SVE *Z* vector registers is called a *Z* multi-vector operand.

R_{VCXBQ} A *Z* multi-vector operand can occupy:

- Consecutively numbered *Z* registers.
- *Z* registers with strided numbering.

D_{NYNRZ} A *Z* multi-vector operand occupying two consecutively numbered *Z* vectors consists of Z*n+0* and Z*n+1*, where *n+x* modulo 32 is a register number in the range 0-31 inclusive.

D_{DZDBM} A *Z* multi-vector operand occupying four consecutively numbered *Z* vectors consists of Z*n+0* to Z*n+3*, where *n+x* modulo 32 is a register number in the range 0-31 inclusive.

D_{VYKCM} The preferred disassembly for a *Z* multi-vector operand of consecutively numbered *Z* vectors is a dash-separated register range, for example { Z0.S-Z1.S } or { Z30.B-Z1.B }. Toolchains must also support assembler source code that uses the alternative comma-separated list notation, for example { Z0.S, Z1.S } or { Z30.B, Z31.B, Z0.B, Z1.B }. Disassemblers can provide an option to select between the dash-separated range and comma-separated list notations.

D_{PCYZS} A *Z* multi-vector operand occupying two *Z* vectors with strided register numbering consists of a first register in the range Z0-Z7 or Z16-Z23, followed by a second register with a number that is 8 higher than the first register.

D_{RZTTV} A *Z* multi-vector operand occupying four *Z* vectors with strided register numbering consists of a first register in the range Z0-Z3 or Z16-Z19, followed by three registers each with a number that is 4 higher than the previous register.

D_{DMTSL} The preferred disassembly for a *Z* multi-vector operand of *Z* vectors with strided register numbering is a comma-separated register list, for example { Z0.D, Z8.D } or { Z0.H, Z4.H, Z8.H, Z12.H }.

B2.4.2 ZA multi-slice operands

D_{JMCTK} A multi-vector operand consisting of two or four *ZA* tile slices is called a *ZA* multi-slice operand.

R_{SCHNH} A *ZA* multi-slice operand can occupy:

- Consecutively numbered horizontal *ZA* tile slices.
- Consecutively numbered vertical *ZA* tile slices.

D_{JFDSB} In instructions operating on *ZA* multi-slice operands, the lowest-numbered slice is:

- A multiple of 2 for a two-slice *ZA* operand.
- A multiple of 4 for a four-slice *ZA* operand.

The lowest-numbered slice is selected by the sum of a 32-bit general-purpose slice index register W*s* and an immediate slice index offset *offs*.

R_{XMMKZ} Instructions operating on the following *ZA* multi-slice operands are treated as UNDEFINED:

- The four-slice operand in a 64-bit element tile when SVL is 128 bits.
- The two-slice operand in a 128-bit element tile when SVL is 128 bits.

- The four-slice operand in a 128-bit element tile when SVL is 128 bits or 256 bits.

D_{GJTMX} The preferred disassembly for a *ZA* multi-slice operand is as follows:

- ZA*t*H.*T*[W*s*, *offs1*:*offs2*], for horizontal *ZA* two-slice operands, where *offs2* = *offs1* + 1.
- ZA*t*H.*T*[W*s*, *offs1*:*offs4*], for horizontal *ZA* four-slice operands, where *offs4* = *offs1* + 3.
- ZA*t*V.*T*[W*s*, *offs1*:*offs2*], for vertical *ZA* two-slice operands, where *offs2* = *offs1* + 1.
- ZA*t*V.*T*[W*s*, *offs1*:*offs4*], for vertical *ZA* four-slice operands, where *offs4* = *offs1* + 3.

B2.4.3 ZA multi-vector operands

D_{RGXBK} A multi-vector operand consisting of two, four, eight, or sixteen *ZA* array vectors is called a *ZA* multi-vector operand.

D_{TGDRF} One *ZA* array vector is called a *ZA* single-vector group.

D_{FCYGL} Two consecutively numbered vectors in the *ZA* array are called a *ZA* double-vector group.

D_{GCTYB} Four consecutively numbered vectors in the *ZA* array are called a *ZA* quad-vector group.

I_{PMQRQ} The *ZA* multi-vector operand consists of one, two, or four vector groups, where a vector group is one of the following:

- *ZA* single-vector group.
- *ZA* double-vector group.
- *ZA* quad-vector group.

I_{KLBYZ} The SME2 architecture includes multi-vector instructions that access a *ZA* multi-vector operand consisting of the same number of vector groups as there are vectors in each *Z* multi-vector operand.

I_{HPKZM} The preferred disassembly for a *ZA* multi-vector operand consisting of two or four vector groups, defined in declarations D_{KQZYZ}, D_{JWRSN} and D_{TTNGH}, includes the symbol `VGx2` or `VGx4`, respectively. The symbol `VGx2` or `VGx4` can optionally be omitted in assembler source code if it can be inferred from the other operands.

D_{CLJBX} In instructions that access a *ZA* multi-vector operand, the lowest-numbered vector is selected by the sum of a 32-bit general-purpose vector select register W*v* and an immediate vector select offset *offs*, modulo one of the following values:

- SVL_B when the operand consists of one *ZA* vector group.
- $SVL_B/2$ when the operand consists of two *ZA* vector groups.
- $SVL_B/4$ when the operand consists of four *ZA* vector groups.

B2.4.4 ZA multi-vector operands of single-vector groups

D_{QFPHH} In instructions where the *ZA* multi-vector operand consists of two single-vector groups, each vector group is held in a separate half of the *ZA* array. The halves of the *ZA* array are as follows, where *n* is in the range 0 to ($SVL_B/2$ - 1) inclusive:

- ZA[*n*+*0*].
- ZA[$SVL_B/2$ + *n*+*0*].

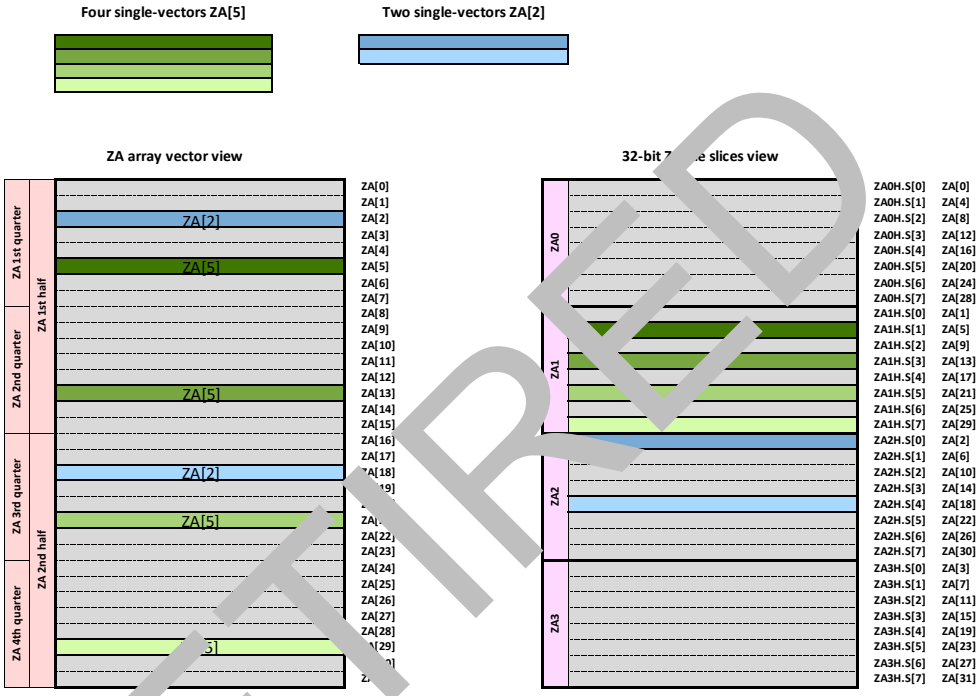
D_{TTHGQ} In instructions where the *ZA* multi-vector operand consists of four single-vector groups, each vector group is held in a separate quarter of the *ZA* array. The quarters of the *ZA* array are as follows, where *n* is in the range 0 to ($SVL_B/4$ - 1) inclusive:

- ZA[*n*+*0*].
- ZA[$SVL_B/4$ + *n*+*0*].
- ZA[$SVL_B/2$ + *n*+*0*].
- ZA[SVL_B*3/4 + *n*+*0*].

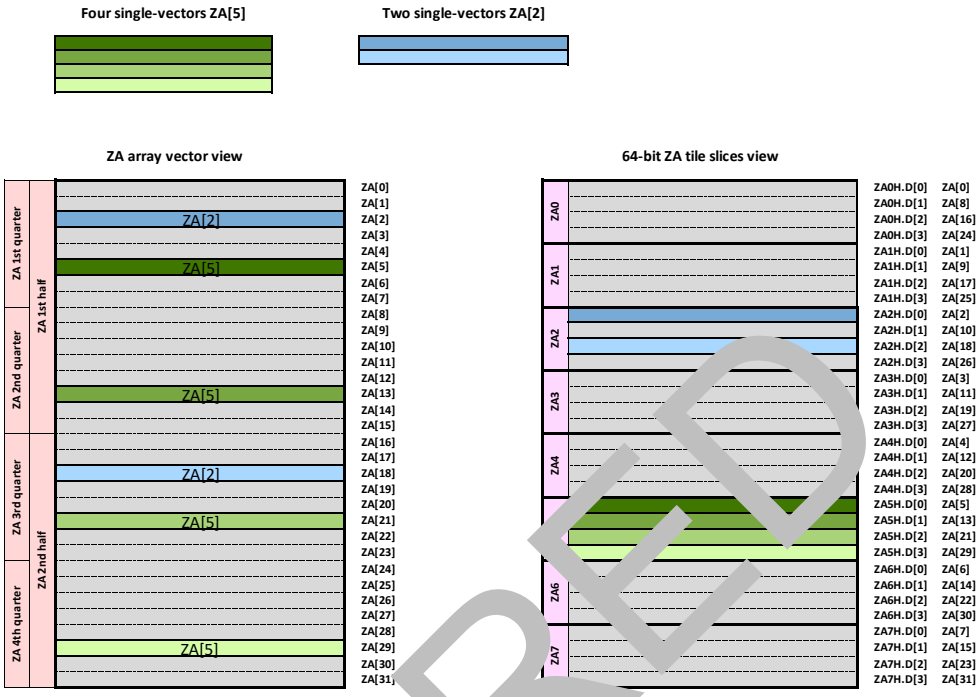
DKQZYZ The preferred disassembly for a *ZA* multi-vector operand of single-vector groups is as follows, where *offs* is an immediate in the range 0-7 inclusive, and *T* is one of B, H, S, or D:

- ZA.*T*[W*v*, *offs*, VGx2], when the operand consists of two single-vector groups.
- ZA.*T*[W*v*, *offs*, VGx4], when the operand consists of four single-vector groups.

IBYBQLI The mapping between *ZA* multi-vector operands of single-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:



IMLNNG The mapping between *ZA* multi-vector operands of single-vector groups and 64-bit element *ZA* tile slices when SVL = 256 bits is illustrated in the following diagram:



B2.4.3.2 ZA multi-vector operands of double-vector groups

D_{RBSQJ} In instructions where the *ZA* multi-vector operand consists of one double-vector group, the vector group is held in *ZA* array vectors ZA[*n+0*] to ZA[*n+1*], where *n* is a multiple of 2 in the range 0 to (SVL_B - 2) inclusive.

D_{KKVVG} In instructions where the *ZA* multi-vector operand consists of two double-vector groups, each vector group is held in a separate half of the *ZA* array. The halves of the *ZA* array are as follows, where *n* is a multiple of 2 in the range 0 to (SVL_B/2 - 2) inclusive:

- ZA[*n+0*] to ZA[*n+1*].
- ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+1*].

D_{VMYGN} In instructions where the *ZA* multi-vector operand consists of four double-vector groups, each vector group is held in a separate quarter of the *ZA* array. The quarters of the *ZA* array are as follows, where *n* is a multiple of 2 in the range 0 to (SVL_B/4 - 2) inclusive:

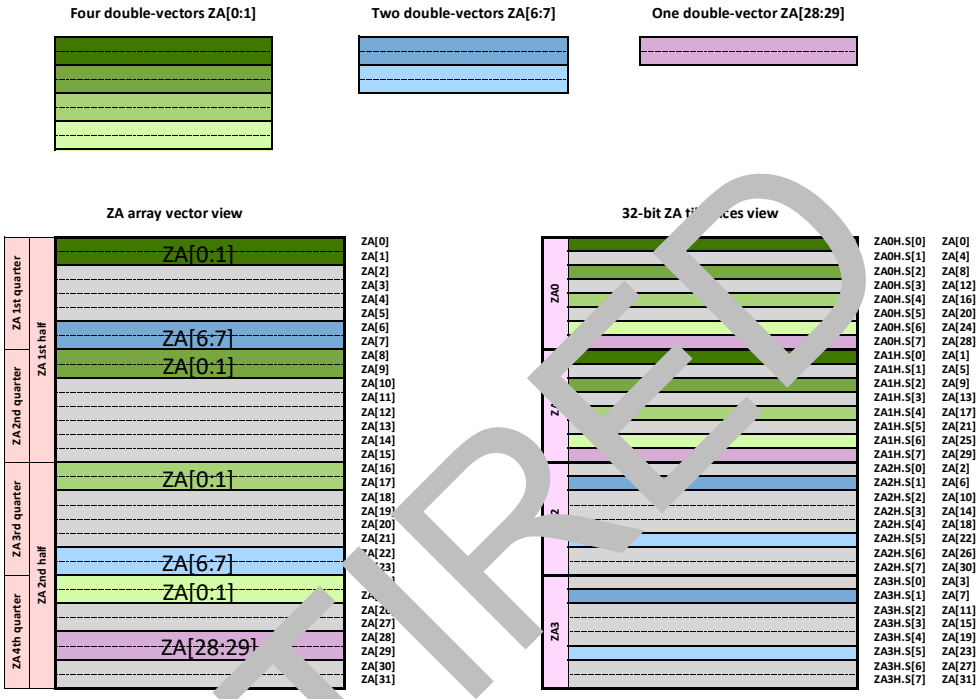
- ZA[*n+0*] to ZA[*n+1*].
- ZA[SVL_B/4 + *n+0*] to ZA[SVL_B/4 + *n+1*].
- ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+1*].
- ZA[SVL_B*3/4 + *n+0*] to ZA[SVL_B*3/4 + *n+1*].

D_{JWRSN} The preferred disassembly for a *ZA* multi-vector operand of double-vector groups is as follows, where *offs2* = *offs1* + *1*, and *T* is one of B, H, S, or D:

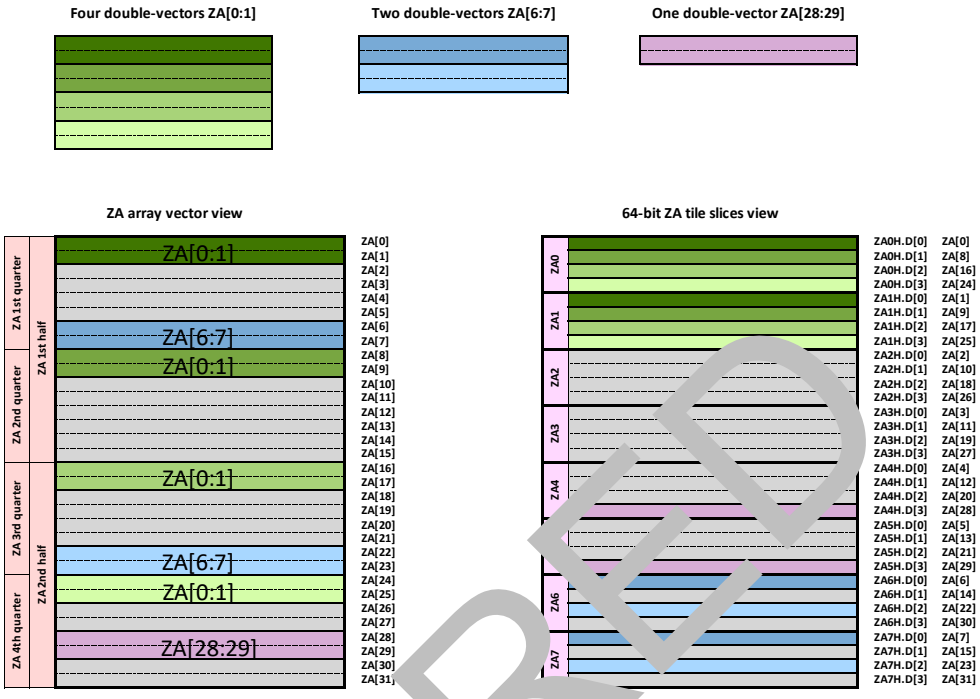
- ZA.*T*[W*v*, *offs1*:*offs2*], where *offs1* is a multiple of 2 in the range 0-14 inclusive, when the operand consists of one double-vector group.
- ZA.*T*[W*v*, *offs1*:*offs2*, VGx2], where *offs1* is a multiple of 2 in the range 0-6 inclusive, when the operand consists of two double-vector groups.

- ZA.*T*[W*v*, *offs1*:*offs2*, VGx4], where *offs1* is a multiple of 2 in the range 0-6 inclusive, when the operand consists of four double-vector groups.

I_LZRTK The mapping between *ZA* multi-vector operands of double-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:



I_JYQTB The mapping between *ZA* multi-vector operands of double-vector groups and 64-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:



B2.4.3.3 ZA multi-vector operands of quad-vector groups

D_{WSTWB} In instructions where the *ZA* multi-vector operand consists of one quad-vector group, the vector group is held in *ZA* array vectors ZA[*n+0*] to ZA[*n+3*], where *n* is a multiple of 4 in the range 0 to (SVL_B - 4) inclusive.

D_{QJXHS} In instructions where the *ZA* multi-vector operand consists of two quad-vector groups, each vector group is held in a separate half of the *ZA* array. The halves of the *ZA* array are as follows, where *n* is a multiple of 4 in the range 0 to (SVL_B/2 - 4) inclusive:

- ZA[*n+0*] to ZA[*n+3*].
- ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+3*].

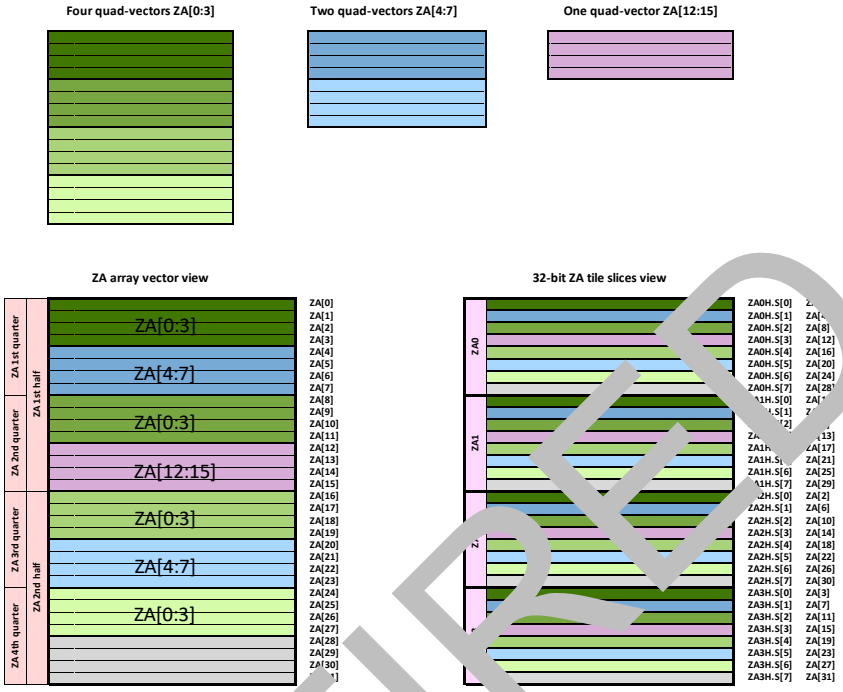
D_{BQWJD} In instructions where the *ZA* multi-vector operand consists of four quad-vector groups, each vector group is held in a separate quarter of the *ZA* array. The quarters of the *ZA* array are as follows, where *n* is a multiple of 4 in the range 0 to (SVL_B/4 - 4) inclusive:

- ZA[*n+0*] to ZA[*n+3*].
- ZA[SVL_B/4 + *n+0*] to ZA[SVL_B/4 + *n+3*].
- ZA[SVL_B/2 + *n+0*] to ZA[SVL_B/2 + *n+3*].
- ZA[SVL_B*3/4 + *n+0*] to ZA[SVL_B*3/4 + *n+3*].

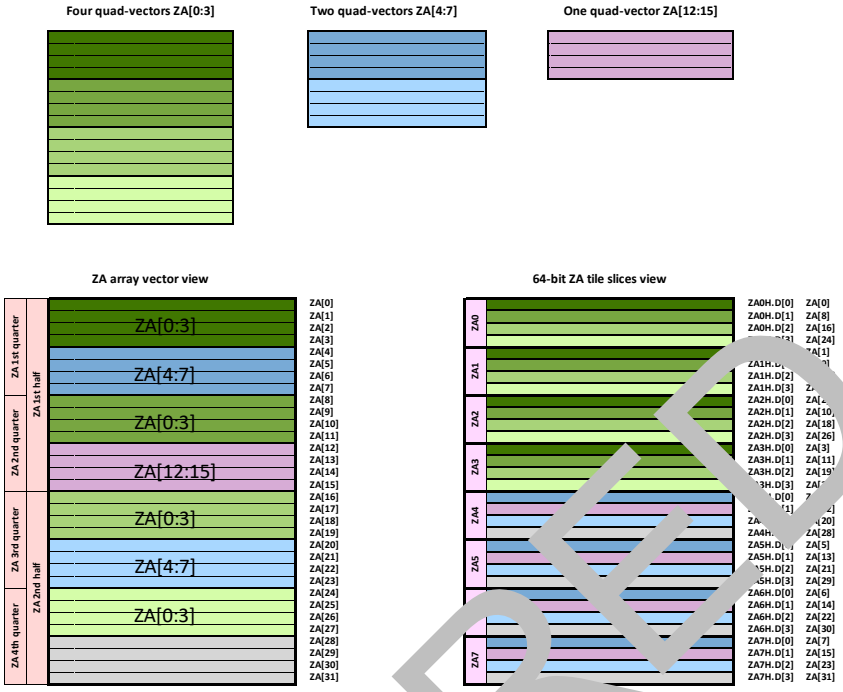
D_{TTNGH} The preferred disassembly for a *ZA* multi-vector operand of quad-vector groups is as follows, where *offs4* = *offs1* + 3, and *T* is one of B, H, S, or D:

- ZA.*T*[W*v*, *offs1*:*offs4*], where: *offs1* is a multiple of 4 in the range 0-12 inclusive, when the operand consists of one quad-vector group.
- ZA.*T*[W*v*, *offs1*:*offs4*, VGx2], where *offs1* is 0 or 4, when the operand consists of two quad-vector groups.
- ZA.*T*[W*v*, *offs1*:*offs4*, VGx4], where *offs1* is 0 or 4, when the operand consists of four quad-vector groups.

I_{ZNSGW} The mapping between *ZA* multi-vector operands of quad-vector groups and 32-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:



I_{KBMLX} The mapping between *ZA* multi-vector operands of quad-vector groups and 64-bit element *ZA* tile slices when SVL is 256 bits is illustrated in the following diagram:



B2.5 SME2 Multi-vector predication

D_{JFJZX} SME2 introduces the multi-vector predication concept in *Streaming SVE mode*, named *predicate-as-counter*.

I_{LLLXP} The existing SVE predication concept is referred to as *predicate-as-mask*.

I_{DSFKR} SME2 multi-vector instructions interpret the lowest-numbered 16 bits of SVE predicate registers *P0-P15* as a *predicate-as-counter* encoding.

D_{QBDRH} A *predicate-as-counter* encoding includes:

- An invert bit, that encodes whether the element count field is referring to the number of TRUE or FALSE predicate elements.
- A variable-width element count field. This field holds an unsigned integer value of up to 14 bits, that encodes:
  - When the invert bit is 0, the number of consecutive elements starting from element 0 that are TRUE, with the remaining elements being FALSE.
  - When the invert bit is 1, the number of consecutive elements starting from element 0 that are FALSE, with the remaining elements being TRUE.
- A variable-width element size field of up to 4 bits, where the number of trailing zeroes encodes `LSZ`, $\log_2$ of the element size in bytes, or an all-FALSE predicate if all 4 bits are zero.

[15] Invert	[14:(LSZ+1)] Element count	[LSZ:0] Element size	Meaning
x	xxxxxxxxxxxxxx	1	Byte elements, count in [14:1]
x	xxxxxxxxxxxxxx	10	Halfword elements, count in [14:2]
x	xxxxxxxxxxxxx	100	Word elements, count in [14:3]
x	xxxxxxxxxxxxx	1000	Doubleword elements, count in [14:4]
x	xxxxxxxxxxxxx	0000	All-FALSE predicate (any element size)

D_{JGSYR} The canonical all-TRUE *predicate-as-counter* encoding has an element count of zero, with the `invert` bit set to 1 and a nonzero element size field determined by the generating instruction.

D_{BJMYH} The canonical all-FALSE *predicate-as-counter* encoding has an element count of zero, with the `invert` bit set to 0 and an element size field set to `0b0000`.

I_{ZPLKP} A *predicate-as-counter* encoding can represent a consecutive element count in the range of 0 to the maximum number of byte elements in four vectors, minus 1. The architectural maximum vector length of 2048 bits or 256 bytes therefore requires an element count of $\log_2(1024) = 10$ bits, plus one element size bit, plus the invert bit. The additional 4 bits in the element count field are reserved.

D_{HWRFM} In assembler syntax:

- The name *Pg* is used for *predicate-as-mask*.
- The name *PNg* is used for *predicate-as-counter*.

Both *Pg* and *PNg* refer to the same predicate register.

R_{YCRMW} If VL is greater than 128 bits, then an instruction which writes a *predicate-as-counter* encoding to a predicate register sets bits 16 and higher of that register to 0.

R_{SGVTC} If VL is greater than 128 bits, then an instruction which reads a predicate register using the *predicate-as-counter* encoding ignores bits 16 and higher of that register.

R_{YSTVC} An instruction uses only the least significant bits in the element count field of the *predicate-as-counter* register that are required to represent the number of bytes in the current vector length times four, minus 1. The instruction ignores the more significant bits in the element count field.

I_{SKKTX} For example, when VL is 512 bits there are 256 byte elements in four vectors, so the *predicate-as-counter* encoding requires at most an 8-bit element count field [8:1], a 1-bit size field [0], and the invert bit [15]. Therefore, when VL is 512 bits, an instruction uses bits [15] and [8:0] from the *predicate-as-counter* register and ignores bits [14:9] and [63:16].

I_{DKPNQ} The SME2 *WHILE* instructions generate a *predicate-as-counter* encoding. These instructions have an operand that indicates the number of vectors (2 or 4) to be controlled by this predicate, which determines:

- The maximum value that can be stored in the count.
- The number of elements that are considered Active when computing the *Any Active element* and *Last Active element* SVE condition flags.

A canonical all-TRUE encoding is generated when the number of TRUE elements is equal to or exceeds the limit of the number of elements in a vector times the number of vectors.

The canonical all-FALSE encoding is generated when the number of TRUE elements is zero.

I_{JBYVM} The SME2 *PTRUE* instruction generates a canonical all-TRUE *predicate-as-counter* encoding.

I_{BXWKF} The SVE *PFALSE* instruction generates the canonical all-FALSE *predicate-as-counter* encoding.

I_{TRVQQ} The SME2 *PEXT* instruction converts a *predicate-as-counter* encoding into a *predicate-as-mask* encoding. Since a *predicate-as-counter* encoding allows more predicate elements than can be represented in a *predicate-as-mask* encoding, this instruction takes an operand to extract different portions of a wider mask corresponding to a *predicate-as-counter* encoding.

I_{QSHRN} The SME2 *CNTP* instruction converts a *predicate-as-counter* encoding into a total Active element count value that is placed in a general-purpose register.

CNTP has an operand that indicates the limit of the number of elements to be counted. The limit corresponds to the total number of elements in either 2 or 4 vectors.

I_{ZMDPF} Predicated SME2 multi-vector instructions interpret the value in their Governing predicate register using the *predicate-as-counter* encoding to determine the number and size of consecutive Active elements. When the element size of the instruction operation is different from the element size in the *predicate-as-counter* encoding, the number of Active elements of the instruction operation is also different from the number of *predicate-as-counter* Active elements.

See also:

- CNTP.
- PEXT (predicate).
- PEXT (predicate pair).
- PFALSE.
- PTRUE.
- WHILEGE.
- WHILEGT.
- WHILEHI.
- WHILEHS.
- WHILELE.
- WHILELO.
- WHILELS.
- WHILELT.

B2.6 SME2 Lookup table

D_{KHZMV} When SME2 is implemented, a PE has a 512-bit architectural register *ZT0* to support the lookup table feature.

D_{WDJBC} The *ZT0* register holds 8-bit, 16-bit, or 32-bit lookup table elements that are stored in the least significant bits of 32-bit table entries. The lowest numbered 32 bits in the register hold table entry 0.

R_{JQXLS} The lookup table in the *ZT0* register can be accessed using fully packed 2-bit or 4-bit indices from a numbered portion of one source *Z* vector register.

I_{BRRGG} When the lookup table *ZT0* is addressed by 2-bit indices, four different table elements (0-3) of a given element size can be accessed. When the lookup table *ZT0* is addressed by 4-bit indices, 16 different table elements (0-15) of a given element size can be accessed.

R_{JKYRB} The indexed 8-bit, 16-bit, or 32-bit table elements are read from the *ZT* register and packed into consecutive elements of an SVE *Z* vector or *Z* multi-vector operand.

I_{HQTSR} The validity and accessibility of the *ZT0* register are enabled by PSTATE.ZA. For more information, see B1.2 *Process state* and B1.2.2 *PSTATE.ZA*.

See also:

- C1.1.2 *Traps and exceptions*.
- C1.4.8 *SVCR*.

Chapter B3
Floating-point behaviors

B3.1 Overview

SME modifies so[illegible]e of the A[illegible]profile floating-point behaviors when a PE is in *Streaming SVE mode*, and introduces an FPCR control w[illegible]h e[illegible]nds BFloat16 dot product calculations to support a wider range of numeric behaviors.

See also:

- [illegible]PCR.
- [illegible]B1.2 *P*[illegible]

B3.2 Supported floating-point data types

R_{VSCHZ} The following BFloat16 instructions operate on the BFloat16 and the IEEE 754-2008 Single-precision floating-point data types, as defined respectively in sections *BFloat16 floating-point format* and *Single-precision floating-point format* of *Arm® Architecture Reference Manual for A-profile architecture* [1]:

- The SME `BFMOPA` and `BFMOPS` floating-point instructions defined in D1.1 *SME and SME2 data-processing instructions*.
- The SME2 multi-vector `BFCVT`, `BFCVTN`, `BFDOT`, `BFMLAL`, `BFMLSL`, and `BFVDOT` floating-point instructions defined in D1.1 *SME and SME2 data-processing instructions*.
- The SVE2 `BFMLSLB` and `BFMLSLT` floating-point instructions that are introduced by SME2 and defined in D1.2 *SVE2 data-processing instructions*.

R_{BYPSW} The floating-point instructions defined in Chapter D1 *SME instructions* operate on the IEEE 754-2008 floating-point data types as defined in the following *Arm® Architecture Reference Manual for A-profile architecture* [1] sections:

- *Half-precision floating-point formats* (but not the Arm alternative half-precision format).
- *Single-precision floating-point format*.
- *Double-precision floating-point format*.

B3.3 BFloat16 behaviors

If FEAT_EBF16 is implemented, the Extended BFloat16 behaviors can be enabled for the BFloat16 instructions. This section describes how the BFloat16 instruction behaviors are changed by FEAT_EBF16.

R_{BSHYK} When `ID_AA64ZFR0_EL1.BF16` and `ID_AA64ISAR1_EL1.BF16` have the value `0b0010`, the PE implements FEAT_EBF16 and supports the `FPCR.EBF` control.

R_{RKGSJ} If FEAT_EBF16 is implemented, then:

- FEAT_EBF16 is enabled when `FPCR.EBF` is 1.
- FEAT_EBF16 is not enabled when `FPCR.EBF` is 0.

D_{MVPSG} Unless stated otherwise, the rules in this section describe the behaviors of the following instructions:

- The SME `BFMOPA` and `BFMOPS` instructions.
- The SME2 `BFDOT` and `BFVDOT` instructions.
- The Advanced SIMD and SVE `BFDOT` and `BFMMLA` instructions.

B3.3.1 Common BFloat16 behaviors

I_{CYTKF} The Common BFloat16 behaviors are the behaviors currently defined in *Arm® Architecture Reference Manual for A-profile architecture* [1] which are not changed by the additional `FPCR.EBF` control.

R_{RDCPW} The instructions specified in D_{MVPSG} that detect exceptional floating-point conditions produce the expected single-precision default result but do not modify the cumulative floating-point exception flag bits, `FPSR.{IDC,IXC,UFC,OFC,DZC,IOC}`.

R_{PFFFF} The instructions specified in D_{MVPSG} generate default NaN values, behaving as if `FPCR.DN` has an Effective value of 1.

See also:

- `FPSR`, Floating-point Status Register in *Arm® Architecture Reference Manual for A-profile architecture* [1].
- FPCR.

B3.3.2 Standard BFloat16 behaviors

I_{CMSBQ} The Standard BFloat16 behaviors are the behaviors currently defined in *Arm® Architecture Reference Manual for A-profile architecture* [1] which can be changed by the `FPCR.EBF` control provided by FEAT_EBF16.

R_{JFWDV} If FEAT_EBF16 is either not implemented or not enabled, then the instructions specified in D_{MVPSG} ignore the `FPCR.RMode` control and use the rounding mode defined for BFloat16 in section *Round to Odd mode* of *Arm® Architecture Reference Manual for A-profile architecture* [1].

R_{WBLQD} If FEAT_EBF16 is either not implemented or not enabled, then the instructions specified in D_{MVPSG} flush denormalized inputs and outputs to zero, behaving as if `FPCR.FZ` has an Effective value of 1.

R_{JPNSN} If FEAT_EBF16 is either not implemented or not enabled, then the instructions specified in D_{MVPSG} perform unfused multiplies and additions with intermediate rounding of all products and sums.

B3.3.3 Extended BFloat16 behaviors

I_{MFJMJ} The Extended BFloat16 behaviors are the behaviors that can be enabled by the `FPCR.EBF` control provided by FEAT_EBF16.

R_{RQKQZ} If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} support all four IEEE 754 rounding modes selected by the `FPCR.RMode` control.

R_{JZVPD} If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} honor the `FPCR.FZ` control.

R_{LJGTX} If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} perform a fused two-way sum-of-products for each pair of adjacent BFloat16 elements in the source vectors, without intermediate rounding of the products, but rounding the single-precision sum before addition to the single-precision accumulator element.

R_{CQFQT} If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} generate the default NaN as intermediate sum-of-products when any of the following are true:

- Any multiplier input is a NaN.
- Any product is infinity × 0.0.
- There are infinite products with differing signs.

R_{WQDBR} If FEAT_EBF16 is implemented and enabled, then the instructions specified in D_{MVPSG} generate an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.

R_{YPGLB} When FEAT_AFP is implemented and FEAT_EBF16 is implemented and enabled, the instructions specified in D_{MVPSG} honor the `FPCR.AH` and `FPCR.FIZ` controls.

I_{WKNML} When FEAT_AFP is implemented and FEAT_EBF16 is implemented and enabled, the following alternate floating-point behaviors affect the instructions specified in D_{MVPSG}:

- When `FPCR.AH` is 1, the sign bit of a generated default NaN result is set to 1 instead of 0.
- When `FPCR.AH` is 1 and `FPCR.FZ` is 1, a denormal result detected after rounding with an unbounded exponent has been applied, is flushed to zero.
- When `FPCR.AH` is 1, the `FPCR.FZ` control does not cause denormalized inputs to be flushed to zero.
- When `FPCR.FIZ` is 1, all denormalized inputs are flushed to zero.

B3.4 Floating-point behaviors in Streaming SVE mode

D_{DMPBW} Unless stated otherwise, the rules in this section describe the behaviors of the following instructions:

- The floating-point instructions that are legal in Streaming SVE mode, and operate on half-precision, single-precision, and double-precision input data types, placing their results in SIMD&FP registers or SVE *Z* vector registers.
- The SVE `BFMLALB` and `BFMLALT` instructions.
- The floating-point instructions introduced by SME2 that place their results in one or more SVE *Z* vector registers:
  - The `BFCVT`, `BFCVTN`, `FCLAMP`, `FCVT`, `FCVTN`, `FCVTZS`, `FCVTZU`, `FMAX`, `FMAXNM`, `FMIN`, `FMINNM`, `FRINTA`, `FRINTM`, `FRINTN`, `FRINTP`, `SCVTF`, and `UCVTF` instructions, as defined in D1.1 *SME and SME2 data-processing instructions*.
  - The `BFMLSLB`, `BFMLSLT`, `FDOT`, and `FCLAMP` instructions, as defined in D1.2 *SVE2 data-processing instructions*.

R_{PHDZL} When the PE is in *Streaming SVE mode*, the instructions specified in D_{DMPBW} honor the Non-streaming scalar and SVE floating-point behaviors, as governed by the `FPCR.{DN,FZ,RMode,FZ16,AH,FIZ}` controls.

R_{GTYSK} When the PE is in *Streaming SVE mode*, the instructions specified in D_{DMPBW} that detect exceptional floating-point conditions produce the expected default result and can update the appropriate cumulative floating-point exception flag bits in `FPSR.{IDC,IXC,UFC,OFC,DZC,IOC}`.

R_{PYSCC} The floating-point behaviors followed by the `FCLAMP` instruction are identical to the behaviors followed when executing `FMAXNM` and `FMINNM` in order.

R_{FBFNT} When the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, the *Effective value* of the `FPCR` is 0 if all of the `IDE`, `IXE`, `UFE`, `OFE`, `DZE`, and `IOE` floating-point exception trap enable controls, and the `NEP` element predicate control, are 0 for all purposes other than a direct read or write of the register.

See also:

- `FPSR`, Floating-point Status Register, *Arm® Architecture Reference Manual for A-profile architecture* [1].
- FPCR.

B3.5 Floating-point behaviors targeting the ZA array

D_{HTZVK} Unless stated otherwise, the rules in this section describe the behaviors of the SME and SME2 floating-point instructions that place their results in the *ZA* array, except `BFMOPA`, `BFMOPS`, `BFDOT`, and `BFVDOT`.

For the behaviors of the BFloat16 instructions, see B3.3 *BFloat16 behaviors*.

R_{TGSKG} The instructions specified in D_{HTZVK} that detect exceptional floating-point conditions produce the expected IEEE 754 default result but do not modify any of the cumulative floating-point exception flag bits, `FPSR.{IDC,IXC,UFC,OFC,DZC,IOC}`.

R_{RKHHZ} The instructions specified in D_{HTZVK} generate default NaN values, behaving as if `FPCR.DN` has an Effective value of 1.

R_{TCLRM} The instructions specified in D_{HTZVK} support all four IEEE 754 rounding modes selected by the `FPCR.RMode` control.

R_{VVVNR} The instructions specified in D_{HTZVK} honor the `FPCR.FZ` control.

R_{TXKVK} The instructions specified in D_{HTZVK} that accumulate dot products of pairs of adjacent half-precision elements in the source vectors into single-precision elements in the *ZA* array honor the `FPCR.EBF16` control.

R_{JRRMJ} The instructions specified in D_{HTZVK} that multiply single elements from each source vector and accumulate their product into the *ZA* array perform a fused multiply-add to each accumulator tile or multi-vector operand element without intermediate rounding.

R_{NNCFV} The instructions specified in D_{HTZVK} that accumulate sum-of-products of pairs of adjacent half-precision elements in the source vectors into single-precision elements in the *ZA* array perform a fused sum-of-products without intermediate rounding of the products, but rounding the single-precision sum before addition to the accumulator tile or multi-vector operand element.

R_{QPKJC} The instructions specified in D_{HTZVK} that accumulate dot products of pairs of adjacent half-precision elements in the source vectors into single-precision elements in the *ZA* array generate the default NaN as intermediate sum-of-products when any of the following are true:

- Any multiplication input is a NaN.
- Any product is infinity × 0.0.
- There are infinite products with differing signs.

R_{ZBLND} The instructions specified in D_{HTZVK} that accumulate dot products of pairs of adjacent half-precision elements in the source vectors into single-precision elements in the *ZA* array generate an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.

R_{RPSLK} When FEAT_AFP is implemented, the instructions specified in D_{HTZVK} honor the `FPCR.AH` and `FPCR.FIZ` controls.

I_{YPCHJ} When FEAT_AFP is implemented, the following alternate floating-point behaviors affect the instructions specified in D_{HTZVK}:

- When `FPCR.AH` is 1, the sign bit of a generated default NaN result is set to 1 instead of 0.
- When `FPCR.AH` is 1 and `FPCR.FZ` is 1, a denormal result, detected after rounding with an unbounded exponent has been applied, is flushed to zero.
- When `FPCR.AH` is 1, the `FPCR.FZ` control does not cause denormalized inputs to be flushed to zero.
- When `FPCR.FIZ` is 1, all denormalized single-precision and double-precision inputs are flushed to zero.

Part C
SME System Level Programmers' Model

Chapter C1
System management

C1.1 Overview

I_{ZTQKZ} The SME System Management architecture provides mechanisms for system software to:

- Discover the presence of SME.
- Discover the capabilities of SME.
- Control SME usage.
- Monitor SME usage.

The architecture consists of extensions to processor mode, the Exception model, and System registers for trap control and identification.

I_{PZVYW} SME extends the AArch64 System registers and processor state, by introducing the following:

- An SME presence identification field added to `ID_AA64PFR1_EL1`.
- An SME-specific ID register, `ID_AA64SMFR0_EL1`, for SME feature discovery.
- Configuration settings for SME in `CPACR_EL1`, `CPTR_EL2`, `CPTR_EL3`, `HCR_EL2`, `HCRX_EL2`, `HFGRTR_EL2`, and `HFGWTR_EL2`.
- An SME exception type, with new `ESR_ELx.EC` and `ESR_ELx.ISS` encodings.
- A field in `ESR_ELx.ISS` to signal that an imprecise `FAR_ELx` value has been reported on a synchronous Data Abort exception.
- SME controls to set the Effective Streaming SVE vector length in `SMCR_EL1`, `SMCR_EL2`, and `SMCR_EL3`.
- ID and control registers that influence streaming execution priority in multiprocessor systems: `SMIDR_EL1`, `SMPRI_EL1`, and `SMPRIMAP_EL2`.
- Fields that enable the software thread ID register `TPIDR2_EL0` in `SCR_EL3`, `SCTLR_EL2`, and `SCTLR_EL1`.

I_{HVHCP} SME2 extends the SME System registers as follows:

- The SMTC exception syndrome field for an exception due to SME functionality in ESR_EL1, ESR_EL2, and ESR_EL3 is extended to identify trapping of accesses to the *ZT0* register.
- A value is added to the ID_AA64PFR1_EL1.SME field, to identify the presence of the *ZT0* register.
- An SMEver field is added to ID_AA64SMFR0_EL1 to indicate the presence of the mandatory SME2 instructions.
- The existing F64F64 and I16I64 fields in ID_AA64SMFR0_EL1 are extended to cover multi-vector instructions that generate double-precision and 64-bit integer results.
- A BI32I32 field is added to ID_AA64SMFR0_EL1 to identify the presence of SME2 instructions that accumulate thirty-two 1-bit binary outer products into 32-bit integer tiles.
- A I16I32 field is added to ID_AA64SMFR0_EL1 to identify the presence of SME2 instructions that accumulate 16-bit outer products into 32-bit integer tiles.
- An EZT0 field is added to the SMCR_EL1, SMCR_EL2, and SMCR_EL3 registers, to enable access to the *ZT0* register.

See also:

- C1.3.1 *CPACR_EL1*.
- C1.3.2 *CPTR_EL2*.
- C1.3.3 *CPTR_EL3*.
- C1.3.4 *ESR_EL1, ESR_EL2, and ESR_EL3*.
- C1.3.5 *HCR_EL2*.
- C1.3.6 *HCRX_EL2*.
- C1.3.7 *HFGRTR_EL2*.
- C1.3.8 *HFGWTR_EL2*.
- C1.3.9 *ID_AA64PFR1_EL1*.
- C1.3.11 *SCR_EL3*.
- C1.3.12 *SCTLR_EL1*.
- C1.3.13 *SCTLR_EL2*.
- C1.4.1 *ID_AA64SMFR0_EL1*.
- C1.4.2 *SMCR_EL1*.
- C1.4.3 *SMCR_EL2*.
- C1.4.4 *SMCR_EL3*.
- C1.4.5 *SMIDR_EL1*.
- C1.4.6 *SMPRI_EL1*.
- C1.4.7 *SMPRIMAP_EL2*.
- C1.4.8 *SVCR*.

C1.1.1 Identification

I_{XTNNP} ID_AA64PFR1_EL1.SME indicates whether SME is implemented in a PE.

I_{RLFXX} If SME is implemented, the SME features that are implemented in a PE are determined from ID_AA64SMFR0_EL1.

See also:

- C1.3.9 *ID_AA64PFR1_EL1*.
- C1.4.1 *ID_AA64SMFR0_EL1*.

C1.1.2 Traps and exceptions

D_{DMBHW} The SME-related instructions that can be configured to trap by CPACR_EL1, CPTR_EL2, and CPTR_EL3 controls, unless otherwise stated, include all of the following:

- SME data-processing instructions.
- SME mode change instructions SMSTART and SMSTOP.
- AArch64 MRS and MSR instructions which directly access any of the SVCR, SMCR_EL1, SMCR_EL2, or SMCR_EL3 registers.

D_{RGBVJ} The following SME2 instructions that access the *ZT0* register can be configured to trap by `SMCR_ELx.EZT0` controls:

- `LDR` (ZT0).
- `LUTI2`, `LUTI4`.
- `MOVT`.
- `STR` (ZT0).
- `ZERO` (ZT0).

I_{MQBFG} Execution of SME-related instructions can be trapped by supervisor software. The mechanisms provided are:

- `CPACR_EL1`, which enables execution of SME-related instructions at EL0 or EL1 to be trapped to EL1 or EL2.
- `CPTR_EL2`, which enables execution of SME-related instructions at EL0, EL1 or EL2 to be trapped to EL2.
- `CPTR_EL3`, which enables execution of SME-related instructions at any Exception level to be trapped to EL3.

I_{TLWWF} Execution of SME2 instructions that access *ZT0* can be trapped by supervisor software. The mechanisms provided are:

- `SMCR_EL1.EZT0`, which enables execution of SME2 instructions that access *ZT0* at EL0 or EL1 to be trapped to EL1 or EL2.
- `SMCR_EL2.EZT0`, which enables execution of SME2 instructions that access *ZT0* at EL0, EL1, or EL2 to be trapped to EL2.
- `SMCR_EL3.EZT0`, which enables execution of SME2 instructions that access *ZT0* at any Exception level to be trapped to EL3.

I_{XKMKY} SME adds an exception syndrome value `0b011101` (`0x1D`) which is used to identify instructions that are trapped by any of the following:

- The SME controls in `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3`.
- The SME2 controls `SMCR_EL1.EZT0`, `SMCR_EL2.EZT0`, and `SMCR_EL3.EZT0`.
- The `PSTATE.SM` and `PSTATE.ZA` mode.

I_{FWJZB} Exceptions reported with the exception syndrome value `0b011101` (`0x1D`) are mapped onto the Trap type in section *Exceptions to Exception elements and encoding* of chapter *The Embedded Trace Extension* and section *Filtering on type* of chapter *The Branch Record Buffer Extension* in *Arm® Architecture Reference Manual for A-profile architecture* [1].

See also:

- C1.2.1 *Exception priorities*.
- C1.3.1 *CPACR_EL1*.
- C1.3.2 *CPTR_EL2*.
- C1.3.3 *CPTR_EL3*.
- C1.3.4 *ESR_EL2, and ESR_EL3*.
- C1.4.2 *SMCR_EL1*.
- C1.4.3 *SMCR_EL2*.
- C1.4.4 *SMCR_EL3*.

C1.1.3 Vector lengths

D_{QQRNR} The *Effective Streaming SVE vector length* (SVL) is the accessible length in bits of the *ZA* array vectors and Streaming SVE vector registers at the current Exception level. SVL is determined by the `LEN` field of the appropriate SME `SMCR_ELx` registers, as defined in rule R_{GWVHP}.

I_{BHFWG} SVL is used explicitly by the unpredicated SME `LDR` (vector), `STR` (vector), and `ZERO` (tile) instructions which can access the *ZA* array irrespective of whether the PE is in *Streaming SVE mode*.

R_{VCQBB} The *Effective SVE vector length* (VL) is equal to SVL when the PE is in *Streaming SVE mode*.

I_{LRBQY} VL is determined by the `LEN` field of the `ZCR_ELx` registers when the PE is not in *Streaming SVE mode* and FEAT_SVE is implemented.

R_{JRCSH} An implementation is permitted to support any subset of the architecturally defined SVL values.

I_{FQKMN} For example, this means that the set of supported SVLs might be discontiguous and might not start at the smallest permitted SVL.

R_{RZNVH} An implementation is permitted but not required to support more than one SVL.

R_{WDKGR} An implementation is permitted to support a set of SVLs that do not overlap with the set of VLs that are supported when the PE is not in *Streaming SVE mode*.

I_{GZRPL} There is no requirement for the Maximum implemented Streaming SVE vector length to be greater than or equal to the Maximum implemented SVE vector length.

R_{GWVHP} The Effective Streaming SVE vector length at a given Exception level is determined from the requested length, encoded as a multiple of 16 bytes in `SMCR_EL1.LEN`, `SMCR_EL2.LEN`, or `SMCR_EL3.LEN`, according to the Exception level, following these steps:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, the Effective length is the minimum implemented Streaming SVE vector length.
2. If this is the highest implemented Exception level and the requested length is greater than the maximum implemented Streaming SVE vector length, then the Effective length is the maximum implemented Streaming SVE vector length.
3. If this is not the highest implemented Exception level and the requested length is greater than the Effective length at the next more privileged implemented Exception level in the current Security state, then the Effective length at the more privileged Exception level is used.
4. If the requested length is not supported by the implementation, then the Effective length is the highest supported Streaming SVE vector length that is less than the requested length.
5. Otherwise, the Effective length is the requested length.

I_{VRRYR} The set of supported values of SVL for Exception level ELx (where ELx is EL1, EL2 or EL3) can be discovered by privileged software in a similar way to determining the set of supported values of VL. For example, when SME is enabled by the appropriate control fields in `CPACR_EL1`, `CPTR_EL2` and `CPTR_EL3`:

1. Request an out of range vector length of 8192 bytes by writing `0x1ff` to `SMCR_ELx[8:0]`.
2. Use the SME `RDSVL` instruction to read SVL.
3. If `SMCR_ELx` requests a supported Streaming SVE vector length, the requested length in bytes will be returned by `RDSVL`.
4. If `SMCR_ELx` requests an unsupported Streaming SVE vector length, a supported length in bytes will be returned by `RDSVL`.
5. If the returned length is less than or equal to the requested vector length, and greater than 16 bytes (128 bits), then request the next lower vector length by writing `(len/16)-2` to `SMCR_ELx[8:0]` and go to step 2.

R_{YRPDH} When SVL changes from a smaller to a larger value without leaving *Streaming SVE mode*, a new area of storage becomes architecturally visible in the Streaming SVE registers and, if enabled, the *ZA* storage. The values in the area common to the previous and current length are preserved, and the values in the newly accessible area are a CONSTRAINED UNPREDICTABLE choice between the following:

- Zero.
- The value the bits had before executing at the more constrained size.

I_{HGYPV} The SVL might change without leaving *Streaming SVE mode* because of an explicit action such as a write to `SMCR_ELx`, or an implicit action such as taking an exception to an Exception level with a less constrained SVL.

I_{PDLWX} The SVL can be raised and then restored to a previous value without affecting the original contents of the Streaming SVE registers and the *ZA* storage.

I_{FMWZZ} Supervisory software must guarantee that values generated by one body of software are not observable by another body of software in a different trust or security scope. When SVL is increased, steps must be taken to ensure the newly accessible area does not contain values unrelated to another body of software. This might be achieved by ensuring that the PE exits *Streaming SVE mode* and disables the *ZA* storage when performing a context switch, or by explicitly resetting all register values.

I_{BRMMV} System software provides a maximum SVL to lower-privileged software, which might further constrain the SVL. However, system software must initialize and context switch values consistent with the maximum SVL provided and should not make assumptions about any smaller size being in use by lower-privileged software. For example, if a hypervisor exposes an SVL of 512 to a VM, that VM might choose to constrain SVL to 256. The hypervisor should still save and restore 512-bit vectors to prevent leakage of values between VMs, because the VM might later raise its SVL to 512 and must not be able to observe values created by other software in the newly visible upper portion of the registers.

See also:

- C1.4.2 *SMCR_EL1*.
- C1.4.3 *SMCR_EL2*.
- C1.4.4 *SMCR_EL3*.

C1.1.4 Streaming execution priority

D_{DXMSW} Streaming execution refers to the execution of instructions by a PE when that PE is in *Streaming SVE mode*.

I_{CMRVS} Arm expects a variety of implementation styles for SME, including styles where more than one PE shares SME and Streaming SVE compute resources.

I_{PQNJS} Shared SME and Streaming SVE compute resources are called *Streaming Mode Compute Unit* (SMCU).

I_{MZXWD} For implementations that share an SMCU, this architecture provides per-PE mechanisms that software can use to dynamically prioritize performance characteristics experienced by each PE.

See also:

- C1.2.4 *Streaming execution priority for shared implementations*.

C1.2 Processor behavior

C1.2.1 Exception priorities

I_{QZNRN} I_{ZFGJP} in the *Prioritization of Synchronous exceptions taken to AArch64 state* section of *Arm® Architecture Reference Manual for A-profile architecture* [1] provides a full list of exception priorities. The rules in this section provide additional detail for the prioritization of SME-related exceptions.

R_{GTKQD} Exceptions due to configuration settings and modes resulting from the attempted execution of an SME data-processing instruction are evaluated in the following order from highest to lowest priority:

1. If FEAT_SME is not implemented, then SME and SME2 instructions are UNDEFINED.
2. If FEAT_SME2 is not implemented, then SME2 instructions are UNDEFINED.
3. If `CPACR_EL1.SMEN` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x1D` with `ISS` code `0x0000000`.
4. If `CPACR_EL1.FPEN` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x07`.
5. If `CPTR_EL2.SMEN` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x1D` with `ISS` code `0x0000000`.
6. If `CPTR_EL2.FPEN` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x07`.
7. If `CPTR_EL2.TSM` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x1D` with `ISS` code `0x0000000`.
8. If `CPTR_EL2.TFP` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x07`.
9. If `CPTR_EL3.ESM` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x1D` with `ISS` code `0x0000000`.
10. If `CPTR_EL3.TFP` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x07`.
11. If the PE is not in *Streaming SVE mode*, SME and SME2 instructions that access the SVE registers *Z0-Z31* or *P0-P15* generate an SME exception, reported using `ESR_ELx.EC` value `0x1D` with `ISS` code `0x0000002`.
12. If the *ZA* storage is disabled, SME and SME2 instructions that access *ZA* and SME2 instructions that access *ZT0* generate an SME exception, reported using `ESR_ELx.EC` value `0x1D` with `ISS` code `0x0000003`.
13. If accesses to *ZT0* are not enabled according to R_{CSPYJ}, then the SME2 instructions that access *ZT0* generate an SME exception, reported using `ESR_ELx.EC` value `0x1D` with `ISS` code `0x0000004`.
14. Otherwise, the instruction executes.

R_{XQKHH} Exceptions due to configuration settings resulting from attempted execution of `MRS` or `MSR` instructions which directly access one of the `SVCR`, `SMCR_EL1`, `SMCR_EL2`, or `SMCR_EL3` registers, are evaluated in the following order from highest to lowest priority:

1. If FEAT_SME is not implemented, then the instruction is UNDEFINED.
2. If `CPACR_EL1.SMEN` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x1D`, with `ISS` code `0x0000000`.
3. If `CPTR_EL2.SMEN` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x1D`, with `ISS` code `0x0000000`.
4. If `CPTR_EL2.TSM` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x1D`, with `ISS` code `0x0000000`.
5. If `CPTR_EL3.ESM` configures the instruction to trap, it is reported using `ESR_ELx.EC` value `0x1D`, with `ISS` code `0x0000000`.
6. Otherwise, the instruction executes.

R_{PLYVH} Exceptions due to configuration settings and modes resulting from the attempted execution of an SVE or SVE2 instruction when FEAT_SVE is not implemented or when the PE is in *Streaming SVE mode*, are evaluated in the following order from highest to lowest priority:

1. If FEAT_SME is not implemented and FEAT_SVE is not implemented, then the instruction is UNDEFINED.
2. If FEAT_SVE is not implemented and the instruction is illegal when the PE is in *Streaming SVE mode*, then the instruction is UNDEFINED.

3. If FEAT_SME is not implemented and the instruction is defined as part of the SME architecture in D1.2 *SVE2 data-processing instructions*, then the instruction is UNDEFINED.
4. If CPACR_EL1.SMEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x1D with ISS code 0x0000000.
5. If CPACR_EL1.FPEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
6. If CPTR_EL2.SMEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x1D with ISS code 0x0000000.
7. If CPTR_EL2.FPEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
8. If CPTR_EL2.TSM configures the instruction to trap, it is reported using ESR_ELx.EC value 0x1D with ISS code 0x0000000.
9. If CPTR_EL2.TFP configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
10. If CPTR_EL3.ESM configures the instruction to trap, it is reported using ESR_ELx.EC value 0x1D with ISS code 0x0000000.
11. If CPTR_EL3.TFP configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
12. If the PE is in *Streaming SVE mode* and the SVE instruction is *illegal* in that mode, then an SME exception is taken, using ESR_ELx.EC value 0x1D with ISS code 0x0000001.
13. If the PE is not in *Streaming SVE mode* and FEAT_SVE is not implemented, then an SME exception is taken, using ESR_ELx.EC value 0x1D with ISS code 0x0000002.
14. Otherwise, the instruction executes.

R_{ZTKXF} Exceptions due to configuration settings resulting from the attempted execution of an SVE or SVE2 instruction when FEAT_SVE is implemented and when the PE is not in *Streaming SVE mode*, are evaluated in the following order from highest to lowest priority:

1. If FEAT_SME is not implemented and the instruction is defined as part of the SME architecture in D1.2 *SVE2 data-processing instructions*, then the instruction is UNDEFINED.
2. If CPACR_EL1.ZEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x19.
3. If CPACR_EL1.FPEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
4. If CPTR_EL2.ZEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x19.
5. If CPTR_EL2.FPEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
6. If CPTR_EL2.TZ configures the instruction to trap, it is reported using ESR_ELx.EC value 0x19.
7. If CPTR_EL2.TFP configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
8. If CPTR_EL3.EZ configures the instruction to trap, it is reported using ESR_ELx.EC value 0x19.
9. If CPTR_EL3.TFP configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
10. Otherwise, the instruction executes.

R_{DTCLZ} Exceptions due to configuration settings and modes resulting from the attempted execution of an AArch64 Advanced SIMD and floating-point instruction when the PE is in *Streaming SVE mode* are evaluated in the following order from highest to lowest priority:

1. If CPACR_EL1.FPEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
2. If CPTR_EL2.FPEN configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
3. If CPTR_EL2.TFP configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
4. If CPTR_EL3.TFP configures the instruction to trap, it is reported using ESR_ELx.EC value 0x07.
5. If the instruction is *illegal* when the PE is in *Streaming SVE mode*, then an SME exception is taken, using ESR_ELx.EC value 0x1D with ISS code 0x0000001.
6. Otherwise, the instruction executes.

I_{NWNQZ} When the PE is in *Streaming SVE mode* or FEAT_SVE is not implemented, the CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, and CPTR_EL3.ESM controls configure SVE instructions to trap, and the CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, and CPTR_EL3.EZ controls do not cause any SVE instructions to be trapped.

I_{PKGPR} When the PE is not in *Streaming SVE mode* and FEAT_SVE is implemented, the CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, and CPTR_EL3.EZ controls configure SVE instructions to trap, and the CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, and CPTR_EL3.ESM controls do not cause any SVE instructions to be trapped.

R_{ZZBRC} An Undefined Instruction exception due to the pairing of an SVE MOVPRFX with an instruction which cannot be predictably prefixed has a higher exception priority than a PSTATE mode-dependent SME exception with

ESR_ELx.EC value 0x1D and an ISS code that is not 0x0000000.

See also:

- *Prioritization of Synchronous exceptions taken to AArch64 state* in *Arm® Architecture Reference Manual for A-profile architecture* [1].
- C1.3.1 *CPACR_EL1*.
- C1.3.2 *CPTR_EL2*.
- C1.3.3 *CPTR_EL3*.
- C1.3.4 *ESR_EL1, ESR_EL2, and ESR_EL3*.
- Chapter E1 *Instructions affected by SME*.

C1.2.2 Synchronous Data Abort

R_{JXPNL} If SME is implemented, then when the PE takes a Data Abort exception that sets ESR_ELx.ISV to 0 and is caused by either an SME load/store instruction, or an SVE contiguous vector load/store instruction when the PE is in Streaming SVE mode, the PE:

- Sets ESR_ELx.FnP to 1 if the value written to the corresponding FAR_ELx might not be the same as the faulting virtual address that generated the Data Abort.
- Otherwise, sets ESR_ELx.FnP to 0.

R_{XMWQT} If the PE sets ESR_ELx.ISV to 0 and ESR_ELx.FnP to 0 when taking a Data Abort exception, then the PE sets the corresponding FAR_ELx to any address within the *naturally-aligned fault granule* which contains the faulting virtual address that generated the Data Abort.

D_{BRGHW} The *naturally-aligned fault granule* is one of the following:

- A 16-byte tag granule when ESR_ELx.DFSC is 0b010001, indicating a Synchronous Tag Check fault.
- An IMPLEMENTATION DEFINED granule when ESR_ELx.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault.
- Otherwise, the smallest implemented translation granule.

See also:

- FAR_EL1.
- FAR_EL2.
- FAR_EL3.
- C1.3.4 *ESR_EL1, ESR_EL2, and ESR_EL3*.

C1.2.3 Validity of SME and SVE state

I_{VBJBR} The Effective values of PSTATE.SM and PSTATE.ZA configure whether SME architectural state is valid and accessible.

I_{WFHKZ} Fields in CPACR_EL1, CPTR_EL2, and CPTR_EL3 configure whether SME-related instructions can be executed or are trapped.

I_{KKFXN} The fields SMCR_EL1.EZT0, SMCR_EL2.EZT0, and SMCR_EL3.EZT0 configure whether SME2 instructions that access the *ZT0* register can be executed or are trapped.

R_{CSPYJ} When SME2 is implemented, accesses to the *ZT0* register are enabled when all of the following are true:

- The access is from EL0 and SMCR_EL1.EZT0 is 1, if EL2 is implemented and enabled in the current Security state and HCR_EL2.{E2H,TGE} is not {1,1}.
- The access is from EL1 and SMCR_EL1.EZT0 is 1.
- The access is from EL0, EL1, or EL2 and SMCR_EL2.EZT0 is 1, if EL2 is implemented and enabled in the current Security state.
- The access is from EL0, EL1, EL2, or EL3 and SMCR_EL3.EZT0 is 1.

R_{XCCXW} The controls for trapping SME-related instructions and the controls for the validity of SME architectural state are independent.

I_{JGRTR} Because the trap and architectural state validity are controlled independently, the following scenarios are all permissible:

- Instructions trap, state invalid.
  - For example, an OS traps the first usage of SME-related instructions by a process.
- Instructions trap, state valid.
  - For example, a process was running with valid SME architectural state and an OS configures traps to detect when the next usage of SME architectural state occurs.
  - Enabling the trap does not affect or corrupt the SME architectural state.
- Instructions permitted, state invalid.
  - For example, a process is permitted to execute SME-related instructions but is currently not running in *Streaming SVE mode*. SME data-processing instructions which access SVE vector or predicate registers are *illegal* and are trapped, but SVE instructions operate on the Non-streaming SVE register state. The process can execute an `SMSTART` instruction to enter *Streaming SVE mode*.
  - For example, a process is running in *Streaming SVE mode*, but is not enabled access to the *ZA* storage. SME instructions that access *ZA* are *illegal* and are trapped, but the process can execute an `SMSTART` instruction to enable access to the *ZA* storage.
- Instructions permitted, state valid.
  - For example, a process is running in *Streaming SVE mode*, and has enabled access to *ZA* storage.

R_{SWQGH} An exception return from AArch64 to AArch32 Execution state does not change the values of `PSTATE.SM` and `PSTATE.ZA`.

R_{MZLVB} An exception taken from AArch32 to AArch64 Execution state does not change the values of `PSTATE.SM` and `PSTATE.ZA`.

R_{GXKNK} When a PE is executing in the AArch32 Execution state, the Effective value of `PSTATE.SM` is 0.

I_{MWQNV} When `PSTATE.SM` is 1, a change in Execution state from AArch64 to AArch32, or from AArch32 to AArch64, causes all implemented bits of the SVE registers (including SIMD&FP registers) and the `FPSR` to be reset to a fixed value, which software must mitigate.

I_{WYKRM} The Effective value of `PSTATE.ZA` does not change in AArch32 Execution state. Therefore, a transition between AArch64 and AArch32 Execution state when `PSTATE.ZA` is 1 has no effect on the contents of *ZA* storage, or the *ZT0* register when SME2 is implemented.

I_{VGWQW} An implementation might use the activity of the `PSTATE.SM` and `PSTATE.ZA` bits to influence the choice of power-saving states for both functional units and retention of architected state.

C1.2.4 Streaming execution priority for shared implementations

I_{YYRZQ} Execution of certain instructions by a PE in *Streaming SVE mode* might experience a performance dependency on other PEs in the system that are also executing instructions in *Streaming SVE mode*. For example, this might occur when a *Streaming Mode Compute Unit* (SMCU) is shared between PEs.

I_{WPQVV} The architecture provides a mechanism to control the streaming execution priority of a PE, in `SMPRI_EL1`. The streaming execution priority of a PE is relative to the streaming execution priority of other PEs, when a performance dependency exists between PEs executing in *Streaming SVE mode*.

D_{DGRTS} All PEs that share a given SMCU form a *Priority domain*.

D_{YQFWM} Different *Priority domains* represent unrelated SMCUs.

R_{WPVQK} All PEs in a *Priority domain* have the same value of `SMIDR_EL1.Affinity`.

R_{CVLSF} PEs in differing *Priority domains* have different values of `SMIDR_EL1.Affinity`.

R_{GGDRC}

The streaming execution priority in SMPRI_EL1 affects execution of a PE relative to all other PEs in the same *Priority domain*.

I_{WVGGW} System software can use the streaming execution priority mechanism to manage scenarios where multiple concurrent software threads contend on shared SMCUs.

R_{RQXFC} The streaming execution priority mechanism affects the execution of instructions by a shared SMCU when the PE is in *Streaming SVE mode* and does not directly control the execution of other types of instruction.

I_{HQXBH} The streaming execution priority mechanism is optional.

I_{KTYTD} An implementation that does not share SMCUs or has no performance dependency between PEs might not need to limit or prioritize execution of one PE relative to another.

I_{YBQNW} The architecture considers *Priority domains* to be non-overlapping sets, meaning that in a shared-SMCU system a PE is associated with at most one SMCU.

C1.2.4.1 Streaming execution context manageme

I_{PRNMJ} Arm expects that the SVE- and SME-related instructions used by save, resto and cle routines for the *Streaming SVE mode* SVE register state, the *ZA* array state, and the *ZT* register when S is implemented, are limited to using the following SME and SVE instructions:

- SME LDR (vector) and STR (vector) instructions.
- SME2 LDR (ZT0) and STR (ZT0) instructions.
- SVE LDR (vector) and STR (vector) instructions.
- SVE LDR (predicate) and STR (predicate) instructions.
- SME ZERO (tile) instruction.
- SME2 ZERO (ZT0) instruction.
- SVE DUP (immediate) instruction with zero immediate.
- SVE PFALSE instruction.

For implementations with a shared SMCU, PEs are expected to execute these instructions in a way that experiences a reduced effect of contention for the SMCU from other PEs, compared to other SME and SVE instructions executed in *Streaming SVE mode*.

C1.2.4.2 Streaming execution priority control

I_{RMDCP} The streaming execution priority is controlled by a 4-bit priority value. When the streaming execution priority mechanism is not supported, the priority value is ignored.

I_{FJQRG} A higher priority value corresponds to a higher streaming execution priority. Priority value 15 is the highest priority.

I_{QHKWP} The behavior of any given priority value relative to that of another PE is IMPLEMENTATION DEFINED.

C1.2.4.3 Streaming execution priority virtualization

I_{SQBCZ} The Effective streaming execution priority is either the value configured in SMPRI_EL1, or the value of SMPRI_EL1 mapped into a new value by indexing the fields in SMPRIMAP_EL2. This choice is affected by the current Exception level, and the HCRX_EL2.SMPME configuration.

I_{LSZZG} A hypervisor can use SMPRIMAP_EL2 to map the virtual streaming execution priority values written into SMPRI_EL1 by a guest OS into different physical priority values.

See also:

- C1.3.6 *HCRX_EL2*.
- C1.4.5 *SMIDR_EL1*.
- C1.4.6 *SMPRI_EL1*.

- C1.4.7 *SMPRIMAP_EL2*.

C1.2.5 Security considerations

I_{DXRGG} All SME load and store instructions adhere to the memory access permissions model in *The AArch64 Virtual Memory System Architecture* chapter of *Arm® Architecture Reference Manual for A-profile architecture* [1].

I_{MGLWR} SME architectural state can be access-controlled, meaning that higher levels of privilege can trap access to the state from the same or lower levels of privilege.

For example, execution of SME instructions including entry to or exit from *Streaming SVE mode* in EL0 might be trapped to EL2.

I_{CYPJJ} System software has controls available to save and restore state between unrelated pieces of software, and must ensure that steps are taken to preserve isolation and privacy.

I_{TDPHC} Operations performed in *Streaming SVE mode* respect the requirements of PSTATE.DIT. DIT requires data-independent timing when enabled.

C1.3 Changes to existing System registers

C1.3.1 CPACR_EL1

R_{QBTKS} If SME is implemented, the field `CPACR_EL1.SMEN` is defined at bits [25:24]. For more information, see CPACR_EL1.

I_{ZNSLS} The set of SME-related instructions trapped by this control is defined by rule D_{DMBHW} in C1.1.2 *Traps and exceptions*.

See also:

- C1.1.2 *Traps and exceptions*.
- C1.2.1 *Exception priorities*.
- C1.2.3 *Validity of SME and SVE state*.
- C1.3.4 *ESR_EL1, ESR_EL2, and ESR_EL3*.

C1.3.2 CPTR_EL2

R_{ZXZZC} If SME is implemented, FEAT_VHE is implemented, and `HC[illegible].E2H` is 1[illegible]e field `CPTR_EL2.SMEN` is defined at bits [25:24]. For more information, see CPTR_EL[illegible]

R_{QLKFH} When SME is implemented, FEAT_VHE is im[illegible]emented [illegible]nd `HCR_E[illegible].E2H` is 0, the field `CPTR_EL2.TSM` is defined at bit [12]. For more information, see CPTR[illegible]L2.

I_{DYLZC} The set of SME-related instructions t[illegible]ped by [illegible]s control is defined by rule D_{DMBHW} in C1.1.2 *Traps and exceptions*.

See also:

- C1.1.2 *Traps and exc[illegible]ptions*.
- C1.2.1 *Exceptio[illegible]riorities*.
- C1.2.3 *Validit[illegible]f SME and SVE sta*
- C1.3.4 *ESR[illegible]EL1, ESR[illegible]EL2, and ESR_EL3*.

C1.3.3 CPTR_EL[illegible]

R_{NPVSR} If S[illegible]E is im[illegible]ted, the field `CPTR_EL3.ESM` is defined at bit [12]. For more information, see CPTR_EL3.

I_{LGJZY} The set [illegible]SME-related instructions trapped by this control is defined by rule D_{DMBHW} in C1.1.2 *Traps and exceptions*.

See also:

- C1.1.2 *Traps and exceptions*.
- C1.2.1 *Exception priorities*.
- C1.2.3 *Validity of SME and SVE state*.
- C1.3.4 *ESR_EL1, ESR_EL2, and ESR_EL3*.

C1.3.4 ESR_EL1, ESR_EL2, and ESR_EL3

D_{DZSWB} If SME is implemented, an Exception Class value `0x1D` is added to `ESR_EL1`, `ESR_EL2`, and `ESR_EL3`, for exceptions taken from AArch64.

EC	ISS[2:0]	Meaning
0b011101	0b000	Access to SME functionality trapped as a result of CPACR_EL1.SMEN, CPTR_EL2.SMEN, CPTR_EL2.TSM, or CPTR_EL3.ESM, that is not reported using EC 0b000000.
0b011101	0b001	Illegal Advanced SIMD, SVE, or SVE2 instruction trapped because PSTATE.SM is 1
0b011101	0b010	Illegal SME instruction trapped because PSTATE.SM is 0
0b011101	0b011	Illegal SME instruction trapped because PSTATE.ZA is 0

Other values of ISS[2:0] are reserved.

D_{MSSSD} If SME2 is implemented, for Exception Class 0x1D, the following ISS value is added to ESR_EL1, ESR_EL2, and ESR_EL3:

EC	ISS[2:0]	Meaning
0b011101	0b100	Access to the SME2 ZT0 register trapped as a result of SMCR_EL1.EZT0, SMCR_EL2.EZT0, or SMCR_EL3.EZT0.

R_{DYSFV} If SME is implemented, then the following field is defined in the ESR_EL1, ESR_EL2, and ESR_EL3 ISS encoding for an exception from a Data Abort, when the DFSC value is 0b100100 (0x24) or 0b100101 (0x25):

Field	Name	Meaning
[15]	FnP	**When ISV == 0:** FAR not Precise **0b0** The FAR holds the faulting virtual address that generated the Data Abort. **0b1** The FAR holds any virtual address within the *naturally-aligned fault granule* (see D_{BRGHW}) that contains the faulting virtual address that generated a Data Abort exception due to an SVE contiguous vector load/store instruction when the PE is in Streaming SVE mode, or by an SME load/store instruction. On Warm reset, this field resets to an architecturally UNKNOWN value.

C1.3.5 HCR_EL2

R_{BWHVR} If SME is implemented, the HCR_EL2.TID3 field causes accesses to ID_AA64SMFR0_EL1 to be trapped.

R_{ZRBBT} If SME is implemented, the HCR_EL2.TID1 field causes accesses to SMIDR_EL1 to be trapped.

See also:

- HCR_EL2.
- ID_AA64SMFR0_EL1.
- SMIDR_EL1.

C1.3.6 HCRX_EL2

R_{YDHJV} If SME is implemented, the field HCRX_EL2.SMPME is defined at bit [5]. For more information, see HCRX_EL2.

See also:

- C1.2.4 *Streaming execution priority for shared implementations*.
- C1.4.6 *SMPRI_EL1*.
- C1.4.7 *SMPRIMAP_EL2*.

C1.3.7 HFGRTR_EL2

R_{SXDSB} If SME is implemented, the fields `HFGRTR_EL2.nTPIDR2_EL0` and `HFGRTR_EL2.nSMPRI_EL1` are defined at bits [55] and [54]. For more information, see HFGRTR_EL2.

See also:

- B1.2.4 *TPIDR2_EL0*.
- C1.4.6 *SMPRI_EL1*.

C1.3.8 HFGWTR_EL2

R_{XKLBN} If SME is implemented, the fields `HFGWTR_EL2.nTPIDR2_EL0` and `HFGWTR_EL2.nSMPRI_EL1` are defined at bits [55] and [54]. For more information, see HFGWTR_EL2.

See also:

- B1.2.4 *TPIDR2_EL0*.
- C1.4.6 *SMPRI_EL1*.

C1.3.9 ID_AA64PFR1_EL1

R_{KHPZL} If SME is implemented, the field `ID_AA64PFR1_EL1.SME` is defined at bits [27:24].

R_{LYHCJ} If SME2 is implemented, the value `0b0010` is added to `ID_AA64PFR1_EL1.SME` at bit position [27:24].

I_{DJQVZ} A nonzero value in `ID_AA64PFR1_EL1.SME` does not imply that `ID_AA64PFR0_EL1.SVE` must also contain a nonzero value.

See also:

- C1.3.9 *ID_AA64PFR1_EL1*.

C1.3.10 ID_AA64ZFR0_EL1

R_{SYRGK} If SME is implemented, the `ID_AA64ZFR0_EL1` register identifies the implemented features of the SVE instruction set when any of `ID_AA64PFR0_EL1.SVE` and `ID_AA64PFR1_EL1.SME` are nonzero.

R_{JLSQK} If SME is implemented, then `ID_AA64ZFR0_EL1.SVEver` has a nonzero value, indicating that *legal* SVE and SVE2 instructions can be executed when the PE is in *Streaming SVE mode*.

R_{RFZRM} If SME is implemented and `ID_AA64PFR0_EL1.SVE` is nonzero, then FEAT_SVE2 is implemented and SVE and SVE2 instructions can be executed when the PE is not in *Streaming SVE mode*.

R_{XFFLH} If SME is implemented and `ID_AA64PFR0_EL1.SVE` is zero, then FEAT_SVE is not implemented and the `ID_AA64ZFR0_EL1` fields named `F64MM`, `F32MM`, `SM4`, `SHA3`, `BitPerm`, and `AES` hold the value zero.

See also:

- ID_AA64PFR0_EL1, defined in *Arm® Architecture Registers, for A-profile architecture* [2].
- C1.3.9 *ID_AA64PFR1_EL1*.
- C1.3.10 *ID_AA64ZFR0_EL1*.

C1.3.11 SCR_EL3

R_{TCPTK} If SME is implemented, the field `SCR_EL3.EnTP2` is defined at bit [41]. For more information, see SCR_EL3.

See also:

- B1.2.4 *TPIDR2_EL0*.

C1.3.12 SCTLR_EL1

R_{NMVMQ} If SME is implemented, the field `SCTLR_EL1.EnTP2` is defined at bit [60]. For more information, see SCTLR_EL1.

R_{MXHMD} When EL2 is implemented and enabled in the current Security state and `HCR_EL2.{E2H, TGE}` is `{1, 1}`, the `SCTLR_EL1.EnTP2` control has no effect on execution at EL0 and the `SCTLR_EL2.EnTP2` control is used for this purpose.

See also:

- B1.2.4 *TPIDR2_EL0*.

C1.3.13 SCTLR_EL2

R_{KGMHQ} If SME is implemented and `HCR_EL2.{E2H, TGE}` is `{1, 1}`, the field `SCTLR_EL2.EnTP2` is defined at bit [60]. For more information, see SCTLR_EL2.

See also:

- B1.2.4 *TPIDR2_EL0*.

C1.3.14 ZCR_EL1, ZCR_EL2, and ZCR_EL3

I_{CRKQJ} If FEAT_SVE is implemented, then the `ZCR_ELx` registers have their described effect on the Effective SVE vector length only when the PE is not in *Streaming SVE mode*.

C1.4 SME-specific System registers

C1.4.1 ID_AA64SMFR0_EL1

I_{DVWNQ} The AArch64 SME Feature ID Register describes the set of implemented SME data-processing instructions.

D_{GRPHH} If SME is implemented, the register `ID_AA64SMFR0_EL1` is added. For more information, see ID_AA64SMFR0_EL1.

C1.4.2 SMCR_EL1

I_{KKHZL} The Streaming SVE Mode Control Register for EL1 configures the Effe[illegible]aming SVE vector length and accessibility of the SME2 *ZT0* register, when executing at EL1 or EL0.

R_{HRTZQ} If SME is implemented, the register `SMCR_EL1` is added. For more inf[illegible]ation, see SM[illegible]R_EL1.

R_{NWXVG} When EL2 is implemented and enabled in the current Secur[illegible]y state an[illegible]CR_EL2.{E2H, TGE} is {1, 1}, the `SMCR_EL1` register has no effect on execution at EL0 and EL[illegible] and the SMCR_EL[illegible]re[illegible]ster is used for this purpose.

C1.4.3 SMCR_EL2

I_{WTNZY} The Streaming Mode Control Register for EL2 conf[illegible]ures the E[illegible]ective Streaming SVE vector length and accessibility of the SME2 *ZT0* register when [illegible]ecuti[illegible]g at E[illegible]d at EL1 or EL0 in the same Security state as EL2.

R_{JPZPH} If SME is implemented, the regist[illegible] SMCR_EL[illegible] is add[illegible] For more information, see SMCR_EL2.

C1.4.4 SMCR_EL3

I_{VVCBL} The Streaming M[illegible] Control Register f[illegible] EL3 configures the Effective Streaming SVE vector length and accessibility of t[illegible] SME2 Z[illegible]0 register, when executing at EL3, EL2, EL1, or EL0.

R_{DBGWC} If SME is implemen[illegible]the regist[illegible]SMCR_EL3 is added. For more information, see SMCR_EL3.

C1.4.5 SMID[illegible]_EL1

I_{MZBSJ} The Stre[illegible]ing Mode Identification Register provides additional information about the *Streaming SVE mode* implementa[illegible]n.

D_{NBDMK} If SME is implemented, the register `SMIDR_EL1` is added. For more information, see SMIDR_EL1.

C1.4.6 SMPRI_EL1

I_{PJDXF} The Streaming Mode Priority register configures the streaming execution priority for instructions executed in *Streaming SVE mode* on a shared SMCU at any Exception level.

R_{JKMFH} If SME is implemented, the register `SMPRI_EL1` is added. For more information, see SMPRI_EL1.

R_{DWGZP} In an implementation that shares execution resources between PEs, higher streaming execution priority values are allocated more processing resource than other PEs configured with lower streaming execution priority values in the same *Priority domain*.

R_{DFQLX} The precise meaning and behavior of each streaming execution priority value is IMPLEMENTATION DEFINED.

I_{BLMYK} If system software does not support differentiation of streaming execution priority of threads, it is safe to use a value of 0 for all threads.

R_{SBCRG} All SMCUs in the system have a consistent interpretation of the streaming execution priority values.

See also:

- C1.2.4 *Streaming execution priority for shared implementations*.

C1.4.7 SMPRIMAP_EL2

I_{HVXRH} The Streaming Mode Priority Mapping register maps the current virtual streaming execution priority value to a physical streaming execution priority value for instructions executed in *Streaming SVE mode* on a shared SMCU at EL1 or EL0 in the same Security state as EL2.

D_{CHVZD} If SME is implemented, the register `SMPRIMAP_EL2` is added. For more information, see SMPRIMAP_EL2.

C1.4.8 SVCR

I_{JGCTD} The Streaming Vector Control Register provides direct access to the `PSTATE.SM` and `PSTATE.ZA` mode bits from any Exception level.

D_{JXVQJ} If SME is implemented, the register `SVCR` is added. For more information, see SVCR.

See also:

- B1.2.3 *Changing PSTATE.SM and PSTATE.ZA*.

Chapter C2
Interaction with other A-profile architectural features

I_MDMBB This section describes the interaction of SME with other aspects and features of the A-profile architecture.

It covers:

- Watchpoint
- Self-hosted debug
- External debug.
- Memory Tagging Extension (MTE).
- Reliability, Availability, and Serviceability (RAS).
- Memory System Resource Partitioning and Monitoring (MPAM).
- Transactional Memory Extension (TME).
- Memory consistency model.

See also:

- *Arm® Architecture Reference Manual for A-profile architecture* [1].
- *Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture* [4].
- *Arm® Architecture Reference Manual Supplement, The Transactional Memory Extension (TME), for A-profile architecture* [5].

C2.1 Watchpoints

R_{PDGZL} For a memory access or set of contiguous memory accesses generated by an SVE contiguous vector load/store instruction when the PE is in *Streaming SVE mode*, or by an SME load/store instruction, if a watchpoint matches a range where the lowest accessed address is rounded down to the nearest multiple of 16 bytes and the highest accessed address is rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint does not the match the range of the original access or set of contiguous accesses, then it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint debug event is triggered.

R_{XKRPV} If a watchpoint matches only the rounded access address ranges of Inactive elements in a predicated vector load/store instruction, then it does not trigger a Watchpoint debug event.

I_{YVSFL} If a Watchpoint debug event is triggered by a match on a rounded access address range that would not have been triggered by the original access address range, then this may report a false-positive match. Debug software must attempt to detect and step over false-positive matches. The architecture does not permit missed, or false-negative matches.

C2.1.1 Reporting watchpoints

R_{KDRCX} If SME is implemented, then the following fields are added to `ESR_EL1` and `ESR_EL2` in the ISS encoding for an exception from a Watchpoint exception, when the `EC` value is `0x34` or `0x35`.

Field	Name	Meaning
[24]	ISV	`RES0`
[23:18]	WPT	Watchpoint number, 0-15 inclusive. All other values are reserved.
[17]	WPTV	Watchpoint number Valid. **0b0** The `WPT` field is invalid, and holds an UNKNOWN value. **0b1** The `WPT` field is valid, and holds the number of a watchpoint that triggered a Watchpoint debug event.
[16]	WPF	Watchpoint might be false-positive. **0b0** The watchpoint matched the original access or set of contiguous accesses. **0b1** The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the nearest multiple of 16 bytes and the highest accessed address was rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint might not have matched the original access or set of contiguous accesses.
[15]	FnP	FAR not Precise. This field only has meaning if the `FAR` is valid; that is, when the `FnV` field is 0. If the `FnV` field is 1, the `FnP` field is 0. **0b0** If the `FnV` field is 0, the `FAR` holds the virtual address of an access or set of contiguous accesses that triggered a Watchpoint debug event. **0b1** The `FAR` holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered a Watchpoint debug event.

Field	Name	Meaning
[10]	FnV	FAR not Valid. **0b0** The FAR is valid, and its value is as described by the FnP field. **0b1** The FAR is invalid, and holds an UNKNOWN value.

R_{MFRPZ} If SME is implemented, then the following field is added to the EDDEVID1 register:

Field	Name	Meaning
[7:4]	HSR	Indicates support for the External Debug Halt Status Register (EDHSR). Defined values are: **0b0000** EDHSR not implemented, and the PE follows behaviors consistent with all of the EDHSR fields having a zero value. **0b0001** EDHSR implemented. All other values are reserved. If FEAT_SME is implemented, the permitted values are 0b0000 and 0b0001. If FEAT_SME is not implemented, the only permitted value is 0b0000.

R_{QBKWY} If SME is implemented, then the read-only External Debug Halt Status Register (EDHSR) may be implemented at offset 0x038. The field EDDEVID1.HSR1 indicates whether the EDHSR is implemented.

R_{LXGXN} If the EDHSR is implemented, it is in the Core power domain.

R_{SDSFM} If the EDHSR is implemented, then it is only valid when the PE is in Debug state and EDSCR.STATUS indicates a Watchpoint debug event (0b[illegible]1011), otherwise it is an UNKNOWN value.

The EDHSR fields are defined as follows:

Field	Name	Meaning
[23:18]	WPT	Watchpoint number, 0-15 inclusive. All other values are reserved.
[17]	WPTV	Watchpoint number Valid. **0b0** The WPT field is invalid, and holds an UNKNOWN value. **0b1** The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint debug event. On a Cold reset, this field resets to an architecturally UNKNOWN value.
[16]	WPF	Watchpoint might be false-positive. **0b0** The watchpoint matched the original access or set of contiguous accesses. **0b1** The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the nearest multiple of 16 bytes and the highest accessed address was rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint might not have matched the original access or set of contiguous accesses. On a Cold reset, this field resets to an architecturally UNKNOWN value.

Field	Name	Meaning
[15]	FnP	FAR not Precise. This field only has meaning if the EDWAR is valid; that is, when the FnV field is 0. If the FnV field is 1, the FnP field is 0. **0b0** If the FnV field is 0, the EDWAR holds the virtual address of an access or set of contiguous accesses that triggered a Watchpoint debug event. **0b1** The EDWAR holds any address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered a Watchpoint debug event. On a Cold reset, this field resets to an architecturally UNKNOWN value.
[10]	FnV	FAR not Valid. **0b0** The EDWAR is valid, and its value is as described by the FnP field. **0b1** The EDWAR is invalid, and holds an UNKNOWN value. On a Cold reset, this field resets to an architecturally UNKNOWN value.

R_{XWDJT} If the EDHSR is not implemented, then the PE must follow behaviors consistent with all of the EDHSR fields having a zero value.

R_{CXZCY} If a Watchpoint debug event is triggered by an SVE contiguous load/store instruction when the PE is in *Streaming SVE mode*, or by an SME load/store instruction, then the virtual address recorded in FAR_ELx or EDWAR must be derived from an address that is both:

- In the inclusive range between:
  - The lowest address accessed by the vector instruction that triggered the watchpoint, or the lowest rounded address as permitted by R_{PDGZL}.
  - The highest watchpointed address accessed by the vector instruction that triggered the watchpoint, or the highest watchpointed address in the address range permitted by R_{PDGZL}.
- Within a naturally-aligned block of memory.

R_{SQDKJ} If an instruction generates a watchpoint match where the the watchpointed data address or data address range is not accessed by the instruction, the PE:

- Sets ESR_ELx.WPF to 1, on taking a Watchpoint exception generated by the watchpoint match.
- Sets EDHSR.WPF to 1, on entering Debug state on a Watchpoint debug event generated by the watchpoint match.

Otherwise, ESR_ELx.WPF or EDHSR.WPF (as applicable) is set to an IMPLEMENTATION DEFINED choice of 0 or 1.

For example, when R_{PDGZL} applies, an SVE contiguous vector load/store instruction when the PE is in *Streaming SVE mode*, or an SME load/store instruction might generate a watchpoint match for a data address or data address range that the instruction does not access. Arm strongly recommends that ESR_ELx.WPF or EDHSR.WPF (as applicable) is set to 0 for all other cases.

R_{KSSHC} If a watchpoint matches an access that is due to an SVE contiguous load/store instruction when the PE is in Streaming SVE mode, or is due to an SME load/store instruction, then the PE:

- Sets ESR_ELx.FnV to an IMPLEMENTATION DEFINED value, 0 or 1, on taking a Watchpoint exception generated by the watchpoint match.
- Sets EDHSR.FnV to an IMPLEMENTATION DEFINED value, 0 or 1, on entering Debug state on a Watchpoint debug event generated by the watchpoint match.
- Otherwise, ESR_ELx.FnV or EDHSR.FnV (as applicable) is set to 0.

R_{RLWSF} When the PE sets ESR_ELx.FnV to 0 on taking a Watchpoint exception generated by the watchpoint match:

- If the lowest watchpointed address higher than or the same as the address recorded in FAR_ELx might not have been accessed by the instruction, other than as permitted by R_{PDGZL}, then the PE sets ESR_ELx.FnP to 1.

- Otherwise, the PE sets `ESR_ELx.FnP` to 0.

R_{CJWYX} When the PE sets `EDHSR.FnV` to 0 on entering Debug state on a Watchpoint debug event generated by a watchpoint match:

- If the lowest watchpointed address higher than or the same as the address recorded in `EDWAR` might not have been accessed by the instruction, other than as permitted by R_{PDGZL}, then the PE sets `EDHSR.FnP` to 1.
- Otherwise, the PE sets `EDHSR.FnP` to 0.

R_{DTWTH} When a Watchpoint exception is triggered by a watchpoint match:

- If the PE sets any of `ESR_ELx.FnV`, `ESR_ELx.FnP`, or `ESR_ELx.WPF` to 1, then the PE sets `ESR_ELx.WPTV` to 1.
- If the PE sets all of `ESR_ELx.FnV`, `ESR_ELx.FnP`, and `ESR_ELx.WPF` to 0, then the PE sets `ESR_ELx.WPTV` to an IMPLEMENTATION DEFINED value, 0 or 1.

R_{BNXVL} When an entry to Debug state is triggered by a watchpoint match:

- If the PE sets any of `EDHSR.FnV`, `EDHSR.FnP`, or `EDHSR.WPF` to 1, then the PE sets `EDHSR.WPTV` to 1.
- If the PE sets all of `EDHSR.FnV`, `EDHSR.FnP`, and `EDHSR.WPF` to 0, then the PE sets `EDHSR.WPTV` to an IMPLEMENTATION DEFINED value, 0 or 1.

R_{PVYNL} On a watchpoint match generated by watchpoint <*n*>:

- If the PE sets `ESR_ELx.WPTV` to 1 on taking a Watchpoint exception generated by the watchpoint match, then `ESR_ELx.WPT` is set to <*n*>.
- If the PE sets `EDHSR.WPTV` to 1 on entering Debug state on a Watchpoint debug event generated by the watchpoint match, then `EDHSR.WPT` is set to <*n*>.
- Otherwise, `ESR_ELx.WPT` or `EDHSR.WPT` (as applicable) are UNKNOWN.

R_{KHSFH} When an instruction generates multiple watchpoint matches and the PE sets `ESR_ELx.WPTV` or `EDHSR.WPTV` to 1, then it is UNPREDICTABLE which watchpoint is reported in `ESR_ELx.WPT` or `EDHSR.WPT` (as applicable).

D_{LXZPC} The *naturally-aligned block of memory* is all of the following:

- A power-of-two size.
- No larger than the `DC ZVA` block size if `ESR_ELx.FnP` or `EDHSR.FnP` (as appropriate) is 0.
- No larger than the smallest implemented translation granule if `ESR_ELx.FnP` or `EDHSR.FnP` (as appropriate) is 1.
- Contains a watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint, or a watchpointed address in the address range permitted by R_{PDGZL}.

The size of the block is IMPLEMENTATION DEFINED.

There is no architected means of discovering the size.

D_{LTGKY} A *watchpointed address* is an address that a watchpoint is watching.

C2.2 Self-hosted debug

I_{CDBZX} SME has no additional effect on self-hosted debug.

C2.3 External debug

R_{XQQRS} The following SME-related instructions are unchanged in Debug state:

- `MOVA` (tile to vector, single).
- `MOVA` (vector to tile, single).
- `MRS SVCR`.
- `MSR SVCR`.
- `RDSVL`.

R_{CSRRS} If SME2 is implemented, the following instructions are defined only when the PE is in Debug state:

- `MOVT` (ZT0 to scalar).
- `MOVT` (scalar to ZT0).

R_{TNZLR} All other SME-related instructions are CONSTRAINED UNPREDICTABLE in Debug state, with the same set of CONSTRAINED UNPREDICTABLE options as other instructions in Debug state, as defined in *Arm® Architecture Reference Manual for A-profile architecture* [1].

C2.4 Memory Tagging Extension (MTE)

The following rules apply when the optional FEAT_MTE feature is implemented.

R_{BGGMD} When the Memory Tagging Extension is implemented, it is IMPLEMENTATION DEFINED whether memory accesses due to SME, SVE, and SIMD&FP load and store instructions executed when the PE is in *Streaming SVE mode* will perform a Tag Check.

R_{GLYMK} When the Memory Tagging Extension is implemented, it is IMPLEMENTATION DEFINED whether memory accesses due to the following instructions will perform a Tag Check:

- SME `LDR` (vector) and `STR` (vector) instructions.
- SME2 `LDR` (ZT0) and `STR` (ZT0) instructions.

I_{RBPTM} An implementation of FEAT_MTE is only expected to perform Tag Checking when the PE is in *Streaming SVE mode* if it can do so with a similar relative performance impact to Tag Checking memory accesses due to SVE and SIMD&FP load and store instructions executed when the PE is not in *Streaming SVE mode*.

C2.5 Reliability, Availability, and Serviceability (RAS)

R_{RVYHY} Rules I_{NTXKV} and R_{NQDWB} in the *RAS PE Architecture* chapter of *Arm® Architecture Reference Manual for A-profile architecture* [1] are extended by adding the *ZA* storage, and the *ZT0* register when SME2 is implemented, to any list of program-visible architectural state or registers that includes the SIMD&FP or SVE registers.

C2.6 Memory Partitioning and Monitoring (MPAM)

The following System registers are modified or added when the optional FEAT_MPAM feature is implemented.

C2.6.1 MPAMSM_EL1

R_{PPJJP} If SME and FEAT_MPAM are implemented, the register `MPAMSM_EL1` is added to generate MPAM labels for memory requests issued at any Exception level by the following instructions:

- SME load/store instructions.
- When the PE is in *Streaming SVE mode*, SVE and SIMD&FP load/store instructions, and SVE prefetch instructions.

For more information, see MPAMSM_EL1.

C2.6.2 MPAM2_EL2

R_{LJVWP} If SME, FEAT_MPAM, and EL2 are implemented, the fi[illegible] `MPAM2_[illegible]2.EnMPAMS[illegible]` is defined at bit [50]. For more information, see MPAM2_EL2.

C2.7 Transactional Memory Extension (TME)

The following rules apply when the optional FEAT_TME feature is implemented.

R_{KHVVR} Executing a TSTART instruction when PSTATE.SM is 1 fails the transaction with the ERR cause.

R_{LYBMR} Executing any of the following instructions while in Transactional state will cause the transaction to fail with the ERR cause:

- An SME LDR (vector), STR (vector), or ZERO (tile) instruction.
- An SME2 LDR (ZT0), STR (ZT0), or ZERO (ZT0) instruction.

I_{TNZSW} Any MSR instruction that writes to the PSTATE.SM or PSTATE.ZA bits in Transactional state, including the SMSTART and SMSTOP aliases, are UNDEFINED according to the rules in *Arm® Architecture Reference Manual Supplement, The Transactional Memory Extension (TME), for A-profile architecture* [5] and will cause the transaction to fail with the ERR cause, without trapping.

For more information about the rules, see the "MSR (register)" and "MSR (immediate)" sections in *Arm® Architecture Reference Manual Supplement, The Transactional Memory Extension (TME), for A-profile architecture* [5].

C2.8 Memory consistency model

R_{BQSCG} Any access to memory performed by an SME load/store instruction, or an SVE load/store instruction when the PE is in *Streaming SVE mode*, is subject to the same rules that govern an SVE memory access in *The AArch64 Application Level Memory Model* chapter of *Arm® Architecture Reference Manual for A-profile architecture* [1].

R_{SMWPF} When the PE is in *Streaming SVE mode*, any access to memory performed by a SIMD&FP load/store instruction is subject to the same rules that govern a SIMD&FP memory access in *The AArch64 Application Level Memory Model* chapter of *Arm® Architecture Reference Manual for A-profile architecture* [1].

R_{XHFBX} When the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level, any access to Device memory performed by a SIMD&FP load/store instruction is relaxed such that it might behave as if:

- The Gathering attribute is set, regardless of the configured value of the nG attribute.
- The Reordering attribute is set, regardless of the configured value of the nR attribute.
- The Early Acknowledgement attribute is set, regardless of the configured value of the nE attribute.

Whether or not attributes are classified as mismatched is determined strictly by the memory attributes derived from the translation table entry.

R_{HBBTV} If a pair of memory reads access the same location, and at least one of the reads is generated by a SIMD&FP load instruction then, for a given observer, the pair of reads is not required to satisfy the *internal visibility* requirement when the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level.

Part D
SME Instruction Set

Chapter D1
SME instructions

This chapter defines the instructions added to the A64 instruction set when SME is implemented.

This content is from the **2022-12** version of *Arm® A64 Instruction Set Architecture, for A-profile architecture* [3], which contains the definitive details of the instruction set.

D1.1 SME and SME2 data-processing instructions

The following SME data-processing instructions are added by the SME or SME2 architecture.

The SME data-processing instructions are available when SME or SME2 is implemented, and are identified by the presence of the FEAT_SME symbol, or a call to one of the `HaveSME` pseudocode functions.

The SME2 data-processing instructions are available when SME2 is implemented, and are identified by the presence of the FEAT_SME2 symbol, or a call to the `HaveSME2` pseudocode function.

D1.1.1 ADD (to vector)

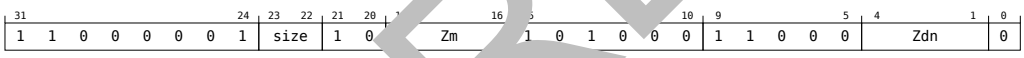
Add replicated single vector to multi-vector with multi-vector result

Add elements of the second source vector to the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

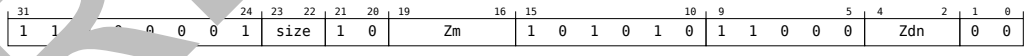
Two registers
(FEAT_SME2)




```
ADD     { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
ADD     { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'00');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = element1 + element2;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.2 ADD (array accumulators)

Add multi-vector to ZA array vector accumulators

The instruction operates on two or four ZA single-vector groups.

Destructively add all elements of the two or four source vectors to the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

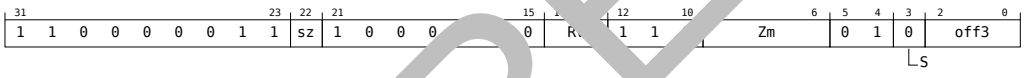
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer v ant is implem ted.

It has encodings from 2 classes: Two ZA single-vectors and Fo ZA si -vectors

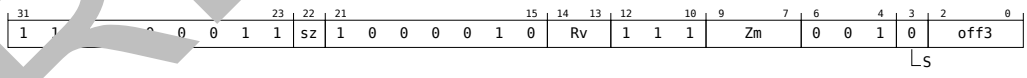
Two ZA single-vectors
(FEAT_SME2)




```
ADD     ZA.<T>[<Wv>, <offs>{, V 2}], { Zm1>.<T>-<Zm2>.<T> }
```

```
if !HaveSME2() then UNDEFIN ;
if sz == '1' && !HaveSME 4() then UN FINED;
integer v = UInt('010' );
constant integer esiz = 32 << nt(sz);
integer m = UInt(Zm 0');
integer offset = nt(off3);
constant intege nreg =
```

Four ZA single-vec
(FEAT_ IE




```
ADD     ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = ZAvector[
    bits(VL) operand2 = Z[m+r, VL]
    for e = 0 to elements-1
        bits(esize) element
        bits(esize) eleme
        Elem[result, e,
    ZAvector[vec, VL] =
    vec = vec + vst
```

D1.1.3 ADD (array results, multiple and single vector)

Add replicated single vector to multi-vector with ZA array vector results

The instruction operates on two or four ZA single-vector groups.

Add all corresponding elements of the second source vector and the two or four first source vectors and place the results in the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

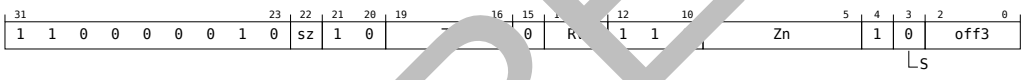
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

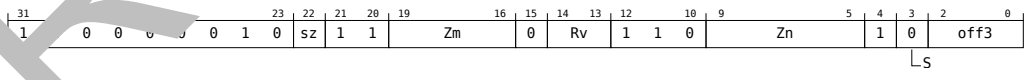
Two ZA single-vectors
(FEAT_SME2)




```
ADD     ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
ADD     ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 + element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.4 ADD (array results, multiple vectors)

Add multi-vector to multi-vector with ZA array vector results

The instruction operates on two or four ZA single-vector groups.

Add all corresponding elements of the two or four second source vectors and first source vectors and place the results in the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

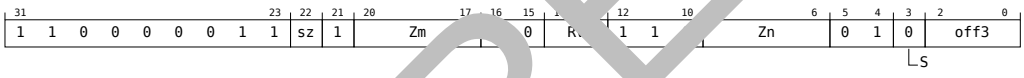
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

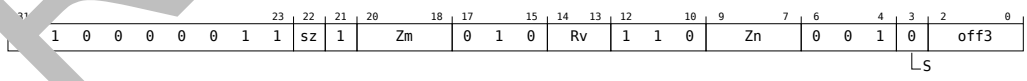
Two ZA single-vectors
(FEAT_SME2)




```
ADD     ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
ADD     ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 + element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.5 ADDHA

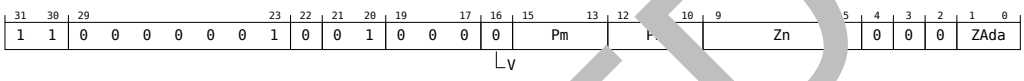
Add horizontally vector elements to ZA tile

Add each element of the source vector to the corresponding active element of each horizontal slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a horizontal slice is considered active if its corresponding element in the second governing predicate is TRUE and the element corresponding to its horizontal slice number in the first governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

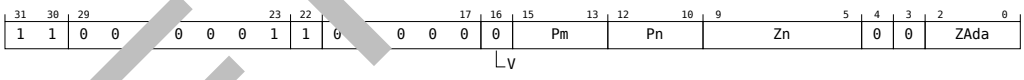
32-bit
(FEAT_SME)




```
ADDHA   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S
```

```
1  if !HaveSME() then UNDEFINED;
2  constant integer esize = 32;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer da = UInt(ZAda);
```

64-bit
(FEAT_SME_I16I64)




```
ADDHA   <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D
```

```
1  if !HaveSME() then UNDEFINED;
2  constant integer esize = 64;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer da = UInt(ZAda);
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand_src = Z[n, VL];
bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for col = 0 to dim-1
    bits(esize) element = Elem[operand_src, col, esize];
    for row = 0 to dim-1
        bits(esize) res = Elem[operand_acc, row*dim+col, esize];
        if (ActivePredicateElement(mask1, row, esize) &&
              ActivePredicateElement(mask2, col, esize)) then
            res = res + element;
        Elem[result, row*dim+col, esize] = res;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

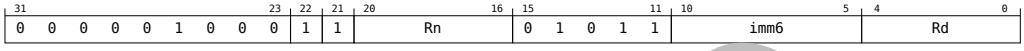
D1.1.6 ADDSPL

Add multiple of Streaming SVE predicate register size to scalar register

Add the Streaming SVE predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME
(FEAT_SME)




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

```
1  if !HaveSME() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer d = UInt(Rd);
4  integer imm = SInt(imm6);
```

Assembler Symbols

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

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

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
1  CheckSMEEnabled();
2  constant integer SVL = CurrentSVL;
3  integer len = imm * SVL DIV 64;
4  bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
5  bits(64) result = operand1 + len;
6
7  if d == 31 then
8      SP[] = result;
9  else
10     X[d, 64] = result;
```

Operational information

If PSTATE.DIT is 1:

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

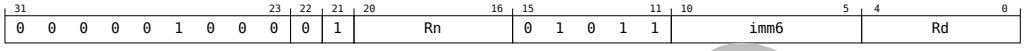
D1.1.7 ADDSVL

Add multiple of Streaming SVE vector register size to scalar register

Add the Streaming SVE vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.

This instruction does not require the PE to be in Streaming SVE mode.

SME
(FEAT_SME)




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

```
1  if !HaveSME() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer d = UInt(Rd);
4  integer imm = SInt(imm6);
```

Assembler Symbols

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

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

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
1  CheckSMEEnabled();
2  constant integer SVL = CurrentSVL;
3  integer len = imm * SVL DIV 8;
4  bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
5  bits(64) result = operand1 + len;
6
7  if d == 31 then
8      SP[] = result;
9  else
10     X[d, 64] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.8 ADDVA

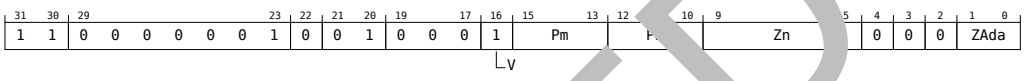
Add vertically vector elements to ZA tile

Add each element of the source vector to the corresponding active element of each vertical slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a vertical slice is considered active if its corresponding element in the first governing predicate is TRUE and the element corresponding to its vertical slice number in the second governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

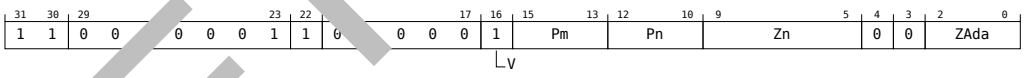
32-bit
(FEAT_SME)




```
ADDVA   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S
```

```
1  if !HaveSME() then UNDEFINED;
2  constant integer esize = 32;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer da = UInt(ZAda);
```

64-bit
(FEAT_SME_I16I64)




```
ADDVA   <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D
```

```
1  if !HaveSME_I16I64() then UNDEFINED;
2  constant integer esize = 64;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer da = UInt(ZAda);
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand_src = Z[n, VL];
bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    bits(esize) element = Elem[operand_src, row, esize];
    for col = 0 to dim-1
        bits(esize) res = Elem[operand_acc, row*dim+col, esize];
        if (ActivePredicateElement(mask1, row, esize) &&
              ActivePredicateElement(mask2, col, esize)) then
            res = res + element;
        Elem[result, row*dim+col, esize] = res;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

D1.1.9 BFCVT

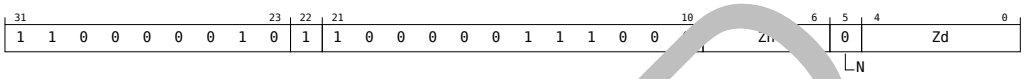
Multi-vector floating-point convert from single-precision to packed BFloat16 format

Convert to BFloat16 from single-precision, each element of the two source vectors, and place the results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
BFCVT   <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
bits(VL) result;

bits(VL) operand1 = Z[n+0, VL];
bits(VL) operand2 = Z[n+1, VL];
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, e, 32];
    bits(32) element2 = Elem[operand2, e, 32];
    bits(16) res1 = FPConvertBF(element1, FPCR[]);
    bits(16) res2 = FPConvertBF(element2, FPCR[]);
    Elem[result, e, 16] = res1;
    Elem[result, elements+e, 16] = res2;

Z[d, VL] = result;
```

D1.1.10 BFCVTN

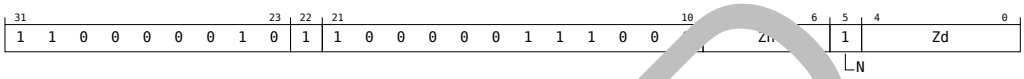
Multi-vector floating-point convert from single-precision to interleaved BFloat16 format

Convert to BFloat16 from single-precision, each element of the two source vectors, and place the two-way interleaved results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
BFCVTN  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
bits(VL) result;

bits(VL) operand1 = Z[n+0, VL];
bits(VL) operand2 = Z[n+1, VL];
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, e, 32];
    bits(32) element2 = Elem[operand2, e, 32];
    bits(16) res1 = FPConvertBF(element1, FPCR[]);
    bits(16) res2 = FPConvertBF(element2, FPCR[]);
    Elem[result, 2*e + 0, 16] = res1;
    Elem[result, 2*e + 1, 16] = res2;

Z[d, VL] = result;
```

D1.1.11 BFDOT (multiple and indexed vector)

Multi-vector BFloat16 floating-point dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups.

The BF16 pairs within the second source vector are specified using an immediate index which selects the same BF16 pair position within each 128-bit vector segment. The element index range is from 0 to 3. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter th[illegible]ber of ZA array vectors.

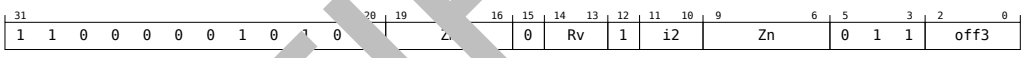
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ope[illegible]d consists o[illegible]vo or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for dis[illegible]embly, but o[illegible]onal in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 num[illegible]ical behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-ve[illegible]and F[illegible] ZA sin[illegible]e-vectors

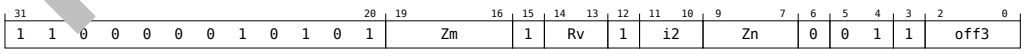
Two ZA single-vectors
(FEAT_SME2)




```
BFDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

Four [illegible]ingle-vectors
(FEAT_[illegible]E2)




```
BFDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MO vstrid
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL
    bits(VL) operand2 = Z[m, V
    bits(VL) operand3 = ZAv tor[vec, V
    for e = 0 to elements
        bits(16) elt1_a  El [operand1, 2  e + 0, 16];
        bits(16) elt1_b = Elem erand1, 2 * e + 1, 16];
        integer seg ntbase = e  e MOD eltspersegment);
        integer s  segmentbase + dex;
        bits(16 elt2_a  Elem[operand2, 2 * s + 0, 16];
        bits( elt2_ = Elem[operand2, 2 * s + 1, 16];
        bits(32 sum  Elem[operand3, e, 32];
        sum = BFD  d(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        esult , 32  = sum;
    Z ector[ c, VL]  sult;
    ec = vec + vstride
```

D1.1.12 BFDOT (multiple and single vector)

Multi-vector BFloat16 floating-point dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

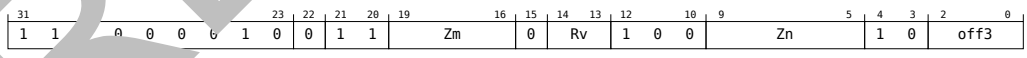
It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors
(FEAT_SME2)


```
BFDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
BFDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn"

plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
        bits(32) sum = Elem[operand3, e, 32];
        sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, e, 32] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.13 BFDOT (multiple vectors)

Multi-vector BFloat16 floating-point dot-product

The instruction operates on two or four ZA single-vector groups.

The instruction computes the dot product of a pair of BF16 values held in the corresponding 32-bit elements of the two or four first and second source vectors. The single-precision dot product results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

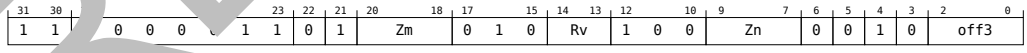
It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors
(FEAT_SME2)


```
BFDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
BFDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a

multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
        bits(32) sum = Elem[operand3, e, 32];
        sum = BFDOT32(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, e, 32] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.14 BFMLAL (multiple and indexed vector)

Multi-vector BFloat16 floating-point multiply-add long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups.

The BF16 elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

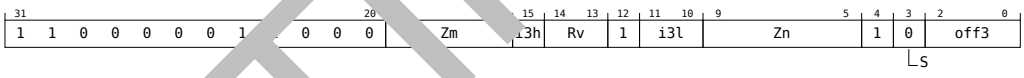
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the Zm operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behavior.

This instruction is unpredicated.

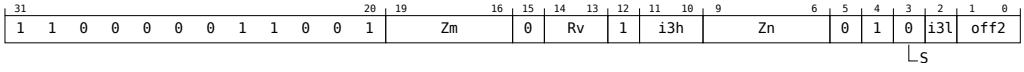
It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors and Four ZA double-vectors

One ZA double-vector
(FEAT_SME2)



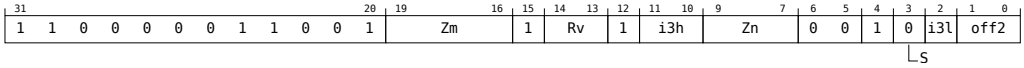

```
BFMLAL  ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 1;
```

Two ZA double-vectors
(FEAT_SME2)




```
BFMLAL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)




```
BFMLAL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, s, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = BFNeg(element1);
            Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.15 BFMLAL (multiple and single vector)

Multi-vector BFloat16 floating-point multiply-add long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

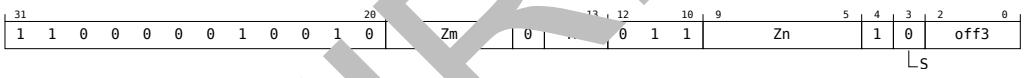
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA [illegible]d consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is [illegible]referred [illegible] disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behav[illegible]rs.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector , [illegible] ZA [illegible]uble-vectors and Four ZA double-vectors

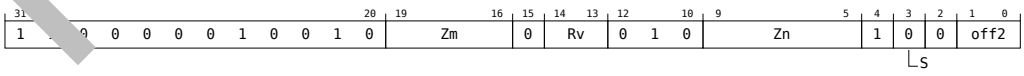
One ZA double-vector
(FEAT_SME2)




```
BFMLAL  ZA.S[<Wv>, <off[illegible]>[illegible]offsl>], <Z[illegible].H, <Zm>.H
```

```
if !HaveSME2() then [illegible]NDEFINED;
integer v = UInt([illegible]10':Rv);
integer n = UIn[illegible]Zn);
integer m = U[illegible]('0':Z[illegible];
integer offset [illegible]UIn[illegible]off3:'0'[illegible];
boolean sub[illegible]op = [illegible]SE;
constan[illegible]nte[illegible]er nr[illegible] = 1;
```

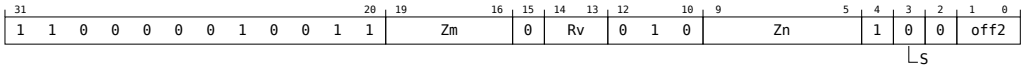
Tw[illegible]A doub[illegible]vectors
(FEA[illegible]SM[illegible]2)




```
BFMLAL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)




```
BFMLAL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = BFNeg(element1);
            Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.16 BFMLAL (multiple vectors)

Multi-vector BFloat16 floating-point multiply-add long

The instruction operates on two or four ZA double-vector groups.

This BFloat16 floating-point multiply-add long instruction widens all 16-bit BFloat16 elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

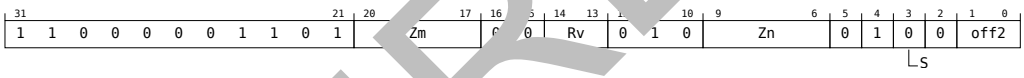
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors a[illegible] Four Z[illegible] double-[illegible]ors

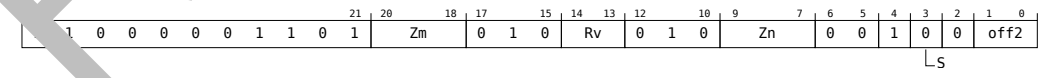
Two ZA double-vectors
(FEAT_SME2)




```
BFMLAL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('01':Rv);
3  integer n = UInt(Zn:'0');
4  integer m = UInt(Zm:'0');
5  integer offset = UInt(off2:'0');
6  boolean sub_op = FALSE;
7  constant integer nreg = 2;
```

Four [illegible]A doub[illegible]-vectors
(FE[illegible]T_SME[illegible]




```
BFMLAL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  integer n = UInt(Zn:'00');
4  integer m = UInt(Zm:'00');
5  integer offset = UInt(off2:'0');
6  boolean sub_op = FALSE;
7  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field

times 2.

<offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[n, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(vec+r) MOD 32, VL];
    bits(VL) operand2 = Z[(vec+r+m) MOD 32, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = BFNeg(element1);
            Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.17 BFMLSL (multiple and indexed vector)

Multi-vector BFloat16 floating-point multiply-subtract long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups.

The BF16 elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

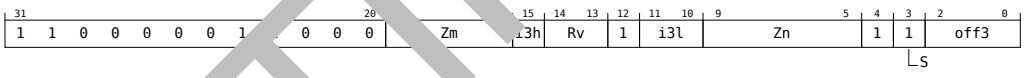
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the Zm operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behavior.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors and Four ZA double-vectors

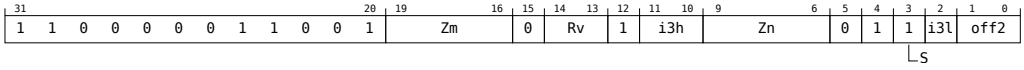
One ZA double-vector
(FEAT_SME2)




```
BFMLSL  ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 1;
```

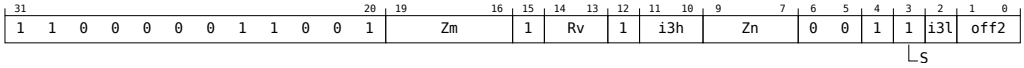
Two ZA double-vectors
(FEAT_SME2)




```
BFMLSL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)




```
BFMLSL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, s, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = BFNeg(element1);
            Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.18 BFMLSL (multiple and single vector)

Multi-vector BFloat16 floating-point multiply-subtract long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

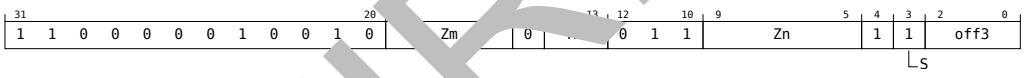
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ... consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is ...eferred ... disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behav...rs.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector , ... ZA ...uble-vectors and Four ZA double-vectors

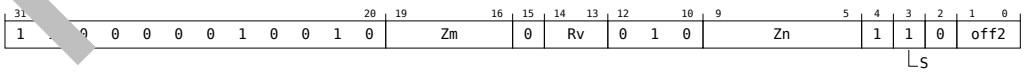
One ZA double-vector
(FEAT_SME2)




```
BFMLSL  ZA.S[<Wv>, <off...>...offsl>], <Z...>.H, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  integer n = UInt(Zn);
4  integer m = UInt('0':Zm);
5  integer offset = UInt(off3:'0');
6  boolean sub_op = TRUE;
7  constant integer nreg = 1;
```

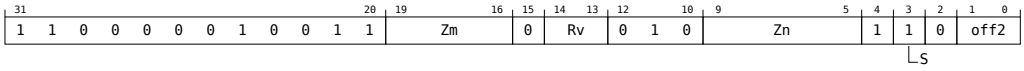
Tw... ZA doub... ...ectors
(FEA... SM...2)




```
BFMLSL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  integer n = UInt(Zn);
4  integer m = UInt('0':Zm);
5  integer offset = UInt(off2:'0');
6  boolean sub_op = TRUE;
7  constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)




```
BFMLSL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = BFNeg(element1);
            Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.19 BFMLSL (multiple vectors)

Multi-vector BFloat16 floating-point multiply-subtract long

The instruction operates on two or four ZA double-vector groups.

This BFloat16 floating-point multiply-subtract long instruction widens all 16-bit BFloat16 elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

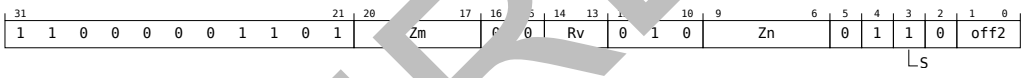
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors a[illegible] Four Z[illegible] double-[illegible]ors

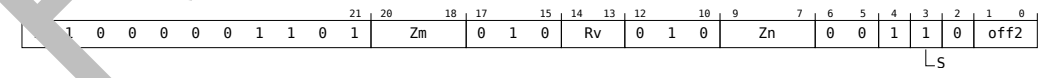
Two ZA double-vectors
(FEAT_SME2)




```
BFMLSL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('01':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four [illegible]A doub[illegible]-vectors
(FE[illegible]T_SME[illegible]




```
BFMLSL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field

times 2.

<offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[n, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(vec+r) MOD 32, VL];
    bits(VL) operand2 = Z[(vec+r+m) MOD 32, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = BFNeg(element1);
            Elem[result, e, 32] = BFMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.20 BFMOPA

BFloat16 sum of outer products and accumulate

The BFloat16 floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile.

This instruction multiplies the $SVL_S \times 2$ sub-matrix of BFloat16 values held in the first source vector by the $2 \times SVL_S$ sub-matrix of BFloat16 values in the second source vector.

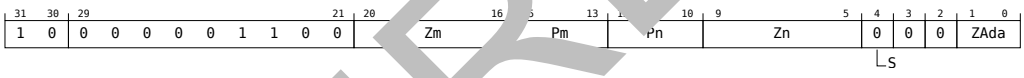
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME BFloat16 numerical behaviour.

SME
(FEAT_SME)




```
BFMOPA  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;
```

```
11
12  for row = 0 to dim-1
13      for col = 0 to dim-1
14          // determine row/col predicates
15          boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
16          boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
17          boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
18          boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));
19
20          bits(32) sum = Elem[operand3, row*dim+col, 32];
21          if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
22              bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0',
                    ↪16));
23              bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0',
                    ↪16));
24              bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0',
                    ↪16));
25              bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0',
                    ↪16));
26              if sub_op then
27                  boolean honor_altfp = FALSE;   // Alternate handling ignored
28                  if prow_0 then erow_0 = BFNeg(erow_0, honor_altfp);
29                  if prow_1 then erow_1 = BFNeg(erow_1, honor_altfp);
30              sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);
31
32          Elem[result, row*dim+col, 32] = sum;
33
34  ZAtile[da, 32, dim*dim*32] = result;
```

D1.1.21 BFMOPS

BFloat16 sum of outer products and subtract

The BFloat16 floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile.

This instruction multiplies the $SVL_S \times 2$ sub-matrix of BFloat16 values held in the first source vector by the $2 \times SVL_S$ sub-matrix of BFloat16 values in the second source vector.

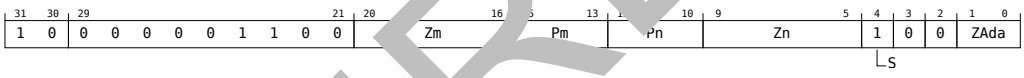
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME BFloat16 numerical behaviour.

SME
(FEAT_SME)




```
BFMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
1  if !HaveSME() then UNDEFINED;
2  integer a = UInt(Pn);
3  integer b = UInt(Pm);
4  integer n = UInt(Zn);
5  integer m = UInt(Zm);
6  integer da = UInt(ZAda);
7  boolean sub_op = TRUE;
```

Assembler Symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer dim = VL DIV 32;
5  bits(PL) mask1 = P[a, PL];
6  bits(PL) mask2 = P[b, PL];
7  bits(VL) operand1 = Z[n, VL];
8  bits(VL) operand2 = Z[m, VL];
9  bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
10 bits(dim*dim*32) result;
```

```
11
12  for row = 0 to dim-1
13      for col = 0 to dim-1
14          // determine row/col predicates
15          boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
16          boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
17          boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
18          boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));
19
20          bits(32) sum = Elem[operand3, row*dim+col, 32];
21          if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
22              bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0',
                    ↪16));
23              bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0',
                    ↪16));
24              bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0',
                    ↪16));
25              bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0',
                    ↪16));
26              if sub_op then
27                  boolean honor_altfp = FALSE;   // Alternate handling ignored
28                  if prow_0 then erow_0 = BFNeg(erow_0, honor_altfp);
29                  if prow_1 then erow_1 = BFNeg(erow_1, honor_altfp);
30              sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);
31
32          Elem[result, row*dim+col, 32] = sum;
33
34  ZAtile[da, 32, dim*dim*32] = result;
```

D1.1.22 BFVDOT

Multi-vector BFloat16 floating-point vertical dot-product by indexed element

The instruction operates on two ZA single-vector groups.

The instruction computes the vertical dot product of the corresponding BF16 elements held in the two first source vectors with pair of BF16 values held in the indexed 32-bit element of the second source vector. The single-precision dot product results are destructively added to the corresponding single-precision elements of the two ZA single-vector groups.

The BF16 pairs within the second source vector are specified using an immediate index which selects the same BF16 pair position within each 128-bit vector segment. The element index range is from 0 to 3.

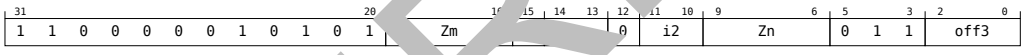
The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand consists of two ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME2 ZA-targeting BFloat16 numerical behaviors.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
BFVDOT  ZA.S[<Wv>, <offs>, VGx2}], { Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('01':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 2;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
```

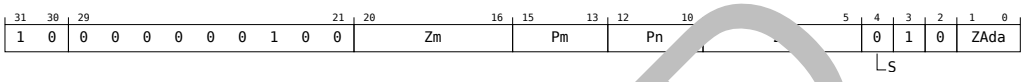
```
bits(VL) result;

for r = 0 to 1
    bits(VL) operand1a = Z[n, VL];
    bits(VL) operand1b = Z[n+1, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(16) elt1_a = Elem[operand1a, 2 * e + r, 16];
        bits(16) elt1_b = Elem[operand1b, 2 * e + r, 16];
        bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
        bits(32) sum = Elem[operand3, e, 32];
        sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, e, 32] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.23 BMOPA

Bitwise exclusive NOR population count outer product and accumulate

This instruction works with 32-bit element ZA tile. This instruction generates an outer product of the first source $SVL_S \times 1$ vector and the second source $1 \times SVL_S$ vector. Each outer product element is obtained as population count of the bitwise XNOR result of the corresponding 32-bit elements of the first source vector and the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is inactive the corresponding destination tile element remains unmodified. The resulting $SVL_S \times SVL_S$ product is then destructively added to the destination tile.

SME2
(FEAT_SME2)




```
BMOPA   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

<ZAda> Is the name of the Z[illegible]ile [illegible]0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of t[illegible]first governing [illegible]calable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name [illegible] the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name o[illegible]e first sou[illegible]e scalable vector register, encoded in the "Zn" field.

<Zm> [illegible] the na[illegible]e of the [illegible]d source scalable vector register, encoded in the "Zm" field.

Oper[illegible]ion

```
CheckSti[illegible]mingSVEAndZAEnabled();
constant [illegible]eger VL = CurrentVL;
constant int[illegible]er PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    bits(esize) element1 = Elem[operand1, row, esize];
    for col = 0 to dim-1
        bits(esize) element2 = Elem[operand2, col, esize];
        bits(esize) element3 = Elem[operand3, row*dim + col, esize];
        if (ActivePredicateElement(mask1, row, esize) &&
              ActivePredicateElement(mask2, col, esize)) then
            integer res = BitCount(NOT(element1 EOR element2));
            if sub_op then res = -res;
            Elem[result, row*dim + col, esize] = element3 + res;
```

```
        else
            Elem[result, row*dim + col, esize] = element3;
ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

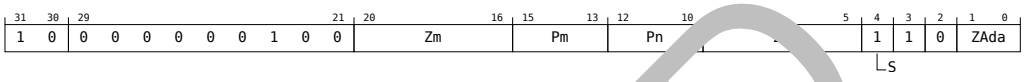
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

D1.1.24 BMOPS

Bitwise exclusive NOR population count outer product and subtract

This instruction works with 32-bit element ZA tile. This instruction generates an outer product of the first source $SVL_S \times 1$ vector and the second source $1 \times SVL_S$ vector. Each outer product element is obtained as population count of the bitwise XNOR result of the corresponding 32-bit elements of the first source vector and the second source vector. Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is inactive the corresponding destination tile element remains unmodified. The resulting $SVL_S \times SVL_S$ product is then destructively subtracted from the destination tile.

SME2
(FEAT_SME2)




```
BMOPS   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

<ZAda> Is the name of the Z tile 0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    bits(esize) element1 = Elem[operand1, row, esize];
    for col = 0 to dim-1
        bits(esize) element2 = Elem[operand2, col, esize];
        bits(esize) element3 = Elem[operand3, row*dim + col, esize];
        if (ActivePredicateElement(mask1, row, esize) &&
              ActivePredicateElement(mask2, col, esize)) then
            integer res = BitCount(NOT(element1 EOR element2));
            if sub_op then res = -res;
            Elem[result, row*dim + col, esize] = element3 + res;
```

```
        else
            Elem[result, row*dim + col, esize] = element3;
ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

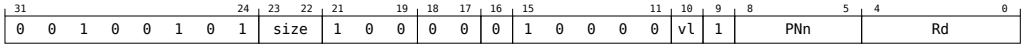
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

D1.1.25 CNTP

Set scalar to count from predicate-as-counter

Counts the number of true elements in the source predicate and places the scalar result in the destination general-purpose register.

SME2
(FEAT_SME2)




```
CNTP    <Xd>, <PNn>.<T>, <vl>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(PNn);
integer d = UInt(Rd);
constant integer width = 2 << UInt(vl);
```

Assembler Symbols

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

<PNn> Is the name of the first source scalable predicate register, with predicate-as-counter encoding, encoded in the "PNn" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) pred = P[n, PL];
bits(PL*4) mask = CounterToPredicate(pred<15:0>, PL*4);
bits(64) sum = Zeros(64);
constant integer limit = elements * width;

for e = 0 to limit-1
    if ActivePredicateElement(mask, e, esize) then
        sum = sum + 1;
X[d, 64] = sum;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.26 FADD

Floating-point add multi-vector to ZA array vector accumulators

The instruction operates on two or four ZA single-vector groups.

Destructively add all elements of the two or four source vectors to the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

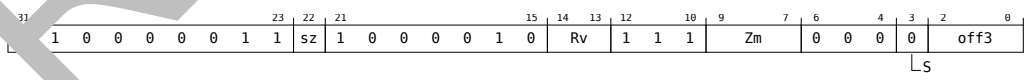
It has encodings from 2 classes: Two ZA single-vectors and four ZA single-vectors

Two ZA single-vectors
(FEAT_SME2)


```
FADD    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four single-vectors
(FEAT_SME2)




```
FADD    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = ZAvector[vec, VL];
    bits(VL) operand2 = Zm[r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAdd_ZA(element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.27 FCLAMP

Multi-vector floating-point clamp to minimum/maximum number

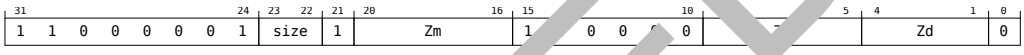
Clamp each floating-point element in the two or four destination vectors to between the floating-point minimum value in the corresponding element of the first source vector and the floating-point maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors. If at least one element value contributing to a result is numeric and the other is either numeric or a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

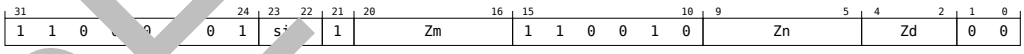
Two registers
(FEAT_SME2)




```
FCLAMP  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer n = UInt(Zn);
5  integer m = UInt(Zm);
6  integer d = UInt(Zd:'0');
7  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FCLAMP  { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer n = UInt(Zn);
5  integer m = UInt(Zm);
6  integer d = UInt(Zd:'00');
7  constant integer nreg = 4;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = Z[d+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        Elem[results[r], e, esize] = FPMinNum(FPMaxNum(element1, element3, FPCR[]),
            ↪element2, FPCR[]);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.28 FCVT

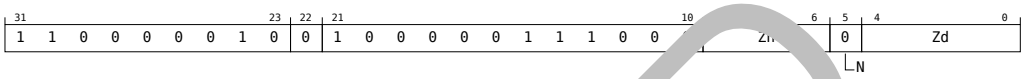
Multi-vector floating-point convert from single-precision to packed half-precision

Convert to half-precision from single-precision, each element of the two source vectors, and place the results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
FCVT    <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
bits(VL) result;

bits(VL) operand1 = Z[n+0, VL];
bits(VL) operand2 = Z[n+1, VL];
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, e, 32];
    bits(32) element2 = Elem[operand2, e, 32];
    bits(16) res1 = FPConvertSVE(element1, FPCR[], 16);
    bits(16) res2 = FPConvertSVE(element2, FPCR[], 16);
    Elem[result, e, 16] = res1;
    Elem[result, elements+e, 16] = res2;

Z[d, VL] = result;
```

D1.1.29 FCVTN

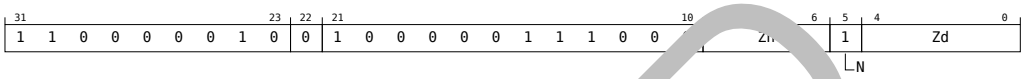
Multi-vector floating-point convert from single-precision to interleaved half-precision

Convert to half-precision from single-precision, each element of the two source vectors, and place the two-way interleaved results in the half-width destination elements.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
FCVTN   <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
bits(VL) result;

bits(VL) operand1 = Z[n+0, VL];
bits(VL) operand2 = Z[n+1, VL];
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, e, 32];
    bits(32) element2 = Elem[operand2, e, 32];
    bits(16) res1 = FPConvertSVE(element1, FPCR[], 16);
    bits(16) res2 = FPConvertSVE(element2, FPCR[], 16);
    Elem[result, 2*e + 0, 16] = res1;
    Elem[result, 2*e + 1, 16] = res2;

Z[d, VL] = result;
```

D1.1.30 FCVTZS

Multi-vector floating-point convert to signed integer, rounding toward zero

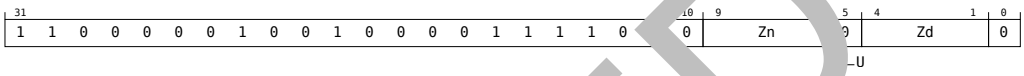
Convert to the signed 32-bit integer nearer to zero from single-precision, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

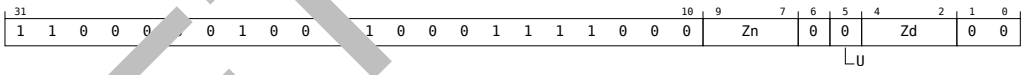
Two registers
(FEAT_SME2)




```
FCVTZS  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Four registers
(FEAT_SME2)




```
FCVTZS  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPToFixed(element, , unsig d, FPCR[ , rounding, 32);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.31 FCVTZU

Multi-vector floating-point convert to unsigned integer, rounding toward zero

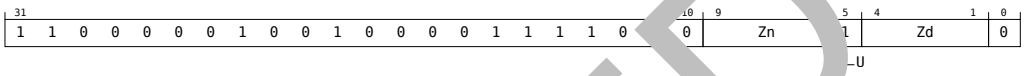
Convert to the unsigned 32-bit integer nearer to zero from single-precision, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

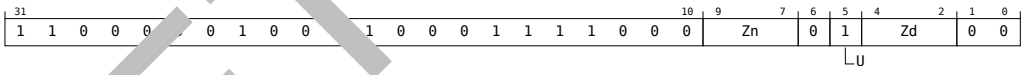
Two registers
(FEAT_SME2)




```
FCVTZU  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

Four registers
(FEAT_SME2)




```
FCVTZU  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPToFixed(element, , unsig d, FPCR[ , rounding, 32);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.32 FDOT (multiple and indexed vector)

Multi-vector half-precision floating-point dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first source vectors and the indexed 32-bit element of the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The element index range is from 0 to 3. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

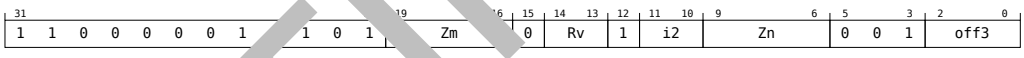
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behavior.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

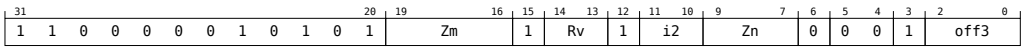
Two ZA single-vectors
(FEAT_SME2)




```
FDOT    ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
FDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z[illegible], encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offs[illegible]) MOD [illegible]ride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = [illegible] VL];
    bits(VL) operand2 = Z[m, [illegible]
    bits(VL) operand[illegible] = ZAvector[illegible]ec, VL];
    for e = 0 to e[illegible]ments-1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        integer [illegible]egme[illegible]base = e - (e MOD eltspersegment);
        integer [illegible]egmentb[illegible] + index;
        bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
        bits(32) sum = Elem[operand3, e, 32];
        sum = FPTwoWayDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, e, 32] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.33 FDOT (multiple and single vector)

Multi-vector half-precision floating-point dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first source vectors and the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

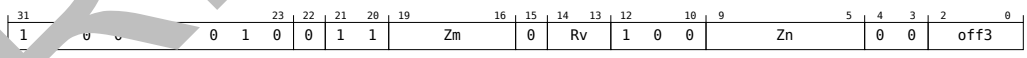
It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors
(FEAT_SME2)


```
FDOT    ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  integer n = UInt(Zn);
4  integer m = UInt('0':Zm);
5  integer offset = UInt(off3);
6  constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
FDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  integer n = UInt(Zn);
4  integer m = UInt('0':Zm);
5  integer offset = UInt(off3);
6  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn"

plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
        bits(32) sum = Elem[operand3, e, 32];
        sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, e, 32] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.34 FDOT (multiple vectors)

Multi-vector half-precision floating-point dot-product

The instruction operates on two or four ZA single-vector groups.

The instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in the corresponding 32-bit elements of the two or four first and second source vectors, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

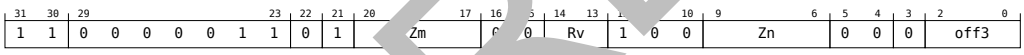
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

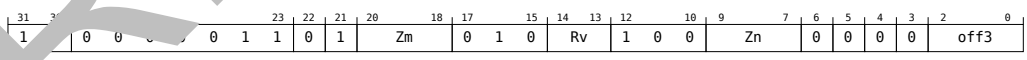
Two ZA single-vectors
(FEAT_SME2)




```
FDOT    ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
FDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
        bits(32)      = Elem[operand3, e, 32];
        FPDotAdd_ZA(, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, , ] = sum;
ZAvector[vec, VL] = result;
vec = vec + vstride;
```

D1.1.35 FMAX (multiple and single vector)

Multi-vector floating-point maximum by vector

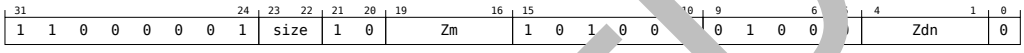
Determine the maximum of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If either element value is NaN then the result is NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

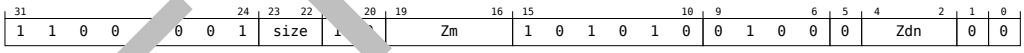
Two registers
(FEAT_SME2)




```
FMAX    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'0');
5  integer m = UInt('0':Zm);
6  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FMAX    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'00');
5  integer m = UInt('0':Zm);
6  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMax(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.36 FMAX (multiple vectors)

Multi-vector floating-point maximum

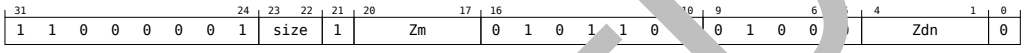
Determine the maximum of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If either element value is NaN then the result is NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

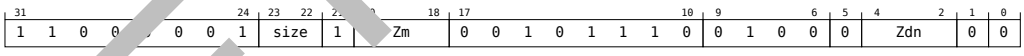
Two registers
(FEAT_SME2)




```
FMAX    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'0');
5  integer m = UInt(Zm:'0');
6  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FMAX    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'00');
5  integer m = UInt(Zm:'00');
6  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMax(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.37 FMAXNM (multiple and single vector)

Multi-vector floating-point maximum number by vector

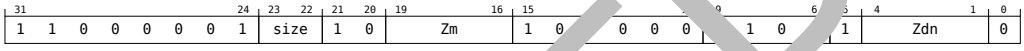
Determine the maximum number value of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

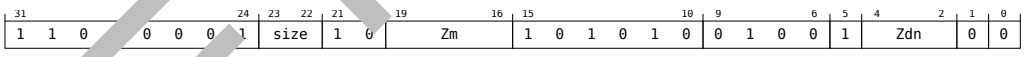
Two registers
(FEAT_SME2)




```
FMAXNM  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Z  .<T>  Zdn2>.<  > }, <Zm>.<T>
```

```
1 if !HaveSME2() then UNDEFINED;
2 if size == '00' then UNDEFINED;
3 constant integer esize = 8 << UInt(size
4 integer dn = UInt(Zdn:'0');
5 integer m = UInt('0':Zm);
6 constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FMAXNM  { <Z  1>.<T>  <Z  4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1 if  veSME  UNDEFINED;
2 if si  '00' then UNDEFINED;
3 constan nteger esize = 8 << UInt(size);
4 integer d  UInt(Zdn:'00');
5 integer m =  nt('0':Zm);
6 constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMaxNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.38 FMAXNM (multiple vectors)

Multi-vector floating-point maximum number

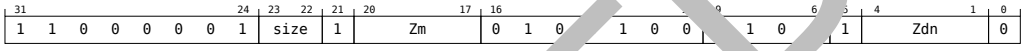
Determine the maximum number value of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

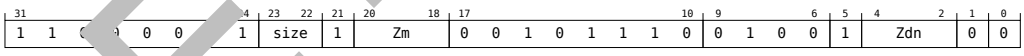
Two registers
(FEAT_SME2)




```
FMAXNM  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Z  .<T>  Zdn2>.<  > }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UIn  size),
4  integer dn = UInt(Zdn:'0');
5  integer m = UInt(Zm:'0');
6  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FMAX  { <Z  1>.<T>-  dn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1  if !Have  E2() then UNDEFINED;
2  if size ==  00' then UNDEFINED;
3  constant int  er esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'00');
5  integer m = UInt(Zm:'00');
6  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMaxNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.39 FMIN (multiple and single vector)

Multi-vector floating-point minimum by vector

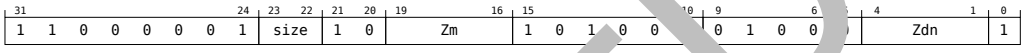
Determine the mininum of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If either element value is NaN then the result is NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

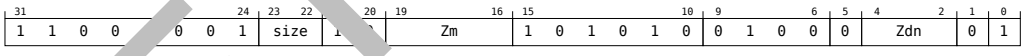
Two registers
(FEAT_SME2)




```
FMIN    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'0');
5  integer m = UInt('0':Zm);
6  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FMIN    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'00');
5  integer m = UInt('0':Zm);
6  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMin(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.40 FMIN (multiple vectors)

Multi-vector floating-point minimum

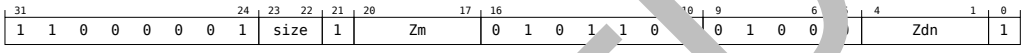
Determine the mininum of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If either element value is NaN then the result is NaN.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

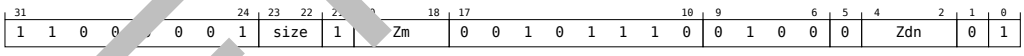
Two registers
(FEAT_SME2)




```
FMIN    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'0');
5  integer m = UInt(Zm:'0');
6  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FMIN    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'00');
5  integer m = UInt(Zm:'00');
6  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMin(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.41 FMINNM (multiple and single vector)

Multi-vector floating-point minimum number by vector

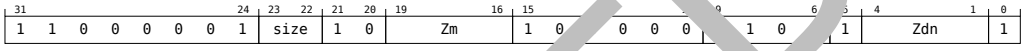
Determine the minimum number value of floating-point elements of the second source vector and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

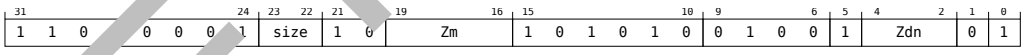
Two registers
(FEAT_SME2)




```
FMINNM  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Z  .<T>  Zdn2>.<  > }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size
4  integer dn = UInt(Zdn:'0');
5  integer m = UInt('0':Zm);
6  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FMINNM  { <Z  1>.<T>  <Z  4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if  veSME  UNDEFINED;
2  if si  '00' then UNDEFINED;
3  constan nteger esize = 8 << UInt(size);
4  integer d  UInt(Zdn:'00');
5  integer m =  nt('0':Zm);
6  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMinNum(element1, element2, FPCR[]);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.42 FMINNM (multiple vectors)

Multi-vector floating-point minimum number

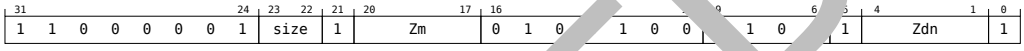
Determine the minimum number value of floating-point elements of the two or four second source vectors and the corresponding floating-point elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

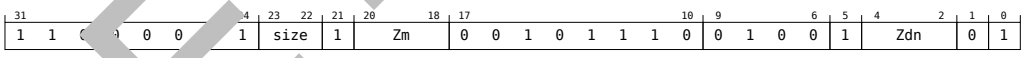
Two registers
(FEAT_SME2)




```
FMINNM  { <Zdn1>.<T>-<Zdn2>.<T> }, { <Z  .<T>  Zdn2>.<  > }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UIn  size),
4  integer dn = UInt(Zdn:'0');
5  integer m = UInt(Zm:'0');
6  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
FMIN   { <Z  1>.<T>-  dn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1  if !Have  E2() then UNDEFINED;
2  if size ==  00' then UNDEFINED;
3  constant int  er esize = 8 << UInt(size);
4  integer dn = UInt(Zdn:'00');
5  integer m = UInt(Zm:'00');
6  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[results[r], e, esize] = FPMinNum(element1, element2, FPCR[]);

for r = 0 to nreg
    Z[dn+r, VL] = results[r];
```

D1.1.43 FMLA (multiple and indexed vector)

Multi-vector floating-point fused multiply-add by indexed element

The instruction operates on two or four ZA single-vector groups.

Multiply the indexed element of the second source vector by the corresponding floating-point elements of the two or four first source vectors and destructively add without intermediate rounding to the corresponding elements of the two or four ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 2 bits depending on the size of the element. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the num ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ope d consists o o or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for dis embly, but o onal in assembler source code.

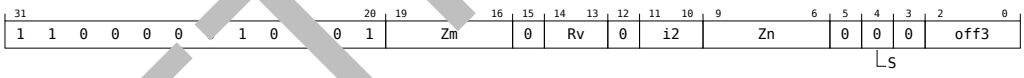
This instruction follows SME ZA-targeting floating-point b aviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether uble-p cision v iant is implemented.

It has encodings from 4 classes: Two ZA sin -vectors f single p sion elements , Two ZA single-vectors of double precision elements , Four ZA single ctors o precision elements and Four ZA single-vectors of double precision elements

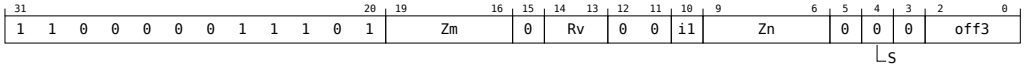
Two ZA single-vectors of single p cisio lemen (FEAT_SME2)




```
FMLA     ZA.S[<W   ffs>{, VGx2}], { <Zn1>.S-<Zn2>.S }, <Zm>.S[<index>]
```

```
1  if !Ha SME2 then DEF ED;
2  inte v = U t('010 );
3  co ant int er esize = 32;
4  inte n UInt( 0');
5  intege = UInt('0':Zm);
6  integer set = UInt(off3);
7  integer in = UInt(i2);
8  boolean sub_op = FALSE;
9  constant integer nreg = 2;
```

Two ZA single-vectors of double precision elements (FEAT_SME_F64F64)

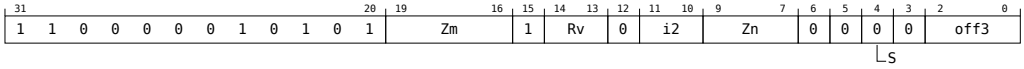



```
FMLA     ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }, <Zm>.D[<index>]
```

```
1  if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
2  integer v = UInt('010':Rv);
3  constant integer esize = 64;
```

```
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

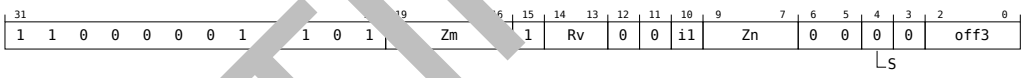
Four ZA single-vectors of single precision elements (FEAT_SME2)




```
FMLA    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.S-<Zn4>.S }, <Zm>.S[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Four ZA single-vectors of double precision elements (FEAT_SME_F64F64)




```
FMLA    ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }, <Zm>.D[<index>]
```

```
if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
integer v = UInt('01':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors of double precision elements and two ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors of double precision elements and four ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as

"Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the four ZA single-vectors of single precision elements and two ZA single-vectors of single precision elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the four ZA single-vectors of double precision elements and two ZA single-vectors of double precision elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        integer segmentbase = e DIV eltspersegment);
        integer s = segmentbase + index;
        bits(esize) element2 = Elem[operand2, s, esize];
        bits(esize) element3 = Elem[operand3, , esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.44 FMLA (multiple and single vector)

Multi-vector floating-point fused multiply-add by vector

The instruction operates on two or four ZA single-vector groups.

Multiply the corresponding floating-point elements of the two or four first source vector with corresponding elements of the second source vector and destructively add without intermediate rounding to the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

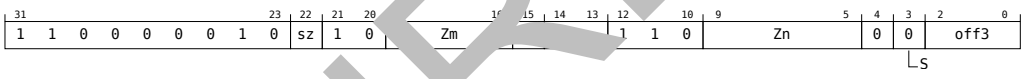
This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

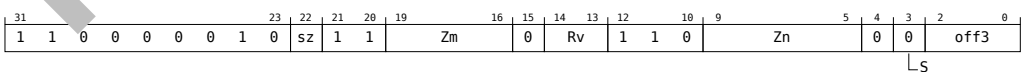
Two ZA single-vectors
(FEAT_SME2)




```
FMLA    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
FMLA    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register, Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offs) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.45 FMLA (multiple vectors)

Multi-vector floating-point fused multiply-add

The instruction operates on two or four ZA single-vector groups.

Multiply the corresponding floating-point elements of the two or four first and second source vectors and destructively add without intermediate rounding to the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

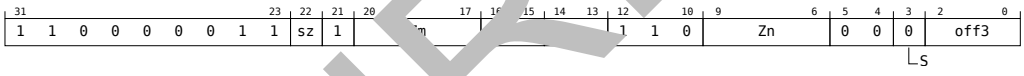
This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

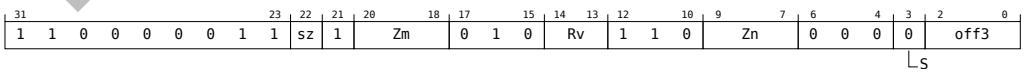
Two ZA single-vectors
(FEAT_SME2)




```
FMLA     ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
FMLA     ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer ss = ( DIV 8);
integer vstride = vectors DIV nreg;
bits(2) vbase = X[v, 2];
integer vec = (vbase) + offset) MOD vstride;
bits( ) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.46 FMLAL (multiple and indexed vector)

Multi-vector floating-point multiply-add long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups.

The half-precision elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

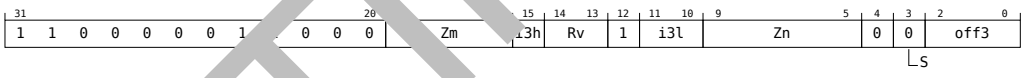
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the Zm operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behavior.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors and Four ZA double-vectors

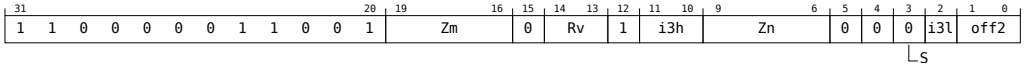
One ZA double-vector
(FEAT_SME2)




```
FMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 1;
```

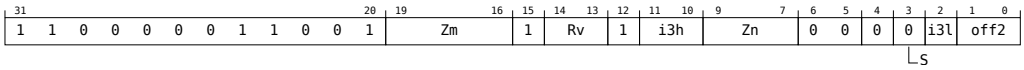
Two ZA double-vectors
(FEAT_SME2)




```
FMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)




```
FMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, s, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = FPNeg(element1);
            Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.47 FMLAL (multiple and single vector)

Multi-vector floating-point multiply-add long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector , Two ZA double-vectors and Four ZA double-vectors

One ZA double-vector
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14-13	12	11	10	9-5	4	3	2-0
1	1	0	0	0	0	0	1	0	0	1	0	Zm	0	Rv	0	1	1	Zn	0	0 (S)	off3

```
FMLAL   ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
boolean sub_op = FALSE;
constant integer nreg = 1;
```

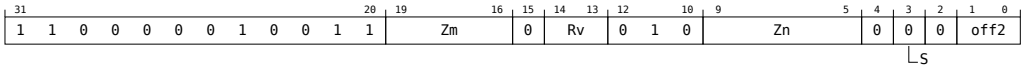
Two ZA double-vectors
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14-13	12	11	10	9-5	4	3	2	1-0
1	1	0	0	0	0	0	1	0	0	1	0	Zm	0	Rv	0	1	0	Zn	0	0 (S)	0	off2

```
FMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)




```
FMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = FPNeg(element1);
            Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.48 FMLAL (multiple vectors)

Multi-vector floating-point multiply-add long

The instruction operates on two or four ZA double-vector groups.

This half-precision floating-point multiply-add long instruction widens all 16-bit half-precision elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

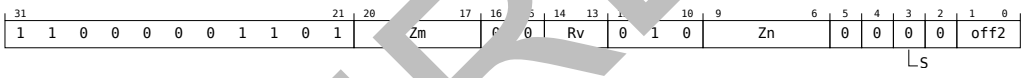
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors a[illegible] Four Z[illegible] double-[illegible]ors

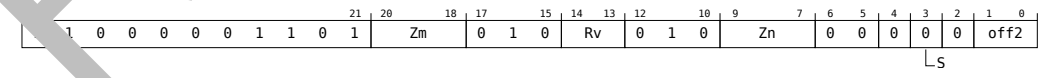
Two ZA double-vectors
(FEAT_SME2)




```
FMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('01':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 2;
```

Four [illegible]A doub[illegible]-vectors
(FE[illegible]T_SME[illegible]




```
FMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
boolean sub_op = FALSE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field

times 2.

<offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[n, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec + (vbase MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[(m+r) MOD 32, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = FPNeg(element1);
            Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.49 FMLS (multiple and indexed vector)

Multi-vector floating-point fused multiply-subtract by indexed element

The instruction operates on two or four ZA single-vector groups.

Multiply the indexed element of the second source vector by the corresponding floating-point elements of the two or four first source vectors and destructively subtract without intermediate rounding from the corresponding elements of the two or four ZA single-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 2 bits depending on the size of the element. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the num ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ope d consists o vo or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for dis embly, but o onal in assembler source code.

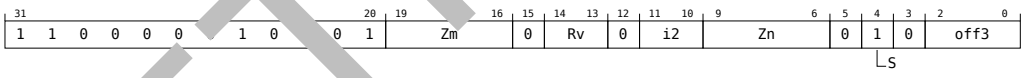
This instruction follows SME ZA-targeting floating-point b aviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether uble-p cision v iant is implemented.

It has encodings from 4 classes: Two ZA sin -vectors f single p sion elements , Two ZA single-vectors of double precision elements , Four ZA single ctors o precision elements and Four ZA single-vectors of double precision elements

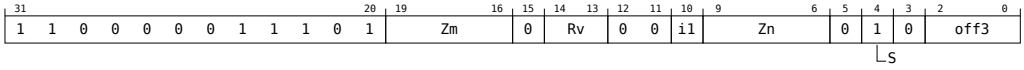
Two ZA single-vectors of single p cisio lemen (FEAT_SME2)




```
FMLS     ZA.S[<W   ffs>{, VGx2}], { <Zn1>.S-<Zn2>.S }, <Zm>.S[<index>]
```

```
1  if !Ha SME2 then DEF ED;
2  inte v = U t('010 );
3  co ant int er esize = 32;
4  inte n UInt( 0');
5  intege = UInt('0':Zm);
6  integer set = UInt(off3);
7  integer in = UInt(i2);
8  boolean sub_op = TRUE;
9  constant integer nreg = 2;
```

Two ZA single-vectors of double precision elements (FEAT_SME_F64F64)

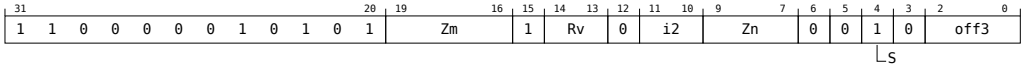



```
FMLS     ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }, <Zm>.D[<index>]
```

```
1  if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
2  integer v = UInt('010':Rv);
3  constant integer esize = 64;
```

```
4  integer n = UInt(Zn:'0');
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off3);
7  integer index = UInt(i1);
8  boolean sub_op = TRUE;
9  constant integer nreg = 2;
```

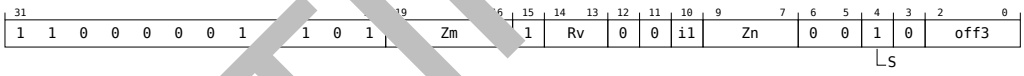
Four ZA single-vectors of single precision elements (FEAT_SME2)




```
FMLS    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.S-<Zn4>.S }, <Zm>.S[<index>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  constant integer esize = 32;
4  integer n = UInt(Zn:'00');
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off3);
7  integer index = UInt(i2);
8  boolean sub_op = TRUE;
9  constant integer nreg = 4;
```

Four ZA single-vectors of double precision elements (FEAT_SME_F64F64)




```
FMLS    ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }, <Zm>.D[<index>]
```

```
1  if !(HaveSME2() && HaveSMEF64F64()) then UNDEFINED;
2  integer v = UInt('01':Rv);
3  constant integer esize = 64;
4  integer n = UInt(Zn:'00');
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off3);
7  integer index = UInt(i1);
8  boolean sub_op = TRUE;
9  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors of double precision elements and two ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors of double precision elements and four ZA single-vectors of single precision elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as

"Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the four ZA single-vectors of single precision elements and two ZA single-vectors of single precision elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the four ZA single-vectors of double precision elements and two ZA single-vectors of double precision elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        integer segmentbase = e DIV eltspersegment);
        integer s = segmentbase + index;
        bits(esize) element2 = Elem[operand2, s, esize];
        bits(esize) element3 = Elem[operand3, , esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.50 FMLS (multiple and single vector)

Multi-vector floating-point fused multiply-subtract by vector

The instruction operates on two or four ZA single-vector groups.

Multiply the corresponding floating-point elements of the two or four first source vector with corresponding elements of the second source vector and destructively subtract without intermediate rounding from the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

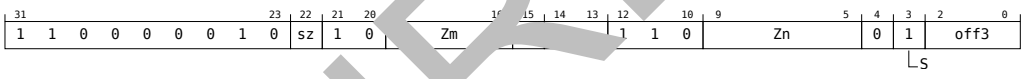
This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

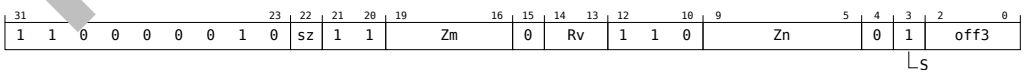
Two ZA single-vectors
(FEAT_SME2)




```
FMLS    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
FMLS    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register, Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offs) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.51 FMLS (multiple vectors)

Multi-vector floating-point fused multiply-subtract

The instruction operates on two or four ZA single-vector groups.

Multiply the corresponding floating-point elements of the two or four first and second source vectors and destructively subtract without intermediate rounding from the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

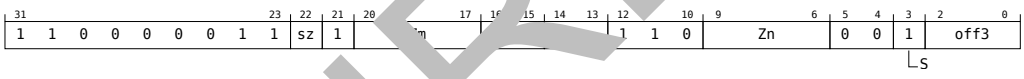
This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

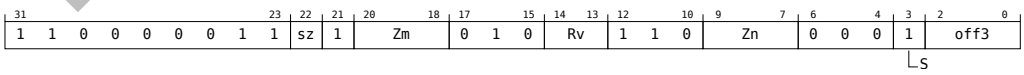
Two ZA single-vectors
(FEAT_SME2)




```
FMLS    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
FMLS    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer ss = ( DIV 8);
integer vstride = vectors DIV nreg;
bits(2) vbase = X[v, 2];
integer vec = (vbase) + offset) MOD vstride;
bits( ) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.52 FMLSL (multiple and indexed vector)

Multi-vector floating-point multiply-subtract long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the indexed element of the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups.

The half-precision elements within the second source vector are specified using a 3-bit immediate index which selects the same element position within each 128-bit vector segment.

The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

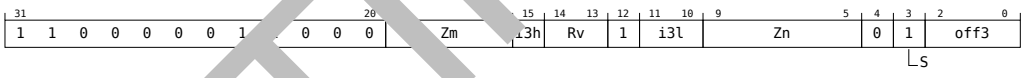
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the Zm operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behavior.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector, Two ZA double-vectors and Four ZA double-vectors

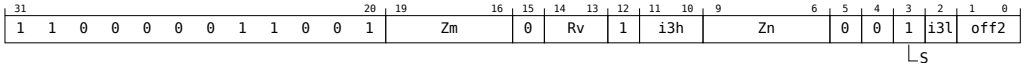
One ZA double-vector
(FEAT_SME2)




```
FMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 1;
```

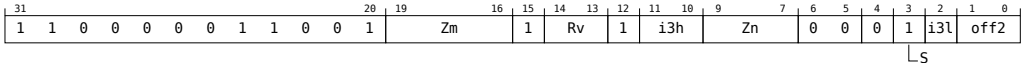
Two ZA double-vectors
(FEAT_SME2)




```
FMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)




```
FMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV 32;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, s, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = FPNeg(element1);
            Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.53 FMLSL (multiple and single vector)

Multi-vector floating-point multiply-subtract long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the one, two, or four first source vectors and the second source vector to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

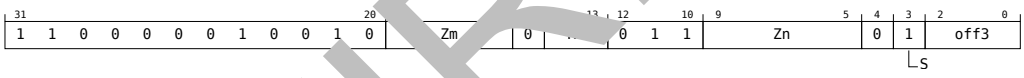
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ... consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is ...eferred ... disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behav...rs.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector , ... ZA ...ouble-vectors and Four ZA double-vectors

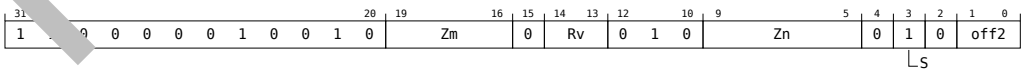
One ZA double-vector
(FEAT_SME2)




```
FMLSL   ZA.S[<Wv>, <off...>...offsl>], <Z...>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(...Zn);
integer m = UI...('0':Z...;
integer offset ...UIn...off3:'0'...;
boolean sub_op = ...E;
constan... ...eger nr... = 1;
```

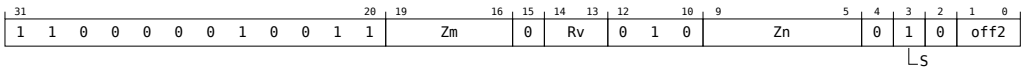
Tw...A doub...vectors
(FEA...SM...2)




```
FMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)




```
FMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = FPNeg(element1);
            Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.54 FMLSL (multiple vectors)

Multi-vector floating-point multiply-subtract long

The instruction operates on two or four ZA double-vector groups.

This half-precision floating-point multiply-subtract long instruction widens all 16-bit half-precision elements in the two or four first and second source vectors to single-precision format, then multiplies the corresponding elements and destructively subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

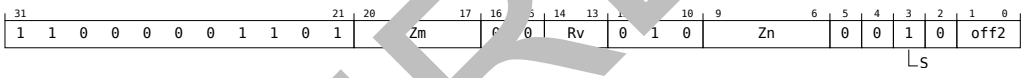
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors a[illegible] Four Z[illegible] double-[illegible]ors

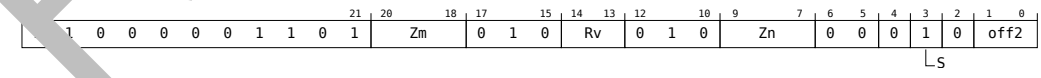
Two ZA double-vectors
(FEAT_SME2)




```
FMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('01':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 2;
```

Four [illegible]A doub[illegible]-vectors
(FE[illegible]T_SME[illegible]




```
FMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
boolean sub_op = TRUE;
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field

times 2.

<offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[n, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec + (vbase MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[(m+r) MOD 32, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            bits(16) element1 = Elem[operand1, 2 * e + i, 16];
            bits(16) element2 = Elem[operand2, 2 * e + i, 16];
            bits(32) element3 = Elem[operand3, e, 32];
            if sub_op then element1 = FPNeg(element1);
            Elem[result, e, 32] = FPMulAddH_ZA(element3, element1, element2, FPCR[]);
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.55 FMOPA (widening)

Half-precision floating-point sum of outer products and accumulate

The half-precision floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile.

This instruction widens the $SVL_S \times 2$ sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened $2 \times SVL_S$ sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

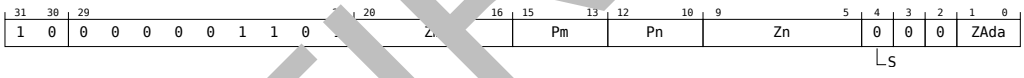
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME
(FEAT_SME)




```
FMOPA   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
1  if !HaveSME() then UNDEFINED;
2  integer a = UInt(Pn);
3  integer b = UInt(Pm);
4  integer n = UInt(Zn);
5  integer m = UInt(Zm);
6  integer za = UInt(ZAda);
7  boolean sub_op = FALSE;
```

Assembler symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer dim = VL DIV 32;
5  bits(PL) mask1 = P[a, PL];
6  bits(PL) mask2 = P[b, PL];
7  bits(VL) operand1 = Z[n, VL];
```

```
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        // determine row/col predicates
        boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
        boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
        boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
        boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));

        bits(32) sum = Elem[operand3, row*dim+col, 32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0',
                ↪16));
            bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0',
                ↪16));
            bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0',
                ↪16));
            bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0',
                ↪16));
            if sub_op then
                if prow_0 then erow_0 = FPNeg(erow_0);
                if prow_1 then erow_1 = FPNeg(erow_1);
            sum = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

        Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

D1.1.56 FMOPA (non-widening)

Floating-point outer product and accumulate

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

These instructions generate an outer product of the first source vector and the second source vector. In case of the single-precision variant, the first source is $SVL_S \times 1$ vector and the second source is $1 \times SVL_S$ vector. In case of the double-precision variant, the first source is $SVL_D \times 1$ vector and the second source is $1 \times SVL_D$ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

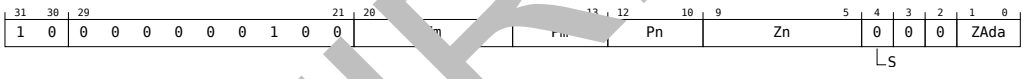
The resulting outer product, $SVL_S \times SVL_S$ in case of single-precision variant or $SVL_D \times SVL_D$ in case of double-precision variant, is then destructively added to the destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

It has encodings from 2 classes: Single-precision and Double-precision

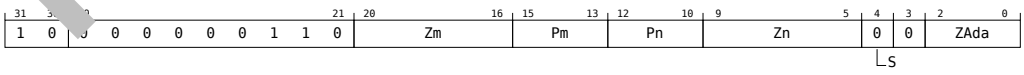
Single-precision
(FEAT_SME)




```
FMOPA   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Double-precision
(FEAT_SME_F64F64)




```
FMOPA   <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D
```

```
if !HaveSMEF64F64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

<ZAda> For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) element1 = Elem[operand1, row, esize];
        bits(esize) element2 = Elem[operand2, col, esize];
        bits(esize) element3 = Elem[operand3, row*dim+col, esize];

        if (ActivePredicateElement(mask1, row, esize) &&
              ActivePredicateElement(mask2, col, esize)) then
            if subtract then element1 = FPNeg(element1);
            Elem[result, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2,
                ↪FPCR[]);
        else
            Elem[result, row*dim+col, esize] = element3;

ZAtile[da, esize, dim*dim*esize] = result;
```

D1.1.57 FMOPS (widening)

Half-precision floating-point sum of outer products and subtract

The half-precision floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile.

This instruction widens the $SVL_S \times 2$ sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened $2 \times SVL_S$ sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

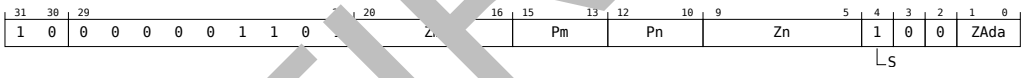
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME
(FEAT_SME)




```
FMOPS   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
```

```
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        // determine row/col predicates
        boolean prow_0 = (ActivePredicateElement(mask1, 2*row + 0, 16));
        boolean prow_1 = (ActivePredicateElement(mask1, 2*row + 1, 16));
        boolean pcol_0 = (ActivePredicateElement(mask2, 2*col + 0, 16));
        boolean pcol_1 = (ActivePredicateElement(mask2, 2*col + 1, 16));

        bits(32) sum = Elem[operand3, row*dim+col, 32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0',
                ↪16));
            bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0',
                ↪16));
            bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0',
                ↪16));
            bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0',
                ↪16));
            if sub_op then
                if prow_0 then erow_0 = FPNeg(erow_0);
                if prow_1 then erow_1 = FPNeg(erow_1);
            sum = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

        Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

D1.1.58 FMOPS (non-widening)

Floating-point outer product and subtract

The single-precision variant works with a 32-bit element ZA tile.

The double-precision variant works with a 64-bit element ZA tile.

These instructions generate an outer product of the first source vector and the second source vector. In case of the single-precision variant, the first source is $SVL_S \times 1$ vector and the second source is $1 \times SVL_S$ vector. In case of the double-precision variant, the first source is $SVL_D \times 1$ vector and the second source is $1 \times SVL_D$ vector.

Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.

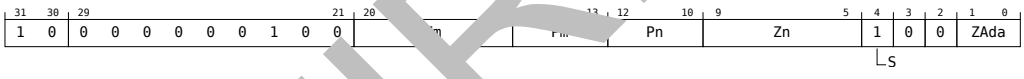
The resulting outer product, $SVL_S \times SVL_S$ in case of single-precision variant or $SVL_D \times SVL_D$ in case of double-precision variant, is then destructively subtracted from the destination tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.

This instruction follows SME ZA-targeting floating-point behaviors.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

It has encodings from 2 classes: Single-precision and Double-precision

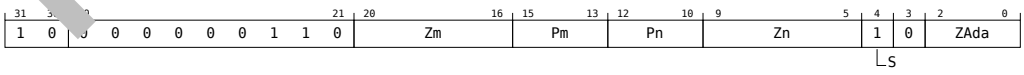
Single-precision
(FEAT_SME)




```
FMOPS   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Double-precision
(FEAT_SME_F64F64)




```
FMOPS   <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D
```

```
if !HaveSMEF64F64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

<ZAda> For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) element1 = Elem[operand1, row, esize];
        bits(esize) element2 = Elem[operand2, col, esize];
        bits(esize) element3 = Elem[operand3, row*dim+col, esize];

        if (ActivePredicateElement(mask1, row, esize) &&
              ActivePredicateElement(mask2, col, esize)) then
            if subtract then element1 = FPNeg(element1);
            Elem[result, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2,
                ↪FPCR[]);
        else
            Elem[result, row*dim+col, esize] = element3;

ZAtile[da, esize, dim*dim*esize] = result;
```

D1.1.59 FRINTA

Multi-vector floating-point round to integral value, to nearest with ties away from zero

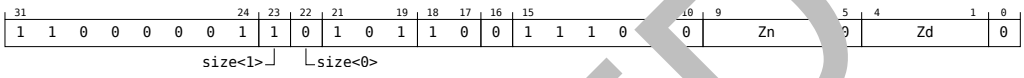
Round to the nearest integral floating-point value, with ties rounding away from zero, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

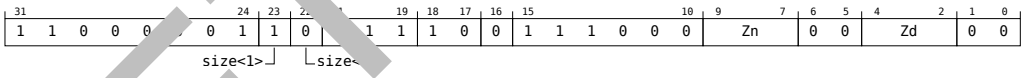
Two registers
(FEAT_SME2)




```
FRINTA  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Zn:'0');
3 integer d = UInt(Zd:'0');
4 constant integer nreg = 2;
5 boolean exact = FALSE;
6 FPRounding rounding = FPRounding_TIEAWAY;
```

Four registers
(FEAT_SME2)




```
FRINTA  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Zn:'00');
3 integer d = UInt(Zd:'00');
4 constant integer nreg = 4;
5 boolean exact = FALSE;
6 FPRounding rounding = FPRounding_TIEAWAY;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPRoundInt(element, FPCR[], rounding, exact);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.60 FRINTM

Multi-vector floating-point round to integral value, toward minus Infinity

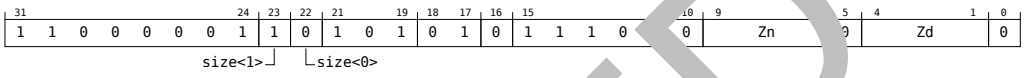
Round down to an integral floating-point value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

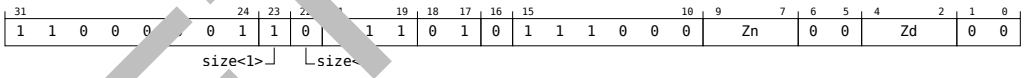
Two registers
(FEAT_SME2)




```
FRINTM  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Zn:'0');
3 integer d = UInt(Zd:'0');
4 constant integer nreg = 2;
5 boolean exact = FALSE;
6 FPRounding rounding = FPRounding_NEGINF;
```

Four registers
(FEAT_SME2)




```
FRINTM  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Zn:'00');
3 integer d = UInt(Zd:'00');
4 constant integer nreg = 4;
5 boolean exact = FALSE;
6 FPRounding rounding = FPRounding_NEGINF;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPRoundInt(element, FPCR[], rounding, exact);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.61 FRINTN

Multi-vector floating-point round to integral value, to nearest with ties to even

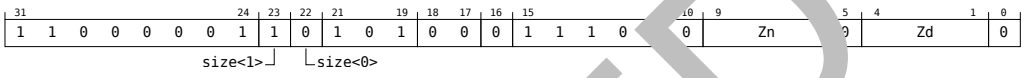
Round to the nearest integral floating-point value, with ties rounding to an even value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

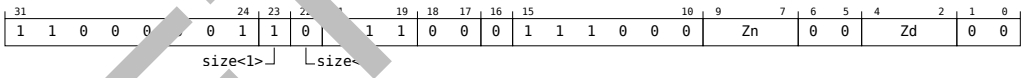
Two registers
(FEAT_SME2)




```
FRINTN  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Zn:'0');
3 integer d = UInt(Zd:'0');
4 constant integer nreg = 2;
5 boolean exact = FALSE;
6 FPRounding rounding = FPRounding_TIEEVEN;
```

Four registers
(FEAT_SME2)




```
FRINTN  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Zn:'00');
3 integer d = UInt(Zd:'00');
4 constant integer nreg = 4;
5 boolean exact = FALSE;
6 FPRounding rounding = FPRounding_TIEEVEN;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPRoundInt(element, FPCR[], rounding, exact);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.62 FRINTP

Multi-vector floating-point round to integral value, toward plus Infinity

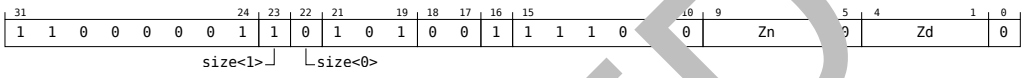
Round up to an integral floating-point value, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

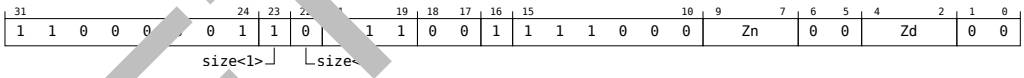
Two registers
(FEAT_SME2)




```
FRINTP  { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Zn:'0');
3 integer d = UInt(Zd:'0');
4 constant integer nreg = 2;
5 boolean exact = FALSE;
6 FPRounding rounding = FPRounding_POSINF;
```

Four registers
(FEAT_SME2)




```
FRINTP  { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Zn:'00');
3 integer d = UInt(Zd:'00');
4 constant integer nreg = 4;
5 boolean exact = FALSE;
6 FPRounding rounding = FPRounding_POSINF;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FPRoundInt(element, FPCR[], rounding, exact);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.63 FSUB

Floating-point subtract multi-vector from ZA array vector accumulators

The instruction operates on two or four ZA single-vector groups.

Destructively subtract all elements of the two or four source vectors from the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.

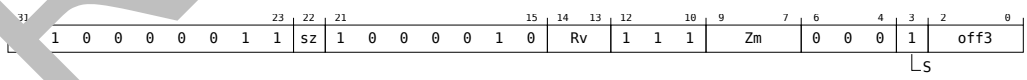
It has encodings from 2 classes: Two ZA single-vectors and four ZA single-vectors

Two ZA single-vectors
(FEAT_SME2)


```
FSUB    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zm1>.<T>-<Zm2>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
FSUB    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEF64F64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = ZAvector[vec, VL];
    bits(VL) operand2 = Zm[r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPSub_ZA(element1, element2, FPCR[]);
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.64 FVDOT

Multi-vector half-precision floating-point vertical dot-product by indexed element

The instruction operates on two ZA single-vector groups.

The instruction computes the vertical fused sum-of-products of the corresponding half-precision floating-point values held in the two first source vectors with pair of half-precision floating-point values in the indexed 32-bit element of the second source vector, without intermediate rounding. The single-precision sum-of-products results are destructively added to the corresponding single-precision elements of the two ZA single-vector groups.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The element index range is from 0 to 3.

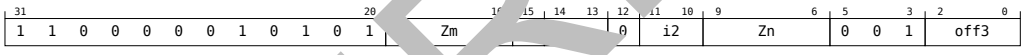
The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand consists of two ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction follows SME ZA-targeting floating-point behaviors.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
FVDOT   ZA.S[<Wv>, <offs>, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('01':Rv);
3  integer n = UInt(Zn:'0');
4  integer m = UInt('0':Zm);
5  integer offset = UInt(off3);
6  integer index = UInt(i2);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer elements = VL DIV 32;
4  integer vectors = VL DIV 8;
5  integer vstride = vectors DIV 2;
6  integer eltspersegment = 128 DIV 32;
7  bits(32) vbase = X[v, 32];
8  integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) result;

for r = 0 to 1
    bits(VL) operand1a = Z[n, VL];
    bits(VL) operand1b = Z[n+1, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(16) elt1_a = Elem[operand1a, 2 * e + r, 16];
        bits(16) elt1_b = Elem[operand1b, 2 * e + r, 16];
        bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
        bits(32) sum = Elem[operand3, e, 32];
        sum = FPDotAdd_ZA(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
        Elem[result, e, 32] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.65 LD1B (scalar plus immediate, consecutive registers)

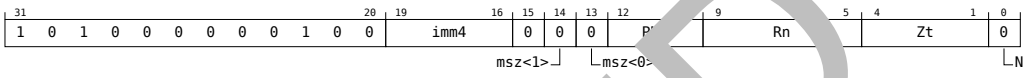
Contiguous load of bytes to multiple consecutive vectors (immediate index)

Contiguous load of unsigned bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

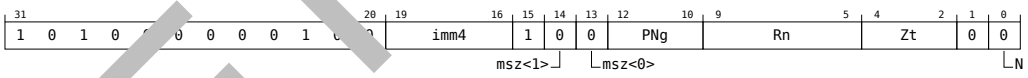
Two registers
(FEAT_SME2)




```
LD1B    { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 8;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LD1B    { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 8;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0> ,VL * nr );
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_L , nonte oral, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredicta eBool(U predictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignme ();
    base = if n == 31 then SP[] else [n, 6

for r = 0 to nreg-1
    for e = 0 to elemen -1
        if ActivePredicateEleme (mask, r * elements + e, esize) then
            bits(64 addr = base (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[ lues[r], e, esi ] = Mem[addr, mbytes, accdesc];
        else
            E [value r], e, esize] = Zeros(esize);

for r = 0 to nreg
    Z[t , v ] = va s[r]
```

D1.1.66 LD1B (scalar plus scalar, consecutive registers)

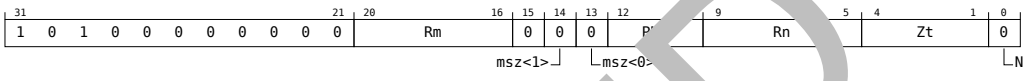
Contiguous load of bytes to multiple consecutive vectors (scalar index)

Contiguous load of unsigned bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

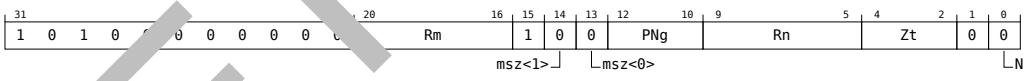
Two registers
(FEAT_SME2)




```
LD1B    { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 8;
```

Four registers
(FEAT_SME2)




```
LD1B    { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 8;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

D1.1.67 LD1B (scalar plus immediate, strided registers)

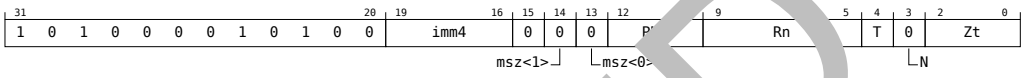
Contiguous load of bytes to multiple strided vectors (immediate index)

Contiguous load of unsigned bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

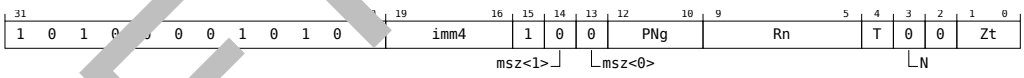
Two registers
(FEAT_SME2)




```
LD1B    { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LD1B    { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.68 LD1B (scalar plus scalar, strided registers)

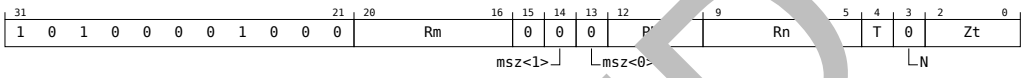
Contiguous load of bytes to multiple strided vectors (scalar index)

Contiguous load of unsigned bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

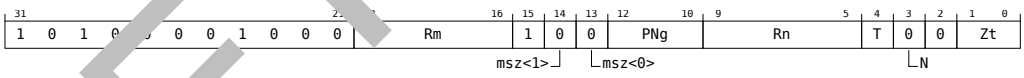
Two registers
(FEAT_SME2)




```
LD1B    { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 8;
```

Four registers
(FEAT_SME2)




```
LD1B    { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 8;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

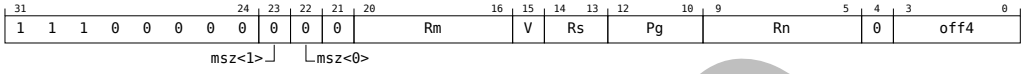
```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDe
nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize)
    if n == 31 && ConstrainU
        CheckSPAlignment()
else
    if n == 31 then Che
    base = if n == 31
    offset = X[m,

for r = 0 to n
    for e = 0
        if Acti
            addr =
            em[va
        else
            em[values[r],

for r
    Z[t,
    t = t
```

D1.1.69 LD1B (scalar plus scalar, tile slice)

Contiguous load of bytes to 8-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 8-bit elements in a vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)




```
LD1B    { ZA0<HV>.B[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>}]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = 0;
integer offset = UInt(off4);
constant integer esize = 8;
boolean vertical = V == '1';
```

Assembler Symbols

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, in the range 0 to 15, encoded in the "off4" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD dim;
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
```

```
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.70 LD1D (scalar plus immediate, consecutive registers)

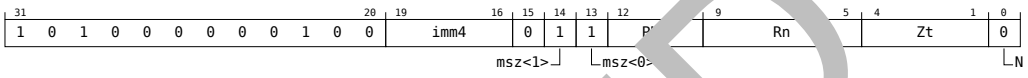
Contiguous load of doublewords to multiple consecutive vectors (immediate index)

Contiguous load of unsigned doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

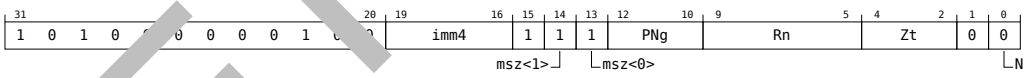
Two registers
(FEAT_SME2)




```
LD1D    { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 64;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LD1D    { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 64;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0> , VL * nr );
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_L , nonte oral, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredicta eBool(U predictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignme ();
    base = if n == 31 then SP[] else [n, 6

for r = 0 to nreg-1
    for e = 0 to elemen -1
        if ActivePredicateEleme (mask, r * elements + e, esize) then
            bits(64 addr = base (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[ lues[r], e, esi ] = Mem[addr, mbytes, accdesc];
        else
            E [value r], e, esize] = Zeros(esize);

for r = 0 to nreg
    Z[t , v ] = va s[r]
```

D1.1.71 LD1D (scalar plus scalar, consecutive registers)

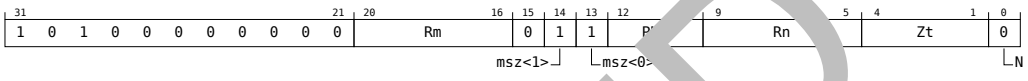
Contiguous load of doublewords to multiple consecutive vectors (scalar index)

Contiguous load of unsigned doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

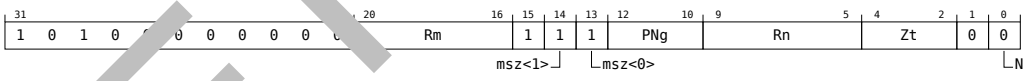
Two registers
(FEAT_SME2)




```
LD1D    { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 64;
```

Four registers
(FEAT_SME2)




```
LD1D    { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 64;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

D1.1.72 LD1D (scalar plus immediate, strided registers)

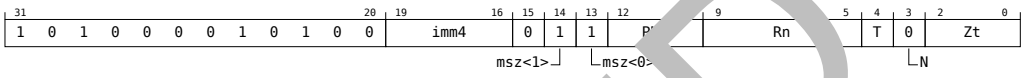
Contiguous load of doublewords to multiple strided vectors (immediate index)

Contiguous load of unsigned doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

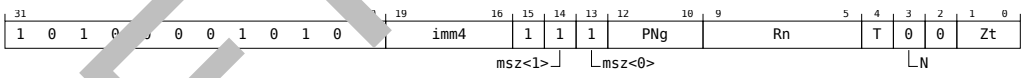
Two registers
(FEAT_SME2)




```
LD1D    { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer tstride = 8;
6  integer t = UInt(T:'0':Zt);
7  constant integer esize = 64;
8  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LD1D    { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer tstride = 4;
6  integer t = UInt(T:'00':Zt);
7  constant integer esize = 64;
8  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.73 LD1D (scalar plus scalar, strided registers)

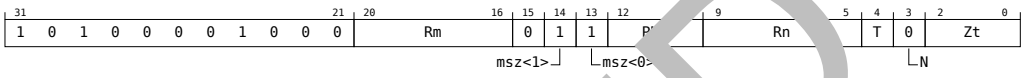
Contiguous load of doublewords to multiple strided vectors (scalar index)

Contiguous load of unsigned doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

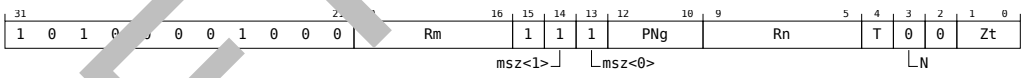
Two registers
(FEAT_SME2)




```
LD1D    { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 64;
```

Four registers
(FEAT_SME2)




```
LD1D    { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 64;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

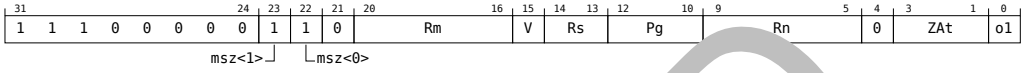
```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15 , PL * nreg
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDe SVE( mop_ nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) n
    if n == 31 && ConstrainU redict leBool mpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment()
else
    if n == 31 then Che SPAl nment();
    base = if n == 31 then SP[] lse X[n, 64];
    offset = X[m, 6 ];

for r = 0 to nr -1
    for e = 0 eleme s-1
        if Activ re cateEle nt(mask, r * elements + e, esize) then
            bits( addr = se + (UInt(offset) + r * elements + e) * mbytes;
            em[va s[r] e, esize] = Mem[addr, mbytes, accdesc];
        else
            em[values[r], e, esize] = Zeros(esize);

for r to nreg-1
    Z[t, ] = values[r];
    t = t stride;
```

D1.1.74 LD1D (scalar plus scalar, tile slice)

Contiguous load of doublewords to 64-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 64-bit elements in a vector. The immediate offset is in the range 0 to 1. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)




```
LD1D    { <ZAt><HV>.D[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]
```

```
1  if !HaveSME() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('0':Pg);
5  integer s = UInt('011':Rs);
6  integer t = UInt(ZAt);
7  integer offset = UInt(o1);
8  constant integer esize = 64;
9  boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile, ZA0-ZA7 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, in the range 0 to 1, encoded in the "o1" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer dim = VL DIV esize;
5  bits(64) base;
6  bits(64) addr;
7  bits(PL) mask = P[g, PL];
8  bits(64) moffs = X[m, 64];
9  bits(32) index = X[s, 32];
10 integer  slice = (UInt(index) + offset) MOD dim;
11 bits(VL) result;
```

```
constant integer mbytes = esize DIV 8;
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.75 LD1H (scalar plus immediate, consecutive registers)

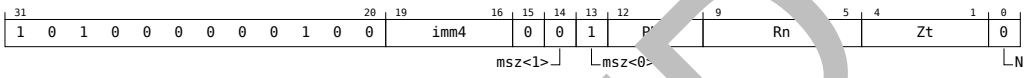
Contiguous load of halfwords to multiple consecutive vectors (immediate index)

Contiguous load of unsigned halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

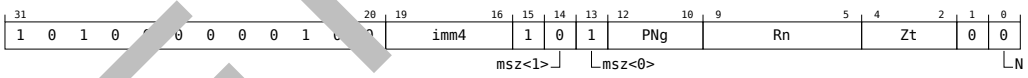
Two registers
(FEAT_SME2)




```
LD1H    { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>, #<imm>, MUL VL]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 16;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LD1H    { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 16;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0> , VL * nr );
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_L , nonte oral, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredicta eBool(U predictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignme ();
    base = if n == 31 then SP[] else [n, 6

for r = 0 to nreg-1
    for e = 0 to eleme -1
        if ActivePredicateEleme (mask, r * elements + e, esize) then
            bits(64 addr = base (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[ lues[r], e, esi = Mem[addr, mbytes, accdesc];
        else
            E [value r], e, esize] = Zeros(esize);

for r = 0 to nreg
    Z[t , v = va s[r]
```

D1.1.76 LD1H (scalar plus scalar, consecutive registers)

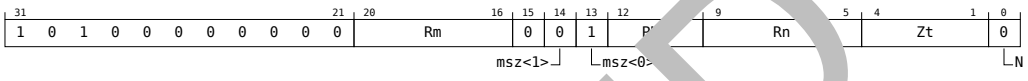
Contiguous load of halfwords to multiple consecutive vectors (scalar index)

Contiguous load of unsigned halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

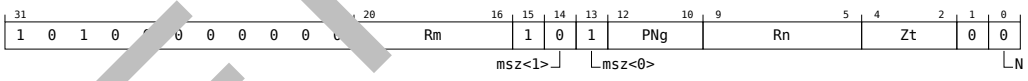
Two registers
(FEAT_SME2)




```
LD1H    { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 16;
```

Four registers
(FEAT_SME2)




```
LD1H    { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 16;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

D1.1.77 LD1H (scalar plus immediate, strided registers)

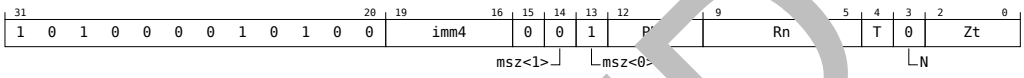
Contiguous load of halfwords to multiple strided vectors (immediate index)

Contiguous load of unsigned halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

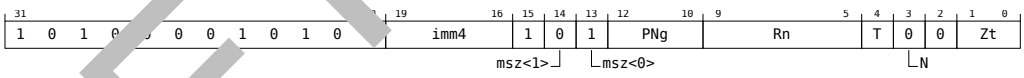
Two registers
(FEAT_SME2)




```
LD1H    { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer tstride = 8;
6  integer t = UInt(T:'0':Zt);
7  constant integer esize = 16;
8  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LD1H    { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer tstride = 4;
6  integer t = UInt(T:'00':Zt);
7  constant integer esize = 16;
8  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.78 LD1H (scalar plus scalar, strided registers)

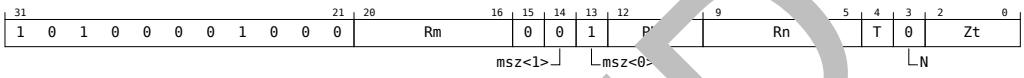
Contiguous load of halfwords to multiple strided vectors (scalar index)

Contiguous load of unsigned halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

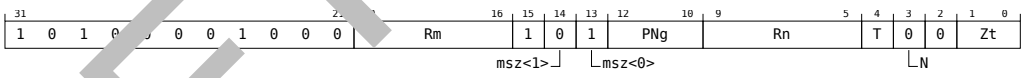
Two registers
(FEAT_SME2)




```
LD1H    { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 16;
```

Four registers
(FEAT_SME2)




```
LD1H    { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 16;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

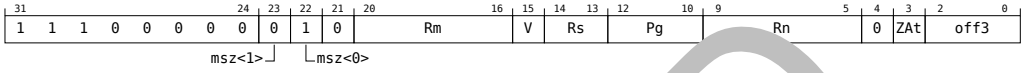
```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDe
nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize)
    if n == 31 && ConstrainU
        CheckSPAlignment()
else
    if n == 31 then Che
    base = if n == 31
    offset = X[m,

for r = 0 to n
    for e = 0
        if Acti
            addr =
            em[va
        else
            em[values[r], e, esize] = Zeros(esize);

for r
    Z[t,
    t = t
```

D1.1.79 LD1H (scalar plus scalar, tile slice)

Contiguous load of halfwords to 16-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 16-bit elements in a vector. The immediate offset is in the range 0 to 7. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)




```
LD1H    { <ZAt><HV>.H[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(off3);
constant integer esize = 16;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, in the range 0 to 7, encoded in the "off3" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD dim;
bits(VL) result;
```

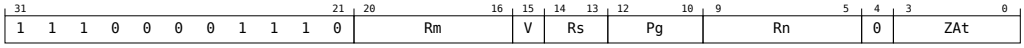
```
constant integer mbytes = esize DIV 8;
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.80 LD1Q

Contiguous load of quadwords to 128-bit element ZA tile slice

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)




```
LD1Q    { <ZAt><HV>.Q[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #4]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = 0;
constant integer esize = 128;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile, ZA0-ZA15, to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD dim;
bits(VL) result;
constant integer mbytes = esize DIV 8;
boolean  contiguous = TRUE;
```

```
boolean  nontemporal = FALSE;
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.81 LD1W (scalar plus immediate, consecutive registers)

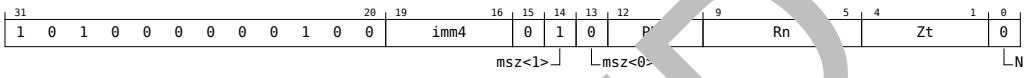
Contiguous load of words to multiple consecutive vectors (immediate index)

Contiguous load of unsigned words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

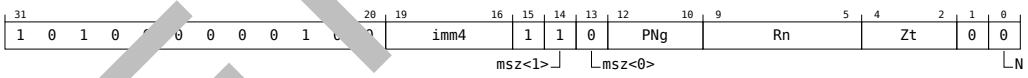
Two registers
(FEAT_SME2)




```
LD1W    { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>, #<imm>, MUL VL]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 32;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LD1W    { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 32;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0> ,VL * nr );
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_L , nonte oral, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredicta eBool(U predictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignme ();
    base = if n == 31 then SP[] else [n, 6

for r = 0 to nreg-1
    for e = 0 to elemen -1
        if ActivePredicateEleme (mask, r * elements + e, esize) then
            bits(64 addr = base (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[ lues[r], e, esi = Mem[addr, mbytes, accdesc];
        else
            E [value r], e, esize] = Zeros(esize);

for r = 0 to nreg
    Z[t , v = va s[r]
```

D1.1.82 LD1W (scalar plus scalar, consecutive registers)

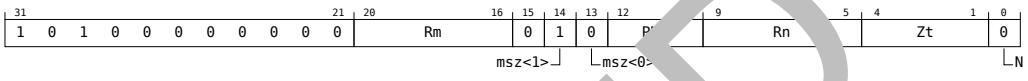
Contiguous load of words to multiple consecutive vectors (scalar index)

Contiguous load of unsigned words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

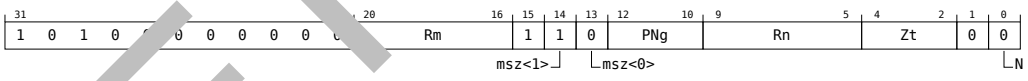
Two registers
(FEAT_SME2)




```
LD1W    { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 32;
```

Four registers
(FEAT_SME2)




```
LD1W    { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 32;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter

encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

D1.1.83 LD1W (scalar plus immediate, strided registers)

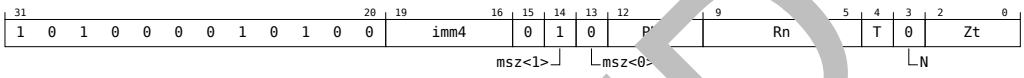
Contiguous load of words to multiple strided vectors (immediate index)

Contiguous load of unsigned words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

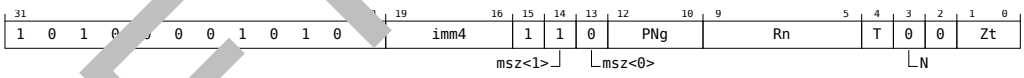
Two registers
(FEAT_SME2)




```
LD1W    { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer tstride = 8;
6  integer t = UInt(T:'0':Zt);
7  constant integer esize = 32;
8  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LD1W    { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer tstride = 4;
6  integer t = UInt(T:'00':Zt);
7  constant integer esize = 32;
8  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.84 LD1W (scalar plus scalar, strided registers)

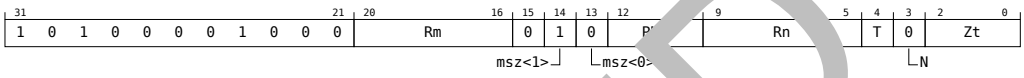
Contiguous load of words to multiple strided vectors (scalar index)

Contiguous load of unsigned words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: Two registers and Four registers

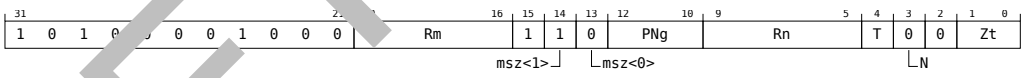
Two registers
(FEAT_SME2)




```
LD1W    { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 32;
```

Four registers
(FEAT_SME2)




```
LD1W    { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 32;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

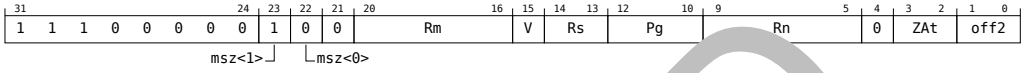
```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15 , PL * nreg
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDe SVE( mop_ nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) n
    if n == 31 && ConstrainU redict leBool mpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment()
else
    if n == 31 then Che SPAl nment();
    base = if n == 31 then SP[] lse X[n, 64];
    offset = X[m, 6 ];

for r = 0 to n g-1
    for e = 0 eleme s-1
        if Activ reElement(mask, r * elements + e, esize) then
            bits( addr = se + (UInt(offset) + r * elements + e) * mbytes;
            em[va s[r] e, esize] = Mem[addr, mbytes, accdesc];
        else
            em[values[r], e, esize] = Zeros(esize);

for r to nreg-1
    Z[t, ] = values[r];
    t = t stride;
```

D1.1.85 LD1W (scalar plus scalar, tile slice)

Contiguous load of words to 32-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 32-bit elements in a vector. The immediate offset is in the range 0 to 3. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)




```
LD1W    { <ZAt><HV>.S[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]
```

```
1  if !HaveSME() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('0':Pg);
5  integer s = UInt('011':Rs);
6  integer t = UInt(ZAt);
7  integer offset = UInt(off2);
8  constant integer esize = 32;
9  boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile, ZA0-ZA3 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, in the range 0 to 3, encoded in the "off2" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer dim = VL DIV esize;
5  bits(64) base;
6  bits(64) addr;
7  bits(PL) mask = P[g, PL];
8  bits(64) moffs = X[m, 64];
9  bits(32) index = X[s, 32];
10 integer  slice = (UInt(index) + offset) MOD dim;
11 bits(VL) result;
```

```
constant integer mbytes = esize DIV 8;
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

for e = 0 to dim - 1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = Mem[addr, mbytes, accdesc];
    else
        Elem[result, e, esize] = Zeros(esize);
    moffs = moffs + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

D1.1.86 LDNT1B (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of bytes to multiple consecutive vectors (immediate index)

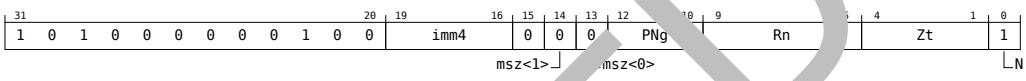
Contiguous load non-temporal of bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

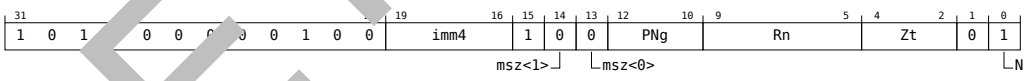
Two registers
(FEAT_SME2)




```
LDNT1B  { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 8;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LDNT1B  { <Zt1>.B-<Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 8;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:  , PL * nreg,
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDesc  E(Me  o_LOA  nor  mporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) th
    if n == 31 && ConstrainUnpredictab  Bool  npred  ble_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPA  gnment
    base = if n == 31 then  [] else X[  64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActiveP  icateElement  ask, r * elements + e, esize) then
            bits  ) addr = base +  ffset * nreg * elements + r * elements + e) * mbytes;
            El  [values[  ], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem  l  s[r], e  esize] = Zeros(esize);

for r =  to  reg-1
    Z  +r, VL  = value  ];
```

D1.1.87 LDNT1B (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of bytes to multiple consecutive vectors (scalar index)

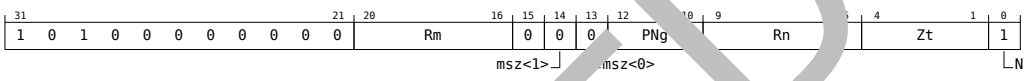
Contiguous load non-temporal of bytes to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

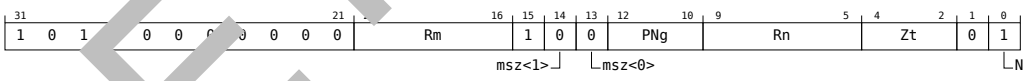
Two registers
(FEAT_SME2)




```
LDNT1B  { <Zt1>.B-<Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 8;
```

Four registers
(FEAT_SME2)




```
LDNT  { <Z >.B-<Zt .B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

```
1  if !Ha  2() then UNDEFINED;
2  integer  = UInt(Rn);
3  integer m  UInt(Rm);
4  integer g =  nt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 8;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

D1.1.88 LDNT1B (scalar plus immediate, strided registers)

Contiguous load non-temporal of bytes to multiple strided vectors (immediate index)

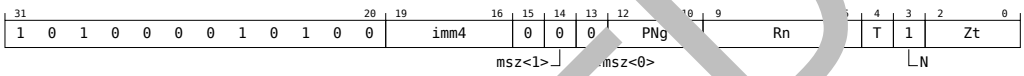
Contiguous load non-temporal of bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

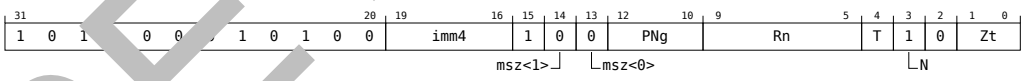
Two registers
(FEAT_SME2)




```
LDNT1B  { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
integer offset = SInt(imm
```

Four registers
(FEAT_SME2)




```
LDN  { <  <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !Have E2() then UNDEFINED;
integer n  UInt(Rn);
integer g =  nt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActiveElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.89 LDNT1B (scalar plus scalar, strided registers)

Contiguous load non-temporal of bytes to multiple strided vectors (scalar index)

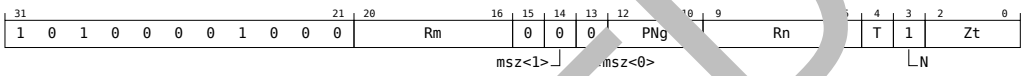
Contiguous load non-temporal of bytes to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

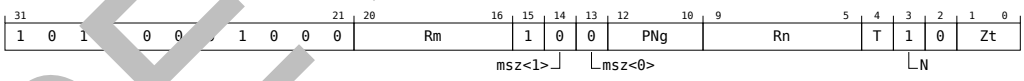
Two registers
(FEAT_SME2)




```
LDNT1B  { <Zt1>.B, <Zt2>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize =
```

Four registers
(FEAT_SME2)




```
LDN  B  { <  <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>/Z, [<Xn|SP>, <Xm>]
```

```
1  if !Have E2() then UNDEFINED;
2  integer n  UInt(Rn);
3  integer m =  nt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 8;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.90 LDNT1D (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of doublewords to multiple consecutive vectors (immediate index)

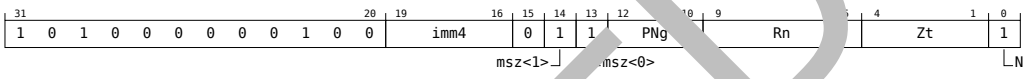
Contiguous load non-temporal of doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

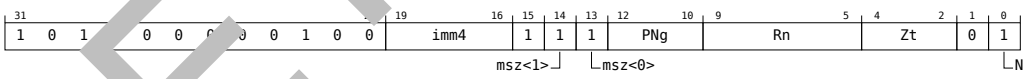
Two registers
(FEAT_SME2)




```
LDNT1D  { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 64;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LDNT1D  { <Zt1>.D-<Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 64;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:  , PL * nreg,
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDesc  E(Me  o_LOA  non  mporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) th
    if n == 31 && ConstrainUnpredictab  Bool  npred  ble_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPA ignment
    base = if n == 31 then  [] else X  64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActiveP  icateElement  ask, r * elements + e, esize) then
            bits  ) addr = base +  ffset * nreg * elements + r * elements + e) * mbytes;
            El  [values  , e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem  l  s[r], e  esize] = Zeros(esize);

for r =  to  reg-1
    Z  +r, VL  = value  ];
```

D1.1.91 LDNT1D (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of doublewords to multiple consecutive vectors (scalar index)

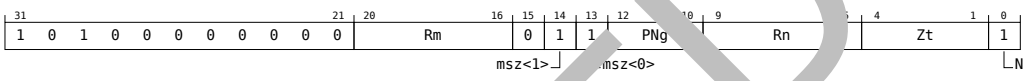
Contiguous load non-temporal of doublewords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

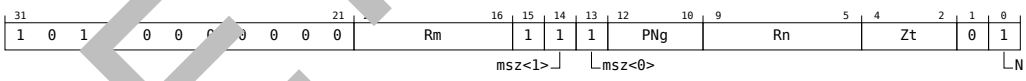
Two registers
(FEAT_SME2)




```
LDNT1D  { <Zt1>.D-<Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 64;
```

Four registers
(FEAT_SME2)




```
LDNT  { <Zt>.D-<Zt>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 64;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

D1.1.92 LDNT1D (scalar plus immediate, strided registers)

Contiguous load non-temporal of doublewords to multiple strided vectors (immediate index)

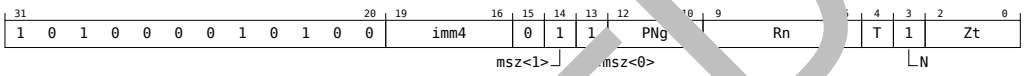
Contiguous load non-temporal of doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

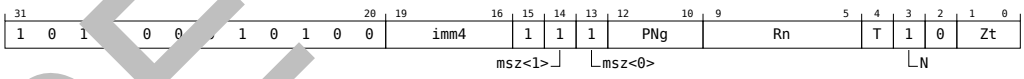
Two registers (FEAT_SME2)




```
LDNT1D  { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
integer offset = SInt(imm
```

Four registers (FEAT_SME2)




```
LDN  D  { <  <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !Have E2() then UNDEFINED;
integer n  UInt(Rn);
integer g =  nt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.93 LDNT1D (scalar plus scalar, strided registers)

Contiguous load non-temporal of doublewords to multiple strided vectors (scalar index)

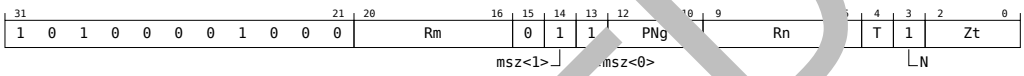
Contiguous load non-temporal of doublewords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

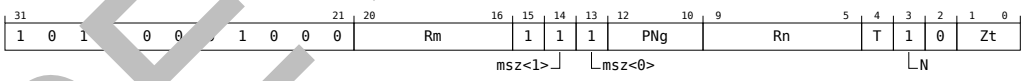
Two registers
(FEAT_SME2)




```
LDNT1D  { <Zt1>.D, <Zt2>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize =
```

Four registers
(FEAT_SME2)




```
LDN  D  { <  <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !Have E2() then UNDEFINED;
2  integer n  UInt(Rn);
3  integer m =  nt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 64;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.94 LDNT1H (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of halfwords to multiple consecutive vectors (immediate index)

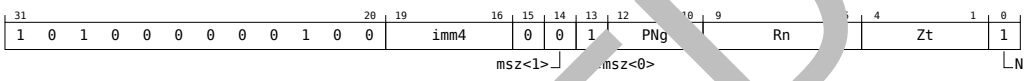
Contiguous load non-temporal of halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

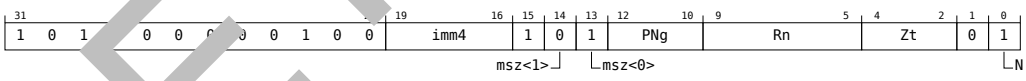
Two registers
(FEAT_SME2)




```
LDNT1H  { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 16;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LDNT1H  { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 16;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:  , PL * nreg,
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDesc  E(Me  o_LOA  non  mporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) th
    if n == 31 && ConstrainUnpredictab  Bool  npred  ble_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPA ignment
    base = if n == 31 then  [] else X  64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActiveP  icateElement  ask, r * elements + e, esize) then
            bits  ) addr = base +  ffset * nreg * elements + r * elements + e) * mbytes;
            El  [values ], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem  l  s[r], e  esize] = Zeros(esize);

for r =  to  reg-1
    Z  +r, VL  = value  ];
```

D1.1.95 LDNT1H (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of halfwords to multiple consecutive vectors (scalar index)

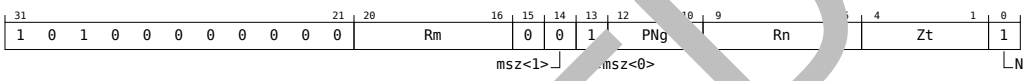
Contiguous load non-temporal of halfwords to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

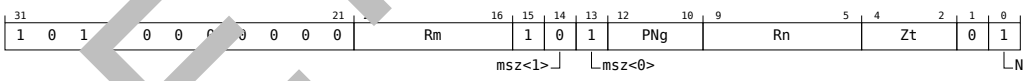
Two registers
(FEAT_SME2)




```
LDNT1H  { <Zt1>.H-<Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 16
```

Four registers
(FEAT_SME2)




```
LDNT1  { <Zt1>.H-<Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 16;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(Mem[ LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableP r(  redic ble_C CKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment(
    base = if n == 31 then SP[] e se X[n, 6 ];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements
        if ActivePredic eEl nt(mask, r elements + e, esize) then
            bits(64) addr = ba + (UInt(offset) + r * elements + e) * mbytes;
            Elem[va es[r], e, e e] = Mem[addr, mbytes, accdesc];
        else
            Ele values[ ], e, esize] = Zeros(esize);

for r = 0 to nr -1
    Z[t+r, VL] = es[r];
```

D1.1.96 LDNT1H (scalar plus immediate, strided registers)

Contiguous load non-temporal of halfwords to multiple strided vectors (immediate index)

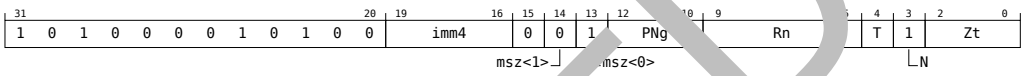
Contiguous load non-temporal of halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

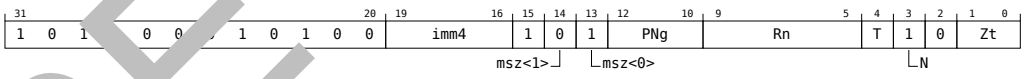
Two registers (FEAT_SME2)




```
LDNT1H  { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
integer offset = SInt(imm
```

Four registers (FEAT_SME2)




```
LDN  H  { <  <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !Have E2() then UNDEFINED;
integer n  UInt(Rn);
integer g =  nt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.97 LDNT1H (scalar plus scalar, strided registers)

Contiguous load non-temporal of halfwords to multiple strided vectors (scalar index)

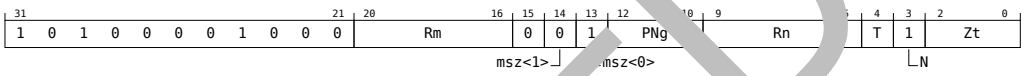
Contiguous load non-temporal of halfwords to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

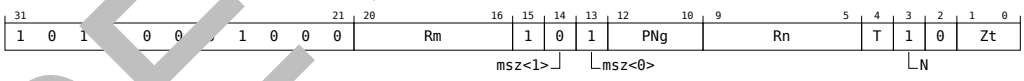
Two registers (FEAT_SME2)




```
LDNT1H  { <Zt1>.H, <Zt2>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
```

Four registers (FEAT_SME2)




```
LDNT1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.98 LDNT1W (scalar plus immediate, consecutive registers)

Contiguous load non-temporal of words to multiple consecutive vectors (immediate index)

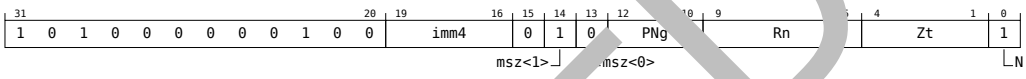
Contiguous load non-temporal of words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

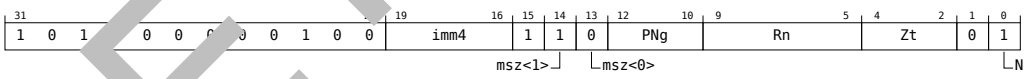
Two registers
(FEAT_SME2)




```
LDNT1W  { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 32;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
LDNT1W  { <Zt1>.S-<Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 32;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:  , PL * nreg,
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDesc  E(Me  o_LOA  nor  mporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) th
    if n == 31 && ConstrainUnpredictab  Bool  npred  ble_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPA  gnment
    base = if n == 31 then  [] else X[  64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActiveP  icateElement  ask, r * elements + e, esize) then
            bits  ) addr = base +  ffset * nreg * elements + r * elements + e) * mbytes;
            El  [values[  ], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem  l  s[r], e  esize] = Zeros(esize);

for r =  to  reg-1
    Z  +r, VL  = value  ];
```

D1.1.99 LDNT1W (scalar plus scalar, consecutive registers)

Contiguous load non-temporal of words to multiple consecutive vectors (scalar index)

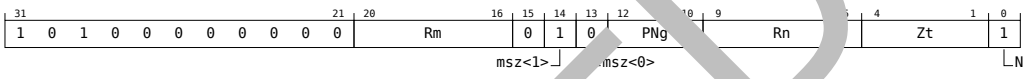
Contiguous load non-temporal of words to elements of two or four consecutive vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

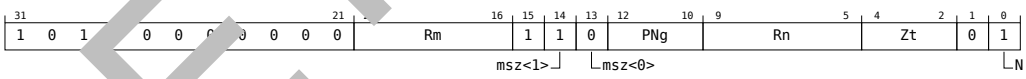
Two registers
(FEAT_SME2)




```
LDNT1W  { <Zt1>.S-<Zt2>.S }, <PNg>/Z, [<Xn|SP>,  >, LSL #
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 32
```

Four registers
(FEAT_SME2)




```
LDNT  { <Z >.S-<Zt .S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !Ha  S  2() then UNDEFINED;
2  integer  = UInt(Rn);
3  integer m  UInt(Rm);
4  integer g =  nt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 32;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(Mem[ LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableP[ predic ble_C CKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment(
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t+r, VL] = values[r];
```

D1.1.100 LDNT1W (scalar plus immediate, strided registers)

Contiguous load non-temporal of words to multiple strided vectors (immediate index)

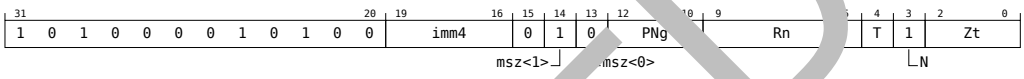
Contiguous load non-temporal of words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

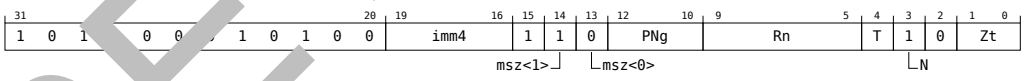
Two registers (FEAT_SME2)




```
LDNT1W  { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
integer offset = SInt(imm
```

Four registers (FEAT_SME2)




```
LDN  W  { <  <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !Have E2() then UNDEFINED;
integer n  UInt(Rn);
integer g =  nt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActiveElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.101 LDNT1W (scalar plus scalar, strided registers)

Contiguous load non-temporal of words to multiple strided vectors (scalar index)

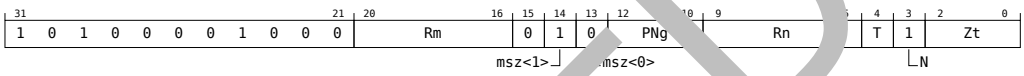
Contiguous load non-temporal of words to elements of two or four strided vector registers from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

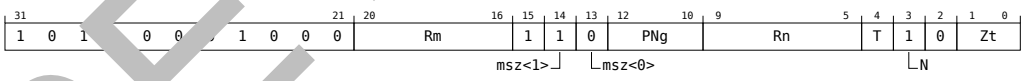
Two registers (FEAT_SME2)




```
LDNT1W  { <Zt1>.S, <Zt2>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 32;
```

Four registers (FEAT_SME2)




```
LDNT1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>/Z, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 32;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
array [0..3] of bits(VL) values;
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, accdesc];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[t, VL] = values[r];
    t = t + tstride;
```

D1.1.102 LDR (vector)

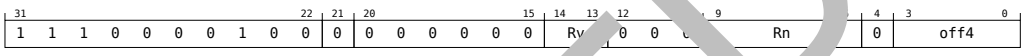
Load ZA array vector

The ZA array vector is selected by the sum of the vector select register and immediate offset, modulo the number of bytes in a Streaming SVE vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME
(FEAT_SME)




```
LDR     ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MU }]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer v = UInt('011':Rv);
integer offset = UInt(off4);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.

<offs> Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the "off4" field.

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

Operation

```
CheckSMEAndZAEnabled();
constant integer SVL = CurrentSVL;
constant integer dim = SVL DIV 8;
bits(64) base;
integer offs = offset * dim;
bits(SVL) result;
bits(32)  vbase = X[v, 32];
integer   vec = (UInt(vbase) + offset) MOD dim;
boolean   contiguous = TRUE;
boolean   nontemporal = FALSE;
boolean   tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

boolean aligned = IsAligned(base + offset, 16);

```

```
25  if !aligned && AlignmentEnforced() then
26      AArch64.Abort(base + moffs, AlignmentFault(accdesc));
27
28  for e = 0 to dim-1
29      Elem[result, e, 8] = AArch64.MemSingle[base + moffs, 1, accdesc, aligned];
30      moffs = moffs + 1;
31
32  ZAvector[vec, SVL] = result;
```

D1.1.103 LDR (ZT0)

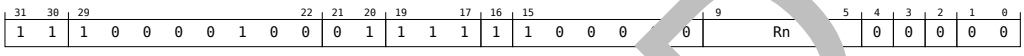
Load ZT0 register

Load the 64-byte ZT0 register from the memory address provided in the 64-bit scalar base register. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2
(FEAT_SME2)




```
LDR     ZT0, [<Xn|SP>]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
```

Assembler Symbols

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

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
constant integer elements = 512 DIV 8;
bits(64) base;
bits(512) result;
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_LOAD, nontemporal, contiguous, tagchecked);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

boolean aligned = IsAligned(base, 16);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base, AlignmentFault(accdesc));

for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + e, 1, accdesc, aligned];

ZT0[512] = result;
```

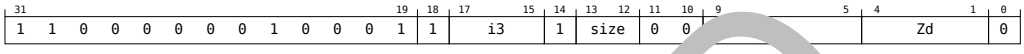
D1.1.104 LUTI2 (two registers)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to two destination vectors using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
LUTI2   { <Zd1>.<T>-<Zd2>.<T> }, ZT0, <Zn>[<inde  ]
```

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 2;
integer n = UInt(Zn);
integer d = UInt(Zd:'0');
integer imm = UInt(i3);
constant integer nreg = 2;
```

Assembler Symbols

<Zd1> Is the name of the first d nation scalab ector register of a multi-vector sequence, encoded as "Zd" times 2.

<T> Is the size speci r, encoded in ize":

size	<T>
00	B
01	H
10	
11	ESERV D

<Zd2> t name of the second destination scalable vector register of a multi-vector sequence, e ded as "Zd" times 2 plus 1.

<Zn> Is the me of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 7, encoded in the "i3" field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result = Z[d+r, VL];
    for e = 0 to elements-1
```

```
14          integer index = UInt(Elem[indexes, base+e, isize]);
15          Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
16      Z[d+r, VL] = result;
```

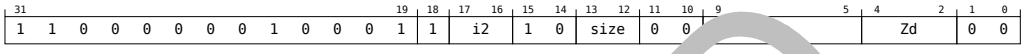
D1.1.105 LUTI2 (four registers)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to four destination vectors using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
LUTI2   { <Zd1>.<T>-<Zd4>.<T> }, ZT0, <Zn>[<index>]
```

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 2;
integer n = UInt(Zn);
integer d = UInt(Zd:'00');
integer imm = UInt(i2);
constant integer nreg = 4;
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result = Z[d+r, VL];
    for e = 0 to elements-1
```

```
14          integer index = UInt(Elem[indexes, base+e, isize]);
15          Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
16      Z[d+r, VL] = result;
```

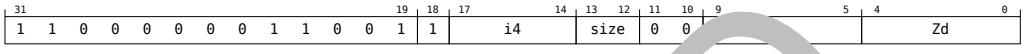
D1.1.106 LUTI2 (single)

Lookup table read with 2-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to one destination vector using packed 2-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
LUTI2   <Zd>.<T>, ZT0, <Zn>[<index>]
```

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 2;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer imm = UInt(i4);
constant integer nreg = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 15, encoded in the "i4" field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result = Z[d+r, VL];
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
        Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
    Z[d+r, VL] = result;
```

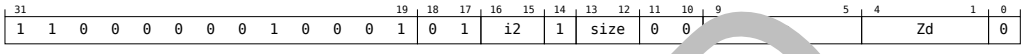
D1.1.107 LUTI4 (two registers)

Lookup table read with 4-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to two destination vectors using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
LUTI4   { <Zd1>.<T>-<Zd2>.<T> }, ZT0, <Zn>[<inde ]
```

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 4;
integer n = UInt(Zn);
integer d = UInt(Zd:'0');
integer imm = UInt(i2);
constant integer nreg = 2;
```

Assembler Symbols

<Zd1> Is the name of the first d nation scalab ector register of a multi-vector sequence, encoded as "Zd" times 2.

<T> Is the size speci r, encoded in ize":

size	<T>
00	B
01	H
10	
11	ESERV D

<Zd2> th name of the second destination scalable vector register of a multi-vector sequence, en ded as "Zd" times 2 plus 1.

<Zn> Is the me of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result = Z[d+r, VL];
    for e = 0 to elements-1
```

```
14          integer index = UInt(Elem[indexes, base+e, isize]);
15          Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
16      Z[d+r, VL] = result;
```

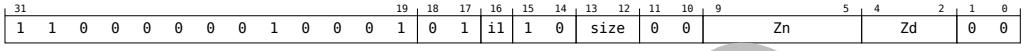
D1.1.108 LUTI4 (four registers)

Lookup table read with 4-bit indexes

Copy 16-bit or 32-bit elements from ZT0 to four destination vectors using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
LUTI4   { <Zd1>.<T>-<Zd4>.<T> }, ZT0, <Zn>[<index>]
```

```
if !HaveSME2() then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 4;
integer n = UInt(Zn);
integer d = UInt(Zd:'00');
integer imm = UInt(i1);
constant integer nreg = 4;
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	RESERVED

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result = Z[d+r, VL];
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
```

```
15          Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
16      Z[d+r, VL] = result;
```

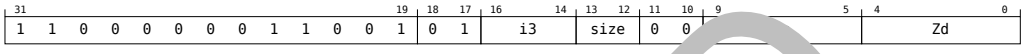
D1.1.109 LUTI4 (single)

Lookup table read with 4-bit indexes

Copy 8-bit, 16-bit or 32-bit elements from ZT0 to one destination vector using packed 4-bit indices from a segment of the source vector register. A segment corresponds to a portion of the source vector that is consumed in order to fill the destination vector. The segment is selected by the vector segment index modulo the total number of segments.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
LUTI4   <Zd>.<T>, ZT0, <Zn>[<index>]
```

```
if !HaveSME2() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer isize = 4;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer imm = UInt(i3);
constant integer nreg = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<index> Is the vector segment index, in the range 0 to 7, encoded in the "i3" field.

Operation

```
CheckStreamingSVEEnabled();
CheckSMEZT0Enabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer segments = esize DIV (isize * nreg);
integer segment = imm MOD segments;
bits(VL) indexes = Z[n, VL];
bits(512) table = ZT0[512];

for r = 0 to nreg-1
    integer base = (segment * nreg + r) * elements;
    bits(VL) result = Z[d+r, VL];
    for e = 0 to elements-1
        integer index = UInt(Elem[indexes, base+e, isize]);
        Elem[result, e, esize] = Elem[table, index, 32]<esize-1:0>;
    Z[d+r, VL] = result;
```

D1.1.110 MOV (tile to vector, two registers)

Move two ZA tile slices to two vector registers

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

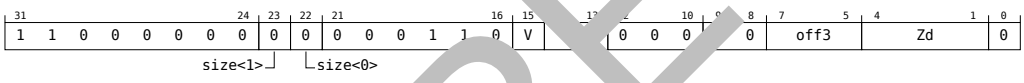
This instruction is unpredicated.

This is an alias of MOVA (tile to vector, two registers). This means:

- The encodings in this description are named to match the encodings of MOVA (tile to vector, two registers).
- The description of MOVA (tile to vector, two registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: 8-bit , 16-bit , 32-bit and 64-bit

8-bit



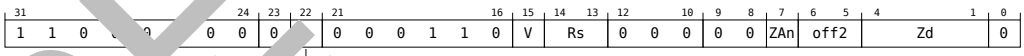

```
MOV     { <Zd1>.B-<Zd2>.B }, ZA<HV>.B[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA    { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

16-bit



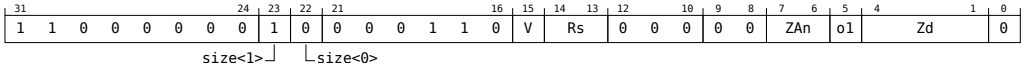

```
MOV     { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA    { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

32-bit



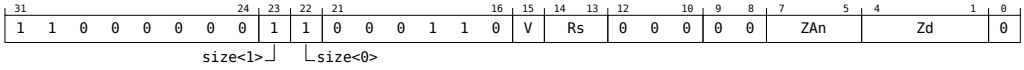

```
MOV     { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA    { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

64-bit




```
MOV     { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA    { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offsf> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.

For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.

For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.

For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.

For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

Operation

The description of MOVA (tile to vector, two registers) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

D1.1.111 MOV (tile to vector, four registers)

Move four ZA tile slices to four vector registers

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

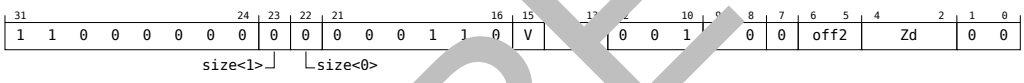
This instruction is unpredicated.

This is an alias of MOVA (tile to vector, four registers). This means:

- The encodings in this description are named to match the encodings of MOVA (tile to vector, four registers).
- The description of MOVA (tile to vector, four registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: 8-bit , 16-bit , 32-bit and 64-bit

8-bit



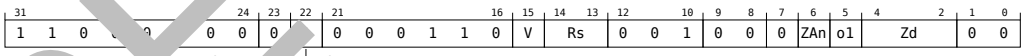

```
MOV     { <Zd1>.B-<Zd4>.B }, ZA<HV>.B<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA    { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

16-bit



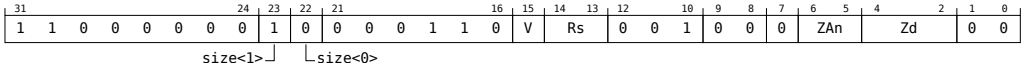

```
MOV     { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA    { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

32-bit



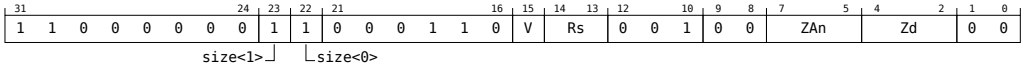

```
MOV     { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA    { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

64-bit




```
MOV     { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]
```

is equivalent to

```
MOVA    { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]
```

and is always the preferred disassembly.

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offsf> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.

For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

Operation

The description of MOVA (tile to vector, four registers) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

D1.1.112 MOV (array to vector, two registers)

Move two ZA single-vector groups to two vector registers

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

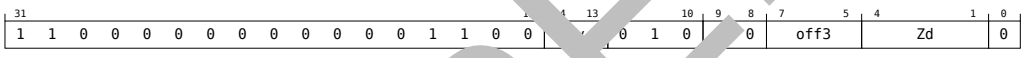
The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (array to vector, two registers). This means:

- The encodings in this description are named to match the encodi[illegible] of MOVA (ar[illegible]y to vector, two registers).
- The description of MOVA (array to vector, two regist[illegible]) gives t[illegible] operati[illegible]al pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any opera[illegible]nal informatio[illegible]or [illegible]s instruction.




```
MOV     { <Zd1>.D-<Zd2>.D }, ZA.D[<W[illegible], <[illegible]2}]
```

is equivalent to

```
MOVA    { <Zd1>.D-<Zd2>.D[illegible], ZA.D[<W[illegible]  <off[illegible]>{, VGx2}]
```

and is always the preferred [illegible]isassem[illegible]y.

Assembler Symbols

<Zd1> Is the nam[illegible]of the firs[illegible]estination scalable vector register of a multi-vector sequence, encoded as "Zd" ti[illegible]2.

<Zd2> Is[illegible]me o[illegible]e secon[illegible]destination scalable vector register of a multi-vector sequence, [illegible]ncoded[illegible]s "Zd" t[illegible]plus 1.

<Wv> Is the [illegible]me of the vector select register W8-W11, encoded in the "Rv" field.

<offs> I[illegible]e vector select offset, in the range 0 to 7, encoded in the "off3" field.

Operation

The description of MOVA (array to vector, two registers) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

D1.1.113 MOV (array to vector, four registers)

Move four ZA single-vector groups to four vector registers

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

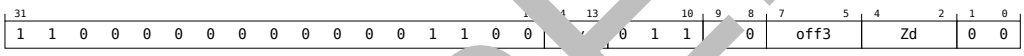
The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (array to vector, four registers). This means:

- The encodings in this description are named to match the encoding of MOVA (array to vector, four registers).
- The description of MOVA (array to vector, four registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.




```
MOV     { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]
```

is equivalent to

```
MOVA    { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]
```

and is always the preferred disassembly.

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

Operation

The description of MOVA (array to vector, four registers) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

D1.1.114 MOV (tile to vector, single)

Move ZA tile slice to vector register

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

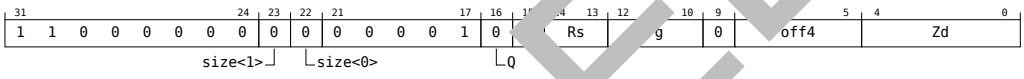
Inactive elements in the destination vector remain unmodified.

This is an alias of MOVA (tile to vector, single). This means:

- The encodings in this description are named to match the encodings of MOVA (tile to vector, single).
- The description of MOVA (tile to vector, single) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for the instruction.

It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

8-bit



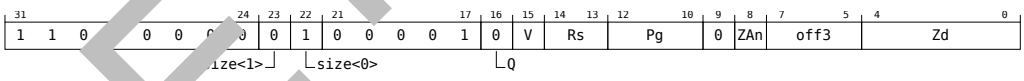

```
MOV     <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]
```

is equivalent to

```
MOVA    <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]
```

and is always the preferred disassembly.

16-bit



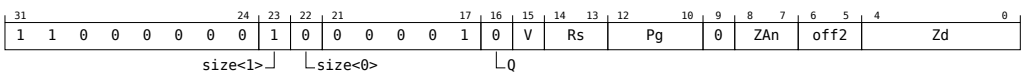

```
MOV     <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]
```

is equivalent to

```
MOVA    <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]
```

and is always the preferred disassembly.

32-bit



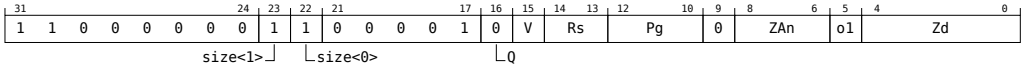

```
MOV     <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]
```

is equivalent to

```
MOVA    <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]
```

and is always the preferred disassembly.

64-bit



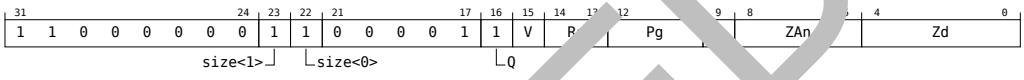

```
MOV     <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]
```

is equivalent to

```
MOVA    <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]
```

and is always the preferred disassembly.

128-bit




```
MOV     <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<W
```

is equivalent to

```
MOVA    <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[
```

and is always the preferred disass oly.

Assembler Symbols

<Zd> Is the name of the destinatio alable vector register, encoded in the "Zd" field.

<Pg> Is the name o e governing scal e predicate register P0-P7, encoded in the "Pg" field.

<ZAn> For the 1 bit varia is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" fiel

F 2-bit v iant: i e name of the ZA tile ZA0-ZA3 to be accessed, encoded in the ZAn" f d.

For th iant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the " field.

For 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" eld.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.

For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.

For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.

For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.

For the 128-bit variant: is the slice index offset 0.

Operation

The description of MOVA (tile to vector, single) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

D1.1.115 MOV (vector to tile, two registers)

Move two vector registers to two ZA tile slices

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

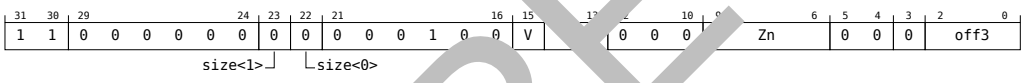
This instruction is unpredicated.

This is an alias of MOVA (vector to tile, two registers). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to tile, two registers).
- The description of MOVA (vector to tile, two registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: 8-bit , 16-bit , 32-bit and 64-bit

8-bit



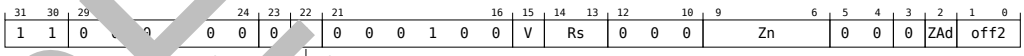

```
MOV      ZA0<HV>.B[<Ws>, <offsf>:<offsl>], { <Zn1>.B-<Zn2>.B }
```

is equivalent to

```
MOVA     ZA0<HV>.B[<Ws>, <offsf>:<offsl>], { <Zn1>.B-<Zn2>.B }
```

and is always the preferred disassembly.

16-bit



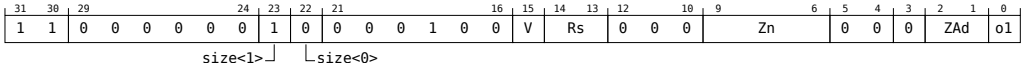

```
MOV      <ZAd><HV>.H[<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn2>.H }
```

is equivalent to

```
MOVA     <ZAd><HV>.H[<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn2>.H }
```

and is always the preferred disassembly.

32-bit



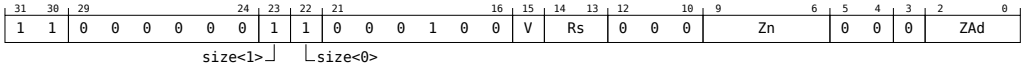

```
MOV      <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn2>.S }
```

is equivalent to

```
MOVA     <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn2>.S }
```

and is always the preferred disassembly.

64-bit




```
MOV     <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Zn1>.D-<Zn2>.D }
```

is equivalent to

```
MOVA    <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Zn1>.D-<Zn2>.D }
```

and is always the preferred disassembly.

Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-Z[illegible]1 to b[illegible]ccessed, en[illegible]oded in the "ZAd" field.

For the 32-bit variant: is the name of the ZA tile [illegible]A0-ZA3[illegible] be acces[illegible], encoded in the "ZAd" field.

For the 64-bit variant: is the name of the [illegible] ZA0-Z[illegible]7 to b[illegible]ccessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indic[illegible]r, en[illegible]".

V	<HV>
0	H
1	V

<Ws> Is the 32-bit nam[illegible] of the slice[illegible]dex register W12-W15, encoded in the "Rs" field.

<offsf> For the 8-bit[illegible]riant: is the slice[illegible]dex offset, pointing to first of two consecutive slices, encoded a[illegible]off3" fie[illegible] times 2.

For the 16-[illegible] v[illegible]ant: is th[illegible]slice index offset, pointing to first of two consecutive slices, en[illegible]as "o[illegible]" field ti[illegible]s 2.

For the [illegible]-bit vari[illegible]is the slice index offset, pointing to first of two consecutive slices, encod[illegible]"[illegible]1" field times 2.

F[illegible]he 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, wit[illegible]mplicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.

For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

The description of MOVA (vector to tile, two registers) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

D1.1.116 MOV (vector to tile, four registers)

Move four vector registers to four ZA tile slices

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

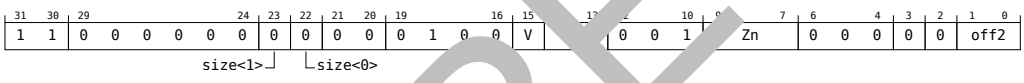
This instruction is unpredicated.

This is an alias of MOVA (vector to tile, four registers). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to tile, four registers).
- The description of MOVA (vector to tile, four registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 4 classes: 8-bit , 16-bit , 32-bit and 64-bit

8-bit



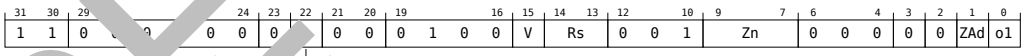

```
MOV     ZA0<HV>.B[<Ws>, <offsf>:<offsl>], { <Zn1>.B-<Zn4>.B }
```

is equivalent to

```
MOVA    ZA0<HV>.B[<Ws>, <offsf>:<offsl>], { <Zn1>.B-<Zn4>.B }
```

and is always the preferred disassembly.

16-bit



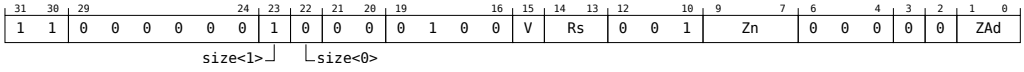

```
MOV     <ZAd><HV>.H[<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn4>.H }
```

is equivalent to

```
MOVA    <ZAd><HV>.H[<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn4>.H }
```

and is always the preferred disassembly.

32-bit



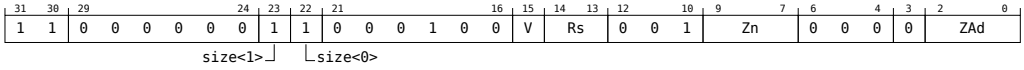

```
MOV     <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn4>.S }
```

is equivalent to

```
MOVA    <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn4>.S }
```

and is always the preferred disassembly.

64-bit




```
MOV     <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Zn1>.D-<Zn4>.D }
```

is equivalent to

```
MOVA    <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Zn1>.D-<Zn4>.D }
```

and is always the preferred disassembly.

Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-Z[illegible]1 to b[illegible]ccessed, en[illegible]oded in the "ZAd" field.

For the 32-bit variant: is the name of the ZA tile[illegible]A0-ZA3[illegible]be acces[illegible]d, encoded in the "ZAd" field.

For the 64-bit variant: is the name of the [illegible]ZA0-Z[illegible]7 to b[illegible]ccessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indic[illegible]r, en[illegible]".

V	<HV>
0	H
1	V

<Ws> Is the 32-bit nam[illegible] of the slice[illegible]dex register W12-W15, encoded in the "Rs" field.

<offsf> For the 8-bit[illegible]riant: is the slice[illegible]dex offset, pointing to first of four consecutive slices, encoded a[illegible]off2" fie[illegible]times 4.

For the 16-[illegible]v[illegible]iant: is th[illegible]slice index offset, pointing to first of four consecutive slices, en[illegible]as "o1" field tim[illegible]4.

For the [illegible]-bit and [illegible]t variant: is the slice index offset, pointing to first of four consecutive slices[illegible]implicit value 0.

<offsl> [illegible]the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, enc[illegible]d as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive slices, with implicit value 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

The description of MOVA (vector to tile, four registers) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

D1.1.117 MOV (vector to array, two registers)

Move two vector registers to two ZA single-vector groups

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (vector to array, two registers). This means:

- The encodings in this description are named to match the encoding of MOVA (vector to array, two registers).
- The description of MOVA (vector to array, two registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.


```
MOV      ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }
```

is equivalent to

```
MOVA     ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }
```

and is always the preferred disassembly.

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

The description of MOVA (vector to array, two registers) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

D1.1.118 MOV (vector to array, four registers)

Move four vector registers to four ZA single-vector groups

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This is an alias of MOVA (vector to array, four registers). This means:

- The encodings in this description are named to match the encoding of MOVA (vector to array, four registers).
- The description of MOVA (vector to array, four registers) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.


```
MOV      ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }
```

is equivalent to

```
MOVA     ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D-<Zn4>.D }
```

and is always the preferred disassembly.

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

The description of MOVA (vector to array, four registers) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

D1.1.119 MOV (vector to tile, single)

Move vector register to ZA tile slice

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

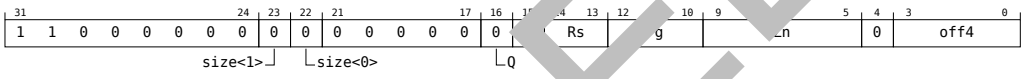
Inactive elements in the destination slice remain unmodified.

This is an alias of MOVA (vector to tile, single). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to tile, single).
- The description of MOVA (vector to tile, single) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

8-bit



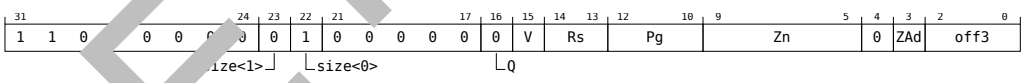

```
MOV     ZA0<HV>.B[<Ws>, <offs>], <Pg>/M, <Zn>.B
```

is equivalent to

```
MOVA    ZA0<HV>.B[<Ws>, <offs>], <Pg>/M, <Zn>.B
```

and is always the preferred disassembly.

16-bit



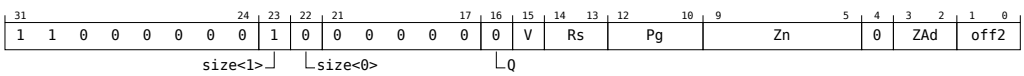

```
MOV     <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H
```

is equivalent to

```
MOVA    <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H
```

and is always the preferred disassembly.

32-bit



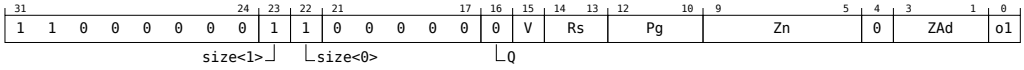

```
MOV     <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S
```

is equivalent to

```
MOVA    <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S
```

and is always the preferred disassembly.

64-bit



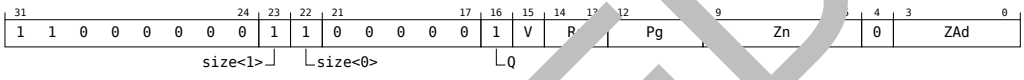

```
MOV     <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D
```

is equivalent to

```
MOVA    <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D
```

and is always the preferred disassembly.

128-bit




```
MOV     <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q
```

is equivalent to

```
MOVA    <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q
```

and is always the preferred disassembly.

Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.

For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.

For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.

For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.

For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

The description of MOVA (vector to tile, single) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

D1.1.120 MOVA (tile to vector, two registers)

Move two ZA tile slices to two vector registers

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

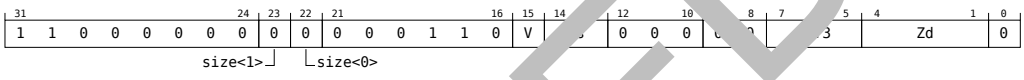
The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This instruction is used by the alias MOV (tile to vector, two registers).

It has encodings from 4 classes: 8-bit , 16-bit , 32-bit and 64-bit

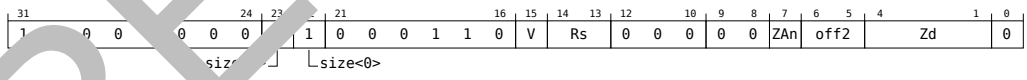
8-bit
(FEAT_SME2)




```
MOVA     { <Zd1>.B-<Zd2>.B }, ZA0<HV>.B[<Ws>, <offsf>:<offsl>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer s = UInt('011':Rs);
3  constant integer nreg = 2;
4  constant integer esize = 8;
5  integer d = UInt(Zd:'0');
6  integer n = 0;
7  integer offset = UInt(offs:'0');
8  boolean vertical = V == '1';
```

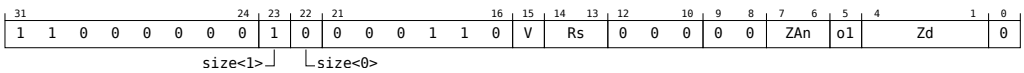
16-bit
(FEAT_SME2)




```
MOVA     { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer s = UInt('011':Rs);
3  constant integer nreg = 2;
4  constant integer esize = 16;
5  integer d = UInt(Zd:'0');
6  integer n = UInt(ZAn);
7  integer offset = UInt(off2:'0');
8  boolean vertical = V == '1';
```

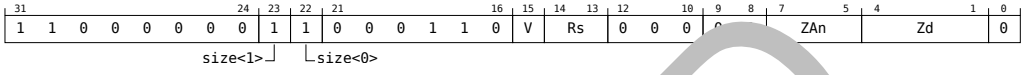
32-bit
(FEAT_SME2)



```
MOVA    { <Zd1>.S-<Zd2>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 32;
integer d = UInt(Zd:'0');
integer n = UInt(ZAn);
integer offset = UInt(o1:'0');
boolean vertical = V == '1';
```

64-bit
(FEAT_SME2)




```
MOVA    { <Zd1>.D-<Zd2>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 64;
integer d = UInt(Zd:'0');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offsf> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.

For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.

For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.

For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices,

with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off2" field times 2 plus 1.

For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
integer slices = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1
    bits(VL) result = ZAslice[n, esize, vertical, slice + r, VL];
    Z[d + r, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.121 MOVA (tile to vector, four registers)

Move four ZA tile slices to four vector registers

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

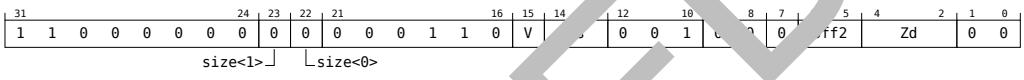
The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This instruction is used by the alias MOV (tile to vector, four registers).

It has encodings from 4 classes: 8-bit , 16-bit , 32-bit and 64-bit

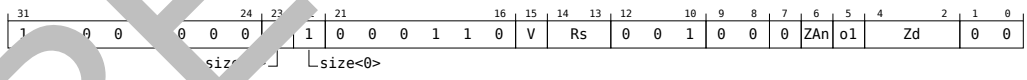
8-bit
(FEAT_SME2)




```
MOVA     { <Zd1>.B-<Zd4>.B }, ZA0<HV>.B[<Ws>, <offsf>:<offsl>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer s = UInt('011':Rs);
3  constant integer nreg = 4;
4  constant integer esize = 8;
5  integer d = UInt(Zd:'00');
6  integer n = 0;
7  integer offset = UInt(o[illegible]:'00');
8  boolean vertical = V == '1';
```

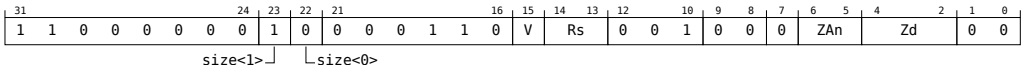
16-bit
(FEAT_SME2)




```
MOVA     { <Zd1>.H-<Zd4>.H }, <ZAn><HV>.H[<Ws>, <offsf>:<offsl>]
```

```
1  if !HaveSME[illegible] then UNDEFINED;
2  integer s = UInt('011':Rs);
3  constant integer nreg = 4;
4  constant integer esize = 16;
5  integer d = UInt(Zd:'00');
6  integer n = UInt(ZAn);
7  integer offset = UInt(o1:'00');
8  boolean vertical = V == '1';
```

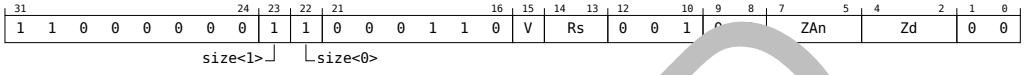
32-bit
(FEAT_SME2)



```
MOVA    { <Zd1>.S-<Zd4>.S }, <ZAn><HV>.S[<Ws>, <offsf>:<offsl>]
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 32;
integer d = UInt(Zd:'00');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

64-bit
(FEAT_SME2)




```
MOVA    { <Zd1>.D-<Zd4>.D }, <ZAn><HV>.D[<Ws>, <offsf>:<offsl>]
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 64;
integer d = UInt(Zd:'00');
integer n = UInt(ZAn);
integer offset = 0;
boolean vertical = V == '1';
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offsf> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.

For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices,

encoded as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, with implicit value 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
integer slices = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1
    bits(VL) result = ZAslice[n, esize, vertical, slice + r, VL];
    Z[d + r, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.122 MOVA (array to vector, two registers)

Move two ZA single-vector groups to two vector registers

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

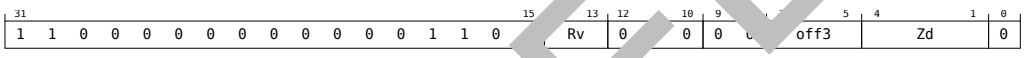
The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (array to vector, two registers)

SME2
(FEAT_SME2)




```
MOVA    { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = ZAvector[vec, VL];
    Z[d + r, VL] = result;
    vec = vec + vstride;
```

Operational information

If PSTATE.DIT is 1:

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

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

D1.1.123 MOVA (array to vector, four registers)

Move four ZA single-vector groups to four vector registers

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

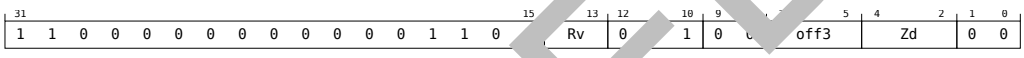
The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (array to vector, four register

SME2
(FEAT_SME2)




```
MOVA    { <Zd1>.D-<Zd4>.D }, ZA.D[<Wv>, <offs>{, VGx4}]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer d = UInt(Zd:'00');
constant integer nreg = 4;
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = ZAvector[vec, VL];
    Z[d + r, VL] = result;
    vec = vec + vstride;
```

Operational information

If PSTATE.DIT is 1:

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

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

D1.1.124 MOVA (tile to vector, single)

Move ZA tile slice to vector register

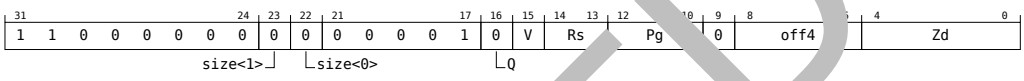
The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination vector remain unmodified.

This instruction is used by the alias MOV (tile to vector, single).

It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

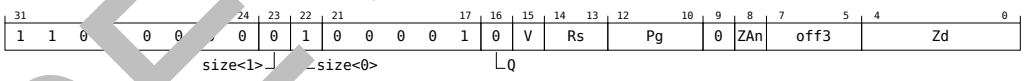
8-bit
(FEAT_SME)




```
MOVA     <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <offs>]
```

```
1  if !HaveSME() then UNDEFINED;
2  integer g = UInt(Pg);
3  integer s = UInt('011':Rs);
4  integer n = 0;
5  integer offset = UInt(off4);
6  constant integer esize = 8;
7  integer d = UInt(Zd);
8  boolean vertical = V == '
```

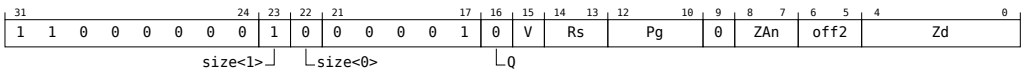
16-bit
(FEAT_SME)




```
MOV      <Zd      Pg>/M, <ZAn><HV>.H[<Ws>, <offs>]
```

```
1  if !Have    E() then UNDEFINED;
2  integer g    UInt(Pg);
3  integer s =    nt('011':Rs);
4  integer n = UInt(ZAn);
5  integer offset = UInt(off3);
6  constant integer esize = 16;
7  integer d = UInt(Zd);
8  boolean vertical = V == '1';
```

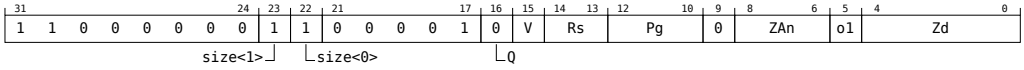
32-bit
(FEAT_SME)




```
MOVA     <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <offs>]
```

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = UInt(off2);
constant integer esize = 32;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

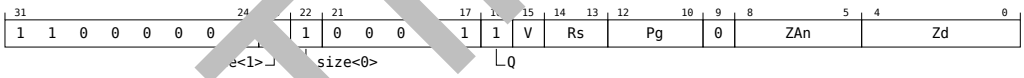
64-bit
(FEAT_SME)




```
MOVA    <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <offs>]
```

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = UInt(o1);
constant integer esize = 64;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

128-bit
(FEAT_SME)




```
MOVA    <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <offs>]
```

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer offset = 0;
constant integer esize = 128;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.

For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.

For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.

For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.

For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.

For the 128-bit variant: is the slice index offset 0.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD di
bits(VL) operand = ZAslice[n, esize, v ical, lice, V
bits(VL) result = Z[d, VL];

for e = 0 to dim-1
    bits(esize) element = Elem[ope d, e, size];
    if ActivePredicateElement ask, esize then
        Elem[result, e, esize] = eleme

Z[d, VL] = result;
```

Operational inform on

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

  The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

  The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:

  – The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

  – The values of the NZCV flags.

D1.1.125 MOVA (vector to tile, two registers)

Move two vector registers to two ZA tile slices

The instruction operates on two consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

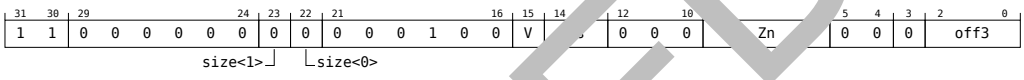
The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 2 in the range 0 to the number of elements in a 128-bit vector segment minus 2.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to tile, two registers).

It has encodings from 4 classes: 8-bit , 16-bit , 32-bit and 64-bit

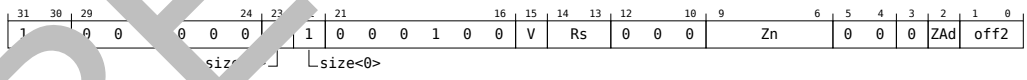
8-bit
(FEAT_SME2)




```
MOVA     ZA0<HV>.B[<Ws>, <offsf>:<offsl>], { <Zn1>.B-<Zn2>.B }
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 8;
integer n = UInt(Zn:'0');
integer d = 0;
integer offset = UInt(offs:'0');
boolean vertical = V == '1';
```

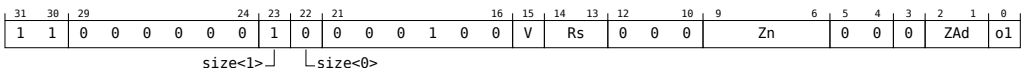
16-bit
(FEAT_SME2)




```
MOVA     ZAd><HV>.H[<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn2>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(ZAd);
integer offset = UInt(off2:'0');
boolean vertical = V == '1';
```

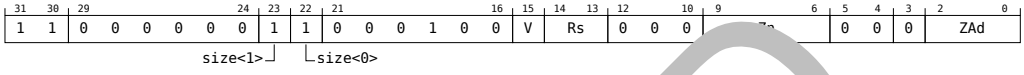
32-bit
(FEAT_SME2)



```
MOVA    <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer d = UInt(ZAd);
integer offset = UInt(o1:'0');
boolean vertical = V == '1';
```

64-bit
(FEAT_SME2)




```
MOVA    <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Z  >.D-<Z  >.D }
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 2;
constant integer esize = 64;
integer n = UInt(Zn:'0');
integer d = UInt(ZAd);
integer offset = 0;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAd> For the 16-bit variant: i e name of t ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32-bit v ant: is the na of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.

For the 6 bit vari is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

<HV> I he h zonta vert al slice indicator, encoded in "V":

V	<HV>
0	
1	V

<Ws> Is the 3 -bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offsf> For the 8-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off3" field times 2.

For the 16-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "off2" field times 2.

For the 32-bit variant: is the slice index offset, pointing to first of two consecutive slices, encoded as "o1" field times 2.

For the 64-bit variant: is the slice index offset, pointing to first of two consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "off3" field times 2 plus 1.

For the 16-bit variant: is the slice index offset, pointing to last of two consecutive slices,

encoded as "off2" field times 2 plus 1.

For the 32-bit variant: is the slice index offset, pointing to last of two consecutive slices, encoded as "o1" field times 2 plus 1.

For the 64-bit variant: is the slice index offset, pointing to last of two consecutive slices, with implicit value 1.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
integer slices = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1
    bits(VL) result = Z[n + r, VL];
    ZAslice[d, esize, vertical, slice + r, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.126 MOVA (vector to tile, four registers)

Move four vector registers to four ZA tile slices

The instruction operates on four consecutive horizontal or vertical slices within a named ZA tile of the specified element size.

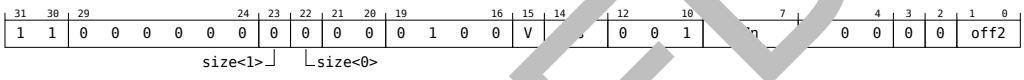
The consecutive slice numbers within the tile are selected starting from the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is a multiple of 4 in the range 0 to the number of elements in a 128-bit vector segment minus 4.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to tile, four registers).

It has encodings from 4 classes: 8-bit , 16-bit , 32-bit and 64-bit

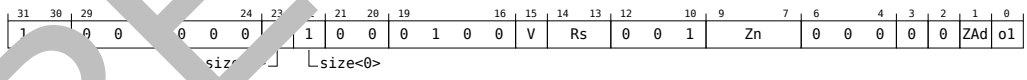
8-bit
(FEAT_SME2)




```
MOVA     ZA0<HV>.B[<Ws>, <offsf>:<offsl>], { <Zn1>.B-<Zn4>.B }
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 8;
integer n = UInt(Zn:'00');
integer d = 0;
integer offset = UInt(o1:'00');
boolean vertical = V == '1';
```

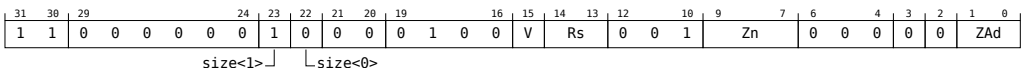
16-bit
(FEAT_SME2)




```
MOVA     ZAd><HV>.H[<Ws>, <offsf>:<offsl>], { <Zn1>.H-<Zn4>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 16;
integer n = UInt(Zn:'00');
integer d = UInt(ZAd);
integer offset = UInt(o1:'00');
boolean vertical = V == '1';
```

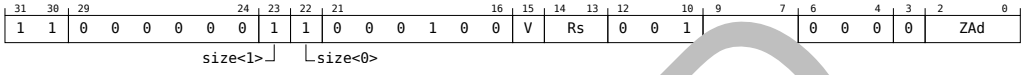
32-bit
(FEAT_SME2)



```
MOVA    <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer d = UInt(ZAd);
integer offset = 0;
boolean vertical = V == '1';
```

64-bit
(FEAT_SME2)




```
MOVA    <ZAd><HV>.D[<Ws>, <offsf>:<offsl>], { <Z  >.D-<Z  >.D }
```

```
if !HaveSME2() then UNDEFINED;
integer s = UInt('011':Rs);
constant integer nreg = 4;
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer d = UInt(ZAd);
integer offset = 0;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAd> For the 16-bit variant: i e name of t ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32-bit v ant: is the na of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.

For the 6 bit vari is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

<HV> I he h zonta vert al slice indicator, encoded in "V":

V	< V>
0	
1	V

<Ws> Is the 3 -bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offsf> For the 8-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "off2" field times 4.

For the 16-bit variant: is the slice index offset, pointing to first of four consecutive slices, encoded as "o1" field times 4.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to first of four consecutive slices, with implicit value 0.

<offsl> For the 8-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "off2" field times 4 plus 3.

For the 16-bit variant: is the slice index offset, pointing to last of four consecutive slices, encoded as "o1" field times 4 plus 3.

For the 32-bit and 64-bit variant: is the slice index offset, pointing to last of four consecutive

slices, with implicit value 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
if nreg == 4 && esize == 64 && VL == 128 then UNDEFINED;
integer slices = VL DIV esize;
bits(32) index = X[s, 32];
integer slice = ((UInt(index) - (UInt(index) MOD nreg)) + offset) MOD slices;

for r = 0 to nreg-1
    bits(VL) result = Z[n + r, VL];
    ZAslice[d, esize, vertical, slice + r, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.127 MOVA (vector to array, two registers)

Move two vector registers to two ZA single-vector groups

The instruction operates on two ZA single-vector groups. The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

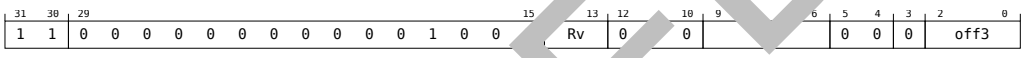
The VECTOR GROUP symbol VGx2 indicates that the instruction operates on two ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to array, two registers).

SME2
(FEAT_SME2)




```
MOVA    ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D, <Zn2>.D }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer n = UInt(Zn:'0');
constant integer nreg = 2;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = Z[n + r, VL];
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Operational information

If PSTATE.DIT is 1:

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

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

D1.1.128 MOVA (vector to array, four registers)

Move four vector registers to four ZA single-vector groups

The instruction operates on four ZA single-vector groups. The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

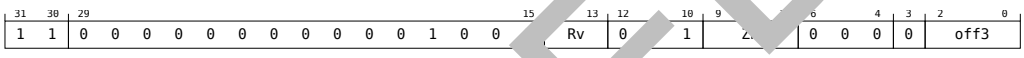
The VECTOR GROUP symbol VGx4 indicates that the instruction operates on four ZA single-vector groups.

The preferred disassembly syntax uses a 64-bit element size, but an assembler should accept any element size if it is used consistently for all operands. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

This instruction is used by the alias MOV (vector to array, four register

SME2
(FEAT_SME2)




```
MOVA    ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.D <Zn4>.D }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
integer offset = UInt(off3);
integer n = UInt(Zn:'00');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" tim

<Zn4> Is the name of the last scalable vector register of a multi-vector sequence, encoded as "Zn" times

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;

for r = 0 to nreg-1
    bits(VL) result = Z[n + r, VL];
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

Operational information

If PSTATE.DIT is 1:

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

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

D1.1.129 MOVA (vector to tile, single)

Move vector register to ZA tile slice

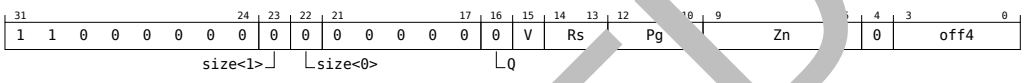
The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of such elements in a vector. The immediate offset is in the range 0 to the number of elements in a 128-bit vector segment minus 1.

Inactive elements in the destination slice remain unmodified.

This instruction is used by the alias MOV (vector to tile, single).

It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

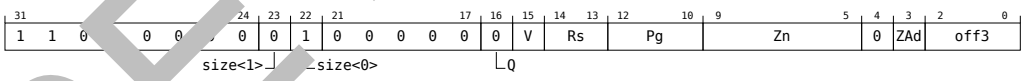
8-bit
(FEAT_SME)




```
MOVA     ZA0<HV>.B[<Ws>, <offs>], <Pg>/M, <Zn>.B
```

```
1  if !HaveSME() then UNDEFINED;
2  integer g = UInt(Pg);
3  integer s = UInt('011':Rs);
4  integer n = UInt(Zn);
5  integer d = 0;
6  integer offset = UInt(off4);
7  constant integer esize = 8;
8  boolean vertical = V == '1';
```

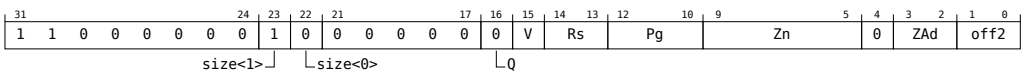
16-bit
(FEAT_SME)




```
MOVA     <ZAd><HV>.H[<Ws>, <offs>], <Pg>/M, <Zn>.H
```

```
1  if !HaveSME() then UNDEFINED;
2  integer g = UInt(Pg);
3  integer s = UInt('011':Rs);
4  integer n = UInt(Zn);
5  integer d = UInt(ZAd);
6  integer offset = UInt(off3);
7  constant integer esize = 16;
8  boolean vertical = V == '1';
```

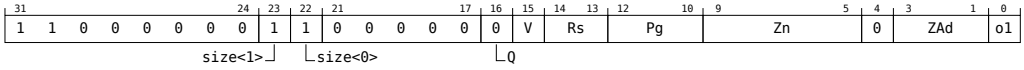
32-bit
(FEAT_SME)




```
MOVA     <ZAd><HV>.S[<Ws>, <offs>], <Pg>/M, <Zn>.S
```

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer offset = UInt(off2);
constant integer esize = 32;
boolean vertical = V == '1';
```

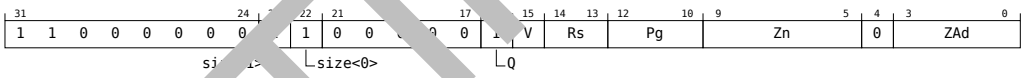
64-bit
(FEAT_SME)




```
MOVA    <ZAd><HV>.D[<Ws>, <offs>], <Pg>/M, <Zn>.D
```

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer offset = UInt(o1);
constant integer esize = 64;
boolean vertical = V == '1';
```

128-bit
(FEAT_SME)




```
MOVA    <ZAd><HV>.Q[<Ws>, <offs>], <Pg>/M, <Zn>.Q
```

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer offset = 0;
constant integer esize = 128;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.

For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.

For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "off4" field.

For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "off3" field.

For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "off2" field.

For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "o1" field.

For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD dim;
bits(VL) result = ZAslice[d, esize, vertical, slice, VL];

for e = 0 to dim-1
    bits(esize) element = Elem[operand, e, esize];
    if ActivePredicateElement(mask, e, esize) then
        Elem[result, e, esize] = element;

ZAslice[d, esize, vertical, slice, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

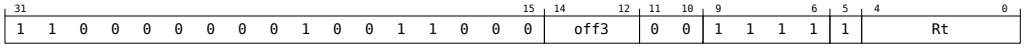
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

D1.1.130 MOVT (ZT0 to scalar)

Move 8 bytes from ZT0 to general-purpose register

Move 8 bytes to a general-purpose register from the ZT0 register at the byte offset specified by the immediate index. This instruction is UNDEFINED in Non-debug state.

SME2
(FEAT_SME2)




```
MOVT    <Xt>, ZT0[<offs>]
```

```
1  if !HaveSME2() || !Halted() then UNDEFINED;
2  integer t = UInt(Rt);
3  integer offset = UInt(off3);
```

Assembler Symbols

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

<offs> Is the immediate byte offset, a multiple of 8 in the range of 0 to 56, encoded in the "off3" field as <offs>/8.

Operation

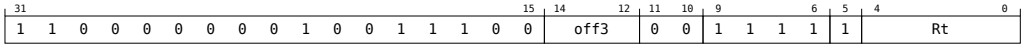
```
1  CheckSMEEnabled();
2  CheckSMEZT0Enabled();
3  bits(512) operand = ZT0[51 :0];
4
5  X[t, 64] = Elem[operand, offset, 64];
```

D1.1.131 MOVT (scalar to ZT0)

Move 8 bytes from general-purpose register to ZT0

Move 8 bytes to the ZT0 register at the byte offset specified by the immediate index from a general-purpose register. This instruction is UNDEFINED in Non-debug state.

SME2
(FEAT_SME2)




```
MOVT    ZT0[<offs>], <Xt>
```

```
if !HaveSME2() || !Halted() then UNDEFINED;
integer t = UInt(Rt);
integer offset = UInt(off3);
```

Assembler Symbols

<offs> Is the immediate byte offset, a multiple of 8 in the range of 0 to 56 encoded in the "off3" field as <offs>/8.

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

Operation

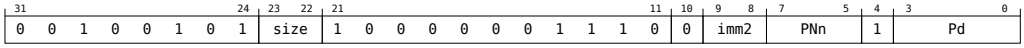
```
CheckSMEEnabled();
CheckSMEZT0Enabled();
bits(512) result = ZT0[512
4
Elem[result, offset, 64] = X[t, 64];
ZT0[512] = result;
```

D1.1.132 PEXT (predicate)

Set predicate from predicate-as-counter

Expands the source predicate-as-counter into a four-predicate wide mask and copies one quarter of it into the destination predicate register.

SME2
(FEAT_SME2)




```
PEXT    <Pd>.<T>, <PNn>[<imm>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt('1':PNn);
integer d = UInt(Pd);
integer part = UInt(imm2);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<PNn> Is the name of the first source scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNn" field.

<imm> Is the element index, in the range 0 to 3, encoded in the "imm2" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) pred = P[n, PL];
bits(PL*4) mask = CounterToPredicate(pred<15:0>, PL*4);
bits(PL) result;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit pbit = PredicateElement(mask, part * elements + e, esize);
    Elem[result, e, psize] = ZeroExtend(pbit, psize);

P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

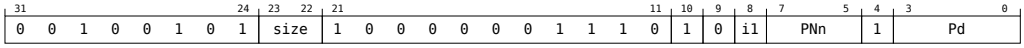
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

D1.1.133 PEXT (predicate pair)

Set pair of predicates from predicate-as-counter

Expands the source predicate-as-counter into a four-predicate wide mask and copies two quarters of it into the destination predicate registers.

SME2
(FEAT_SME2)




```
PEXT    { <Pd1>.<T>, <Pd2>.<T> }, <PNn>[<imm>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt('1':PNn);
integer d0 = UInt(Pd);
integer d1 = (UInt(Pd) + 1) MOD 16;
integer part = UInt(i1);
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded in the "Pd" field.

<PNn> Is the name of the first source scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNn" field.

<imm> Is the element index in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) pred = P[n, PL];
bits(PL*4) mask = CounterToPredicate(pred<15:0>, PL*4);
bits(PL) result0;
bits(PL) result1;
constant integer psize = esize DIV 8;

for e = 0 to elements-1
    bit pbit = PredicateElement(mask, part * 2 * elements + e, esize);
    Elem[result0, e, psize] = ZeroExtend(pbit, psize);

for e = 0 to elements-1
    bit pbit = PredicateElement(mask, part * 2 * elements + elements + e, esize);
    Elem[result1, e, psize] = ZeroExtend(pbit, psize);

P[d0, PL] = result0;
P[d1, PL] = result1;
```

Operational information

If PSTATE.DIT is 1:

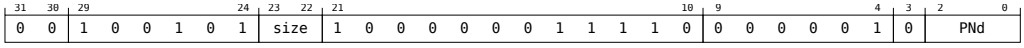
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

D1.1.134 PTRUE

Initialise predicate-as-counter to all active

Set the destination predicate as all-active elements, using the predicate-as-counter encoding.

SME2
(FEAT_SME2)




```
PTRUE   <PNd>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer d = UInt('1':PNd);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8 to PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

Operation

```
1  CheckStreamingSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer elements = VL DIV esize;
5  bits(PL) result = CounterPredicateCount(esize, elements, elements, FALSE, PL);
6  P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

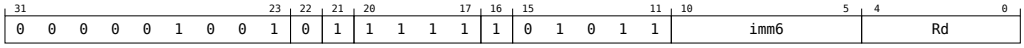
D1.1.135 RDSVL

Read multiple of Streaming SVE vector register size to scalar register

Multiply the Streaming SVE vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.

This instruction does not require the PE to be in Streaming SVE mode.

SME
(FEAT_SME)




```
RDSVL   <Xd>, #<imm>
```

```
1  if !HaveSME() then UNDEFINED;
2  integer d = UInt(Rd);
3  integer imm = SInt(imm6);
```

Assembler Symbols

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

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
1  CheckSMEEnabled();
2  constant integer SVL = CurrentSVL;
3  integer len = imm * (SVL DIV 8);
4  X[d, 64] = len<63:0>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.136 SCLAMP

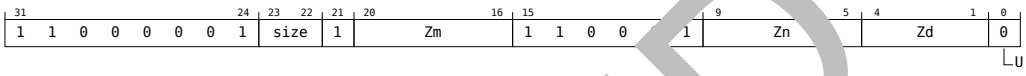
Multi-vector signed clamp to minimum/maximum vector

Clamp each signed element in the two or four destination vectors to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

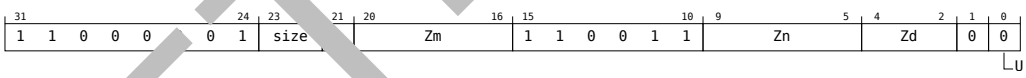
Two registers
(FEAT_SME2)




```
SCLAMP  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer n = UInt(Zn);
4  integer m = UInt(Zm);
5  integer d = UInt(Zd:'0');
6  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
SCLAMP  { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer n = UInt(Zn);
4  integer m = UInt(Zm);
5  integer d = UInt(Zd:'00');
6  constant integer nreg = 4;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = Z[d+r, VL];
    for e = 0 to elements-1
        integer element1 = SInt(Elem[operand1, e, esize]);
        integer element2 = SInt(Elem[operand2, e, esize]);
        integer element3 = SInt(Elem[operand3, e, esize]);
        integer res = Min(Max(element1, element3), element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.137 SCVTF

Multi-vector signed integer convert to floating-point

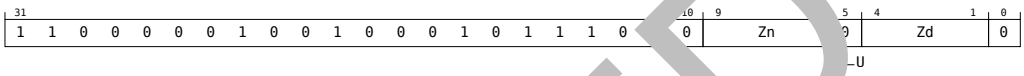
Convert to single-precision from signed 32-bit integer, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

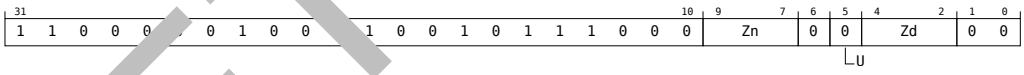
Two registers
(FEAT_SME2)




```
SCVTF   { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Four registers
(FEAT_SME2)




```
SCVTF   { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn:'00');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FixedToFP(element, , unsigned, FPCR[], rounding, 32);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.138 SDOT (2-way, multiple and indexed vector)

Multi-vector signed integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups.

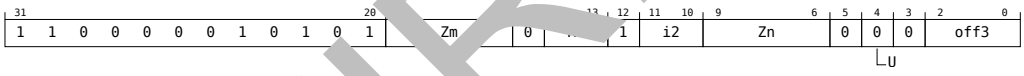
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter th[illegible]ber of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ope[illegible]d consists o[illegible]vo or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for dis[illegible]embly, but op[illegible]onal in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a[illegible] Four [illegible]A single-vectors

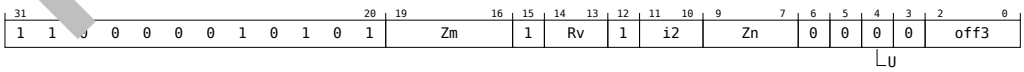
Two ZA single-vectors
(FEAT_SME2)




```
SDOT     ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg
```

Four [illegible]A single-vec[illegible]
(FEAT_[illegible]E2)




```
SDOT     ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z[illegible], encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = [illegible] VL];
    bits(VL) operand2 = Z[m, [illegible]
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer index = segmentbase + index;
        for i = 0 to 1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.139 SDOT (2-way, multiple and single vector)

Multi-vector signed integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

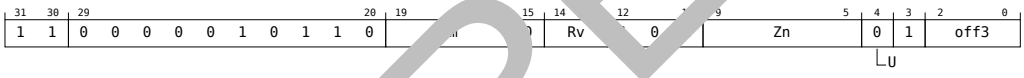
The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser[illegible]t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F[illegible] ZA sing[illegible]vectors

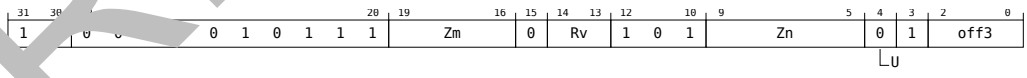
Two ZA single-vectors
(FEAT_SME2)




```
SDOT     ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
SDOT     ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize
        for i = 0 to 1
            integer element1 = SInt(Elem[
            integer element2 = SInt(Ele
            sum = sum + element1 * el
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.140 SDOT (2-way, multiple vectors)

Multi-vector signed integer dot-product

The instruction operates on two or four ZA single-vector groups.

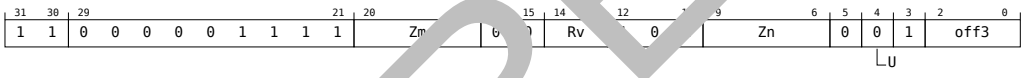
The signed integer dot product instruction computes the dot product of two signed 16-bit integer values held in each 32-bit element of the two or four first source vectors and two signed 16-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser[illegible]t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F[illegible] ZA sing[illegible]vectors

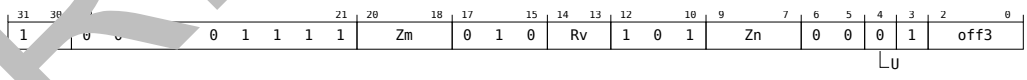
Two ZA single-vectors
(FEAT_SME2)




```
SDOT    ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA [illegible]-vect[illegible]
(FEAT[illegible]ME2)




```
SDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a

multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.141 SDOT (4-way, multiple and indexed vector)

Multi-vector signed integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

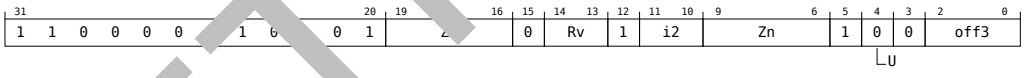
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors of 32-bit elements , Two ZA single-vectors of 64-bit elements , Four ZA single-vectors of 32-bit elements and Four ZA single-vectors of 64-bit elements

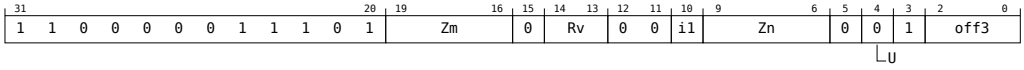
Two ZA single-vectors of 32-bit elements
(FEAT_SME2)




```
SDOT     ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

Two ZA single-vectors of 64-bit elements
(FEAT_SME_I16I64)

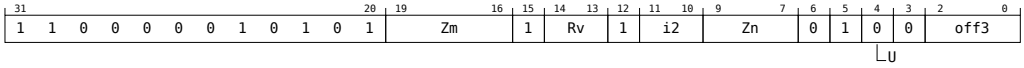



```
SDOT     ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
```

```
integer offset = UInt(off3);
integer index = UInt(i1);
constant integer nreg = 2;
```

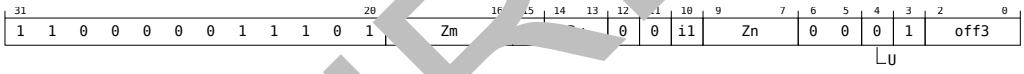
Four ZA single-vectors of 32-bit elements (FEAT_SME2)




```
SDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

Four ZA single-vectors of 64-bit elements (FEAT_SME_I16I64)




```
SDOT    ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors of 32-bit elements and two ZA single-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors of 32-bit elements and four ZA single-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the four ZA single-vectors of 32-bit elements and two ZA single-vectors of 32-bit

elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the four ZA single-vectors of 64-bit elements and two ZA single-vectors of 64-bit elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.142 SDOT (4-way, multiple and single vector)

Multi-vector signed integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

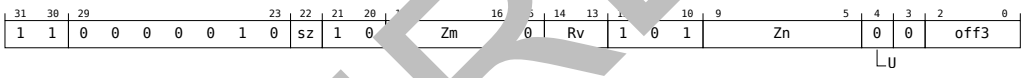
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

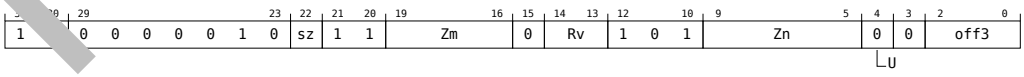
Two ZA single-vectors
(FEAT_SME2)




```
SDOT    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
SDOT    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.143 SDOT (4-way, multiple vectors)

Multi-vector signed integer dot-product

The instruction operates on two or four ZA single-vector groups.

The signed integer dot product instruction computes the dot product of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand [illegible] of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

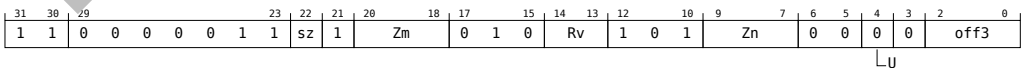
It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors
(FEAT_SME2)


```
SDOT    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm2>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
SDOT    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

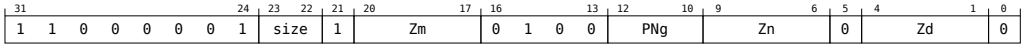
D1.1.144 SEL

Multi-vector conditionally select elements from two vectors

Read active elements from the two or four first source vectors and inactive elements from the two or four second source vectors and place in the corresponding elements of the two or four destination vectors.

It has encodings from 2 classes: Two registers and Four registers

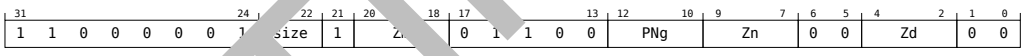
Two registers
(FEAT_SME2)




```
SEL     { <Zd1>.<T>-<Zd2>.<T> }, <PNg>, { <Zn1>.<T>-<Zn2>.<T> }, {
    ↪<Zm1>.<T>-<Zm2>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer d = UInt(Zd:'0');
integer g = UInt('1':PNg);
constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
SEL     { <Zd1>.<T>-<Zd4>.<T> }, <PNg>, { <Zn1>.<T>-<Zn4>.<T> }, {
    ↪<Zm1>.<T>-<Zm4>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer d = UInt(Zd:'00');
integer g = UInt('1':PNg);
constant integer nreg = 4;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence,

encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            Elem[results[r], e, esize] = Elem[operand1, e, esize];
        else
            Elem[results[r], e, esize] = Elem[operand2, e, esize];

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.145 SMAX (multiple and single vector)

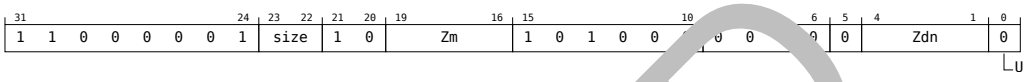
Multi-vector signed maximum by vector

Determine the signed maximum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

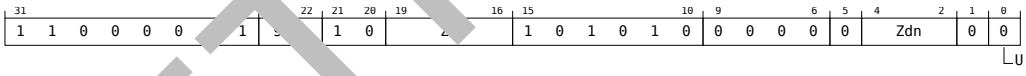
Two registers
(FEAT_SME2)




```
SMAX    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 2;
6  boolean unsigned = FALSE;
```

Four registers
(FEAT_SME2)




```
SMAX    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 4;
6  boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Max(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.146 SMAX (multiple vectors)

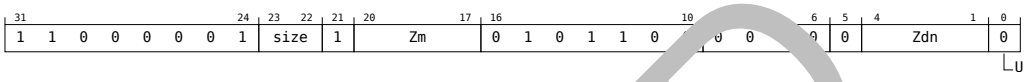
Multi-vector signed maximum

Determine the signed maximum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

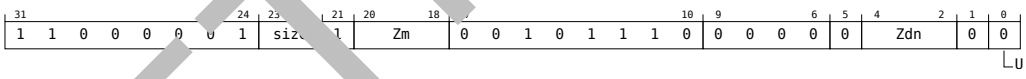
Two registers
(FEAT_SME2)




```
SMAX    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt(Zm:'0');
5  constant integer nreg = 2;
6  boolean unsigned = FALSE;
```

Four registers
(FEAT_SME2)




```
SMAX    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'00');
4  integer m = UInt(Zm:'00');
5  constant integer nreg = 4;
6  boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Max(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.147 SMIN (multiple and single vector)

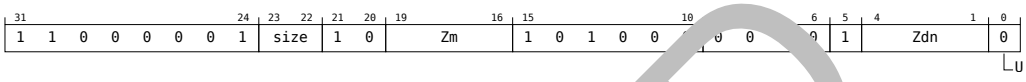
Multi-vector signed minimum by vector

Determine the signed minimum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

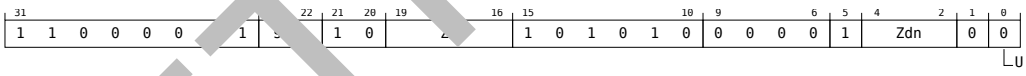
Two registers
(FEAT_SME2)




```
SMIN    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T>, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 2;
6  boolean unsigned = FALSE;
```

Four registers
(FEAT_SME2)




```
SMIN    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 4;
6  boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Min(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.148 SMIN (multiple vectors)

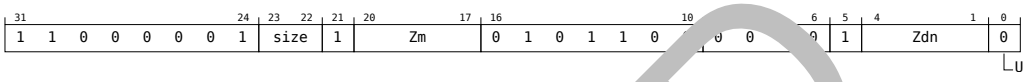
Multi-vector signed minimum

Determine the signed minimum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

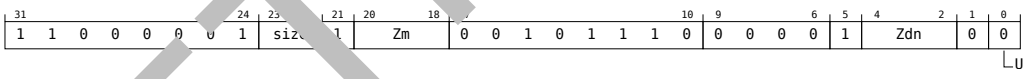
Two registers
(FEAT_SME2)




```
SMIN    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt(Zm:'0');
5  constant integer nreg = 2;
6  boolean unsigned = FALSE;
```

Four registers
(FEAT_SME2)




```
SMIN    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'00');
4  integer m = UInt(Zm:'00');
5  constant integer nreg = 4;
6  boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Min(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.149 SMLAL (multiple and indexed vector)

Multi-vector signed integer multiply-add long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This signed integer multiply-add long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA double-vector groups.

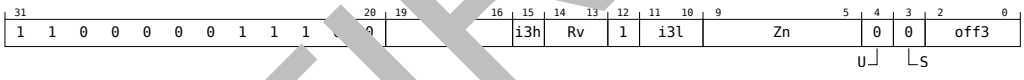
The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and the immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

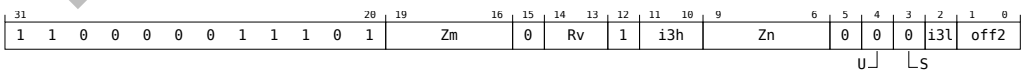
It has encodings from 3 classes: One ZA double-vector , Two ZA double-vectors and Four ZA double-vectors

One ZA double-vector
(FEAT_SME2)



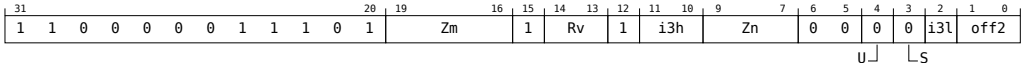

```
SMLAL   ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

Two ZA double-vectors
(FEAT_SME2)




```
SMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)




```
SMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, s, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.150 SMLAL (multiple and single vector)

Multi-vector signed integer multiply-add long by vector

The instruction operates on one, two, or four ZA double-vector groups.

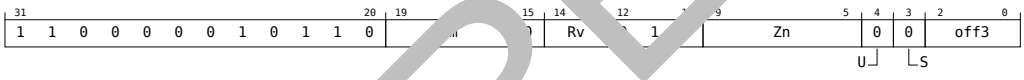
This signed integer multiply-add long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector , Two [illegible] double-[illegible]tors and [illegible]our ZA double-vectors

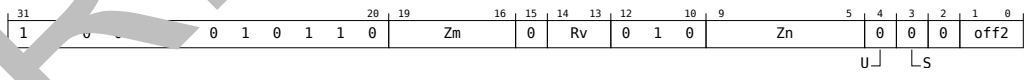
One ZA double-vector
(FEAT_SME2)




```
SMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn);
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off3:'0');
7  constant integer nreg = 2;
```

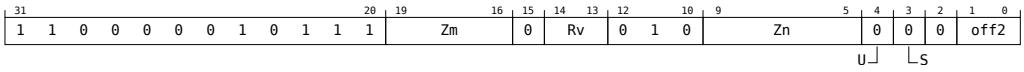
Two ZA [illegible]-vectors
(FEAT_SME2)




```
SMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn);
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off2:'0');
7  constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)



```
SMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.151 SMLAL (multiple vectors)

Multi-vector signed integer multiply-add long

The instruction operates on two or four ZA double-vector groups.

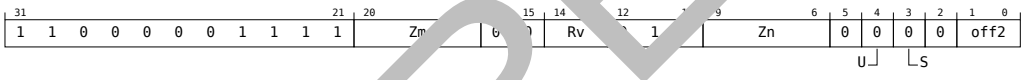
This signed integer multiply-add long instruction multiplies each signed 16-bit element in the two or four first source vectors with each signed 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and F[illegible]r ZA dou[illegible]e-vector

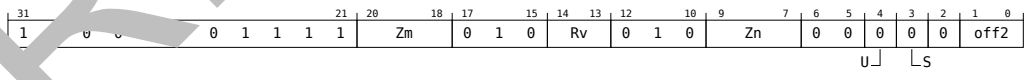
Two ZA double-vectors
(FEAT_SME2)




```
SMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)




```
SMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec DIV 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.152 SMLALL (multiple and indexed vector)

Multi-vector signed integer multiply-add long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This signed integer multiply-add long long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all, half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

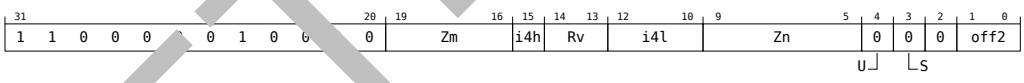
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: One ZA quad-vector of 32-bit elements , One ZA quad-vector of 64-bit elements , Two ZA quad-vectors of 32-bit elements , Two ZA quad-vectors of 64-bit elements , Four ZA quad-vectors of 32-bit elements and Four ZA quad-vectors of 64-bit elements

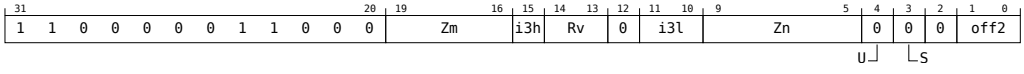
One ZA quad-vector of 32-bit elements
(FEAT_SME2)




```
SMLALL  ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.B, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('0':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

One ZA quad-vector of 64-bit elements
(FEAT_SME_I16I64)

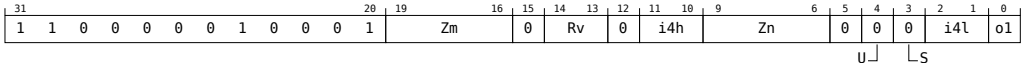



```
SMLALL  ZA.D[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
```

```
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

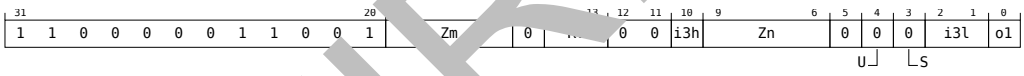
Two ZA quad-vectors of 32-bit elements (FEAT_SME2)




```
SMLALL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

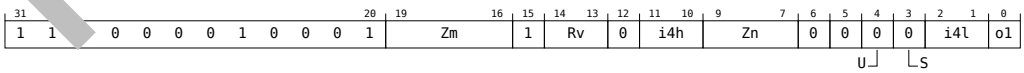
Two ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)




```
SMLALL  ZA.D[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

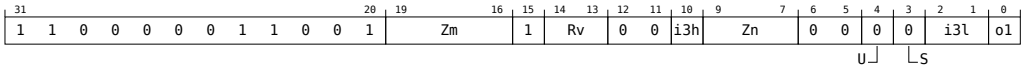
Four ZA quad-vectors of 32-bit elements (FEAT_SME2)




```
SMLALL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

Four ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)




```
SMLALL  ZA.D[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
1  if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
2  constant integer esize = 64;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn:'00');
5  integer m = UInt('0':Zm);
6  integer offset = UInt(o1:'00');
7  integer index = UInt(i3h:i3l);
8  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4 plus

For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 4 * segmentbase + index;
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, s, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.153 SMLALL (multiple and single vector)

Multi-vector signed integer multiply-add long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This signed integer multiply-add long long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

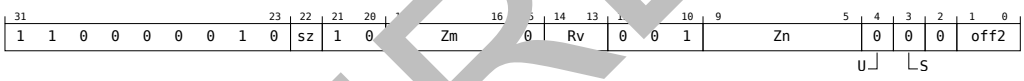
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: One ZA quad-vector , Two ZA quad-vectors and Four ZA quad-vectors

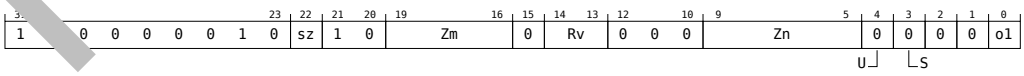
One ZA quad-vector
(FEAT_SME2)




```
SMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], <Zn>.<Tb>, <Zm>.<Tb>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3  constant integer esize = 32 << UInt(sz);
4  integer v = UInt('010':Rv);
5  integer n = UInt(Zn);
6  integer m = UInt('0':Zm);
7  integer offset = UInt(off2:'0');
8  constant integer nreg = 1;
```

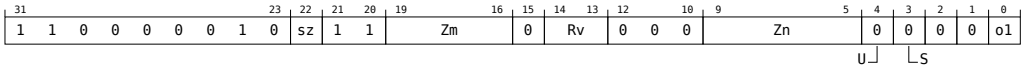
Two ZA quad-vectors
(FEAT_SME2)




```
SMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3  constant integer esize = 32 << UInt(sz);
4  integer v = UInt('010':Rv);
5  integer n = UInt(Zn);
6  integer m = UInt('0':Zm);
7  integer offset = UInt(o1:'00');
8  constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
SMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in "sz":

sz	
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.154 SMLALL (multiple vectors)

Multi-vector signed integer multiply-add long long

The instruction operates on two or four ZA quad-vector groups.

This signed integer multiply-add long long instruction multiplies each signed 8-bit or 16-bit element in the two or four first source vectors with each signed 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

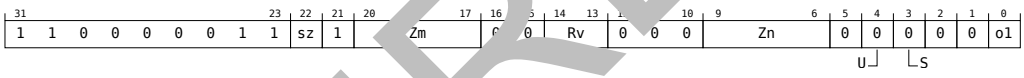
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA quad-vectors and four ZA quad-vectors

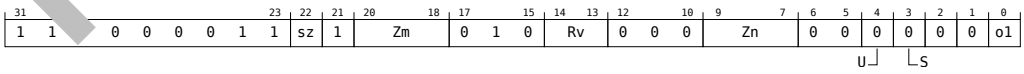
Two ZA quad-vectors
(FEAT_SME2)




```
SMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm2>.<Tb> }
```

```
1  if !HaveSME2() then UNDEFINED;
2  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3  constant integer esize = 32 << UInt(sz);
4  integer v = UInt('010':Rv);
5  integer n = UInt(Zn:'0');
6  integer m = UInt(Zm:'0');
7  integer offset = UInt(o1:'0');
8  constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
SMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm4>.<Tb> }
```

```
1  if !HaveSME2() then UNDEFINED;
2  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3  constant integer esize = 32 << UInt(sz);
4  integer v = UInt('010':Rv);
5  integer n = UInt(Zn:'00');
6  integer m = UInt(Zm:'00');
7  integer offset = UInt(o1:'00');
8  constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
```

```
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.155 SMLSL (multiple and indexed vector)

Multi-vector signed integer multiply-subtract long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the one, two, or four ZA double-vector groups.

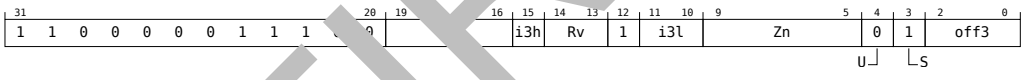
The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

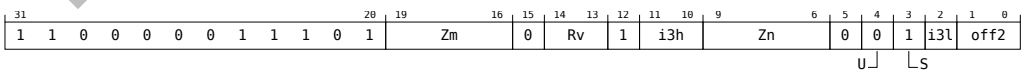
It has encodings from 3 classes: One ZA double-vector , Two ZA double-vectors and Four ZA double-vectors

One ZA double-vector
(FEAT_SME2)



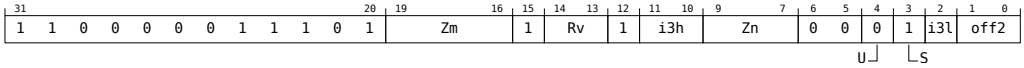

```
SMLSL   ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

Two ZA double-vectors
(FEAT_SME2)




```
SMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)




```
SMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, s, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.156 SMLSL (multiple and single vector)

Multi-vector signed integer multiply-subtract long by vector

The instruction operates on one, two, or four ZA double-vector groups.

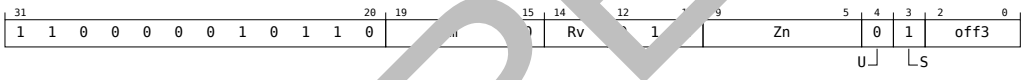
This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the one, two, or four first source vectors with each signed 16-bit element in the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector , Two [illegible] double-[illegible]tors and [illegible]our ZA double-vectors

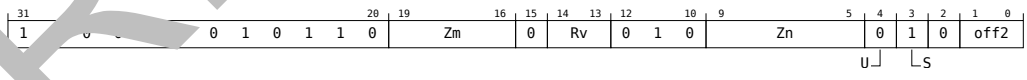
One ZA double-vector
(FEAT_SME2)




```
SMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn);
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off3:'0');
7  constant integer nreg = 2;
```

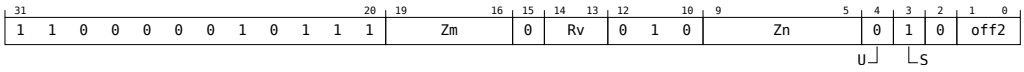
Two ZA [illegible]-vectors
(FEAT_SME2)




```
SMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn);
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off2:'0');
7  constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)



```
SMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant, is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant, is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.157 SMLSL (multiple vectors)

Multi-vector signed integer multiply-subtract long

The instruction operates on two or four ZA double-vector groups.

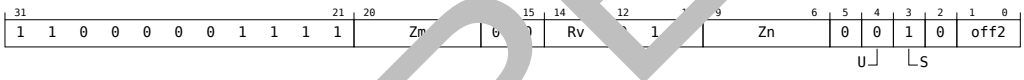
This signed integer multiply-subtract long instruction multiplies each signed 16-bit element in the two or four first source vectors with each signed 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and F[illegible]r ZA dou[illegible]e-vector

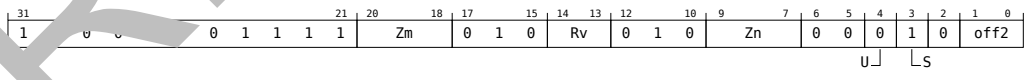
Two ZA double-vectors
(FEAT_SME2)




```
SMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)




```
SMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec DIV 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.158 SMLSLL (multiple and indexed vector)

Multi-vector signed integer multiply-subtract long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This signed integer multiply-subtract long long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all, half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

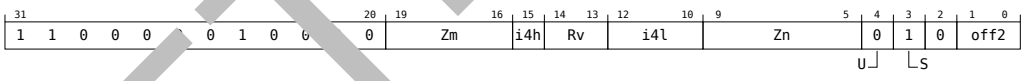
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: One ZA quad-vector of 32-bit elements , One ZA quad-vector of 64-bit elements , Two ZA quad-vectors of 32-bit elements , Two ZA quad-vectors of 64-bit elements , Four ZA quad-vectors of 32-bit elements and Four ZA quad-vectors of 64-bit elements

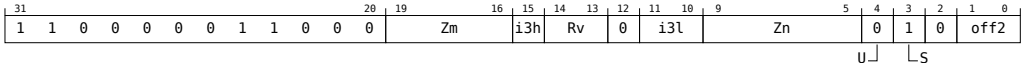
One ZA quad-vector of 32-bit elements
(FEAT_SME2)




```
SMLSLL  ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.B, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('0':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

One ZA quad-vector of 64-bit elements
(FEAT_SME_I16I64)

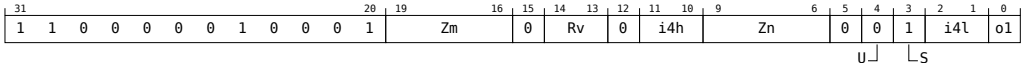



```
SMLSLL  ZA.D[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
```

```
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

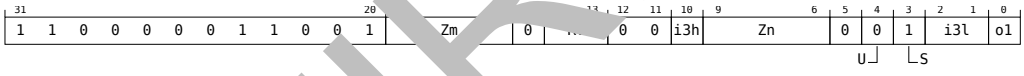
Two ZA quad-vectors of 32-bit elements (FEAT_SME2)




```
SMLSLL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

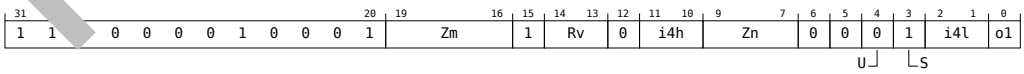
Two ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)




```
SMLSLL  ZA.D[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

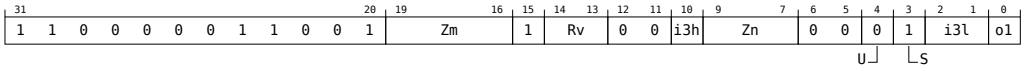
Four ZA quad-vectors of 32-bit elements (FEAT_SME2)




```
SMLSLL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

Four ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)




```
SMLSLL  ZA.D[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
1  if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
2  constant integer esize = 64;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn:'00');
5  integer m = UInt('0':Zm);
6  integer offset = UInt(o1:'00');
7  integer index = UInt(i3h:i3l);
8  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4 plus

For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegme
            integer s = 4 * segmentbase + index;
            integer element1 = SInt(Elem[operand1, 4
            integer element2 = SInt(Elem[operand2,
            bits(esize) product = (element1 * el
            Elem[result, e, esize] = Elem[opera
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.159 SMLSLL (multiple and single vector)

Multi-vector signed integer multiply-subtract long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This signed integer multiply-subtract long long instruction multiplies each signed 8-bit or 16-bit element in the one, two, or four first source vectors with each signed 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

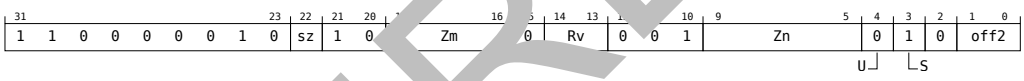
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disasse[illegible]t optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit int[illegible]er varian[illegible] implem[illegible]ted.

It has encodings from 3 classes: One ZA quad-vector , Tw[illegible] ZA quad[illegible]ectors a[illegible]our ZA quad-vectors

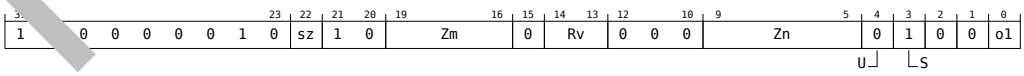
One ZA quad-vector
(FEAT_SME2)




```
SMLSLL  ZA.<T>[<Wv>, <of[illegible]t>:<offsl>[illegible] <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !Ha[illegible]SMEI16I64() [illegible]hen UNDEFINED;
constant integer [illegible]ize = 32 << UIn[illegible]sz);
integer v = UInt[illegible]010':Rv);
integer n = UI[illegible](Zn);
integer m = UIn[illegible]'0'[illegible]m);
integer offset = [illegible](off2:'[illegible]');
constant [illegible]er ni[illegible] = 1;
```

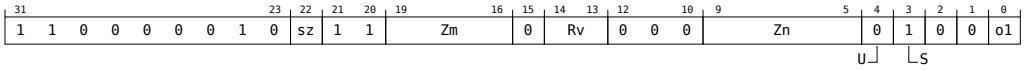
Tw[illegible]A quad[illegible]ectors
(FEA[illegible]SM[illegible]2)




```
SMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
SMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in "sz":

sz	
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.160 SMLSLL (multiple vectors)

Multi-vector signed integer multiply-subtract long long

The instruction operates on two or four ZA quad-vector groups.

This signed integer multiply-subtract long long instruction multiplies each signed 8-bit or 16-bit element in the two or four first source vectors with each signed 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

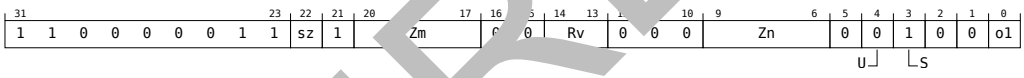
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA quad-vectors and four ZA quad-vectors

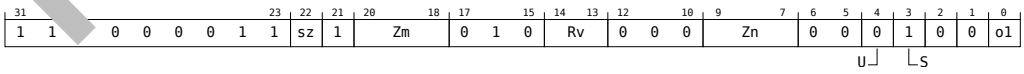
Two ZA quad-vectors
(FEAT_SME2)




```
SMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm2>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
SMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
```

```
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.161 SMOPA (2-way)

Signed integer sum of outer products and accumulate

This instruction works with a 32-bit element ZA tile.

The signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of signed 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive [illegible]umn element[illegible] of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 [illegible]nsecutive ro[illegible] elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2
(FEAT_SME2)


```
SMOPA   <ZAda>.S, <Pn>/M, <P >/M, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEF ;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt( );
integer m = UInt( m);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean unsigned = FALSE;
```

Assembler Symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
```

```
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 1
            if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
                    ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
                prod = (Int(Elem[operand1, 2*row + k, esize DIV 2], unsigned) *
                        Int(Elem[operand2, 2*col + k, esize DIV 2], unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

D1.1.162 SMOPA (4-way)

Signed integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

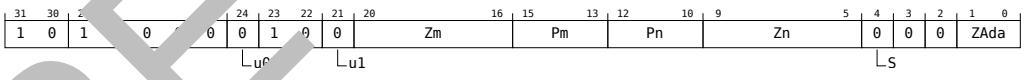
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

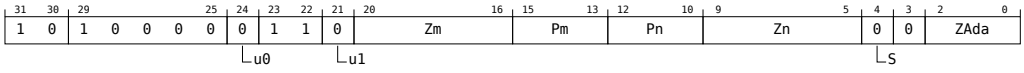
32-bit
(FEAT_SME)




```
SMOPA  <ZA>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

64-bit
(FEAT_SME_I16I64)



```
SMOPA   <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
                ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.

D1.1.163 SMOPS (2-way)

Signed integer sum of outer products and subtract

This instruction works with a 32-bit element ZA tile.

The signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of signed 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively subtracted from the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2
(FEAT_SME2)


```
SMOPS   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer m = UInt(Zm);
7  integer da = UInt(ZAda);
8  boolean sub_op = TRUE;
9  boolean unsigned = FALSE;
```

Assembler Symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer dim = VL DIV esize;
5  bits(PL) mask1 = P[a, PL];
6  bits(PL) mask2 = P[b, PL];
7  bits(VL) operand1 = Z[n, VL];
8  bits(VL) operand2 = Z[m, VL];
9  bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
10 bits(dim*dim*esize) result;
```

```
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 1
            if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
                    ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
                prod = (Int(Elem[operand1, 2*row + k, esize DIV 2], unsigned) *
                        Int(Elem[operand2, 2*col + k, esize DIV 2], unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

D1.1.164 SMOPS (4-way)

Signed integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

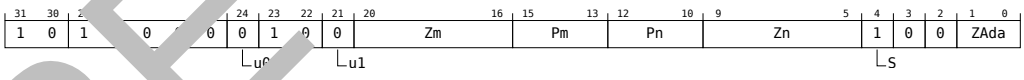
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

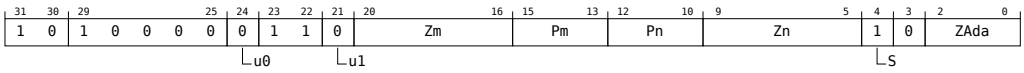
32-bit
(FEAT_SME)




```
SMOPS  <Zn>.<T>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

```
1  if !HaveSME() then UNDEFINED;
2  constant integer esize = 32;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer m = UInt(Zm);
7  integer da = UInt(ZAda);
8  boolean sub_op = TRUE;
9  boolean op1_unsigned = FALSE;
10 boolean op2_unsigned = FALSE;
```

64-bit
(FEAT_SME_I16I64)



```
SMOPS   <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
                ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.

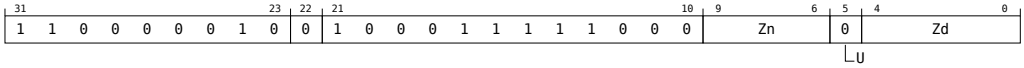
D1.1.165 SQCVT (two registers)

Multi-vector signed saturating extract narrow

Saturate the signed integer value in each element of the two source vectors to half the orginal source element width, and place the results in the half-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQCVT   <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        integer element = SInt(Elem[operand, e, 2 * esize]);
        Elem[result, r*elements + e, esize] = SignedSat(element, esize);

Z[d, VL] = result;
```

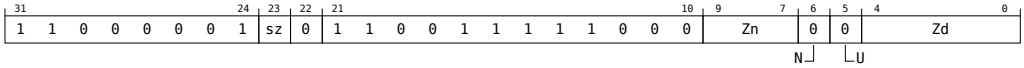
D1.1.166 SQCVT (four registers)

Multi-vector signed saturating extract narrow

Saturate the signed integer value in each element of the four source vectors to quarter the orginal source element width, and place the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQCVT   <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        integer element = SInt(Elem[operand, e, 4 * esize]);
        Elem[result, r*elements + e, esize] = SignedSat(element, esize);

Z[d, VL] = result;
```

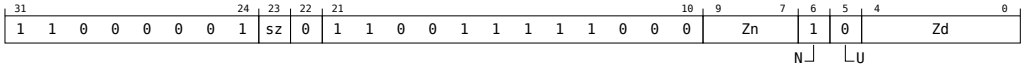
D1.1.167 SQCVTN

Multi-vector signed saturating extract narrow and interleave

Saturate the signed integer value in each element of the four source vectors to quarter the orginal source element width, and place the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQCVTN  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        integer element = SInt(Elem[operand, e, 4 * esize]);
        Elem[result, 4*e + i, esize] = SignedSat(element, esize);

Z[d, VL] = result;
```

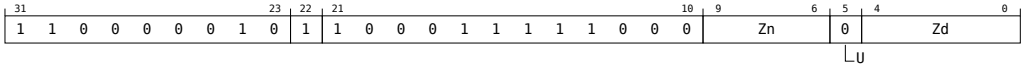
D1.1.168 SQCVTU (two registers)

Multi-vector signed saturating unsigned extract narrow

Saturate the signed integer value in each element of the two source vectors to unsigned integer value that is half the orginal source element width, and place the results in the half-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQCVTU  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        integer element = SInt(Elem[operand, e, 2 * esize]);
        Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

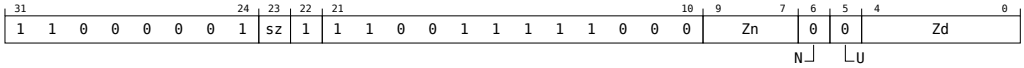
D1.1.169 SQCVTU (four registers)

Multi-vector signed saturating unsigned extract narrow

Saturate the signed integer value in each element of the four source vectors to unsigned integer value that is quarter the orginal source element width, and place the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQCVTU  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        integer element = SInt(Elem[operand, e, 4 * esize]);
        Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

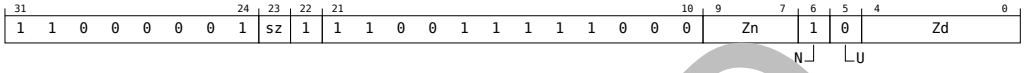
D1.1.170 SQCVTUN

Multi-vector signed saturating unsigned extract narrow and interleave

Saturate the signed integer value in each element of the four source vectors to unsigned integer value that is quarter the orginal source element width, and place the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQCVTUN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        integer element = SInt(Elem[operand, e, 4 * esize]);
        Elem[result, 4*e + i, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

D1.1.171 SQDMULH (multiple and single vector)

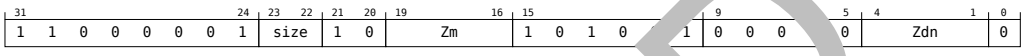
Multi-vector signed saturating doubling multiply high by vector

Multiply then double the corresponding signed elements of the two or four first source vectors and the signed elements of the second source vector, and destructively place the most significant half of the result in the corresponding elements of the two or four first source vectors. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

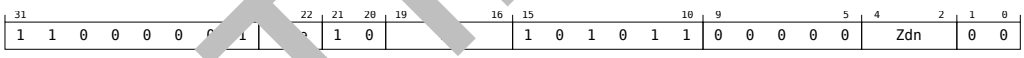
Two registers
(FEAT_SME2)




```
SQDMULH { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
SQDMULH { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'00');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = SInt(Elem[operand1, e, esize]);
        integer element2 = SInt(Elem[operand2, e, esize]);
        integer res = 2 * element1 * element2;
        Elem[results[r], e, esize] = SignedSat(res >> esize, esize);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.172 SQDMULH (multiple vectors)

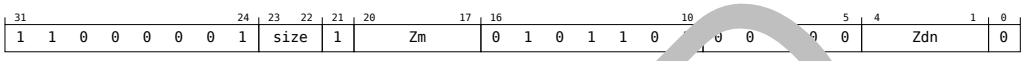
Multi-vector signed saturating doubling multiply high

Multiply then double the corresponding signed elements of the two or four first and second source vectors, and destructively place the most significant half of the result in the corresponding elements of the two or four first source vectors. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

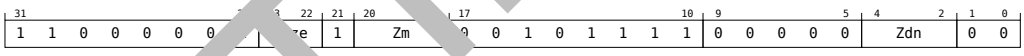
Two registers
(FEAT_SME2)




```
SQDMULH { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1 if !HaveSME2() then UNDEFINED;
2 constant integer esize = 8 << UInt(size);
3 integer dn = UInt(Zdn:'0');
4 integer m = UInt(Zm:'0');
5 constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
SQDMULH { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1 if !HaveSME2() then UNDEFINED;
2 constant integer esize = 8 << UInt(size);
3 integer dn = UInt(Zdn:'00');
4 integer m = UInt(Zm:'00');
5 constant integer nreg = 4;
```

Assembler symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = SInt(Elem[operand1, e, esize]);
        integer element2 = SInt(Elem[operand2, e, esize]);
        integer res = 2 * element1 * element2;
        Elem[results[r], e, esize] = SignedSat(res >> esize, esize);

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

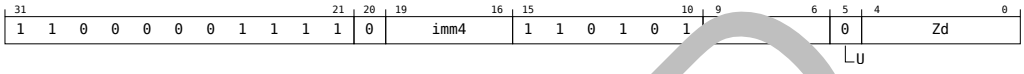
D1.1.173 SQRSHR (two registers)

Multi-vector signed saturating rounding shift right narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQRSHR  <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 16;
3  integer n = UInt(Zn:'0');
4  integer d = UInt(Zd);
5  integer shift = esize - UInt(imm4);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
1  CheckStreamingSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer elements = VL DIV (2 * esize);
4  bits(VL) result;
5  integer round_const = 1 << (shift-1);
6
7  for r = 0 to 1
8      bits(VL) operand = Z[n+r, VL];
9      for e = 0 to elements-1
10         bits(2 * esize) element = Elem[operand, e, 2 * esize];
11         integer res = (SInt(element) + round_const) >> shift;
12         Elem[result, r*elements + e, esize] = SignedSat(res, esize);
13
14 Z[d, VL] = result;
```

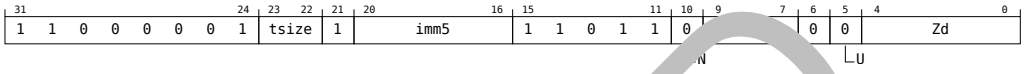
D1.1.174 SQRSHR (four registers)

Multi-vector signed saturating rounding shift right narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQRSHR  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer esize;
3  case tsize of
4      when '00' UNDEFINED;
5      when '01' esize = 8;
6      when '1x' esize = 16;
7  integer n = UInt(Zn:'00');
8  integer d = UInt(Zd);
9  integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	S
1x	D

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

Operation

```
1  CheckStreamingSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer elements = VL DIV (4 * esize);
```

```
bits(VL) result;
integer round_const = 1 << (shift-1);

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (SInt(element) + round_const) >> shift;
        Elem[result, r*elements + e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

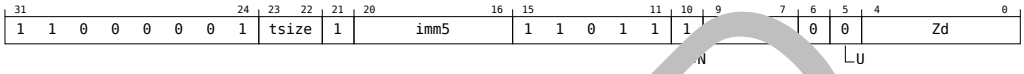
D1.1.175 SQRSHRN

Multi-vector signed saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQRSHRN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

```
if !HaveSME2() then UNDEFINED;
integer esize;
case tsize of
    when '00' UNDEFINED;
    when '01' esize = 8;
    when '1x' esize = 16;
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	S
1x	D

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
```

```
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (SInt(element) + round_const) >> shift;
        Elem[result, 4*e + i, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

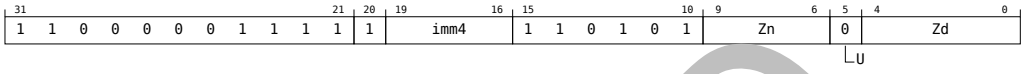
D1.1.176 SQRSHRU (two registers)

Multi-vector signed saturating rounding shift right unsigned narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQRSHRU <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
integer shift = esize - UInt(imm4);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;
integer round_const = 1 << (shift-1);

for r = 0 to 1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(2 * esize) element = Elem[operand, e, 2 * esize];
        integer res = (SInt(element) + round_const) >> shift;
        Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

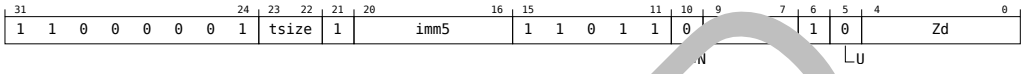
D1.1.177 SQRSHRU (four registers)

Multi-vector signed saturating rounding shift right unsigned narrow by immediate

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQRSHRU <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer esize;
3  case tsize of
4      when '00' UNDEFINED;
5      when '01' esize = 8;
6      when '1x' esize = 16;
7  integer n = UInt(Zn:'00');
8  integer d = UInt(Zd);
9  integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

Operation

```
1  CheckStreamingSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer elements = VL DIV (4 * esize);
```

```
bits(VL) result;
integer round_const = 1 << (shift-1);

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (SInt(element) + round_const) >> shift;
        Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

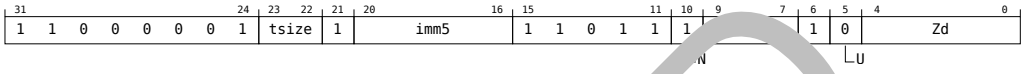
D1.1.178 SQRSHRUN

Multi-vector signed saturating rounding shift right unsigned narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SQRSHRUN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

```
if !HaveSME2() then UNDEFINED;
integer esize;
case tsize of
    when '00' UNDEFINED;
    when '01' esize = 8;
    when '1x' esize = 16;
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	S
1x	D

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
```

```
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (SInt(element) + round_const) >> shift;
        Elem[result, 4*e + i, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

D1.1.179 SRSHL (multiple and single vector)

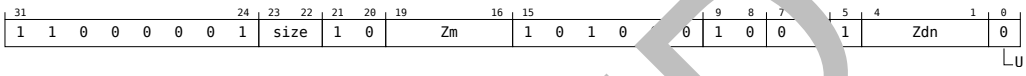
Multi-vector signed rounding shift left by vector

Shift active signed elements of the two or four first source vectors by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

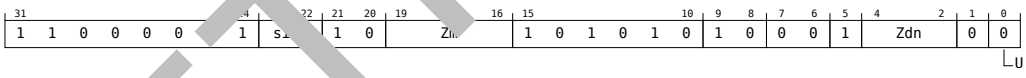
Two registers
(FEAT_SME2)




```
SRSHL   { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
SRSHL   { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element =  SInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, es  ]), esize);
        integer res;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >>  ift;
        Elem[results[r], e, esize] = res<  1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.180 SRSHL (multiple vectors)

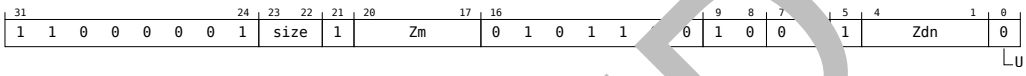
Multi-vector signed rounding shift left

Shift active signed elements of the two or four first source vectors by corresponding elements of the two or four second source vectors and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

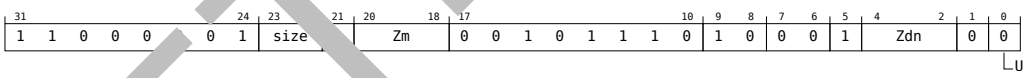
Two registers
(FEAT_SME2)




```
SRSHL   { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt(Zm:'0');
5  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
SRSHL   { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'00');
4  integer m = UInt(Zm:'00');
5  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element =  SInt(Elem[op  an  , e, es  e]);
        integer shift = ShiftSat( nt(Ele  perand2, e, esize]), esize);
        integer res;
        if shift >= 0 then
            res = element  shift;
        else
            shift = -sh ft;
            res = (e ement + (1  (shift - 1))) >> shift;
        Elem[resul  [r], e, esize]  res<esize-1:0>;

for r = 0 to n g-1
    Z[dn+r, VL]  res ts[r];
```

D1.1.181 ST1B (scalar plus immediate, consecutive registers)

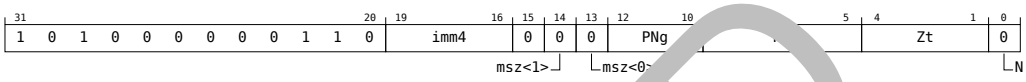
Contiguous store of bytes from multiple consecutive vectors (immediate index)

Contiguous store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

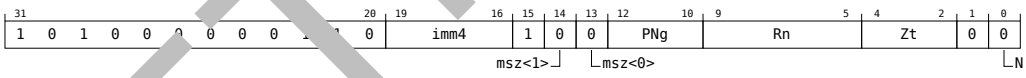
Two registers
(FEAT_SME2)




```
ST1B    { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 8;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
ST1B    { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 8;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.182 ST1B (scalar plus scalar, consecutive registers)

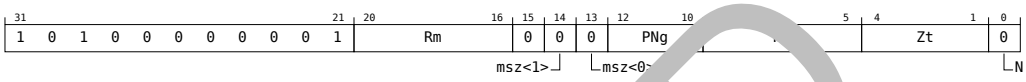
Contiguous store of bytes from multiple consecutive vectors (scalar index)

Contiguous store of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

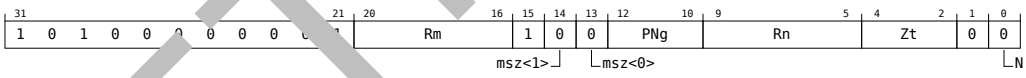
Two registers
(FEAT_SME2)




```
ST1B    { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 8;
```

Four registers
(FEAT_SME2)




```
ST1B    { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 8;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.183 ST1B (scalar plus immediate, strided registers)

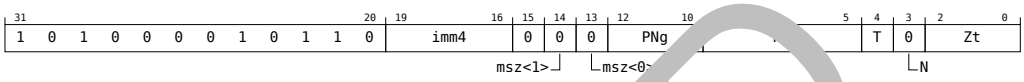
Contiguous store of bytes from multiple strided vectors (immediate index)

Contiguous store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

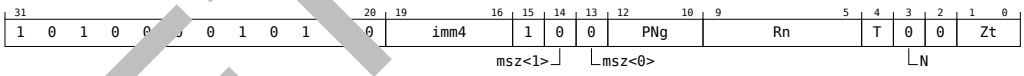
Two registers
(FEAT_SME2)




```
ST1B    { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>{, <imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer tstride = 8;
6  integer t = UInt(T:'0':Zt);
7  constant integer esize = 8;
8  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
ST1B    { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer tstride = 4;
6  integer t = UInt(T:'00':Zt);
7  constant integer esize = 8;
8  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<PL-1:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ElemPredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t+tstride) MOD 32;
```

D1.1.184 ST1B (scalar plus scalar, strided registers)

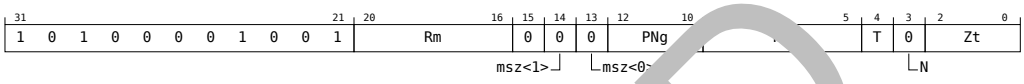
Contiguous store of bytes from multiple strided vectors (scalar index)

Contiguous store of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

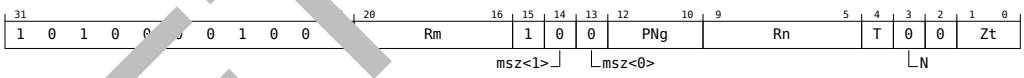
Two registers
(FEAT_SME2)




```
ST1B    { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 8;
```

Four registers
(FEAT_SME2)




```
ST1B    { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 8;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

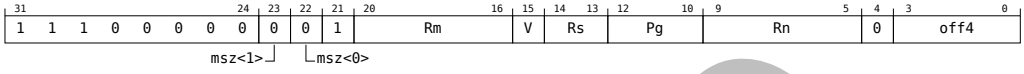
```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(n<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(memop, nontemporal, contiguous,
    tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t + 1) MOD 32;
```

D1.1.185 ST1B (scalar plus scalar, tile slice)

Contiguous store of bytes from 8-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 8-bit elements in a vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)




```
ST1B    { ZA0<HV>.B[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>}]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = 0;
integer offset = UInt(off4);
constant integer esize = 8;
boolean vertical = V == '1';
```

Assembler Symbols

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, in the range 0 to 15, encoded in the "off4" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
```

```
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

D1.1.186 ST1D (scalar plus immediate, consecutive registers)

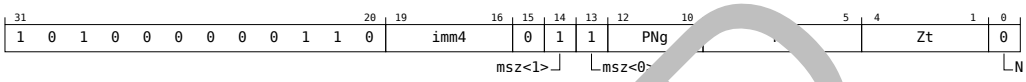
Contiguous store of doublewords from multiple consecutive vectors (immediate index)

Contiguous store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

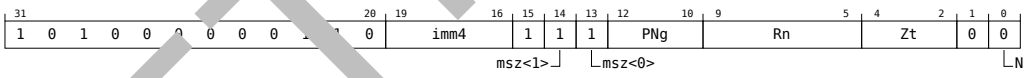
Two registers
(FEAT_SME2)




```
ST1D    { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 64;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
ST1D    { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 64;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.187 ST1D (scalar plus scalar, consecutive registers)

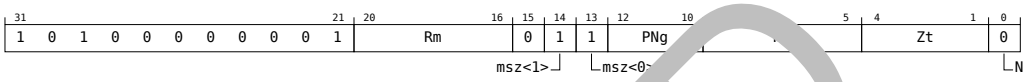
Contiguous store of doublewords from multiple consecutive vectors (scalar index)

Contiguous store of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

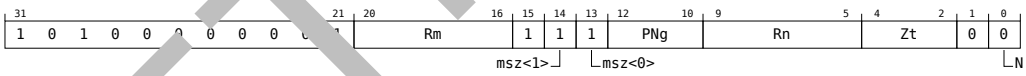
Two registers
(FEAT_SME2)




```
ST1D    { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 64;
```

Four registers
(FEAT_SME2)




```
ST1D    { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 64;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.188 ST1D (scalar plus immediate, strided registers)

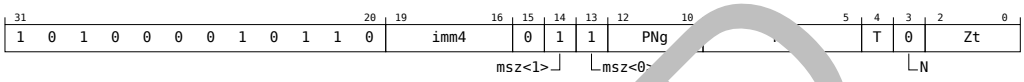
Contiguous store of doublewords from multiple strided vectors (immediate index)

Contiguous store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

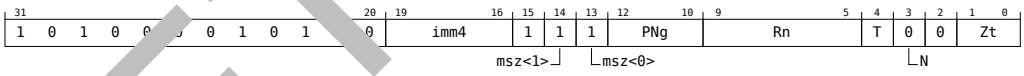
Two registers
(FEAT_SME2)




```
ST1D    { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>{, <imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
ST1D    { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<PL*nreg-1:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ElemPredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t+tstride) MOD 32;
```

D1.1.189 ST1D (scalar plus scalar, strided registers)

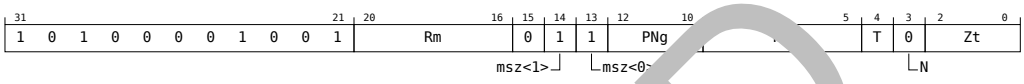
Contiguous store of doublewords from multiple strided vectors (scalar index)

Contiguous store of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

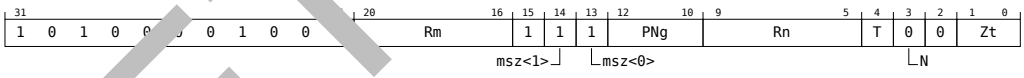
Two registers
(FEAT_SME2)




```
ST1D    { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 64;
```

Four registers
(FEAT_SME2)




```
ST1D    { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 64;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

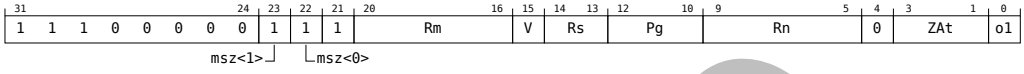
```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(n<15:0> PL * nr );
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDe SVE( mop , nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, e e) th
    if n == 31 && Constrain predictab Bool(U predictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment(
else
    if n == 31 then CheckSPAlig ent();
    base = if n = then SP[] e e X[n, 64];
    offset = X[m 4];

for r = 0 to n q-1
    src = Z[t, V ;
    for e 0 to e ments-1
        if A ivePre cat lement(mask, r * elements + e, esize) then
            b s(64) a = base + (UInt(offset) + r * elements + e) * mbytes;
            m[addr, mbytes, accdesc] = Elem[src, e, esize];
    t
```

D1.1.190 ST1D (scalar plus scalar, tile slice)

Contiguous store of doublewords from 64-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 64-bit elements in a vector. The immediate offset is in the range 0 to 1. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 8 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)




```
ST1D    { <ZAt><HV>.D[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #3}]
```

```
1  if !HaveSME() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('0':Pg);
5  integer s = UInt('011':Rs);
6  integer t = UInt(ZAt);
7  integer offset = UInt(o1);
8  constant integer esize = 64;
9  boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile, ZA0-ZA7 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, in the range 0 to 1, encoded in the "o1" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer dim = VL DIV esize;
5  bits(64) base;
6  bits(64) addr;
7  bits(PL) mask = P[g, PL];
8  bits(64) moffs = X[m, 64];
9  bits(32) index = X[s, 32];
10 integer  slice = (UInt(index) + offset) MOD dim;
11 bits(VL) src;
12 constant integer mbytes = esize DIV 8;
```

```
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

D1.1.191 ST1H (scalar plus immediate, consecutive registers)

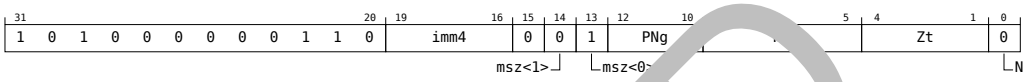
Contiguous store of halfwords from multiple consecutive vectors (immediate index)

Contiguous store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

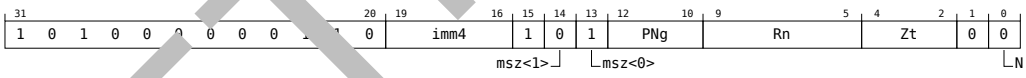
Two registers
(FEAT_SME2)




```
ST1H    { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 16;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
ST1H    { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 16;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.192 ST1H (scalar plus scalar, consecutive registers)

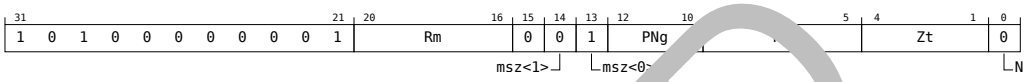
Contiguous store of halfwords from multiple consecutive vectors (scalar index)

Contiguous store of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

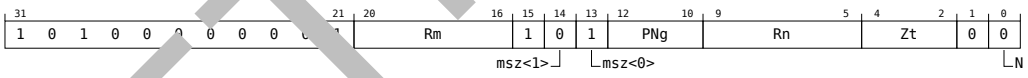
Two registers
(FEAT_SME2)




```
ST1H    { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 16;
```

Four registers
(FEAT_SME2)




```
ST1H    { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 16;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.193 ST1H (scalar plus immediate, strided registers)

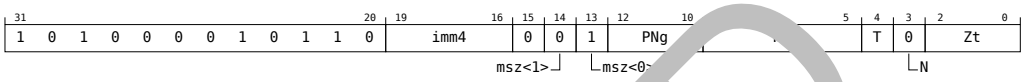
Contiguous store of halfwords from multiple strided vectors (immediate index)

Contiguous store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

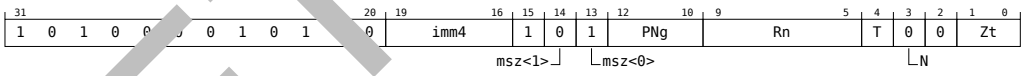
Two registers
(FEAT_SME2)




```
ST1H    { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>{, <imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer tstride = 8;
6  integer t = UInt(T:'0':Zt);
7  constant integer esize = 16;
8  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
ST1H    { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer tstride = 4;
6  integer t = UInt(T:'00':Zt);
7  constant integer esize = 16;
8  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<PL*nreg-1:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ElemPredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = t + tstride;
```

D1.1.194 ST1H (scalar plus scalar, strided registers)

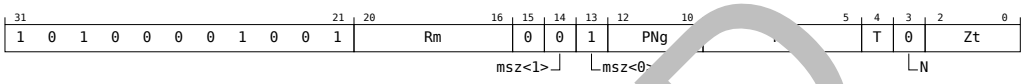
Contiguous store of halfwords from multiple strided vectors (scalar index)

Contiguous store of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

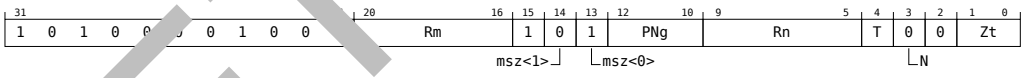
Two registers
(FEAT_SME2)




```
ST1H    { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 16;
```

Four registers
(FEAT_SME2)




```
ST1H    { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 16;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

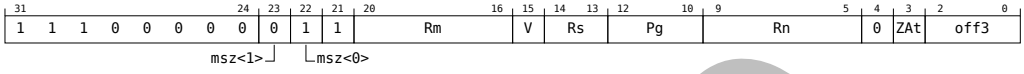
```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(n<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(memop, nontemporal, contiguous,
    tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t + 1) MOD 32;
```

D1.1.195 ST1H (scalar plus scalar, tile slice)

Contiguous store of halfwords from 16-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 16-bit elements in a vector. The immediate offset is in the range 0 to 7. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 2 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)




```
ST1H    { <ZAt><HV>.H[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #1}]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(off3);
constant integer esize = 16;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile, ZA0-ZA1 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, in the range 0 to 7, encoded in the "off3" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
```

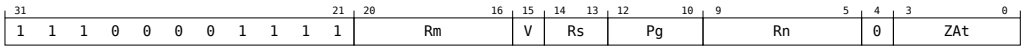
```
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

D1.1.196 ST1Q

Contiguous store of quadwords from 128-bit element ZA tile slice

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)




```
ST1Q    { <ZAt><HV>.Q[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #4}]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = 0;
constant integer esize = 128;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
```

```
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

D1.1.197 ST1W (scalar plus immediate, consecutive registers)

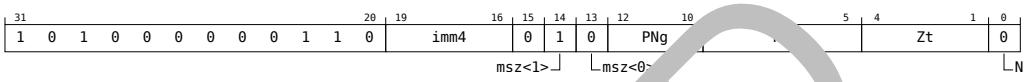
Contiguous store of words from multiple consecutive vectors (immediate index)

Contiguous store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

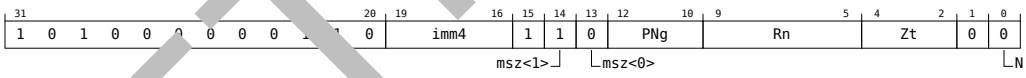
Two registers
(FEAT_SME2)




```
ST1W    { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 32;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
ST1W    { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 32;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.198 ST1W (scalar plus scalar, consecutive registers)

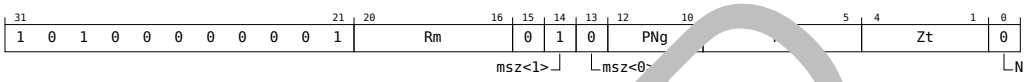
Contiguous store of words from multiple consecutive vectors (scalar index)

Contiguous store of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

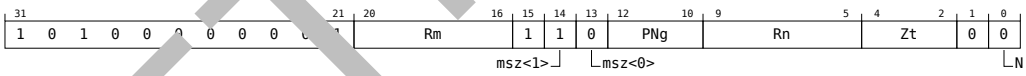
Two registers
(FEAT_SME2)




```
ST1W    { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 32;
```

Four registers
(FEAT_SME2)




```
ST1W    { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 32;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.199 ST1W (scalar plus immediate, strided registers)

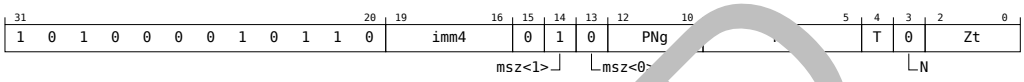
Contiguous store of words from multiple strided vectors (immediate index)

Contiguous store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

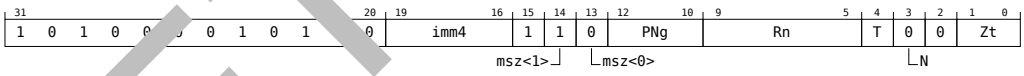
Two registers
(FEAT_SME2)




```
ST1W    { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>{, <imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
ST1W    { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<PL*nreg-1:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if IsActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t+tstride) MOD 32;
```

D1.1.200 ST1W (scalar plus scalar, strided registers)

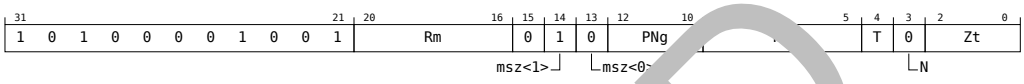
Contiguous store of words from multiple strided vectors (scalar index)

Contiguous store of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

It has encodings from 2 classes: Two registers and Four registers

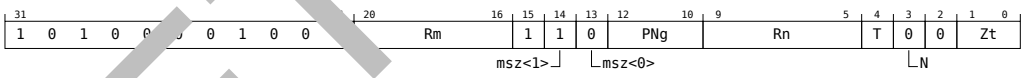
Two registers
(FEAT_SME2)




```
ST1W    { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 32;
```

Four registers
(FEAT_SME2)




```
ST1W    { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 32;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

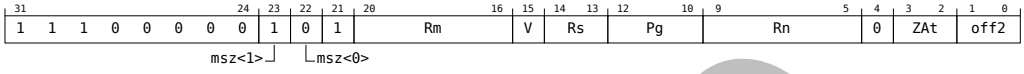
```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(n<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(memop, nontemporal, contiguous,
    tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t + 1) MOD 32;
```

D1.1.201 ST1W (scalar plus scalar, tile slice)

Contiguous store of words from 32-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and immediate offset, modulo the number of 32-bit elements in a vector. The immediate offset is in the range 0 to 3. The memory address is generated by a 64-bit scalar base and an optional 64-bit scalar offset which is multiplied by 4 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)




```
ST1W    { <ZAt><HV>.S[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #2}]
```

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer offset = UInt(off2);
constant integer esize = 32;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile, ZA0-ZA3 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in "V":

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<offs> Is the slice index offset, in the range 0 to 3, encoded in the "off2" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

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

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) moffs = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + offset) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;
```

```
boolean  contiguous = TRUE;
boolean  nontemporal = FALSE;
boolean  tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(moffs) * mbytes;
    if ActivePredicateElement(mask, e, esize) then
        Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    moffs = moffs + 1;
```

D1.1.202 STNT1B (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of bytes from multiple consecutive vectors (immediate index)

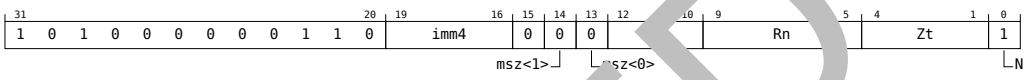
Contiguous store non-temporal of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

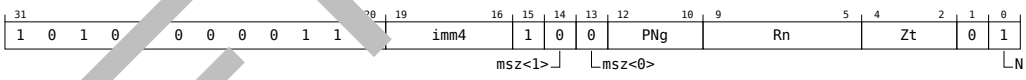
Two registers
(FEAT_SME2)




```
STNT1B  { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 8;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
STNT1B  { <Zt>.B-<Zt+4>  }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 8;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<1 0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDesc E(Me o_STOR nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpre table (Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckS lignment();
    base = if n == 31 th [] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, V ,
    for e = 0 to lements-1
        if Acti ePredic eElement(mask, r * elements + e, esize) then
            bi (64) ur = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[ d mbytes, ccdesc] = Elem[src, e, esize];
```

D1.1.203 STNT1B (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of bytes from multiple consecutive vectors (scalar index)

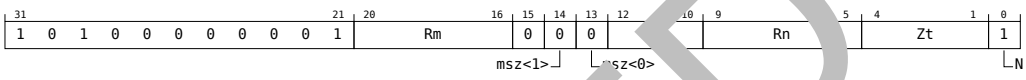
Contiguous store non-temporal of bytes from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

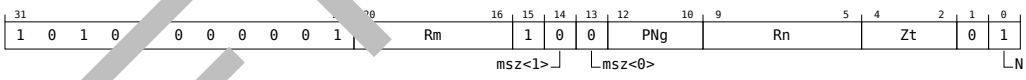
Two registers
(FEAT_SME2)




```
STNT1B  { <Zt1>.B-<Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Rn);
3 integer m = UInt(Rm);
4 integer g = UInt('1':PNg);
5 constant integer nreg = 2;
6 integer t = UInt(Zt:'0');
7 constant integer esize = 8;
```

Four registers
(FEAT_SME2)




```
STNT1B  { <Zt1>.B-<Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Rn);
3 integer m = UInt(Rm);
4 integer g = UInt('1':PNg);
5 constant integer nreg = 4;
6 integer t = UInt(Zt:'00');
7 constant integer esize = 8;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + UInt(offset) + r * elements + e) * mbytes;
            MemStore(addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.204 STNT1B (scalar plus immediate, strided registers)

Contiguous store non-temporal of bytes from multiple strided vectors (immediate index)

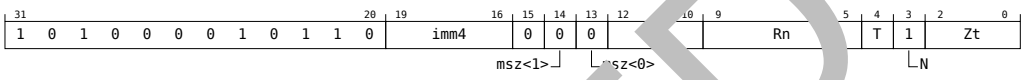
Contiguous store non-temporal of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

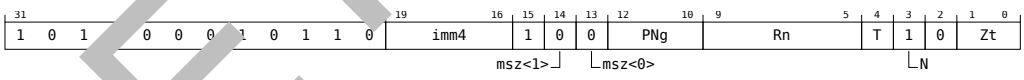
Two registers
(FEAT_SME2)




```
STNT1B  { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Rn);
3 integer g = UInt('1':PNg);
4 constant integer nreg = 2;
5 integer tstride = 8;
6 integer t = UInt(T:'0':Zt);
7 constant integer esize = 8;
8 integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
STNT1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Rn);
3 integer g = UInt('1':PNg);
4 constant integer nreg = 4;
5 integer tstride = 4;
6 integer t = UInt(T:'00':Zt);
7 constant integer esize = 8;
8 integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or

Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterTo[illegible]redicate[illegible]ed<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != [illegible];
AccessDescriptor accdesc = [illegible]ateAccDesc[illegible](MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveEleme[illegible](mask, esize) then
    if n == 31 && ConstrainUnpredic[illegible]bleBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSP[illegible]ignment[illegible];
else
    if n == 31 t[illegible]heckSPA[illegible]gnment();
    base[illegible] n == [illegible] then [illegible][] else X[n, 64];

for [illegible] = 0 to [illegible]reg-1
    [illegible]rc = Z[illegible]VL];
    [illegible] e = 0 to [illegible]ents-1
        [illegible] ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            [illegible]em[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = t + [illegible]stride;
```

D1.1.205 STNT1B (scalar plus scalar, strided registers)

Contiguous store non-temporal of bytes from multiple strided vectors (scalar index)

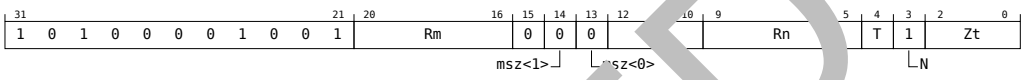
Contiguous store non-temporal of bytes from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

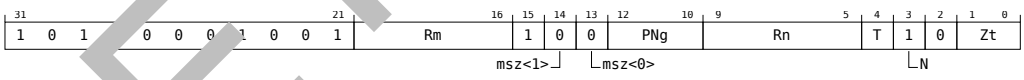
Two registers
(FEAT_SME2)




```
STNT1B  { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 8;
```

Four registers
(FEAT_SME2)




```
STNT1B  { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <PNg>, [<Xn|SP>, <Xm>]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 8;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or

Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t + tstride);
```

D1.1.206 STNT1D (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of doublewords from multiple consecutive vectors (immediate index)

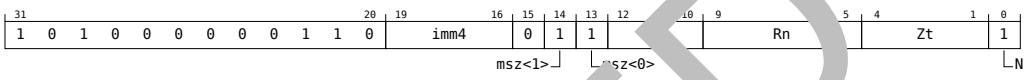
Contiguous store non-temporal of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

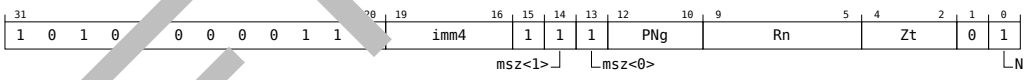
Two registers
(FEAT_SME2)




```
STNT1D  { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 64;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
STNT1D  { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 64;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<1 0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDesc E(Me o_STO nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpre table (Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckS lignment();
    base = if n == 31 th [] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, V ,
    for e = 0 to lements-1
        if Acti ePredic eElement(mask, r * elements + e, esize) then
            bi (64) ur = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[ d mbytes, ccdesc] = Elem[src, e, esize];
```

D1.1.207 STNT1D (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of doublewords from multiple consecutive vectors (scalar index)

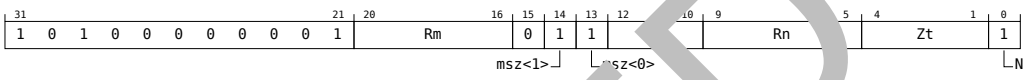
Contiguous store non-temporal of doublewords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

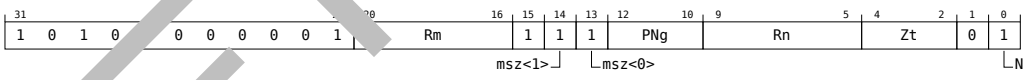
Two registers
(FEAT_SME2)




```
STNT1D  { <Zt1>.D-<Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 64;
```

Four registers
(FEAT_SME2)




```
STNT1D  { <Zt1>.D-<Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 64;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + UInt(offset) + r * elements + e) * mbytes;
            MemStore(addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.208 STNT1D (scalar plus immediate, strided registers)

Contiguous store non-temporal of doublewords from multiple strided vectors (immediate index)

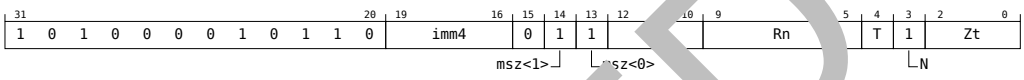
Contiguous store non-temporal of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

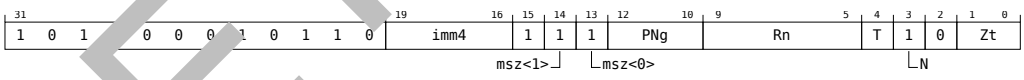
Two registers
(FEAT_SME2)




```
STNT1D  { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Rn);
3 integer g = UInt('1':PNg);
4 constant integer nreg = 2;
5 integer tstride = 8;
6 integer t = UInt(T:'0':Zt);
7 constant integer esize = 64;
8 integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
STNT1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Rn);
3 integer g = UInt('1':PNg);
4 constant integer nreg = 4;
5 integer tstride = 4;
6 integer t = UInt(T:'00':Zt);
7 constant integer esize = 64;
8 integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or

Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterTo[illegible]edicate[illegible]ed<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != [illegible];
AccessDescriptor accdesc = [illegible]ateAccDesc[illegible](MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveEleme[illegible](mask, esize) then
    if n == 31 && ConstrainUnpredic[illegible]bleBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSP[illegible]ignment[illegible];
else
    if n == 31 t[illegible]heckSPA[illegible]gnment();
    base[illegible] n == [illegible] then [illegible][] else X[n, 64];

for [illegible] = 0 to [illegible]reg-1
    [illegible]rc = Z[illegible]VL];
    [illegible] e = 0 to [illegible]ents-1
        [illegible] ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            [illegible]em[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = t + [illegible]stride;
```

D1.1.209 STNT1D (scalar plus scalar, strided registers)

Contiguous store non-temporal of doublewords from multiple strided vectors (scalar index)

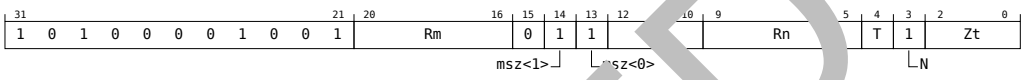
Contiguous store non-temporal of doublewords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

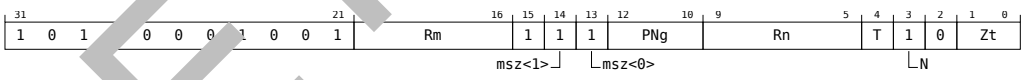
Two registers
(FEAT_SME2)




```
STNT1D  { <Zt1>.D, <Zt2>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 64;
```

Four registers
(FEAT_SME2)




```
STNT1D  { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <PNg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 64;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or

Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, ESIZE, nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t + tstride);
```

D1.1.210 STNT1H (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of halfwords from multiple consecutive vectors (immediate index)

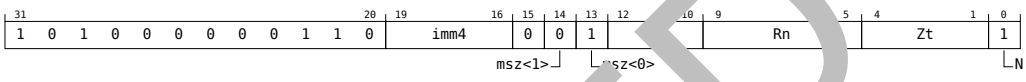
Contiguous store non-temporal of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

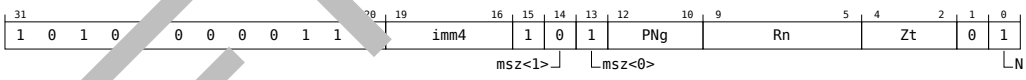
Two registers
(FEAT_SME2)




```
STNT1H  { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 16;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
STNT1H  { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 16;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<1 0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDesc E(Me o_STOR nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpre table (Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckS lignment();
    base = if n == 31 th [] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, V ,
    for e = 0 to lements-1
        if Acti ePredic eElement(mask, r * elements + e, esize) then
            bi (64) ur = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[ d mbytes, ccdesc] = Elem[src, e, esize];
```

D1.1.211 STNT1H (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of halfwords from multiple consecutive vectors (scalar index)

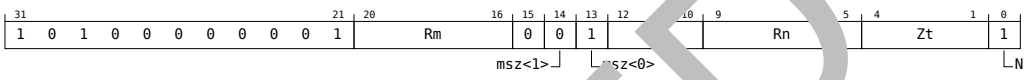
Contiguous store non-temporal of halfwords from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

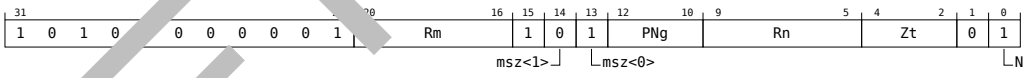
Two registers
(FEAT_SME2)




```
STNT1H  { <Zt1>.H-<Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer t = UInt(Zt:'0');
constant integer esize = 16;
```

Four registers
(FEAT_SME2)




```
STNT1H  { <Zt1>.H-<Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer t = UInt(Zt:'00');
constant integer esize = 16;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + UInt(offset) + r * elements + e) * mbytes;
            MemStore(addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.212 STNT1H (scalar plus immediate, strided registers)

Contiguous store non-temporal of halfwords from multiple strided vectors (immediate index)

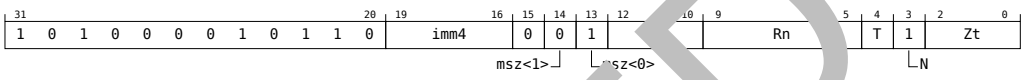
Contiguous store non-temporal of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

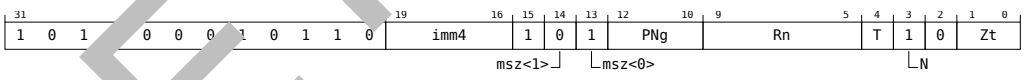
Two registers
(FEAT_SME2)




```
STNT1H  { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Rn);
3 integer g = UInt('1':PNg);
4 constant integer nreg = 2;
5 integer tstride = 8;
6 integer t = UInt(T:'0':Zt);
7 constant integer esize = 16;
8 integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
STNT1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1 if !HaveSME2() then UNDEFINED;
2 integer n = UInt(Rn);
3 integer g = UInt('1':PNg);
4 constant integer nreg = 4;
5 integer tstride = 4;
6 integer t = UInt(T:'00':Zt);
7 constant integer esize = 16;
8 integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or

Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterTo[illegible]redicate[illegible]ed<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != [illegible];
AccessDescriptor accdesc = [illegible]ateAccDesc[illegible](MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveEleme[illegible](mask, esize) then
    if n == 31 && ConstrainUnpredic[illegible]bleBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSP[illegible]ignment[illegible];
else
    if n == 31 t[illegible]heckSPA[illegible]gnment();
    base[illegible] n == [illegible] then [illegible][] else X[n, 64];

for [illegible] = 0 to [illegible]reg-1
    [illegible]rc = Z[illegible]VL];
    [illegible] e = 0 to [illegible]ents-1
        [illegible] ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            [illegible]em[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = t + [illegible]stride;
```

D1.1.213 STNT1H (scalar plus scalar, strided registers)

Contiguous store non-temporal of halfwords from multiple strided vectors (scalar index)

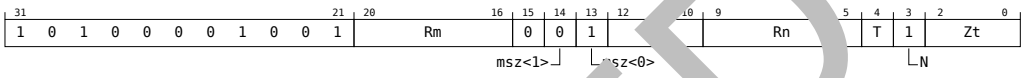
Contiguous store non-temporal of halfwords from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

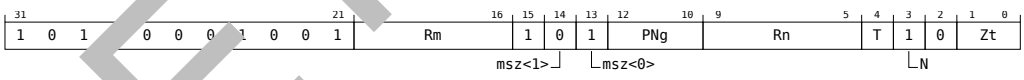
Two registers
(FEAT_SME2)




```
STNT1H  { <Zt1>.H, <Zt2>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 16;
```

Four registers
(FEAT_SME2)




```
STNT1H  { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <PNg>, [<Xn|SP>, <Xm>, LSL #1]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 16;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or

Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, ESIZE, nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t + tstride);
```

D1.1.214 STNT1W (scalar plus immediate, consecutive registers)

Contiguous store non-temporal of words from multiple consecutive vectors (immediate index)

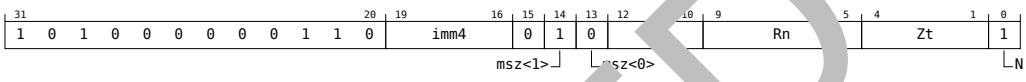
Contiguous store non-temporal of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

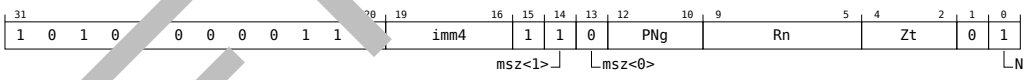
Two registers
(FEAT_SME2)




```
STNT1W  { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 2;
5  integer t = UInt(Zt:'0');
6  constant integer esize = 32;
7  integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
STNT1W  { <Zt4>.S-<Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer g = UInt('1':PNg);
4  constant integer nreg = 4;
5  integer t = UInt(Zt:'00');
6  constant integer esize = 32;
7  integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<1 0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDesc E(Me o_STOR nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpre table (Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckS lignment();
    base = if n == 31 th [] else X[n, 64];

for r = 0 to nreg-1
    src = Z[t+r, V ,
    for e = 0 to lements-1
        if Acti ePredic eElement(mask, r * elements + e, esize) then
            bi (64) ur = base + (offset * nreg * elements + r * elements + e) * mbytes;
            Mem[ d mbytes, ccdesc] = Elem[src, e, esize];
```

D1.1.215 STNT1W (scalar plus scalar, consecutive registers)

Contiguous store non-temporal of words from multiple consecutive vectors (scalar index)

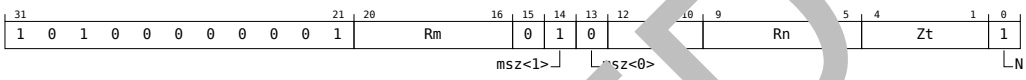
Contiguous store non-temporal of words from elements of two or four consecutive vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

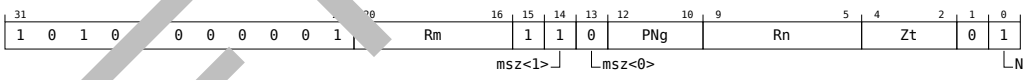
Two registers
(FEAT_SME2)




```
STNT1W  { <Zt1>.S-<Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer t = UInt(Zt:'0');
7  constant integer esize = 32;
```

Four registers
(FEAT_SME2)




```
STNT1W  { <Zt1>.S-<Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer t = UInt(Zt:'00');
7  constant integer esize = 32;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 2.

For the four registers variant: is the name of the first scalable vector register to be transferred, encoded as "Zt" times 4.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" times 4 plus 3.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" times 2 plus 1.

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + UInt(offset) + r * elements + e) * mbytes;
            MemStore(addr, mbytes, accdesc] = Elem[src, e, esize];
```

D1.1.216 STNT1W (scalar plus immediate, strided registers)

Contiguous store non-temporal of words from multiple strided vectors (immediate index)

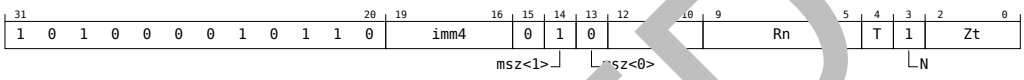
Contiguous store non-temporal of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and immediate index which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

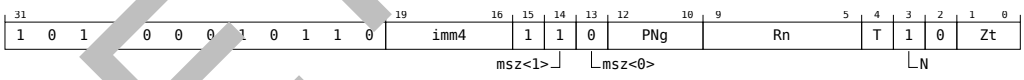
Two registers
(FEAT_SME2)




```
STNT1W  { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 2;
integer tstride = 8;
integer t = UInt(T:'0':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Four registers
(FEAT_SME2)




```
STNT1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
integer g = UInt('1':PNg);
constant integer nreg = 4;
integer tstride = 4;
integer t = UInt(T:'00':Zt);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or

Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

<imm> For the two registers variant: is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

For the four registers variant: is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterTo[illegible]edicate[illegible]ed<15:0>, PL * nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = n != [illegible];
AccessDescriptor accdesc = [illegible]ateAccDesc[illegible](MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveEleme[illegible](mask, esize) then
    if n == 31 && ConstrainUnpredic[illegible]bleBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSP[illegible]ignment[illegible];
else
    if n == 31 t[illegible]heckSPA[illegible]gnment();
    base[illegible] n == [illegible] then [illegible][] else X[n, 64];

for [illegible] = 0 to [illegible]reg-1
    [illegible]rc = Z[illegible]VL];
    [illegible] e = 0 to [illegible]ents-1
        [illegible] ActivePredicateElement(mask, r * elements + e, esize) then
            bits(64) addr = base + (offset * nreg * elements + r * elements + e) * mbytes;
            [illegible]em[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = t + [illegible]stride;
```

D1.1.217 STNT1W (scalar plus scalar, strided registers)

Contiguous store non-temporal of words from multiple strided vectors (scalar index)

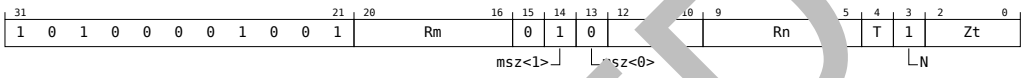
Contiguous store non-temporal of words from elements of two or four strided vector registers to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated.

Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: Two registers and Four registers

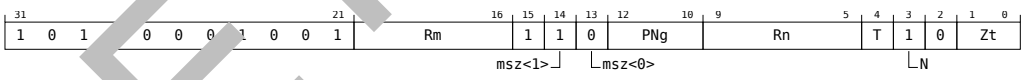
Two registers
(FEAT_SME2)




```
STNT1W  { <Zt1>.S, <Zt2>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 2;
6  integer tstride = 8;
7  integer t = UInt(T:'0':Zt);
8  constant integer esize = 32;
```

Four registers
(FEAT_SME2)




```
STNT1W  { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <PNg>, [<Xn|SP>, <Xm>, LSL #2]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer m = UInt(Rm);
4  integer g = UInt('1':PNg);
5  constant integer nreg = 4;
6  integer tstride = 4;
7  integer t = UInt(T:'00':Zt);
8  constant integer esize = 32;
```

Assembler Symbols

<Zt1> For the two registers variant: is the name of the first scalable vector register Z0-Z7 or Z16-Z23 to be transferred, encoded as "T:'0':Zt".

For the four registers variant: is the name of the first scalable vector register Z0-Z3 or Z16-Z19 to be transferred, encoded as "T:'00':Zt".

<Zt2> For the two registers variant: is the name of the second scalable vector register Z8-Z15 or Z24-Z31 to be transferred, encoded as "T:'1':Zt".

For the four registers variant: is the name of the second scalable vector register Z4-Z7 or

Z20-Z23 to be transferred, encoded as "T:'01':Zt".

<Zt3> Is the name of the third scalable vector register Z8-Z11 or Z24-Z27 to be transferred, encoded as "T:'10':Zt".

<Zt4> Is the name of the fourth scalable vector register Z12-Z15 or Z28-Z31 to be transferred, encoded as "T:'11':Zt".

<PNg> Is the name of the governing scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNg" field.

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

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

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer mbytes = esize DIV 8;
bits(64) offset;
bits(64) base;
bits(VL) src;
bits(PL) pred = P[g, PL];
bits(PL * nreg) mask = CounterToPredicate(pred<3:0>, ESIZE, nreg);
boolean contiguous = TRUE;
boolean nontemporal = TRUE;
boolean tagchecked = TRUE;
AccessDescriptor accdesc = CreateAccDescSVE(MemOp_STORE, nontemporal, contiguous,
    ↪tagchecked);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    src = Z[t+r, VL];
    for e = 0 to elements-1
        if ActivePredicateElement(mask, r * elements + e, esize) then
            addr = base + (UInt(offset) + r * elements + e) * mbytes;
            Mem[addr, mbytes, accdesc] = Elem[src, e, esize];
    t = (t + tstride);
```

D1.1.218 STR (vector)

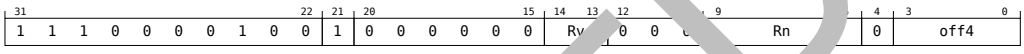
Store ZA array vector

The ZA array vector is selected by the sum of the vector select register and immediate offset, modulo the number of bytes in a Streaming SVE vector. The immediate offset is in the range 0 to 15. The memory address is generated by a 64-bit scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME
(FEAT_SME)




```
STR     ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MU }]
```

```
1  if !HaveSME() then UNDEFINED;
2  integer n = UInt(Rn);
3  integer v = UInt('011':Rv);
4  integer offset = UInt(off4);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.

<offs> Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the "off4" field.

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

Operation

```
1  CheckSMEAndZAEnabled();
2  constant integer SVL = CurrentSVL;
3  constant integer dim = SVL DIV 8;
4  bits(64) base;
5  integer offs = offset * dim;
6  bits(SVL) src;
7  bits(32)  vbase = X[v, 32];
8  integer   vec = (UInt(vbase) + offset) MOD dim;
9  boolean   contiguous = TRUE;
10 boolean   nontemporal = FALSE;
11 boolean   tagchecked = n != 31;
12 AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
       ↪tagchecked);
13
14 if HaveTME() && TSTATE.depth > 0 then
15     FailTransaction(TMFailure_ERR, FALSE);
16
17 if n == 31 then
18     CheckSPAlignment();
19     base = SP[];
20 else
21     base = X[n, 64];
22
23 src = ZAvector[vec, SVL];
```

```
24
25 boolean aligned = IsAligned(base + offset, 16);
26
27 if !aligned && AlignmentEnforced() then
28     AArch64.Abort(base + moffs, AlignmentFault(accdesc));
29
30 for e = 0 to dim-1
31     AArch64.MemSingle[base + moffs, 1, accdesc, aligned] = Elem[src, e, 8];
32     moffs = moffs + 1;
```

D1.1.219 STR (ZT0)

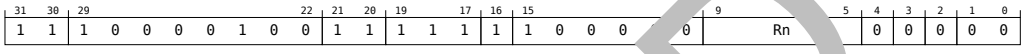
Store ZT0 register

Store the 64-byte ZT0 register to the memory address provided in the 64-bit scalar base register. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2
(FEAT_SME2)




```
STR     ZT0, [<Xn|SP>]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Rn);
```

Assembler Symbols

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

Operation

```
CheckSMEEnabled();
CheckSMEZT0Enabled();
constant integer elements = 512 DIV 8;
bits(64) base;
bits(512) table = ZT0[512-1:0];
boolean contiguous = TRUE;
boolean nontemporal = FALSE;
boolean tagchecked = n != 31;
AccessDescriptor accdesc = CreateAccDescSME(MemOp_STORE, nontemporal, contiguous,
    tagchecked);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

boolean aligned = IsAligned(base, 16);

if !aligned && AlignmentEnforced() then
    AArch64.Abort(base, AlignmentFault(accdesc));

for e = 0 to elements-1
    AArch64.MemSingle[base + e, 1, accdesc, aligned] = Elem[table, e, 8];
```

D1.1.220 SUB (array accumulators)

Subtract multi-vector from ZA array vector accumulators

The instruction operates on two or four ZA single-vector groups.

Destructively subtract all elements of the two or four source vectors from the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

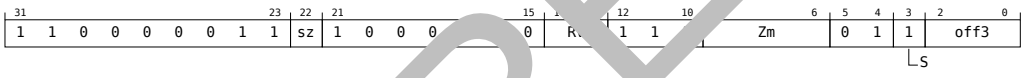
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer v ant is implem ted.

It has encodings from 2 classes: Two ZA single-vectors and Fo ZA si -vectors

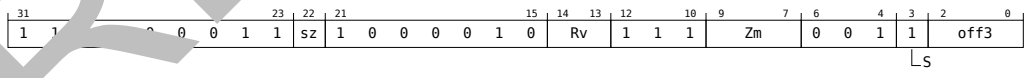
Two ZA single-vectors
(FEAT_SME2)




```
SUB     ZA.<T>[<Wv>, <offs>{, V 2}], { Zm1>.<T>-<Zm2>.<T> }
```

```
if !HaveSME2() then UNDEFIN ;
if sz == '1' && !HaveSME 4() then UN FINED;
integer v = UInt('010' );
constant integer esize = 32 << nt(sz);
integer m = UInt(Zm 0');
integer offset = nt(off3);
constant intege nreg =
```

Four ZA single-vec
(FEAT_)




```
SUB     ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zm1>.<T>-<Zm4>.<T> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = ZAvector[
    bits(VL) operand2 = Z[m+r, VL]
    for e = 0 to elements-1
        bits(esize) element
        bits(esize) eleme
        Elem[result, e,
    ZAvector[vec, VL] =
    vec = vec + vst
```

D1.1.221 SUB (array results, multiple and single vector)

Subtract replicated single vector from multi-vector with ZA array vector results

The instruction operates on two or four ZA single-vector groups.

Subtract all corresponding elements of the second source vector from the two or four first source vectors and place the results in the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

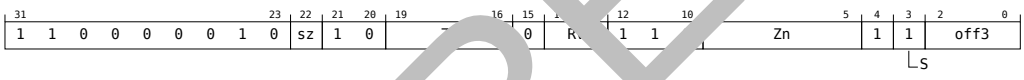
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

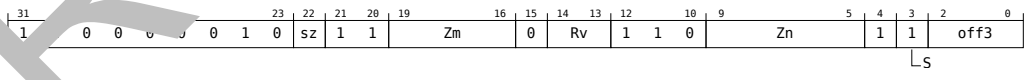
Two ZA single-vectors
(FEAT_SME2)




```
SUB     ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
SUB     ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 - element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.222 SUB (array results, multiple vectors)

Subtract multi-vector from multi-vector with ZA array vector results

The instruction operates on two or four ZA single-vector groups.

Subtract all corresponding elements of the two or four second source vectors from first source vectors and place the results in the corresponding elements of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

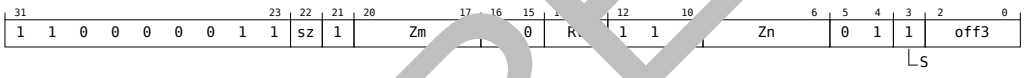
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

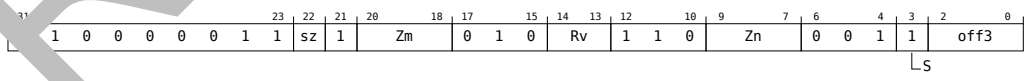
Two ZA single-vectors
(FEAT_SME2)




```
SUB     ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
SUB     ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<T>-<Zn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = element1 - element2;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.223 SUDOT (multiple and indexed vector)

Multi-vector signed by unsigned integer dot-product by indexed elements

The instruction operates on two or four ZA single-vector groups.

The signed by unsigned integer dot product instruction computes the dot product of four signed 8-bit integer values held in each 32-bit element of the two or four first source vectors and four unsigned 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups.

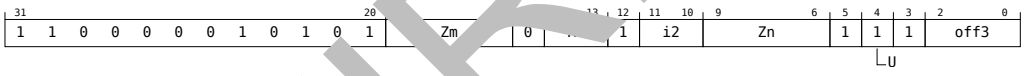
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter th[illegible]ber of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ope[illegible]d consists o[illegible]vo or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for dis[illegible]embly, but op[illegible]onal in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a[illegible] Four [illegible]A single-vectors

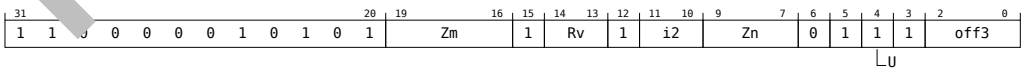
Two ZA single-vectors
(FEAT_SME2)




```
SUDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg
```

Four [illegible]A single-vectors
(FEAT_[illegible]E2)




```
SUDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z[illegible], encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offs) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = [illegible] VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer index = segmentbase + index;
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.224 SUDOT (multiple and single vector)

Multi-vector signed by unsigned integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

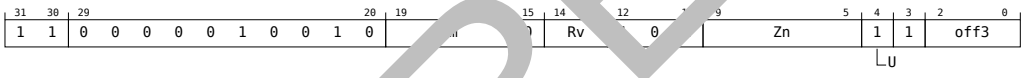
The signed by unsigned integer dot product instruction computes the dot product of four signed 8-bit integer values held in each 32-bit element of the two or four first source vectors and four unsigned 8-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser[illegible]t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F[illegible] ZA sing[illegible]vectors

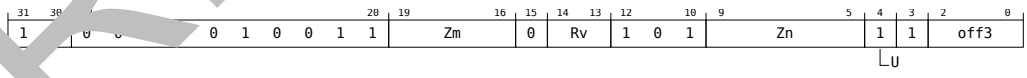
Two ZA single-vectors
(FEAT_SME2)




```
SUDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
SUDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * r + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.225 SUMLALL (multiple and indexed vector)

Multi-vector signed by unsigned integer multiply-add long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This signed by unsigned integer multiply-add long long instruction multiplies each signed 8-bit element in the one, two, or four first source vectors with each unsigned 8-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The element index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 4 bits. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors and Four ZA quad-vectors

One ZA quad-vector
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14–13	12–10	9–5	4	3	2	1–0
1	1	0	0	0	0	0	1	0	0	0	0	Zm	i4h	Rv	i4l	Zn	1 (U)	0 (S)	1	off2

```
SUMLALL ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.B, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

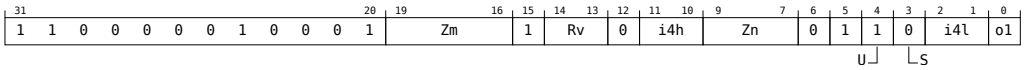
Two ZA quad-vectors
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15	14–13	12	11–10	9–6	5	4	3	2–1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm	0	Rv	0	i4h	Zn	1	1 (U)	0 (S)	i4l	o1

```
SUMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

Four ZA quad-vectors (FEAT_SME2)




```
SUMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 4 * segmentbase + index;
            integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, s, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.226 SUMLALL (multiple and single vector)

Multi-vector signed by unsigned integer multiply-add long long by vector

The instruction operates on two or four ZA quad-vector groups.

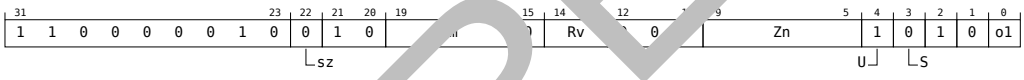
This signed by unsigned integer multiply-add long long instruction multiplies each signed 8-bit element in the two or four first source vectors with each unsigned 8-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

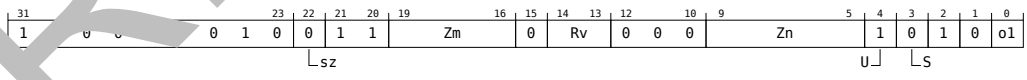
Two ZA quad-vectors
(FEAT_SME2)




```
SUMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
SUMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[ + i
        for e = 0 to elements-1
            integer element1 = SInt(Elem[ rand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt( em[op nd2, 4 * e + i, esize DIV 4]);
            bits(esize) pro ct = (ele nt1 * lement2)<esize-1:0>;
            Elem[result, size] = Ele operand3, e, esize] + product;
        ZAvector[vec +  VL] = result;
    vec = vec + vstride;
```

D1.1.227 SUMOPA

Signed by unsigned integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

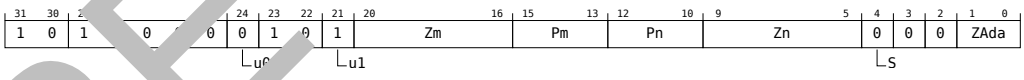
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

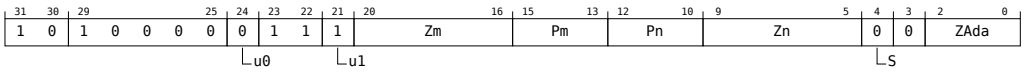
32-bit
(FEAT_SME)




```
SUMOPA  <Zda>.<T>, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

```
1  if !HaveSME() then UNDEFINED;
2  constant integer esize = 32;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer m = UInt(Zm);
7  integer da = UInt(ZAda);
8  boolean sub_op = FALSE;
9  boolean op1_unsigned = FALSE;
10 boolean op2_unsigned = TRUE;
```

64-bit
(FEAT_SME_I16I64)



```
SUMOPA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
                ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.

D1.1.228 SUMOPS

Signed by unsigned integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

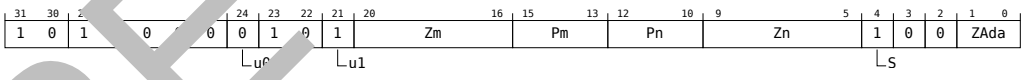
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

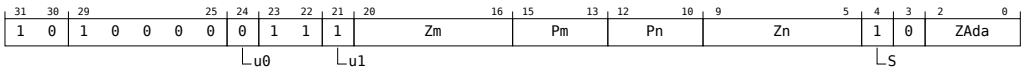
32-bit
(FEAT_SME)




```
SUMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

```
1  if !HaveSME() then UNDEFINED;
2  constant integer esize = 32;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer m = UInt(Zm);
7  integer da = UInt(ZAda);
8  boolean sub_op = TRUE;
9  boolean op1_unsigned = FALSE;
10 boolean op2_unsigned = TRUE;
```

64-bit
(FEAT_SME_I16I64)



```
SUMOPS  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
                ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.

D1.1.229 SUNPK

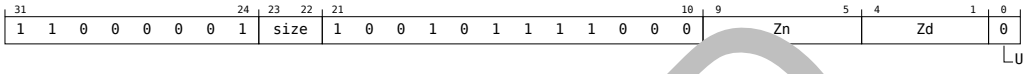
Unpack and sign-extend multi-vector elements

Unpack elements from one or two source vectors and then sign-extend them to place in elements of twice their size within the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

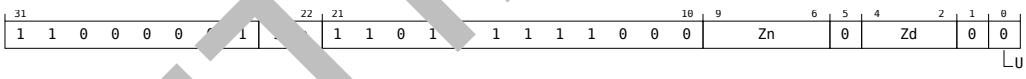
Two registers
(FEAT_SME2)




```
SUNPK   { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<Tb>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer n = UInt(Zn);
5  integer d = UInt(Zd:'0');
6  constant integer nreg = 2;
7  boolean unsigned = FALSE;
```

Four registers
(FEAT_SME2)




```
SUNPK   { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<Tb>-<Zn2>.<Tb> }
```

```
1  if !HaveSME2() then UNDEFINED;
2  if size == '00' then UNDEFINED;
3  constant integer esize = 8 << UInt(size);
4  integer n = UInt(Zn:'0');
5  integer d = UInt(Zd:'00');
6  constant integer nreg = 4;
7  boolean unsigned = FALSE;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
constant integer sreg = nreg DIV 4;
array [0..3] of bits(VL) results;

for r = 0 to sreg-1
    bits(VL) operand = Z[n+r, VL];
    for i = 0 to 1
        for e = 0 to elements-1
            bits(hsize) element = Elem[operand, i*elements + e, hsize];
            Elem[results[2*r+i], e, esize] = Extend(element, esize, unsigned);

for r = 0 to sreg-1
    Z[d+r, VL] = results[r];
```

D1.1.230 SUVDOT

Multi-vector signed by unsigned integer vertical dot-product by indexed element

The instruction operates on four ZA single-vector groups.

The signed by unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding signed 8-bit elements from the four first source vectors and four unsigned 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the four ZA single-vector groups.

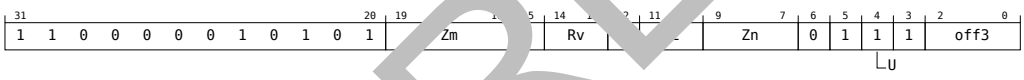
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the numb[illegible] of ZA [illegible]ray vectors.

The VECTOR GROUP symbol VGx4 indicates that the ZA operand c[illegible]sists of four ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in a[illegible]mbler source [illegible]ode.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SUVDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Wv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 4;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 3
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = SInt(Elem[operand1, 4 * e + r, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.231 SVDOT (2-way)

Multi-vector signed integer vertical dot-product by indexed element

The instruction operates on two ZA single-vector groups.

The signed integer vertical dot product instruction computes the vertical dot product of the corresponding two signed 16-bit integer values held in the two first source vectors and two signed 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit element of two ZA single-vector groups.

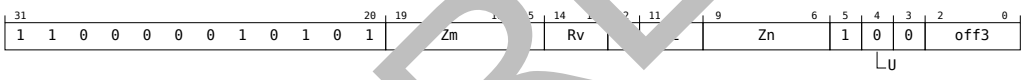
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand consists of two ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
SVDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  constant integer esize = 32;
4  integer n = UInt(Zn:'0');
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off3);
7  integer index = UInt(i2);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer elements = VL DIV esize;
4  integer vectors = VL DIV 8;
5  integer vstride = vectors DIV 2;
6  integer eltspersegment = 128 DIV esize;
7  bits(32) vbase = X[v, 32];
8  integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 1
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = SInt(Elem[operand1, 2 * e + r, esize DIV 2]);
            integer element2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.232 SVDOT (4-way)

Multi-vector signed integer vertical dot-product by indexed element

The instruction operates on four ZA single-vector groups.

The signed integer vertical dot product instruction computes the vertical dot product of the corresponding four signed 8-bit or 16-bit integer values held in the four first source vectors and four signed 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit or 64-bit element of the four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

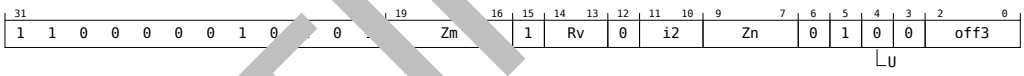
The VECTOR GROUP symbol VGx4 indicates that the ZA operand consists of four ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit to 64-bit variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

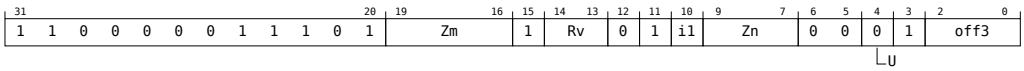
32-bit
(FEAT_SME2)




```
SVDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

64-bit
(FEAT_SME2 && FEAT_SMEI16I64)




```
SVDOT   ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 4;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 3
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = SInt(Elem[operand1, 4 * e + r, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.233 UCLAMP

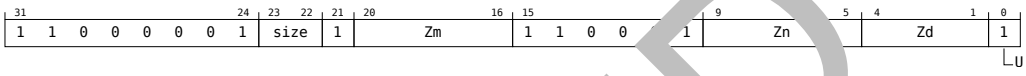
Multi-vector unsigned clamp to minimum/maximum vector

Clamp each unsigned element in the two or four destination vectors to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the two or four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

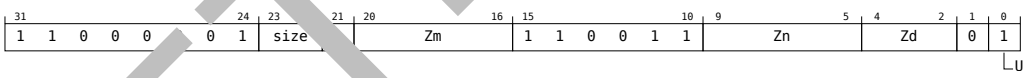
Two registers
(FEAT_SME2)




```
UCLAMP  { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
UCLAMP  { <Zd1>.<T>-<Zd4>.<T> }, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'00');
constant integer nreg = 4;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = Z[d+r, VL];
    for e = 0 to elements-1
        integer element1 = UInt(Elem[operand1, e, esize]);
        integer element2 = UInt(Elem[operand2, e, esize]);
        integer element3 = UInt(Elem[operand3, e, esize]);
        integer res = Min(Max(element1, element3), element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.234 UCVTF

Multi-vector unsigned integer convert to floating-point

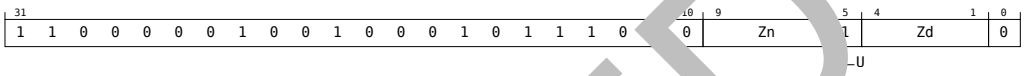
Convert to single-precision from unsigned 32-bit integer, each element of the two or four source vectors, and place the results in the corresponding elements of the two or four destination vectors.

This instruction follows SME2 floating-point numerical behaviors corresponding to instructions that place their results in one or more SVE Z vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

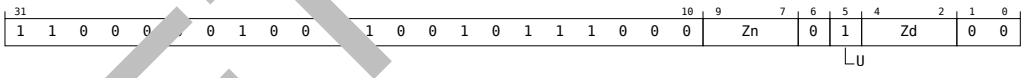
Two registers
(FEAT_SME2)




```
UCVTF   { <Zd1>.S-<Zd2>.S }, { <Zn1>.S-<Zn2>. }
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer n = UInt(Zn:'0');
3  integer d = UInt(Zd:'0');
4  constant integer nreg = 2;
5  boolean unsigned = TRUE;
6  FPRounding rounding = FPRoundingM  e(FPCR  ;
```

Four registers
(FEAT_SME2)




```
UCVTF   { <Zd1>.S-<Zd4>.S }, { <Zn1>.S-<Zn4>.S }
```

```
1  if  veSME2(  then UN  INED;
2  int  r n =  :'00');
3  integ  d  UInt(Zd:'00');
4  constan  nteger nreg = 4;
5  boolean u  igned = TRUE;
6  FPRounding  nding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 32;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(32) element = Elem[operand, e, 32];
        Elem[results[r], e, 32] = FixedToFP(element, , unsigned, FPCR[], rounding, 32);

for r = 0 to nreg-1
    Z[d+r, VL] = results[r];
```

D1.1.235 UDOT (2-way, multiple and indexed vector)

Multi-vector unsigned integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups.

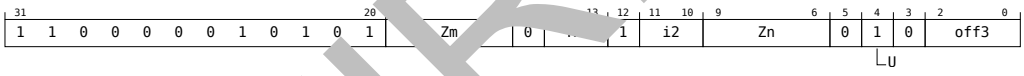
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter th[illegible]ber of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ope[illegible]d consists o[illegible]vo or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for dis[illegible]embly, but op[illegible]onal in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a[illegible] Four [illegible]A single-vectors

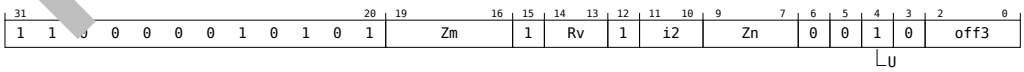
Two ZA single-vectors
(FEAT_SME2)




```
UDOT    ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg
```

Four [illegible]A single-vector[illegible]
(FEAT_[illegible]E2)




```
UDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z[illegible], encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offs) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = [illegible] VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer index = segmentbase + index;
        for i = 0 to 1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.236 UDOT (2-way, multiple and single vector)

Multi-vector unsigned integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

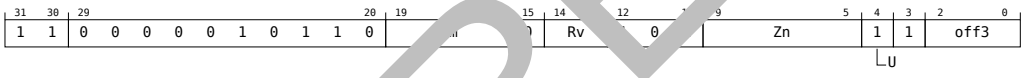
The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser[illegible]t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F[illegible] ZA sing[illegible]vectors

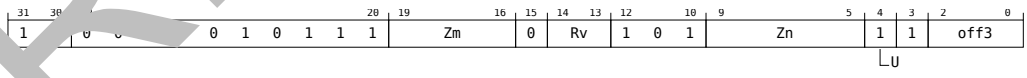
Two ZA single-vectors
(FEAT_SME2)




```
UDOT    ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA [illegible]-vect[illegible]
(FEAT[illegible]ME2)




```
UDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize
        for i = 0 to 1
            integer element1 = UInt(Elem[
            integer element2 = UInt(Ele
            sum = sum + element1 * el
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.237 UDOT (2-way, multiple vectors)

Multi-vector unsigned integer dot-product

The instruction operates on two or four ZA single-vector groups.

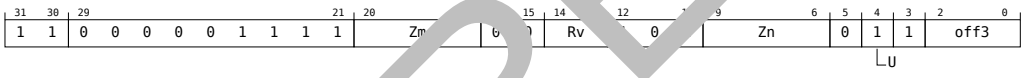
The unsigned integer dot product instruction computes the dot product of two unsigned 16-bit integer values held in each 32-bit element of the two or four first source vectors and two unsigned 16-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser[illegible]t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F[illegible] ZA sing[illegible]vectors

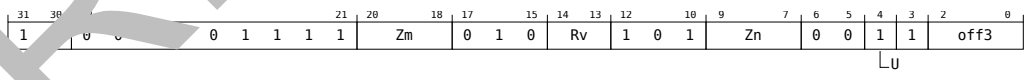
Two ZA single-vectors
(FEAT_SME2)




```
UDOT    ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  constant integer esize = 32;
4  integer n = UInt(Zn:'0');
5  integer m = UInt(Zm:'0');
6  integer offset = UInt(off3);
7  constant integer nreg = 2;
```

Four ZA [illegible]-vect[illegible]
(FEAT[illegible]ME2)




```
UDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  constant integer esize = 32;
4  integer n = UInt(Zn:'00');
5  integer m = UInt(Zm:'00');
6  integer offset = UInt(off3);
7  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a

multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.238 UDOT (4-way, multiple and indexed vector)

Multi-vector unsigned integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

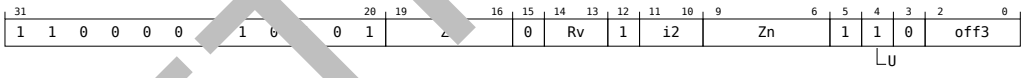
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 4 classes: Two ZA single-vectors of 32-bit elements , Two ZA single-vectors of 64-bit elements , Four ZA single-vectors of 32-bit elements and Four ZA single-vectors of 64-bit elements

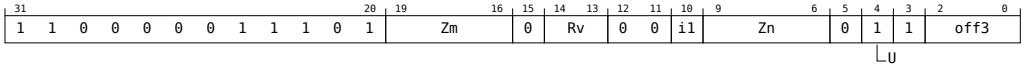
Two ZA single-vectors of 32-bit elements
(FEAT_SME2)




```
UDOT    ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 2;
```

Two ZA single-vectors of 64-bit elements
(FEAT_SME_I16I64)

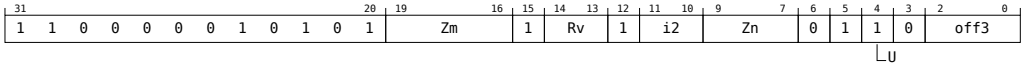



```
UDOT    ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
```

```
integer offset = UInt(off3);
integer index = UInt(i1);
constant integer nreg = 2;
```

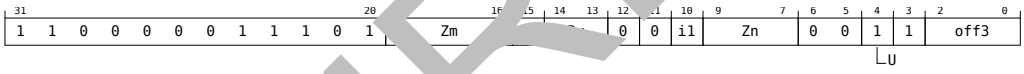
Four ZA single-vectors of 32-bit elements (FEAT_SME2)




```
UDOT    ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

Four ZA single-vectors of 64-bit elements (FEAT_SME_I16I64)




```
UDOT    ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors of 32-bit elements and two ZA single-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors of 32-bit elements and four ZA single-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the four ZA single-vectors of 32-bit elements and two ZA single-vectors of 32-bit

elements variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the four ZA single-vectors of 64-bit elements and two ZA single-vectors of 64-bit elements variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.239 UDOT (4-way, multiple and single vector)

Multi-vector unsigned integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

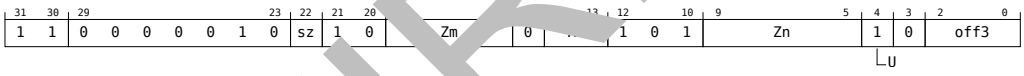
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand [illegible] of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

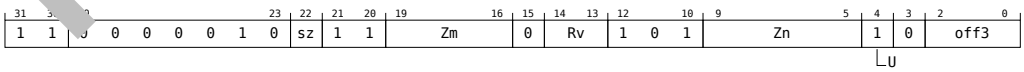
Two ZA single-vectors
(FEAT_SME2)




```
UDOT    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
UDOT    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 2^log;
bits(32) vbase = X[n, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nregs-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.240 UDOT (4-way, multiple vectors)

Multi-vector unsigned integer dot-product

The instruction operates on two or four ZA single-vector groups.

The unsigned integer dot product instruction computes the dot product of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the two or four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the two or four second source vectors. The widened dot product result is destructively added to the corresponding 32-bit or 64-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand [illegible] of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

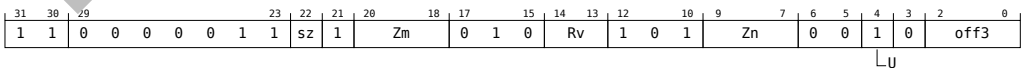
It has encodings from 2 classes: Two ZA single-vectors and Four ZA single-vectors

Two ZA single-vectors
(FEAT_SME2)


```
UDOT    ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm2>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
UDOT    ZA.<T>[<Wv>, <offs>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer elements = VL DIV esize;
4  integer vectors = VL DIV 8;
5  integer vstride = vectors DIV nreg;
6  bits(32) vbase = X[v, 32];
7  integer vec = (UInt(vbase) + offset) MOD vstride;
8  bits(VL) result;
9
10 for r = 0 to nreg-1
11     bits(VL) operand1 = Z[n+r, VL];
12     bits(VL) operand2 = Z[m+r, VL];
13     bits(VL) operand3 = ZAvector[vec, VL];
14     for e = 0 to elements-1
15         bits(esize) sum = Elem[operand3, e, esize];
16         for i = 0 to 3
17             integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
18             integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
19             sum = sum + element1 * element2;
20         Elem[result, e, esize] = sum;
21     ZAvector[vec, VL] = result;
22     vec = vec + vstride;
```

D1.1.241 UMAX (multiple and single vector)

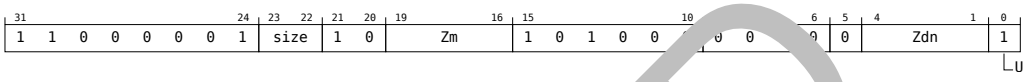
Multi-vector unsigned maximum by vector

Determine the unsigned maximum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

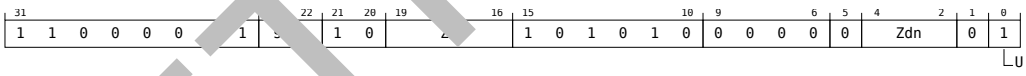
Two registers
(FEAT_SME2)




```
UMAX    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T>, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 2;
6  boolean unsigned = TRUE;
```

Four registers
(FEAT_SME2)




```
UMAX    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 4;
6  boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Max(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.242 UMAX (multiple vectors)

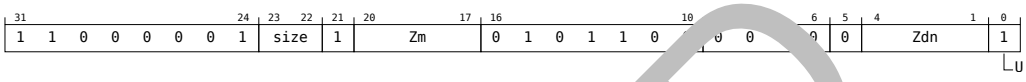
Multi-vector unsigned maximum

Determine the unsigned maximum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

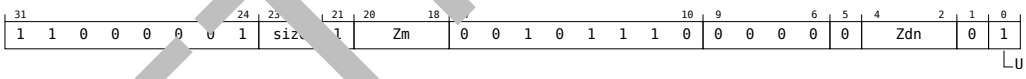
Two registers
(FEAT_SME2)




```
UMAX    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt(Zm:'0');
5  constant integer nreg = 2;
6  boolean unsigned = TRUE;
```

Four registers
(FEAT_SME2)




```
UMAX    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'00');
4  integer m = UInt(Zm:'00');
5  constant integer nreg = 4;
6  boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Max(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.243 UMIN (multiple and single vector)

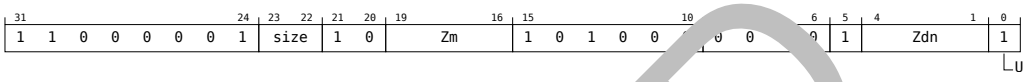
Multi-vector unsigned minimum by vector

Determine the unsigned minimum of elements of the second source vector and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

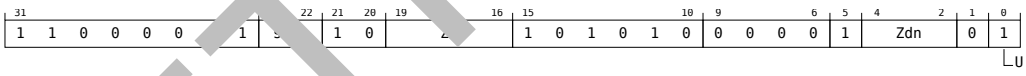
Two registers
(FEAT_SME2)




```
UMIN    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T>, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 2;
6  boolean unsigned = TRUE;
```

Four registers
(FEAT_SME2)




```
UMIN    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 4;
6  boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Min(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.244 UMIN (multiple vectors)

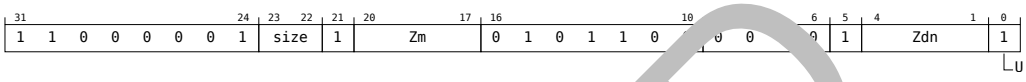
Multi-vector unsigned minimum

Determine the unsigned minimum of elements of the two or four second source vectors and the corresponding elements of the two or four first source vectors and destructively place the results in the corresponding elements of the two or four first source vectors.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

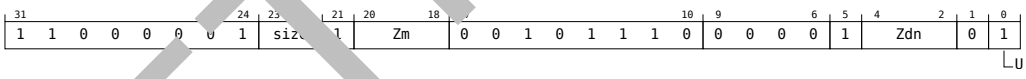
Two registers
(FEAT_SME2)




```
UMIN    { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt(Zm:'0');
5  constant integer nreg = 2;
6  boolean unsigned = TRUE;
```

Four registers
(FEAT_SME2)




```
UMIN    { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'00');
4  integer m = UInt(Zm:'00');
5  constant integer nreg = 4;
6  boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element1 = Int(Elem[operand1, e, esize], unsigned);
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer res = Min(element1, element2);
        Elem[results[r], e, esize] = res<esize-1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.245 UMLAL (multiple and indexed vector)

Multi-vector unsigned integer multiply-add long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA double-vector groups.

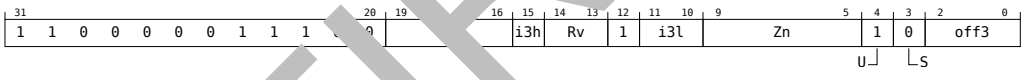
The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

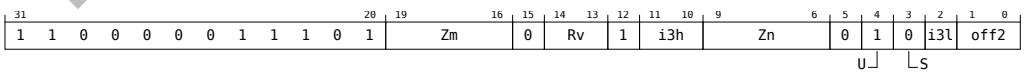
It has encodings from 3 classes: One ZA double-vector , Two ZA double-vectors and Four ZA double-vectors

One ZA double-vector
(FEAT_SME2)



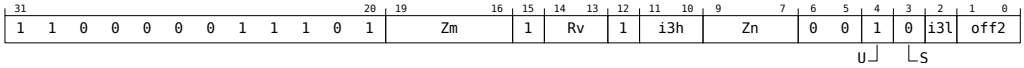

```
UMLAL   ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

Two ZA double-vectors
(FEAT_SME2)




```
UMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)




```
UMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, s, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.246 UMLAL (multiple and single vector)

Multi-vector unsigned integer multiply-add long by vector

The instruction operates on one, two, or four ZA double-vector groups.

This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector , Two ZA double-vectors and Four ZA double-vectors

One ZA double-vector
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14-13	12	11	10	9-5	4	3	2-0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	1	Zn	1	0	off3
																			U	S	

```
UMLAL   ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
constant integer nreg = 1;
```

Two ZA double-vectors
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14-13	12	11	10	9-5	4	3	2	1-0
1	1	0	0	0	0	0	1	0	1	1	0	Zm	0	Rv	0	1	0	Zn	1	0	0	off2
																			U	S		

```
UMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14-13	12	11	10	9-5	4	3	2	1-0
1	1	0	0	0	0	0	1	0	1	1	1	Zm	0	Rv	0	1	0	Zn	1	0	0	off2
																			U	S		

```
UMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.247 UMLAL (multiple vectors)

Multi-vector unsigned integer multiply-add long

The instruction operates on two or four ZA double-vector groups.

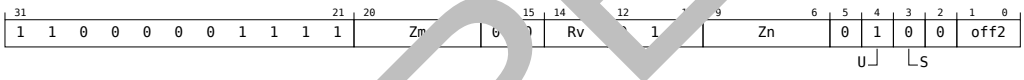
This unsigned integer multiply-add long instruction multiplies each unsigned 16-bit element in the two or four first source vectors with each unsigned 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible]d for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and F[illegible]r ZA dou[illegible]e-vector

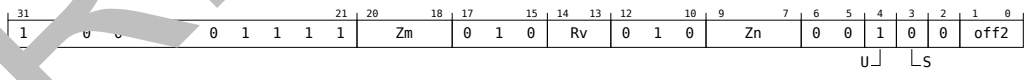
Two ZA double-vectors
(FEAT_SME2)




```
UMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(off2:'0');
constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)




```
UMLAL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec DIV 2);

for r = 0 to nreg-1
    bits(VL) operand3 = Z[n+r, VL];
    bits(VL) operand4 = Z[m+r, VL];
    for i = 0 to 1
        bits(VL) operand1 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.248 UMLALL (multiple and indexed vector)

Multi-vector unsigned integer multiply-add long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned integer multiply-add long long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all, half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

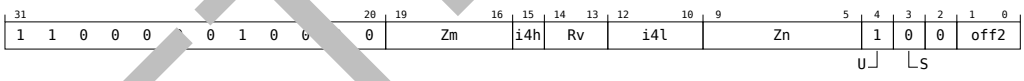
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: One ZA quad-vector of 32-bit elements , One ZA quad-vector of 64-bit elements , Two ZA quad-vectors of 32-bit elements , Two ZA quad-vectors of 64-bit elements , Four ZA quad-vectors of 32-bit elements and Four ZA quad-vectors of 64-bit elements

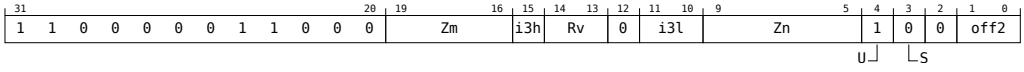
One ZA quad-vector of 32-bit elements
(FEAT_SME2)




```
UMLALL  ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.B, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('0':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

One ZA quad-vector of 64-bit elements
(FEAT_SME_I16I64)

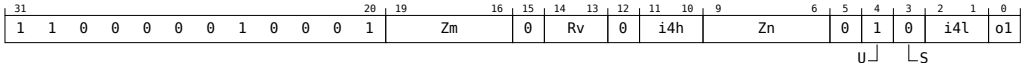



```
UMLALL  ZA.D[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
```

```
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

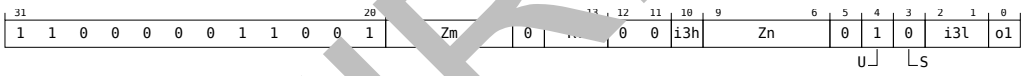
Two ZA quad-vectors of 32-bit elements (FEAT_SME2)




```
UMLALL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

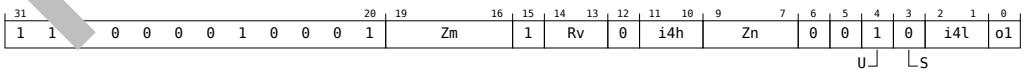
Two ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)




```
UMLALL  ZA.D[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

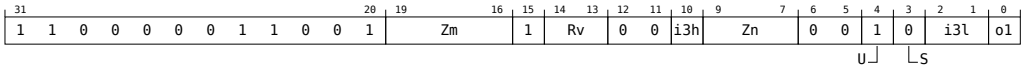
Four ZA quad-vectors of 32-bit elements (FEAT_SME2)




```
UMLALL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

Four ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)




```
UMLALL  ZA.D[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
1  if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
2  constant integer esize = 64;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn:'00');
5  integer m = UInt('0':Zm);
6  integer offset = UInt(o1:'00');
7  integer index = UInt(i3h:i3l);
8  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4 plus

For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegme
            integer s = 4 * segmentbase + index;
            integer element1 = UInt(Elem[operand1, 4
            integer element2 = UInt(Elem[operand2,
            bits(esize) product = (element1 * el
            Elem[result, e, esize] = Elem[opera
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.249 UMLALL (multiple and single vector)

Multi-vector unsigned integer multiply-add long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned integer multiply-add long long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

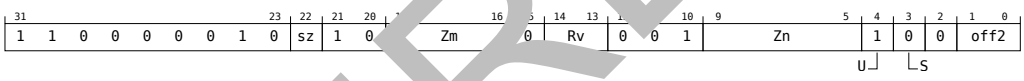
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: One ZA quad-vector , Two ZA quad-vectors and Four ZA quad-vectors

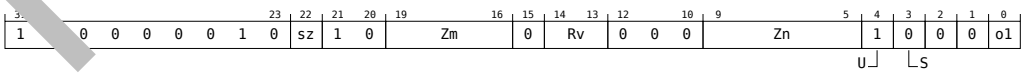
One ZA quad-vector
(FEAT_SME2)




```
UMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], <Zn>.<Tb>, <Zm>.<Tb>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3  constant integer esize = 32 << UInt(sz);
4  integer v = UInt('010':Rv);
5  integer n = UInt(Zn);
6  integer m = UInt('0':Zm);
7  integer offset = UInt(off2:'0');
8  constant integer nreg = 1;
```

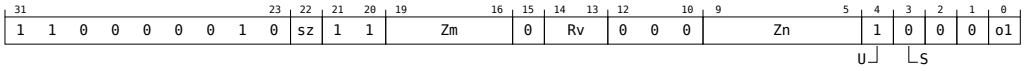
Two ZA quad-vectors
(FEAT_SME2)




```
UMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

```
1  if !HaveSME2() then UNDEFINED;
2  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3  constant integer esize = 32 << UInt(sz);
4  integer v = UInt('010':Rv);
5  integer n = UInt(Zn);
6  integer m = UInt('0':Zm);
7  integer offset = UInt(o1:'00');
8  constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
UMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector variant, is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in "sz":

sz	
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.250 UMLALL (multiple vectors)

Multi-vector unsigned integer multiply-add long long

The instruction operates on two or four ZA quad-vector groups.

This unsigned integer multiply-add long long instruction multiplies each unsigned 8-bit or 16-bit element in the two or four first source vectors with each unsigned 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively adds these values to the corresponding 32-bit or 64-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

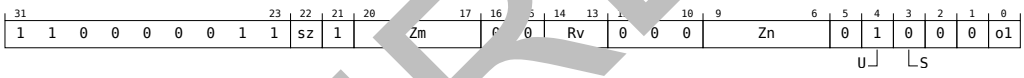
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA quad-vectors and four ZA quad-vectors

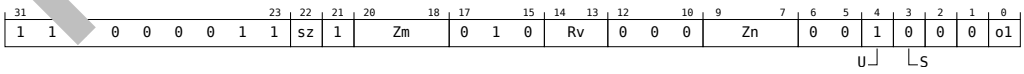
Two ZA quad-vectors
(FEAT_SME2)




```
UMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm2>.<Tb> }
```

```
1  if !HaveSME2() then UNDEFINED;
2  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3  constant integer esize = 32 << UInt(sz);
4  integer v = UInt('010':Rv);
5  integer n = UInt(Zn:'00');
6  integer m = UInt(Zm:'00');
7  integer offset = UInt(o1:'00');
8  constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
UMLALL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm4>.<Tb> }
```

```
1  if !HaveSME2() then UNDEFINED;
2  if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
3  constant integer esize = 32 << UInt(sz);
4  integer v = UInt('010':Rv);
5  integer n = UInt(Zn:'00');
6  integer m = UInt(Zm:'00');
7  integer offset = UInt(o1:'00');
8  constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
```

```
            integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.251 UMLSL (multiple and indexed vector)

Multi-vector unsigned integer multiply-subtract long by indexed element

The instruction operates on one, two, or four ZA double-vector groups.

This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit indexed element of the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the one, two, or four ZA double-vector groups.

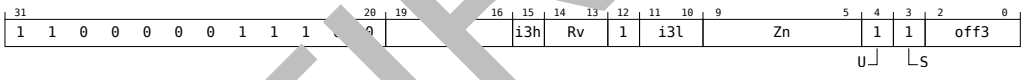
The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to 7, encoded in 3 bits. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

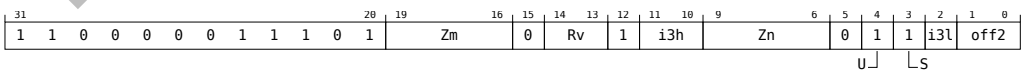
It has encodings from 3 classes: One ZA double-vector , Two ZA double-vectors and Four ZA double-vectors

One ZA double-vector
(FEAT_SME2)



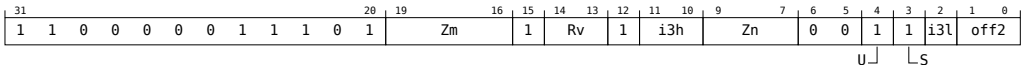

```
UMLSL   ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

Two ZA double-vectors
(FEAT_SME2)




```
UMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]

if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

Four ZA double-vectors (FEAT_SME2)




```
UMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Wv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 2 * segmentbase + index;
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, s, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.252 UMLSL (multiple and single vector)

Multi-vector unsigned integer multiply-subtract long by vector

The instruction operates on one, two, or four ZA double-vector groups.

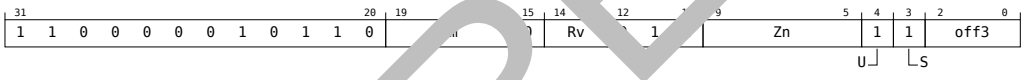
This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the one, two, or four first source vectors with each unsigned 16-bit element in the second source vector, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the one, two, or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA double-vector , Two [illegible] double-[illegible]tors and [illegible]our ZA double-vectors

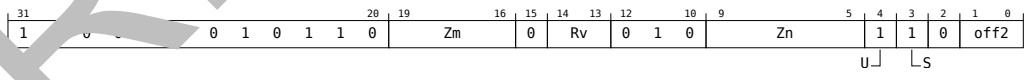
One ZA double-vector
(FEAT_SME2)




```
UMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn);
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off3:'0');
7  constant integer nreg = 2;
```

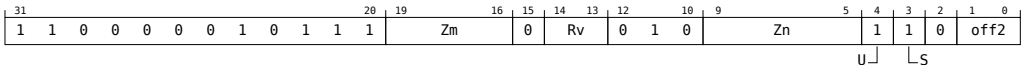
Two ZA [illegible]-vectors
(FEAT_SME2)




```
UMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn);
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off2:'0');
7  constant integer nreg = 2;
```

Four ZA double-vectors
(FEAT_SME2)



```
UMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA double-vector variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off3" field times 2.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> For the one ZA double-vector variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off3" field times 2 plus 1.

For the four ZA double-vectors and two ZA double-vectors variant: is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 2);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 1
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.253 UMLSL (multiple vectors)

Multi-vector unsigned integer multiply-subtract long

The instruction operates on two or four ZA double-vector groups.

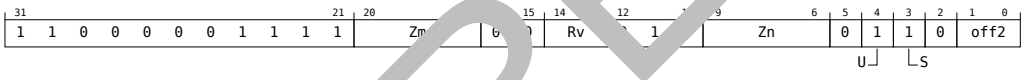
This unsigned integer multiply-subtract long instruction multiplies each unsigned 16-bit element in the two or four first source vectors with each unsigned 16-bit element in the two or four second source vectors, widens each product to 32-bits and destructively subtracts these values from the corresponding 32-bit elements of the two or four ZA double-vector groups. The lowest of the two consecutive vector numbers forming the double-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA double-vector groups respectively. The VECTOR GROUP symbol is pr[illegible] for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA double-vectors and F[illegible]r ZA dou[illegible]e-vector

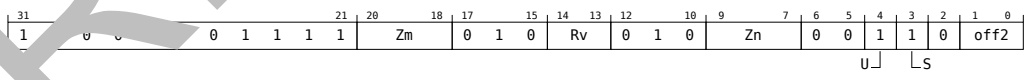
Two ZA double-vectors
(FEAT_SME2)




```
UMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, { <Zm1>.H-<Zm2>.H }
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn:'0');
5  integer m = UInt(Zm:'0');
6  integer offset = UInt(off2:'0');
7  constant integer nreg = 2;
```

Four ZA d[illegible]le-vec[illegible]s
(FEAT[illegible]ME2)




```
UMLSL   ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, { <Zm1>.H-<Zm4>.H }
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn:'00');
5  integer m = UInt(Zm:'00');
6  integer offset = UInt(off2:'0');
7  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of two consecutive vectors, encoded as "off2" field times 2.

<offsl> Is the vector select offset, pointing to last of two consecutive vectors, encoded as "off2" field times 2 plus 1.

<Zn1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA double-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec DIV 2);

for r = 0 to nreg-1
    bits(VL) operand3 = Z[n+r*vstride, VL];
    bits(VL) operand1 = Z[m+r*vstride, VL];
    for i = 0 to 1
        bits(VL) operand2 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.254 UMLSLL (multiple and indexed vector)

Multi-vector unsigned integer multiply-subtract long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned integer multiply-subtract long long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit indexed element of second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 3 to 4 bits depending on the size of the element. The lowest of the four consecutive vector numbers forming the quad-vector group within all, half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

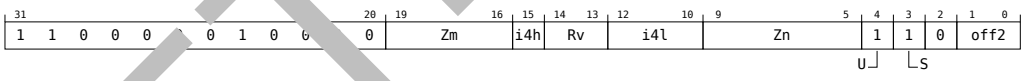
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 6 classes: One ZA quad-vector of 32-bit elements , One ZA quad-vector of 64-bit elements , Two ZA quad-vectors of 32-bit elements , Two ZA quad-vectors of 64-bit elements , Four ZA quad-vectors of 32-bit elements and Four ZA quad-vectors of 64-bit elements

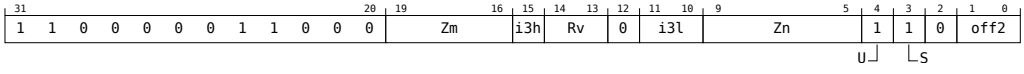
One ZA quad-vector of 32-bit elements
(FEAT_SME2)




```
UMLSLL  ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.B, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('0':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

One ZA quad-vector of 64-bit elements
(FEAT_SME_I16I64)

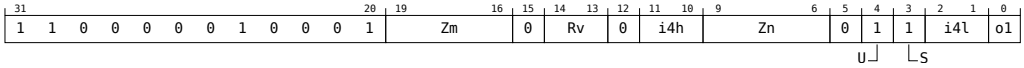



```
UMLSLL  ZA.D[<Wv>, <offsf>:<offsl>], <Zn>.H, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
```

```
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 1;
```

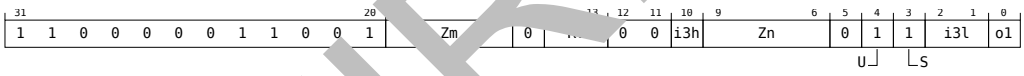
Two ZA quad-vectors of 32-bit elements (FEAT_SME2)




```
UMLSLL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

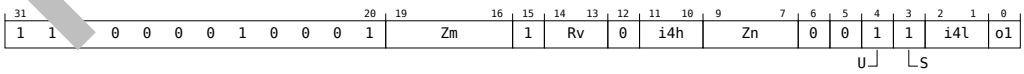
Two ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)




```
UMLSLL  ZA.D[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 2;
```

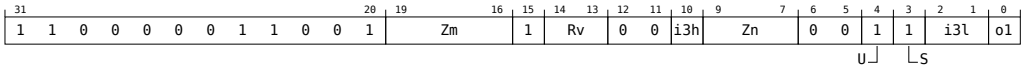
Four ZA quad-vectors of 32-bit elements (FEAT_SME2)




```
UMLSLL  ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

Four ZA quad-vectors of 64-bit elements (FEAT_SME_I16I64)




```
UMLSLL  ZA.D[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
constant integer esize = 64;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i3h:i3l);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector of 32-bit elements and one ZA quad-vector of 64-bit elements variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4 plus

For the four ZA quad-vectors of 32-bit elements, four ZA quad-vectors of 64-bit elements, two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA quad-vectors of 32-bit elements and two ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors of 32-bit elements and four ZA quad-vectors of 64-bit elements variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the four ZA quad-vectors of 32-bit elements, one ZA quad-vector of 32-bit elements and two ZA quad-vectors of 32-bit elements variant: is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

For the four ZA quad-vectors of 64-bit elements, one ZA quad-vector of 64-bit elements and two ZA quad-vectors of 64-bit elements variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegme
            integer s = 4 * segmentbase + index;
            integer element1 = UInt(Elem[operand1, 4
            integer element2 = UInt(Elem[operand2,
            bits(esize) product = (element1 * el
            Elem[result, e, esize] = Elem[opera
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.255 UMLSLL (multiple and single vector)

Multi-vector unsigned integer multiply-subtract long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned integer multiply-subtract long long instruction multiplies each unsigned 8-bit or 16-bit element in the one, two, or four first source vectors with each unsigned 8-bit or 16-bit element in the second source vector, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

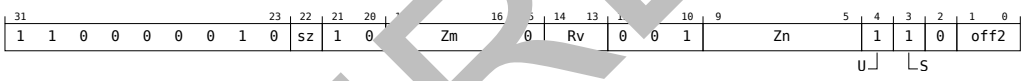
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 3 classes: One ZA quad-vector , Two ZA quad-vectors and Four ZA quad-vectors

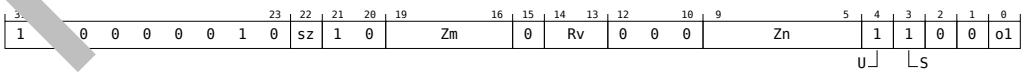
One ZA quad-vector
(FEAT_SME2)




```
UMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], <Zn>.<Tb>, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'0');
constant integer nreg = 1;
```

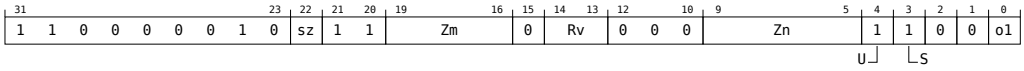
Two ZA quad-vectors
(FEAT_SME2)




```
UMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
UMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, <Zm>.<Tb>
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.256 UMLSLL (multiple vectors)

Multi-vector unsigned integer multiply-subtract long long

The instruction operates on two or four ZA quad-vector groups.

This unsigned integer multiply-subtract long long instruction multiplies each unsigned 8-bit or 16-bit element in the two or four first source vectors with each unsigned 8-bit or 16-bit element in the one, two, or four second source vectors, widens each product to 32-bits or 64-bits and destructively subtracts these values from the corresponding 32-bit or 64-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

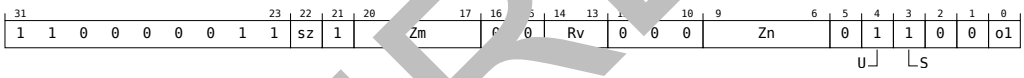
The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: Two ZA quad-vectors and four ZA quad-vectors

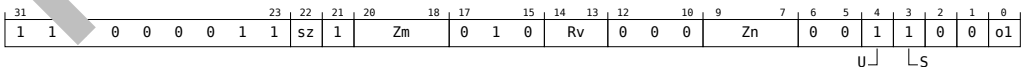
Two ZA quad-vectors
(FEAT_SME2)




```
UMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.<Tb>-<Zn2>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm2>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Four ZA quad-vectors
(FEAT_SME2)




```
UMLSLL  ZA.<T>[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.<Tb>-<Zn4>.<Tb> }, {
    ↪<Zm1>.<Tb>-<Zm4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
if sz == '1' && !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	S
1	D

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	B
1	H

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
```

```
            integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.257 UMOPA (2-way)

Unsigned integer sum of outer products and accumulate

This instruction works with a 32-bit element ZA tile.

The unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of unsigned 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively added to the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive [illegible]umn element[illegible] of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 [illegible]nsecutive ro[illegible] elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2
(FEAT_SME2)


```
UMOPA   <ZAda>.S, <Pn>/M, <P >/M, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEF ;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt( );
integer m = UInt( m);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean unsigned = TRUE;
```

Assembler Symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
```

```
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 1
            if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
                    ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
                prod = (Int(Elem[operand1, 2*row + k, esize DIV 2], unsigned) *
                        Int(Elem[operand2, 2*col + k, esize DIV 2], unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

D1.1.258 UMOPA (4-way)

Unsigned integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

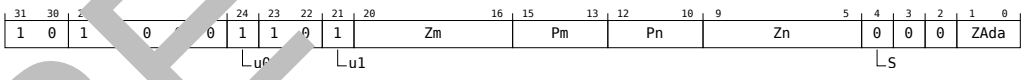
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

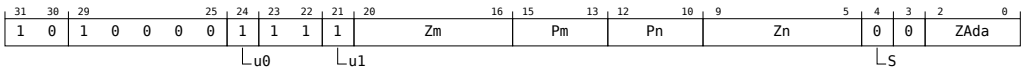
32-bit
(FEAT_SME)




```
UMOPA  <Za>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

64-bit
(FEAT_SME_I16I64)



```
UMOPA   <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
               ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.

D1.1.259 UMOPS (2-way)

Unsigned integer sum of outer products and subtract

This instruction works with a 32-bit element ZA tile.

The unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. The first source holds $SVL_S \times 2$ sub-matrix of unsigned 16-bit integer values, and the second source holds $2 \times SVL_S$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer sum of outer products is then destructively subtracted from the 32-bit integer destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix, and each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

SME2
(FEAT_SME2)


```
UMOPS   <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer a = UInt(Pn);
4  integer b = UInt(Pm);
5  integer n = UInt(Zn);
6  integer m = UInt(Zm);
7  integer da = UInt(ZAda);
8  boolean sub_op = TRUE;
9  boolean unsigned = TRUE;
```

Assembler Symbols

<ZAda> Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
1  CheckStreamingSVEAndZAEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer dim = VL DIV esize;
5  bits(PL) mask1 = P[a, PL];
6  bits(PL) mask2 = P[b, PL];
7  bits(VL) operand1 = Z[n, VL];
8  bits(VL) operand2 = Z[m, VL];
9  bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
10 bits(dim*dim*esize) result;
```

```
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 1
            if ActivePredicateElement(mask1, 2*row + k, esize DIV 2) &&
                    ActivePredicateElement(mask2, 2*col + k, esize DIV 2) then
                prod = (Int(Elem[operand1, 2*row + k, esize DIV 2], unsigned) *
                        Int(Elem[operand2, 2*col + k, esize DIV 2], unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

D1.1.260 UMOPS (4-way)

Unsigned integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

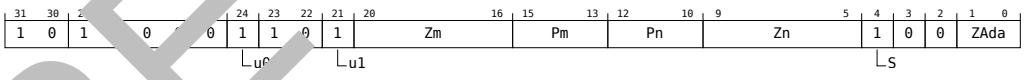
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

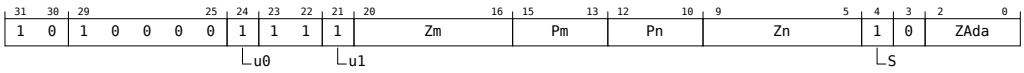
32-bit
(FEAT_SME)




```
UMOPS  <Za>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

64-bit
(FEAT_SME_I16I64)



```
UMOPS   <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
                ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.

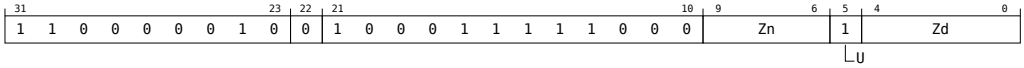
D1.1.261 UQCVT (two registers)

Multi-vector unsigned saturating extract narrow

Saturate the unsigned integer value in each element of the two source vectors to half the orginal source element width, and place the results in the half-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
UQCVT   <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for r = 0 to 1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        integer element = UInt(Elem[operand, e, 2 * esize]);
        Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

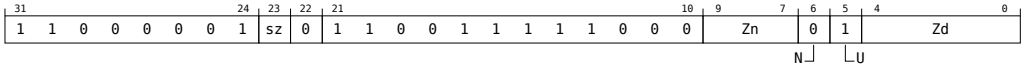
D1.1.262 UQCVT (four registers)

Multi-vector unsigned saturating extract narrow

Saturate the unsigned integer value in each element of the four source vectors to quarter the orginal source element width, and place the results in the quarter-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
UQCVT   <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        integer element = UInt(Elem[operand, e, 4 * esize]);
        Elem[result, r*elements + e, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

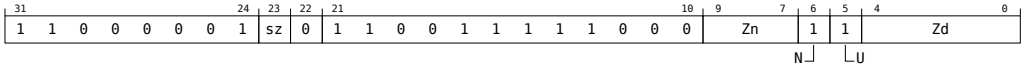
D1.1.263 UQCVTN

Multi-vector unsigned saturating extract narrow and interleave

Saturate the unsigned integer value in each element of the four source vectors to quarter the orginal source element width, and place the four-way interleaved results in the quarter-width destination elements.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
UQCVTN  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(sz);
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "sz":

sz	<T>
0	B
1	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "sz":

sz	<Tb>
0	S
1	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
bits(VL) result;

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        integer element = UInt(Elem[operand, e, 4 * esize]);
        Elem[result, 4*e + i, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

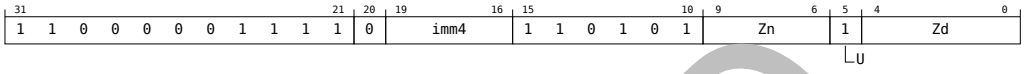
D1.1.264 UQRSHR (two registers)

Multi-vector unsigned saturating rounding shift right narrow by immediate

Shift right by an immediate value, the unsigned integer value in each element of the two source vectors and place the rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
UQRSHR  <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
integer shift = esize - UInt(imm4);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;
integer round_const = 1 << (shift-1);

for r = 0 to 1
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(2 * esize) element = Elem[operand, e, 2 * esize];
        integer res = (UInt(element) + round_const) >> shift;
        Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

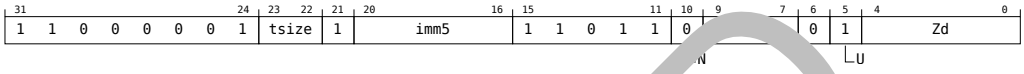
D1.1.265 UQRSHR (four registers)

Multi-vector unsigned saturating rounding shift right narrow by immediate

Shift right by an immediate value, the unsigned integer value in each element of the four source vectors and place the rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
UQRSHR  <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer esize;
3  case tsize of
4      when '00' UNDEFINED;
5      when '01' esize = 8;
6      when '1x' esize = 16;
7  integer n = UInt(Zn:'00');
8  integer d = UInt(Zd);
9  integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

Operation

```
1  CheckStreamingSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer elements = VL DIV (4 * esize);
```

```
bits(VL) result;
integer round_const = 1 << (shift-1);

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    for e = 0 to elements-1
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (UInt(element) + round_const) >> shift;
        Elem[result, r*elements + e, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

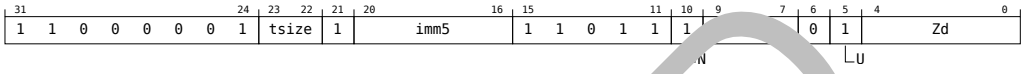
D1.1.266 UQRSHRN

Multi-vector unsigned saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the unsigned integer value in each element of the four source vectors and place the four-way interleaved rounded results in the quarter-width destination elements. Each result element is saturated to the quarter-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per source element.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
UQRSHRN <Zd>.<T>, { <Zn1>.<Tb>-<Zn4>.<Tb> }, #<const>
```

```
if !HaveSME2() then UNDEFINED;
integer esize;
case tsize of
    when '00' UNDEFINED;
    when '01' esize = 8;
    when '1x' esize = 16;
integer n = UInt(Zn:'00');
integer d = UInt(Zd);
integer shift = (8 * esize) - UInt(tsize:imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tsize":

tsize	<T>
00	RESERVED
01	B
1x	H

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Tb> Is the size specifier, encoded in "tsize":

tsize	<Tb>
00	RESERVED
01	S
1x	D

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<const> Is the immediate shift amount, in the range 1 to number of bits per source element, encoded in "tsize:imm5".

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (4 * esize);
```

```
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    for i = 0 to 3
        bits(VL) operand = Z[n+i, VL];
        bits(4 * esize) element = Elem[operand, e, 4 * esize];
        integer res = (UInt(element) + round_const) >> shift;
        Elem[result, 4*e + i, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

D1.1.267 URSHL (multiple and single vector)

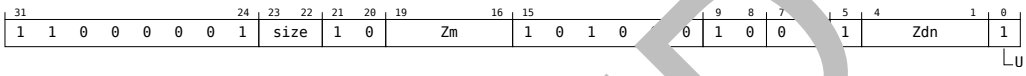
Multi-vector unsigned rounding shift left by vector

Shift active unsigned elements of the two or four first source vectors by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

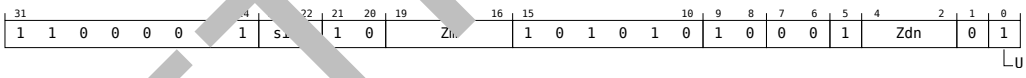
Two registers
(FEAT_SME2)




```
URSHL   { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
URSHL   { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt('0':Zm);
5  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as

"Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for e = 0 to elements-1
        integer element =  UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, es  ]), esize);
        integer res;
        if shift >= 0 then
            res = element << shift;
        else
            shift = -shift;
            res = (element + (1 << (shift - 1))) >>  ift;
        Elem[results[r], e, esize] = res<  1:0>;

for r = 0 to nreg-1
    Z[dn+r, VL] = results[r];
```

D1.1.268 URSHL (multiple vectors)

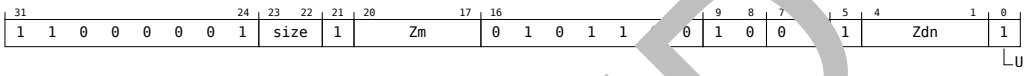
Multi-vector unsigned rounding shift left

Shift active unsigned elements of the two or four first source vectors by corresponding elements of the two or four second source vectors and destructively place the rounded results in the corresponding elements of the two or four first source vectors. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed.

This instruction is unpredicated.

It has encodings from 2 classes: Two registers and Four registers

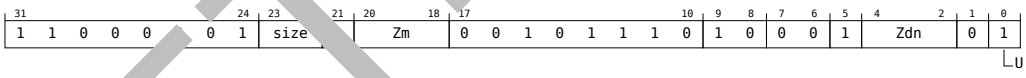
Two registers
(FEAT_SME2)




```
URSHL   { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zdn1>.<T>-<Zdn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'0');
4  integer m = UInt(Zm:'0');
5  constant integer nreg = 2;
```

Four registers
(FEAT_SME2)




```
URSHL   { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zdn1>.<T>-<Zdn4>.<T> }, { <Zm1>.<T>-<Zm4>.<T>
    ↪}
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer dn = UInt(Zdn:'00');
4  integer m = UInt(Zm:'00');
5  constant integer nreg = 4;
```

Assembler Symbols

<Zdn1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zdn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zdn" times 4 plus 3.

<Zdn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zdn" times 2 plus 1.

<Zm1> For the two registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four registers variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
array [0..3] of bits(VL) results;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[dn+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for e = 0 to elements-1
        integer element =  UInt(Elem[op  an  , e, es  e]);
        integer shift = ShiftSat( nt(Ele  perand2, e, esize]), esize);
        integer res;
        if shift >= 0 then
            res = element  shift;
        else
            shift = -sh ft;
            res = (e ement + (1  (shift - 1))) >> shift;
        Elem[resul  [r], e, esize]  res<esize-1:0>;

for r = 0 to n g-1
    Z[dn+r, VL]  res ts[r];
```

D1.1.269 USDOT (multiple and indexed vector)

Multi-vector unsigned by signed integer dot-product by indexed element

The instruction operates on two or four ZA single-vector groups.

The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups.

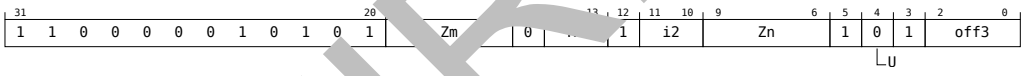
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter th[illegible]ber of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA ope[illegible]d consists o[illegible]vo or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for dis[illegible]embly, but op[illegible]onal in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors a[illegible] Four [illegible]A single-vectors

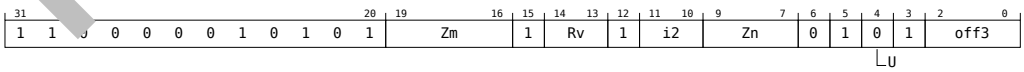
Two ZA single-vectors
(FEAT_SME2)




```
USDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg
```

Four [illegible]A single-vector
(FEAT_[illegible]E2)




```
USDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z[illegible], encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offs) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = [illegible] VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        integer segmentbase = e - (e MOD eltspersegment);
        integer index = segmentbase + index;
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.270 USDOT (multiple and single vector)

Multi-vector unsigned by signed integer dot-product by vector

The instruction operates on two or four ZA single-vector groups.

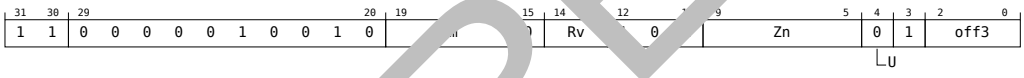
The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser[illegible]t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F[illegible] ZA sing[illegible]vectors

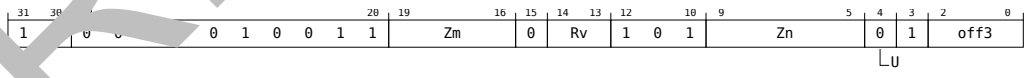
Two ZA single-vectors
(FEAT_SME2)




```
USDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 2;
```

Four ZA single-vectors
(FEAT_SME2)




```
USDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off3);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * r + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.271 USDOT (multiple vectors)

Multi-vector unsigned by signed integer dot-product

The instruction operates on two or four ZA single-vector groups.

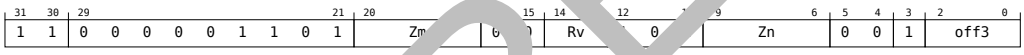
The unsigned by signed integer dot product instruction computes the dot product of four unsigned 8-bit integer values held in each 32-bit element of the two or four first source vectors and four signed 8-bit integer values in the corresponding 32-bit element of the two or four second source vectors. The widened dot product result is destructively added to corresponding 32-bit element of the two or four ZA single-vector groups. The vector numbers forming the single-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA single-vector groups respectively. The VECTOR GROUP symbol is preferred for disasser[illegible]t optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA single-vectors and F[illegible] ZA sing[illegible]vectors

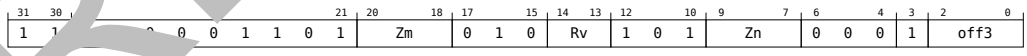
Two ZA single-vectors
(FEAT_SME2)




```
USDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  constant integer esize = 32;
4  integer n = UInt(Zn:'0');
5  integer m = UInt(Zm:'0');
6  integer offset = UInt(off3);
7  constant integer nreg = 2;
```

Four ZA single-vec[illegible]
(FEAT_S[illegible]




```
USDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }
```

```
1  if !HaveSME2() then UNDEFINED;
2  integer v = UInt('010':Rv);
3  constant integer esize = 32;
4  integer n = UInt(Zn:'00');
5  integer m = UInt(Zm:'00');
6  integer offset = UInt(off3);
7  constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA single-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;

for r = 0 to nreg-1
    bits(VL) operand1 = Z[r, VL];
    bits(VL) operand2 = Z[m, VL];
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.272 USMLALL (multiple and indexed vector)

Multi-vector unsigned by signed integer multiply-add long long by indexed element

The instruction operates on one, two, or four ZA quad-vector groups.

This unsigned by signed integer multiply-add long long instruction multiplies each unsigned 8-bit element in the one, two, or four first source vectors with each signed 8-bit indexed element of the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA quad-vector groups.

The elements within the second source vector are specified using an immediate element index which selects the same element position within each 128-bit vector segment. The element index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 4 bits. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector, Two ZA quad-vectors and Four ZA quad-vectors

One ZA quad-vector
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14-13	12-10	9-5	4	3	2	1-0
1	1	0	0	0	0	0	1	0	0	0	0	Zm	i4h	Rv	i4l	Zn	0	0	1	off2
																	U	S		

```
USMLALL ZA.S[<Wv>, <offsf>:<offsl>], <Zn>.B, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(off2:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 1;
```

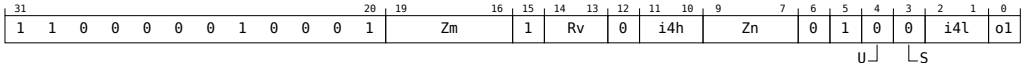
Two ZA quad-vectors
(FEAT_SME2)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15	14-13	12	11-10	9-6	5	4	3	2-1	0
1	1	0	0	0	0	0	1	0	0	0	1	Zm	0	Rv	0	i4h	Zn	1	0	0	i4l	o1
																			U	S		

```
USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 2;
```

Four ZA quad-vectors (FEAT_SME2)




```
USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
integer index = UInt(i4h:i4l);
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> For the one ZA quad-vector variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "off2" field times 4 plus 3.

For the four ZA quad-vectors and two ZA quad-vectors variant: is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 15, encoded in the "i4h:i4l" fields.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec MOD 4);

for r = 0 to nreg-1
```

```
    bits(VL) operand1 = Z[n+r, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer segmentbase = e - (e MOD eltspersegment);
            integer s = 4 * segmentbase + index;
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, s, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.273 USMLALL (multiple and single vector)

Multi-vector unsigned by signed integer multiply-add long long by vector

The instruction operates on one, two, or four ZA quad-vector groups.

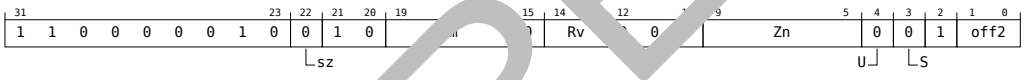
This unsigned by signed integer multiply-add long long instruction multiplies each unsigned 8-bit element in the one, two, or four first source vectors with each signed 8-bit element in the second source vector, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the one, two, or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within all, each half, or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo all, half, or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 3 classes: One ZA quad-vector , Two ZA quad-vectors and Four ZA quad-vectors

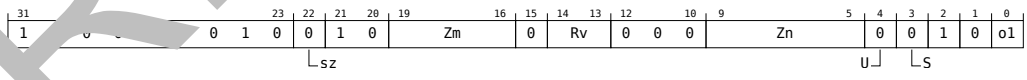
One ZA quad-vector
(FEAT_SME2)




```
USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn);
5  integer m = UInt('0':Zm);
6  integer offset = UInt(off2:'00');
7  constant integer nreg = 2;
```

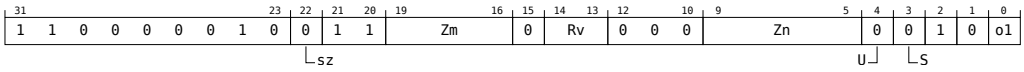
Two ZA quad-vectors
(FEAT_SME2)




```
USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, <Zm>.B
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 32;
3  integer v = UInt('010':Rv);
4  integer n = UInt(Zn);
5  integer m = UInt('0':Zm);
6  integer offset = UInt(o1:'00');
7  constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)



```
USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn);
integer m = UInt('0':Zm);
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> For the one ZA quad-vector variant: is the vector select offset, ... to first of four consecutive vectors, encoded as "off2" field times 4.

For the four ZA quad-vectors and two ZA quad-vectors varia... is the vector ...lect offset, pointing to first of four consecutive vectors, encoded as ... " field ...mes 4.

<offsl> For the one ZA quad-vector variant: is the vecto... select off...et, po...i... to last of four consecutive vectors, encoded as "off2" field times ...plus 3.

For the four ZA quad-vectors and two ZA quad-vecto... variant: i... ...e vector select offset, pointing to last of four consecutive vecto..., enco...ed as "... " fie... times 4 plus 3.

<Zn> Is the name of the first source scalab... vector re...ster, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable ...ector re...ter of a multi-vector sequence, encoded as "Zn".

<Zn4> Is the name of the fourth sca...ble ve...r regis... of a multi-vector sequence, encoded as "Zn" plus 3 modulo 32.

<Zn2> Is the name of the s...ond ...lable vector ...gister of a multi-vector sequence, encoded as "Zn" plus 1 modulo 32.

<Zm> Is the name o...he second source s...able vector register Z0-Z15, encoded in the "Zm" field.

Operation

```
CheckStre...gSVEA...ZAEnabl...();
consta... inte...er VL ...Cur...ntVL;
const...nt inte...er eleme... = VL DIV esize;
int...er vect...s = VL DIV 8;
inte... vs...ide ...ors DIV nreg;
bits(32...base = X[v, 32];
integer ... = (UInt(vbase) + offset) MOD vstride;
bits(VL) r...lt;
vec = vec - ...vec MOD 4);

for r = 0 to nreg-1
    bits(VL) operand1 = Z[(n+r) MOD 32, VL];
    bits(VL) operand2 = Z[m, VL];
    for i = 0 to 3
        bits(VL) operand3 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.274 USMLALL (multiple vectors)

Multi-vector unsigned by signed integer multiply-add long long

The instruction operates on two or four ZA quad-vector groups.

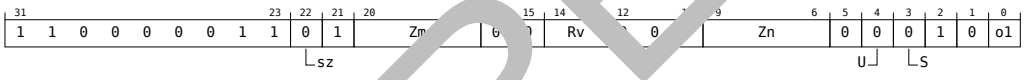
This unsigned by signed integer multiply-add long long instruction multiplies each unsigned 8-bit element in the two or four first source vectors with each signed 8-bit element in the two or four second source vectors, widens each product to 32-bits and destructively adds these values to the corresponding 32-bit elements of the two or four ZA quad-vector groups. The lowest of the four consecutive vector numbers forming the quad-vector group within each half or each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half or quarter the number of ZA array vectors.

The VECTOR GROUP symbol, VGx2 or VGx4, indicates that the ZA operand consists of two or four ZA quad-vector groups respectively. The VECTOR GROUP symbol is preferred for disassembly but optional in assembler source code.

This instruction is unpredicated.

It has encodings from 2 classes: Two ZA quad-vectors and Four ZA quad-vectors

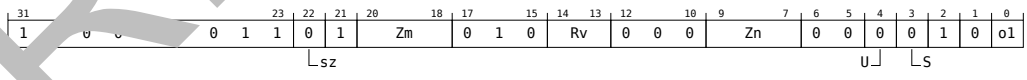
Two ZA quad-vectors
(FEAT_SME2)




```
USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx2}], { <Zn1>.B-<Zn2>.B }, { <Zm1>.B-<Zm2>.B }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'0');
integer m = UInt(Zm:'0');
integer offset = UInt(o1:'00');
constant integer nreg = 2;
```

Four ZA quad-vectors
(FEAT_SME2)




```
USMLALL ZA.S[<Wv>, <offsf>:<offsl>{, VGx4}], { <Zn1>.B-<Zn4>.B }, { <Zm1>.B-<Zm4>.B }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer v = UInt('010':Rv);
integer n = UInt(Zn:'00');
integer m = UInt(Zm:'00');
integer offset = UInt(o1:'00');
constant integer nreg = 4;
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offsf> Is the vector select offset, pointing to first of four consecutive vectors, encoded as "o1" field times 4.

<offsl> Is the vector select offset, pointing to last of four consecutive vectors, encoded as "o1" field times 4 plus 3.

<Zn1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm1> For the two ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 2.

For the four ZA quad-vectors variant: is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zm" times 4.

<Zm4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zm" times 4 plus 3.

<Zm2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zm" times 2 plus 1.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV nreg;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) result;
vec = vec - (vec DIV 4);

for r = 0 to nreg-1
    bits(VL) operand3 = Z[n+r, VL];
    bits(VL) operand2 = Z[m+r, VL];
    for i = 0 to 3
        bits(VL) operand1 = ZAvector[vec + i, VL];
        for e = 0 to elements-1
            element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            bits(esize) product = (element1 * element2)<esize-1:0>;
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
        ZAvector[vec + i, VL] = result;
    vec = vec + vstride;
```

D1.1.275 USMOPA

Unsigned by signed integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

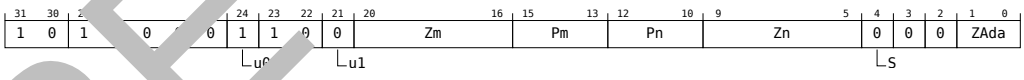
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

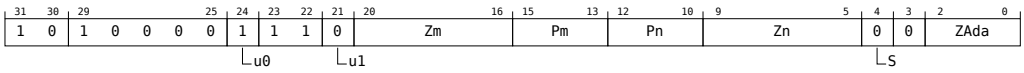
32-bit
(FEAT_SME)




```
USMOPA  <Zda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

64-bit
(FEAT_SME_I16I64)



```
USMOPA  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
                ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

– The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
– The values of the NZCV flags.

D1.1.276 USMOPS

Unsigned by signed integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

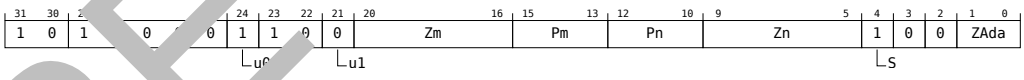
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

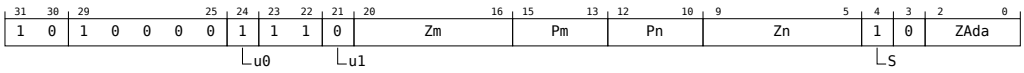
32-bit
(FEAT_SME)




```
USMOPS  <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

64-bit
(FEAT_SME_I16I64)



```
USMOPS  <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H
```

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<ZAda> For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.

For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.

<Pn> Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ActivePredicateElement(mask1, 4*row + k, esize DIV 4) &&
               ActivePredicateElement(mask2, 4*col + k, esize DIV 4) then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
- The values of the NZCV flags.

D1.1.277 USVDOT

Multi-vector unsigned by signed integer vertical dot-product by indexed element

The instruction operates on four ZA single-vector groups.

The unsigned by signed integer vertical dot product instruction computes the vertical dot product of corresponding unsigned 8-bit elements from the four first source vectors and four signed 8-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product result is destructively added to the corresponding 32-bit element of four ZA single-vector groups.

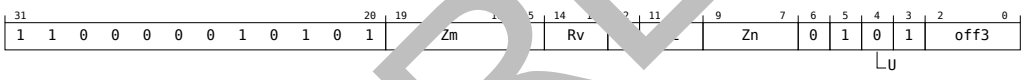
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

The VECTOR GROUP symbol VGx4 indicates that the ZA operand consists of four ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
USVDOT  ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Wv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 4;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 3
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = UInt(Elem[operand1, 4 * e + r, esize DIV 4]);
            integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.278 UUNPK

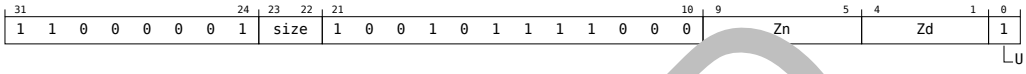
Unpack and zero-extend multi-vector elements

Unpack elements from one or two source vectors and then zero-extend them to place in elements of twice their size within the two or four destination vectors.

This instruction is unpredicated.

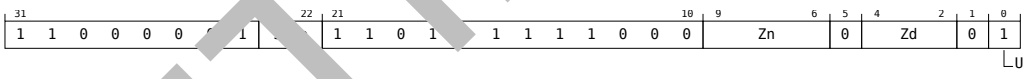
It has encodings from 2 classes: Two registers and Four registers

Two registers
(FEAT_SME2)




```
UUNPK   { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<Tb>

if !HaveSME2() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd:'0');
constant integer nreg = 2;
boolean unsigned = TRUE;
```

Four registers
(FEAT_SME2)




```
UUNPK   { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<Tb>-<Zn2>.<Tb> }

if !HaveSME2() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn:'0');
integer d = UInt(Zd:'00');
constant integer nreg = 4;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd1> For the two registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

For the four registers variant: is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
constant integer sreg = nreg DIV 2;
array [0..3] of bits(VL) results;

for r = 0 to sreg-1
    bits(VL) operand = Z[n+r, VL];
    for i = 0 to 1
        for e = 0 to elements-1
            bits(hsize) element = Elem[operand, i*elements + e, hsize];
            Elem[results[2*r+i], e, esize] = Extend(element, esize, unsigned);

for r = 0 to sreg-1
    Z[d+r, VL] = results[r];
```

D1.1.279 UVDOT (2-way)

Multi-vector unsigned integer vertical dot-product by indexed element

The instruction operates on two ZA single-vector groups.

The unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding two unsigned 16-bit integer values held in the two first source vectors and two unsigned 16-bit integer values in the corresponding indexed 32-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit element of two ZA single-vector groups.

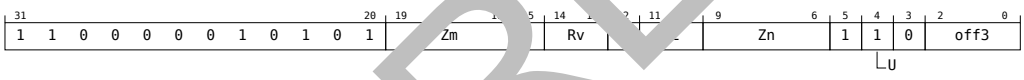
The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3, encoded in 2 bits.

The vector numbers forming the single-vector group within each half of the ZA array are selected by the sum of the vector select register and immediate offset, modulo half the number of ZA array vectors.

The VECTOR GROUP symbol VGx2 indicates that the ZA operand consists of two ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

SME2
(FEAT_SME2)




```
UVDOT   ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'0');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> Is the element index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 2;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
```

```
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 1
    bits(VL) operand3 = ZAvector[vec, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 1
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = UInt(Elem[operand1, 2 * e + r, esize DIV 2]);
            integer element2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, VL] = result;
    vec = vec + vstride;
```

D1.1.280 UVDOT (4-way)

Multi-vector unsigned integer vertical dot-product by indexed element

The instruction operates on four ZA single-vector groups.

The unsigned integer vertical dot product instruction computes the vertical dot product of the corresponding four unsigned 8-bit or 16-bit integer values held in the four first source vectors and four unsigned 8-bit or 16-bit integer values in the corresponding indexed 32-bit or 64-bit element of the second source vector. The widened dot product results are destructively added to the corresponding 32-bit or 64-bit element of the four ZA single-vector groups.

The groups within the second source vector are specified using an immediate element index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group.

The vector numbers forming the single-vector group within each quarter of the ZA array are selected by the sum of the vector select register and immediate offset, modulo quarter the number of ZA array vectors.

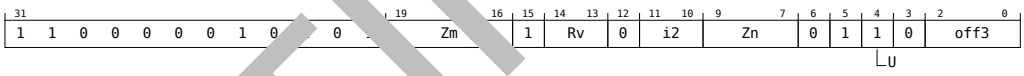
The VECTOR GROUP symbol VGx4 indicates that the ZA operand consists of four ZA single-vector groups. The VECTOR GROUP symbol is preferred for disassembly, but optional in assembler source code.

This instruction is unpredicated.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit to 64-bit variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

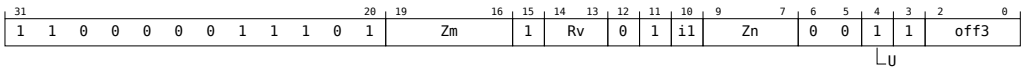
32-bit
(FEAT_SME2)




```
UVDOT   ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.B-<Zn4>.B }, <Zm>.B[<index>]
```

```
if !HaveSME2() then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 32;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i2);
```

64-bit
(FEAT_SME_I16I64)




```
UVDOT   ZA.D[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

```
if !(HaveSME2() && HaveSMEI16I64()) then UNDEFINED;
integer v = UInt('010':Rv);
constant integer esize = 64;
integer n = UInt(Zn:'00');
integer m = UInt('0':Zm);
integer offset = UInt(off3);
integer index = UInt(i1);
```

Assembler Symbols

<Wv> Is the 32-bit name of the vector select register W8-W11, encoded in the "Rv" field.

<offs> Is the vector select offset, in the range 0 to 7, encoded in the "off3" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

<Zm> Is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<index> For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer vectors = VL DIV 8;
integer vstride = vectors DIV 4;
integer eltspersegment = 128 DIV esize;
bits(32) vbase = X[v, 32];
integer vec = (UInt(vbase) + offset) MOD vstride;
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for r = 0 to 3
    bits(VL) operand3 = ZAvector[v, VL];
    for e = 0 to elements-1
        integer segmentbase = e - (e MOD eltspersegment);
        integer s = segmentbase + index;
        bits(esize) sum = Elem[operand3, e, esize];
        for i = 0 to 3
            bits(VL) operand1 = Z[n+i, VL];
            integer element1 = UInt(Elem[operand1, 4 * e + r, esize DIV 4]);
            integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
            sum = sum + element1 * element2;
        Elem[result, e, esize] = sum;
    ZAvector[vec, ] = result;
    vec = vec + vstride;
```

D1.1.281 UZP (four registers)

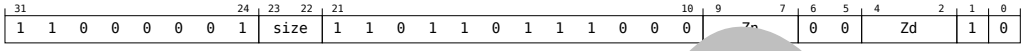
Concatenate elements from four vectors

Concatenate every fourth element from each of the four source vectors and place them in the corresponding elements of the four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

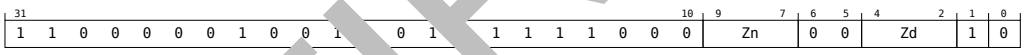
8-bit to 64-bit elements
(FEAT_SME2)




```
UZP     { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<T>-<Zn4>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer n = UInt(Zn:'00');
4  integer d = UInt(Zd:'00');
```

128-bit element
(FEAT_SME2)




```
UZP     { <Zd1>.Q-<Zd4>.Q }, { <Zn1>.Q-<Zn4>.Q }
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 128;
3  integer n = UInt(Zn:'00');
4  integer d = UInt(Zd:'00');
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Z" times

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 4 then UNDEFINED;
constant integer quads = VL DIV (esize * 4);
bits(VL) result0;
bits(VL) result1;
bits(VL) result2;
bits(VL) result3;

for r = 0 to 3
    bits(VL) operand = Z[n+r, VL];
    integer base = r * quads;
    for q = 0 to quads-1
        Elem[result0, base+q, esize] = Elem[operand, 4*q+0, esize];
        Elem[result1, base+q, esize] = Elem[operand, 4*q+1, esize];
        Elem[result2, base+q, esize] = Elem[operand, 4*q+2, esize];
        Elem[result3, base+q, esize] = Elem[operand, 4*q+3, es

Z[d+0, VL] = result0;
Z[d+1, VL] = result1;
Z[d+2, VL] = result2;
Z[d+3, VL] = result3;
```

D1.1.282 UZP (two registers)

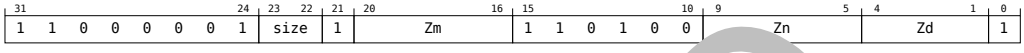
Concatenate elements from two vectors

Concatenate every second element from each of the first and second source vectors and place them in the corresponding elements of the two destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

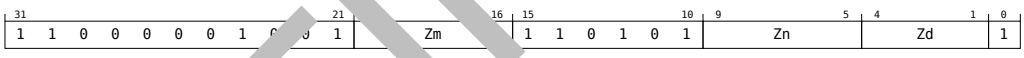
8-bit to 64-bit elements
(FEAT_SME2)




```
UZP     { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
```

128-bit element
(FEAT_SME2)




```
UZP     { <Zd1>.Q-<Zd2>.Q }, <Zn>.Q, <Zm>.Q
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd:'0');
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) result0;
bits(VL) result1;

for r = 0 to 1
    integer base = r * pairs;
    bits(VL) operand = if r == 0 then Z[n, VL] else Z[m, VL];
    for p = 0 to pairs-1
        Elem[result0, base+p, esize] = Elem[operand, 2*p+0, esize];
        Elem[result1, base+p, esize] = Elem[operand, 2*p+1, esize];

Z[d+0, VL] = result0;
Z[d+1, VL] = result1;
```

D1.1.283 WHILEGE

While decrementing signed scalar greater than or equal to scalar (predicate-as-counter)

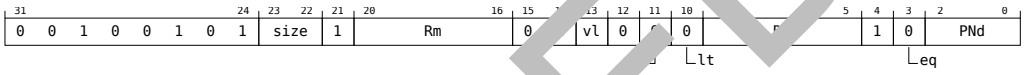
Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element of the group.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SME2
(FEAT_SME2)




```
WHILEGE <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = 64;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d = UInt('1':PNd);
7  boolean unsigned = FALSE;
8  boolean invert = TRUE;
9  SVECmp op = Cmp_GE;
10 integer width = 8 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

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

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

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 - 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.284 WHILEGT

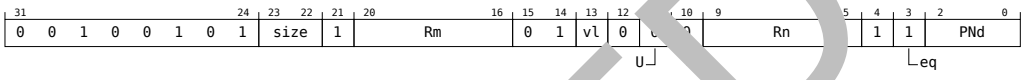
While decrementing signed scalar greater than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SME2
(FEAT_SME2)




```
WHILEGT <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = 64;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d = UInt('1':PNd);
7  boolean unsigned = FALSE;
8  boolean invert = TRUE;
9  SVECmp op = Cmp_GT;
10 integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

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

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

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
1  CheckStreamingSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
```

```
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 - 1;

result = EncodePredCount(esize, elements, count, invert, PL
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert)
P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.285 WHILEHI

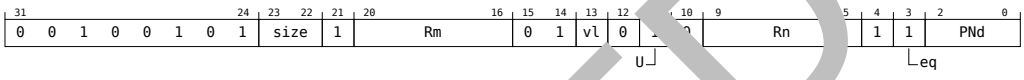
While decrementing unsigned scalar higher than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SME2
(FEAT_SME2)




```
WHILEHI <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = 64;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d = UInt('1':PNd);
7  boolean unsigned = TRUE;
8  boolean invert = TRUE;
9  SVECmp op = Cmp_GT;
10 integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

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

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

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
1  CheckStreamingSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
```

```
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 - 1;

result = EncodePredCount(esize, elements, count, invert, PL
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert)
P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.286 WHILEHS

While decrementing unsigned scalar higher or same as scalar (predicate-as-counter)

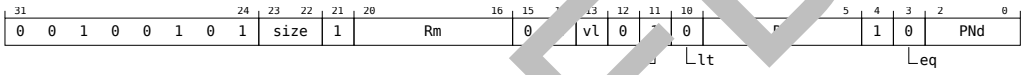
Generate a predicate for a group of two or four vectors that starting from the highest numbered element of the group is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element of the group.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SME2
(FEAT_SME2)




```
WHILEHS <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = 64;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d = UInt('1':PNd);
7  boolean unsigned = TRUE;
8  boolean invert = TRUE;
9  SVECmp op = Cmp_GE;
10 integer width = esize << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the element size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

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

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

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 - 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.287 WHILELE

While incrementing signed scalar less than or equal to scalar (predicate-as-counter)

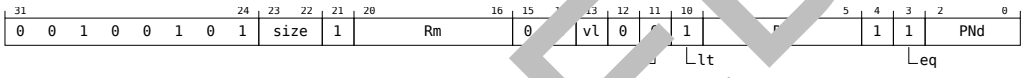
Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element of the group.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SME2
(FEAT_SME2)




```
WHILELE <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = 64;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d = UInt('1':PNd);
7  boolean unsigned = FALSE;
8  boolean invert = FALSE;
9  SVECmp op = Cmp_LE;
10 integer width = 8 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

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

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

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 + 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.288 WHILELO

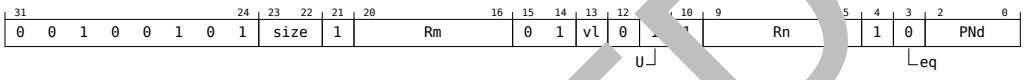
While incrementing unsigned scalar lower than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SME2
(FEAT_SME2)




```
WHILELO <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt('1':PNd);
boolean unsigned = TRUE;
boolean invert = FALSE;
SVECmp op = Cmp_LT;
integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

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

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

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
```

```
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 + 1;

result = EncodePredCount(esize, elements, count, invert, PL)
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert)
P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.289 WHILELS

While incrementing unsigned scalar lower or same as scalar (predicate-as-counter)

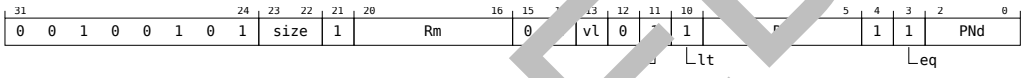
Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element of the group.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SME2
(FEAT_SME2)




```
WHILELS <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = 64;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d = UInt('1':PNd);
7  boolean unsigned = TRUE;
8  boolean invert = FALSE;
9  SVECmp op = Cmp_LE;
10 integer width = 8 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the element size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

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

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

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 + 1;

result = EncodePredCount(esize, elements, count, invert, PL);
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert);
P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.290 WHILELT

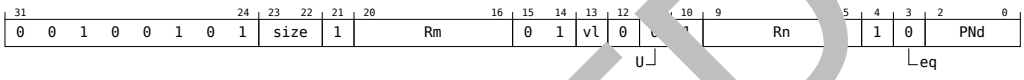
While incrementing signed scalar less than scalar (predicate-as-counter)

Generate a predicate for a group of two or four vectors that starting from the lowest numbered element of the group is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element of the group.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size.

The predicate result is placed in the predicate destination register using the predicate-as-counter encoding. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SME2
(FEAT_SME2)




```
WHILELT <PNd>.<T>, <Xn>, <Xm>, <vl>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = 64;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d = UInt('1':PNd);
7  boolean unsigned = FALSE;
8  boolean invert = FALSE;
9  SVECmp op = Cmp_LT;
10 integer width = 2 << UInt(vl);
```

Assembler Symbols

<PNd> Is the name of the destination scalable predicate register PN8-PN15, with predicate-as-counter encoding, encoded in the "PNd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

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

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

<vl> Is the vl specifier, encoded in "vl":

vl	<vl>
0	VLx2
1	VLx4

Operation

```
1  CheckStreamingSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
```

```
constant integer elements = width * (VL DIV esize);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;
integer count = 0;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    if last then count = count + 1;
    operand1 = operand1 + 1;

result = EncodePredCount(esize, elements, count, invert, PL)
PSTATE.<N,Z,C,V> = PredCountTest(elements, count, invert)
P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

D1.1.291 ZERO (tile)

Zero a list of 64-bit element ZA tiles

Zeroes all bytes within each of the up to eight listed 64-bit element tiles named ZA0.D to ZA7.D, leaving the other 64-bit element tiles unmodified.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

For programmer convenience an assembler must also accept the names of 32-bit, 16-bit, and 8-bit element tiles which are converted into the corresponding set of 64-bit element tiles.

In accordance with the architecturally defined mapping between different element size tiles:

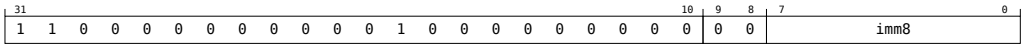
- Zeroing the 8-bit element tile name ZA0.B, or the entire array name, is equivalent to zeroing all eight 64-bit element tiles named ZA0.D to ZA7.D.
- Zeroing the 16-bit element tile name ZA0.H is equivalent to zeroing 64-bit element tiles named ZA0.D, ZA2.D, ZA4.D, and ZA6.D.
- Zeroing the 16-bit element tile name ZA1.H is equivalent to zeroing 64-bit element tiles named ZA1.D, ZA3.D, ZA5.D, and ZA7.D.
- Zeroing the 32-bit element tile name ZA0.S is equivalent to zeroing 64-bit element tiles named ZA0.D and ZA4.D.
- Zeroing the 32-bit element tile name ZA1.S is equivalent to zeroing 64-bit element tiles named ZA1.D and ZA5.D.
- Zeroing the 32-bit element tile name ZA2.S is equivalent to zeroing 64-bit element tiles named ZA2.D and ZA6.D.
- Zeroing the 32-bit element tile name ZA3.S is equivalent to zeroing 64-bit element tiles named ZA3.D and ZA7.D.

The preferred disassembly of this instruction uses the shortest list of tile names that represent the encoded immediate mask.

For example:

- An immediate which encodes 64-bit element tiles ZA0.D, ZA1.D, ZA4.D, and ZA5.D is disassembled as {ZA0.S, ZA1.S}.
- An immediate which encodes 64-bit element tiles ZA0.D, ZA2.D, ZA4.D, and ZA6.D is disassembled as {ZA0.H}.
- An all-ones immediate is disassembled as {ZA}.
- An all-zeros immediate is disassembled as an empty list { }.

SME
(FEAT_SME)




```
ZERO    { <mask> }
```

```
1  if !HaveSME() then UNDEFINED;
2  bits(8) mask = imm8;
3  constant integer esize = 64;
```

Assembler Symbols

<mask> Is a list of up to eight 64-bit element tile names separated by commas, encoded in the "imm8" field.

Operation

```
CheckSMEAndZAEnabled();
constant integer SVL = CurrentSVL;
constant integer dim = SVL DIV esize;
bits(dim*dim*esize) result = Zeros(dim*dim*esize);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

for i = 0 to 7
    if mask<i> == '1' then ZAtile[i, esize, dim*dim*esize] =
```

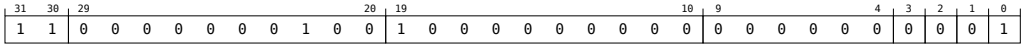
D1.1.292 ZERO (ZT0)

Zero ZT0

Zero all bytes of the ZT0 register.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME2
(FEAT_SME2)




```
ZERO    { ZT0 }
```

```
1  if !HaveSME2() then UNDEFINED;
```

Operation

```
1  CheckSMEEnabled();
2  CheckSMEZT0Enabled();
3
4  if HaveTME() && TSTATE.depth > 0 then
5      FailTransaction(TMFailure_ERR, FALSE);
6
7  ZT0[512] = Zeros(512);
```

D1.1.293 ZIP (four registers)

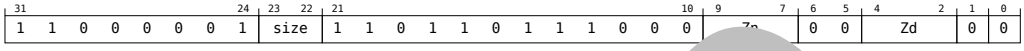
Interleave elements from four vectors

Place the four-way interleaved elements from the four source vectors in the corresponding elements of the four destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

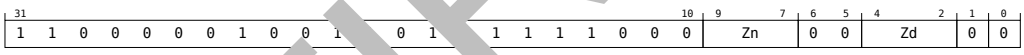
8-bit to 64-bit elements
(FEAT_SME2)




```
ZIP     { <Zd1>.<T>-<Zd4>.<T> }, { <Zn1>.<T>-<Zn4>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer n = UInt(Zn:'00');
4  integer d = UInt(Zd:'00');
```

128-bit element
(FEAT_SME2)




```
ZIP     { <Zd1>.Q-<Zd4>.Q }, { <Zn1>.Q-<Zn4>.Q }
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 128;
3  integer n = UInt(Zn:'00');
4  integer d = UInt(Zd:'00');
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Z... times...

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd4> Is the name of the fourth destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 4 plus 3.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 4.

<Zn4> Is the name of the fourth scalable vector register of a multi-vector sequence, encoded as "Zn" times 4 plus 3.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 4 then UNDEFINED;
constant integer quads = VL DIV (esize * 4);
bits(VL) operand0 = Z[n, VL];
bits(VL) operand1 = Z[n+1, VL];
bits(VL) operand2 = Z[n+2, VL];
bits(VL) operand3 = Z[n+3, VL];
bits(VL) result;

for r = 0 to 3
    integer base = r * quads;
    for q = 0 to quads-1
        Elem[result, 4*q+0, esize] = Elem[operand0, base+q, esize];
        Elem[result, 4*q+1, esize] = Elem[operand1, base+q, esize];
        Elem[result, 4*q+2, esize] = Elem[operand2, base+q, esize];
        Elem[result, 4*q+3, esize] = Elem[operand3, base+q, es
    Z[d+r, VL] = result;
```

D1.1.294 ZIP (two registers)

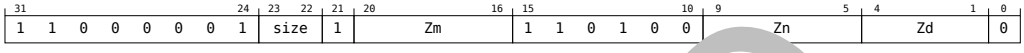
Interleave elements from two vectors

Place the two-way interleaved elements from the first and second source vectors in the corresponding elements of the two destination vectors.

This instruction is unpredicated.

It has encodings from 2 classes: 8-bit to 64-bit elements and 128-bit element

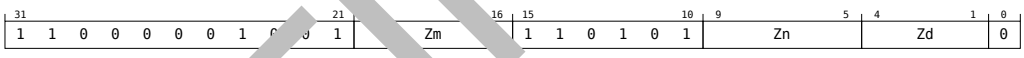
8-bit to 64-bit elements
(FEAT_SME2)




```
ZIP     { <Zd1>.<T>-<Zd2>.<T> }, <Zn>.<T>, <Zm>.<T>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer n = UInt(Zn);
4  integer m = UInt(Zm);
5  integer d = UInt(Zd:'0');
```

128-bit element
(FEAT_SME2)




```
ZIP     { <Zd1>.Q-<Zd2>.Q }, <Zn>.Q, <Zm>.Q
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 128;
3  integer n = UInt(Zn);
4  integer m = UInt(Zm);
5  integer d = UInt(Zd:'0');
```

Assembler Symbols

<Zd1> Is the name of the first destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zd2> Is the name of the second destination scalable vector register of a multi-vector sequence, encoded as "Zd" times 2 plus 1.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand0 = Z[n, VL];
bits(VL) operand1 = Z[m, VL];
bits(VL) result;

for r = 0 to 1
    integer base = r * pairs;
    for p = 0 to pairs-1
        Elem[result, 2*p+0, esize] = Elem[operand0, base+p, esize];
        Elem[result, 2*p+1, esize] = Elem[operand1, base+p, esize];
    Z[d+r, VL] = result;
```

D1.2 SVE2 data-processing instructions

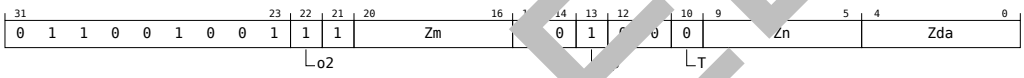
The following SVE2 instructions are added by the SME or SME2 architecture, and are available when the PE is in *Streaming SVE mode*.

D1.2.1 BFMLSLB (vectors)

BFloat16 floating-point multiply-subtract long from single-precision (bottom)

This BFloat16 floating-point multiply-subtract long instruction widens the even-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
BFMLSLB <Zda>.S, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 0, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    if op1_neg then element1 = BFNeg(element1);
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

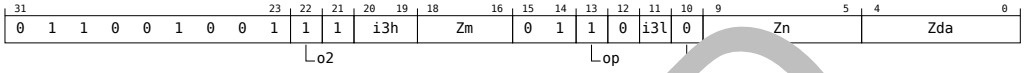
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.2 BFMLSLB (indexed)

BFloat16 floating-point multiply-subtract long from single-precision (bottom, indexed)

This BFloat16 floating-point multiply-subtract long instruction widens the even-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
BFMLSLB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    if op1_neg then element1 = BFNeg(element1);
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

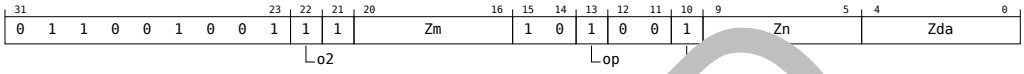
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.3 BFMLSLT (vectors)

BFloat16 floating-point multiply-subtract long from single-precision (top)

This BFloat16 floating-point multiply-subtract long instruction widens the odd-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
BFMLSLT <Zda>.S, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer = VL DIV ;
constant integer elements = VL DIV 32;
bits( VL) operand1 = Z[n, VL];
bits( VL) operand2 = Z[m, VL];
bits( VL) operand3 = Z[da, VL];
bits( VL) result;

for e = 0 elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 1, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    if op1_neg then element1 = BFNeg(element1);
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

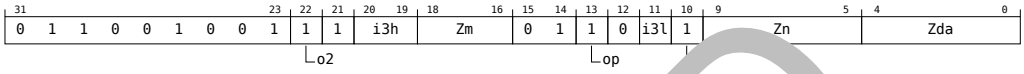
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.4 BFMLSLT (indexed)

BFloat16 floating-point multiply-subtract long from single-precision (top, indexed)

This BFloat16 floating-point multiply-subtract long instruction widens the odd-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
BFMLSLT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    if op1_neg then element1 = BFNeg(element1);
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

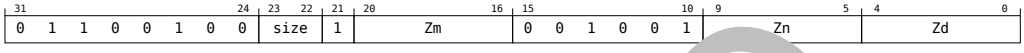
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.5 FCLAMP

Floating-point clamp to minimum/maximum number

Clamp each floating-point element in the destination vector to between the floating-point minimum value in the corresponding element of the first source vector and the floating-point maximum value in the corresponding element of the second source vector and destructively place the clamped results in the corresponding elements of the destination vector. If at least one element value contributing to a result is numeric and the others are either numeric or a quiet NaN, then the result is the numeric value. This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
FCLAMP  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSME2() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) result;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    Elem[result, e, esize] = FPMinNum(FPMaxNum(element1, element3, FPCR[]), element2,
        ↪FPCR[]);
Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

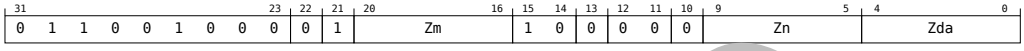
D1.2.6 FDOT (vectors)

Half-precision floating-point dot product

This instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in each 32-bit element of the first source and second source vectors, without intermediate rounding, and then destructively adds the single-precision sum-of-products to the corresponding single-precision element of the destination vector.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
FDOT    <Zda>.S, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = FPDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.7 FDOT (indexed)

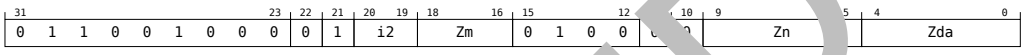
Half-precision floating-point indexed dot product

This instruction computes the fused sum-of-products of a pair of half-precision floating-point values held in each 32-bit element of the first source vector and a pair of half-precision floating-point values in an indexed 32-bit element of the second source vector, without intermediate rounding, and then destructively adds the single-precision sum-of-products to the corresponding single-precision element of the destination vector.

The half-precision floating-point pairs within the second source vector are specified using an immediate index which selects the same pair position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
FDOT    <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

```
if !HaveSME2() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i2);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the immediate index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = FPDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

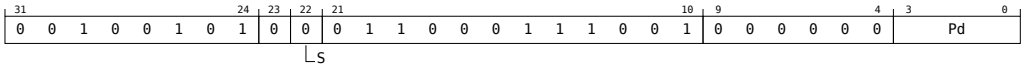
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.8 PFALSE

Set all predicate elements to false

Set all elements in the destination predicate to false.

For programmer convenience, an assembler must also accept predicate-as-counter register name for the destination predicate register.




```
PFALSE  <Pd>.B
```

```
1  if !HaveSVE() && !HaveSME() then UNDEFINED;
2  integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

```
1  CheckSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  P[d, PL] = Zeros(PL);
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then if PSTATE.DIT is 1:

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

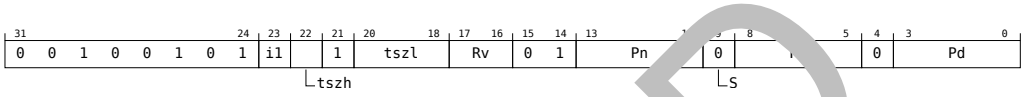
D1.2.9 PSEL

Predicate select between predicate register or all-false

If the indexed element of the second source predicate is true, place the contents of the first source predicate register into the destination predicate register, otherwise set the destination predicate to all-false. The indexed element is determined by the sum of a general-purpose index register and an immediate, modulo the number of elements. Does not set the condition flags.

For programmer convenience, an assembler must also accept predicate-as-counter register names for the destination predicate register and the first source predicate register.

SVE2
(FEAT_SME)




```
PSEL    <Pd>, <Pn>, <Pm>.<T>[<Wv>, <imm>]
```

```
if !HaveSME() then UNDEFINED;
bits(5) imm5 = i1:tszh:tszl;
integer esize;
integer imm;
case tszh:tszl of
    when '0000' UNDEFINED;
    when '1000' esize = 64;  imm = UInt(imm5<4>);
    when 'x100' esize = 32;  imm = UInt(imm5<4:3>);
    when 'xx10' esize = 16;  imm = UInt(imm5<4:2>);
    when 'xxx1' esize = 8;   imm = UInt(imm5<4:1>);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer v = UInt('01':Rv);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in "tszh:tszl":

tszh	tszl	<T>
0	000	RESERVED
x	xx1	B
x	x10	H
x	100	S
1	000	D

<Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.

<imm> Is the element index, in the range 0 to one less than the number of vector elements in a 128-bit vector register, encoded in "i1:tszh:tszl".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
```

```
constant integer elements = VL DIV esize;
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(32) idx = X[v, 32];
integer element = (UInt(idx) + imm) MOD elements;
bits(PL) result;

if ActivePredicateElement(operand2, element, esize) then
    result = operand1;
else
    result = Zeros(PL);

P[d, PL] = result;
```

Operational information

If PSTATE.DIT is 1:

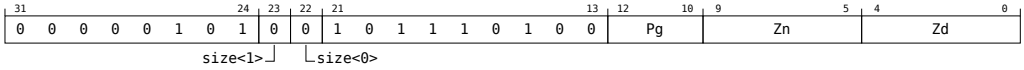
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

D1.2.10 REVD

Reverse 64-bit doublewords in elements (predicated)

Reverse the order of 64-bit doublewords within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

SVE2
(FEAT_SME)




```
REVD    <Zd>.Q, <Pg>/M, <Zn>.Q
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 128;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 64;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ActiveElement(mask, e, esize) then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Reverse(element, swsize);

Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

– The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

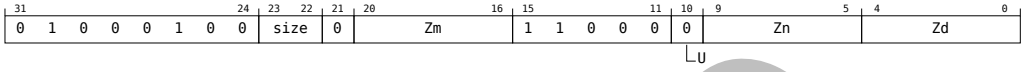
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

D1.2.11 SCLAMP

Signed clamp to minimum/maximum vector

Clamp each signed element in the destination vector to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2
(FEAT_SME)




```
SCLAMP  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
if !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = Min(Max(element1 , element3), element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

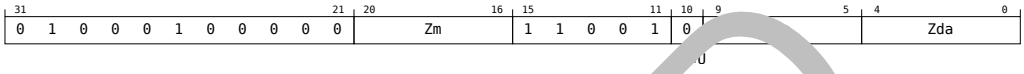
D1.2.12 SDOT (2-way, vectors)

Signed integer dot product

The signed integer dot product instruction computes the dot product of a group of two signed 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two signed 16-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
SDOT    <Zda>.S, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
        integer element2 = SInt(Elem[operand2, 2 * e + i, esize DIV 2]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

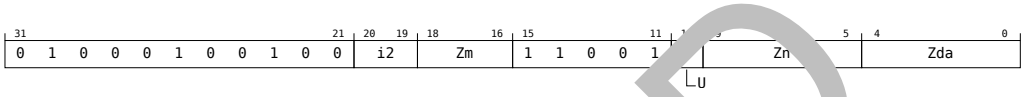
D1.2.13 SDOT (2-way, indexed)

Signed integer indexed dot product

The signed integer indexed dot product instruction computes the dot product of a group of two signed 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two signed 16-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
SDOT    <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the immediate index of a pair of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer element1 = SInt(Elem[operand1, 2 * e + i, esize DIV 2]);
        integer element2 = SInt(Elem[operand2, 2 * s + i, esize DIV 2]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

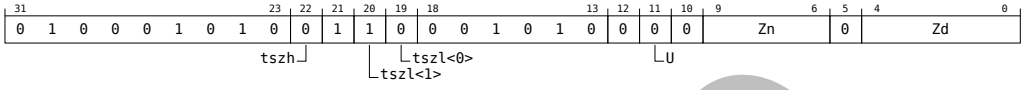
D1.2.14 SQCVTN

Signed saturating extract narrow and interleave

Saturate the signed integer value in each element of the group of two source vectors to half the orginal source element width, and place the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
SQCVTN  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for e = 0 to elements-1
    integer i = e + n;
    bits(VL) operand = Z[n+i, VL];
    integer element = SInt(Elem[operand, e, 2 * esize]);
    Elem[result, 2*e + i, esize] = SignedSat(element, esize);

Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

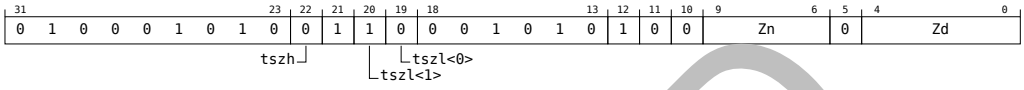
D1.2.15 SQCVTUN

Signed saturating unsigned extract narrow and interleave

Saturate the signed integer value in each element of the group of two source vectors to unsigned integer value that is half the orginal source element width, and place the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
SQCVTUN <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer part = VL DIV 4;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for e = 0 to elements-1
    for i = 0 to 1
        bits(VL) operand = Z[n+i, VL];
        integer element = SInt(Elem[operand, e, 2 * esize]);
        Elem[result, 2*e + i, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

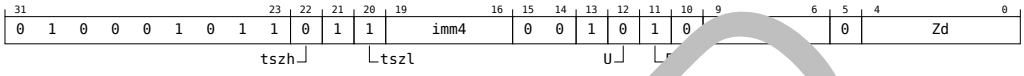
D1.2.16 SQRSHRN

Signed saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
SQRSHRN <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 16;
3  integer n = UInt(Zn:'0');
4  integer d = UInt(Zd);
5  integer shift = esize - UInt(imm4);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
1  CheckSMEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer DL = VL DIV 8;
4  constant integer elements = VL DIV (2 * esize);
5  bits(VL) result;
6  integer round_const = 1 << (shift-1);
7
8  for e = 0 to elements-1
9      for i = 0 to 1
10         bits(VL) operand = Z[n+i, VL];
11         bits(2 * esize) element = Elem[operand, e, 2 * esize];
12         integer res = (SInt(element) + round_const) >> shift;
13         Elem[result, 2*e + i, esize] = SignedSat(res, esize);
14
15 Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

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

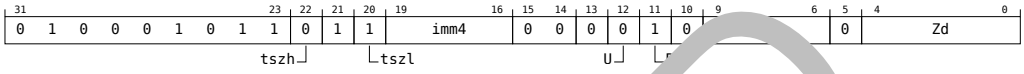
D1.2.17 SQRSHRUN

Signed saturating rounding shift right unsigned narrow by immediate and interleave

Shift right by an immediate value, the signed integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
SQRSHRUN <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 16;
3  integer n = UInt(Zn:'0');
4  integer d = UInt(Zd);
5  integer shift = esize - UInt(imm4);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
1  CheckSMEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer DL = VL DIV 8;
4  constant integer elements = VL DIV (2 * esize);
5  bits(VL) result;
6  integer round_const = 1 << (shift-1);
7
8  for e = 0 to elements-1
9      for i = 0 to 1
10         bits(VL) operand = Z[n+i, VL];
11         bits(2 * esize) element = Elem[operand, e, 2 * esize];
12         integer res = (SInt(element) + round_const) >> shift;
13         Elem[result, 2*e + i, esize] = UnsignedSat(res, esize);
14
15 Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

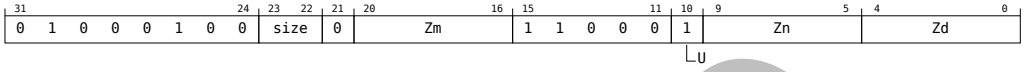
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

D1.2.18 UCLAMP

Unsigned clamp to minimum/maximum vector

Clamp each unsigned element in the destination vector to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2
(FEAT_SME)




```
UCLAMP  <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```
1  if !HaveSME() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  integer n = UInt(Zn);
4  integer m = UInt(Zm);
5  integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
1  CheckSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer elements = VL DIV esize;
4  bits(VL) operand1 = Z[n, VL];
5  bits(VL) operand2 = Z[m, VL];
6  bits(VL) operand3 = Z[d, VL];
7  bits(VL) result;
8
9  for e = 0 to elements-1
10     integer element1 = UInt(Elem[operand1, e, esize]);
11     integer element2 = UInt(Elem[operand2, e, esize]);
12     integer element3 = UInt(Elem[operand3, e, esize]);
13     integer res = Min(Max(element1 , element3), element2);
14     Elem[result, e, esize] = res<esize-1:0>;
15
16 Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

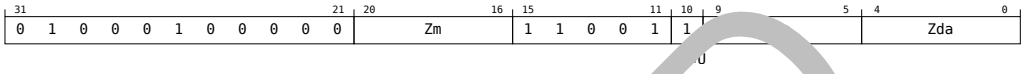
D1.2.19 UDOT (2-way, vectors)

Unsigned integer dot product

The unsigned integer dot product instruction computes the dot product of a group of two unsigned 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two unsigned 16-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
UDOT    <Zda>.S, <Zn>.H, <Zm>.H
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
        integer element2 = UInt(Elem[operand2, 2 * e + i, esize DIV 2]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

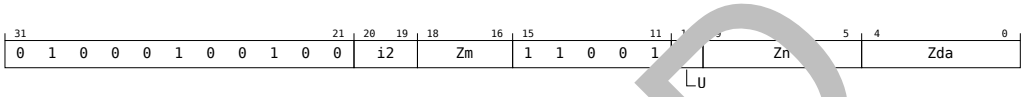
D1.2.20 UDOT (2-way, indexed)

Unsigned integer indexed dot product

The unsigned integer indexed dot product instruction computes the dot product of a group of two unsigned 16-bit integer values held in each 32-bit element of the first source vector multiplied by a group of two unsigned 16-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
UDOT    <Zda>.S, <Zn>.H, <Zm>.H[<imm>]
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

<imm> Is the immediate index of a pair of two 16-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer element1 = UInt(Elem[operand1, 2 * e + i, esize DIV 2]);
        integer element2 = UInt(Elem[operand2, 2 * s + i, esize DIV 2]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

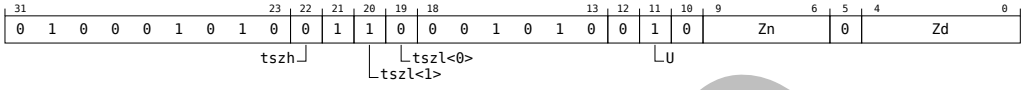
D1.2.21 UQCVTN

Unsigned saturating extract narrow and interleave

Saturate the unsigned integer value in each element of the group of two source vectors to half the orginal source element width, and place the two-way interleaved results in the half-width destination elements.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
UQCVTN  <Zd>.H, { <Zn1>.S-<Zn2>.S }
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Zn:'0');
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) result;

for e = 0 to elements-1
    integer i = e + n;
    bits(VL) operand = Z[n+i, VL];
    integer element = UInt(Elem[operand, e, 2 * esize]);
    Elem[result, 2*e + i, esize] = UnsignedSat(element, esize);

Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

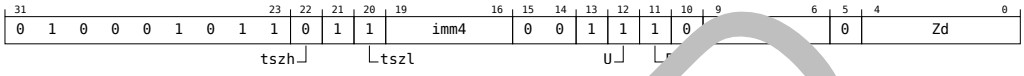
D1.2.22 UQRSHRN

Unsigned saturating rounding shift right narrow by immediate and interleave

Shift right by an immediate value, the unsigned integer value in each element of the group of two source vectors and place the two-way interleaved rounded results in the half-width destination elements. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to 16.

This instruction is unpredicated.

SVE2
(FEAT_SME2)




```
UQRSHRN <Zd>.H, { <Zn1>.S-<Zn2>.S }, #<const>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 16;
3  integer n = UInt(Zn:'0');
4  integer d = UInt(Zd);
5  integer shift = esize - UInt(imm4);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded as "Zn" times 2.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded as "Zn" times 2 plus 1.

<const> Is the immediate shift amount, in the range 1 to 16, encoded in the "imm4" field.

Operation

```
1  CheckSMEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer elements = VL DIV (2 * esize);
5  bits(VL) result;
6  integer round_const = 1 << (shift-1);
7
8  for e = 0 to elements-1
9      for i = 0 to 1
10         bits(VL) operand = Z[n+i, VL];
11         bits(2 * esize) element = Elem[operand, e, 2 * esize];
12         integer res = (UInt(element) + round_const) >> shift;
13         Elem[result, 2*e + i, esize] = UnsignedSat(res, esize);
14
15 Z[d, VL] = result;
```

Operational information

If PSTATE.DIT is 1:

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

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

D1.2.23 WHILEGE (predicate pair)

While decrementing signed scalar greater than or equal to scalar (pair of predicates)

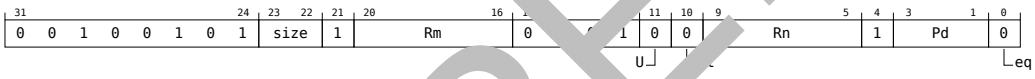
Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element of the pair.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT_SME2)




```
WHILEGE { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = esize;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d0 = UInt(Pd:'0');
7  integer d1 = UInt(Pd:'1');
8  boolean unsigned = FALSE;
9  SVECmp op = CmpGE;
```

Assembler symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

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

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

Operation

```
1  CheckSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer elements = VL DIV esize;
```

```
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = (elements*2)-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.2.24 WHILEGT (predicate pair)

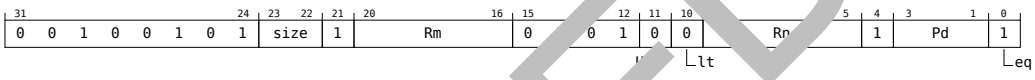
While decrementing signed scalar greater than scalar (pair of predicates)

Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT_SME2)




```
WHILEGT { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = FALSE;
SVECmp op = Cmp_GT;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

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

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

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
```

```
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = (elements*2)-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.2.25 WHILEHI (predicate pair)

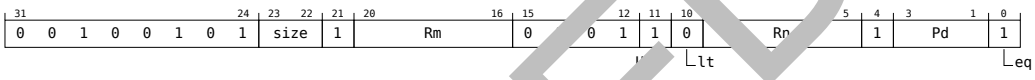
While decrementing unsigned scalar higher than scalar (pair of predicates)

Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT_SME2)




```
WHILEHI { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = TRUE;
SVECmp op = Cmp_GT;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

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

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

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
```

```
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = (elements*2)-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.2.26 WHILEHS (predicate pair)

While decrementing unsigned scalar higher or same as scalar (pair of predicates)

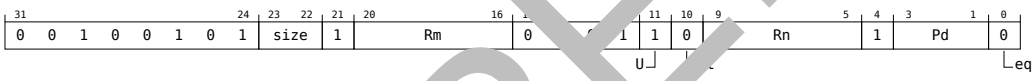
Generate a pair of predicates that starting from the highest numbered element of the pair is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element of the pair.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT_SME2)




```
WHILEHS { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = esize;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d0 = UInt(Pd:'0');
7  integer d1 = UInt(Pd:'1');
8  boolean unsigned = TRUE;
9  SVECmp op = Cmp_GE;
```

Assembler symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

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

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

Operation

```
1  CheckSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer elements = VL DIV esize;
```

```
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = (elements*2)-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.2.27 WHILELE (predicate pair)

While incrementing signed scalar less than or equal to scalar (pair of predicates)

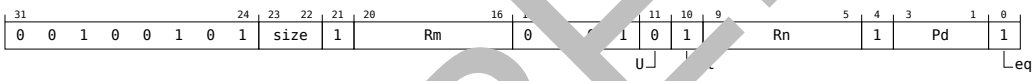
Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element of the pair.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT_SME2)




```
WHILELE { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = esize;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d0 = UInt(Pd:'0');
7  integer d1 = UInt(Pd:'1');
8  boolean unsigned = FALSE;
9  SVECmp op = CmpLE;
```

Assembler symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the element size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

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

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

Operation

```
1  CheckSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer elements = VL DIV esize;
```

```
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to (elements*2)-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.2.28 WHILELO (predicate pair)

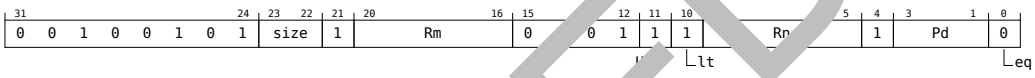
While incrementing unsigned scalar lower than scalar (pair of predicates)

Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT_SME2)




```
WHILELO { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = TRUE;
SVECmp op = Cmp_LT;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

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

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

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
```

```
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to (elements*2)-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.2.29 WHILELS (predicate pair)

While incrementing unsigned scalar lower or same as scalar (pair of predicates)

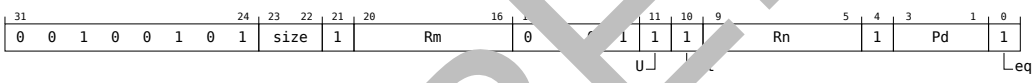
Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element of the pair.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT_SME2)




```
WHILELS { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
1  if !HaveSME2() then UNDEFINED;
2  constant integer esize = 8 << UInt(size);
3  constant integer rsize = esize;
4  integer n = UInt(Rn);
5  integer m = UInt(Rm);
6  integer d0 = UInt(Pd:'0');
7  integer d1 = UInt(Pd:'1');
8  boolean unsigned = TRUE;
9  SVECmp op = Cmp_LE;
```

Assembler symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the element size specifier, encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

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

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

Operation

```
1  CheckSVEEnabled();
2  constant integer VL = CurrentVL;
3  constant integer PL = VL DIV 8;
4  constant integer elements = VL DIV esize;
```

```
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to (elements*2)-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.2.30 WHILELT (predicate pair)

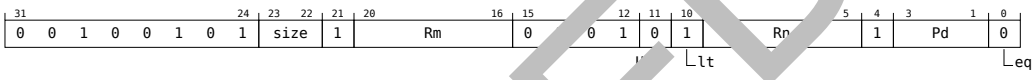
While incrementing signed scalar less than scalar (pair of predicates)

Generate a pair of predicates that starting from the lowest numbered element of the pair is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element of the pair.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The lower-numbered elements are placed in the first predicate destination register, and the higher-numbered elements in the second predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

SVE2
(FEAT_SME2)




```
WHILELT { <Pd1>.<T>, <Pd2>.<T> }, <Xn>, <Xm>
```

```
if !HaveSME2() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d0 = UInt(Pd:'0');
integer d1 = UInt(Pd:'1');
boolean unsigned = FALSE;
SVECmp op = Cmp_LT;
```

Assembler Symbols

<Pd1> Is the name of the first destination scalable predicate register, encoded as "Pd" times 2.

<T> Is the size specifier encoded in "size":

size	<T>
00	B
01	H
10	S
11	D

<Pd2> Is the name of the second destination scalable predicate register, encoded as "Pd" times 2 plus 1.

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

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

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL*2) mask = Ones(PL*2);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
```

```
bits(PL*2) result;
boolean last = TRUE;
constant integer psize = esize DIV 8;

for e = 0 to (elements*2)-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    bit pbit = if last then '1' else '0';
    Elem[result, e, psize] = ZeroExtend(pbit, psize);
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d0, PL] = result<PL-1:0>;
P[d1, PL] = result<PL*2-1:PL>;
```

Operational information

If PSTATE.DIT is 1:

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

D1.3 Base A64 instructions

The following Base A64 instructions are added or modified by the SME or SME2 architecture.

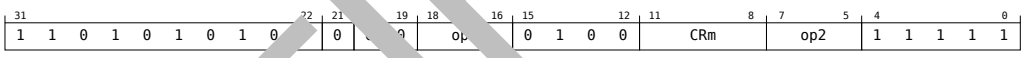
D1.3.1 MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see *Process state, PSTATE*.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If *FEAT_SSBS* is implemented, PSTATE.SSBS.
- If *FEAT_PAN* is implemented, PSTATE.PAN.
- If *FEAT_UAO* is implemented, PSTATE.UAO.
- If *FEAT_DIT* is implemented, PSTATE.DIT.
- If *FEAT_MTE* is implemented, PSTATE.TCO.
- If *FEAT_NMI* is implemented, PSTATE.ALL
- If *FEAT_SME* is implemented, PSTATE.SM and PSTATE.ZA

This instruction is used by the aliases SMSTART, and SMST




```
MSR  <pstatefield>, #<imm>
```

```
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

AArch64.SystemAccess('00', op1, '0100', CRm, op2, '11111', '0');
bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
    when '00x'
        min_EL = EL1;
    when '01x'
        min_EL = EL1;
    when '011'
        min_EL = EL0;
    when '100'
        min_EL = EL2;
    when '101'
        if !HaveVirtHostExt() then
            UNDEFINED;
        min_EL = EL2;
    when '110'
        min_EL = EL3;
    when '111'
        min_EL = EL1;
        need_secure = TRUE;

if (UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && CurrentSecurityState() != SS_Secure))
    ↪then
    UNDEFINED;
```

```
30
31  PSTATEField field;
32  case op1:op2 of
33      when '000 011'
34          if !HaveUAOExt() then UNDEFINED;
35          field = PSTATEField_UAO;
36      when '000 100'
37          if !HavePANExt() then UNDEFINED;
38          field = PSTATEField_PAN;
39      when '000 101' field = PSTATEField_SP;
40      when '001 000'
41          if CRm == '000x' then
42              if !HaveFeatNMI() then UNDEFINED;
43              field = PSTATEField_ALLINT;
44          else
45              UNDEFINED;
46      when '011 010'
47          if !HaveDITExt() then UNDEFINED;
48          field = PSTATEField_DIT;
49      when '011 011'
50          case CRm of
51              when '001x'
52                  if !HaveSME() then UNDEFINED;
53                  field = PSTATEField_SVCRSM;
54              when '010x'
55                  if !HaveSME() then UNDEFINED;
56                  field = PSTATEField_SVCRZA
57              when '011x'
58                  if !HaveSME() then UNDEFINED;
59                  field = PSTATEField_SVCRSMZA;
60              otherwise
61                  UNDEFINED;
62      when '011 100'
63          if !HaveMTEExt() then UNDEFINED;
64          field = PSTATEField_TCO;
65      when '011 110' field = PSTATEField_DAIFSet;
66      when '011 111' field = PSTATEField_DAIFClr;
67      when '011 001'
68          if !HaveSSBSExt() then UNDEFINED;
69          field = PSTATEField_SSBS;
70      otherwise UNDEFINED;
71
72  // Check that an AArch64 MSR to a PSTATE access to the DAIF flags is permitted
73  if PSTATEField IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
74      if !ELUsingAArch32(EL2) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') ||
        ↪SCTLR_EL1.UMA == '0')) then
75          if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
76              AArch64.SystemAccessTrap(EL2, 0x18);
77          else
78              AArch64.SystemAccessTrap(EL1, 0x18);
```


Assembler Symbols

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in "op1:op2:CRm":

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	SEE PSTATE	–
000	010	xxxx	SEE PSTATE	–
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	–
000	11x	xxxx	RESERVED	–
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	–
001	000	01xx	RESERVED	–
001	000	1xxx	RESERVED	–
001	001	xxxx	RESERVED	–
001	01x	xxxx	RESERVED	–
001	1xx	xxxx	RESERVED	–
010	xxx	xxxx	RESERVED	–
011	000	xxxx	RESERVED	–
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	–
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	–
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	–
011	110	xxxx	DAIFSet	–
011	111	xxxx	DAIFClr	–
1xx	xxx	xxxx	RESERVED	–

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field. Restricted to the range 0 to 1, encoded in "CRm<0>", when <pstatefield> is ALLINT, SVCRSM, SVCRSMZA, or SVCRZA.

Alias Conditions

Alias	Is preferred when
SMSTART	op1 == '011' && CRm == '0xx1' && op2 == '011'
SMSTOP	op1 == '011' && CRm == '0xx0' && op2 == '011'

Operation

```
case field of
    when PSTATEField_SSBS
        PSTATE.SSBS = CRm<0>;
    when PSTATEField_SP
        PSTATE.SP = CRm<0>;
    when PSTATEField_DAIFSet
        PSTATE.D = PSTATE.D OR CRm<3>;
        PSTATE.A = PSTATE.A OR CRm<2>;
        PSTATE.I = PSTATE.I OR CRm<1>;
        PSTATE.F = PSTATE.F OR CRm<0>;
    when PSTATEField_DAIFClr
        PSTATE.D = PSTATE.D AND NOT(CRm<3>);
        PSTATE.A = PSTATE.A AND NOT(CRm<2>);
        PSTATE.I = PSTATE.I AND NOT(CRm<1>);
```

```
        PSTATE.F = PSTATE.F AND NOT(CRm<0>);
    when PSTATEField_PAN
        PSTATE.PAN = CRm<0>;
    when PSTATEField_UAO
        PSTATE.UAO = CRm<0>;
    when PSTATEField_DIT
        PSTATE.DIT = CRm<0>;
    when PSTATEField_TCO
        PSTATE.TCO = CRm<0>;
    when PSTATEField_ALLINT
        if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' && CRm<0> ==
            ↪'1') then
            AArch64.SystemAccessTrap(EL2, 0x18);
        PSTATE.ALLINT = CRm<0>;
    when PSTATEField_SVCRSM
        CheckSMEAccess();
        SetPSTATE_SM(CRm<0>);
    when PSTATEField_SVCRZA
        CheckSMEAccess();
        SetPSTATE_ZA(CRm<0>);
    when PSTATEField_SVCRSMZA
        CheckSMEAccess();
        SetPSTATE_SM(CRm<0>);
        SetPSTATE_ZA(CRm<0>);
```

D1.3.2 RPRFM

Range Prefetch Memory signals the memory system that data memory accesses from a specified range of addresses are likely to occur in the near future. The instruction may also signal the memory system about the likelihood of data reuse of the specified range of addresses. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as prefetching locations within the specified address ranges into one or more caches. The memory system may also exploit the data reuse hints to decide whether to retain the data in other caches upon eviction from the innermost caches or to discard it.

The effect of an RPRFM instruction is IMPLEMENTATION DEFINED, but because these signals are only hints, the instruction cannot cause a synchronous Data Abort exception and is guaranteed not to access Device memory. It is valid for the PE to treat this instruction as a NOP.

An RPRFM instruction specifies the type of accesses and range of addresses using the following parameters:

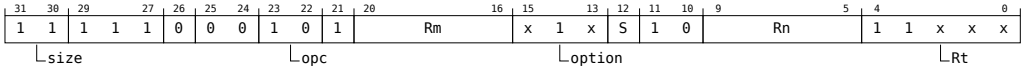
- 'Type', in the <rprfop> operand opcode bits, specifies whether the prefetched data will be accessed by load or store instructions.
- 'Policy', in the <rprfop> operand opcode bits, specifies whether the data is likely to be reused or if it is a streaming, non-temporal prefetch. If a streaming prefetch is specified, then the 'ReuseDistance' parameter is ignored.
- 'BaseAddress', in the 64-bit base register, holds the initial block address for the accesses.
- 'ReuseDistance', in the metadata register bits[63:60], indicates the maximum number of bytes to be accessed by this PE before executing the next RPRFM instruction that specifies the same range. This includes the total number of bytes inside and outside of the range that will be accessed by the same PE. This parameter can be used to influence cache eviction and replacement policies, in order to retain the data in the most optimal levels of the memory hierarchy after each access. If software cannot easily determine the amount of other memory that will be accessed, these bits can be set to zero to indicate that 'ReuseDistance' is not known. Otherwise, these four bits encode decreasing powers of two in the range 512MiB (0b0001) to 32KiB (0b1111).
- 'Stride', in the metadata register bits[59:38], is a signed, two's complement integer encoding of the number of bytes to advance the block address after 'Length' bytes have been accessed, in the range -2MiB to +2MiB-1B. A negative value indicates that the block address is advanced in a descending direction.
- 'Count', in the metadata register bits[37:22], is an unsigned integer encoding of the number of blocks of data to be accessed, minus 1, representing the range 1 to 65536 blocks. If 'Count' is 0, then the 'Stride' parameter is ignored and only a single block of contiguous bytes from 'BaseAddress' to ('BaseAddress' + 'Length' - 1) is described.
- 'Length', in the metadata register bits[21:0], is a signed, two's complement integer encoding of the number of contiguous bytes to be accessed starting from the current block address, without changing the block address, in the range -2MiB to +2MiB-1B. A negative value indicates that the bytes are accessed in a descending direction.

Note

Software is expected to honor the parameters it provides to the RPRFM instruction, and the same PE should access all locations in the range, in the direction specified by the sign of the 'Length' and 'Stride' parameters. A range prefetch is considered active on a PE until all locations in the range have been accessed by the PE. A range prefetch might also be inactivated by the PE prior to completion, for example due to a software context switch or lack of hardware resources.

Software should not specify overlapping addresses in multiple active ranges. If a range is expected to be accessed by both load and store instructions (read-modify-write), then a single range with a 'Type' parameter of PST (prefetch for store) should be specified.

Integer
(FEAT_RPRFM)




```
RPRFM  (<rprfop>|#<imm6>), <Xm>, [<Xn|SP>]
```

```
bits(6) operation = option<2>:option<0>:S:Rt<2:0>;
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<rprfop> Is the range prefetch operation, defined as <type><policy>. <type> is one of:

PLD

Prefetch for load, encoded in the "Rt<0>" field as 0.

PST

Prefetch for store, encoded in the "Rt<0>" field as 1.

<policy> is one of:

KEEP

Retained or temporal prefetch for data that is expected to be kept in caches to be accessed more than once, encoded in the "option<2>:option<0>:S:Rt<2:1>" fields as 0b00000.

STRM

Streaming or non-temporal prefetch, for data that is expected to be accessed once and not reused, encoded in the "option<2>:option<0>:S:Rt<2:1>" fields as 0b00010.

For other encodings of the "option<2>:option<0>:S:Rt<2:0>" fields, use <imm6>.

<imm6> Is the range prefetch operation encoding as an immediate, in the range 0 to 63, encoded in "option<2>:option<0>:S:Rt<2:0>". This syntax is only for encodings that are not representable using <rprfop>.

<Xm> Is the 64-bit name of the general-purpose register that holds an encoding of the metadata, encoded in the "Rm" field.

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

Operation

```
bits(64) address  = if n == 31 then SP[] else X[n, 64];
bits(64) metadata = X[m, 64];
integer stride = SInt(metadata<59:38>);
integer count  = UInt(metadata<37:22>) + 1;
integer length = SInt(metadata<21:0>);
integer reuse;

if metadata<63:60> == '0000' then
    reuse = -1; // Not known
else
    reuse = 32768 << (15 - UInt(metadata<63:60>));
```

```
12
13  Hint_RangePrefetch(address, length, stride, count, reuse, operation);
```

D1.3.3 SMSTART

Enables access to Streaming SVE mode and SME architectural state.

SMSTART enters Streaming SVE mode, and enables the SME ZA storage.

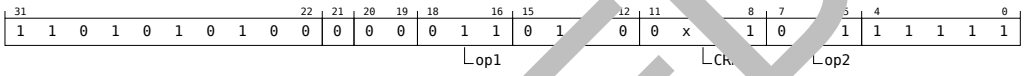
SMSTART SM enters Streaming SVE mode, but does not enable the SME ZA storage.

SMSTART ZA enables the SME ZA storage, but does not cause an entry to Streaming SVE mode.

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

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

System
(FEAT_SME)




```
SMSTART  {<option>}
```

is equivalent to

```
MSR  <pstatefield>, #1
```

and is always the preferred disassembly.

Assembler Symbols

<option> Is an optional mode, encoded in "CRm<2:1>":

CRm<2:1>	<option>
00	RESERVED
01	SM
10	ZA
11	[no specifier]

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in "op1:op2:CRm":

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	SEE PSTATE	–
000	010	xxxx	SEE PSTATE	–
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	–
000	11x	xxxx	RESERVED	–
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	–
001	000	01xx	RESERVED	–
001	000	1xxx	RESERVED	–
001	001	xxxx	RESERVED	–
001	01x	xxxx	RESERVED	–
001	1xx	xxxx	RESERVED	–
010	xxx	xxxx	RESERVED	–
011	000	xxxx	RESERVED	–
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	–
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	–
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	–
011	110	xxxx	DAIFSet	–
011	111	xxxx	DAIFClr	–
1xx	xxx	xxxx	RESERVED	–

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

Operation

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

D1.3.4 SMSTOP

Disables access to Streaming SVE mode and SME architectural state.

SMSTOP exits Streaming SVE mode, and disables the SME ZA storage.

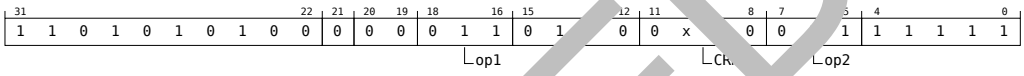
SMSTOP SM exits Streaming SVE mode, but does not disable the SME ZA storage.

SMSTOP ZA disables the SME ZA storage, but does not cause an exit from Streaming SVE mode.

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

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

System
(FEAT_SME)




```
SMSTOP  {<option>}
```

is equivalent to

```
MSR  <pstatefield>, #0
```

and is always the preferred disassembly.

Assembler Symbols

<option> Is an optional mode, encoded in "CRm<2:1>":

CRm<2:1>	<option>
00	RESERVED
01	SM
10	ZA
11	[no specifier]

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in "op1:op2:CRm":

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	SEE PSTATE	–
000	010	xxxx	SEE PSTATE	–
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	–
000	11x	xxxx	RESERVED	–
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	–
001	000	01xx	RESERVED	–
001	000	1xxx	RESERVED	–
001	001	xxxx	RESERVED	–
001	01x	xxxx	RESERVED	–
001	1xx	xxxx	RESERVED	–
010	xxx	xxxx	RESERVED	–
011	000	xxxx	RESERVED	–
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	–
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	–
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	–
011	110	xxxx	DAIFSet	–
011	111	xxxx	DAIFClr	–
1xx	xxx	xxxx	RESERVED	–

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

Operation

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

Part E
Appendices

Chapter E1
Instructions affected by SME

The behavior of some non-SME instructions is affected when SME is implemented and the PE is in *Streaming SVE mode*.

This section lists affected instructions by the type of effect, with a description of the changes. It is a reference summary of information that can be viewed in more detail in *Arm® A64 Instruction Set Architecture, for A-profile architecture* [3].

E1.1 Illegal instructions in Streaming SVE mode

E1.1.1 Illegal Advanced SIMD instructions

The instruction encoding tables in this section are provided as an aid to understanding, and are consistent with the A64 ISA in Armv8.8-A and Armv9.3-A, but will require correction if subsequent versions of the A64 ISA add new instructions which overlap with these encodings.

AArch64 Advanced SIMD instructions with encodings that match the following patterns are *illegal* when the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:

A64 Encoding Pattern	Encoding Block
`0x00 110x xxxx xxxx xxxx xxxx xxxx xxxx`	Advanced SIMD structure load/store
`0xx0 111x xxxx xxxx xxxx xxxx xxxx xxxx`	Advanced SIMD vector operations
`01x1 111x xxxx xxxx xxxx xxxx xxxx xxxx`	Advanced SIMD single-element operations
`1100 1110 xxxx xxxx xxxx xxxx xxxx xxxx`	Advanced SIMD cryptography extensions

With the exception of certain vector to GPR integer move instructions, and some single-element floating-point instructions that match the following patterns, and which execute normally when the PE is in *Streaming SVE mode*:

A64 Encoding Pattern	Instructions or Instruction Class
`0x00 1110 0000 0001 0010 11xx xxxx xxxx`	`SMOV W|Xd,Vn.B[0]`
`0x00 1110 0000 0010 0010 11xx xxxx xxxx`	`SMOV W|Xd,Vn.H[0]`
`0100 1110 0000 0100 0010 11xx xxxx xxxx`	`SMOV Xd,Vn.S[0]`
`0000 1110 0000 0001 0011 11xx xxxx xxxx`	`UMOV Wd,Vn.B[0]`
`0000 1110 0000 0010 0011 11xx xxxx xxxx`	`UMOV Wd,Vn.H[0]`
`0000 1110 0000 0100 0011 11xx xxxx xxxx`	`UMOV Wd,Vn.S[0]`
`0100 1110 0000 1000 0011 11xx xxxx xxxx`	`UMOV Xd,Vn.D[0]`
`0101 1110 xx1x xxxx 11x1 11xx xxxx xxxx`	`FMULX/FRECPS/FRSQRTS (scalar)`
`0101 1110 x10x xxxx 00x1 11xx xxxx xxxx`	`FMULX/FRECPS/FRSQRTS (scalar, FP16)`
`01x1 1110 1x10 0001 11x1 10xx xxxx xxxx`	`FRECPE/FRSQRTE/FRECPX (scalar)`
`01x1 1110 1111 1001 11x1 10xx xxxx xxxx`	`FRECPE/FRSQRTE/FRECPX (scalar, FP16)`

For the avoidance of doubt, A64 scalar floating-point instructions which match following encoding patterns remain *legal* when the PE is in *Streaming SVE mode*:

A64 Encoding Pattern	Instructions or Instruction Class
`x001 111x xxxx xxxx xxxx xxxx xxxx xxxx`	Scalar floating-point operations
`xx10 110x xxxx xxxx xxxx xxxx xxxx xxxx`	Load/store pair of FP registers
`xx01 1100 xxxx xxxx xxxx xxxx xxxx xxxx`	Load FP register (PC-relative literal)

A64 Encoding Pattern	Instructions or Instruction Class
`xx11 1100 xx0x xxxx xxxx xxxx xxxx xxxx`	Load/store FP register (unscaled imm)
`xx11 1100 xx1x xxxx xxxx xxxx xxxx xx10`	Load/store FP register (register offset)
`xx11 1101 xxxx xxxx xxxx xxxx xxxx xxxx`	Load/store FP register (scaled imm)

With the exception of the following floating-point operation which is *illegal* when the PE is in *Streaming SVE mode*:

A64 Encoding Pattern	Instructions or Instruction Class
`0001 1110 0111 1110 0000 00xx xxxx xxxx`	`FJCVTZS`

E1.1.1.1 Vector instructions

This section lists by name those A64 Advanced SIMD instruction pages in which all encoding variants are *illegal* when the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level.

The Advanced SIMD instructions described on the following pages, and their aliases, are affected in this way:

- ABS: Absolute value (vector).
- ADD (vector): Add (vector).
- ADDHN, ADDHN2: Add returning High Narrow.
- ADDP (scalar): Add Pair of elements (scalar).
- ADDP (vector): Add Pairwise (vector).
- ADDV: Add across Vector.
- AESD: AES single round decryption.
- AESE: AES single round encryption.
- AESIMC: AES inverse mix columns.
- AESMC: AES mix columns.
- AND (vector): Bitwise AND (vector).
- BCAX: Bit Clear and XOR.
- BFCVTN, BFCVTN2: Floating-point convert from single-precision to BFloat16 format (vector).
- BFDOT (by element): BFloat16 floating-point dot product (vector, by element).
- BFDOT (vector): BFloat16 floating-point dot product (vector).
- BFMLALB, BFMLALT (by element): BFloat16 floating-point widening multiply-add long (by element).
- BFMLALB, BFMLALT (vector): BFloat16 floating-point widening multiply-add long (vector).
- BFMMLA: BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.
- BIC (vector, immediate): Bitwise bit Clear (vector, immediate).
- BIC (vector, register): Bitwise bit Clear (vector, register).
- BIF: Bitwise Insert if False.
- BIT: Bitwise Insert if True.
- BSL: Bitwise Select.
- CLS (vector): Count Leading Sign bits (vector).
- CLZ (vector): Count Leading Zero bits (vector).
- CMEQ (register): Compare bitwise Equal (vector).
- CMEQ (zero): Compare bitwise Equal to zero (vector).
- CMGE (register): Compare signed Greater than or Equal (vector).
- CMGE (zero): Compare signed Greater than or Equal to zero (vector).
- CMGT (register): Compare signed Greater than (vector).
- CMGT (zero): Compare signed Greater than zero (vector).

- CMHI (register): Compare unsigned Higher (vector).
- CMHS (register): Compare unsigned Higher or Same (vector).
- CMLE (zero): Compare signed Less than or Equal to zero (vector).
- CMLT (zero): Compare signed Less than zero (vector).
- CMTST: Compare bitwise Test bits nonzero (vector).
- CNT: Population Count per byte.
- DUP (element): Duplicate vector element to vector or scalar.
- DUP (general): Duplicate general-purpose register to vector.
- EOR (vector): Bitwise Exclusive OR (vector).
- EOR3: Three-way Exclusive OR.
- EXT: Extract vector from pair of vectors.
- FABD: Floating-point Absolute Difference.
- FABS (vector): Floating-point Absolute value (vector).
- FACGE: Floating-point Absolute Compare Greater than or Equal (vector).
- FACGT: Floating-point Absolute Compare Greater than (vector).
- FADD (vector): Floating-point Add (vector).
- FADDP (scalar): Floating-point Add Pair of elements (scalar).
- FADDP (vector): Floating-point Add Pairwise (vector).
- FCADD: Floating-point Complex Add.
- FCMEQ (register): Floating-point Compare Equal (vector).
- FCMEQ (zero): Floating-point Compare Equal to zero (vector).
- FCMGE (register): Floating-point Compare Greater than or Equal (vector).
- FCMGE (zero): Floating-point Compare Greater than or Equal to zero (vector).
- FCMGT (register): Floating-point Compare Greater than (vector).
- FCMGT (zero): Floating-point Compare Greater than zero (vector).
- FCMLA: Floating-point Complex Multiply Accumulate.
- FCMLA (by element): Floating-point Complex Multiply Accumulate (by element).
- FCMLE (zero): Floating-point Compare Less than or Equal to zero (vector).
- FCMLT (zero): Floating-point Compare Less than zero (vector).
- FCVTAS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).
- FCVTAU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).
- FCVTL, FCVTL2: Floating-point Convert to higher precision Long (vector).
- FCVTMS (vector): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).
- FCVTMU (vector): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).
- FCVTN, FCVTN2: Floating-point Convert to lower precision Narrow (vector).
- FCVTNS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).
- FCVTNU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).
- FCVTPS (vector): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).
- FCVTPU (vector): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).
- FCVTXN, FCVTXN2: Floating-point Convert to lower precision Narrow, rounding to odd (vector).
- FCVTZS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
- FCVTZS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
- FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
- FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).
- FDIV (vector): Floating-point Divide (vector).
- FJCVTZS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.
- FMAX (vector): Floating-point Maximum (vector).
- FMAXNM (vector): Floating-point Maximum Number (vector).
- FMAXNMP (scalar): Floating-point Maximum Number of Pair of elements (scalar).
- FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).
- FMAXNMV: Floating-point Maximum Number across Vector.
- FMAXP (scalar): Floating-point Maximum of Pair of elements (scalar).
- FMAXP (vector): Floating-point Maximum Pairwise (vector).
- FMAXV: Floating-point Maximum across Vector.

- FMIN (vector): Floating-point minimum (vector).
- FMINNM (vector): Floating-point Minimum Number (vector).
- FMINNMP (scalar): Floating-point Minimum Number of Pair of elements (scalar).
- FMINNMP (vector): Floating-point Minimum Number Pairwise (vector).
- FMINNMV: Floating-point Minimum Number across Vector.
- FMINP (scalar): Floating-point Minimum of Pair of elements (scalar).
- FMINP (vector): Floating-point Minimum Pairwise (vector).
- FMINV: Floating-point Minimum across Vector.
- FMLA (by element): Floating-point fused Multiply-Add to accumulator (by element).
- FMLA (vector): Floating-point fused Multiply-Add to accumulator (vector).
- FMLAL, FMLAL2 (by element): Floating-point fused Multiply-Add Long to accumulator (by element).
- FMLAL, FMLAL2 (vector): Floating-point fused Multiply-Add Long to accumulator (vector).
- FMLS (by element): Floating-point fused Multiply-Subtract from accumulator (by element).
- FMLS (vector): Floating-point fused Multiply-Subtract from accumulator (vector).
- FMLSL, FMLSL2 (by element): Floating-point fused Multiply-Subtract Long from accumulator (by element).
- FMLSL, FMLSL2 (vector): Floating-point fused Multiply-Subtract Long from accumulator (vector).
- FMOV (vector, immediate): Floating-point move immediate (vector).
- FMUL (by element): Floating-point Multiply (by element).
- FMUL (vector): Floating-point Multiply (vector).
- FMULX (by element): Floating-point Multiply extended (by element).
- FNEG (vector): Floating-point Negate (vector).
- FRINT32X (vector): Floating-point Round to 32-bit Integer, using current rounding mode (vector).
- FRINT32Z (vector): Floating-point Round to 32-bit Integer toward Zero (vector).
- FRINT64X (vector): Floating-point Round to 64-bit Integer, using current rounding mode (vector).
- FRINT64Z (vector): Floating-point Round to 64-bit Integer toward Zero (vector).
- FRINTA (vector): Floating-point Round to Integral, to nearest with ties to Away (vector).
- FRINTI (vector): Floating-point Round to Integral, using current rounding mode (vector).
- FRINTM (vector): Floating-point Round to Integral, toward Minus infinity (vector).
- FRINTN (vector): Floating-point Round to Integral, to nearest with ties to even (vector).
- FRINTP (vector): Floating-point Round to Integral, toward Plus infinity (vector).
- FRINTX (vector): Floating-point Round to Integral exact, using current rounding mode (vector).
- FRINTZ (vector): Floating-point Round to Integral, toward Zero (vector).
- FSQRT (vector): Floating-point Square Root (vector).
- FSUB (vector): Floating-point Subtract (vector).
- INS (element): Insert vector element from another vector element.
- INS (general): Insert vector element from general-purpose register.
- LD1 (multiple structures): Load multiple single-element structures to one, two, three, or four registers.
- LD1 (single structure): Load one single-element structure to one lane of one register.
- LD1R: Load one single-element structure and Replicate to all lanes (of one register).
- LD2 (multiple structures): Load multiple 2-element structures to two registers.
- LD2 (single structure): Load single 2-element structure to one lane of two registers.
- LD2R: Load single 2-element structure and Replicate to all lanes of two registers.
- LD3 (multiple structures): Load multiple 3-element structures to three registers.
- LD3 (single structure): Load single 3-element structure to one lane of three registers).
- LD3R: Load single 3-element structure and Replicate to all lanes of three registers.
- LD4 (multiple structures): Load multiple 4-element structures to four registers.
- LD4 (single structure): Load single 4-element structure to one lane of four registers.
- LD4R: Load single 4-element structure and Replicate to all lanes of four registers.
- MLA (by element): Multiply-Add to accumulator (vector, by element).
- MLA (vector): Multiply-Add to accumulator (vector).
- MLS (by element): Multiply-Subtract from accumulator (vector, by element).
- MLS (vector): Multiply-Subtract from accumulator (vector).
- MOVI: Move Immediate (vector).
- MUL (by element): Multiply (vector, by element).
- MUL (vector): Multiply (vector).

- MVNI: Move inverted Immediate (vector).
- NEG (vector): Negate (vector).
- NOT: Bitwise NOT (vector).
- ORN (vector): Bitwise inclusive OR NOT (vector).
- ORR (vector, immediate): Bitwise inclusive OR (vector, immediate).
- ORR (vector, register): Bitwise inclusive OR (vector, register).
- PMUL: Polynomial Multiply.
- PMULL, PMULL2: Polynomial Multiply Long.
- RADDHN, RADDHN2: Rounding Add returning High Narrow.
- RAX1: Rotate and Exclusive OR.
- RBIT (vector): Reverse Bit order (vector).
- REV16 (vector): Reverse elements in 16-bit halfwords (vector).
- REV32 (vector): Reverse elements in 32-bit words (vector).
- REV64: Reverse elements in 64-bit doublewords (vector).
- RSHRN, RSHRN2: Rounding Shift Right Narrow (immediate).
- RSUBHN, RSUBHN2: Rounding Subtract returning High Nar
- SABA: Signed Absolute difference and Accumulate.
- SABAL, SABAL2: Signed Absolute difference and Ac mulate Long
- SABD: Signed Absolute Difference.
- SABDL, SABDL2: Signed Absolute Difference Lo
- SADALP: Signed Add and Accumulate Long Pairwise.
- SADDL, SADDL2: Signed Add Long (vec).
- SADDLP: Signed Add Long Pairwise.
- SADDLV: Signed Add Long across Ve or.
- SADDW, SADDW2: Signed Add Wide.
- SCVTF (vector, fixed-point): Signe fixed-p t Convert to Floating-point (vector).
- SCVTF (vector, integer): Si ed integ Conve o Floating-point (vector).
- SDOT (by element): Dot oduct signed thmetic (vector, by element).
- SDOT (vector): Dot P duc gned arithme (vector).
- SHA1C: SHA1 hash update (c se).
- SHA1H: SHA1 ed rotate.
- SHA1M: SH hash update (majorit).
- SHA1P: S 1 hash date (parity).
- SHA1SU0: S A1 hedule u date 0.
- SHA SHA schedul pdate 1.
- S A256H SHA25 ha update (part 1).
- HA256H SHA256 ash update (part 2).
- HA25 256 schedule update 0.
- S 56SU1: SHA256 schedule update 1.
- SHA H: SHA512 Hash update part 1.
- SHA512 2: SHA512 Hash update part 2.
- SHA512SU0: SHA512 Schedule Update 0.
- SHA512SU1: SHA512 Schedule Update 1.
- SHADD: Signed Halving Add.
- SHL: Shift Left (immediate).
- SHLL, SHLL2: Shift Left Long (by element size).
- SHRN, SHRN2: Shift Right Narrow (immediate).
- SHSUB: Signed Halving Subtract.
- SLI: Shift Left and Insert (immediate).
- SM3PARTW1: SM3PARTW1.
- SM3PARTW2: SM3PARTW2.
- SM3SS1: SM3SS1.
- SM3TT1A: SM3TT1A.
- SM3TT1B: SM3TT1B.
- SM3TT2A: SM3TT2A.

- SM3TT2B: SM3TT2B.
- SM4E: SM4 Encode.
- SM4EKEY: SM4 Key.
- SMAX: Signed Maximum (vector).
- SMAXP: Signed Maximum Pairwise.
- SMAXV: Signed Maximum across Vector.
- SMIN: Signed Minimum (vector).
- SMINP: Signed Minimum Pairwise.
- SMINV: Signed Minimum across Vector.
- SMLAL, SMLAL2 (by element): Signed Multiply-Add Long (vector, by element).
- SMLAL, SMLAL2 (vector): Signed Multiply-Add Long (vector).
- SMLSL, SMLSL2 (by element): Signed Multiply-Subtract Long (vector, by element).
- SMLSL, SMLSL2 (vector): Signed Multiply-Subtract Long (vector).
- SMMLA (vector): Signed 8-bit integer matrix multiply-accumulate (vector).
- SMULL, SMULL2 (by element): Signed Multiply Long (vector, by element).
- SMULL, SMULL2 (vector): Signed Multiply Long (vector).
- SQABS: Signed saturating Absolute value.
- SQADD: Signed saturating Add.
- SQDMLAL, SQDMLAL2 (by element): Signed saturating Doubling Multiply-Add Long (by element).
- SQDMLAL, SQDMLAL2 (vector): Signed saturating Doubling Multiply-Add Long.
- SQDMLSL, SQDMLSL2 (by element): Signed saturating Doubling Multiply-Subtract Long (by element).
- SQDMLSL, SQDMLSL2 (vector): Signed saturating Doubling Multiply-Subtract Long.
- SQDMULH (by element): Signed saturating Doubling Multiply returning High half (by element).
- SQDMULH (vector): Signed saturating Doubling Multiply returning High half.
- SQDMULL, SQDMULL2 (by element): Signed saturating Doubling Multiply Long (by element).
- SQDMULL, SQDMULL2 (vector): Signed saturating Doubling Multiply Long.
- SQNEG: Signed saturating Negate.
- SQRDMLAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by
- element).
- SQRDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).
- SQRDMLSH (by element): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).
- SQRDMLSH (vector): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).
- SQRDMULH (by element): Signed saturating Rounding Doubling Multiply returning High half (by element).
- SQRDMULH (vector): Signed saturating Rounding Doubling Multiply returning High half.
- SQRSHL: Signed saturating Rounding Shift Left (register).
- SQRSHRN, SQRSHRN2: Signed saturating Rounded Shift Right Narrow (immediate).
- SQRSHRUN, SQRSHRUN2: Signed saturating Rounded Shift Right Unsigned Narrow (immediate).
- SQSHL (immediate): Signed saturating Shift Left (immediate).
- SQSHL (register): Signed saturating Shift Left (register).
- SQSHLU: Signed saturating Shift Left Unsigned (immediate).
- SQSHRN, SQSHRN2: Signed saturating Shift Right Narrow (immediate).
- SQSHRUN, SQSHRUN2: Signed saturating Shift Right Unsigned Narrow (immediate).
- SQSUB: Signed saturating Subtract.
- SQXTN, SQXTN2: Signed saturating extract Narrow.
- SQXTUN, SQXTUN2: Signed saturating extract Unsigned Narrow.
- SRHADD: Signed Rounding Halving Add.
- SRI: Shift Right and Insert (immediate).
- SRSHL: Signed Rounding Shift Left (register).
- SRSHR: Signed Rounding Shift Right (immediate).
- SRSRA: Signed Rounding Shift Right and Accumulate (immediate).
- SSHL: Signed Shift Left (register).
- SSHLL, SSHLL2: Signed Shift Left Long (immediate).

- SSHR: Signed Shift Right (immediate).
- SSRA: Signed Shift Right and Accumulate (immediate).
- SSUBL, SSUBL2: Signed Subtract Long.
- SSUBW, SSUBW2: Signed Subtract Wide.
- ST1 (multiple structures): Store multiple single-element structures from one, two, three, or four registers.
- ST1 (single structure): Store a single-element structure from one lane of one register.
- ST2 (multiple structures): Store multiple 2-element structures from two registers.
- ST2 (single structure): Store single 2-element structure from one lane of two registers.
- ST3 (multiple structures): Store multiple 3-element structures from three registers.
- ST3 (single structure): Store single 3-element structure from one lane of three registers.
- ST4 (multiple structures): Store multiple 4-element structures from four registers.
- ST4 (single structure): Store single 4-element structure from one lane of four registers.
- SUB (vector): Subtract (vector).
- SUBHN, SUBHN2: Subtract returning High Narrow.
- SUDOT (by element): Dot product with signed and unsigned int gers (vector, r element).
- SUQADD: Signed saturating Accumulate of Unsigned value.
- TBL: Table vector Lookup.
- TBX: Table vector lookup extension.
- TRN1: Transpose vectors (primary).
- TRN2: Transpose vectors (secondary).
- UABA: Unsigned Absolute difference and Accumulate.
- UABAL, UABAL2: Unsigned Absolute dif ence nd Ac mula Long.
- UABD: Unsigned Absolute Difference (ctor).
- UABDL, UABDL2: Unsigned Absolu Differe Long.
- UADALP: Unsigned Add and Accumulat g Pairwise.
- UADDL, UADDL2: Unsigned Ad ong (v or).
- UADDLP: Unsigned Add L g Pairw
- UADDLV: Unsigned sum ng across Ve or.
- UADDW, UADDW2: nsig d Add Wide.
- UCVTF (vector, fixed-point): U signed fixed-point Convert to Floating-point (vector).
- UCVTF (vector teger): Unsigne nteger Convert to Floating-point (vector).
- UDOT (by el ent): Dot Product un gned arithmetic (vector, by element).
- UDOT (ve r): Dot oduct unsigned arithmetic (vector).
- UHADD: Un ng Halving dd.
- UHS Unsig d Halvin ubtract.
- U AX: U signed ax um (vector).
- UMAXP: nsigned M ximum Pairwise.
- MAX Maximum across Vector.
- U : Unsigned Minimum (vector).
- UMI : Unsigned Minimum Pairwise.
- UMINV nsigned Minimum across Vector.
- UMLAL, UMLAL2 (by element): Unsigned Multiply-Add Long (vector, by element).
- UMLAL, UMLAL2 (vector): Unsigned Multiply-Add Long (vector).
- UMLSL, UMLSL2 (by element): Unsigned Multiply-Subtract Long (vector, by element).
- UMLSL, UMLSL2 (vector): Unsigned Multiply-Subtract Long (vector).
- UMMLA (vector): Unsigned 8-bit integer matrix multiply-accumulate (vector).
- UMULL, UMULL2 (by element): Unsigned Multiply Long (vector, by element).
- UMULL, UMULL2 (vector): Unsigned Multiply long (vector).
- UQADD: Unsigned saturating Add.
- UQRSHL: Unsigned saturating Rounding Shift Left (register).
- UQRSHRN, UQRSHRN2: Unsigned saturating Rounded Shift Right Narrow (immediate).
- UQSHL (immediate): Unsigned saturating Shift Left (immediate).
- UQSHL (register): Unsigned saturating Shift Left (register).
- UQSHRN, UQSHRN2: Unsigned saturating Shift Right Narrow (immediate).
- UQSUB: Unsigned saturating Subtract.

- UQXTN, UQXTN2: Unsigned saturating extract Narrow.
- URECPE: Unsigned Reciprocal Estimate.
- URHADD: Unsigned Rounding Halving Add.
- URSHL: Unsigned Rounding Shift Left (register).
- URSHR: Unsigned Rounding Shift Right (immediate).
- URSQRTE: Unsigned Reciprocal Square Root Estimate.
- URSRA: Unsigned Rounding Shift Right and Accumulate (immediate).
- USDOT (by element): Dot Product with unsigned and signed integers (vector, by element).
- USDOT (vector): Dot Product with unsigned and signed integers (vector).
- USHL: Unsigned Shift Left (register).
- USHLL, USHLL2: Unsigned Shift Left Long (immediate).
- USHR: Unsigned Shift Right (immediate).
- USMMLA (vector): Unsigned and signed 8-bit integer matrix multiply-accumulate (vector).
- USQADD: Unsigned saturating Accumulate of Signed value.
- USRA: Unsigned Shift Right and Accumulate (immediate).
- USUBL, USUBL2: Unsigned Subtract Long.
- USUBW, USUBW2: Unsigned Subtract Wide.
- UZP1: Unzip vectors (primary).
- UZP2: Unzip vectors (secondary).
- XAR: Exclusive OR and Rotate.
- XTN, XTN2: Extract Narrow.
- ZIP1: Zip vectors (primary).
- ZIP2: Zip vectors (secondary).

If execution of an illegal Advanced SIMD instruction is attempted when the PE is in *Streaming SVE mode*, and the instructions are not configured to trap, this will cause an SME exception to be taken, as defined by rule R_{DTCLZ} in C1.2.1 *Exception priorities*.

E1.1.1.2 Single-element instructions

This section lists by name those A64 Advanced SIMD instruction pages in which only the SIMD "Vector" encoding variants can be *illegal* when the PE is in *Streaming SVE mode*, but in which the single-element "Scalar" encoding variants are always *legal* in *Streaming SVE mode*.

The Vector encodings of Advanced SIMD instructions described in the following pages are affected in this way:

- FMULX: Floating-point Multiply extended.
- FRECPE: Floating-point Reciprocal Estimate.
- FRECPS: Floating-point Reciprocal Step.
- FRECPX: Floating-point Reciprocal Exponent.[1]
- FRSQRTE: Floating-point Reciprocal Square Root Estimate.
- FRSQRTS: Floating-point Reciprocal Square Root Step.

E1.1.1.3 Element move to general register

The following Advanced SIMD instructions and their aliases can only be *illegal* when the PE is in *Streaming SVE mode* if their immediate vector element index is greater than zero. They are always *legal* in *Streaming SVE mode* when their element index is zero:

- SMOV: Signed Move vector element to general-purpose register.
- UMOV: Unsigned Move vector element to general-purpose register.

The *64-bit to top half of 128-bit* and *Top half of 128-bit to 64-bit* variants from the following instruction page are part of the scalar floating-point instruction set and therefore execute normally when the PE is in *Streaming SVE mode*:

[1] FRECPX is an exception in that it only has a single-element form.

- FMOV (general): Floating-point Move to or from general-purpose register without conversion.

E1.1.2 Illegal SVE instructions

Allocated SVE and SVE2 instructions with encodings that match the following patterns are *illegal* when the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level:

A64 Encoding Pattern	SVE Instructions or Instruction Class
`0000 0100 xx1x xxxx 1010 xxxx xxxx xxxx`	SVE address generation
`0000 0100 xx1x xxxx 1011 0xxx xxxx xxxx`	SVE floating-point trig select coefficient
`0000 0100 xx1x xxxx 1011 10xx xxxx xxxx`	SVE floating-point exponential accelerator
`0000 0101 xx10 0001 100x xxxx xxxx xxxx`	SVE compress active elements
`0000 0101 101x xxxx 000x xxxx xxxx xxxx`	SVE permute vector segments
`0010 0101 xx01 1000 1111 000x xxx0 xxxx`	SVE predicate read from FFR (predicated)
`0010 0101 xx01 1001 1111 0000 0000 xxxx`	SVE predicate read from FFR (unpredicated)
`0010 0101 xx10 1000 1001 000x xxx0 0000`	SVE FFR write from predicate
`0010 0101 xx10 1100 1001 0000 0000 0000`	SVE FFR initialise
`0100 0101 000x xxxx 0110 1xxx xxxx xxxx`	`PMULLB`, `PMULLT` (128-bit result)
`0100 0101 xx0x xxxx 1001 10xx xxxx xxxx`	SVE integer matrix multiply accumulate
`0100 0101 xx0x xxxx 1011 xxxx xxxx xxxx`	SVE2 bitwise permute
`0100 0101 xx1x xxxx 100x xxxx xxxx xxxx`	SVE2 string processing
`0100 0101 xx1x xxxx 1010 00xx xxxx xxxx`	SVE2 histogram generation (segment)
`0100 0101 xx1x xxxx 110x xxxx xxxx xxxx`	SVE2 histogram computation (vector)
`0100 0101 xx10 0000 1110 0x00 000x xxxx`	SVE2 crypto unary operations
`0100 0101 xx1x xxxx 1111 0xxx xxxx xxxx`	SVE2 crypto constructive binary operations
`0100 0101 xx10 001x 1110 0xxx xxxx xxxx`	SVE2 crypto destructive binary operations
`0110 0100 xx1x xxxx 1110 01xx xxxx xxxx`	SVE floating point matrix multiply accumulate
`0110 0101 xx0x xxxx 0000 11xx xxxx xxxx`	`FTSMUL`
`0110 0101 xx01 0xxx 1000 00xx xxxx xxxx`	SVE floating-point trig multiply accumulate coefficient
`0110 0101 xx01 10xx 001x xxxx xxxx xxxx`	SVE floating-point serial reduction (predicated)
`1000 010x xx0x xxxx 0xxx xxxx xxxx xxxx`	SVE 32-bit gather load byte (scalar plus 32-bit unscaled offsets)
`1000 010x x00x xxxx 10xx xxxx xxxx xxxx`	SVE2 32-bit gather non-temporal load (scalar plus 32-bit unscaled offsets)
`1000 010x x00x xxxx 111x xxxx xxx0 xxxx`	SVE 32-bit gather prefetch (vector plus immediate)
`1000 0100 0x1x xxxx 0xxx xxxx xxx0 xxxx`	SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)
`1000 010x x01x xxxx 1xxx xxxx xxxx xxxx`	SVE 32-bit gather load (vector plus immediate)
`1000 0100 1x1x xxxx 0xxx xxxx xxxx xxxx`	SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)
`1000 0101 0x1x xxxx 0xxx xxxx xxxx xxxx`	SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

A64 Encoding Pattern	SVE Instructions or Instruction Class
`1010 0100 001x xxxx 000x xxxx xxxx xxxx`	`LD1ROB` (scalar plus scalar)
`1010 0100 101x xxxx 000x xxxx xxxx xxxx`	`LD1ROH` (scalar plus scalar)
`1010 0101 001x xxxx 000x xxxx xxxx xxxx`	`LD1ROW` (scalar plus scalar)
`1010 0101 101x xxxx 000x xxxx xxxx xxxx`	`LD1ROD` (scalar plus scalar)
`1010 0100 0010 xxxx 001x xxxx xxxx xxxx`	`LD1ROB` (scalar plus immediate)
`1010 0100 1010 xxxx 001x xxxx xxxx xxxx`	`LD1ROH` (scalar plus immediate)
`1010 0101 0010 xxxx 001x xxxx xxxx xxxx`	`LD1ROW` (scalar plus immediate)
`1010 0101 1010 xxxx 001x xxxx xxxx xxxx`	`LD1ROD` (scalar plus immedia
`1010 010x xxxx xxxx 011x xxxx xxxx xxxx`	SVE contiguous first-fa load (scalar p scalar)
`1010 010x xxx1 xxxx 101x xxxx xxxx xxxx`	SVE contiguous no fault l (scalar pl immediate)
`1100 010x xx0x xxxx 0xxx xxxx xxxx xxxx`	SVE 64-bit gath load (scalar p un cked 32-bit unscaled offsets)
`1100 010x x00x xxxx 1x0x xxxx xxxx xxxx`	SVE2 64-bit ther n temporal load (scalar plus unpacked 32-bit unscaled offsets)
`1100 010x x00x xxxx 111x xxxx xxx0 xxxx`	SV 4-bit ther p fetch vector plus immediate)
`1100 010x xx1x xxxx 0xxx xxxx xxxx xxxx`	E 64-bi ather load (scalar plus 32-bit unpacked scaled offsets)
`1100 0100 0x1x xxxx 0xxx xxxx xxx0 xxxx`	SV bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)
`1100 010x x01x xxxx 1xxx xxxx xxxx xxxx`	S 64-bi gather load (vector plus immediate)
`1100 010x x10x xxxx 1xxx xxxx x x xx`	SVE bit gather load (scalar plus 64-bit unscaled offsets)
`1100 010x x11x xxxx 1xxx xx xxxx xxxx`	SVE 64-bit gather load (scalar plus 64-bit scaled offsets)
`1100 0100 011x xxxx 1xx xxxx xx 0 xxxx`	SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)
`1110 010x x00x xxxx 001 xxx xxxx xxxx`	SVE2 64-bit scatter non-temporal store (vector plus scalar)
`1110 010x x10x x 01x x xxxx xxx`	SVE2 32-bit scatter non-temporal store (vector plus scalar)
`1110 010x xx xxxx 1 0x xxxx x x xxxx`	SVE scatter store with 32-bit offset
`1110 010x xxx xx 101x x xxxx xxxx`	SVE scatter store with 64-bit offset
`1110 010x xxxx x 101x xxxx xxxx xxxx`	SVE scatter store with immediate offset

The following SVE and SVE2 instructions and their aliases are affected:

- ADR: Compute vector address.
- AESD: AES single round decryption.
- AESE: AES single round encryption.
- AESIMC: AES inverse mix columns.
- AESMC: AES mix columns.
- BDEP: Scatter lower bits into positions selected by bitmask
- BEXT: Gather lower bits from positions selected by bitmask.
- BFMMLA: BFloat16 floating-point matrix multiply-accumulate.
- BGRP: Group bits to right or left as selected by bitmask.
- COMPACT: Shuffle active elements of vector to the right and fill with zero.
- FADDA: Floating-point add strictly-ordered reduction, accumulating in scalar.
- FEXPA: Floating-point exponential accelerator.

- FMMLA: Floating-point matrix multiply-accumulate.
- FTMAD: Floating-point trigonometric multiply-add coefficient.
- FTSMUL: Floating-point trigonometric starting value.
- FTSSEL: Floating-point trigonometric select coefficient.
- HISTCNT: Count matching elements in vector.
- HISTSEG: Count matchine elements in vector segments.
- LD1B (scalar plus vector): Gather load unsigned bytes to vector (vector index).
- LD1B (vector plus immediate): Gather load unsigned bytes to vector (immediate index).
- LD1D (scalar plus vector): Gather load doublewords to vector (vector index).
- LD1D (vector plus immediate): Gather load doublewords to vector (immediate index).
- LD1H (scalar plus vector): Gather load unsigned halfwords to vector (vector index).
- LD1H (vector plus immediate): Gather load unsigned halfwords to vector (immediate index).
- LD1ROB (scalar plus immediate): Contiguous load and replicate thirty-two bytes (immediate index).
- LD1ROB (scalar plus scalar): Contiguous load and replicate thirty-two bytes (scalar index).
- LD1ROD (scalar plus immediate): Contiguous load and replicate four doublewords (immediate index).
- LD1ROD (scalar plus scalar): Contiguous load and replicate four doublewords (scalar index).
- LD1ROH (scalar plus immediate): Contiguous load and replicate sixteen halfwords (immediate index).
- LD1ROH (scalar plus scalar): Contiguous load and replicate sixteen halfwords (scalar index).
- LD1ROW (scalar plus immediate): Contiguous load and replicate eight words (immediate index).
- LD1ROW (scalar plus scalar): Contiguous load and replicate eight words (scalar index).
- LD1SB (scalar plus vector): Gather load signed bytes to vector (vector index).
- LD1SB (vector plus immediate): Gather load signed bytes to vector (immediate index).
- LD1SH (scalar plus vector): Gather load signed halfwords to vector (vector index).
- LD1SH (vector plus immediate): Gather load signed halfwords to vector (immediate index).
- LD1SW (scalar plus vector): Gather load signed words to vector (vector index).
- LD1SW (vector plus immediate): Gather load signed words to vector (immediate index).
- LD1W (scalar plus vector): Gather load unsigned words to vector (vector index).
- LD1W (vector plus immediate): Gather load unsigned words to vector (immediate index).
- LDFF1B (scalar plus scalar): Contiguous load first-fault unsigned bytes to vector (scalar index).
- LDFF1B (scalar plus vector): Gather load first-fault unsigned bytes to vector (vector index).
- LDFF1B (vector plus immediate): Gather load first-fault unsigned bytes to vector (immediate index).
- LDFF1D (scalar plus scalar): Contiguous load first-fault doublewords to vector (scalar index).
- LDFF1D (scalar plus vector): Gather load first-fault doublewords to vector (vector index).
- LDFF1D (vector plus immediate): Gather load first-fault doublewords to vector (immediate index).
- LDFF1H (scalar plus scalar): Contiguous load first-fault unsigned halfwords to vector (scalar index).
- LDFF1H (scalar plus vector): Gather load first-fault unsigned halfwords to vector (vector index).
- LDFF1H (vector plus immediate): Gather load first-fault unsigned halfwords to vector (immediate index).
- LDFF1SB (scalar plus scalar): Contiguous load first-fault signed bytes to vector (scalar index).
- LDFF1SB (scalar plus vector): Gather load first-fault signed bytes to vector (vector index).
- LDFF1SB (vector plus immediate): Gather load first-fault signed bytes to vector (immediate index).
- LDFF1SH (scalar plus scalar): Contiguous load first-fault signed halfwords to vector (scalar index).
- LDFF1SH (scalar plus vector): Gather load first-fault signed halfwords to vector (vector index).
- LDFF1SH (vector plus immediate): Gather load first-fault signed halfwords to vector (immediate index).
- LDFF1SW (scalar plus scalar): Contiguous load first-fault signed words to vector (scalar index).
- LDFF1SW (scalar plus vector): Gather load first-fault signed words to vector (vector index).
- LDFF1SW (vector plus immediate): Gather load first-fault signed words to vector (immediate index).
- LDFF1W (scalar plus scalar): Contiguous load first-fault unsigned words to vector (scalar index).
- LDFF1W (scalar plus vector): Gather load first-fault unsigned words to vector (vector index).
- LDFF1W (vector plus immediate): Gather load first-fault unsigned words to vector (immediate index).
- LDNF1B: Contiguous load non-fault unsigned bytes to vector (immediate index).
- LDNF1D: Contiguous load non-fault doublewords to vector (immediate index).
- LDNF1H: Contiguous load non-fault unsigned halfwords to vector (immediate index).
- LDNF1SB: Contiguous load non-fault signed bytes to vector (immediate index).
- LDNF1SH: Contiguous load non-fault signed halfwords to vector (immediate index).
- LDNF1SW: Contiguous load non-fault signed words to vector (immediate index).

- LDNF1W: Contiguous load non-fault unsigned words to vector (immediate index).
- LDNT1B (vector plus scalar): Gather load non-temporal unsigned bytes.
- LDNT1D (vector plus scalar): Gather load non-temporal unsigned doublewords.
- LDNT1H (vector plus scalar): Gather load non-temporal unsigned halfwords.
- LDNT1SB: Gather load non-temporal signed bytes.
- LDNT1SH: Gather load non-temporal signed halfwords.
- LDNT1SW: Gather load non-temporal signed words.
- LDNT1W (vector plus scalar): Gather load non-temporal unsigned words.
- MATCH: Detect any matching elements, setting the condition flags.
- NMATCH: Detect no matching elements, setting the condition flags.
- PMULLB: Polynomial multiply long (bottom) [128b result only].
- PMULLT: Polynomial multiply long (top) [128b result only].
- PRFB (scalar plus vector): Gather prefetch bytes (scalar plus vector).
- PRFB (vector plus immediate): Gather prefetch bytes (vector plus i
- PRFD (scalar plus vector): Gather prefetch doublewords (scalar
- PRFD (vector plus immediate): Gather prefetch doublewords (v
- PRFH (scalar plus vector): Gather prefetch halfwords (sca
- PRFH (vector plus immediate): Gather prefetch halfwo
- PRFW (scalar plus vector): Gather prefetch words (s
- PRFW (vector plus immediate): Gather prefetch wo
- RAX1: Bitwise rotate left by 1 and exclusive OR.
- RDFFR (unpredicated): Read the first-fault
- RDFFR, RDFFRS (predicated): Return
- SETFFR: Initialise the first-fault regis
- SM4E: SM4 encryption and decryption.
- SM4EKEY: SM4 key updates.
- SMMLA: Signed integer ma
- ST1B (scalar plus vector
- ST1B (vector plus imm
- ST1D (scalar plus vector): Sca
- ST1D (vector pl
- ST1H (scalar
- ST1H (vec
- ST1W (scala
- ST1
- S
- STNT1D
- STNT1
- S
- TRN
- UMML
- USMMLA: Unsigned by signed integer matrix multiply-accumulate.
- UZP1, UZP2 (vectors, quadwords): Concatenate even or odd quadwords from two vectors.
- WRFFR: Write the first-fault register.
- ZIP1, ZIP2 (vectors, quadwords): Interleave quadwords from two half vectors.

If execution of an illegal SVE or SVE2 instruction is attempted when the PE is in *Streaming SVE mode*, and SVE instructions are not configured to trap, this will cause an SME exception to be taken, as defined by rule R_{PLYVH} in C1.2.1 *Exception priorities*.

E1.2 Unimplemented SVE instructions

If execution of any SVE or SVE2 instruction is attempted when the PE is not in *Streaming SVE mode* and FEAT_SVE or FEAT_SVE2 is not implemented by the PE, and the instructions are not configured to trap, this will cause an SME exception to be taken, as defined by rule R_{PLYVH} in C1.2.1 *Exception priorities*.

E1.3 Reduced performance in Streaming SVE mode

Instructions which are dependent on results generated from vector or SIMD&FP register sources written to a general-purpose destination register, a predicate destination register, or the NZCV condition flags, might be significantly delayed if the PE is in *Streaming SVE mode* and FEAT_SME_FA64 is not implemented or not enabled at the current Exception level.

The following subsections list the instructions that are affected by this change.

E1.3.1 Scalar floating-point instructions

The following scalar floating-point instructions are affected.

- FCCMP: Floating-point Conditional quiet Compare (scalar).
- FCCMPE: Floating-point Conditional signaling Compare (scalar).
- FCMP: Floating-point quiet Compare (scalar).
- FCMPE: Floating-point signaling Compare (scalar).
- FCVTAS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).
- FCVTAU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).
- FCVTMS (scalar): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).
- FCVTMU (scalar): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).
- FCVTNS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).
- FCVTNU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).
- FCVTPS (scalar): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).
- FCVTPU (scalar): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).
- FCVTZS (scalar, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).
- FCVTZS (scalar, integer): Floating-point Convert to Signed integer, rounding toward Zero (scalar).
- FCVTZU (scalar, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).
- FCVTZU (scalar, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).

This only applies to the variants of the following scalar floating-point instructions that write to a general-purpose register:

- FMOV (general): Floating-point Move to or from general-purpose register without conversion.

E1.3.2 SVE instructions

The following SVE instructions are affected.

- ANDS (predicates): Bitwise AND predicates.
- BICS (predicates): Bitwise clear predicates.
- BRKAS: Break after first true condition.
- BRKBS: Break before first true condition.
- BRKNS: Propagate break to next partition.
- BRKPAS: Break after first true condition, propagating from previous partition.
- BRKPBS: Break before first true condition, propagating from previous partition.
- CLASTA (scalar): Conditionally extract element after last to general-purpose register.
- CLASTB (scalar): Conditionally extract last element to general-purpose register.
- CMP<cc> (immediate): Compare vector to immediate.
- CMP<cc> (vectors): Compare vectors.
- CMP<cc> (wide elements): Compare vector to 64-bit wide elements.
- CNTP: Set scalar to count of true predicate elements.
- DECP (scalar): Decrement scalar by count of true predicate elements.
- EORS (predicates): Bitwise exclusive OR predicates.

- FAC<cc>: Floating-point absolute compare vectors.
- FCM<cc> (vectors): Floating-point compare vectors.
- FCM<cc> (zero): Floating-point compare vector with zero.
- INCP (scalar): Increment scalar by count of true predicate elements.
- LASTA (scalar): Extract element after last to general-purpose register.
- LASTB (scalar): Extract last element to general-purpose register.
- NANDS: Bitwise NAND predicates.
- NORS: Bitwise NOR predicates.
- ORNS (predicates): Bitwise inclusive OR inverted predicate.
- ORRS (predicates): Bitwise inclusive OR predicate.
- PFIRST: Set the first active predicate element to true.
- PNEXT: Find next active predicate.
- PTEST: Set condition flags for predicate.
- PTRUES: Initialise predicate from named constraint.
- SQDECP (scalar): Signed saturating decrement scalar by count of true predicate elements.
- SQINCP (scalar): Signed saturating increment scalar by count of true predicate elements.
- UQDECP (scalar): Unsigned saturating decrement scalar by count of true predicate elements.
- UQINCP (scalar): Unsigned saturating increment scalar by count of true predicate elements.

Chapter E2
SME Shared pseudocode

This section provides the full information for shared pseudocode functions added or modified by SME or SME2.

This content is from the **2022-12** version of *Arm® A64 Instruction Set Architecture, for A-profile architecture* [3], which contains the definitive details of the pseudocode.

E2.1 Pseudocode functions

E2.1.1 AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================

AArch64.CheckFPAdvSIMDEnabled()
    AArch64.CheckFPEnabled();
    // Check for illegal use of Advanced
    // SIMD in Streaming SVE Mode
    if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

E2.1.2 BFDotAdd

```
// BFDotAdd()
// ==========
// BFloat16 2-way dot-product and add to single-pre sion
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(32) BFDotAdd(bits(32) addend, bits(16) op1_a, bit (16) op1_b,
                  bits(16) op2_a, bits(16) op2_b, F Type fpc_in)
    FPCRType fpcr = fpcr_in;

    bits(32) prod;

    bits(32) result;
    if !HaveEBF16() || fpcr.EBF == '0' th    // Standard BFloat16 behaviors
        prod = FPAdd_BF16(BFM lH(op1_a, op2_ , BFMulH(op1_b, op2_b));
        result = FPAdd_BF16( ddend, pr );
    else                                     // Extended BFloat16 behaviors
        boolean isbfloa = TRUE;
        boolean fpexc = FALSE;  // Do not generate floating-point exceptions
        fpcr.DN = '1';          // Generate default NaN values
        prod = FP (op1_a, op1_b, 2_a, op2_b, fpcr, isbfloat16, fpexc);
        result = FPAdd(a dend, prod, fpcr, fpexc);

    return resu ;
```

E2.1.3 BFNeg

```
// B eg()
// ===

bits(16) B eg(bits(16) op)
    boolean  nor_altfp = TRUE;    // Honor alternate handling
    return BFNeg(op, honor_altfp);

// BFNeg()
// =======

bits(16) BFNeg(bits(16) op, boolean honor_altfp)

    if honor_altfp && !UsingAArch32() && HaveAltFP() then
        FPCRType fpcr = FPCR[];
        if fpcr.AH == '1' then
            boolean fpexc = FALSE;
            boolean isbfloat16 = TRUE;
            (fptype, -, -) = FPUnpackBase(op, fpcr, fpexc, isbfloat16);
            if fptype IN {FPType_SNaN, FPType_QNaN} then

                return op;        // When fpcr.AH=1, sign of NaN has no consequence

    return NOT(op<15>) : op<14:0>;
```

E2.1.4 CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

E2.1.5 CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// =============================
// Checks for traps on SVE instructions that are not legal in streaming mode.

CheckNonStreamingSVEEnabled()
    CheckSVEEnabled();

    if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

E2.1.6 CheckSMEAccess

```
// CheckSMEAccess()
// ================
// Check that access to SME System registers is enabled.

CheckSMEAccess()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
```

E2.1.7 CheckSMEAndZAEnabled

```
// CheckSMEAndZAEnabled()
// ======================

CheckSMEAndZAEnabled()
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

E2.1.8 CheckSMEEnabled

```
// CheckSMEEnabled()
// =================

CheckSMEEnabled()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTr (EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabl () the
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' th
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TR ;
                when '01' disabled = STATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = F SE;
            if disabled then SME cessTr MEExceptionType_AccessTrap, EL2);

            // Check SIMD&FP  EL2
            case CPTR_EL2.F N of
                when 'x0'  abled = TRU
                when '0  dis led = PSTA .EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disab d = FALSE;
            if dis ed then AArc 4.AdvSIMDFPAccessTrap(EL2);
        else
            if TR_EL2 CM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
            i PTR_EL TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Ch if ac ss disa ed in CPTR_EL3
    if veEL3) n
        if CP _EL3.E '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
        if C R_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
```

E2.1.9 CheckSM ZT0Enabled

```
// CheckSMEZT0Enabled()
// ====================
// Checks for ZT0 enabled.

CheckSMEZT0Enabled()
    // Check if ZA and ZT0 are inactive in PSTATE
    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);

    // Check if EL0/EL1 accesses to ZT0 are disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        if SMCR_EL1.EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL1);

    // Check if EL0/EL1/EL2 accesses to ZT0 are disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL2);
```

```
19
20      // Check if all accesses to ZT0 are disabled in SMCR_EL3
21      if HaveEL(EL3) then
22          if SMCR_EL3.EZT0 == '0' then
23              SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL3);
```

E2.1.10 CheckStreamingSVEAndZAEnabled

```
1  // CheckStreamingSVEAndZAEnabled()
2  // ===============================
3
4  CheckStreamingSVEAndZAEnabled()
5      CheckStreamingSVEEnabled();
6
7      if PSTATE.ZA == '0' then
8          SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

E2.1.11 CheckStreamingSVEEnabled

```
1  // CheckStreamingSVEEnabled()
2  // ==========================
3
4  CheckStreamingSVEEnabled()
5      CheckSMEEnabled();
6
7      if PSTATE.SM == '0' then
8          SMEAccessTrap(SMEExceptionTyp  NotStre ming, PSTATE.EL);
```

E2.1.12 CounterToPredicate

```
1  // CounterToPredicate()
2  // ====================
3
4  bits(width) CounterT Predicate( s(16) pred, integer width)
5      integer count;
6      integer esiz
7      integer el ents;
8      constant i ger  = CurrentVL;
9      constant inte  PL = VL  V 8;
10     inte axbit  Highes etBit(CeilPow2(PL * 4)<15:0>);
11     as rt ma bit <= ;
12     cs(PL*4  result;
13     oolean  t = pred<15> == '1';
14
15     ass  width == PL || width == PL*2 || width == PL*3 || width == PL*4;
16
17     if IsZe (pred<3:0>) then
18         retur Zeros(width);
19
20     case pred<3:0> of
21         when 'xxx1'
22             count = UInt(pred<maxbit:1>);
23             esize = 8;
24         when 'xx10'
25             count = UInt(pred<maxbit:2>);
26             esize = 16;
27         when 'x100'
28             count = UInt(pred<maxbit:3>);
29             esize = 32;
30         when '1000'
31             count = UInt(pred<maxbit:4>);
32             esize = 64;
33
34     elements = (VL * 4) DIV esize;
35     result = Zeros(PL*4);
```

```
    constant integer psize = esize DIV 8;
    for e = 0 to elements-1
        bit pbit = if e < count then '1' else '0';
        if invert then
            pbit = NOT(pbit);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);

    return result<width-1:0>;
```

E2.1.13 CurrentNSVL

```
// CurrentNSVL - non-assignment form
// =================================
// Non-Streaming VL

integer CurrentNSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);

    return vl;
```

E2.1.14 CurrentSVL

```
// CurrentSVL - non-assignment form
// ================================
// Streaming SVL

integer CurrentSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(SMCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(SMCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(SMCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(SMCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(SMCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSMEVectorLength(vl);

    return vl;
```

E2.1.15 CurrentVL

```
// CurrentVL - non-assignment form
// ===============================

integer CurrentVL
    return if HaveSME() && PSTATE.SM == '1' then CurrentSVL else CurrentNSVL;
```

E2.1.16 EncodePredCount

```
// EncodePredCount()
// =================

bits(width) EncodePredCount(integer esize, integer elements,
                            integer count_in, boolean invert_in, integer width)
    integer count = count_in;
    boolean invert = invert_in;
    constant integer PL = CurrentVL DIV 8;
    assert width == PL;
    assert esize IN {8, 16, 32, 64};
    assert count >=0 && count <= elements;
    bits(16) pred;

    if count == 0 then
        return Zeros(width);

    if invert then
        count = elements - count;
    elsif count == elements then
        count = 0;
        invert = TRUE;

    bit inv = (if invert then '1' else '0');
    case esize of
        when 8  pred = inv : count<13:0> : '1';
        when 16 pred = inv : count<12:0> : '10';
        when 32 pred = inv : count<11:0> : '100';
        when 64 pred = inv : count<10:0> : '1000';

    return ZeroExtend(pred, width);
```

E2.1.17 FPAdd_ZA

```
// FPAdd_ZA()
// ==========
// Calculate op1+op2 for SME2 ZA-targeting instructions.

bits(N) FPAdd_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;      // Do not generate floating-point exceptions
    fpcr.DN = '1';              // Generate default NaN values
    return FPAdd(op1, op2, fpcr, fpexc);
```

E2.1.18 FPDot

```
// FPDot()
// =======
// Calculates single-precision result of 2-way 16-bit floating-point dot-product
// with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
// determines whether input operands are BFloat16 or half-precision type.
// and 'fpexc' controls the generation of floating-point exceptions.

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
              bits(N DIV 2) op2_b, FPCRType fpcr, boolean isbfloat16)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
```

```
13
14  bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
15                bits(N DIV 2) op2_b, FPCRType fpcr_in, boolean isbfloat16, boolean fpexc)
16      FPCRType fpcr = fpcr_in;
17
18      assert N == 32;
19      bits(N) result;
20      boolean done;
21      fpcr.AHP = '0';           // Ignore alternative half-precision option
22      rounding = FPRoundingMode(fpcr);
23
24      (type1_a,sign1_a,value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloat16);
25      (type1_b,sign1_b,value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloat16);
26      (type2_a,sign2_a,value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloat16);
27      (type2_b,sign2_b,value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloat16);
28
29      inf1_a = (type1_a == FPType_Infinity); zero1_a = (type1_a
30      inf1_b = (type1_b == FPType_Infinity); zero1_b = (type1
31      inf2_a = (type2_a == FPType_Infinity); zero2_a = (type
32      inf2_b = (type2_b == FPType_Infinity); zero2_b = (type
33
34      (done,result) = FPProcessNaNs4(type1_a, type1_b
35                                     op1_a, op1_b,
36
37      if (((inf1_a && zero2_a) || (zero1_a && inf2_
38          ((inf1_b && zero2_b) || (zero1_b && inf2_b
39          result = FPDefaultNaN(fpcr, N);
40          if fpexc then FPProcessException(FPExc_Invalid
41
42      if !done then
43          // Determine sign and type products will have if it does not cause an Invalid
44          // Operation.
45          signPa = sign1_a EOR sign2_
46          signPb = sign1_b EOR sign2_b;
47          infPa = inf1_a || inf2_a;
48          infPb = inf1_b || inf2_b;
49          zeroPa = zero1_a || zero2_a;
50          zeroPb = zero1_b || zero2_b;
51
52          // Non SNaN-generated Invalid Operation cases are multiplies of zero
53          // by infinity and additions of opposite-signed infinities.
54          invalidop = (inf1_a && zero2_a) || (zero1_a && inf2_a) ||
55              (inf1_b && zero2_b) || (zero1_b && inf2_b) || (infPa && infPb && signPa !=
                ↪signPb));
56
57          if invalidop then
58              result = FPDefaultNaN(fpcr, N);
59              if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
60
61          // Other cases involving infinities produce an infinity of the same sign.
62          elsif (infPa && signPa == '0') || (infPb && signPb == '0') then
63              result = FPInfinity('0', N);
64          elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
65              result = FPInfinity('1', N);
66
67          // Cases where the result is exactly zero and its sign is not determined by the
68          // rounding mode are additions of same-signed zeros.
69          elsif zeroPa && zeroPb && signPa == signPb then
70              result = FPZero(signPa, N);
71
72          // Otherwise calculate fused sum of products and round it.
73          else
74              result_value = (value1_a * value2_a) + (value1_b * value2_b);
75              if result_value == 0.0 then  // Sign of exact zero result depends on rounding
                    ↪mode
76                  result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
77                  result = FPZero(result_sign, N);
78              else
79                  result = FPRound(result_value, fpcr, rounding, fpexc, N);
```

```
80
81      return result;
```

E2.1.19 FPDotAdd

```
1  // FPDotAdd()
2  // ==========
3  // Half-precision 2-way dot-product and add to single-precision.
4
5  bits(N) FPDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
6                   bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr)
7      assert N == 32;
8
9      bits(N) prod;
10     boolean isbfloat16 = FALSE;
11     boolean fpexc = TRUE;      // Generate floating-point exceptions
12     prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
13     result = FPAdd(addend, prod, fpcr, fpexc);
14
15     return result;
```

E2.1.20 FPDotAdd_ZA

```
1  // FPDotAdd_ZA()
2  // =============
3  // Half-precision 2-way dot-product and add to single-precision
4  // for SME ZA-targeting instructions.
5
6  bits(N) FPDotAdd_ZA(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
7                      bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr_in)
8      FPCRType fpcr = fpcr_in;
9      assert N == 32;
10
11     bits(N) prod;
12     boolean isbfloat16 = FALSE;
13     boolean fpexc = FALSE;     // Do not generate floating-point exceptions
14     fpcr.DN = '1';             // Generate default NaN values
15     prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
16     result = FPAdd(addend, prod, fpcr, fpexc);
17
18     return result;
```

E2.1.21 FPMulAdd_ZA

```
1  // FPMulAdd_ZA()
2  // =============
3  // Calculate addend + op1*op2 with a single rounding for SME ZA-targeting
4  // instructions.
5
6  bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr_in)
7      FPCRType fpcr = fpcr_in;
8      boolean fpexc = FALSE;      // Do not generate floating-point exceptions
9      fpcr.DN = '1';              // Generate default NaN values
10     return FPMulAdd(addend, op1, op2, fpcr, fpexc);
```

E2.1.22 FPMulAddH_ZA

```
1  // FPMulAddH_ZA()
2  // ==============
3  // Calculates addend + op1*op2 for SME2 ZA-targeting instructions.
4
5  bits(N) FPMulAddH_ZA(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr_in)
6      FPCRType fpcr = fpcr_in;
7      boolean fpexc = FALSE;      // Do not generate floating-point exceptions
```

```
    fpcr.DN = '1';                // Generate default NaN values
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);
```

E2.1.23 FPProcessDenorms4

```
// FPProcessDenorms4()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms4(FPType type1, FPType type2, FPType type3, FPType type4, integer N,
    ↪FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal || type4 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

E2.1.24 FPProcessNaNs4

```
// FPProcessNaNs4()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it is and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

(boolean, bits(N)) FPProcessNaNs4(FPType type1, FPType type2, FPType type3, FPType type4,
                                  bits(N DIV 2) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                  bits(N DIV 2) op4, FPCRType fpcr, boolean fpexc)

    assert N == 32;

    bits(N) result;
    boolean done;
    // The FPCR.AH control does not affect these checks
    if type1 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type4 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type4 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
    else
        done = FALSE; result = Zeros(N); // 'Don't care' result

    return (done, result);
```

E2.1.25 FPSub_ZA

```
// FPSub_ZA()
// ==========
// Calculates op1-op2 for SME2 ZA-targeting instructions.

```

```
bits(N) FPSub_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;      // Do not generate floating-point exceptions
    fpcr.DN = '1';              // Generate default NaN values
    return FPSub(op1, op2, fpcr, fpexc);
```

E2.1.26 HaveEBF16

```
// HaveEBF16()
// ===========
// Returns TRUE if the EBF16 extension is implemented, FALSE otherwise.

boolean HaveEBF16()
    return IsFeatureImplemented(FEAT_EBF16);
```

E2.1.27 HaveSME

```
// HaveSME()
// =========
// Returns TRUE if the SME extension is implemented, FALSE otherwise.

boolean HaveSME()
    return IsFeatureImplemented(FEAT_SME);
```

E2.1.28 HaveSME2

```
// HaveSME2()
// ==========
// Returns TRUE if the SME2 extension is implemented, FALSE otherwise.

boolean HaveSME2()
    return IsFeatureImplemented(FEAT_SME2);
```

E2.1.29 HaveSMEF64F64

```
// HaveSMEF64F64()
// ===============
// Returns TRUE if the SMEF64F64 extension is implemented, FALSE otherwise.

boolean HaveSMEF64F64()
    return IsFeatureImplemented(FEAT_SME_F64F64);
```

E2.1.30 HaveSMEI16I64

```
// HaveSMEI16I64()
// ===============
// Returns TRUE if the SMEI16I64 extension is implemented, FALSE otherwise.

boolean HaveSMEI16I64()
    return IsFeatureImplemented(FEAT_SME_I16I64);
```

E2.1.31 ImplementedSMEVectorLength

```
// ImplementedSMEVectorLength()
// ============================
// Reduce SVE/SME vector length to a supported value (power of two)

integer ImplementedSMEVectorLength(integer nbits_in)
    integer maxbits = MaxImplementedSVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
```

```
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbits
    while nbits > 128 do
        if IsPow2(nbits) && SupportedPowerTwoSVL(nbits) then return nbits;
        nbits = nbits - 128;

    // Return the smallest supported power-of-two VL
    nbits = 128;
    while nbits < maxbits do
        if SupportedPowerTwoSVL(nbits) then return nbits;
        nbits = nbits * 2;

    // The only option is maxbits
    return maxbits;
```

E2.1.32 InStreamingMode

```
// InStreamingMode()
// =================

boolean InStreamingMode()
    return HaveSME() && PSTATE.SM == '1';
```

E2.1.33 IsFullA64Enabled

```
// IsFullA64Enabled()
// ==================
// Returns TRUE is full A64 is enabled in streaming mode and FALSE othersise.

boolean IsFullA64Enabled()
    if !HaveSMEFullA64() then return FALSE;

    // Check if full SVE disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check full SVE at EL0/EL1
        if SMCR_EL1.FA64 == '0' then return FALSE;

    // Check if full SVE disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.FA64 == '0' then return FALSE;

    // Check if full SVE disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.FA64 == '0' then return FALSE;

    return TRUE;
```

E2.1.34 IsMerging

```
// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRType fpcr)
    bit nep = if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then '0' else fpcr.NEP;
    return HaveAltFP() && !UsingAArch32() && nep == '1';
```

E2.1.35 IsOriginalSVEEnabled

```
// IsOriginalSVEEnabled()
// ======================
// Returns TRUE if access to SVE functionality is enabled at the target
```

```
// exception level and FALSE otherwise.

boolean IsOriginalSVEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HC_EL2.TGE     '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALS

    // Check if access disabled in CPTR_  3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then re  rn FAL

    return TRUE;
```

E2.1.36 IsSMEEnabled

```
// IsSMEEnabled()
// ==============
// Returns TRUE if access to SME functionality is enabled at the target
// exception level and FALSE otherwise.

boolean IsSMEEn  led   its(2) el)
    boolean disab  ,
    if   AArch  (el)  en
        retur  FALSE;

    // Chec  s disabled in CPACR_EL1
    i  N {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        ca  CPACR_EL1.SMEN of
            en 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TSM == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then return FALSE;
```

```
34
35      return TRUE;
```

E2.1.37 IsSVEEnabled

```
1  // IsSVEEnabled()
2  // ==============
3  // Returns TRUE if access to SVE registers is enabled at the target exception
4  // level and FALSE otherwise.
5
6  boolean IsSVEEnabled(bits(2) el)
7      if HaveSME() && PSTATE.SM == '1' then
8          return IsSMEEnabled(el);
9      elsif HaveSVE() then
10         return IsOriginalSVEEnabled(el);
11     else
12         return FALSE;
```

E2.1.38 Lookup

```
1  bits(512) _ZT0;
```

E2.1.39 MaybeZeroSVEUppers

```
1  // MaybeZeroSVEUppers()
2  // ====================
3
4  MaybeZeroSVEUppers(bits(2) target_el)
5      boolean lower_enabled;
6
7      if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
8          return;
9
10     if target_el == EL3 then
11         if EL2Enabled() then
12             lower_enabled = IsFPEnabled(EL2);
13         else
14             lower_enabled = IsFPEnabled(EL1);
15     elsif target_el == EL2 then
16         assert !ELUsingAArch32(EL2);
17         if HCR_EL2.TGE == '0' then
18             lower_enabled = IsFPEnabled(EL1);
19         else
20             lower_enabled = IsFPEnabled(EL0);
21     else
22         assert target_el == EL1 && !ELUsingAArch32(EL1);
23         lower_enabled = IsFPEnabled(EL0);
24
25     if lower_enabled then
26         constant integer VL = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
27         constant integer PL = VL DIV 8;
28         for n = 0 to 31
29             if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
30                 _Z[n] = ZeroExtend(_Z[n]<VL-1:0>, MAX_VL);
31         for n = 0 to 15
32             if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
33                 _P[n] = ZeroExtend(_P[n]<PL-1:0>, MAX_PL);
34         if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
35             _FFR = ZeroExtend(_FFR<PL-1:0>, MAX_PL);
36         if HaveSME() && PSTATE.ZA == '1' then
37             constant integer SVL = CurrentSVL;
38             constant integer accessiblevecs = SVL DIV 8;
39             constant integer allvecs = MaxImplementedSVL() DIV 8;
40
41             for n = 0 to accessiblevecs - 1
```

```
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = ZeroExtend(_ZA[n]<SVL-1:0>, MAX_VL);
            for n = accessiblevecs to allvecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = Zeros(MAX_VL);
```

E2.1.40 PredCountTest

```
// PredCountTest()
// ===============

bits(4) PredCountTest(integer elements, integer count, boolean invert)
    bit n, z, c, v;
    z = (if count == 0 then '1' else '0');              // none active
    if !invert then
        n = (if count != 0 then '1' else '0');          // first active
        c = (if count == elements then '0' else '1');   // NOT last active
    else
        n = (if count == elements then '1' else '0');   // NOT first active
        c = (if count != 0 then '0' else '1');          // last active
    v = '0';

    return n:z:c:v;
```

E2.1.41 ResetSMEState

```
// ResetSMEState()
// ===============

ResetSMEState()
    integer vectors = MAX_VL DIV 8;
    for n = 0 to vectors -
        _ZA[n] = Zeros(MA );
    _ZT0 = Zeros(ZT0_LE );
```

E2.1.42 ResetSVEState

```
// ResetSVESta ()
// =============

ResetSV state
    f r n = 0 to 31
        _Z[n] = Zeros(MAX_VL);
    f n = 0 to
        [n] = Zeros(MAX_PL);
    _FFR = Zeros(MAX_PL);
    FPSR = roExtend(0x0800009f<31:0>, 64);
```

E2.1.43 SetPSTATE_SM

```
// SetPSTATE_SM()
// ==============

SetPSTATE_SM(bit value)
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
```

E2.1.44 SetPSTATE_SVCR

```
// SetPSTATE_SVCR
// ==============

```

```
SetPSTATE_SVCR(bits(32) svcr)
    SetPSTATE_SM(svcr<0>);
    SetPSTATE_ZA(svcr<1>);
```

E2.1.45 SetPSTATE_ZA

```
// SetPSTATE_ZA()
// ==============

SetPSTATE_ZA(bit value)
    if PSTATE.ZA != value then
        ResetSMEState();
        PSTATE.ZA = value;
```

E2.1.46 SMEAccessTrap

```
// SMEAccessTrap()
// ===============
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SMEAccessTrap(SMEExceptionType etype, bits(2) target_el_in)
    bits(2) target_el = target_el_in;
    assert UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL0 then
        target_el = EL1;
    boolean route_to_el2;
    route_to_el2 = PSTATE.EL == EL0 && target_el == EL1 && EL2Enabled() && HCR_EL2.TGE ==
        ↪'1';

    exception = ExceptionSyndrome(Exception_SMEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    case etype of
        when SMEExceptionType_AccessTrap
            exception.syndrome<2:0> = '000';
        when SMEExceptionType_Streaming
            exception.syndrome<2:0> = '001';
        when SMEExceptionType_NotStreaming
            exception.syndrome<2:0> = '010';
        when SMEExceptionType_InactiveZA
            exception.syndrome<2:0> = '011';
        when SMEExceptionType_InaccessibleZT0
            exception.syndrome<2:0> = '100';

    if route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

E2.1.47 System

```
// System Registers
// ================

array bits(MAX_VL) _ZA[0..255];
```

E2.1.48 ZAhslice

```
// ZAhslice[] - non-assignment form
// ================================

bits(width) ZAhslice[integer tile, integer esize, integer slice, integer width]
    assert esize IN {8, 16, 32, 64, 128};
```

```
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    return ZAvector[tile + slice * tiles, width];

// ZAhslice[] - assignment form
// ============================

ZAhslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    ZAvector[tile + slice * tiles, width] = value;
```

E2.1.49 ZAslice

```
// ZAslice[] - non-assignment form
// ===============================

bits(width) ZAslice[integer tile, integer esize, boolean vertical, integer slice, integer
    ↪width]
    bits(width) result;

    if vertical then
        result = ZAvslice[tile, esize, slice, width];
    else
        result = ZAhslice[tile, esize, slice, width];

    return result;

// ZAslice[] - assignment form
// ===========================

ZAslice[integer tile, integer esize, boolean vertical,
        integer slice, integer width] = bits(width) value
    if vertical then
        ZAvslice[tile, esize, slice, width] = value;
    else
        ZAhslice[tile, esize, slice, width] = value;
```

E2.1.50 ZAtile

```
// ZAtile[] - non-assignment form
// ==============================

bits(width) ZAtile[integer tile, integer esize, integer width]
    constant integer SVL = CurrentSVL;
    integer slices = SVL DIV esize;
    assert width == SVL * slices;
    bits(width) result;

    for slice = 0 to slices-1
        Elem[result, slice, SVL] = ZAhslice[tile, esize, slice, SVL];

    return result;

// ZAtile[] - assignment form
// ==========================

ZAtile[integer tile, integer esize, integer width] = bits(width) value
    constant integer SVL = CurrentSVL;
    integer slices = SVL DIV esize;
```

```
    assert width == SVL * slices;

    for slice = 0 to slices-1
        ZAhslice[tile, esize, slice, SVL] = Elem[value, slice, SVL];
```

E2.1.51 ZAvector

```
// ZAvector[] - non-assignment form
// ================================

bits(width) ZAvector[integer index, integer width]
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    return _ZA[index]<width-1:0>;

// ZAvector[] - assignment form
// ============================

ZAvector[integer index, integer width] = bits(width) value
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    if ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
        _ZA[index] = ZeroExtend(value, MAX_VL);
    else
        _ZA[index]<width-1:0> = value;
```

E2.1.52 ZAvslice

```
// ZAvslice[] - non-assignment form
// ================================

bits(width) ZAvslice[integer tile, integer esize, integer slice, integer width]
    integer slices = CurrentSVL DIV esize;
    bits(width) result;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[result, s, esize] = Elem[hslice, slice, esize];

    return result;

// ZAvslice[] - assignment form
// ============================

ZAvslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
    integer slices = CurrentSVL DIV esize;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[hslice, slice, esize] = Elem[value, s, esize];
        ZAhslice[tile, esize, s, width] = hslice;
```

E2.1.53 ZT0

```
// ZT0[] - non-assignment form
// ===========================

bits(width) ZT0[integer width]
    assert width == 512;
    return _ZT0<width-1:0>;

// ZT0[] - assignment form
// =======================
```

```
10
11 ZT0[integer width] = bits(width) value
12     assert width == 512;
13     _ZT0<width-1:0> = value;
```

Chapter E3
System registers affected by SME

This section provides the full information for System registers added or modified by SME or SME2.

This content is from the **2022-12** version of *Arm® Architecture Registers, for A-profile architecture* [2], which contains the definitive version of the register information.

E3.1 SME-Specific System registers

System registers that are added to support SME architecture.

E3.1.1 ID_AA64SMFR0_EL1, SME Feature ID register 0

The ID_AA64SMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented features of the AArch64 Scalable Matrix Extension.

The fields in this register do not follow the standard ID scheme. See Alternative ID scheme used for ID_AA64SMFR0_EL1 .

Configuration

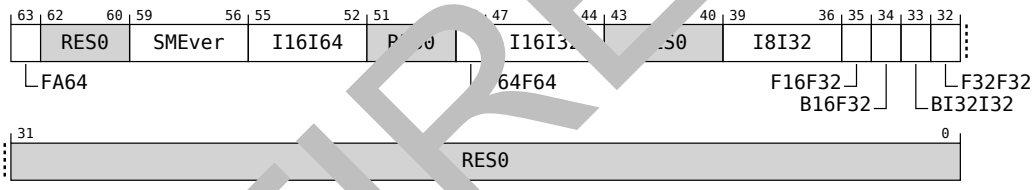
Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_AA64SMFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64SMFR0_EL1 bit assignments are:



FA64, bit [63]

Indicates support for execution of the full A64 instruction set when the PE is in Streaming SVE mode. Defined values are:

FA64	Meaning
0b0	Only those A64 instructions defined as being legal can be executed in Streaming SVE mode.
0b1	All implemented A64 instructions are legal for execution in Streaming SVE mode, when enabled by SMCR_EL1.FA64, SMCR_EL2.FA64, and SMCR_EL3.FA64.

FEAT_SME_FA64 implements the functionality identified by the value 0b1.

Bits [62:60]

Reserved, RES0.

SMEver, bits [59:56]

When ID_AA64PFR1_EL1.SME != 0b0000:

Indicates support for SME instructions when FEAT_SME is implemented. Defined values are:

SMEver	Meaning
0b0000	The mandatory SME instructions are implemented.
0b0001	As 0b0000, and adds the mandatory SME2 instructions.

All other values are reserved.

If FEAT_SME is implemented and FEAT_SME2 is not implemented, the only permitted value is 0b0000.

If FEAT_SME2 is implemented the only permitted value is 0b0001.

Otherwise:

RES0

I16I64, bits [55:52]

Indicates SME support for instructions that accumulate into 64-bit integer elements in the ZA array. Defined values are:

I16I64	Meaning
0b0000	Instructions that accumulate into 64-bit integer elements in the ZA array are not implemented.
0b1111	The variants of the ADDHA, ADDVA, SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS instructions that accumulate into 64-bit integer tiles are implemented. When FEAT_SME2 is implemented, the variants of the ADD, ADDA, SDOT, SMLALL, SMLSLL, SUB, SUBA, SVDOT, UDOT, UMLALL, UMLSLL, and UVDOT instructions that accumulate into 64-bit integer elements in ZA array vectors are implemented.

All other values are reserved.

FEAT_SME_I16I64 implements the functionality identified by the value 0b1111.

The only permitted values are 0b0000 and 0b1111.

Bits [51:49]

Reserved, RES0.

F64F64, bit [48]

Indicates SME support for instructions that accumulate into FP64 double-precision floating-point elements in the ZA array. Defined values are:

F64F64	Meaning
0b0	Instructions that accumulate into double-precision floating-point elements in the ZA array are not implemented.
0b1	The variants of the FMOPA and FMOPS instructions that accumulate into double-precision tiles are implemented. When FEAT_SME2 is implemented, the variants of the FADD, FMLA, FMLS, and FSUB instructions that accumulate into double-precision elements in ZA array vectors are imple[illegible]

FEAT_SME_F64F64 implements the functionality identified by the value 0b1.

I16I32, bits [47:44]

Indicates SME2 support for instructions that accumulate [illegible]bit [illegible]ter products into 32-bit integer tiles. Defined values are:

I16[illegible]2	Meaning
[illegible]b0000	Instructions that accumulate 16-bit outer products into 32-bit integer tiles are not implemented.
[illegible]b0101	The SMOPA (2-way), SMOPS (2-way), UMOPA (2-way), and UMOPS (2-way) instructions that accumulate 16-bit outer products into 32-bit integer tiles are implemented.

All other [illegible] are re[illegible]rved.

If FE[illegible]_SME[illegible]s implel[illegible]ted, the only permitted value is 0b0101. Otherwise, the only permitted value is 0b0[illegible]0.

Bits [43:40]

Reserved, RE[illegible]

I8I32, bits [39:36]

Indicates SME support for instructions that accumulate 8-bit integer outer products into 32-bit integer tiles. Defined values are:

I8I32	Meaning
0b0000	Instructions that accumulate 8-bit outer products into 32-bit tiles are not implemented.
0b1111	The SMOPA, SMOPS, SUMOPA, SUMOPS, UMOPA, UMOPS, USMOPA, and USMOPS instructions that accumulate 8-bit outer products into 32-bit tiles are implemented.

All other values are reserved.

If FEAT_SME is implemented, the only permitted value is 0b1111.

F16F32, bit [35]

Indicates SME support for instructions that accumulate FP16 half-precision floating-point outer products into FP32 single-precision floating-point tiles. Defined values are:

F16F32	Meaning
0b0	Instructions that accumulate half-precision outer products into single-precision tiles are not implemented.
0b1	The FMOPA and FMOPS instructions that accumulate half-precision outer products into single-precision tiles are implemented.

If FEAT_SME is implemented, the only permitted value is 0b1.

B16F32, bit [34]

Indicates SME support for instructions that accumulate BFloat16 outer products into FP32 single-precision floating-point tiles. Defined values are:

B16F32	Meaning
0b0	Instructions that accumulate BFloat16 outer products into single-precision tiles are not implemented.
0b1	The BFMOPA and BFMOPS instructions that accumulate BFloat16 outer products into single-precision tiles are implemented.

If FEAT_SME is implemented, the only permitted value is 0b1.

BI32I32, bit [33]

Indicates SME support for instructions that accumulate thirty-two 1-bit binary outer products into 32-bit integer tiles. Defined values are:

BI32I32	Meaning
0b0	Instructions that accumulate 1-bit binary outer products into 32-bit integer tiles are not implemented.
0b1	The BMOPA and BMOPS instructions that accumulate 1-bit binary outer products into 32-bit integer tiles are implemented.

If FEAT_SME2 is implemented, the only permitted value is 0b1. Otherwise, the only permitted value is 0b0.

F32F32, bit [32]

Indicates SME support for instructions that accumulate FP32 single-precision floating-point outer products into single-precision floating-point tiles. Defined values are:

F32F32	Meaning
0b0	Instructions that accumulate single-precision outer products into single-precision tiles are not implemented.
0b1	The FMOPA and FMOPS instructions that accumulate single-precision outer products into single-precision tiles are implemented.

If FEAT_SME is implemented, the only permitted value is 0b1.

Bits [31:0]

Reserved, RES0.

Accessing ID_AA64SMFR0_EL1

This register is read-only and can be accessed from EL1 and higher.

This register is only accessible from the AArch64 state.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64SMFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b101

```
if PSTATE.EL == EL0 then
    if !IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64SMFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64SMFR0_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_AA64SMFR0_EL1;
```

E3.1.2 MPAMSM_EL1, MPAM Streaming Mode Register

The MPAMSM_EL1 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests that are:

- Issued due to the execution of SME load and store instructions.
- Issued when the PE is in Streaming SVE mode due to the execution of SVE and SIMD&FP load and store instructions and SVE prefetch instructions.

If an implementation uses a shared SMCU, then the MPAM labels in this register have precedence over the labels in MPAM0_EL1, MPAM1_EL1, MPAM2_EL2, and MPAM3_EL3.

If an implementation includes an SMCU that is not shared with other PEs, then it is IMPLEMENTATION DEFINED whether the MPAM labels in this register have precedence over the labels in MPAM0_EL1, MPAM1_EL1, MPAM2_EL2, and MPAM3_EL3.

The MPAM labels in this register are only used if MPAM1_EL1.MPAMEN is 1.

For memory requests issued from EL0, the MPAM PARTID in this register is virtual and mapped into a physical PARTID when all of the following are true:

- EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} is not {1, 1}.
- The MPAM virtualization option is implemented and MPAMHCR_EL2.EL0_VPMEN is 1.

For memory requests issued from EL1, the MPAM PARTID in this register is virtual and mapped into a physical PARTID when all of the following are true:

- EL2 is implemented and enabled in the current Security state.
- The MPAM virtualization option is implemented and MPAMHCR_EL2.EL1_VPMEN is 1.

Configuration

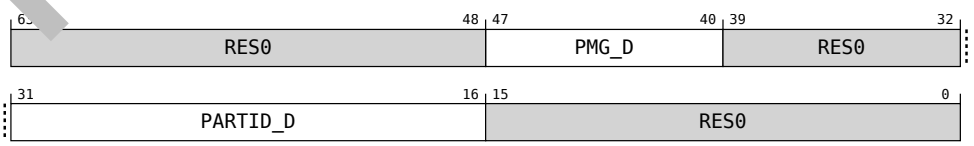
This register is present only when FEAT_MPAM is implemented and FEAT_SME is implemented. Otherwise, direct accesses to MPAMSM_EL1 are UNDEFINED.

Attributes

MPAMSM_EL1 is a 64-bit register.

Field descriptions

The MPAMSM_EL1 bit assignments are:



Bits [63:48]

Reserved, RES0.

PMG_D, bits [47:40]

Performance monitoring group property for PARTID_D.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [39:32]

Reserved, RES0.

PARTID_D, bits [31:16]

Partition ID for requests issued due to the execution at any Exception level of SME load and store instructions and, when the PE is in Streaming SVE mode, SVE and SIMD&FP load and store instructions and SVE prefetch instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [15:0]

Reserved, RES0.

Accessing MPAMSM_EL1

None of the fields in this register are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, MPAMSM_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && MPAM2_EL2.EnMPAMSM == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MPAMSM_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = MPAMSM_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = MPAMSM_EL1;
```

MSR MPAMSM_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && MPAM2_EL2.EnMPAMSM == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        MPAMSM_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        MPAMSM_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    MPAMSM_EL1 = X[t, 64];
```

E3.1.3 SMCR_EL1, SME Control Register (EL1)

The SMCR_EL1 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at Exception levels EL1 and EL0.

Configuration

This register has no effect if the PE is not in Streaming SVE mode.

When HCR_EL2.{E2H, TGE} == {1, 1} and EL2 is enabled in the current Security state, this register has no effect on execution at EL0 and EL1.

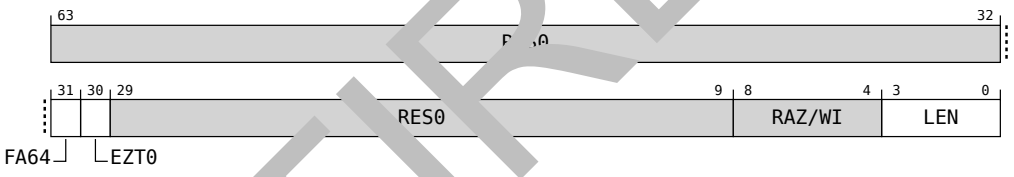
This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMCR_EL1 are UNDEFINED.

Attributes

SMCR_EL1 is a 64-bit register.

Field descriptions

The SMCR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

FA64, bit [31]

When FEAT_SME_FA64 is implemented:

Controls whether execution of an A64 instruction is considered legal when the PE is in Streaming SVE mode.

FA64	Meaning
0b0	This control does not cause any instruction to be treated as legal in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL1 and EL0, if they are treated as legal at more privileged Exception levels in the current Security state.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at EL1 and EL0 of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

The exception is reported using ESR_EL1.EC or ESR_EL2.EC value 0x1D, with an ISS code of 0x0000004, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.

EZT0	Meaning
0b0	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b1	This control does not cause execution of any instruction to be trapped.

Changes to this field only affect whether instructions that access ZT0 are trapped. They do not affect the contents of ZT0, which remain valid so long as PSTATE.ZA is

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [29:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ

LEN, bits [3:0]

Requests an Effective Streaming SVE vector length (SVL) at EL1 of (LEN+1)*128 bits. This field also defines the Effective Streaming SVE vector length at EL0 when EL2 is not implemented, or EL2 is not enabled in the current Security state, or HCR_EL2.{E2H,TGE} is not {1,1}.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL1.

For all purposes other than returning the result of a direct read of SMCR_EL1, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.

2. If EL2 is implemented and enabled in the current Security state, and the requested length is greater than the Effective length at EL2, then the Effective length at EL2 is used.
3. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.
4. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL1.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access fr EL3 using e mnemonic SMCR_EL1 or SMCR_EL12 are not guaranteed to be ordered with respect accesses ing the er mnemonic.

Accesses to this register use the following encodings in th ystem r ister enc ng space:

MRS <Xt>, SMCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b 00	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && veEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap pr rity when SDD = '1'" && CPTR_EL3.ESM == '0' then
        UNDEFIN ;
    elsif CPA EL1.SM == 'x0' then
        AArch64 st AccessTr p(EL1, 0x1D);
    elsif EL2Enab () && HC EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        Arc 4.Sys mAcce Trap(EL2, 0x1D);
    el if EL2 abled( HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArc 4.SystemA essTrap(EL2, 0x1D);
    sif H && CPTR_EL3.ESM == '0' then
        Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        el
            rch64.SystemAccessTrap(EL3, 0x1D);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x1F0];
    else
        X[t, 64] = SMCR_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    elsif HCR_EL2.E2H == '1' then
```

```
        X[t, 64] = SMCR_EL2;
    else
        X[t, 64] = SMCR_EL1;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SMCR_EL1;
```

MSR SMCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif CPACR_EL1.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x1F0] = X[t, 64];
    else
        SMCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    elsif HCR_EL2.E2H == '1' then
        SMCR_EL2 = X[t, 64];
    else
        SMCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL1 = X[t, 64];
```

MRS <Xt>, SMCR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x1F0];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
            ↪"EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif CPTR_EL2.SMEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            X[t, 64] = SMCR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if CPTR_EL3.ESM == '0' then
            AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            X[t, 64] = SMCR_EL1;
    else
        UNDEFINED;
```

MSR SMCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x1F0] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
            ↪"EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
            UNDEFINED;
        elsif CPTR_EL2.SMEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
        elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
            if Halted() && EDSCR.SDD == '1' then
```

```
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            SMCR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if CPTR_EL3.ESM == '0' then
            AArch64.SystemAccessTrap(EL3, 0x1D);
        else
            SMCR_EL1 = X[t, 64];
    else
        UNDEFINED;
```

E3.1.4 SMCR_EL2, SME Control Register (EL2)

The SMCR_EL2 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at Exception levels EL2, EL1, and EL0.

Configuration

This register has no effect if the PE is not in Streaming SVE mode, or if EL2 is not enabled in the current Security state.

If EL2 is not implemented, this register is RES0 from EL3.

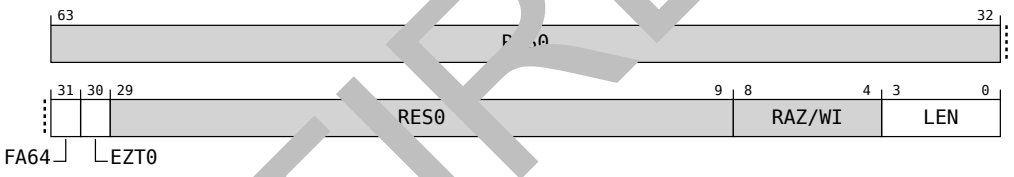
This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMCR_EL2 are UNDEFINED.

Attributes

SMCR_EL2 is a 64-bit register.

Field descriptions

The SMCR_EL2 bit assignments are:



Bits [63:32]

Reserved, RES0.

FA64, bit [31]

When FEAT_SME_FA64 is implemented:

Controls whether execution of an A64 instruction is considered legal when the PE is in Streaming SVE mode.

FA64	Meaning
0b0	This control does not cause any instruction to be treated as legal in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL2, if they are treated as legal at EL3.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at EL2, EL1, and EL0 of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL2, when EL2 is enabled in the current Security state.

The exception is reported using ESR_EL2.EC value 0x1D, with an ISS code of 0x0000004, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.

EZT0	Meaning
0b0	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

Changes to this field only affect whether instructions that access ZT0 are trapped. They do not affect the contents of ZT0, which remain valid so long as PSTATE.ZA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [29:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Streaming SVE vector length (SVL) at EL2 of (LEN+1)*128 bits. This field also defines the Effective Streaming SVE vector length at EL0 when EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H,TGE} is {1,1}.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL2.

For all purposes other than returning the result of a direct read of SMCR_EL2, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
2. If EL3 is implemented and the requested length is greater than the Effective length at EL3, then the Effective length at EL3 is used.

3. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL2.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SMCR_EL2 or SMCR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SMCR_EL2;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SMCR_EL2;
```

MSR SMCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b110

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

```
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL2 = X[t, 64];
```

MRS <Xt>, SMCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif CPACR_EL1.SMEN == '0' then
        AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x1F0];
    else
        X[t, 64] = SMCR_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
```

```
            AArch64.SystemAccessTrap(EL3, 0x1D);
    elsif HCR_EL2.E2H == '1' then
        X[t, 64] = SMCR_EL2;
    else
        X[t, 64] = SMCR_EL1;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SMCR_EL1;
```

MSR SMCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif CPACR_EL1.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    elsif EL2Enabled() && HCR_EL2.<E2H,NV1,NV> == '111' then
        NVMem[0xF0] = X[t, 64];
    else
        SMCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif CPTR_EL3.ESM == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    elsif HCR_EL2.E2H == '1' then
        SMCR_EL2 = X[t, 64];
    else
        SMCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL1 = X[t, 64];
```

E3.1.5 SMCR_EL3, SME Control Register (EL3)

The SMCR_EL3 characteristics are:

Purpose

This register controls aspects of Streaming SVE that are visible at all Exception levels.

Configuration

This register has no effect if the PE is not in Streaming SVE mode.

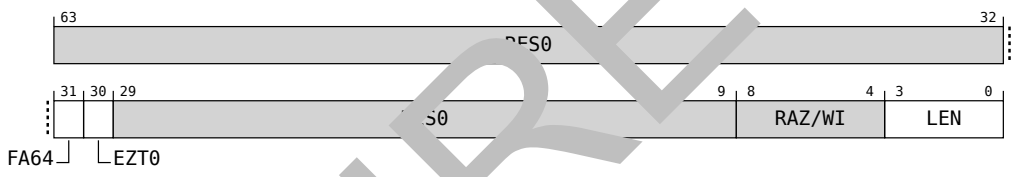
This register is present only when FEAT_SME is implemented and EL3 is implemented. Otherwise, direct accesses to SMCR_EL3 are UNDEFINED.

Attributes

SMCR_EL3 is a 64-bit register.

Field descriptions

The SMCR_EL3 bit assignments are:



Bits [63:32]

Reserved, RES0.

FA64, bit [31]

When FEAT_SME_FA64 is implemented:

Controls whether execution of an A64 instruction is considered legal when the PE is in Streaming SVE mode.

FA64	Meaning
0b0	This control does not cause any instruction to be treated as legal in Streaming SVE mode.
0b1	This control causes all implemented A64 instructions to be treated as legal in Streaming SVE mode at EL3.

Arm recommends that portable SME software should not rely on this optional feature, and that operating systems should provide a means to test for compliance with this recommendation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EZT0, bit [30]

When FEAT_SME2 is implemented:

Traps execution at all Exception levels of the LDR, LUTI2, LUTI4, MOVT, STR, and ZERO instructions that access the ZT0 register to EL3.

The exception is reported using ESR_EL3.EC value 0x1D, with an ISS code of 0x0000004, at a lower priority than a trap due to PSTATE.SM or PSTATE.ZA.

EZT0	Meaning
0b0	This control causes execution of these instructions at all Exception levels to be trapped.
0b1	This control does not cause execution of any instruction to be trapped.

Changes to this field only affect whether instructions that access ZT0 are trapped. They do not affect the contents of ZT0, which remain valid so long as PSTATE.ZA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [29:9]

Reserved, RES0.

Bits [8:4]

Reserved, RAZ/WI.

LEN, bits [3:0]

Requests an Effective Streaming SVE vector length (SVL) at EL3 of (LEN+1)*128 bits.

The Streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive. An implementation can support any subset of the architecturally permitted lengths.

When the PE is in Streaming SVE mode, the Effective SVE vector length (VL) is equal to SVL.

When FEAT_SVE is implemented, and the PE is not in Streaming SVE mode, VL is equal to the Effective Non-streaming SVE vector length. See ZCR_EL3.

For all purposes other than returning the result of a direct read of SMCR_EL3, the PE selects the Effective Streaming SVE vector length by performing checks in the following order:

1. If the requested length is less than the minimum implemented Streaming SVE vector length, then the Effective length is the minimum implemented Streaming SVE vector length.
2. Otherwise, the Effective length is the highest supported Streaming SVE vector length that is less than or equal to the requested length.

An indirect read of SMCR_EL3.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMCR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = SMCR_EL3;
```

MSR SMCR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        SMCR_EL3 = X[t, 64];
```

E3.1.6 SMIDR_EL1, Streaming Mode Identification Register

The SMIDR_EL1 characteristics are:

Purpose

Provides additional identification mechanisms for scheduling purposes, for a PE that supports Streaming SVE mode.

Configuration

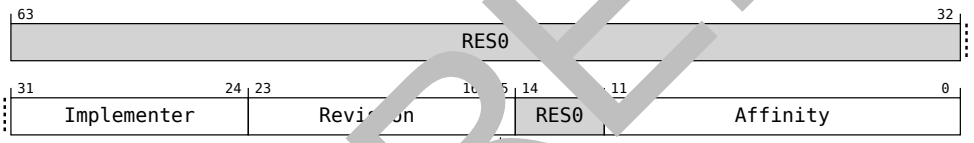
This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMIDR_EL1 are UNDEFINED.

Attributes

SMIDR_EL1 is a 64-bit register.

Field descriptions

The SMIDR_EL1 bit assignments are:



Bits [63:32]

Reserved, RES0.

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm. Assigned codes include the following:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.

Implementer	Meaning
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

It is not required that this value is the same as the value of MIDR_EL1.Implementer.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is **RO**.

Revision, bits [23:16]

Revision number for the Streaming Mode Compute Unit (SMCU).

This field has an IMPLEMENTATION DEFINED value.

Access to this field is **RO**.

SMPS, bit [15]

Indicates support for Streaming SVE mode execution priority.

SMPS	Meaning
0b0	Priority control not supported.
0b1	Priority control supported.

Bits [14:12]

Reserved, RES0.

Affinity, bits [11:0]

The SMCU affinity of the accessing PE.

- A value of zero indicates that the PE's implementation of Streaming SVE mode is not shared with other PEs.
- Otherwise, the value identifies which SMCU is associated with this PE. The Affinity value associated with each SMCU is unique within the system as a whole.

Accessing SMIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = SMIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SMIDR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SMIDR_EL1;
```

E3.1.7 SMPRI_EL1, Streaming Mode Priority Register

The SMPRI_EL1 characteristics are:

Purpose

Configures the streaming execution priority for instructions executed on a shared Streaming Mode Compute Unit (SMCU) when the PE is in Streaming SVE mode at any Exception Level.

Configuration

When SMIDR_EL1.SMPS is '0', this register is RES0.

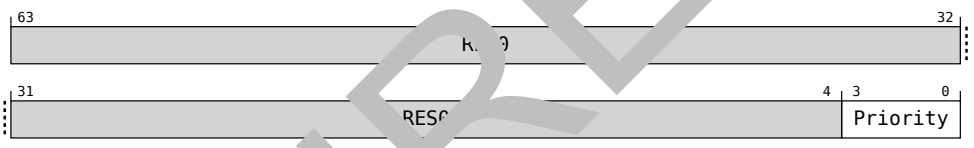
This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMPRI_EL1 are UNDEFINED.

Attributes

SMPRI_EL1 is a 64-bit register.

Field descriptions

The SMPRI_EL1 bit assignments are:



Bits [63:4]

Reserved, RES0.

Priority, bits [3:0]

Streaming execution priority value.

Either this value is used directly, or it is mapped into an effective priority value using SMPRIMAP_EL2.

This value is used directly when any of the following are true:

- The current Exception level is EL3 or EL2.
- The current Exception level is EL1 or EL0, if EL2 is implemented and enabled in the current Security state and HCR_EL2.SMPME is '0'.
- The current Exception level is EL1 or EL0, if EL2 is either not implemented or not enabled in the current Security state.

The precise meaning and behavior of each streaming execution priority value is IMPLEMENTATION DEFINED.

In an implementation that shares execution resources between PEs, higher priority values are allocated more processing resource than other PEs configured with lower priority values in the same Priority domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMPRI_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMPRI_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.nSMPRI_EL1 == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = SMPRI_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = SMPRI_EL1;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = SMPRI_EL1;
```

MSR SMPRI_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.nSMPRI_EL1 == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SMPRI_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
```

```
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SMPRI_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SMPRI_EL1 = X[t, 64];
```

E3.1.8 SMPRIMAP_EL2, Streaming Mode Priority Mapping Register

The SMPRIMAP_EL2 characteristics are:

Purpose

Maps the value in SMPRI_EL1 to a streaming execution priority value for instructions executed at EL1 and EL0 in the same Security states as EL2.

Configuration

When SMIDR_EL1.SMPS is ‘0’, this register is RES0.

If EL2 is not implemented, this register is RES0 from EL3.

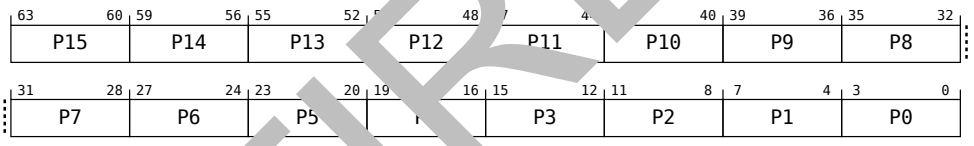
This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SMPRIMAP_EL2 are UNDEFINED.

Attributes

SMPRIMAP_EL2 is a 64-bit register.

Field descriptions

The SMPRIMAP_EL2 bit assignments are:



When all of the following are true, the value in SMPRI_EL1 is mapped to a streaming execution priority using this register:

- The current Exception level is EL1 or EL0.
- EL2 is implemented and enabled in the current Security state.
- HCRX_EL2.SMPME is ‘1’.

Otherwise, SMPRI_EL1 holds the streaming execution priority value.

P15, bits [63:60]

Priority Mapping Entry 15. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1 Priority value is ‘15’.

This value is the highest streaming execution priority.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P14, bits [59:56]

Priority Mapping Entry 14. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘14’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P13, bits [55:52]

Priority Mapping Entry 13. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '13'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P12, bits [51:48]

Priority Mapping Entry 12. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '12'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P11, bits [47:44]

Priority Mapping Entry 11. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '11'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P10, bits [43:40]

Priority Mapping Entry 10. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '10'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P9, bits [39:36]

Priority Mapping Entry 9. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '9'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P8, bits [35:32]

Priority Mapping Entry 8. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '8'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P7, bits [31:28]

Priority Mapping Entry 7. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is '7'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P6, bits [27:24]

Priority Mapping Entry 6. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘6’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P5, bits [23:20]

Priority Mapping Entry 5. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘5’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P4, bits [19:16]

Priority Mapping Entry 4. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘4’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P3, bits [15:12]

Priority Mapping Entry 3. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘3’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P2, bits [11:8]

Priority Mapping Entry 2. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘2’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P1, bits [7:4]

Priority Mapping Entry 1. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘1’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

P0, bits [3:0]

Priority Mapping Entry 0. This entry is used when priority mapping is supported and enabled, and the SMPRI_EL1.Priority value is ‘0’.

This value is the lowest streaming execution priority.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SMPRIMAP_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SMPRIMAP_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x1F8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = SMPRIMAP_EL2;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = SMPRIMAP_EL2;
```

MSR SMPRIMAP_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1F8] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SMPRIMAP_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.ESM == '0' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        SMPRIMAP_EL2 = X[t, 64];
```

E3.1.9 SVCR, Streaming Vector Control Register

The SVCR characteristics are:

Purpose

Controls Streaming SVE mode and SME behavior.

Configuration

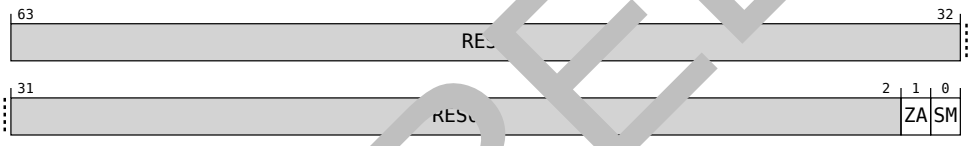
This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to SVCR are UNDEFINED.

Attributes

SVCR is a 64-bit register.

Field descriptions

The SVCR bit assignments are:



Bits [63:2]

Reserved, RES0.

ZA, bit [1]

Enables SME ZA storage. If FEAT_SME2 is implemented, also enables SME2 ZT0 storage.

When this storage is disabled, execution of an instruction which can access it is trapped. The exception is reported using an ESR_ELx.{EC, SME_EC} value of {0x1D, 0x3}.

The possible values of this bit are:

ZA	Meaning
0b0	SME ZA storage and, if implemented, ZT0 storage are invalid and not accessible. This control causes execution at any Exception level of instructions that can access this storage to be trapped.
0b1	SME ZA storage and, if implemented, ZT0 storage are valid and accessible. This control does not cause execution of any instructions to be trapped.

When a write to SVCR.ZA changes the value of PSTATE.ZA from 0 to 1, all implemented bits of the storage are set to zero.

Changes to this field do not have an effect on the SVE vector and predicate registers and FPSR.

A direct or indirect read of ZA appears to occur in program order relative to a direct write of SVCR, and to `MSR SVCRZA` and `MSR SVCRSMZA` instructions, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to `0b0`.

SM, bit [0]

Enables Streaming SVE mode.

When the PE is in Streaming SVE mode, the Streaming SVE vector length (SVL) applies to SVE instructions, and execution at any Exception level of an instruction which is illegal in that mode is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of {0x1D, 0x1}.

When the PE is not in Streaming SVE mode, the SVE vector length (VL) applies to SVE instructions, and execution at any Exception level of an instruction which is only legal in that mode is trapped. The exception is reported using an ESR_ELx.{EC, SMTC} value of {0x1D, 0x2}.

The possible values of this bit are:

SM	Meaning
`0b0`	The PE is not in Streaming SVE mode.
`0b1`	The PE is in Streaming SVE mode.

When a write to SVCR.SM changes the value of PSTATE.SM, the following applies:

- When changed from 0 to 1, an entry to Streaming SVE mode is performed.
- When changed from 1 to 0, an exit from Streaming SVE mode is performed.
- All implemented bits of the SVE registers Z0-Z31, P0-P15, and FFR in the new mode are set to zero.
- FPSR in the new mode is set to 0x0000_0000_0800_009f, in which all cumulative status bits are set to 1.

Changes to this field do not have an effect on SME ZA storage or, if implemented, ZT0 storage.

A direct or indirect read of SM appears to occur in program order relative to a direct write of SVCR, and to `MSR SVCRSM` and `MSR SVCRSMZA` instructions, without the need for explicit synchronization.

The reset behavior of this field is:

- On a Warm reset, this field resets to `0b0`.

Accessing SVCR

SVCR is read/write and can be accessed from any Exception level.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SVCR

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.SMEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x1D);
```

```
        else
            AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.SMEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = Zeros(62):PSTATE.<ZA,SM>;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif CPACR_EL1.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' th
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN ==  then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' th
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1
    else
        X[t, 64] = Zeros(62):PSTA <ZA,
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL && EDS SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority wh SDD == '1 && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap L2, 0x1D);
    elsif HCR_EL2 H == '1' && C EL2.SMEN == 'x0' then
        AArch64 stemAccessTrap(EL2 0x1D);
    elsif Have (EL3) & CPTR_EL3.ESM == '0' then
        if Hal () & EDSCR.SDD == '1' then
            UNDE ;

            rch64. te ccessTrap(EL3, 0x1D);
        se
            X[t Zeros(62):PSTATE.<ZA,SM>;
elsi STA L.EL == 3 then
    if R_EL3.ESM == '0' then
        A ch64.SystemAccessTrap(EL3, 0x1D);
    else
        X[t, 64] = Zeros(62):PSTATE.<ZA,SM>;
```


MSR SVCR, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.SMEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
```

```
            AArch64.SystemAccessTrap(EL2, 0x1D);
        else
            AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.SMEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '0' && CPTR_EL2.TSM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x1D);
    else
        SetPSTATE_SVCR(X[t, 32]);
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boole IMPLEM ATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.ESM  '0' then
        UNDEFINED;
    elsif CPACR_EL1.SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '0' && CP _EL2.TSM ==  en
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' &&  TR_EL .SMEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x1D);
    elsif HaveEL(EL3) && CPTR_EL3.ESM ==  en
        if Halted() && EDSCR.SDD == '1'  en
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL  0 D);
    else
        SetPSTATE_SVCR(X[t, 32 );
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL( 3) && EDSCR DD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority  e DD == '1'"  CPTR_EL3.ESM == '0' then
        UNDEFINED;
    elsif HCR_EL2.E2 == '0' &&  TR_EL2.TSM == '1' then
        AArch64.Sy emAccessTrap( . 0x1D);
    elsif HCR_EL E2H == '1' && CPT EL2.SMEN == 'x0' then
        AArch6 SystemA essTrap(EL2, 0x1D);
    elsif Have (EL3)  & CPTR_EL3.ESM == '0' then
        if Halte  & EDSCR D == '1' then
            UNDEFI ;
        else
            A rch64.Sy mAccessTrap(EL3, 0x1D);
    lse
        S STATE_ R(X[t, 32]);
elsif  TE.EL == EL3 then
    if CP EL3.ESM == '0' then
        AAr 64.SystemAccessTrap(EL3, 0x1D);
    else
        SetPSTATE_SVCR(X[t, 32]);
```


MSR SVCRSM, #<imm>

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b001x	0b011

MSR SVCRZA, #<imm>

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b010x	0b011

MSR SVCRSMZA, #<imm>

op0	op1	CRn	CRm	op2
0b00	0b011	0b0100	0b011x	0b011

E3.1.10 TPIDR2_EL0, EL0 Read/Write Software Thread ID Register 2

The TPIDR2_EL0 characteristics are:

Purpose

Provides a location where SME-aware software executing at EL0 can store thread identifying information, for context management purposes.

The PE makes no use of this register.

Configuration

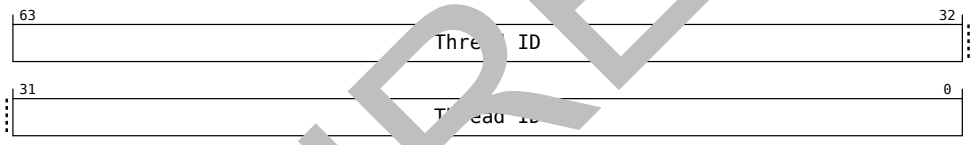
This register is present only when FEAT_SME is implemented. Otherwise, direct accesses to TPIDR2_EL0 are UNDEFINED.

Attributes

TPIDR2_EL0 is a 64-bit register.

Field descriptions

The TPIDR2_EL0 bit assignments are:



Bits [63:0]

Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR2_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TPIDR2_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b101

```
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnTP2 == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnTP2 == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.nTPIDR2_EL0 == '0' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TPIDR2_EL0;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGRTR_EL2.nTPIDR2_EL0 == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TPIDR2_EL0;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = TPIDR2_EL0;
elsif PSTATE.EL == EL3 then
    X[t, 64] = TPIDR2_EL0;
```

MSR TPIDR2_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b101

```
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif !EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && SCTLR_EL1.EnTP2 == '0' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && SCTLR_EL2.EnTP2 == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' && IsFeatureImplemented(FEAT_FGT) &&
        ↪(!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGWTR_EL2.nTPIDR2_EL0 == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TPIDR2_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
```

```
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGWTR_EL2.nTPIDR2_EL0 == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TPIDR2_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.EnTP2 == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.EnTP2 == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        TPIDR2_EL0 = X[t, 64];
elsif PSTATE.EL == EL3 then
    TPIDR2_EL0 = X[t, 64];
```

E3.1.11 EDHSR, External Debug Halting Syndrome Register

The EDHSR characteristics are:

Purpose

Holds syndrome information for a debug event.

Configuration

EDHSR is in the Core power domain

EDHSR is in the Core power domain.

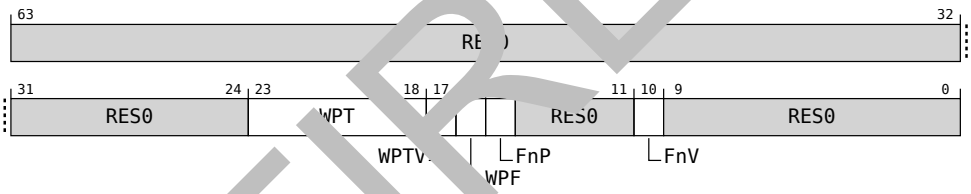
This register is present only when FEAT_Debugv8p2 is implemented and an implementation implements EDHSR. Otherwise, direct accesses to EDHSR are RES0.

Attributes

EDHSR is a 64-bit register.

Field descriptions

The EDHSR bit assignments are:



Bits [63:24]

Reserved, RES0.

WPT, bits [23:1

Watchpoint number. When EDHSR.WPTV is 1, holds the index of a watchpoint that triggered the Watchpoint debug event.

The reset behavior of this field is:

- On a warm reset, this field resets to an architecturally UNKNOWN value.

WPTV, bit [1

Watchpoint number valid.

WPTV	Meaning
0b0	EDHSR.WPT field is not valid, and holds an UNKNOWN value.
0b1	EDHSR.WPT field is valid, and holds the number of a watchpoint that triggered the Watchpoint debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WPF, bit [16]

Watchpoint match might be False.

WPF	Meaning	Applies
0b0	The watchpoint matched the original access or set of contiguous accesses.	
0b1	The watchpoint matched an access or set of contiguous accesses where the lowest accessed address was rounded down to the nearest multiple of 16 bytes and the highest accessed address was rounded up to the nearest multiple of 16 bytes minus 1, but the watchpoint might not have matched the original address of the access or set of contiguous accesses.	When FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FnP, bit [15]

EDWAR not Precise.

FnP	Meaning	Applies
0b0	If the EDWAR is valid, it holds the virtual address of an access or sequence of contiguous accesses that triggered the Watchpoint debug event.	
0b1	If the EDWAR is valid, it holds any virtual address within the smallest implemented translation granule that contains the virtual address of an access or set of contiguous accesses that triggered the Watchpoint debug event.	When FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [14:11]

Reserved, RES0.

FnV, bit [10]

EDWAR not Valid.

FnV	Meaning	Applies
0b0	EDWAR is valid.	
0b1	EDWAR is not valid, and holds an UNKNOWN value.	When FEAT_SME is implemented or FEAT_SVE is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing EDHSR

Accesses to this register use the following encodings in th xtern debug interface:

EDHSR can be accessed through the external debu rface.

Compo	Offset	Instance
Deb	0x038	EDHSR

This interface is accessible follo :

- When DoubleLoc tatus(), or !Isc ePowered() or OSLockStatus() access to this register returns an ERROR.
- Otherwise acc s to this register is R .

E3.2 Changes to existing System registers

System registers that are updated with additional fields, values, or description changes, to support SME functionality.

E3.2.1 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

Purpose

Controls access to trace, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

When EL2 is implemented and enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, the fields in this register have no effect on execution at EL0 and EL1. In this case, the controls provided by CPTR_EL2 are used.

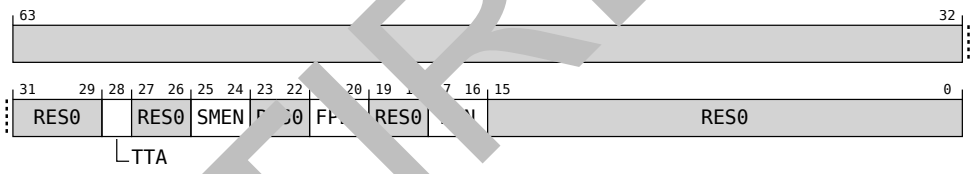
AArch64 system register CPACR_EL1 bits [31:0] are architectura[illegible]pped to AArch32 system register CPACR[31:0].

Attributes

CPACR_EL1 is a 64-bit register.

Field descriptions

The CPACR_EL1 bit assignments are:



Bits [63:29]

Reserved, RES0.

TTA, bit [28]

Traps EL[illegible] and EL1 Sys[illegible]m register accesses to all implemented trace registers from both Execution states to EL1, or to EL2 when [illegible] is imple[illegible]ted and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:

- [illegible]AA[illegible]4 st[illegible]ccesses to trace registers are trapped, reported using ESR_ELx.EC value 0x18.
- In A[illegible]rch32 state, MRC and MCR accesses to trace registers are trapped, reported using ESR_ELx.EC value 0x05.
- In AArch32 state, MRRC and MCRR accesses to trace registers are trapped, reported using ESR_ELx.EC value 0x0C.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	This control causes EL0 and EL1 System register accesses to all implemented trace registers to be trapped.

- The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception

is higher priority than an exception that would be generated because the value of CPACR_EL1.TTA is 1.
- The Arm architecture does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not implemented, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

SMEN, bits [25:24]

When FEAT_SME is implemented:

Traps execution at EL1 and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR or SMCR_EL1 System registers to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

When instructions that directly access the SVCR System registers are trapped with reference to this control, the `MSR SVCRSM`, `MSR SVCRZA`, and `MSR SVCRSMZA` instructions are also trapped.

The exception is reported using ESR_ELx.EC value 0x1D, with an ISS code of 0x0000000.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPACR_EL1.SMEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

SMEN	Meaning
0b00	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b01	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
0b10	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [23:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps execution at EL1 and EL0 of instructions that access the Advanced SIMD and floating-point registers from both Execution states to EL1, reported using ESR_ELx.EC value 0x07, or to EL2 reported using ESR_ELx.EC value 0x00 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to FPCR, FPSR, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.
- In AArch32 state, FPSCR, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers.

Traps execution at EL1 and EL0 of SME and SVE instructions to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using ESR_ELx.EC value 0x07.

A trap taken as a result of CPACR_EL1.SMEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

A trap taken as a result of CPACR_EL1.ZEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

FPEN	Meaning
0b00	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b01	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
0b10	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.FPEN is not 0b11.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:18]

Reserved, RES0.

ZEN, bits [17:16]

When FEAT_SVE is implemented:

Traps execution at EL1 and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL1 System register to EL1, or to EL2 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPACR_EL1.ZEN has precedence over a trap taken as a result of CPACR_EL1.FPEN.

ZEN	Meaning
0b00	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b01	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
0b10	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [15:0]

Reserved, RES0.

Accessing CPACR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CPACR_EL1 or CPACR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
        UNDEFINED;
    elsif EL2Enabled() && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGRTR_EL2.CPACR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```

```
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x100];
    else
        X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        X[t, 64] = CPTR_EL2;
    else
        X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = CPACR_EL1;
```

MSR CPACR_EL1, <Xt>

op0	op1	CRm	op2
0b11	0b0	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 the
    UNDEFINED;
elsif PSTATE.EL == 1 then
    if Halted() & HaveEL(EL3) && CR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap iority hen SDD == '1'" && CPTR_EL3.TCPAC == '1' then
        UNDEF D;
    elsif EL2En led && CPTR EL2.TCPAC == '1' then
        AA ch64.S emAccess ap(EL2, 0x18);
    els EL nable && FeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== ') && R_EL2.CPACR_EL1 == '1' then
        AArc 4.SystemA cessTrap(EL2, 0x18);
    sif && CPTR_EL3.TCPAC == '1' then
        Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
        el
            rch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x100] = X[t, 64];
    else
        CPACR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        CPTR_EL2 = X[t, 64];
    else
        CPACR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
```

```
    CPACR_EL1 = X[t, 64];
```

MRS <Xt>, CPACR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x100];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
            ↪"EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            X[t, 64] = CPACR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = CPACR_EL1;
    else
        UNDEFINED;
```

MSR CPACR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x100] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
            ↪"EL3 trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
            UNDEFINED;
        elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
```

```
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            CPACR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        CPACR_EL1 = X[t, 64];
    else
        UNDEFINED;
```

E3.2.2 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of accesses to CPACR, CPACR_EL1, trace, Activity Monitor, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 system register CPTR_EL2 bits [31:0] are architecturally mapped to AArch32 system register HCPTR[31:0].

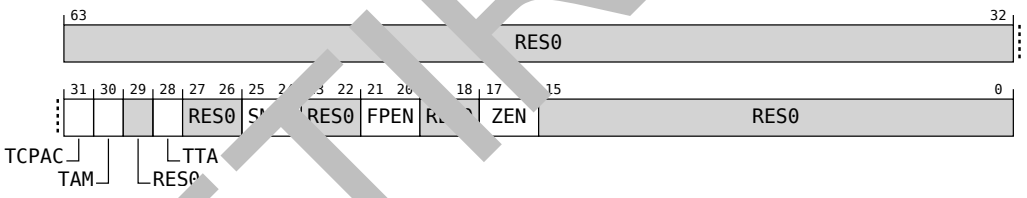
Attributes

CPTR_EL2 is a 64-bit register.

Field descriptions

The CPTR_EL2 bit assignments are:

When FEAT_VHE is implemented and HCR_EL2.E2H == 1:



Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

In AArch64 state, traps accesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x18.

In AArch32 state, traps accesses to CPACR from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x03.

TCPAC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2, when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

CPACR_EL1 and CPACR are not accessible at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitor registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x18:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch64 state, MRC or MCR accesses to the following registers are trapped to EL2 and reported using ESR_ELx.EC value 0x03:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n> and AMEVTYPER1<n>.
- In AArch32 state, MRRC or MCRR accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using ESR_ELx.EC value 0x04.

TAM	Meaning
0b0	Accesses from EL1 and EL0 to Activity Monitor registers are not trapped.
0b1	Accesses from EL1 and EL0 to Activity Monitor registers are trapped to EL2, when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [29]

Reserved, RES0.

TTA, bit [28]

Traps System register accesses to all implemented trace registers from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x05.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt at EL0, EL1 or EL2, to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state, unless HCR_EL2.TGE is 0 and it is trapped by CPACR.NSTRCDIS or CPACR_EL1.TTA. When HCR_EL2.TGE is 1, any attempt at EL0 or EL2 to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state.

The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or ETE, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.

EL2 does not provide traps on trace register accesses through the optional Memory-mapped interface.

System register accesses to the trace registers can have side effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:26]

Reserved, RES0.

SMEN, bits [25:24]

When FEAT_SME is implemented:

Traps execution at EL2, EL1, and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR, SMCR_EL1, or SMCR_EL2 System registers to EL2, when EL2 is enabled in the current Security state.

When instructions that directly access the SVCR System register are trapped with reference to this control, the `MSR SVCRSM`, `MSR SVCRZA`, and `MSR SVCRSMZA` instructions are also trapped.

The exception is reported using ESR_EL2.EC value of 0x1D, with an ISS code of 0x0000000.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL2.SMEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

SMEN	Meaning
0b00	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

SMEN	Meaning
0b01	When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
0b10	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [23:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps execution at EL2, EL1, and EL0 of instructions that access the Advanced SIMD and floating-point registers from both Execution states to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x07.

Traps execution at EL2, EL1, and EL0 of SME and SVE instructions to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x07.

A trap taken as a result of CPTR_EL2.SMEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

A trap taken as a result of CPTR_EL2.ZEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

FPEN	Meaning
0b00	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b01	When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
0b10	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether

these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.FPEN is not 0b11.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:18]

Reserved, RES0.

ZEN, bits [17:16]

When FEAT_SVE is implemented:

Traps execution at EL2, EL1, and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL1 or ZCR_EL2 System registers at EL1, when EL2 is enabled in the current Security state.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPTR_EL2.ZEN has precedence over a trap taken as a result of CPTR_EL2.FPEN.

ZEN	Meaning
0b00	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b01	When HCR_EL2.TGE is 0, this control does not cause execution of any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL2 to be trapped.
0b10	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

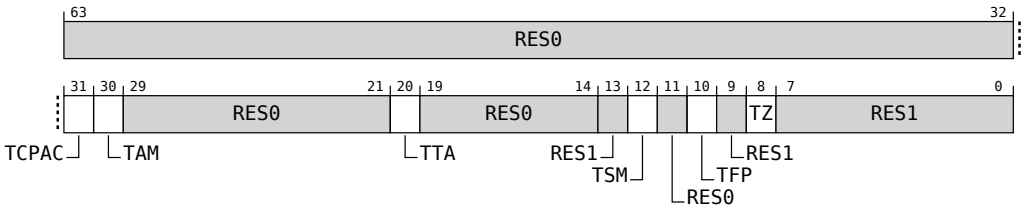
Otherwise:

RES0

Bits [15:0]

Reserved, RES0.

Otherwise:



This format applies in all Armv8.0 implementations.

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

In AArch64 state, traps accesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x18.

In AArch32 state, traps accesses to CPACR from EL1 when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC value 0x03.

TCPAC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the following registers are trapped to EL2, when EL2 is enabled in the current Security state: • CPACR_EL1. • CPACR.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

CPACR_EL1 and CPACR are not accessible at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL1 and EL0 accesses to all Activity Monitor registers to EL2, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x18:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0,

AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.

- In AArch32 state, MCR or MRC accesses to the following registers are trapped to EL2 and reported using ESR_ELx.EC value 0x03:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCLR0, AMCNTENCLR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and AMEVTYPER1<n>.
- In AArch32 state, MCRR or MRRC accesses to AMEVCNTR0<n> and AMEVCNTR1<n>, are trapped to EL2, reported using ESR_ELx.EC value 0x04.

TAM	Meaning
0b0	Accesses from EL1 and EL0 to Activity Monitor registers are not trapped.
0b1	Accesses from EL1 and EL0 to Activity Monitor registers are trapped to EL2, when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [29:21]

Reserved, RES0.

TTA, bit [20]

Traps System register accesses to all implemented trace registers from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x05.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt at EL0, EL1, or EL2, to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state, unless it is trapped by one of the following controls: • CPACR_EL1.TTA. • CPACR.TRCDIS.

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

Bit [13]

Reserved, RES1.

TSM, bit [12]

When FEAT_SME is implemented:

Traps execution at EL2, EL1, and EL0 of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SVCR, SMCR_EL1, or SMCR_EL2 System registers to EL2, when EL2 is enabled in the current Security state.

When instructions that directly access the SVCR System register are trapped with reference to this control, the `MSR SVCRSM`, `MSR SVCRZA`, and `MSR SVCRSMZA` instructions are also trapped.

The exception is reported using ESR_EL2.EC value of 0x1D, with an ISS code of 0x0000000.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL2.TSM has precedence over a trap taken as a result of CPTR_EL2.TFP.

TSM	Meaning
0b0	This control does not cause execution of any instructions to be trapped.
0b1	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

Bit [11]

Reserved, RES0.

TFP, bit [10]

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from both

Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x07:
  - FPCR, FPSR, FPEXC32_EL2, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using ESR_ELx.EC value 0x07:
  - MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. For the purposes of this trap, the architecture defines a VMSR access to FPSID from EL1 or higher as an access to a SIMD and floating-point register. Otherwise, permitted VMSR accesses to FPSID are ignored.

Traps execution at the same Exception levels of SME and SVE instructions to EL2, when EL2 is enabled in the current Security state. The exception is reported using ESR_ELx.EC val[illegible]0x07.

A trap taken as a result of CPTR_EL2.TSM has precedence over a tr[illegible]taken as a res[illegible]of CPTR_EL2.TFP.

A trap taken as a result of CPTR_EL2.TZ has precedence over a [illegible]rap tak[illegible]as a result [illegible]f CPTR_EL2.TFP.

TFP	M[illegible]ning
	This co[illegible]l does not cause execution of any [illegible]ns to be trapped.
0b1	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

FPEXC32_EL2 is not accessibl[illegible]rom EL0 us[illegible]AArch64.

FPSID, MVFR0, MVFR1, [illegible]d FP[illegible]C are not acc[illegible]ssible from EL0 using AArch32.

The reset behavior of t[illegible]s field is:

- On a Warm [illegible]et, this f[illegible]ld resets to an architecturally UNKNOWN value.

Bit [9]

Reser[illegible]d, RES1

TZ, [illegible] [8]

When FE[illegible] SVE is implemented:

Traps execution at EL2, EL1, and EL0 of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL2 or ZCR_EL1 System registers to EL2, when EL2 is enabled in the current Security state.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPTR_EL2.TZ has precedence over a trap taken as a result of CPTR_EL2.TFP.

TZ	Meaning
0b0	This control does not cause execution of any instructions to be trapped.
0b1	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

Bits [7:0]

Reserved, RES1.

Accessing CPTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CPTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = CPTR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = CPTR_EL2;
```

MSR CPTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
```

```
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPTR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    CPTR_EL2 = X[t, 64];
```

MRS <Xt>, CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
        UNDEFINED;
    elsif EL2Enabled() && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGRTR_EL2.CPACR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x100];
    else
        X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        X[t, 64] = CPTR_EL2;
    else
        X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = CPACR_EL1;
```

MSR CPACR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

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

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL3.TCPAC == '1' then
        UNDEFINED;
    elsif EL2Enabled() && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGWTR_EL2.CPACR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x100] = X[t, 64];
    else
        CPACR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolea IMPLEMENT ION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && CPTR_EL CPAC ==  ' then
        UNDEFINED;
    elsif HaveEL(EL3) && CPTR_EL3.TCPAC == '1' t n
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL 0x18,
    elsif HCR_EL2.E2H == '1' then
        CPTR_EL2 = X[t, 64];
    else
        CPACR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    CPACR_EL1 = X[t, 64];
```

E3.2.3 CPTR_EL3, Architectural Feature Trap Register (EL3)

The CPTR_EL3 characteristics are:

Purpose

Controls trapping to EL3 of accesses to CPACR, CPACR_EL1, HCPTR, CPTR_EL2, trace, Activity Monitor, SME, Streaming SVE, SVE, and Advanced SIMD and floating-point functionality.

Configuration

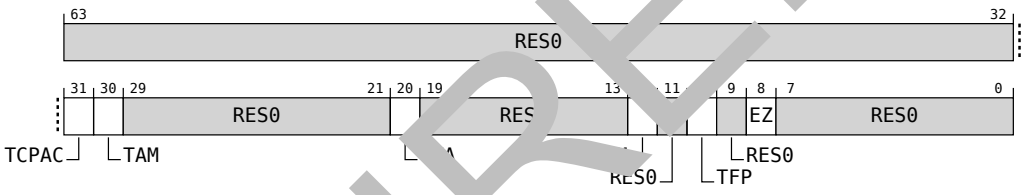
This register is present only when EL3 is implemented. Otherwise, direct accesses to CPTR_EL3 are UNDEFINED.

Attributes

CPTR_EL3 is a 64-bit register.

Field descriptions

The CPTR_EL3 bit assignments are:



Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

Traps all of the following to EL3, from both Execution states and any Security state.

- EL2 accesses to CPTR_EL2 reported using ESR_ELx.EC value 0x18, or HCPTR, reported using ESR_ELx.EC value 0x03.
- EL2 and EL1 accesses to CPACR_EL1 reported using ESR_ELx.EC value 0x18, or CPACR reported using ESR_ELx.EC value 0x03.

When CPTR_EL3.TCPAC is:

TCPAC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are trapped to EL3, unless they are trapped by CPTR_EL2.TCPAC.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TAM, bit [30]

When FEAT_AMUv1 is implemented:

Trap Activity Monitor access. Traps EL2, EL1, and EL0 accesses to all Activity Monitor registers to EL3.

Accesses to the Activity Monitors registers are trapped as follows:

- In AArch64 state, the following registers are trapped to EL3 and reported with ESR_ELx.EC value 0x18:
  - AMUSERENR_EL0, AMCFGR_EL0, AMCGCR_EL0, AMCNTENCLR0_EL0, AMCNTENCLR1_EL0, AMCNTENSET0_EL0, AMCNTENSET1_EL0, AMCR_EL0, AMEVCNTR0<n>_EL0, AMEVCNTR1<n>_EL0, AMEVTYPER0<n>_EL0, and AMEVTYPER1<n>_EL0.
- In AArch32 state, accesses with MRC or MCR to the following regist[illegible]orted with ESR_ELx.EC value 0x03:
  - AMUSERENR, AMCFGR, AMCGCR, AMCNTENCL[illegible] AMCNTEN[illegible]LR1, AMCNTENSET0, AMCNTENSET1, AMCR, AMEVTYPER0<n>, and [illegible]MEV[illegible]PER1<n>.
- In AArch32 state, accesses with MRRC or MCRR to[illegible]e following re[illegible]ter[illegible]eported with ESR_ELx.EC value 0x04:
  - AMEVCNTR0<n>, AMEVCNTR1<n>.

TAM	Me[illegible]ning
[illegible]b0	Accesses from EL2, EL1, and EL0 to Activity Monitor registers are not trapped.
0b1	Accesses from EL2, EL1, and EL0 to Activity Monitor registers are trapped to EL3.

The reset behavior o[illegible]his field is:

- On a Warm reset, thi[illegible]ield resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [[illegible]:21]

Reserved, R[illegible]0.

TTA, bit [20]

Traps System register accesses. Accesses to the trace registers, from all Exception levels, any Security state, and both Execution states are trapped to EL3 as follows:

- In AArch64 state, Trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL3 and reported using EC syndrome value 0x18.
- In AArch32 state, accesses using MCR or MRC to the Trace registers with cpnum=14, opc1=1, and CRn<0b1000 are reported using EC syndrome value 0x05.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.

TTA	Meaning
0b1	Any System register access to the trace registers is trapped to EL3, unless it is trapped by CPACR.TRCDIS, CPACR_EL1.TTA, or CPTR_EL2.TTA.

If System register access to trace functionality is not supported, this bit is RES0.

The ETMv4 architecture and ETE do not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4 or FEAT_ETE, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than this trap exception.

EL3 does not provide traps on trace register accesses through the Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see 'Configurable instruction controls'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

ESM, bit [12]

When FEAT_SME is implemented:

Traps execution of SME instructions, SVE instructions when FEAT_SVE is not implemented or the PE is in Streaming SVE mode, and instructions that directly access the SMCR_EL1, SMCR_EL2, SMCR_EL3, SMPRI_EL1, SMPRIMAP_EL2, or SVCR System registers, from all Exception levels and any Security state, to EL3.

When instructions that directly access the SVCR System register are trapped with reference to this control, the MSR SVCRSM, MSR SVCRZA, and MSR SVCRSMZA instructions are also trapped.

When direct accesses to SMPRI_EL1 and SMPRIMAP_EL2 are trapped, the exception is reported using an ESR_EL3.EC value of 0x1D. Otherwise, the exception is reported using an ESR_EL3.EC value of 0x18. The ISS code is 0x000000.

This field does not affect whether Streaming SVE or SME register values are valid.

A trap taken as a result of CPTR_EL3.ESM has precedence over a trap taken as a result of CPTR_EL3.TFP.

ESM	Meaning
0b0	This control causes execution of these instructions at all Exception levels to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [11]

Reserved, RES0.

TFP, bit [10]

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from all Exception levels, any Security state, and both Execution states, to EL3.

This includes the following registers, all reported using ESR_ELx.EC value 0x07:

- FPCR, FPSR, FPEXC32_EL2, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-S31 registers.
- MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-Q15, including their views as D0-D31 registers or S0-S31 registers.
- VMSR accesses to FPSID.

Permitted VMSR accesses to FPSID are ignored, but for the purposes of this trap the architecture defines a VMSR access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

Traps execution at all Exception levels of SME and SVE instructions to EL3 from any Security state. The exception is reported using ESR_ELx.EC value 0x07.

A trap taken as a result of CPTR_EL3.ESM has precedence over a trap taken as a result of CPTR_EL3.TFP.

A trap taken as a result of CPTR_EL3.EZ has precedence over a trap taken as a result of CPTR_EL3.TFP.

Defined values are:

TFP	Meaning
0b0	This control does not cause execution of any instructions to be trapped.
0b1	This control causes execution of these instructions at all Exception levels to be trapped.

FPEXC32_EL2 is not accessible from EL0 using AArch64.

FPSID, MVFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [9]

Reserved, RES0.

EZ, bit [8]

When FEAT_SVE is implemented:

Traps execution of SVE instructions when the PE is not in Streaming SVE mode, and instructions that directly access the ZCR_EL3, ZCR_EL2, or ZCR_EL1 System registers, from all Exception levels and any Security state, to EL3.

The exception is reported using ESR_ELx.EC value 0x19.

A trap taken as a result of CPTR_EL3.EZ has precedence over a trap taken as a result of CPTR_EL3.TFP.

EZ	Meaning
0b0	This control causes execution of these instructions at all Exception levels to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [7:0]

Reserved, RES0.

Accessing CPTR_EL3

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CPTR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = CPTR_EL3;
```

MSR CPTR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    CPTR_EL3 = X[t, 64];
```

E3.2.4 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL1.

Configuration

AArch64 system register FAR_EL1 bits [31:0] are architecturally mapped to AArch32 system register DFAR31:0.

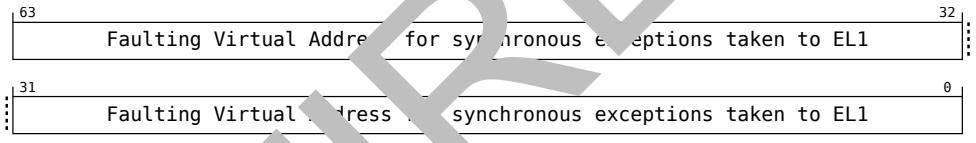
AArch64 system register FAR_EL1 bits [63:32] are architecturally mapped to AArch32 system register IFAR31:0.

Attributes

FAR_EL1 is a 64-bit register.

Field descriptions

The FAR_EL1 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchron exceptions taken to EL1. Exceptions that set the FAR_EL1 are Instruction Aborts (EC 0x20 or 0 21), Data Aborts (0x24 or 0x25), PC alignment faults (EC 0x22), and Watchpoints (EC 0x34 or 0x35). ES _EL1.EC holds the EC value for the exception.

For a synchronou External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI<0|1> == 1 for th translation regime in use when the abort was generated, then the top eight bits of FAR EL1 a UNKN N.

For synchro s External abort other than a synchronous External abort on a translation table walk, this field is valid only i ESR_EL1. nV is 0, and FAR_EL1 is UNKNOWN if ESR_EL1.FnV is 1.

If a memo fault that sets FAR_EL1, other than a Tag Check Fault, is generated from a data cache maintenance or other DC inst tion, this field holds the address specified in the register argument of the instruction.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL1 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If the exception that updates FAR_EL1 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL1.{ISV,FnP} to {0,1} on taking a Data Abort exception, or sets ESR_EL1.{FnV,FnP} to {0,1} on taking a Watchpoint exception, the PE sets FAR_EL1 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception or Watchpoint exception.

The naturally-aligned fault granule is one of:

- When ESR_EL1.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL1.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL1 on a synchronous exception from any of the Memory Copy and Memory Set instructions represents the first element that has not been copied or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^{64} for stage 1, and the IPA range for stage 2. The relevant translation granule is:
  - For MMU faults generated at stage 1, the current stage 1 translation granule.
  - For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  - If FEAT_RME is implemented, for a synchronous Data Abort generated as the result of a GPF, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check failure, the value is the lowest address that failed the Tag Check within the block size of the load or store.
- For a Watchpoint exception, the value is an address range of the size defined by the DCZID_EL0.BS field. This address does not need to be the element with a watchpoint, but can be some earlier element.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort exception or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging'.

For a Synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL0 makes FAR_EL1 become UNKNOWN.

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

For all other exceptions taken to EL1, FAR_EL1 is UNKNOWN.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic FAR_EL1 or FAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, FAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGRTR_EL2.FAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x220];
    else
        X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = FAR_EL2;
    else
        X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = FAR_EL1;
```

MSR FAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGWTR_EL2.FAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x220] = X[t, 64];
    else
        FAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        FAR_EL2 = X[t, 64];
    else
        FAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    FAR_EL1 = X[t, 64];
```

MRS <Xt>, FAR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x220];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = FAR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = FAR_EL1;
    else
        UNDEFINED;
```

MSR FAR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x220] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        FAR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        FAR_EL1 = X[t, 64];
    else
        UNDEFINED;
```

MRS <Xt>, FAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = FAR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = FAR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = FAR_EL2;
```

MSR FAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b01	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        FAR_EL1 = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    FAR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    FAR_EL2 = X[t, 64];
```

E3.2.5 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

AArch64 system register FAR_EL2 bits [31:0] are architecturally mapped to AArch32 system register HDFAR[31:0].

AArch64 system register FAR_EL2 bits [63:32] are architecturally mapped to AArch32 system register HIFAR[31:0].

When EL2 is implemented, AArch64 system register FAR_EL2 bits [31:0] are architecturally mapped to AArch32 system register DFAR31:0.

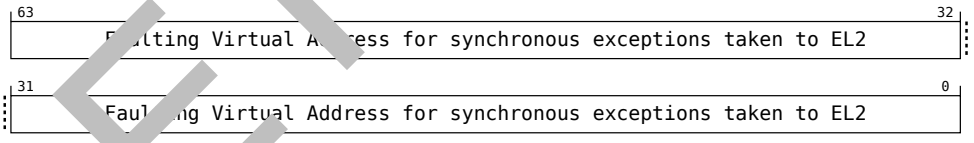
When EL2 is implemented, AArch64 system register FAR_EL2 bits [63:32] are architecturally mapped to AArch32 system register IFAR31:0.

Attributes

FAR_EL2 is a 64-bit register.

Field descriptions

The FAR_EL2 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are Instruction Aborts (EC 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), PC alignment faults (EC 0x22), and Watchpoints (EC 0x34 or 0x35). ESR_EL2.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL2 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL2.FnV is 0, and FAR_EL2 is UNKNOWN if ESR_EL2.FnV is 1.

If a memory fault that sets FAR_EL2, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL2 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If the exception that updates FAR_EL2 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL2.{ISV,FnP} to {0,1} on taking a Data Abort exception, or sets ESR_EL2.{FnV,FnP} to {0,1} on taking a Watchpoint exception, the PE sets FAR_EL2 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception or Watchpoint exception.

The naturally-aligned fault granule is one of:

- When ESR_EL2.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL2.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL2 on a synchronous exception from any of the Memory Copy and Memory Set instructions represents the first element that has not been copied or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^{64} for stage 1, and the PARange for stage 2. The relevant translation granule is:
  - For MMU faults generated at stage 1, the current stage 1 translation granule.
  - For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  - If FEAT_RME is implemented, for a synchronous data abort generated as the result of a GPF, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check failure, the value is the lowest address that failed the Tag Check within the block size of the load or store.
- For a Watchpoint exception, the value is an address range of the size defined by the DCZID_EL0.BS field. This address does not need to be the element with a watchpoint, but can be some earlier element.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort exception or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging'.

For a synchronous Tag Check Fault abort, bits[63:60] are UNKNOWN.

Execution at EL1 or EL0 makes FAR_EL2 become UNKNOWN.

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

For all other exceptions taken to EL2, FAR_EL2 is UNKNOWN.

FAR_EL2 is made UNKNOWN on an exception return from EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic FAR_EL2 or FAR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, FAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' the
        X[t, 64] = FAR_EL1;
    elsif EL2Enabled() && HCR_EL2.NV ==  1' the
        AArch64.SystemAccessTrap(EL2,  x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = FAR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = FAR_EL2;
```

MSR FAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b1	0b100	0b0110	0b0000	0b000

```
if PS  TE  == EL  hen
    UND  NED;
elsif PST  E.EL == EL1 then
    if EL2E  bled() && HCR_EL2.<NV2,NV> == '11' then
        FAR_EL1 = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    FAR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    FAR_EL2 = X[t, 64];
```

MRS <Xt>, FAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGRTR_EL2.FAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x220];
    else
        X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = FAR_EL2;
    else
        X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = FAR_EL1;
```

MSR FAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGWTR_EL2.FAR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x220] = X[t, 64];
    else
        FAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        FAR_EL2 = X[t, 64];
    else
        FAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    FAR_EL1 = X[t, 64];
```

E3.2.6 FAR_EL3, Fault Address Register (EL3)

The FAR_EL3 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions and PC alignment fault exceptions that are taken to EL3.

Configuration

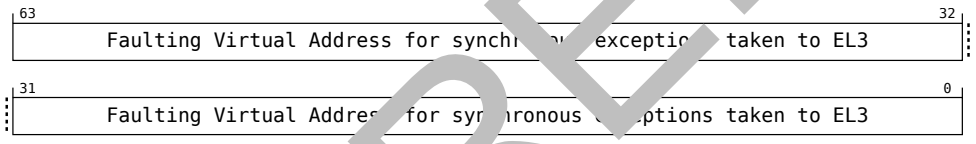
This register is present only when EL3 is implemented. Otherwise, direct accesses to FAR_EL3 are UNDEFINED.

Attributes

FAR_EL3 is a 64-bit register.

Field descriptions

The FAR_EL3 bit assignments are:



Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL3. Exceptions that set the FAR_EL3 are Instruction Aborts (EC 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), and PC alignment faults (EC 0x22). ESR_EL3.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL3 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL3.FnV is 0, and FAR_EL3 is UNKNOWN if ESR_EL3.FnV is 1.

If a memory fault that sets FAR_EL3, other than a Tag Check Fault, is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

On an exception due to a Tag Check Fault caused by a data cache maintenance or other DC instruction, the address held in FAR_EL3 is IMPLEMENTATION DEFINED as one of the following:

- The lowest address that gave rise to the fault.
- The address specified in the register argument of the instruction as generated by MMU faults caused by DC ZVA.

If the exception that updates FAR_EL3 is taken from an Exception level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

When the PE sets ESR_EL3.{ISV,FnP} to {0,1} on taking a Data Abort exception, the PE sets FAR_EL3 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Data Abort exception.

The naturally-aligned fault granule is one of:

- When ESR_EL3.DFSC is 0b010001, indicating a Synchronous Tag Check fault, it is a 16-byte tag granule.
- When ESR_EL3.DFSC is 0b11010x, indicating an IMPLEMENTATION DEFINED fault, it is an IMPLEMENTATION DEFINED granule.
- Otherwise, it is the smallest implemented translation granule.

When FEAT_MOPS is implemented, the value in FAR_EL3 on a synchronous exception from any of the Memory Copy and Memory Set instructions represents the first element that has not been copied or set, and is determined as follows:

- For a Data Abort generated by the MMU, the value is within the address range of the relevant translation granule, aligned to the size of the relevant translation granule of the address that generated the Data Abort. Bits[(n-1):0] of the value are UNKNOWN, where 2^n is the relevant translation granule size in bytes. For the purpose of calculating the relevant translation granule, if the MMU is disabled for a stage of translation, then the current translation granule size is equal to 2^{64} for stage 1, and the PARange for stage 2. The relevant translation granule is:
  - For MMU faults generated at stage 1, the current stage 1 translation granule.
  - For MMU faults generated at stage 2, the smaller of the current stage 1 translation granule and the current stage 2 translation granule.
  - If FEAT_RME is implemented, for a synchronous data abort generated as the result of a GPF, the smallest of the current stage 1 translation granule, the current stage 2 translation granule and the configured granule size in GPCCR_EL3.PGS.
- For a Data Abort generated by a Tag Check fault, the value is the lowest address that failed the Tag Check within the block size of the load or store.
- Otherwise, the value is the lowest address in the block size of the load or store.

For a Data Abort exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging'.

For a synchronous Tag Check fault abort, bits[60:56] are UNKNOWN.

Execution at EL2, EL1, or EL0 makes FAR_EL3 become UNKNOWN.

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lowest address that gave rise to the fault that is reported. When different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

For all other exceptions taken to EL3, FAR_EL3 is UNKNOWN.

FAR_EL3 is made UNKNOWN on an exception return from EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL3

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, FAR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

```
1 if PSTATE.EL == EL0 then
```

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = FAR_EL3;
```

MSR FAR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    FAR_EL3 = X[t, 64];
```

E3.2.7 FPCR, Floating-point Control Register

The FPCR characteristics are:

Purpose

Controls floating-point behavior.

Configuration

It is IMPLEMENTATION DEFINED whether the Len and Stride fields can be programmed to nonzero values, which will cause some AArch32 floating-point instruction encodings to be UNDEFINED, or whether these fields are RAZ.

AArch64 system register FPCR bits [26:15] are architecturally mapped to AArch32 system register FPSCR[26:15].

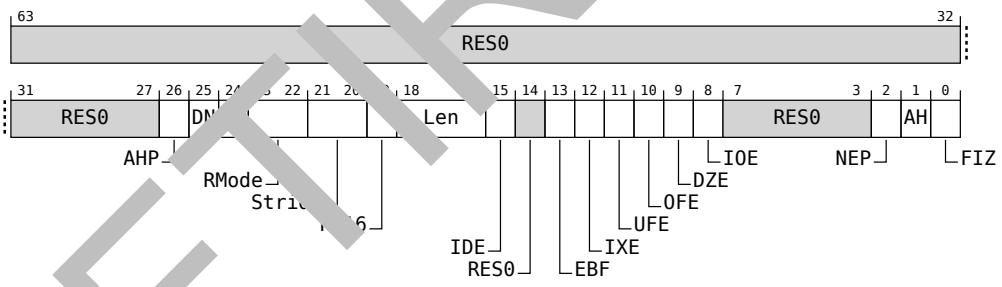
AArch64 system register FPCR bits [12:8] are architecturally mapped to AArch32 system register FPSCR[12:8].

Attributes

FPCR is a 64-bit register.

Field descriptions

The FPCR bit assignments are:



Bits [63:27]

Reserved, RES0.

AHP, bit [26]

Alternative half-precision control bit.

AHP	Meaning
0b0	IEEE half-precision format selected.
0b1	Alternative half-precision format selected.

This bit is used only for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the FEAT_FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DN, bit [25]

Default NaN use for NaN propagation.

DN	Meaning
0b0	NaN operands propagate through to the output of a floating-point operation.
0b1	Any operation involving one or more NaNs returns the Default NaN. This bit has no effect on the output of FABS, FMAX*, FMIN*, and FNEG instructions, and a default N[illegible]er returned as a result of these instru[illegible]ons.

The value of this bit controls both scalar and Advanced SIMD [illegible]loating-poi[illegible]rithm[illegible]c.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally [illegible]NOWN val[illegible].

FZ, bit [24]

Flushing denormalized numbers to zero contro[illegible]it.

FZ	Meaning
0b0	If FPCR.AH is 0, the flushing to zero of single-precision and double-precision denormalized inputs to, and outputs of, floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero. If FPCR.AH is 1, the flushing to zero of single-precision and double-precision denormalized outputs of floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero.
0b1	If FPCR.AH is 0, denormalized single-precision and double-precision inputs to, and outputs from, floating-point instructions are flushed to zero. If FPCR.AH is 1, denormalized single-precision and double-precision outputs from floating-point instructions are flushed to zero.

For more information, see ‘Flushing denormalized numbers to zero’ and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field.

RMode	Meaning
0b00	Round to Nearest (RN) mode.
0b01	Round towards Plus Infinity (RP) mode.
0b10	Round towards Minus Infinity (RM) mode.
0b11	Round towards Zero (RZ) mode.

The specified rounding mode is used by both scalar and Advanced SIMD floating-point instructions.

If FPCR.AH is 1, then the following instructions use Round to Nearest mode regardless of the value of this bit:

- The FRECPE, FRECPS, FRECPX, FRSQRTE, and FRSQRTS instructions.
- The BFCVT, BFCVTN, BFCVTN2, BFCVTNT, BFMLALB, and BFMLALT instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

This field has no function in AArch64 state and non-zero values are ignored during execution in AArch64 state.

This field is included only for context saving and restoration of the AArch32 FPSCR.Stride field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FZ16, bit [19]

When FEAT_FP16 is implemented:

Flushing denormalized numbers to zero control bit on half-precision data-processing instructions.

FZ16	Meaning
0b0	For some instructions, this bit disables flushing to zero of inputs and outputs that are half-precision denormalized numbers.
0b1	Flushing denormalized numbers to zero enabled. For some instructions that do not convert a half-precision input to a higher precision output, this bit enables flushing to zero of inputs and outputs that are half-precision denormalized numbers.

The value of this bit applies to both scalar and Advanced SIMD floating-point half-precision calculations.

For more information, see ‘Flushing denormalized numbers to zero’ and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Len, bits [18:16]

This field has no function in AArch64 state, and nonzero values are ignored during execution in AArch64 state.

This field is included only for context saving and restoration of the AArch32 FPSCR.Len field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDE, bit [15]

Input Denormal floating-point exception trap enable.

IDE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IDC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IDC bit.

When the PE is in Streaming SVE mode and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IDE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [14]

Reserved, RES0.

EBF, bit [13]

When FEAT_EBF16 is implemented:

The value of this bit controls the numeric behaviors of BFloat16 dot product calculations performed by the BFDOT, BFMMLA, BFMOPA, and BFMOPS instructions. If FEAT_SME2 is implemented, this also controls BFVDOT instruction.

When ID_AA64ISAR1_EL1.BF16 and ID_AA64ZFR0_EL1.BF16 are 0b0010, the PE supports the FPCR.EBF field. Otherwise, FPCR.EBF is RES0.

EBF	Meaning
0b0	These instructions use the standard BFloat16 behaviors: • Ignoring the FPCR.RMode control and using the rounding mode defined for BFloat16. For more information, see 'Round to Odd mode'. • Flushing denormalized inputs and outputs to zero, as if the FPCR.FZ and FPCR.FIZ controls had the value '1'. • Performing unfused multiplies and additions with intermediate rounding of all products and sums.
0b1	These instructions use the extended BFloat16 behaviors: • Supporting all four IEEE 754 rounding modes selected by the FPCR.RMode control. • Optionally, flushing denormalized inputs and outputs to zero, as governed by the FPCR.FZ and FPCR.FIZ controls. • Performing a fused two-way sum-of-products for each pair of adjacent BFloat16 elements, without intermediate rounding of the products, but rounding the single-precision sum before addition to the accumulator. • Generating the default NaN as intermediate sum-of-products when any multiplier input is a NaN, or any product is infinity × 0.0, or there are infinite products with differing signs. • Generating an intermediate sum-of-products of the same infinity when there are infinite products all with the same sign.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

IXE, bit [12]

Inexact floating-point exception trap enable.

IXE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IXC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IXC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IXE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFE, bit [11]

Underflow floating-point exception trap enable.

UFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.UFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs and Flush-to-zero is not enabled, the PE does not update the FPSR.UFC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.UFE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFE, bit [10]

Overflow floating-point exception trap enable.

OFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.OFC bit is set to 1.

OFE	Meaning
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.OFC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.OFE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZE, bit [9]

Divide by Zero floating-point exception trap enable.

DZE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.DZC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.DZC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.DZE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOE, bit [8]

Invalid Operation floating-point exception trap enable.

IOE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IOC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IOC bit.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.IOE is treated as 0 for all purposes other than a direct read or write of the FPCR.

The Effective value of this bit controls both scalar and vector floating-point arithmetic.

If the implementation does not support this exception, this bit is RAZ/WI.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [7:3]

Reserved, RES0.

NEP, bit [2]

When FEAT_AFP is implemented:

Controls how the output elements other than the lowest element of the vector are determined for Advanced SIMD scalar instructions.

NEP	Meaning
0b0	Does not affect how the output elements other than the lowest are determined for Advanced SIMD scalar instructions.

NEP	Meaning
0b1	The output elements other than the lowest are taken from the following registers: • For 3-input scalar versions of the FMLA (by element) and FMLS (by element) instructions, the <Hd>, <Sd>, or <Dd> register. • For 3-input versions of the FMADD, FMSUB, FNMADD, and FNMSUB instructions, the <Ha>, <Sa>, or <Da> register. • For 2-input scalar versions of the FACGE, FACGT, FCMEQ (register), FCMGE (register), and FCMGT (register) instructions, the <Hm>, <Sm>, or <Dm> register. • For 2-input scalar versions of the FABD, FADD (scalar), FDIV (scalar), FMAX (scalar), FMAXNM (scalar), FMIN (scalar), FMINNM (scalar), FMUL (by element), FMUL (scalar), FMULX (by element), FMULX, FNMUL (scalar), FRECPS, FRSQRTS, and FSUB (scalar) instructions, the <Hn>, <Sn>, or <Dn> register. • For 1-input scalar versions of the following instructions, the <Hd>, <Sd>, or <Dd> register: – The (vector) versions of the FCVTAS, FCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, and FCVTPU instructions. – The (vector, fixed-point) and (vector, integer) versions of the FCVTZS, FCVTZU, SCVTF, and UCVTF instructions. – The (scalar) versions of the FABS, FNEG, FRINT32X, FRINT32Z, FRINT64X, FRINT64Z, FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ, and FSQRT instructions. – The (scalar, fixed-point) and (scalar, integer) versions of the SCVTF and UCVTF instructions. – The BFCVT, FCVT, FCVTXN, FRECPE, FRECPX, and FRSQRTE instructions.

When the PE is in Streaming SVE mode, and FEAT_SME_FA64 is not implemented or not enabled, the value of FPCR.NEP is treated as 0 for all purposes other than a direct read or write of the FPCR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

AH, bit [1]

When FEAT_AFP is implemented:

Alternate Handling. Controls alternate handling of floating-point numbers.

The Arm architecture supports two models for handling some of the corner cases of the floating-point behaviors, such as the nature of flushing of denormalized numbers, the detection of tini[illegible]nd other exceptions and a range of other behaviors. The value of the FPCR.AH bit selects between these[illegible]odels.

For more information on the FPCR.AH bit, see 'Flushing deno[illegible]alized numbe[illegible]to zero', Floating- point exceptions and exception traps and the pseudocode of the floati[illegible]point [illegible]tructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architectura[illegible]UNKN[illegible]WN value.

Otherwise:

RES0

FIZ, bit [0]

When FEAT_AFP is implemente[illegible]

Flush Inputs to Zero. Control[illegible]ether single-p[illegible]ision, double-precision and BFloat16 input operands that are denormalized numbers are [illegible]ushed [illegible]zero.

FIZ	Meaning
0b0	The flushing to zero of single-precision and double-precision denormalized inputs to floating-point instructions not enabled by this control, but other factors might cause the input denormalized numbers to be flushed to zero.
0b1	Denormalized single-precision and double-precision inputs to most floating-point instructions flushed to zero.

For more information, see 'Flushing denormalized numbers to zero' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Accessing FPCR

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, FPCR

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

```
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.FPEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPCR;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif CPACR_EL1.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPCR;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPCR;
elsif PSTATE.EL == EL3 then
```

```
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        X[t, 64] = FPCR;
```

MSR FPCR, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

```
if PSTATE.EL == EL0 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPTR_EL1.FPEN != '11' then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif EL2Enabled() && HCR_EL2.<E2H,TGE> == '1 && CPTR_EL2.FPEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif EL2Enabled() && HCR_EL2.E2H == CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0 7);
    elsif EL2Enabled() && HCR_EL2.E2H = '1' & CPTR_EL2.FP == '1' then
        AArch64.SystemAccessTrap(EL2, 07);
    elsif HaveEL(EL3) && CPTR_EL3 TFP = then
        if Halted() && EDSCR.SDD = '1' t
            UNDEFINED;
        else
            AArch64.Syste essTrap(EL 0x07);
    else
        FPCR = X[t, 64];
elsif PSTATE.EL == 1 then
    if Halted() & aveEL(EL3) && CR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap iority hen SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEF D;
    elsif CPACR 1. EN == 'x0' then
        AArch64.S emAccess ap(EL1, 0x07);
    els EL nable && R_EL2.E2H != '1' && CPTR_EL2.TFP == '1' then
        AArch .Syste ssTrap(EL2, 0x07);
    sif EL nabled() & HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AA mAccessTrap(EL2, 0x07);
    el aveEL(EL3) && CPTR_EL3.TFP == '1' then
        Halted() && EDSCR.SDD == '1' then
        UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t, 64];
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'" && CPTR_EL3.TFP == '1' then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.FPEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif HaveEL(EL3) && CPTR_EL3.TFP == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t, 64];
```

```
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.TFP == '1' then
        AArch64.SystemAccessTrap(EL3, 0x07);
    else
        FPCR = X[t, 64];
```

E3.2.8 HCRX_EL2, Extended Hypervisor Configuration Register

The HCRX_EL2 characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

The bits in this register behave as if they are 0 for all purposes other than direct reads of the register if:

- EL2 is not enabled in the current Security state.
- SCR_EL3.HXEn is 0.

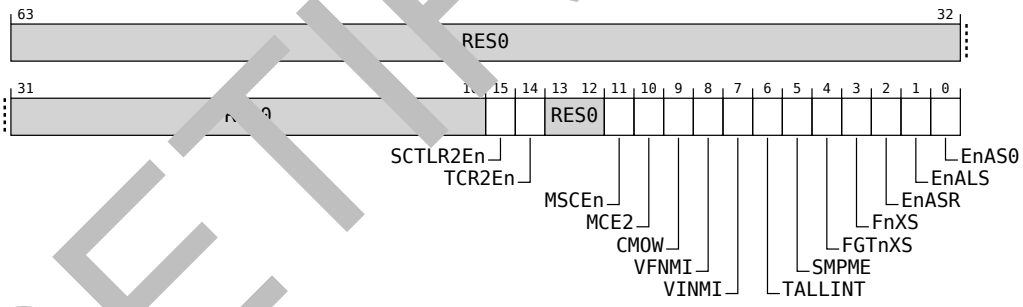
This register is present only when FEAT_HCX is implemented. Otherwise, direct accesses to HCRX_EL2 are UNDEFINED.

Attributes

HCRX_EL2 is a 64-bit register.

Field descriptions

The HCRX_EL2 bit assignments are:



Bits [63:16]

Reserved, RES0.

SCTLR2En, bit [15]

When FEAT_SCTLR2 is implemented:

SCTLR2_EL1 Enable. In AArch64 state, accesses to SCTLR2_EL1 are trapped to EL2 and reported using EC syndrome value 0x18.

SCTLR2En	Meaning
0b0	Accesses to SCTLR2_EL1 at EL1 are trapped to EL2, unless the access generates a higher priority exception. The value in SCTLR2_EL1 is treated as 0.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TCR2En, bit [14]

When FEAT_TCR2 is implemented:

TCR2_EL1 Enable. In AArch64 state, accesses to TCR2_EL1 are trapped to EL2 and reported using EC syndrome value 0x18.

TCR2En	Meaning
0b0	Accesses to TCR2_EL1 at EL1 are trapped to EL2, unless this access generates a higher priority exception. The value in TCR2_EL1 is treated as UNKNOWN.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [13:12]

Reserved, RES0

MSCEn, bit [11]

When FEAT_MOPS is implemented:

Memory Set and Memory Copy instructions Enable. Enables execution of the CPY*, SETG*, SETP*, SETM*, and SETE* instructions at EL1 or EL0.

MSCEn	Meaning
0b0	Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL1 or EL0.
0b1	This control does not cause any instructions to be UNDEFINED.

This bit behaves as if it is 1 if any of the following are true:

- EL2 is not implemented or enabled.

- The value of HCR_EL2.{E2H, TGE} is {1, 1}.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to `0b0`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

MCE2, bit [10]

When FEAT_MOPS is implemented:

Controls Memory Copy and Memory Set exceptions generated as part of attempting to execute the Memory Copy and Memory Set instructions from EL1.

MCE2	Meaning
`0b0`	Memory Copy and Memory Set exceptions generated from EL1 are taken to EL1.
`0b1`	Memory Copy and Memory Set exceptions generated from EL1 are taken to EL2.

When the value of HCR_EL2.{E2H, TGE} is {1, 1}, this control does not affect any exceptions due to the higher priority SCTLR_EL2.MSCEn control.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to `0b0`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

CMOW, bit [9]

When FEAT_CMOW is implemented:

Controls cache maintenance instruction permission for the following instructions executed at EL1 or EL0.

- IC IVAU, DC CIVAC, DC CIGDVAC and DC CIGVAC.
- ICIMVAU, DCCIMVAC.

CMOW	Meaning
`0b0`	These instructions executed at EL1 or EL0 with stage 2 read permission, but without stage 2 write permission do not generate a stage 2 permission fault.

CMOW	Meaning
0b1	These instructions executed at EL1 or EL0, if enabled as a result of SCTLR_EL1.UCI==1, with stage 2 read permission, but without stage 2 write permission generate a stage 2 permission fault.

For this control, stage 2 has write permission if S2AP[1] is 1 or DBM is 1 in the stage 2 descriptor. The instructions do not cause an update to the dirty state.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

VFNMI, bit [8]

When FEAT_NMI is implemented:

Virtual FIQ Interrupt with Superpriority. Enables signaling of virtual FIQ interrupts with Superpriority.

VFNMI	Meaning
0b0	When HCR_EL2.VF is 1, a signaled pending virtual FIQ interrupt does not have Superpriority.
0b1	When HCR_EL2.VF is 1, a signaled pending virtual FIQ interrupt has Superpriority.

When HCR_EL2.VF is 0, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

VINMI, bit [7]

When FEAT_NMI is implemented:

Virtual IRQ Interrupt with Superpriority. Enables signaling of virtual IRQ interrupts with Superpriority.

VINMI	Meaning
0b0	When HCR_EL2.VI is 1, a signaled pending virtual IRQ interrupt does not have Superpriority.
0b1	When HCR_EL2.VI is 1, a signaled pending virtual IRQ interrupt has Superpriority.

When HCR_EL2.VI is 0, this bit has no effect.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TALLINT, bit [6]

When FEAT_NMI is implemented:

Traps the following writes at EL1 using AArch64 to EL2 when EL3 is implemented and enabled:

- MSR (register) writes of ALLINT.
- MSR (immediate) writes of ALLINT with a value of 1.

TALLINT	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified MSR accesses at EL1 using AArch64 are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SMPME, bit [5]

When FEAT_SME is implemented:

Streaming Mode Priority Mapping Enable.

Controls mapping of the value of SMPRI_EL1.Priority for streaming execution priority at EL0 or EL1.

SMPME	Meaning
0b0	The effective priority value is taken from SMPRI_EL1.Priority.

SMPME	Meaning
0b1	The effective priority value is: • When the current Exception level is EL2 or EL3, the value of SMPRI_EL1.Priority. • When the current Exception level is EL0 or EL1, the value of the SMPRIMAP_EL2 field corresponding to the value of SMPRI_EL1.Priority.

When SMIDR_EL1.SMPS is '0', this field is RES0.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

FGTnXS, bit [4]

When FEAT_XS is implemented:

Determines if the fine-grained trap in HFGITR_EL2 that apply to each of the TLBI maintenance instructions that are accessible at EL1 also apply to the corresponding TLBI maintenance instructions with the nXS qualifier.

FGTnXS	Meaning
0b0	The fine-grained trap in the HFGITR_EL2 that applies to a TLBI maintenance instruction at EL1 also applies to the corresponding TLBI instruction with the nXS qualifier at EL1.
0b1	The fine-grained trap in the HFGITR_EL2 that applies to a TLBI maintenance instruction at EL1 does not apply to the corresponding TLBI instruction with the nXS qualifier at EL1.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

FnXS, bit [3]

When FEAT_XS is implemented:

Determines the behavior of TLBI instructions affected by the XS attribute.

This control bit also determines whether an AArch64 DSB instruction behaves as a DSB instruction with an nXS qualifier when executed at EL0 and EL1.

FnXS	Meaning
0b0	This control does not have any effect on the behavior of the TLBI maintenance instructions.
0b1	A TLBI maintenance instruction without the nXS qualifier executed at EL1 behaves in the same way as the corresponding TLBI maintenance instruction with the nXS qualifier. An AArch64 DSB instruction executed at EL1 or EL0 behaves in the same way as the corresponding DSB instruction with the nXS qualifier executed at EL1 or EL0.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnASR, bit [2]

When FEAT_LS64_V implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an ST64BV instruction at EL0 or EL1 to EL2.

EnASR	Meaning
0b0	Execution of an ST64BV instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnASR. Execution of an ST64BV instruction at EL1 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000000.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnALS, bit [1]

When FEAT_LS64 is implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an LD64B or ST64B instruction at EL0 or EL1 to EL2.

EnALS	Meaning
0b0	Execution of an LD64B or ST64B instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnALS. Execution of an LD64B or ST64B instruction at EL1 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnAS0, bit [0]

When FEAT_LS64_ACCDATA is implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an ST64BV0 instruction at EL0 or EL1 to EL2.

EnAS0	Meaning
0b0	Execution of an ST64BV0 instruction at EL0 is trapped to EL2 if the execution is not trapped by SCTLR_EL1.EnAS0. Execution of an ST64BV0 instruction at EL1 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Accessing HCRX_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, HCRX_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0xA0];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.HXEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = HCRX_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = HCRX_EL2;
```

MSR HCRX_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0xA0] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.HXEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        HCRX_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
```

```
21      HCRX_EL2 = X[t, 64];
```

E3.2.9 HFGRTR_EL2, Hypervisor Fine-Grained Read Trap Register

The HFGRTR_EL2 characteristics are:

Purpose

Provides controls for traps of MRS and MRC reads of System registers.

Configuration

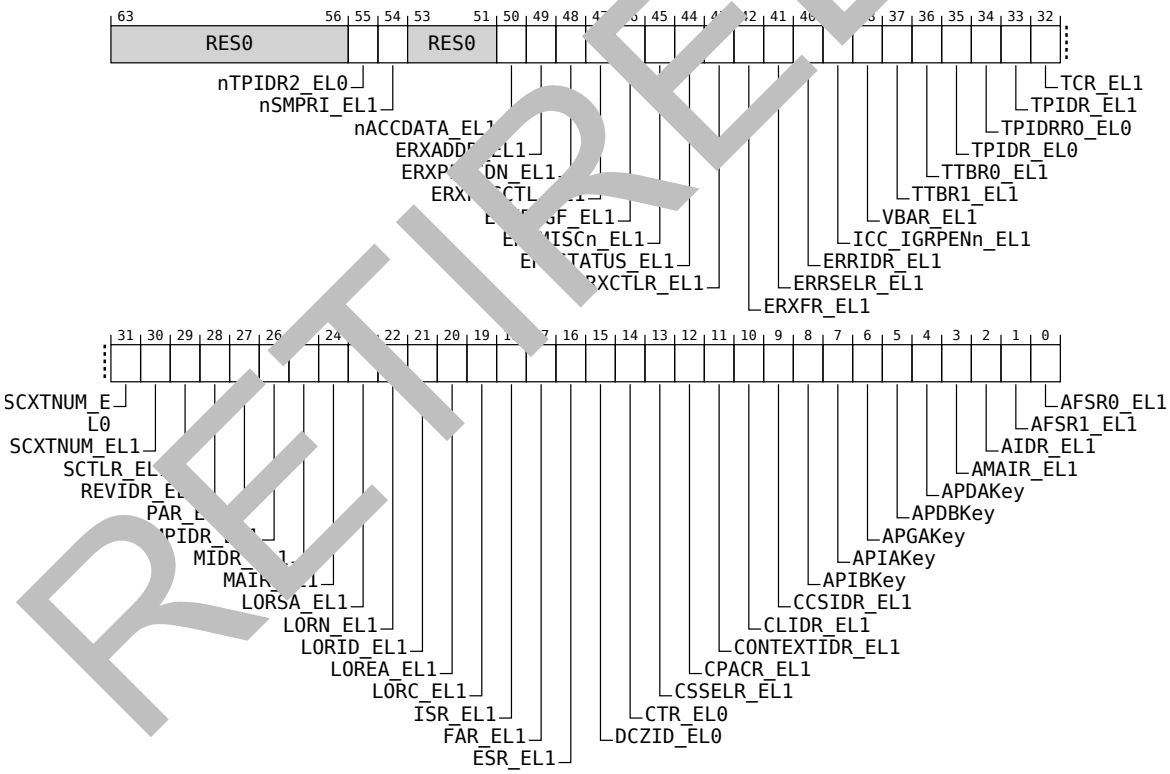
This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HFGRTR_EL2 are UNDEFINED.

Attributes

HFGRTR_EL2 is a 64-bit register.

Field descriptions

The HFGRTR_EL2 bit assignments are:



Bits [63:56]

Reserved, RES0.

nTPIDR2_EL0, bit [55]

When FEAT_SME is implemented:

Trap MRS reads of TPIDR2_EL0 at EL1 and EL0 using AArch64 to EL2.

nTPIDR2_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TPIDR2_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of TPIDR2_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

nSMPRI_EL1, bit [54]

When FEAT_SME is implemented:

Trap MRS reads of SMPRI_EL1 at EL1 using AArch64 to EL2.

nSMPRI_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SMPRI_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of SMPRI_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

Bits [53:51]

Reserved, RES0.

nACCDATA_EL1, bit [50]

When FEAT_LS64_ACCDATA is implemented:

Trap MRS reads of ACCDATA_EL1 at EL1 using AArch64 to EL2.

nACCDATA_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ACCDATA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.
0b1	MRS reads of ACCDATA_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

ERXADDR_EL1, bit [49]

When FEAT_RAS is implemented:

Trap MRS reads of ERXADDR_EL1 at EL1 us[illegible] AArch64 to EL2.

ERXADDR_EL1	Meaning
0b0	MRS reads of ERXADDR_EL1 are not trapped by this mechanism.
[illegible]	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing th[illegible]ield has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXPFGCDN_EL1, bit [48]

When FEAT_RASv1p1 is implemented:

Trap MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

ERXPFGCDN_EL1	Meaning
0b0	MRS reads of ERXPFGCDN_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGCDN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXPFGCTL_EL1, bit [47]

When FEAT_RASv1p1 is implemented:

Trap MRS reads of ERXPFGCTL_EL1 at EL1 using AArch64 to EL2.

ERXPFGCTL_EL1	Meaning
0b0	MRS reads of ERXPFGCTL_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:

- ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
- ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXPFGF_EL1, bit [46]

When FEAT_RAS is implemented:

Trap MRS reads of ERXPFGF_EL1 at EL1 using AArch64 to EL2.

ERXPFGF_EL1	Meaning
0b0	MRS reads of ERXPFGF_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXPFGF_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXMISCn_EL1, bit [45]

When FEAT_RAS is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- ERXMISC0_EL1.
- ERXMISC1_EL1.
- ERXMISC2_EL1.
- ERXMISC3_EL1.

ERXMISCn_EL1	Meaning
0b0	MRS reads of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true.
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXSTATUS_EL1, bit [44]

When FEAT_RAS is implemented:

Trap MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.

ERXSTATUS_EL1	Meaning
0b0	MRS reads of ERXSTATUS_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.

– Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXCTLR_EL1, bit [43]

When FEAT_RAS is implemented:

Trap MRS reads of ERXCTLR_EL1 at EL1 using AArch64 to EL2.

ERXCTLR_EL1	Meaning
0b0	MRS reads of ERXCTLR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and the ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXFR_EL1, bit [42]

When FEAT_RAS is implemented:

Trap MRS reads of ERXFR_EL1 at EL1 using AArch64 to EL2.

ERXFR_EL1	Meaning
0b0	MRS reads of ERXFR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERXFR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERRSELR_EL1, bit [41]

When FEAT_RAS is implemented:

Trap MRS reads of ERRSELR_EL1 at EL1 using AArch64 to EL2.

ERRSELR_EL1	Meaning
0b0	MRS reads of ERRSELR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERRIDR_EL1, bit [40]

When FEAT_RAS is implemented:

Trap MRS reads of ERRIDR_EL1 at EL1 using AArch64 to EL2.

ERRIDR_EL1	Meaning
0b0	MRS reads of ERRIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ERRIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true.
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ICC_IGRPENn_EL1, bit [39]

When FEAT_GICv3 is implemented.

Trap MRS reads of ICC_IGRPEN<n>_EL1 to EL1 using AArch64 to EL2.

ICC_IGRPENn_EL1	Meaning
0b0	MRS reads of ICC_IGRPEN<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

VBAR_EL1, bit [38]

Trap MRS reads of VBAR_EL1 at EL1 using AArch64 to EL2.

VBAR_EL1	Meaning
0b0	MRS reads of VBAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TTBR1_EL1, bit [37]

Trap MRS reads of TTBR1_EL1 at EL1 using AArch64 to EL2.

TTBR1_EL1	Meaning
0b0	MRS reads of TTBR1_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TTBR0_EL1, bit [36]

Trap MRS reads of TTBR0_EL1 at EL1 using AArch64 to EL2.

TTBR0_EL1	Meaning
0b0	MRS reads of TTBR0_EL1 are not trapped by this mechanism.

TTBR0_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TPIDR_EL0, bit [35]

Trap MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURW at EL0 using AArch32 when EL1 is using AArch64 to EL2.

TPIDR_EL0	Meaning
0b0	MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURW at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception: • MRS reads of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of TPIDRURW at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TPIDRRO_EL0, bit [34]

Trap MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 when EL1 is using AArch64 to EL2.

TPIDRRO_EL0	Meaning
0b0	MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 and MRC reads of TPIDRURO at EL0 using AArch32 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the read generates a higher priority exception: • MRS reads of TPIDRRO_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MRC reads of TPIDRURO at EL0 using AArch32 are trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TPIDR_EL1, bit [33]

Trap MRS reads of TPIDR_EL1 at EL1 using AArch64 to EL2.

TPIDR_EL1	Meaning
0b0	MRS reads of TPIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TCR_EL1, bit [32]

Trap MRS reads of any of the following registers at EL1 using AArch64 to EL2.

- TCR_EL1.
- TCR2_EL1, if FEAT_TCR2 is implemented.

TCR_EL1	Meaning
0b0	MRS reads of the specified registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of the specified registers at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

SCXTNUM_EL0, bit [31]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MRS reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.

SCXTNUM_EL0	Meaning
0b0	MRS reads of SCXTNUM_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

SCXTNUM_EL1, bit [30]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MRS reads of SCXTNUM_EL1 at EL1 using AArch64 to EL2.

SCXTNUM_EL1	Meaning
0b0	MRS reads of SCXTNUM_EL1 are not trapped by this mechanism.

SCXTNUM_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

SCTLR_EL1, bit [29]

Trap MRS reads of any of the following registers at EL1 using AArch64 to EL2.

- SCTLR_EL1.
- SCTLR2_EL1, if FEAT_SCTLR2 is implemented.

SCTLR_EL1	Meaning
0b0	MRS reads of the specified registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of the specified registers at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

REVIDR_EL1, bit [28]

Trap MRS reads of REVIDR_EL1 at EL1 using AArch64 to EL2.

REVIDR_EL1	Meaning
0b0	MRS reads of REVIDR_EL1 are not trapped by this mechanism.

REVIDR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of REVIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

PAR_EL1, bit [27]

Trap MRS reads of PAR_EL1 at EL1 using AArch64 to EL2.

PAR_EL1	Meaning
0b0	MRS reads of PAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

MPIDR_EL1, bit [26]

Trap MRS reads of MPIDR_EL1 at EL1 using AArch64 to EL2.

MPIDR_EL1	Meaning
0b0	MRS reads of MPIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

MIDR_EL1, bit [25]

Trap MRS reads of MIDR_EL1 at EL1 using AArch64 to EL2.

MIDR_EL1	Meaning
0b0	MRS reads of MIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

MAIR_EL1, bit [24]

Trap MRS reads of MAIR_EL1 at EL1 using AArch64 to EL2.

MAIR_EL1	Meaning
0b0	MRS reads of MAIR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

LORSA_EL1, bit [23]

When FEAT_LOR is implemented:

Trap MRS reads of LORSA_EL1 at EL1 using AArch64 to EL2.

LORSA_EL1	Meaning
0b0	MRS reads of LORSA_EL1 are not trapped by this mechanism.

LORSA_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

LORN_EL1, bit [22]

When FEAT_LOR is implemented:

Trap MRS reads of LORN_EL1 at EL1 using AA n64 to L2.

LO N_EL1	Meaning
0b0	MRS reads of LORN_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The re t b avior o field is:

- On a arm reset, this field resets to 0b0.

Otherwise:

RES0

LORID_EL1, bit [21]

When FEAT_LOR is implemented:

Trap MRS reads of LORID_EL1 at EL1 using AArch64 to EL2.

LORID_EL1	Meaning
0b0	MRS reads of LORID_EL1 are not trapped by this mechanism.

LORID_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORID_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

LOREA_EL1, bit [20]

When FEAT_LOR is implemented:

Trap MRS reads of LOREA_EL1 at EL1 using AArch64 to EL2.

LOREA_EL1	Meaning
0b0	MRS reads of LOREA_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

LORC_EL1, bit [19]

When FEAT_LOR is implemented:

Trap MRS reads of LORC_EL1 at EL1 using AArch64 to EL2.

LORC_EL1	Meaning
0b0	MRS reads of LORC_EL1 are not trapped by this mechanism.

LORC_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

ISR_EL1, bit [18]

Trap MRS reads of ISR_EL1 at EL1 using AArch64 to EL2.

ISR_EL1	Meaning
	MRS reads of ISR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ISR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

FAR_EL1, bit [17]

Trap MRS reads of FAR_EL1 at EL1 using AArch64 to EL2.

FAR_EL1	Meaning
0b0	MRS reads of FAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

ESR_EL1, bit [16]

Trap MRS reads of ESR_EL1 at EL1 using AArch64 to EL2.

ESR_EL1	Meaning
0b0	MRS reads of ESR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of ESR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b

DCZID_EL0, bit [15]

Trap MRS reads of DCZID_EL0 at EL1 and EL0 using AArch64 to EL2.

DCZID_EL0	Meaning
0b0	MRS reads of DCZID_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of DCZID_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

CTR_EL0, bit [14]

Trap MRS reads of CTR_EL0 at EL1 and EL0 using AArch64 to EL2.

CTR_EL0	Meaning
0b0	MRS reads of CTR_EL0 are not trapped by this mechanism.

CTR_EL0	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CTR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

CSSELR_EL1, bit [13]

Trap MRS reads of CSSELR_EL1 at EL1 using AArch64 to EL2.

CSSELR_EL1	Meaning
0b0	MRS reads of CSSELR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

CPACR_EL1, bit [12]

Trap MRS reads of CPACR_EL1 at EL1 using AArch64 to EL2.

CPACR_EL1	Meaning
0b0	MRS reads of CPACR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

CONTEXTIDR_EL1, bit [11]

Trap MRS reads of CONTEXTIDR_EL1 at EL1 using AArch64 to EL2.

CONTEXTIDR_EL1	Meaning
0b0	MRS reads of CONTEXTIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CONTEXTIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b

CLIDR_EL1, bit [10]

Trap MRS reads of CLIDR_EL1 EL1 using rch64 EL2.

CLIDR_EL1	Meaning
0b0	MRS reads of CLIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CLIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

CCSIDR_EL1, bit [9]

Trap MRS reads of CCSIDR_EL1 at EL1 using AArch64 to EL2.

CCSIDR_EL1	Meaning
0b0	MRS reads of CCSIDR_EL1 are not trapped by this mechanism.

CCSIDR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of CCSIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

APIBKey, bit [8]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

APIBKey	Meaning
0b0	MRS reads of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

APIAKey, bit [7]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

APIAKey	Meaning
0b0	MRS reads of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

APGAKey, bit [6]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

APGAKey	Meaning
0b0	MRS reads of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

APDBKey, bit [5]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

APDBKey	Meaning
0b0	MRS reads of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

APDAKey, bit [4]

When FEAT_PAuth is implemented:

Trap MRS reads of multiple System registers. Enables a trap on MRS reads at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDAKeyHi_EL1.
- APDAKeyLo_EL1.

APDAKey	Meaning
0b0	MRS reads of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

AMAIR_EL1, bit [3]

Trap MRS reads of AMAIR_EL1 at EL1 using AArch64 to EL2.

AMAIR_EL1	Meaning
0b0	MRS reads of AMAIR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

AIDR_EL1, bit [2]

Trap MRS reads of AIDR_EL1 at EL1 using AArch64 to EL

AIDR_EL1	Meaning
0b0	MRS reads of AIDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

AFSR1_EL1, bit [1]

Trap MRS reads of AFSR1_EL1 at EL1 using AArch64 to EL2.

AFSR1_EL1	Meaning
0b0	MRS reads of AFSR1_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

AFSR0_EL1, bit [0]

Trap MRS reads of AFSR0_EL1 at EL1 using AArch64 to EL2.

AFSR0_EL1	Meaning
0b0	MRS reads of AFSR0_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MRS reads of AFSR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the read generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b

Accessing HFGRTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, HFGRTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x1B8];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = HFGRTR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = HFGRTR_EL2;
```

MSR HFGRTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        NVMem[0x1B8] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolea IMPLEMENT ION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3 GTEn ==  then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' t
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL 0x18
    else
        HFGRTR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    HFGRTR_EL2 = X[t, 64];
```

E3.2.10 HFGWTR_EL2, Hypervisor Fine-Grained Write Trap Register

The HFGWTR_EL2 characteristics are:

Purpose

Provides controls for traps of MSR and MCR writes of System registers.

Configuration

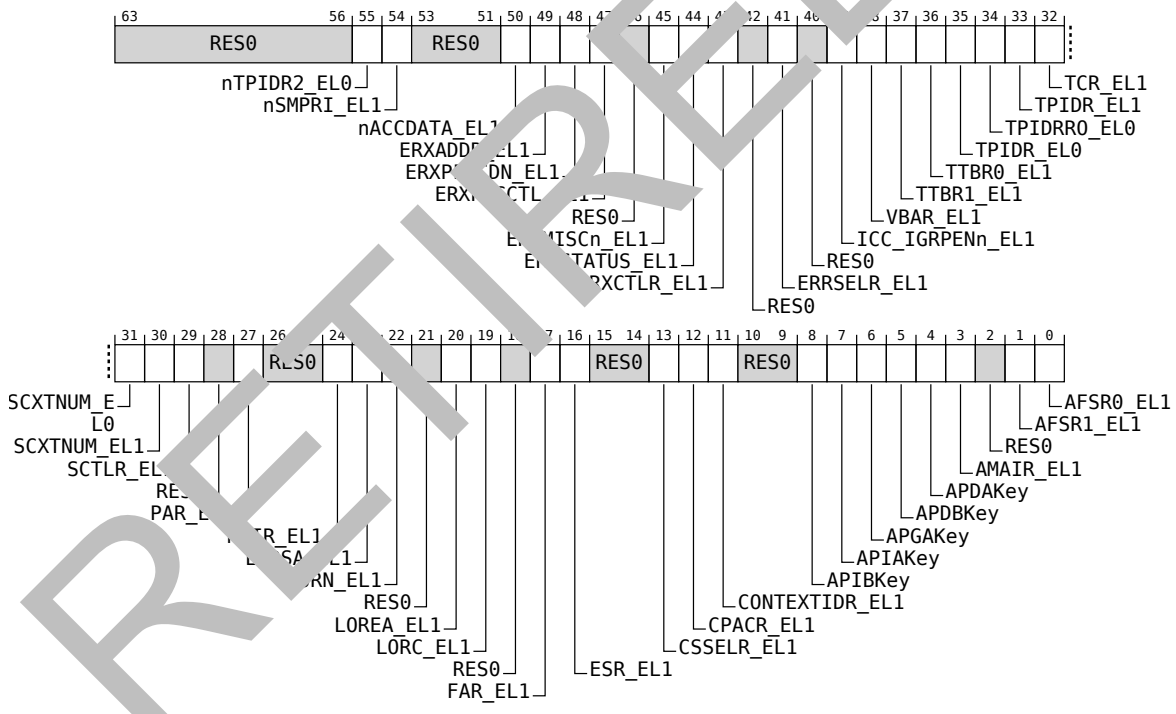
This register is present only when FEAT_FGT is implemented. Otherwise, direct accesses to HFGWTR_EL2 are UNDEFINED.

Attributes

HFGWTR_EL2 is a 64-bit register.

Field descriptions

The HFGWTR_EL2 bit assignments are:



Bits [63:56]

Reserved, RES0.

nTPIDR2_EL0, bit [55]

When FEAT_SME is implemented:

Trap MSR writes of TPIDR2_EL0 at EL1 and EL0 using AArch64 to EL2.

nTPIDR2_EL0	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDR2_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of TPIDR2_EL0 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

nSMPRI_EL1, bit [54]

When FEAT_SME is implemented:

Trap MSR writes of SMPRI_EL1 at EL1 using AArch64 to EL2.

nSMPRI_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SMPRI_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of SMPRI_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

Bits [53:51]

Reserved, RES0.

nACCDATA_EL1, bit [50]

When FEAT_LS64_ACCDATA is implemented:

Trap MSR writes of ACCDATA_EL1 at EL1 using AArch64 to EL2.

nACCDATA_EL1	Meaning
0b0	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ACCDATA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.
0b1	MSR writes of ACCDATA_EL1 are not trapped by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

ERXADDR_EL1, bit [49]

When FEAT_RAS is implemented:

Trap MSR writes of ERXADDR_EL1 at EL1 using AArch64 to EL2.

ERXADDR_EL1	Meaning
0b0	MSR writes of ERXADDR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXADDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXPFGCDN_EL1, bit [48]

When FEAT_RASv1p1 is implemented:

Trap MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 to EL2.

ERXPFGCDN_EL1	Meaning
0b0	MSR writes of ERXPFGCDN_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXPFGCDN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXPFGCTL_EL1, bit [47]

When FEAT_RASv1p1 is implemented:

Trap MSR writes of ERXPFGCTL_EL1 at EL1 using AArch64 to EL2.

ERXPFGCTL_EL1	Meaning
0b0	MSR writes of ERXPFGCTL_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXPFGCTL_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:

- ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
- ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [46]

Reserved, RES0.

ERXMISCn_EL1, bit [45]

When FEAT_RAS is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- ERXMISC0_EL1.
- ERXMISC1_EL1.
- ERXMISC2_EL1.
- ERXMISC3_EL1.

ERXMISCn_EL1	Meaning
0b0	MSR writes of the specified System registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the specified System registers are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXSTATUS_EL1, bit [44]

When FEAT_RAS is implemented:

Trap MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 to EL2.

ERXSTATUS_EL1	Meaning
0b0	MSR writes of ERXSTATUS_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXSTATUS_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to an architecturally UNKNOWN value.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ERXCTLR_EL1, bit [40]

When FEAT_RAS is implemented:

Trap MSR writes of ERXCTLR_EL1 at EL1 using AArch64 to EL2.

ERXCTLR_EL1	Meaning
0b0	MSR writes of ERXCTLR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERXCTLR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [42]

Reserved, RES0.

ERRSELR_EL1, bit [41]

When FEAT_RAS is implemented:

Trap MSR writes of ERRSELR_EL1 at EL1 using AArch64 to EL2.

ERRSELR_EL1	Meaning
0b0	MSR writes of ERRSELR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ERRSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is 0.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [40]

Reserved, RES0.

ICC_IGRPENn_EL1, bit [39]

When FEAT_GICv3 is implemented:

Trap MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 to EL2.

ICC_IGRPENn_EL1	Meaning
0b0	MSR writes of ICC_IGRPEN<n>_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ICC_IGRPEN<n>_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

VBAR_EL1, bit [38]

Trap MSR writes of VBAR_EL1 at EL1 using AArch64 to EL2.

VBAR_EL1	Meaning
0b0	MSR writes of VBAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of VBAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TTBR1_EL1, bit [37]

Trap MSR writes of TTBR1_EL1 at EL1 using AArch64 to EL2.

TTBR1_EL1	Meaning
0b0	MSR writes of TTBR1_EL1 are not trapped by this mechanism.

TTBR1_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TTBR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TTBR0_EL1, bit [36]

Trap MSR writes of TTBR0_EL1 at EL1 using AArch64 to EL

TTBR0_EL1	eaning
0b0	s of TTBR0_EL1 are not trapped by this echanism.
0	EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TTBR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavio f thi ield is:

- On reset, is fiel esets to 0b0.

TP R_EL0, [35]

Trap MSR rites of TPIDR_EL0 at EL1 and EL0 using AArch64 and MCR writes of TPIDRURW at EL0 using AArch32 w n EL1 is using AArch64 to EL2.

TPIDR_EL0	Meaning
0b0	MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 and MCR writes of TPIDRURW at EL0 using AArch32 are not trapped by this mechanism.

TPIDR_EL0	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, EL1 is using AArch64, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then, unless the write generates a higher priority exception: • MSR writes of TPIDR_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18. • MCR [illegible] of TPIDRURW at EL0 using AArch32 [illegible] trapped to EL2 and reported with EC syndrome value 0x03.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TPIDRRO_EL0, bit [34]

Trap MSR writes of TPIDRRO_EL0 at EL1 using AArch64 to EL2.

TPIDRRO_EL0	Meaning
0b0	MSR writes of TPIDRRO_EL0 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDRRO_EL0 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TPIDR_EL1, bit [33]

Trap MSR writes of TPIDR_EL1 at EL1 using AArch64 to EL2.

TPIDR_EL1	Meaning
0b0	MSR writes of TPIDR_EL1 are not trapped by this mechanism.

TPIDR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of TPIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

TCR_EL1, bit [32]

Trap MSR writes of any of the following registers at EL1 using AArch64 to EL2.

- TCR_EL1.
- TCR2_EL1, if FEAT_TCR2 is implemented.

TCR_EL1	Meaning
0b0	MSR writes of the specified registers are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of the specified registers at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

SCXTNUM_EL0, bit [31]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 to EL2.

SCXTNUM_EL0	Meaning
0b0	MSR writes of SCXTNUM_EL0 are not trapped by this mechanism.

SCXTNUM_EL0	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} != {1, 1}, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCXTNUM_EL0 at EL1 and EL0 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

SCXTNUM_EL1, bit [30]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap MSR writes of SCXTNUM_EL1 at EL1 using AArch64 to EL2.

SCXTNUM_EL1	Meaning
0b0	MSR writes of SCXTNUM_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of SCXTNUM_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

SCTLR_EL1, bit [29]

Trap MSR writes of any of the following registers at EL1 using AArch64 to EL2.

- SCTLR_EL1.
- SCTLR2_EL1, if FEAT_SCTLR2 is implemented.

SCTLR_EL1	Meaning
0b0	MSR writes of the specified registers are not trapped by this mechanism.

SCTLR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of the specified registers at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Bit [28]

Reserved, RES0.

PAR_EL1, bit [27]

Trap MSR writes of PAR_EL1 at EL1 using AArch64 to EL2.

PAR_EL1	Meaning
0	MSR writes of PAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of PAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Bits [26:25]

Reserved, RES0.

MAIR_EL1, bit [24]

Trap MSR writes of MAIR_EL1 at EL1 using AArch64 to EL2.

MAIR_EL1	Meaning
0b0	MSR writes of MAIR_EL1 are not trapped by this mechanism.

MAIR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of MAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

LORSA_EL1, bit [23]

When FEAT_LOR is implemented:

Trap MSR writes of LORSA_EL1 at EL1 using AArch64 to EL2.

LORSA_EL1	Meaning
0b0	MSR writes of LORSA_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORSA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

LORN_EL1, bit [22]

When FEAT_LOR is implemented:

Trap MSR writes of LORN_EL1 at EL1 using AArch64 to EL2.

LORN_EL1	Meaning
0b0	MSR writes of LORN_EL1 are not trapped by this mechanism.

LORN_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORN_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

Bit [21]

Reserved, RES0.

LOREA_EL1, bit [20]

When FEAT_LOR is implemented:

Trap MSR writes of LOREA_EL1 at EL1 using AArch64 to EL2.

LOREA_EL1	Meaning
0b0	MSR writes of LOREA_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LOREA_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

LORC_EL1, bit [19]

When FEAT_LOR is implemented:

Trap MSR writes of LORC_EL1 at EL1 using AArch64 to EL2.

LORC_EL1	Meaning
0b0	MSR writes of LORC_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of LORC_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

Bit [18]

Reserved, RES0.

FAR_EL1, bit [17]

Trap MSR writes of FAR_EL1 at EL1 using AArch64 to EL2.

FAR_EL1	Meaning
0b0	MSR writes of FAR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of FAR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

ESR_EL1, bit [16]

Trap MSR writes of ESR_EL1 at EL1 using AArch64 to EL2.

ESR_EL1	Meaning
0b0	MSR writes of ESR_EL1 are not trapped by this mechanism.

ESR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of ESR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Bits [15:14]

Reserved, RES0.

CSSELR_EL1, bit [13]

Trap MSR writes of CSSELR_EL1 at EL1 using AA[illegible]4 to E[illegible].

[illegible]SELR[illegible]1	Meaning
0b0	MSR writes of CSSELR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CSSELR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

CPACR_EL1, bit [12]

Trap MSR writes of CPACR_EL1 at EL1 using AArch64 to EL2.

CPACR_EL1	Meaning
0b0	MSR writes of CPACR_EL1 are not trapped by this mechanism.

CPACR_EL1	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CPACR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

CONTEXTIDR_EL1, bit [11]

Trap MSR writes of CONTEXTIDR_EL1 at EL1 using AArch[illegible] to EL2.

CONTEXTIDR_EL1	[illegible]eaning
0b0	[illegible] writes of CONTEXTIDR_EL1 are not trapped by this mechanism.
0b1	[illegible] EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of CONTEXTIDR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavio[illegible]f thi[illegible]ield is:

- On [illegible] reset, [illegible]is field[illegible]esets to 0b0.

Bit[illegible]10:9]

Reserved, RES0.

APIBKey, b[illegible]8]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIBKeyHi_EL1.
- APIBKeyLo_EL1.

APIBKey	Meaning
0b0	MSR writes of the System registers listed above are not trapped by this mechanism.

APIBKey	Meaning
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

APIAKey, bit [7]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APIAKeyHi_EL1.
- APIAKeyLo_EL1.

APIAKey	Meaning
0b0	MSR writes of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

APGAKey, bit [6]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APGAKeyHi_EL1.
- APGAKeyLo_EL1.

APGAKey	Meaning
0b0	MSR writes of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

APDBKey, bit [5]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDBKeyHi_EL1.
- APDBKeyLo_EL1.

APDBKey	Meaning
0b0	MSR writes of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

APDAKey, bit [4]

When FEAT_PAuth is implemented:

Trap MSR writes of multiple System registers. Enables a trap on MSR writes at EL1 using AArch64 of any of the following AArch64 System registers to EL2:

- APDAKeyHi_EL1.
- APDAKeyLo_EL1.

APDAKey	Meaning
0b0	MSR writes of the System registers listed above are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes at EL1 using AArch64 of any of the System registers listed above are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

AMAIR_EL1, bit [3]

Trap MSR writes of AMAIR_EL1 at EL1 using AArch64 to EL2.

AMAIR_EL1	Meaning
0b0	MSR writes of AMAIR_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AMAIR_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Bit [2]

Reserved, RES0.

AFSR1_EL1, bit [1]

Trap MSR writes of AFSR1_EL1 at EL1 using AArch64 to EL2.

AFSR1_EL1	Meaning
0b0	MSR writes of AFSR1_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AFSR1_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

AFSR0_EL1, bit [0]

Trap MSR writes of AFSR0_EL1 at EL1 using AArch64 to EL2.

AFSR0_EL1	Meaning
0b0	MSR writes of AFSR0_EL1 are not trapped by this mechanism.
0b1	If EL2 is implemented and enabled in the current Security state, and either EL3 is not implemented or SCR_EL3.FGTEn == 1, then MSR writes of AFSR0_EL1 at EL1 using AArch64 are trapped to EL2 and reported with EC syndrome value 0x18, unless the write generates a higher priority exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Accessing HFGWTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, HFGWTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV> == '11' then
        X[t, 64] = NVMem[0x1C0];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = HFGWTR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = HFGWTR_EL2;
```

MSR HFGWTR_EL2, <Xt>

op0	op1	CR	CRm	op2
0b11	0b100	0001	0b0001	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<N2,NV> == '11' then
        NVMem[0x1C0] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3
        ↪trap priority when SDD == '1'" && SCR_EL3.FGTEn == '0' then
        UNDEFINED;
    elsif HaveEL(EL3) && SCR_EL3.FGTEn == '0' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        HFGWTR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    HFGWTR_EL2 = X[t, 64];
```

E3.2.11 ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1

The ID_AA64ISAR1_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

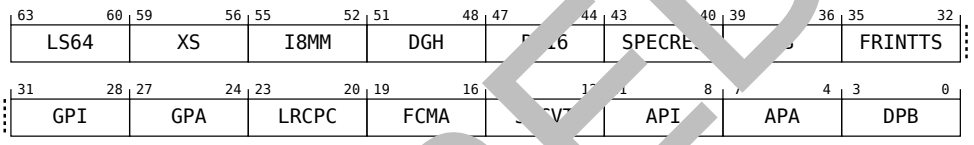
For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers .

Attributes

ID_AA64ISAR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ISAR1_EL1 bit assignments are:



LS64, bits [63:60]

Indicates support for LD64B and ST64B instructions, and the ACCDATA_EL1 register. Defined values of this field are:

LS64	Meaning
0b0000	The LD64B, ST64B, ST64BV, and ST64BV0 instructions, the ACCDATA_EL1 register, and associated traps are not supported.
0b0001	The LD64B and ST64B instructions are supported.
0b0010	The LD64B, ST64B, and ST64BV instructions, and their associated traps are supported.
0b0011	The LD64B, ST64B, ST64BV, and ST64BV0 instructions, the ACCDATA_EL1 register, and their associated traps are supported.

All other values are reserved.

FEAT_LS64 implements the functionality identified by 0b0001.

FEAT_LS64_V implements the functionality identified by 0b0010.

FEAT_LS64_ACCDATA implements the functionality identified by 0b0011.

From Armv8.7, the permitted values are 0b0000, 0b0001, 0b0010, and 0b0011.

XS, bits [59:56]

Indicates support for the XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields in AArch64 state. Defined values are:

XS	Meaning
0b0000	The XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields are not supported.
0b0001	The XS attribute, the TLBI and DSB instructions with the nXS qualifier, and the HCRX_EL2.{FGTnXS, FnXS} fields are supported.

All other values are reserved.

FEAT_XS implements the functionality identified by 0b0001.

From Armv8.7, the only permitted value is 0b0001.

I8MM, bits [55:52]

Indicates support for Advanced SIMD and Floating-point Int8 matrix multiplication instructions in AArch64 state. Defined values are:

I8MM	Meaning
0b0000	Int8 matrix multiplication instructions are not implemented.
0b0001	SMMLA, SUDOT, UMMLA, USMMLA, and USDOT instructions are implemented.

All other values are reserved.

FEAT_I8MM implements the functionality identified by 0b0001.

When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ZFR0_EL1.I8MM.

From Armv8.6, the only permitted value is 0b0001.

DGH, bits [51:48]

Indicates support for the Data Gathering Hint instruction. Defined values are:

DGH	Meaning
0b0000	Data Gathering Hint is not implemented.
0b0001	Data Gathering Hint is implemented.

All other values are reserved.

FEAT_DGH implements the functionality identified by 0b0001.

From Armv8.0, the permitted values are 0b0000 and 0b0001.

If the DGH instruction has no effect in preventing the merging of memory accesses, the value of this field is 0b0000.

BF16, bits [47:44]

Indicates support for Advanced SIMD and Floating-point BFloat16 instructions in AArch64 state. Defined values are:

BF16	Meaning
0b0000	BFloat16 instructions are not implemented.
0b0001	BFCVT, BFCVTN, BFCVTN2, BFDOT, BFMLALB, BFMLALT, and BFMMLA instructions are implemented.
0b0010	As 0b0001, but the FPCR.EBF field is also supported.

All other values are reserved.

FEAT_BF16 adds the functionality identified by 0b0001.

FEAT_EBF16 adds the functionality identified by 0b0010.

When FEAT_SVE or FEAT_SME is implemented, this field must return the same value as ID_AA64ZFR0_EL1.BF16.

From Armv8.6 and Armv9.1, the value 0b0000 is not permitted.

SPECRES, bits [43:40]

Indicates support for prediction invalidation instructions in AArch64 state. Defined values are:

SPECRES	Meaning
0b0000	Prediction invalidation instructions are not implemented.
0b0001	CFP RCTX, DVP RCTX and CPP RCTX instructions are implemented.

All other values are reserved.

FEAT_SPECRES implements the functionality identified by 0b0001.

From Armv8.5, the value 0b0000 is not permitted.

SB, bits [39:36]

Indicates support for SB instruction in AArch64 state. Defined values are:

SB	Meaning
0b0000	SB instruction is not implemented.
0b0001	SB instruction is implemented.

All other values are reserved.

FEAT_SB implements the functionality identified by 0b0001.

In Armv8.0, the permitted values are 0b0000 and 0b0001.

From Armv8.5, the only permitted value is 0b0001.

FRINTTS, bits [35:32]

Indicates support for the FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are implemented. Defined values are:

FRINTTS	Meaning
0b0000	FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are not implemented.
0b0001	FRINT32Z, FRINT32X, FRINT64Z, and FRINT64X instructions are implemented.

All other values are reserved.

FEAT_FRINTTS implements the functionality identified by 0b0001.

From Armv8.5, the only permitted value is 0b0001.

GPI, bits [31:28]

Indicates support for an IMPLEMENTATION DEFINED algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:

GPI	Meaning
0b0000	Generic Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.
0b0001	Generic Authentication using an IMPLEMENTATION DEFINED algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

FEAT_PACIMP implements the functionality identified by 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPA is nonzero, or the value of ID_AA64ISAR2_EL1.GPA3 is nonzero, this field must have the value 0b0000.

GPA, bits [27:24]

Indicates whether the QARMA5 algorithm is implemented in the PE for generic code authentication in AArch64 state. Defined values are:

GPA	Meaning
0b0000	Generic Authentication using the QARMA5 algorithm is not implemented.
0b0001	Generic Authentication using the QARMA5 algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

FEAT_PACQARMA5 implements the functionality identified by 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPI is nonzero, or the value of ID_AA64ISAR2_EL1.GPA3 is nonzero, this field must have the value 0b0000.

LRCPC, bits [23:20]

Indicates support for weaker release consistency, RCpc, [illegible]ed mo[illegible]. Defined values are:

LRCPC	Meaning
0b[illegible]	RCpc instructions are not implemented.
0b0001	The no offset LDAPR, LDAPRB, and LDAPRH instructions are implemented.
0b0010	As 0b0001, and the LDAPR (unscaled immediate) and STLR (unscaled immediate) instructions are implemented.

All other values [illegible] reserv[illegible].

FEAT_LRCPC impl[illegible]nts the f[illegible]tionality identified by the value 0b0001.

FEAT_LRCPC2 implem[illegible]e functionality identified by the value 0b0010.

Fro[illegible] Armv8[illegible]ue 0b0000 is not permitted.

From Ar[illegible]8.4, the value 0b0001 is not permitted.

FCMA, bits [19:16]

Indicates support for complex number addition and multiplication, where numbers are stored in vectors. Defined values are:

FCMA	Meaning
0b0000	The FCMLA and FCADD instructions are not implemented.
0b0001	The FCMLA and FCADD instructions are implemented.

All other values are reserved.

FEAT_FCMA implements the functionality identified by the value 0b0001.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is 0b0000.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is 0b0001.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.

JSCVT, bits [15:12]

Indicates support for JavaScript conversion from double precision floating point values to integers in AArch64 state. Defined values are:

JSCVT	Meaning
0b0000	The FJCVTZS instruction is not implemented.
0b0001	The FJCVTZS instruction is implemented.

All other values are reserved.

FEAT_JSCVT implements the functionality identified by 0b0001.

In Armv8.0, Armv8.1, and Armv8.2, the only permitted value is 0b0000.

From Armv8.3, if Advanced SIMD or Floating-point is implemented, the only permitted value is 0b0001.

From Armv8.3, if Advanced SIMD or Floating-point is not implemented, the only permitted value is 0b0000.

API, bits [11:8]

Indicates whether an IMPLEMENTATION DEFINED algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:

API	Meaning
0b0000	Address Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.
0b0001	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.
0b0010	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE, and the HaveEnhancedPAC2() function returning FALSE.
0b0011	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

API	Meaning
0b0100	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
0b0101	Address Authentication using an IMPLEMENTATION DEFINED algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

FEAT_PAuth implements the functionality identified by 0b0001.

FEAT_EPAC implements the functionality identified by 0b0010.

FEAT_PAuth2 implements the functionality identified by 0b0011.

FEAT_FPAC implements the functionality identified by 0b0100.

FEAT_FPACCOMBINE implements the functionality identified by 0b0101.

When this field is nonzero, FEAT_PACIMP is implemented.

In Armv8.3, the permitted values are 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

From Armv8.6, the permitted values are 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.APA is nonzero, or the value of ID_AA64ISAR2_EL1.APA3 is nonzero, this field must have the value 0b0000.

APA, bits [7:4]

Indicates whether the QARMA5 algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction. Defined values are:

APA	Meaning
0b0000	Address Authentication using the QARMA5 algorithm is not implemented.
0b0001	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC() and HaveEnhancedPAC2() functions returning FALSE.
0b0010	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC() function returning TRUE and the HaveEnhancedPAC2() function returning FALSE.

APA	Meaning
0b0011	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning FALSE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
0b0100	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning FALSE, and the HaveEnhancedPAC() function returning FALSE.
0b0101	Address Authentication using the QARMA5 algorithm is implemented, with the HaveEnhancedPAC2() function returning TRUE, the HaveFPAC() function returning TRUE, the HaveFPACCombined() function returning TRUE, and the HaveEnhancedPAC() function returning FALSE.

All other values are reserved.

FEAT_PAuth implements the functionality identified by 0b0001.

FEAT_EPAC implements the functionality identified by 0b0010.

FEAT_PAuth2 implements the functionality identified by 0b0011.

FEAT_FPAC implements the functionality identified by 0b0100.

FEAT_FPACCOMBINED implements the functionality identified by 0b0101.

When this field is nonzero, FEAT_PACQARMA5 is implemented.

In Armv8.3, the permitted values are 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

From Armv8.6, the permitted values are 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.API is nonzero, or the value of ID_AA64ISAR2_EL1.APA3 is nonzero, this field must have the value 0b0000.

DPB, bits [3:0]

Data Persistence writeback. Indicates support for the DC CVAP and DC CVADP instructions in AArch64 state. Defined values are:

DPB	Meaning
0b0000	DC CVAP not supported.
0b0001	DC CVAP supported.
0b0010	DC CVAP and DC CVADP supported.

All other values are reserved.

FEAT_DPB implements the functionality identified by the value 0b0001.

FEAT_DPB2 implements the functionality identified by the value 0b0010.

In Armv8.2, the permitted values are 0b0001 and 0b0010.

From Armv8.5, the only permitted value is 0b0010.

Accessing ID_AA64ISAR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64ISAR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64ISAR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64ISAR1_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_AA64ISAR1_EL1;
```

E3.2.12 ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of information about implemented PE features in AArch64 state.

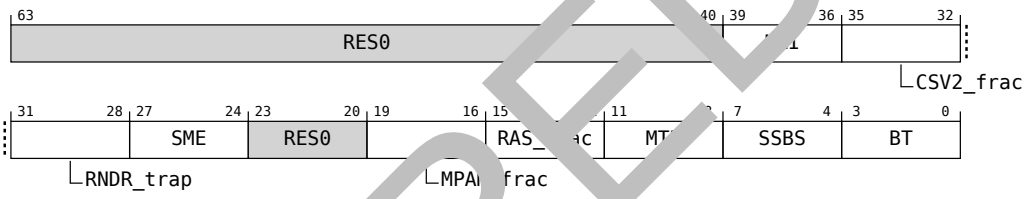
For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers .

Attributes

ID_AA64PFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64PFR1_EL1 bit assignments are:



Bits [63:40]

Reserved, RES0.

NMI, bits [39:36]

Non-maskable Interrupt. Indicates support for Non-maskable interrupts. Defined values are:

NMI	Meaning
0b0000	SCTLR_ELx.{SPINTMASK, NMI} and PSTATE.ALLINT with its associated instructions are not supported.
0b0001	SCTLR_ELx.{SPINTMASK, NMI} and PSTATE.ALLINT with its associated instructions are supported.

All other values are reserved.

FEAT_NMI implements the functionality identified by the value 0b0001.

From Armv8.8, the only permitted value is 0b0001.

CSV2_frac, bits [35:32]

CSV2 fractional field. Defined values are:

CSV2_frac	Meaning
0b0000	Either ID_AA64PFR0_EL1.CSV2 is not 0b0001, or the implementation does not disclose whether FEAT_CSV2_1p1 is implemented. FEAT_CSV2_1p2 is not implemented.
0b0001	FEAT_CSV2_1p1 is implemented, but FEAT_CSV2_1p2 is not implemented.
0b0010	FEAT_CSV2_1p2 is implemented.

All other values are reserved.

FEAT_CSV2_1p1 implements the functionality identified by the value 0b0001.

FEAT_CSV2_1p2 implements the functionality identified by the value 0b0010.

From Armv8.0, the permitted values are 0b0000, 0b0001, and 0b0010.

The values 0b0001 and 0b0010 are permitted only when ID_AA64PFR0_EL1.CSV2 is 0b0001.

RNDR_trap, bits [31:28]

Random Number trap to EL3 field. Defined values are:

RNDR_trap	Meaning
0b0000	Trapping of RNDR and RNDRRS to EL3 is not supported.
0b0001	Trapping of RNDR and RNDRRS to EL3 is supported. SCR_EL3.TRNDR is present.

All other values are reserved.

FEAT_RNG_TRAP implements the functionality identified by the value 0b0001.

SME, bits [23]

Scalable Matrix Extension. Defined values are:

SME	Meaning
0b0000	SME architectural state and programmers' model are not implemented.
0b0001	SME architectural state and programmers' model are implemented.
0b0010	As 0b0001, plus the SME2 ZT0 register.

All other values are reserved.

FEAT_SME implements the functionality identified by the value 0b0001.

FEAT_SME2 implements the functionality identified by the value 0b0010.

From Armv9.2, the permitted values are 0b0000, 0b0001, and 0b0010.

If implemented, refer to ID_AA64SMFR0_EL1 and ID_AA64ZFR0_EL1 for information about which SME and SVE instructions are available.

Bits [23:20]

Reserved, RES0.

MPAM_frac, bits [19:16]

Indicates the minor version number of support for the MPAM Extension.

Defined values are:

MPAM_frac	Meani
0b0000	The r or version nu ber of the MPAM xtensio 0.
0b0001	The minor v io umber of the MPAM ex sion is 1.

All other values are reserved.

When combined with the major version num from 64PFR0_EL1.MPAM, The combined “major.minor” version is:

AM Exte n vers	MPAM	MPAM_frac
No plemented.	0b0000	0b0000
v0.1 is lemented.	0b0000	0b0001
v1.0 is implemented.	0b0001	0b0000
v1.1 is implemented.	0b0001	0b0001

For ore infor ation, see The Memory Partitioning and Monitoring (MPAM) Extension .

RAS_ bits [15:12]

RAS Extens fractional field. Defined values are:

RAS_frac	Meaning
0b0000	If ID_AA64PFR0_EL1.RAS == 0b0001, RAS Extension implemented.

RAS_frac	Meaning
0b0001	If ID_AA64PFR0_EL1.RAS == 0b0001, as 0b0000 and adds support for: • Additional ERXMISC<m>_EL1 System registers. • Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension. Error recor[illegible]essed through System registers confor[illegible]to RAS [illegible]tem Architecture v1.1, whic[illegible]ncludes simp[illegible]ications to ERR<[illegible]STATUS, an[illegible]support for the optional RAS Tim[illegible]tamp and [illegible]AS Common Fault Injection M[illegible]l E[illegible]tensions.

All other values are reserved.

FEAT_RASv1p1 implements the functionality identified by the va[illegible]0b0001.

This field is valid only if ID_AA64PFR0_EL[illegible]RAS [illegible]

MTE, bits [11:8]

Support for the Memory Taggi[illegible]Extension. D[illegible]ned values are:

MTE	Meaning
0b0000	Memory Tagging Extension is not implemented.
0b0001	Instruction-only Memory Tagging Extension is implemented.
0b0010	Full Memory Tagging Extension is implemented.
0b0011	Memory Tagging Extension is implemented with support for asymmetric Tag Check Fault handling.

All other values are reserved.

FEAT_MTE implements the functionality identified by the value 0b0001.

FEAT_MTE2 implements the functionality identified by the value 0b0010.

FEAT_MTE3 implements the functionality identified by the value 0b0011.

In Armv8.5, the permitted values are 0b0000, 0b0001, 0b0010, and 0b0011.

From Armv8.7, the value 0b0010 is not permitted.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state. Defined values are:

SSBS	Meaning
0b0000	AArch64 provides no mechanism to control the use of Speculative Store Bypassing.
0b0001	AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.
0b0010	As 0b0001, and adds the MSR and MRS instructions to directly read and write the PSTATE.SSBS field.

All other values are reserved.

FEAT_SSBS implements the functionality identified by the value 0b0001.

FEAT_SSBS2 implements the functionality identified by the value 0b0010.

BT, bits [3:0]

Branch Target Identification mechanism support in AArch64 state. Defined values are:

BT	Meaning
0b0000	The Branch Target Identification mechanism is not implemented.
0b0001	The Branch Target Identification mechanism is implemented.

All other values are reserved.

FEAT_BTI implements the functionality identified by the value 0b0001.

From Armv8.5, the only permitted value is 0b0001.

Accessing ID_AA64PFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64PFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```

```
12      else
13          X[t, 64] = ID_AA64PFR1_EL1;
14  elsif PSTATE.EL == EL2 then
15      X[t, 64] = ID_AA64PFR1_EL1;
16  elsif PSTATE.EL == EL3 then
17      X[t, 64] = ID_AA64PFR1_EL1;
```

E3.2.13 ID_AA64ZFR0_EL1, SVE Feature ID register 0

The ID_AA64ZFR0_EL1 characteristics are:

Purpose

Provides additional information about the implemented features of the AArch64 Scalable Vector Extension instruction set, when one or more of FEAT_SVE and FEAT_SME is implemented.

For general information about the interpretation of the ID registers, see Principles of the ID scheme for fields in ID registers .

Configuration

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

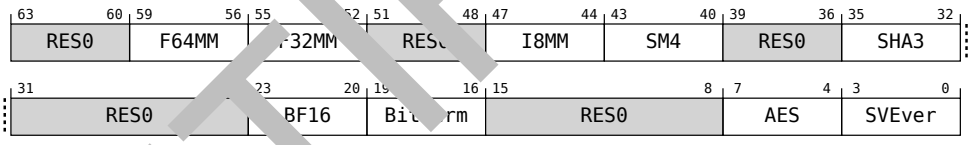
If FEAT_SME is implemented and FEAT_SVE is not implemented, then SVE instructions can only be executed when the PE is in Streaming SVE mode and the instructions are legal to execute in Streaming SVE mode.

Attributes

ID_AA64ZFR0_EL1 is a 64-bit register.

Field descriptions

The ID_AA64ZFR0_EL1 bit assignments are:



Bits [63:60]

Reserved, RES0.

F64MM, bits [59:56]

Indicates support for SVE FP64 double-precision floating-point matrix multiplication instructions. Defined values are:

F64MM	Meaning
0b0000	Double-precision matrix multiplication and related SVE instructions are not implemented.
0b0001	Double-precision variant of the FMMLA instruction, and the LD1RO* instructions are implemented. The 128-bit element variants of the SVE TRN1, TRN2, UZP1, UZP2, ZIP1, and ZIP2 instructions are also implemented.

All other values are reserved.

FEAT_F64MM implements the functionality identified by 0b0001.

From Armv8.2, the permitted values are 0b0000 and 0b0001.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

F32MM, bits [55:52]

Indicates support for the SVE FP32 single-precision floating-point matrix multiplication instruction. Defined values are:

F32MM	Meaning
0b0000	Single-precision matrix multiplication instruction is not implemented.
0b0001	Single-precision variant of the FMMLA instruction is implemented.

All other values are reserved.

FEAT_F32MM implements the functionality identified by 0b0001.

From Arm v8.2, the permitted values are 0b0000 and 0b0001.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

Bits [51:48]

Reserved, RES0.

I8MM, bits [47:44]

Indicates support for SVE Int8 matrix multiplication instructions. Defined values are:

I8MM	Meaning
0b0000	SVE Int8 matrix multiplication instructions are not implemented.
0b0001	SVE SMMLA, SUDOT, UMMLA, USMMLA, and USDOT instructions are implemented.

All other values are reserved.

FEAT_I8MM implements the functionality identified by 0b0001.

When Advanced SIMD and SVE are both implemented, this field must return the same value as ID_AA64ISAR1_EL1.I8MM.

From Armv8.6, the only permitted value is 0b0001.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the SVE instructions SMMLA, UMMLA, and USMMLA, irrespective of the value of this field.

SM4, bits [43:40]

Indicates support for SVE SM4 instructions. Defined values are:

SM4	Meaning
0b0000	SVE SM4 instructions are not implemented.
0b0001	SVE SM4E and SM4EKEY instructions are implemented.

All other values are reserved.

FEAT_SVE_SM4 implements the functionality identified by 0b0001.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

Bits [39:36]

Reserved, RES0.

SHA3, bits [35:32]

Indicates support for the SVE SHA3 instructions. Defined values are:

SHA3	Meaning
0b0000	SVE SHA3 instructions are not implemented.
0b0001	SVE RAX1 instruction is implemented.

All other values are reserved.

FEAT_SVE_SHA3 implements the functionality identified by 0b0001.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

Bits [31:24]

Reserved, RES0.

BF16, bits [23:20]

Indicates support for SVE BFloat16 instructions. Defined values are:

BF16	Meaning
0b0000	SVE BFloat16 instructions are not implemented.
0b0001	SVE BFCVT, BFCVTNT, BFDOT, BFMLALB, BFMLALT, and BFMMLA instructions are implemented.
0b0010	As 0b0001, but the FPCR.EBF field is also supported.

All other values are reserved.

FEAT_BF16 adds the functionality identified by 0b0001.

FEAT_EBF16 adds the functionality identified by 0b0010.

This field must return the same value as ID_AA64ISAR1_EL1.BF16.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the SVE instruction BFMMLA, irrespective of the value of this field.

From Armv8.6 and Armv9.1, the value 0b0000 is not permitted.

BitPerm, bits [19:16]

Indicates support for SVE bit permute instructions. Defined values are:

BitPerm	Meaning
0b0000	SVE bit permute instructions are not implemented.
0b0001	SVE BDEP, BEXT, and BGRP instructions are implemented.

All other values are reserved.

FEAT_SVE_BitPerm implements the functionality identified by 0b0001.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

Bits [15:8]

Reserved, RES0.

AES, bits [7:4]

Indicates support for SVE AES instructions. Defined values are:

AES	Meaning
0b0000	SVE AES* instructions are not implemented.
0b0001	SVE AESE, AESD, AESMC, and AESIMC instructions are implemented.
0b0010	As 0b0001, plus 64-bit source element variants of SVE PMULLB and PMULLT instructions are implemented.

All other values are reserved.

FEAT_SVE_AES implements the functionality identified by the value 0b0001.

FEAT_SVE_PMULL128 implements the functionality identified by the value 0b0010.

The permitted values are 0b0000 and 0b0010.

When the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and

enabled, software should not attempt to execute the instructions described by nonzero values of this field, irrespective of the value of this field.

SVEver, bits [3:0]

Indicates support for SVE instructions when one or more of FEAT_SME and FEAT_SVE is implemented. Defined values are:

SVEver	Meaning
0b0000	The SVE instructions are implemented.
0b0001	As 0b0000, and adds the mandatory SVE2 instructions.

All other values are reserved.

From Armv9, if this register is present, the value 0b0000 is not permitted.

FEAT_SVE2 implements the functionality identified by 0b0001 when the PE is not in Streaming SVE mode.

FEAT_SME implements the functionality identified by 0b0001 when the PE is in Streaming SVE mode.

Accessing ID_AA64ZFR0_EL1

Accesses to this register use the following encodings in the system register encoding space:

MRS <Xt>, ID_AA64ZFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && (IsFeatureImplemented(FEAT_FGT) || !IsZero(ID_AA64ZFR0_EL1) ||
        ↪boolean IMPLEMENTATION_DEFINED "ID_AA64ZFR0_EL1 trapped by HCR_EL2.TID3") &&
        ↪HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64ZFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64ZFR0_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = ID_AA64ZFR0_EL1;
```

E3.2.14 MPAM2_EL2, MPAM2 Register (EL2)

The MPAM2_EL2 characteristics are:

Purpose

Holds information to generate MPAM labels for memory requests when executing at EL2.

Configuration

This register has no effect if EL2 is not enabled in the current Security state.

When EL3 is implemented, AArch64 system register MPAM2_EL2 bit [63] is architecturally mapped to AArch64 system register MPAM3_EL3[63].

AArch64 system register MPAM2_EL2 bit [63] is architecturally mapped to AArch64 system register MPAM1_EL1[63].

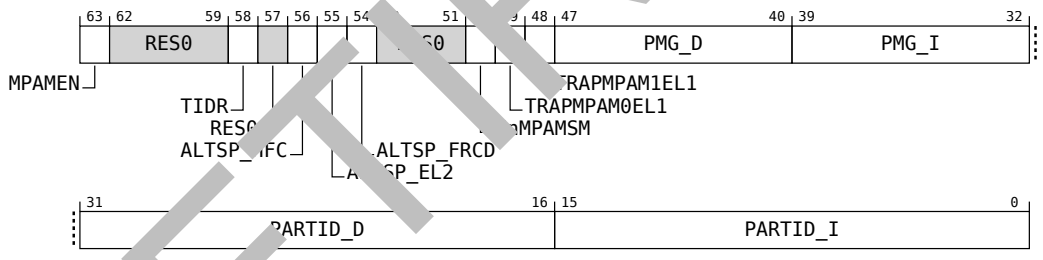
This register is present only when FEAT_MPAM is implemented. Otherwise, direct accesses to MPAM2_EL2 are UNDEFINED.

Attributes

MPAM2_EL2 is a 64-bit register.

Field descriptions

The MPAM2_EL2 bit assignments are:



MPAMEN, bit [63]

MPAM Enable. MPAM is disabled when MPAMEN == 1. When disabled, all PARTIDs and PMGs are output as their default values for the corresponding ID space.

MPAMEN	Meaning
0b0	The default PARTID and default PMG are output in MPAM information from all Exception levels.
0b1	MPAM information is output based on the MPAMn_ELx register for ELn according to the MPAM configuration.

If EL3 is not implemented, this field is read/write.

If EL3 is implemented, this field is read-only and reads the current value of the read/write MPAM3_EL3.MPAMEN bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Accessing this field has the following behavior:

- **RW** if !HaveEL(EL3)
- Otherwise, access to this field is **RO**

Bits [62:59]

Reserved, RES0.

TIDR, bit [58]

When (FEAT_MPAMv0p1 is implemented or FEAT_MPAMv1p1 is implemented) and MPAMIDR_EL1.HAS_TIDR == 1:

TIDR traps accesses to MPAMIDR_EL1 from EL1 to EL2.

TIDR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Trap accesses to MPAMIDR_EL1 from EL1 to EL2.

MPAMHCR_EL2.TRAP_MPAMIDR_EL1 = 1 also traps MPAMIDR_EL1 accesses from EL1 to EL2. If either TIDR or TRAP_MPAMIDR_EL1 are 1, accesses are trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [57]

Reserved.

ALTSP_HFC, bit [56]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == 1:

Hierarchical force of alternative PARTID space controls. When MPAM3_EL3.ALTSP_HEN is 0, ALTSP controls in MPAM2_EL2 have no effect. When MPAM3_EL3.ALTSP_HEN is 1, this bit selects whether the PARTIDs in MPAM1_EL1 and MPAM0_EL1 are in the primary (0) or alternative (1) PARTID space for the security state.

ALTSP_HFC	Meaning
0b0	When MPAM3_EL3.ALTSP_HEN is 1, the PARTID space of MPAM1_EL1.PARTID_I, MPAM1_EL1.PARTID_D, MPAM0_EL1.PARTID_I, and MPAM0_EL1.PARTID_D are in the primary PARTID space for the Security state.

ALTSP_HFC	Meaning
0b1	When MPAM3_EL3.ALTSP_HEN is 1, the PARTID space of MPAM1_EL1.PARTID_I, MPAM1_EL1.PARTID_D, MPAM0_EL1.PARTID_I, and MPAM0_EL1.PARTID_D are in the alternative PARTID space for the Security state.

This control has no effect when MPAM3_EL3.ALTSP_HEN is 0.

For more information, see 'Alternative PARTID spaces and selection' in Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A (ARM DDI 0598).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ALTSP_EL2, bit [55]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == 1:

Select alternative PARTID space for PARTIDs in MPAM2_EL2 when MPAM3_EL3.ALTSP_HEN is 1.

ALTSP_EL2	Meaning
0b0	When MPAM3_EL3.ALTSP_HEN is 1, selects the primary PARTID space for MPAM2_EL2.PARTID_I and MPAM2_EL2.PARTID_D.
0b1	When MPAM3_EL3.ALTSP_HEN is 1, selects the alternative PARTID space for MPAM2_EL2.PARTID_I and MPAM2_EL2.PARTID_D.

For more information, see 'Alternative PARTID spaces and selection' in Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A (ARM DDI 0598).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ALTSP_FRCD, bit [54]

When FEAT_RME is implemented and MPAMIDR_EL1.HAS_ALTSP == 1:

Alternative PARTID forced for PARTIDs in this register.

ALTSP_FRCD	Meaning
0b0	The PARTIDs in this register are using the primary PARTID space.
0b1	The PARTIDs in this register are using the alternative PARTID space.

This bit indicates that a higher Exception level has forced the PARTIDs in this register to use the alternative PARTID space defined for the current Security state. In EL2, it is also 1 when MPAM2_EL2.ALTSP_EL2 is 1.

For more information, see 'Alternative PARTID spaces and selection' in Arm® Architecture Reference Manual Supplement, Memory System Resource Partitioning and Monitoring (MPAM) for Armv8-A (ARM DDI 0598).

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is **RO**.

Otherwise:

RES0

Bits [53:51]

Reserved, RES0.

EnMPAMSM, bit [50]

When FEAT_SME is implemented:

Traps execution at EL1 of instructions that directly access the MPAMSM_EL1 register to EL2. The exception is reported using ESR_EL2.EC value 0x1D.

EnMPAMSM	Meaning
0b0	This control causes execution of these instructions at EL1 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

This field has no effect on accesses to MPAMSM_EL1 from EL2 or EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TRAPMPAM0EL1, bit [49]

Trap accesses from EL1 to the MPAM0_EL1 register trap to EL2.

TRAPMPAM0EL1	Meaning
0b0	Accesses to MPAM0_EL1 from EL1 are not trapped.
0b1	Accesses to MPAM0_EL1 from EL1 are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b1.
  - When EL3 is implemented, this field resets to an architecturally UNKNOWN value.

TRAPMPAM1EL1, bit [48]

Trap accesses from EL1 to the MPAM1_EL1 register trap to EL2.

TRAPMPAM1EL1	Meaning
0b0	Accesses to MPAM1_EL1 from EL1 are not trapped.
0b1	Accesses to MPAM1_EL1 from EL1 are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset:
  - When EL3 is not implemented, this field resets to 0b1.
  - When EL3 is implemented, this field resets to an architecturally UNKNOWN value.

PMG_D, bits [40:32]

Performance monitoring group for data accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PMG_I, bits [39:32]

Performance monitoring group for instruction accesses.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_D, bits [31:16]

Partition ID for data accesses, including load and store accesses, made from EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PARTID_I, bits [15:0]

Partition ID for instruction accesses made from EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MPAM2_EL2

None of the fields in this register are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, MPAM2_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        X[t, 64] = MPAM2_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = MPAM2_EL2;
```

MSR MPAM2_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
```

```
        else
            AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        MPAM2_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    MPAM2_EL2 = X[t, 64];
```

MRS <Xt>, MPAM1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && MPAM2_EL2.TRMPAM1EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x900];
    else
        X[t, 64] = MPAM1_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        X[t, 64] = MPAM2_EL2;
    else
        X[t, 64] = MPAM1_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = MPAM1_EL1;
```

MSR MPAM1_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
```

```
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && MPAM2_EL2.TRAPMPAM1EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x900] = X[t, 64];
    else
        MPAM1_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && MPAM3_EL3.TRAPLOWER == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        MPAM2_EL2 = X[t, 64];
    else
        MPAM1_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    MPAM1_EL1 = X[t, 64];
```

E3.2.15 SCR_EL3, Secure Configuration Register

The SCR_EL3 characteristics are:

Purpose

Defines the configuration of the current Security state. It specifies:

- The Security state of EL0, EL1, and EL2. The Security state is Secure, Non-secure, or Realm.
- The Execution state at lower Exception levels.
- Whether IRQ, FIQ, SError interrupts, and External abort exceptions are taken to EL3.
- Whether various operations are trapped to EL3.

Configuration

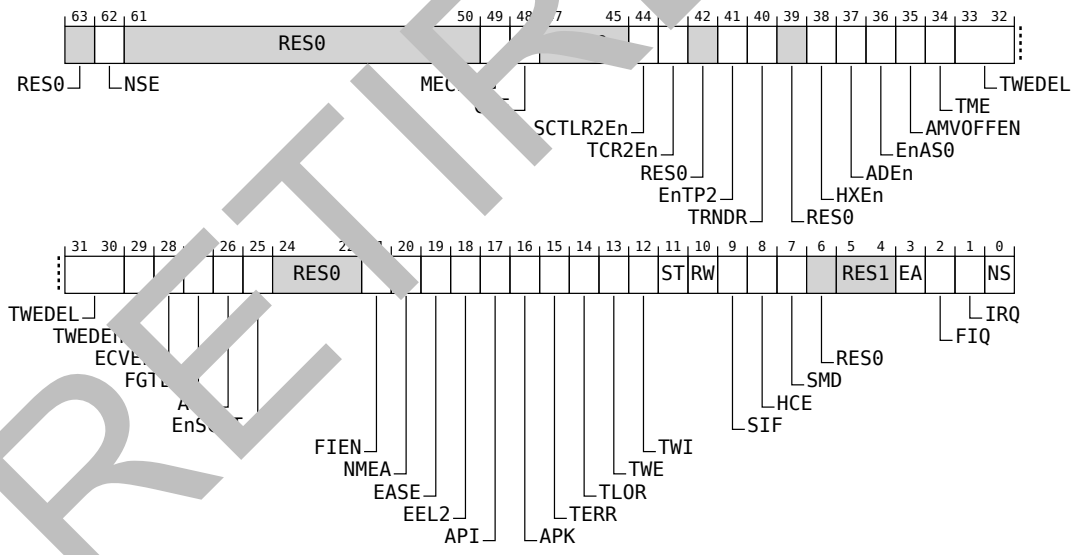
This register is present only when EL3 is implemented. Otherwise, [illegible] accesses to SCR_EL3 are UNDEFINED.

Attributes

SCR_EL3 is a 64-bit register.

Field descriptions

The SCR_EL3 bit assignments are:



Bit [63]

Reserved, RES0.

NSE, bit [62]

When FEAT_RME is implemented

NSE, bit [62]

This field, evaluated with SCR_EL3.NS, selects the Security state of EL2 and lower Exception levels.

For a description of the values derived by evaluating NS and NSE together, see SCR_EL3.NS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise

NSE, bit [62]

Reserved, RES0, and the Effective value of this bit is 0b0.

Bits [61:50]

Reserved, RES0.

MECEn, bit [49]

When FEAT_MEC is implemented:

Enables access to the following EL2 MECID registers, from EL2:

- MECID_P0_EL2.
- MECID_A0_EL2
- MECID_P1_EL2
- MECID_A1_EL2
- VMECID_P_EL2
- VMECID_A_EL2

Accesses to these registers are trapped and reported using EC value 0x18.

MECEn	Meaning
0b0	Accesses from EL2 to a listed MECID register are trapped to EL3. The value of a listed EL2 MECID register is treated as 0 for all purposes other than direct reads or writes to the register from EL3.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

GPF, bit [48]

When FEAT_RME is implemented:

Controls the reporting of Granule protection faults at EL0, EL1 and EL2.

GPF	Meaning
0b0	This control does not cause exceptions to be routed from EL0, EL1 or EL2 to EL3.

GPF	Meaning
0b1	GPFs at EL0, EL1 and EL2 are routed to EL3 and reported as Granule Protection Check exceptions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [47:45]

Reserved, RES0.

SCTLR2En, bit [44]

When FEAT_SCTLR2 is implemented:

SCTLR2_ELx register trap control. Enables acce[illegible]to [illegible]TLR2_[illegible]L1 an[illegible]SCTLR2_EL2 registers.

SCTL[illegible]	Meaning
0b0	EL1 and EL2 accesses to SCTLR2_EL1 and SCTLR2_EL2 registers are disabled, and trapped to EL3. The values in these registers are treated as 0.
0b1	This control does not cause any instructions to be trapped.

Traps are re[illegible]orted us[illegible]an ESR [illegible]L3.EC value of 0x18.

Traps [illegible]e not ta[illegible]n if ther[illegible]higher priority exception generated by the access.

The [illegible]et beh[illegible]is field is:

- On [illegible]Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TCR2En, bit [43]

When FEAT_TCR2 is implemented:

TCR2_ELx register trap control. Enables access to TCR2_EL1 and TCR2_EL2 registers.

TCR2En	Meaning
0b0	EL1 and EL2 accesses to TCR2_EL1 and TCR2_EL2 registers are disabled, and trapped to EL3. The values in these registers are treated as 0.

TCR2En	Meaning
0b1	This control does not cause any instructions to be trapped.

Traps are reported using an ESR_EL3.EC value of 0x18.

Traps are not taken if there is a higher priority exception generated by the access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [42]

Reserved, RES0.

EnTP2, bit [41]

When FEAT_SME is implemented:

Traps instructions executed at EL2, EL1, a[illegible]EL0 tha[illegible]access TPIDR2_EL0 to EL3. The exception is reported using ESR_ELx.EC value 0x18.

En[illegible]2	Meaning
0b0	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The res[illegible]beha[illegible]r of th[illegible]field[illegible]:

[illegible]On a Wa[illegible]reset, this field resets to an architecturally UNKNOWN value.

Otherw[illegible]

RES0

TRNDR, bit [40]

When FEAT_RNG_TRAP is implemented:

Controls trapping of reads of RNDR and RNDRRS. The exception is reported using ESR_ELx.EC value 0x18.

TRNDR	Meaning
0b0	This control does not cause RNDR and RNDRRS to be trapped. When FEAT_RNG is implemented: • ID_AA64ISAR0_EL1.RNDR returns the value 0b0001. When FEAT_RNG is not implemented: • ID_AA64ISAR0_EL1.RNDR returns the value 0b0000. • MRS reads of RNDR and RNDRRS are UNDEFINED.
0b1	ID_AA64ISAR0_EL1.RNDR returns the value 0b0001. Any attempt to read RNDR or RNDRRS is trapped to EL3.

When FEAT_RNG is not implemented, Arm recommends that SCR_EL3.RNDR is initialized before entering Exception levels below EL3 and not subsequently modified.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

Bit [39]

Reserved, RES0.

HXEn, bit [38]

When FEAT_HCX is implemented:

Enables access to the HCRX_EL2 register at EL2 from EL3.

HXEn	Meaning
0b0	Accesses at EL2 to HCRX_EL2 are trapped to EL3. Indirect reads of HCRX_EL2 return 0.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ADEn, bit [37]

When FEAT_LS64_ACCDATA is implemented:

Enables access to the ACCDATA_EL1 register at EL1 and EL2.

ADEn	Meaning
0b0	Accesses to ACCDATA_EL1 at EL1 and EL2 are trapped to EL3, unless the accesses are trapped to EL2 by the EL2 fine-grained trap.
0b1	This control does not cause accesses to ACCDATA_EL1 to be trapped.

If the HFGWTR_EL2.nACCDATA_EL1 or HFGRTR_EL2.nACCDATA_EL1 traps are enabled, they take priority over this trap.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnAS0, bit [36]

When FEAT_LS64_ACCDATA is implemented:

Traps execution of an ST64BV0 instruction at EL0 or EL1, or EL2 to EL3.

EnAS0	Meaning
0b0	EL0 execution of an ST64BV0 instruction is trapped to EL3, unless it is trapped to EL1 by SCTLR_EL1.EnAS0, or to EL2 by either HCRX_EL2.EnAS0 or SCTLR_EL2.EnAS0. EL1 execution of an ST64BV0 instruction is trapped to EL3, unless it is trapped to EL2 by HCRX_EL2.EnAS0. EL2 execution of an ST64BV0 instruction is trapped to EL3.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

AMVOFFEN, bit [35]

When FEAT_AMUv1p1 is implemented:

Activity Monitors Virtual Offsets Enable.

AMVOFFEN	Meaning
0b0	Accesses to AMEVCNTVOFF0<n>_EL2 and AMEVCNTVOFF1<n>_EL2 at EL2 are trapped to EL3. Indirect reads of the virtual offset registers are zero.
0b1	Accesses to AMEVCNTVOFF0<n>_EL2 and AMEVCNTVOFF1<n>_EL2 are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TME, bit [34]

When FEAT_TME is implemented:

Enables access to the TSTART, TCOMMIT, TTEST and TCANCEL instructions at EL0, EL1 and EL2.

TME	Meaning
0b0	EL0, EL1 and EL2 accesses to TSTART, TCOMMIT, TTEST and TCANCEL instructions are UNDEFINED.
0b1	This control does not cause any instruction to be UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TWEDEL, bits [33:30]

When FEAT_TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCR_EL3.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCR_EL3.TWE as $2^{(TWEDEL + 8)}$ cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TWEDEn, bit [29]

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCR_EL3.TWE.

Traps are reported using an ESR_ELx.EC value of 0x01.

TWEDEn	Meaning
0b0	The dela[illegible] the trap is IMPL[illegible]ENTATION[illegible]EFINED.
0b1	The d[illegible]y for taking [illegible]e trap is at least the [illegible]umber [illegible]ycles def[illegible]ed in SCR_EL3.[illegible]VED[illegible].

The reset behavior of this field is:

- On a Warm reset, this field resets to an ar[illegible]itectura[illegible]y UNKN[illegible] value.

Otherwise:

RES0

ECVEn, bit [28]

When FEAT_ECV is implemented:

ECV Enable. Enable[illegible]ccess to the CN[illegible]OFF_EL2 register.

ECVEn	Meaning
0b0	EL2 accesses to CNTPOFF_EL2 are trapped to EL3, and the value of CNTPOFF_EL2 is treated as 0 for all purposes other than direct reads or writes to the register from EL3.
0b1	EL2 accesses to CNTPOFF_EL2 are not trapped to EL3 by this mechanism.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

FGTEn, bit [27]

When FEAT_FGT is implemented:

Fine-Grained Traps Enable. When EL2 is implemented, enables the traps to EL2 controlled by HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2, and HFGWTR_EL2, and controls access to

those registers.

If EL2 is not implemented but EL3 is implemented, FEAT_FGT implements the MDCR_EL3.TDCC traps.

FGTEn	Meaning
0b0	EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are trapped to EL3, and the traps to EL2 controlled by those registers are disabled.
0b1	EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2 registers are not trapped to EL3 by this mechanism.

Traps caused by accesses to the fine-grained trap registers are reported using an ESR_ELx.EC value of 0x18 and its associated ISS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ATA, bit [26]

When FEAT_MTE2 is implemented:

Allocation Tag Access. Controls access to Allocation Tags, System registers for Memory tagging, and prevention of Tag checking, at EL2, EL1 and EL0.

ATA	Meaning
0b0	Access to Allocation Tags is prevented at EL2, EL1, and EL0. Accesses at EL1 and EL2 to GCR_EL1, RGSR_EL1, TFSR_EL1, TFSR_EL2 or TFSRE0_EL1 that are not UNDEFINED or trapped to a lower Exception level are trapped to EL3. Accesses at EL2 using MRS or MSR with the register name TFSR_EL12 that are not UNDEFINED are trapped to EL3. Memory accesses at EL2, EL1, and EL0 are not subject to a Tag Check operation.
0b1	This control does not prevent access to Allocation Tags at EL2, EL1, and EL0. This control does not prevent Tag checking at EL2, EL1, and EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnSCXT, bit [25]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Enables access to the SCXTNUM_EL2, SCXTNUM_EL1, and SCXTNUM_EL0 registers.

EnSCXT	Meaning
0b0	Accesses at EL0, EL1, and EL2 to SCXTNUM_EL0, SCXTNUM_EL1, or SCXTNUM_EL2 registers are trapped to EL3 if they are not trapped by a higher priority exception, and the values of these registers are treated as 0.
0b1	This control does not cause any accesses to be trapped, or register values to be treated as 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [24:22]

Reserved, RES0.

FIEN, bit [21]

When FEAT_RASv1p1 is implemented:

Fault Injection enable. Trap accesses to the registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1 from EL1 and EL2 to EL3, reported using an ESR_ELx.EC value of 0x18.

FIEN	Meaning
0b0	Accesses to the specified registers from EL1 and EL2 generate a Trap exception to EL3.
0b1	This control does not cause any instructions to be trapped.

If EL3 is not implemented, the Effective value of SCR_EL3.FIEN is 0b1.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

NMEA, bit [20]

When FEAT_DoubleFault is implemented:

Non-maskable External Aborts. Controls whether PSTATE.A masks SError exceptions at EL3.

NMEA	Meaning
0b0	SError exceptions are not taken at EL3 if PSTATE.A == 1.
0b1	SError exceptions are taken at EL3 regardless of the value of PSTATE.A.

This field is ignored by the PE and treated as zero when SCR_EL3.EA == 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

EASE, bit [19]

When FEAT_DoubleFault is implemented:

External aborts and SError interrupt vector.

EASE	Meaning
0b0	Synchronous External abort exceptions taken to EL3 are taken to the appropriate synchronous exception vector offset from VBAR_EL3.
0b1	Synchronous External abort exceptions taken to EL3 are taken to the appropriate SError interrupt vector offset from VBAR_EL3.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

EEL2, bit [18]

When FEAT_SEL2 is implemented:

Secure EL2 Enable.

EEL2	Meaning
0b0	All behaviors associated with Secure EL2 are disabled. All registers, including timer registers, defined by FEAT_SEL2 are UNDEFINED, and those timers are disabled.
0b1	All behaviors associated with Secure EL2 are enabled.

When the value of this bit is 1, then:

- When SCR_EL3.NS == 0, the SCR_EL3.RW bit is treated as 1 for all purposes other than reading or writing the register.
- If Secure EL1 is using AArch32, then any of the following operations executed in Secure EL1, is trapped to Secure EL2, using the EC value of ESR_EL2.EC== 0x3 :
  - A read or write of the SCR.
  - A read or write of the NSACR.
  - A read or write of the MVBAR.
  - A read or write of the SDCR.
  - Execution of an ATS12NSO** instruction.
- If Secure EL1 is using AArch32, then any of the following operations, executed in Secure EL1, is trapped to Secure EL2 using the EC value of ESR_EL2.EC== 0x0 :
  - Execution of an SRS instruction that accesses R13_mon.
  - Execution of an RFE (Banked register) or MSR (Banked register) instruction that would access SPSR_mon, R13_mon, or R14_mon.

If the Effective value of SCR_EL3.EEL2 is 0, then these operations executed in Secure EL1 using AArch32 are trapped to EL3.

A Secure only implementation that does not implement EL3 but implements EL2, behaves as if SCR_EL3.EEL2 == 1.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

API, bit [17]

When FEAT_SEL2 is implemented and FEAT_PAuth is implemented

API, bit [17]

Controls the use of the following instructions related to Pointer Authentication. Traps are reported using an ESR_ELx.EC value of 0x09:

- PACGA, which is always enabled.

- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB, RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETAA, ERETAB, LDRAA and LDRAB when:
  - In EL0, when HCR_EL2.TGE == 0 or HCR_EL2.E2H == 0, and the associated SCTLR_EL1.En<N><M> == 1.
  - In EL0, when HCR_EL2.TGE == 1 and HCR_EL2.E2H == 1, and the associated SCTLR_EL2.En<N><M> == 1.
  - In EL1, when the associated SCTLR_EL1.En<N><M> == 1.
  - In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

API	Meaning
0b0	The use of any instruction related to pointer authentication in any exception level except EL3 when the instructions are enabled are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.API bit.
0b1	This control does not cause any instructions to be trapped.

An instruction is trapped only if Pointer Authentication is enabled for that instruction, for more information, see 'PAC generation and verification keys'.

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SEL2 is not implemented and FEAT_PAuth is implemented

API, bit [17]

Control the use of instructions related to Pointer Authentication:

PACGA

- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB, PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZ, RETAA, RETAB, BRAA, BRAB, ERAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ, ERETAA, ERETAB, LDRAA and LDRAB when:
  - In Non-secure EL0, when HCR_EL2.TGE == 0 or HCR_EL2.E2H == 0, and the associated SCTLR_EL1.En<N><M>== 1.
  - In Non-secure EL0, when HCR_EL2.TGE == 1 and HCR_EL2.E2H == 1, and the associated SCTLR_EL2.En<N><M> == 1.
  - In Secure EL0, when the associated SCTLR_EL1.En<N><M> == 1.
  - In Secure or Non-secure EL1, when the associated SCTLR_EL1.En<N><M> == 1.
  - In EL2, when the associated SCTLR_EL2.En<N><M> == 1.

API	Meaning
0b0	The use of any instruction related to pointer authentication in any Exception level except EL3 when the instructions are enabled are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.API bit.
0b1	This control does not cause any instructions to be trapped.

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

APK, bit [16]

When FEAT_PAuth is implemented:

Trap registers holding "key" values for Pointer Authentication. Traps accesses to the following registers, using an ESR_ELx.EC value of 0x18, from EL1 or EL2 to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps:

- APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1.
- APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1.
- APGAKeyLo_EL1, and APGAKeyHi_EL1.

APK	Meaning
0b0	Access to the registers holding "key" values for pointer authentication from EL1 or EL2 are trapped to EL3 unless they are trapped to EL2 as a result of the HCR_EL2.APK bit or other traps.
0b1	This control does not cause any instructions to be trapped.

For more information, see 'PAC generation and verification keys'.

If FEAT_PAuth is implemented but EL3 is not implemented, the system behaves as if this bit is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TERR, bit [15]

When FEAT_RAS is implemented:

Trap accesses of error record registers. Enables a trap to EL3 on accesses of error record registers.

TERR	Meaning
0b0	Accesses of the specified error record registers are not trapped by this mechanism.
0b1	Accesses of the specified error record registers at EL2 and EL1 are trapped to EL3, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.
- MRS accesses to ERRIDR_EL1 and ERXFR_EL1.
- If FEAT_RASv1p1 is implemented, MRS and MSR accesses to ERXMISC2_EL1 and ERXMISC3_EL1.

In AArch32 state, the instructions affected by this control are:

- MRC and MCR accesses to ERRSELR, ERXADDR, ERXADDR2, ERXCTLR, ERXCTLR2, ERXMISC0, ERXMISC1, ERXMISC2, ERXMISC3, and ERXSTATUS.
- MRC accesses to ERRIDR, ERXFR, and ERXFR2.
- If FEAT_RASv1p1 is implemented, MRC and MCR accesses to ERXMISC4, ERXMISC5, ERXMISC6, and ERXMISC7.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL3.

Trapped AArch64 instructions are reported using EC syndrome value 0x18.

Trapped AArch32 instructions are reported using EC syndrome value 0x03.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TLOR, bit [14]

When FEAT_LOR is implemented:

Trap LOR registers. Traps accesses to the LORSA_EL1, LOREA_EL1, LORN_EL1, LORC_EL1, and LORID_EL1 registers from EL1 and EL2 to EL3, unless the access has been trapped to EL2.

TLOR	Meaning
0b0	This control does not cause any instructions to be trapped.

TLOR	Meaning
0b1	EL1 and EL2 accesses to the LOR registers that are not UNDEFINED are trapped to EL3, unless it is trapped HCR_EL2.TLOR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TWE, bit [13]

Traps EL2, EL1, and EL0 execution of WFE instructions to EL3, from any Security state and both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

TWE	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFE instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE, HCR.TWE, SCTLR_EL1.nTWE, SCTLR_EL2.nTWE, or HCR_EL2.TWE.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Since WFE and WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about when WFE instructions can cause the PE to enter a low-power state, see 'Wait for Event mechanism and Send event'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TWI, bit [12]

Traps EL2, EL1, and EL0 execution of WFI instructions to EL3, from any Security state and both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFIT instruction.

TWI	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt to execute a WFI instruction at any Exception level lower than EL3 is trapped to EL3, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI, HCR.TWI, SCTLR_EL1.nTWI, SCTLR_EL2.nTWI, or HCR_EL2.TWI.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about when WFI instructions cause the PE to enter a low-power state, see 'Wait for Interrupt'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ST, bit [11]

Traps Secure EL1 accesses to the Counter-timer Physical Secure timer registers to EL3, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

ST	Meaning
0b0	Secure EL1 using AArch64 accesses to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 are trapped to EL3 when Secure EL2 is disabled. If Secure EL2 is enabled, the behavior is as if the value of this field was 0b1.
0b1	This control does not cause any instructions to be trapped.

Accesses to the Counter-timer Physical Secure timer registers are always enabled at EL3. These registers are not accessible at EL0.

When FEAT_RME is implemented and Secure state is not implemented, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RW, bit [10]

When EL1 is capable of using AArch32 or EL2 is capable of using AArch32:

Execution state control for lower Exception levels.

RW	Meaning
0b0	Lower levels are all AArch32.
0b1	The next lower level is AArch64. If EL2 is present: • EL2 is AArch64. • EL2 controls EL1 and EL0 behaviors. If EL2 is not present: • EL1 is AArch64. • EL0 is determined by the Execution state described in the current process state when executing at EL0.

If AArch32 state is supported by the implementation at EL1, SCR_EL3.NS == 1, and AArch32 state is not supported by the implementation at EL2, the Effective value of this bit is 1.

If AArch32 state is supported by the implementation at EL1, FEAT_EL2 is implemented and SCR_EL3.{EEL2, NS} == {1, 0}, the Effective value of this bit is 1.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RAO/WI

SIF, bit [9]

Secure instruction fetch. When the PE is in Secure state, this bit disables instruction execution from memory marked in the first stage of translation as being Non-secure.

SIF	Meaning
0b0	Secure state instruction execution from memory marked in the first stage of translation as being Non-secure is permitted.
0b1	Secure state instruction execution from memory marked in the first stage of translation as being Non-secure is not permitted.

When FEAT_RME is implemented and Secure state is not implemented, this bit is RES0.

When FEAT_PAN3 is implemented, it is IMPLEMENTATION DEFINED whether SCR_EL3.SIF is also used to determine instruction access permission for the purpose of PAN.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HCE, bit [8]

Hypervisor Call instruction enable. Enables HVC instructions at EL3 and, if EL2 is enabled in the current Security state, at EL2 and EL1, in both Execution states, reported using an ESR_ELx.EC value of 0x00.

HCE	Meaning
0b0	HVC instructions are UNDEFINED.
0b1	HVC instructions are enabled at EL3, EL2, and EL1.

HVC instructions are always UNDEFINED at EL0 and, if Secure EL2 is disabled, at Secure EL1. Any resulting exception is taken from the current Exception level to the current Exception level.

If EL2 is not implemented, this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SMD, bit [7]

Secure Monitor Call disable. Disables SMC instructions at EL1 and above, from any Security state and both Execution states, reported using an ESR_ELx.EC value of 0x17.

SMD	Meaning
0b0	SMC instructions are enabled at EL3, EL2 and EL1.
0b1	SMC instructions are UNDEFINED.

SMC instructions are always UNDEFINED at EL0. Any resulting exception is taken from the current Exception level to the current Exception level.

If HCR_EL2.TSC or HCR.TSC traps attempted EL1 execution of SMC instructions to EL2, that trap has priority over this disable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

EA, bit [3]

External Abort and SError interrupt routing.

EA	Meaning
0b0	When executing at Exception levels below EL3, External aborts and SError interrupts are not taken to EL3. In addition, when executing at EL3: • SError interrupts are not taken. • External aborts are taken to EL3.
0b1	When executing at any Exception level, External aborts and SError interrupts are taken to EL3.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN val

FIQ, bit [2]

Physical FIQ Routing.

FI	Meaning
0	When executing at Exception levels below EL3, physical FIQ interrupts are not taken to EL3. When executing at EL3, physical FIQ interrupts are not taken.
0b1	When executing at any Exception level, physical FIQ interrupts are taken to EL3.

For more informatio ee 'Async onous exception routing'.

The re behav r of thi s:

- On a W this field resets to an architecturally UNKNOWN value.

IRQ, bit [

Physical IRQ Routing.

IRQ	Meaning
0b0	When executing at Exception levels below EL3, physical IRQ interrupts are not taken to EL3. When executing at EL3, physical IRQ interrupts are not taken.
0b1	When executing at any Exception level, physical IRQ interrupts are taken to EL3.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [0]

When FEAT_RME is implemented

NS, bit [0]

Non-secure bit. This field is used in combination with SCR_EL3.NSE to select the Security state of EL2 and lower Exception levels.

NSE	NS	Meaning
0b0	0b0	Secure.
0b0	0b1	Non-secure.
0b1	0b0	Reserved.
0b1	0b1	Realm.

When Secure state is not implemented, SCR_EL3.NS is RES1 and the effective value is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise

NS, bit [0]

Non-secure bit.

NS	Meaning
0b0	Indicates that EL0 and EL1 are in Secure state. When FEAT_SEL2 is implemented and SCR_EL3.EEL2 == 1, then EL2 is using AArch64 and in Secure state.
0b1	Indicates that Exception levels lower than EL3 are in Non-secure state, so memory accesses from those Exception levels cannot access Secure memory.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCR_EL3

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SCR_EL3

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SCR_EL3;
```

MSR SCR_EL3, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SCR_EL3 = X[t, 64];
```

E3.2.16 SCTLR_EL1, System Control Register (EL1)

The SCTLR_EL1 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL1 and EL0.

Configuration

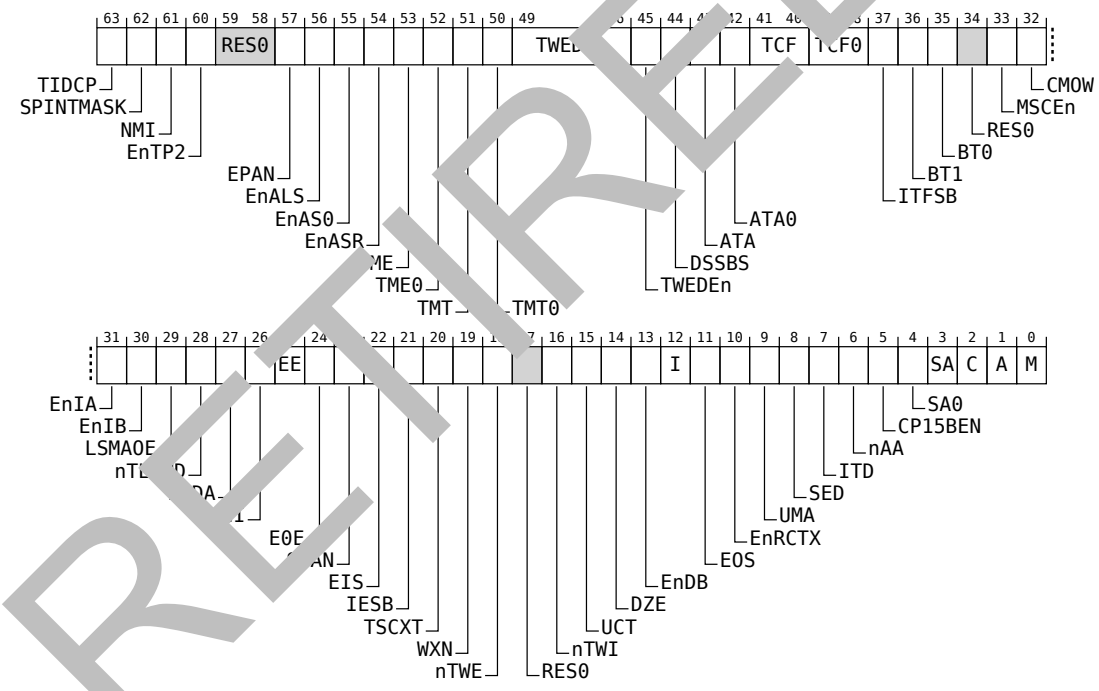
AArch64 system register SCTLR_EL1 bits [31:0] are architecturally mapped to AArch32 system register SCTLR[31:0].

Attributes

SCTLR_EL1 is a 64-bit register.

Field descriptions

The SCTLR_EL1 bit assignments are:



TIDCP, bit [63]

When FEAT_TIDCP1 is implemented:

Trap IMPLEMENTATION DEFINED functionality. When HCR_EL2.{E2H, TGE} != {1, 1}, traps EL0 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL1.

TIDCP	Meaning
0b0	No instructions accessing the System register or System instruction spaces are trapped by this mechanism.

TIDCP	Meaning
0b1	Instructions accessing the following System register or System instruction spaces are trapped to EL1 by this mechanism: • In AArch64 state, EL0 access to the encodings in the following reserved encoding spaces are trapped and reported using EC syndrome 0x18: – IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}. – IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name. In AArch32 state, EL0 MCR and MRC access to the following encodings are trapped and reported using EC syndrome 0x03: – All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}. – All coproc==p15, CRn==c10, opc1 =={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}. – All coproc==p15, CRn==c11, opc1=={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL1.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL1.

SPINTMASK	Meaning
0b0	Does not cause PSTATE.SP to mask interrupts. PSTATE.ALLINT is set to 1 on taking an exception to EL1.

SPINTMASK	Meaning
0b1	When PSTATE.SP is 1 and execution is at EL1, an IRQ or FIQ interrupt that is targeted to EL1 is masked regardless of any denotion of Superpriority. PSTATE.ALLINT is set to 0 on taking an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

NMI, bit [61]

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

NM	Meaning
0b0	This control does not affect interrupt masking behavior.
0b1	This control enables all of the following: • The use of the PSTATE.ALLINT interrupt mask. • IRQ and FIQ interrupts to have Superpriority as an additional attribute. • PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:

- On a Warm reset:
  - When EL2 is not implemented and EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnTP2, bit [60]

When FEAT_SME is implemented:

Traps instructions executed at EL0 that access TPIDR2_EL0 to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using ESR_ELx.EC value 0x18.

EnTP2	Meaning
0b0	This control causes execution of these instructions at EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

If FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [59:58]

Reserved, RES0.

EPAN, bit [57]

When FEAT_PAN3 is implemented:

Enhanced Privileged Access Never. When PSTATE.PAN is 1, determines whether an EL1 data access to a page with stage 1 EL0 instruction access permission generates a Permission fault as a result of the Privileged Access Never mechanism.

EPAN	Meaning
0b0	No additional Permission faults are generated by this mechanism.
0b1	An EL1 data access to a page with stage 1 EL0 data access permission or stage 1 EL0 instruction access permission generates a Permission fault. Any speculative data accesses that would generate a Permission fault as a result of PSTATE.PAN = 1 if the accesses were not speculative, will not cause an allocation into a cache.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnALS, bit [56]

When FEAT_LS64 is implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an LD64B or ST64B instruction at EL0 to EL1.

EnALS	Meaning
0b0	Execution of an LD64B or ST64B instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnAS0, bit [55]

When FEAT_LS64_ACCDATA is implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an ST64BV0 instruction at EL0 to EL1.

EnAS0	Meaning
0b0	Execution of an ST64BV0 instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnASR, bit [54]

When FEAT_LS64_V is implemented:

When HCR_EL2.{E2H, TGE} != {1, 1}, traps execution of an ST64BV instruction at EL0 to EL1.

EnASR	Meaning
0b0	Execution of an ST64BV instruction at EL0 is trapped to EL1.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TME, bit [53]

When FEAT_TME is implemented:

Enables the Transactional Memory Extension at EL1.

TME	Meaning
0b0	Any attempt to execute a TSTART instruction at EL1 is trapped to EL1, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL1.
0b1	This control does not cause any TSTART instruction to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TME0, bit [52]

When FEAT_TME is implemented:

Enables the Transactional Memory Extension at EL0.

TME0	Meaning
0b0	Any attempt to execute a TSTART instruction at EL0 is trapped to EL1, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL0.
0b1	This control does not cause any TSTART instruction to be trapped.

If FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TMT, bit [51]

When FEAT_TME is implemented:

Forces a trivial implementation of the Transactional Memory Extension at EL1.

TMT	Meaning
0b0	This control does not cause any TSTART instruction to fail.
0b1	When the TSTART instruction is executed at EL1, the transaction fails with a TRIVIAL failure cause.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TMT0, bit [50]

When FEAT_TME is implemented:

Forces a trivial implementation of the Transactional Memory Extension at EL0.

TMT0	Meaning
0b0	This control does not cause any TSTART instruction to fail.
0b1	When the TSTART instruction is executed at EL0, the transaction fails with a TRIVIAL failure cause.

If FEAT_VHE is implemented, EL2 is implemented and enabled in the current Security state, and HCR_EL2.{E2H, TGE} == {1, 1}, this field has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TWEDEL, bits [49:46]

When FEAT_TWED is implemented:

TWE Delay. A 4-bit unsigned number that, when SCTLR_EL1.TWEDEn is 1, encodes the minimum delay in taking a trap of WFE* caused by SCTLR_EL1.nTWE as $2^{(TWEDEL + 8)}$ cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TWEDEn, bit [45]

When FEAT_TWED is implemented:

TWE Delay Enable. Enables a configurable delayed trap of the WFE* instruction caused by SCTLR_EL1.nTWE.

TWEDEn	Meaning
0b0	The delay for taking the trap is IMPLEMENTATION DEFINED.
0b1	The delay for taking the trap is at least the number of cycles defined in SCTLR_EL1.TWEDEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL1.
0b1	PSTATE.SSBS is set to 1 on an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

RES0

ATA, bit [43]

When FEAT_MTE2 is implemented:

Allocation Tag Access in EL1.

When SCR_EL3.ATA == 1 and HCR_EL2.ATA == 1, controls access to Allocation Tags and Tag Check operations in EL1.

ATA	Meaning
0b0	Access to Allocation Tags is prevented at EL1. Memory accesses at EL1 are not subject to a Tag Check operation.
0b1	This control does not prevent access to Allocation Tags at EL1. Tag Checked memory accesses at EL1 are subject to a Tag Check operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ATA0, bit [42]

When FEAT_MTE2 is implemented:

Allocation Tag Access in EL0.

When SCR_EL3.ATA == 1, HCR_EL2.ATA == 1, EL2.{E2H, TGE} != {1, 1}, controls access to Allocation Tags and Tag Check operations in EL0.

ATA0	Meaning
0b0	Access to Allocation Tags is prevented at EL0. Memory accesses at EL0 are not subject to a Tag Check operation.
0b1	This control does not prevent access to Allocation Tags at EL0. Tag Checked memory accesses at EL0 are subject to a Tag Check operation.

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TCF, bits [41:40]

When FEAT_MTE2 is implemented:

Tag Check Fault in EL1. Controls the effect of Tag Check Faults due to Loads and Stores in EL1.

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

TCF	Meaning	Applies
0b00	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	
0b10	Tag Check Faults are asynchronously accumulated.	
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	When FEAT_MTE3 is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TCF0, bits [39:38]

When FEAT_MTE2 is implemented:

Tag Check Fault in EL0. When HCR_EL2.{E2H,TGE} != {1,1}, controls the effect of Tag Check Faults due to Loads and Stores in EL0.

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

Software may change this control bit on a context switch.

TCF0	Meaning	Applies
0b00	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	
0b10	Tag Check Faults are asynchronously accumulated.	
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	When FEAT_MTE3 is implemented

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ITFSB, bit [37]

When FEAT_MTE2 is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL1, all Tag Check Faults due to instructions executed before exception entry, that are reported

asynchronously, are synchronized into TFSRE0_EL1 and TFSR_EL1 registers.

ITFSB	Meaning
0b0	Tag Check Faults are not synchronized on entry to EL1.
0b1	Tag Check Faults are synchronized on entry to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

BT1, bit [36]

When FEAT_BTI is implemented:

PAC Branch Type compatibility at EL1.

BT1	Meaning
0b0	When the PE is executing at EL1, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
0b1	When the PE is executing at EL1, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

BT0, bit [35]

When FEAT_BTI is implemented:

PAC Branch Type compatibility at EL0.

BT0	Meaning
0b0	When the PE is executing at EL0, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
0b1	When the PE is executing at EL0, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

When the value of HCR_EL2.{E2H, TGE} is {1, 1}, the value of SCTLR_EL1.BT0 has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [34]

Reserved, RES0.

MSCEn, bit [33]

When FEAT_MOPS is implemented and (HCR_EL2.E2H == 0 or HCR_EL2.TGE == 0):

Memory Copy and Memory Set instructions Enable. Enables execution of the Memory Copy and Memory Set instructions at EL0.

MSCEn	Meaning
0b0	Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL0.
0b1	This control does not cause any instructions to be UNDEFINED.

When FEAT_MOPS is implemented and HCR_EL2.{E2H, TGE} is {1, 1}, the Effective value of this bit is 0b1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

CMOW, bit [32]

When FEAT_CMOW is implemented:

Controls cache maintenance instruction permission for the following instructions executed at EL0.

- IC IVAU, DC CIVAC, DC CIGDVAC and DC CIGVAC.

CMOW	Meaning
0b0	These instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, do not generate a stage 1 permission fault.
0b1	If enabled as a result of SCTLR_EL1.UCI==1, these instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, generate a stage 1 permission fault.

When AArch64.HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

For this control, stage 1 has write permission if all of the following apply:

- AP[2] is 0 or DBM is 1 in the stage 1 descriptor.
- Where APTable is in use, APTable[1] is 0 for all levels of the translation table.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnIA, bit [31]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see System register control of pointer authentication .

EnIA	Meaning
0b0	Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.
0b1	Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see System register control of pointer authentication .

EnIB	Meaning
0b0	Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.
0b1	Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented:

Load Multiple and Store Multiple Atomicity and Ordering Enable.

LSMAOE	Meaning
0b0	For all memory accesses at EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses and the memory accesses are not required to be ordered.
0b1	The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

nTLSMD, bit [28]

When FEAT_LSMAOC is implemented:

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

nTLSMD	Meaning
0b0	All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.

nTLSMD	Meaning
0b1	All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

EnDA, bit [27]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see System register control of pointer authentication .

EnDA	Meaning
0b0	Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.
0b1	Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions, to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

This applies to DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.

If FEAT_MTE is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP, and DC CGDVAP.

If FEAT_DPB2 and FEAT_MTE are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

UCI	Meaning
0b0	Execution of the specified instructions at EL0 using AArch64 is trapped.
0b1	This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EE, bit [25]

Endianness of data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime.

EE	Meaning
0b0	Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.
0b1	Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

Endianness of data accesses at EL0.

E0E	Meaning
0b0	Explicit data accesses at EL0 are little-endian.
0b1	Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0, then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0, then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of LDTR, LDTRH, LDTRSH, LDTRSW, STTR, and STTRH instructions executed at EL1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SPAN, bit [23]

When FEAT_PAN is implemented:

Set Privileged Access Never, on taking an exception to EL1.

SPAN	Meaning
0b0	PSTATE.PAN is set to 1 on taking an exception to EL1.
0b1	The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

EIS, bit [22]

When FEAT_ExS is implemented:

Exception Entry is Context Synchronizing.

EIS	Meaning
0b0	The taking of an exception to EL1 is not a context synchronizing event.

EIS	Meaning
0b1	The taking of an exception to EL1 is a context synchronizing event.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

If SCTLR_EL1.EIS is set to 0b0:

- Indirect writes to ESR_EL1, FAR_EL1, SPSR_EL1, ELR_EL1 are synchronized on exception entry to EL1, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL1.EIS:

- Changes to the PSTATE information on entry to EL1.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

IESB, bit [21]

When FEAT_IESB is implemented:

Implicit Error Synchronization event enable. Possible values are:

IESB	Meaning
0b0	Disabled.
0b1	An implicit error synchronization event is added: • At each exception taken to EL1. • Before the operational pseudocode of each ERET instruction executed at EL1.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSX instruction taken to EL1 and before each DRPS instruction executed at EL1, in addition to the other cases where it is added.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TSCXT, bit [20]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.

TSCXT	Meaning
0b0	EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.
0b1	EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The value of SCXTNUM_EL0 is treated as 0.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL1&0 translation regime, this bit can force all memory regions that are writable to be treated as XN.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	Any region that is writable in the EL1&0 translation regime is forced to XN for accesses from software executing at EL1 or EL0.

This bit applies only when SCTLR_EL1.M bit is set.

The WXN bit is permitted to be cached in a TLB.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL1, or to EL2 when it is implemented and enabled for the current

Security state and HCR_EL2.TGE is 1, from both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFET instruction.

nTWE	Meaning
0b0	Any attempt to execute a WFE instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [17]

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, reported using an ESR_ELx.EC value of 0x01.

When FEAT_WFxT is implemented, this trap also applies to the WFIT instruction.

nTWI	Meaning
0b0	Any attempt to execute a WFI instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

UCT	Meaning
0b0	Accesses to the CTR_EL0 from EL0 using AArch64 are trapped.
0b1	This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

If FEAT_MTE is implemented, this trap also applies to DC GVA and DC GZVA.

DZE	Meaning
0b0	Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions this trap applies to are not supported.
0b1	This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL1&0 translation regime.

For more information, see System register control of pointer authentication .

EnDB	Meaning
0b0	Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.
0b1	Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

I, bit [12]

Stage 1 instruction access Cacheability control, for accesses at EL0 and EL1:

I	Meaning
0	All instruction access to Stage 1 Normal memory from EL0 and EL1 are Stage 1 Non-cacheable. If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
0b1	This control has no effect on the Stage 1 Cacheability of instruction access to Stage 1 Normal memory from EL0 and EL1. If the value of SCTLR_EL1.M is 0, instruction accesses from stage 1 of the EL1&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

When the value of the HCR_EL2.DC bit is 1, then instruction access to Normal memory from EL0 and EL1 are Cacheable regardless of the value of the SCTLR_EL1.I bit.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
  - When EL2 is not implemented and EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

EOS, bit [11]

When FEAT_ExS is implemented:

Exception Exit is Context Synchronizing.

EOS	Meaning
0b0	An exception return from EL1 is not a context synchronizing event
0b1	An exception return from EL1 is a context synchronizing event

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

If SCTLR_EL1.EOS is set to 0b0:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL1.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL1 and ELR_EL1 on exception return is synchronized.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

EnRCTX, bit [10]

When FEAT_SPECRES is implemented:

Enable EL0 access to the following System instructions:

- CFPRCTX, DVPRCTX and CPPRCTX instructions.
- CFP RCTX, DVP RCTX and CPP RCTX instructions.

EnRCTX	Meaning
0b0	EL0 access to these instructions is disabled, and these instructions are trapped to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.
0b1	EL0 access to these instructions is enabled.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

UMA, bit [9]

User Mask Access. Traps EL0 execution of MSR and MRS instructions that access the PSTATE.{D, A, I, F} masks to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from AArch64 state only, reported using an ESR_ELx.EC value of 0x18.

UMA	Meaning
0b0	Any attempt at EL0 using AArch64 to execute an MRS, MSR(REGISTER), or MSR(IMMEDIATE) instruction that accesses the DAIF is trapped.
0b1	This control does not cause any instructions to be trapped.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SED, bit [8]

When EL0 is capable of using AArch32:

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

SED	Meaning
0b0	SETEND instruction execution is enabled at EL0 using AArch32.
0b1	SETEND instructions are UNDEFINED at EL0 using AArch32 and any attempt at EL0 to access a SETEND instruction generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, reported using an ESR_ELx.EC value of 0x00.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

ITD, bit [7]

When EL0 is capable of using AArch32:

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

ITD	Meaning
0b0	All IT instruction functionality is enabled at EL0 using AArch32.
0b1	Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED and generates an exception, reported using an ESR_EL1.EC value of 0x00, to EL1 or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1: • All encodings of the IT instruction with hw1[3:0]!=0b1000. All encodings of the subsequent instruction with the following values for hw1: – 0b11xxxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM. – 0b1011xxxxxxxxxxx: All instructions in 'Miscellaneous 16-bit instructions'. – 0b10100xxxxxxxxxx: ADD Rd, PC, #imm – 0b01001xxxxxxxxxx: LDR Rd, [PC, #imm] – 0b0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC. – 0b010001xx1xxxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers unpredictable cases with BLX Rn. These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block. It is IMPLEMENTATION DEFINED whether the IT instruction is treated as: • A 16-bit instruction, that can only be followed by another 16-bit instruction. • The first half of a 32-bit instruction. This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED. An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information, see Changes to an ITD control by an instruction in an IT block .

ITD is optional, but if it is implemented in the SCTLR_EL1 then it must also be implemented in the SCTLR_EL2, HSCTLR, and SCTLR.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement ITD, access to this field is **RAZ/WI**.

Otherwise:

RES1

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL1 and EL0 under certain conditions.

The following instructions generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH.

nAA	Meaning
0b0	Unaligned accesses by the specified instructions generate an Alignment fault.
0b1	This control does not generate Alignment faults.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

CP15BEN, bit [5]

When EL0 is capable of using AArch32:

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from EL0:

CP15BEN	Meaning
0b0	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED and generates an exception to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1. The exception is reported using an ESR_ELx.EC value of 0x00.
0b1	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR_EL1 then it must also be implemented in the SCTLR_EL2, HSCTLR, and SCTLR.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement CP15BEN, access to this field is **RAO/WI**.

Otherwise:

RES0

SA0, bit [4]

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL1 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Stage 1 Cacheability control, for data accesses.

C	Meaning
0b0	All data access to Stage 1 Normal memory from EL0 and EL1, and all Normal memory accesses from unified cache to the EL1&0 Stage 1 translation tables, are treated as Stage 1 Non-cacheable.
0b1	This control has no effect on the Stage 1 Cacheability of: • Data access to Normal memory from EL0 and EL1. • Normal memory accesses to the EL1&0 Stage 1 translation tables.

When the Effective value of the HCR_EL2.DC bit in the current Security state is 1, the PE ignores SCTLR_EL1.C. This means that EL0 and EL1 data accesses to Normal memory are Cacheable.

When FEAT_VHE is implemented, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
  - When EL2 is not implemented and EL2 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0.

A	Meaning
0b0	Alignment fault checking disabled when executing at EL1 or EL0. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.
0b1	Alignment fault checking enabled when executing at EL1 or EL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

If FEAT_MOPS is implemented, SETG* instructions have an alignment check regardless of the value of the A bit.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on execution at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL1&0 stage 1 address translation.

M	Meaning
0b0	EL1&0 stage 1 address translation disabled. See the SCTLR_EL1.I field for the behavior of instruction accesses to Normal memory.
0b1	EL1&0 stage 1 address translation enabled.

If the Effective value of HCR_EL2.{DC, TGE} in the current Security state is not {0, 0} then the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of the field.

When FEAT_VHE is implemented, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The reset behavior of this field is:

- On a Warm reset:
  - When EL2 is not implemented and EL3 is not implemented, this field resets to 0b0.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Accessing SCTLR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic SCTLR_EL1 or SCTLR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SCTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
1  if PSTATE.EL == EL0 then
2      UNDEFINED;
3  elsif PSTATE.EL == EL1 then
4      if EL2Enabled() && HCR_EL2.TRVM == '1' then
5          AArch64.SystemAccessTrap(EL2, 0x18);
6      elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
           ↪== '1') && HFGRTR_EL2.SCTLR_EL1 == '1' then
7          AArch64.SystemAccessTrap(EL2, 0x18);
8      elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
9          X[t, 64] = NVMem[0x110];
10     else
11         X[t, 64] = SCTLR_EL1;
12 elsif PSTATE.EL == EL2 then
13     if HCR_EL2.E2H == '1' then
```

```
        X[t, 64] = SCTLR_EL2;
    else
        X[t, 64] = SCTLR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SCTLR_EL1;
```

MSR SCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGWTR_EL2.SCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '11' then
        NVMem[0x110] = X[t, 64];
    else
        SCTLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SCTLR_EL2 = X[t, 64];
    else
        SCTLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    SCTLR_EL1 = X[t, 64];
```

MRS <Xt>, SCTLR_EL12

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        X[t, 64] = NVMem[0x110];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = SCTLR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        X[t, 64] = SCTLR_EL1;
    else
        UNDEFINED;
```

MSR SCTLR_EL12, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '101' then
        NVMem[0x110] = X[t, 64];
    elsif EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SCTLR_EL1 = X[t, 64];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        SCTLR_EL1 = X[t, 64];
    else
        UNDEFINED;
```

E3.2.17 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL2.

When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, these controls apply also to execution at EL0.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

This register has no effect if EL2 is not enabled in the current Security state.

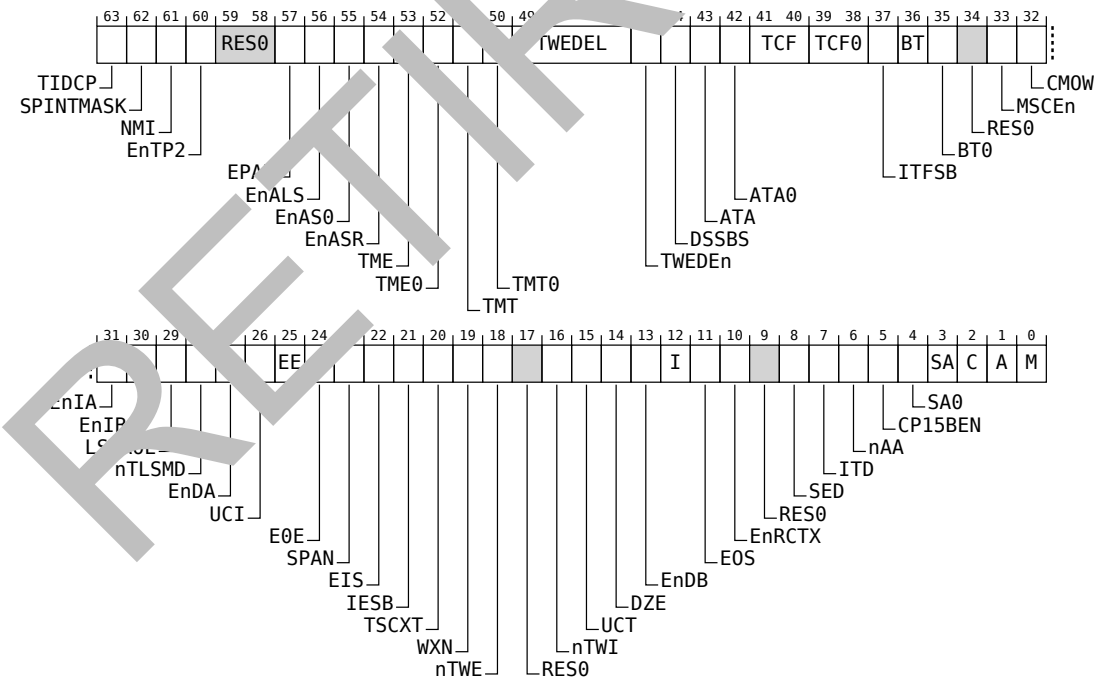
AArch64 system register SCTLR_EL2 bits [31:0] are architectually mapped to AArch32 system register HSCTLR[31:0].

Attributes

SCTLR_EL2 is a 64-bit register.

Field descriptions

The SCTLR_EL2 bit assignments are:



TIDCP, bit [63]

When FEAT_TIDCP1 is implemented and HCR_EL2.E2H == 1:

Trap IMPLEMENTATION DEFINED functionality. Traps EL0 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2.

TIDCP	Meaning
0b0	No instructions accessing the System register or System instruction spaces are trapped by this mechanism.
0b1	If HCR_EL2.TGE==0, no instructions accessing the System register or System instruction spaces are trapped by this mechanism. If HCR_EL2.TGE==1, instructions accessing the following System register or System instruction spaces are trapped to EL2 by this mechanism: • In AArch64 state, EL0 access to the encodings in the following reserved encoding spaces are trapped and reported using EC syndrome 0x18: – IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}. – IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name. • In AArch32 state, EL0 MCR and MRC access to the following encodings are trapped and reported using EC syndrome 0x03: – All coproc==p15, CRn==c9, opc1 == {0-7}, CRm == {c0-c2, c5-c8}, opc2 == {0-7}. – All coproc==p15, CRn==c10, opc1 =={0-7}, CRm == {c0, c1, c4, c8}, opc2 == {0-7}. – All coproc==p15, CRn==c11, opc1=={0-7}, CRm == {c0-c8, c15}, opc2 == {0-7}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SPINTMASK, bit [62]

When FEAT_NMI is implemented:

SP Interrupt Mask enable. When SCTLR_EL2.NMI is 1, controls whether PSTATE.SP acts as an interrupt mask, and controls the value of PSTATE.ALLINT on taking an exception to EL2.

SPINTMASK	Meaning
0b0	Does not cause PSTATE.SP to mask interrupts. PSTATE.ALLINT is set to 1 on taking an exception to EL2.
0b1	When PSTATE.SP is 1 and execution is at EL2, an IRQ or FIQ interrupt that is targeted to EL2 is masked regardless of any denotion of Superpriority. PSTATE.ALLINT is set to 0 on taking an exception to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

NMI, bit [61]

When FEAT_NMI is implemented:

Non-maskable Interrupt enable.

NMI	Meaning
0b0	This control does not affect interrupt masking behavior.
0b1	This control enables all of the following: • The use of the PSTATE.ALLINT interrupt mask. • IRQ and FIQ interrupts to have Superpriority as an additional attribute. • PSTATE.SP to be used as an interrupt mask.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

Otherwise:

RES0

EnTP2, bit [60]

When FEAT_SME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

EnTP2, bit [60]

Traps instructions executed at EL0 that access TPIDR2_EL0 to EL2 when EL2 is implemented and enabled for the current Security state. The exception is reported using ESR_ELx.EC value 0x18.

EnTP2	Meaning
0b0	This control causes execution of these instructions at EL0 to be trapped.
0b1	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

EnTP2, bit [60]

IGNORED.

Otherwise:

RES0

Bits [59:58]

Reserved, RES0.

EPAN, bit [57]

When FEAT_PAN3 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

EPAN, bit [57]

Enhanced Privileged Access Never. When PSTATE.PAN is 1, determines whether an EL2 data access to a page with EL0 instruction access permission generates a Permission fault as a result of the Privileged Access Never mechanism.

EPAN	Meaning
0b0	No additional Permission faults are generated by this mechanism.
0b1	An EL2 data access to a page with stage 1 EL0 data access permission or stage 1 EL0 instruction access permission generates a Permission fault. Any speculative data accesses that would generate a Permission fault as a result of PSTATE.PAN = 1 if the accesses were not speculative, will not cause an allocation into a cache.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_PAN3 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

EPAN, bit [57]

IGNORED.

Otherwise:

RES0

EnALS, bit [56]

When FEAT_LS64 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

EnALS, bit [56]

Traps execution of an LD64B or ST64B instruction at EL0 to EL2.

EnALS	Meaning
0b0	Execution of an LD64B or ST64B instruction at EL0 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an LD64B or ST64B instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000002.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LS64 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

EnALS, bit [56]

IGNORED.

Otherwise:

RES0

EnAS0, bit [55]

When FEAT_LS64_ACCDATA is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

EnAS0, bit [55]

Traps execution of an ST64BV0 instruction at EL0 to EL2.

EnAS0	Meaning
0b0	Execution of an ST64BV0 instruction at EL0 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV0 instruction is reported using an ESR_ELx.EC value of 0x0A, with an ISS code of 0x0000001.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LS64_ACCDATA is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

EnAS0, bit [55]

IGNORED.

Otherwise:

RES0

EnASR, bit [54]

When FEAT_LS64_V is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

EnASR, bit [54]

Traps execution of an ST64BV instruction at EL0 to EL2.

EnASR	Meaning
0b0	Execution of an ST64BV instruction at EL0 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

A trap of an ST64BV instruction is reported using an EC value of 0x0A, with an ISS code of 0x0000000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LS64_V is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

EnASR, bit [54]

IGNORED.

Otherwise:

RES0

TME, bit [34]

When FEAT_TME is implemented:

Enables the Transactional Memory Extension at EL2.

TME	Meaning
0b0	Any attempt to execute a TSTART instruction at EL2 is trapped, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL2.
0b1	This control does not cause any TSTART instruction to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TME0, bit [52]

When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

TME0, bit [52]

Enables the Transactional Memory Extension at EL0.

TME0	Meaning
0b0	Any attempt to execute a TSTART instruction at EL0 is trapped to EL2, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL0.
	This control does not cause any TSTART instruction to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

TME0, bit [52]

IGNORED.

Otherwise:

RES0

TMT, bit [51]

When FEAT_TME is implemented:

Forces a trivial implementation of the Transactional Memory Extension at EL2.

TMT	Meaning
0b0	This control does not cause any TSTART instruction to fail.
0b1	When the TSTART instruction is executed at EL2, the transaction fails with a TRIVIAL failure cause.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TMT0, bit [50]

When FEAT_TME is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

TMT0, bit [50]

Forces a trivial implementation of the Transactional Memory Extension at EL0.

TMT0	Meaning
0b0	This control does not cause any TSTART instructi[illegible]
0b1	Whe[illegible]e TSTART [illegible]truction is executed at EL0, [illegible] transaction [illegible]ls with a TRIVIAL [illegible]ailure c[illegible]e.

The reset behavior of this field is:

- On a Warm reset, this field resets to an archi[illegible]lly UN[illegible]OWN [illegible]ue.

When FEAT_TME is implemented, HCR_F[illegible].E2H = [illegible]1 and HC[illegible]_EL2.TGE == 0

TMT0, bit [50]

IGNORED.

Otherwise:

RES0

TWEDEL, bits [49[illegible]5]

When FEAT_TW[illegible]D is i[illegible]plemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

TWEDE[illegible][3:0] [illegible]bits [4[illegible]46]

TW[illegible] Delay. A [illegible]-bit unsigned number that, when SCTLR_EL2.TWEDEn is 1, encodes the minimum delay in takin[illegible] tra[illegible]f WFE [illegible]sed by SCTLR_EL2.nTWE as $2^{(TWEDEL + 8)}$ cycles.

The reset [illegible]havior of this field is:

- On a Wa[illegible]n reset, this field resets to an architecturally UNKNOWN value.

When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

TWEDEL, bits [3:0] of bits [49:46]

IGNORED.

Otherwise:

RES0

TWEDEn, bit [45]

When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

TWEDEn, bit [45]

TWE Delay Enable. Enables a configurable delayed trap of the WFE instruction caused by SCTLR_EL2.nTWE.

TWEDEn	Meaning
0b0	The delay for taking a WFE trap is IMPLEMENTATION DEFINED.
0b1	The delay for taking a WFE trap is at least the number of cycles defined in SCTLR_EL2.TWEDEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_TWED is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

TWEDEn, bit [45]

IGNORED.

Otherwise:

RES0

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL2.
0b1	PSTATE.SSBS is set to 1 on an exception to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

RES0

ATA, bit [43]

When FEAT_MTE2 is implemented:

Allocation Tag Access in EL2.

When SCR_EL3.ATA is 1, controls access to Allocation Tags and Tag Check operations in EL2.

ATA	Meaning
0b0	Access to Allocation Tags is prevented at EL2. Memory accesses at EL2 are not subject to a Tag Check operation.
0b1	This control does not prevent access to Allocation Tags at EL2. Tag Checked memory accesses at EL2 are subject to a Tag Check operation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ATA0, bit [42]

When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

ATA0, bit [42]

Allocation Tag Access in EL0.

When SCR_EL3.ATA is 1, controls access to Allocation Tags and Tag Check operations in EL0.

ATA0	Meaning
0b0	Access to Allocation Tags is prevented at EL0. Memory accesses at EL0 are not subject to a Tag Check operation.
0b1	This control does not prevent access to Allocation Tags at EL0. Tag Checked memory accesses at EL0 are subject to a Tag Check operation.

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

ATA0, bit [42]

IGNORED.

Otherwise:

RES0

TCF, bits [41:40]

When FEAT_MTE2 is implemented:

Tag Check Fault in EL2. Controls the effect of Tag Check Faults due to Loads and Stores in EL2.

TCF	Meaning	Applies
0b00	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	
0b10	Tag Check Faults are asynchronously accumulated.	
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	When FEAT_MTE3 is implemented

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TCF0, bits [39:38]

When FEAT_MTE3 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

TCF0, bits [1:0] of bits [39:38]

Tag Check Fault in EL0. Controls the effect of Tag Check Faults due to Loads and Stores in EL0.

TCF0	Meaning	Applies
0b00	Tag Check Faults have no effect on the PE.	
0b01	Tag Check Faults cause a synchronous exception.	
0b10	Tag Check Faults are asynchronously accumulated.	
0b11	Tag Check Faults cause a synchronous exception on reads, and are asynchronously accumulated on writes.	When FEAT_MTE3 is implemented

If FEAT_MTE3 is not implemented, the value 0b11 is reserved.

Software may change this control bit on a context switch.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_MTE2 is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

TCF0, bits [1:0] of bits [39:38]

IGNORED.

Otherwise:

RES0

ITFSB, bit [37]

When FEAT_MTE2 is implemented:

When synchronous exceptions are not being generated by Tag Check Faults, this field controls whether on exception entry into EL2, all Tag Check Faults due to instructions executed before exception entry, that are reported asynchronously, are synchronized into TFSRE0_EL1, TFSR_EL1 and TFSR_EL2 registers.

ITFSB	Meaning
0b0	Tag Check Faults are not synchronized on entry to EL2.
0b1	Tag Check Faults are synchronized on entry to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

BT, bit [36]

When FEAT_BTI is implemented

PAC Branch Type compatibility at EL2.

When HCR_EL2.{E2H, TGE} == {1, 1}, this bit is named BT1.

BT	Meaning
0b0	When the PE is executing at EL2, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
0b1	When the PE is executing at EL2, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

BT0, bit [35]

When FEAT_BTI is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

BT0, bit [35]

PAC Branch Type compatibility at EL0.

BT0	Meaning
0b0	When the PE is executing at EL0, PACIASP and PACIBSP are compatible with PSTATE.BTYPE == 0b11.
0b1	When the PE is executing at EL0, PACIASP and PACIBSP are not compatible with PSTATE.BTYPE == 0b11.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_BTI is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

BT0, bit [35]

IGNORED.

Otherwise:

RES0

Bit [34]

Reserved, RES0

MSCEn, bit [33]

When FEAT_MOPS is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

MSCEn, bit [33]

Memory Copy and Memory Set instructions Enable. Enables execution of the Memory Copy and Memory Set instructions at EL0.

MSCEn	Meaning
0b0	Execution of the Memory Copy and Memory Set instructions is UNDEFINED at EL0.
0b1	This control does not cause any instructions to be UNDEFINED.

When FEAT_MOPS is implemented and HCR_EL2.{E2H, TGE} is not {1, 1}, the Effective value of this bit is 0b1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_MOPS is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

MSCEn, bit [33]

IGNORED.

Otherwise:

RES0

CMOW, bit [32]

When FEAT_CMOW is implemented and HCR_EL2.E2H == 1:

Controls cache maintenance instruction permission for the following instructions executed at EL0.

- IC IVAU, DC CIVAC, DC CIGDVAC and DC CIGVAC.

CMOW	Meani
0b0	These instruc executed at EL0 with stage 1 re permission, but without stage 1 write ermissio o not generate a stage 1 permission ult.
0b1	If oled as a result of SCTLR_EL2.UCI==1, these instructions executed at EL0 with stage 1 read permission, but without stage 1 write permission, generate a stage 1 permission fault.

When HCR_EL2.TGE is 0 us bi s no effect o execution at EL0.

For this control, stage 1 as write perm ion if all of the following apply:

- AP[2] is 0 or BM is 1 in the stage 1 descriptor.
- Where AP ble is in se, APTable[1] is 0 for all levels of the translation table.

This bit is permitted e cached a TLB.

The re behav r of this s is:

- On a W this field resets to an architecturally UNKNOWN value.

Otherwi

RES0

EnIA, bit [31]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIAKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see System register control of pointer authentication .

EnIA	Meaning
0b0	Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is not enabled.

EnIA	Meaning
0b1	Pointer authentication (using the APIAKey_EL1 key) of instruction addresses is enabled.

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. Specifically, when the field is 1, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APIBKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see System register control of pointer authentication .

EnIB	Meaning
0b0	Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is not enabled.
0b1	Pointer authentication (using the APIBKey_EL1 key) of instruction addresses is enabled.

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. Specifically, when the field is 1, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

LSMAOE, bit [29]

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

LSMAOE, bit [29]

Load Multiple and Store Multiple Atomicity and Ordering Enable.

LSMAOE	Meaning
0b0	For all memory accesses at EL0, A32 and T32 Load Multiple and Store Multiple can have an interrupt taken during the sequence memory accesses, and the memory accesses are not required to be ordered.
0b1	The ordering and interrupt behavior of A32 and T32 Load Multiple and Store Multiple at EL0 is as defined for Armv8.0.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

LSMAOE, bit [29]

IGNORED.

Otherwise:

RES1

nTLSMD, bit [28]

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

nTLSMD, bit [28]

No Trap Load Multiple and Store Multiple to Device-nGRE/Device-nGnRE/Device-nGnRnE memory.

nTLSMD	Meaning
0b0	All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are trapped and generate a stage 1 Alignment fault.
0b1	All memory accesses by A32 and T32 Load Multiple and Store Multiple at EL0 that are marked at stage 1 as Device-nGRE/Device-nGnRE/Device-nGnRnE memory are not trapped.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LSMAOC is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

nTLSMD, bit [28]

IGNORED.

Otherwise:

RES1

EnDA, bit [27]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDAKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see System register control of pointer authentication.

EnDA	Meaning
0b0	Pointer authentication (using the APDAKey_EL1 key) of data addresses is not enabled.
0b1	Pointer authentication (using the APDAKey_EL1 key) of data addresses is enabled.

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. Specifically, when the field is 1, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When this field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

UCI, bit [26]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

UCI, bit [26]

Traps execution of cache maintenance instructions at EL0 to EL2, from AArch64 state only. This applies to DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU.

If FEAT_DPB2 is implemented, this trap also applies to DC CVADP.

If FEAT_MTE is implemented, this trap also applies to DC CIGVAC, DC CIGDVAC, DC CGVAC, DC CGDVAC, DC CGVAP, and DC CGDVAP.

If FEAT_DPB2 and FEAT_MTE are implemented, this trap also applies to DC CGVADP and DC CGDVADP.

UCI	Meaning
0b0	Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

If the Point of Coherency is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean, or clean and invalidate instruction that operates by VA to the point of coherency can be trapped when the value of this control is 1.

If the Point of Unification is before any level of data cache, it is IMPLEMENTATION DEFINED whether the execution of any data or unified cache clean by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

If the Point of Unification is before any level of instruction cache, it is IMPLEMENTATION DEFINED whether the execution of any instruction cache invalidate by VA to the Point of Unification instruction can be trapped when the value of this control is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

UCI, bit [26]

IGNORED.

Otherwise:

RES0

EE, bit [25]

Endianness of data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime.

EE	Meaning
0b0	Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are little-endian.
0b1	Explicit data accesses at EL2, stage 1 translation table walks in the EL2 or EL2&0 translation regime, and stage 2 translation table walks in the EL1&0 translation regime are big-endian.

If an implementation does not provide Big-endian support at Exception levels higher than EL0, this bit is RES0.

If an implementation does not provide Little-endian support at Exception levels higher than EL0, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

E0E, bit [24]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

E0E, bit [24]

Endianness of data accesses at EL0.

E0E	Meaning
0b0	Explicit data accesses at EL0 are little-endian.
0b1	Explicit data accesses at EL0 are big-endian.

If an implementation only supports Little-endian accesses at EL0, then this bit is RES0. This option is not permitted when SCTLR_EL1.EE is RES1.

If an implementation only supports Big-endian accesses at EL0, then this bit is RES1. This option is not permitted when SCTLR_EL1.EE is RES0.

This bit has no effect on the endianness of LDTR, LDTRH, LDTRSH, LDTRSW, STTR, and STTRH instructions executed at EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

E0E, bit [24]

IGNORED.

Otherwise:

RES0

SPAN, bit [23]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

SPAN, bit [23]

Set Privileged Access Never on taking an exception to EL2.

SPAN	Meaning
0b0	PSTATE.PAN is set to 1 on taking an exception to EL2.
0b1	The value of PSTATE.PAN is left unchanged on taking an exception to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

SPAN, bit [23]

IGNORED.

Otherwise:

RES1

EIS, bit [22]

When FEAT_ExS is implemented:

Exception entry is a context synchronization event.

EIS	Meaning
0b0	The taking of an exception to EL2 is not a context synchronization event.
0b1	The taking of an exception to EL2 is a context synchronization event.

If SCTLR_EL2.EIS is set to 0b0:

- Indirect writes to ESR_EL2, FAR_EL2, SPSR_EL2, ELR_EL2, and VPFAR_EL2 are synchronized on exception entry to EL2, so that a direct read of the register after exception entry sees the indirectly written value caused by the exception entry.
- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL2.EIS:

- Changes to the PSTATE information on entry to EL2.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

IESB, bit [21]

When FEAT_IESB is implemented:

Implicit Error Synchronization event enable.

IESB	Meaning
0b0	Disabled.
0b1	An implicit error synchronization event is added: • At each exception taken to EL2. • Before the operational pseudocode of each ERET instruction executed at EL2.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value

might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSX instruction taken to EL2 and before each DRPS instruction executed at EL2, in addition to the other cases where it is added.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TSCXT, bit [20]

When (FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented), HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

TSCXT, bit [20]

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.

TSCXT	Meaning
0b0	EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.
0b1	EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL2, and the SCXTNUM_EL0 value is treated as 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_CSV2_2 is not implemented, FEAT_CSV2_1p2 is not implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

Bit [0]

Reserved, RES

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented), HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

TSCXT, bit [20]

IGNORED.

Otherwise:

RES0

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 or EL2&0 translation regime, this bit can force all memory regions that are writable to be treated as XN.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	Any region that is writable in the EL2 or EL2&0 translation regime is forced to XN for accesses from software executing at EL2.

This bit applies only when SCTLR_EL2.M bit is set.

The WXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

nTWE, bit [18]

Traps execution of WFE instructions at EL0 to EL2, from both Execution states.

nTWE	Meaning
0b0	Any attempt to execute a WFE instruction at EL0 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFE instruction is only trapped if the instruction passes its condition code check.

Since WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

nTWE, bit [18]

IGNORED.

Otherwise:

RES1

Bit [17]

Reserved, RES0.

nTWI, bit [16]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

nTWI, bit [16]

Traps execution of WFI instructions at EL0 to EL2, from both Execution states.

nTWI	Meaning
0b0	Any attempt to execute a WFI instruction at EL0 is trapped to EL2, if the instruction would otherwise cause the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

In AArch32 state, the attempted execution of a conditional WFI instruction is only trapped if the instruction passes its condition code check.

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

nTWI, bit [16]

IGNORED.

Otherwise

RES1

UCT, bit [15]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL2, from AArch64 state only.

UCT	Meaning
0b0	Accesses to the CTR_EL0 from EL0 using AArch64 are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

UCT, bit [15]

IGNORED.

Otherwise:

RES0

DZE, bit [14]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

DZE, bit [14]

Traps execution of DC ZVA instructions at EL0 to EL2, from AArch64 state only.

If FEAT_MTE is implemented, this trap also applies to DC GVA and DC GZVA.

DZE	Meaning
0b0	Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped to EL2. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions that this trap applies to are not supported.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

DZE, bit [14]

IGNORED.

Otherwise:

RES0

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication (using the APDBKey_EL1 key) of instruction addresses in the EL2 or EL2&0 translation regime.

For more information, see System register control of pointer authentication .

EnDB	Meaning
0b0	Pointer authentication (using the APDBKey_EL1 key) of data addresses is not enabled.
0b1	Pointer authentication (using the APDBKey_EL1 key) of data addresses is enabled.

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. Specifically, when the field is 1, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When the field is 0, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

I, bit [12]

Instruction access Cacheability control, for access at EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H,TGE} == {1,1}, EL0.

	Meaning
0b0	All instruction accesses to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache. When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, all instruction accesses to Normal memory from EL0 are Non-cacheable for all levels of instruction and unified cache. If SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
0b1	This control has no effect on the Cacheability of instruction access to Normal memory from EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, instruction access to Normal memory from EL0. If the value of SCTLR_EL2.M is 0, instruction accesses from stage 1 of the EL2 or EL2&0 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL3 translation regime.

When EL2 is disabled in the current Security state or HCR_EL2.{E2H,TGE} != {1,1}, this bit has no effect on the EL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

EOS, bit [11]

When FEAT_ExS is implemented:

Exception exit is a context synchronization event.

EOS	Meaning
0b0	An exception return from EL2 is not a context synchronization event.
0b1	An exception return from EL2 is a context synchronization event.

If SCTLR_EL2.EOS is set to 0b0:

- Memory transactions, including instruction fetches, from an Exception level always use the translation resources associated with that translation regime.
- Exception Catch debug events are synchronous debug events.
- DCPS* and DRPS instructions are context synchronization events.

The following are not affected by the value of SCTLR_EL2.EOS:

- The indirect write of the PSTATE and PC values from SPSR_EL2 and ELR_EL2 on exception return is synchronized.
- Behavior of accessing the banked copies of the stack pointer using the SP register name for loads, stores, and data processing instructions.
- Exit from Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

EnRCTX, bit [10]

When FEAT_SPECRES is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

EnRCTX, bit [10]

Enable EL0 access to the following System instructions:

- CFPRCTX, DVPRCTX and CPPRCTX instructions.
- CFP RCTX, DVP RCTX and CPP RCTX instructions.

EnRCTX	Meaning
0b0	EL0 access to these instructions is disabled, and these instructions are trapped to EL1.
0b1	EL0 access to these instructions is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_SPECRES is implemented, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

EnRCTX, bit [10]

IGNORED.

Otherwise:

RES0

Bit [9]

Reserved, RES0.

SED, bit [8]

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

SED, bit [8]

SETEND instruction disable. Disables SETEND instructions at EL0 using AArch32.

SED	Meaning
0b0	SETEND instruction execution is enabled at EL0 using AArch32.
0b1	SETEND instructions are UNDEFINED at EL0 using AArch32.

If the implementation does not support mixed-endian operation at any Exception level, this bit is RES1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When EL0 can only use AArch64, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

Bit [0]

Reserved, RES1.

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

SED, bit [8]

IGNORED.

Otherwise:

RES0

ITD, bit [7]

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

ITD, bit [7]

IT Disable. Disables some uses of IT instructions at EL0 using AArch32.

ITD	Meaning
0b0	All IT instruction functionality is enabled at EL0 using AArch32.
0b1	Any attempt at EL0 using AArch32 to execute any of the following is UNDEFINED: • All encodings of the IT instruction with hw1[3:0]!=1000. • All encodings of the subsequent instruction with the following values for hw1: – 0b11xxxxxxxxxxxxxx: All 32-bit instructions, and the 16-bit instructions B, UDF, SVC, LDM, and STM. – 0b1011xxxxxxxxxxxx: All instructions in 'Miscellaneous 16-bit instructions'. – 0b10100xxxxxxxxxx: ADD Rd, PC, #imm – 0b01001xxxxxxxxxx: LDR Rd, [PC, #imm] – 0b0100x1xxx1111xxx: ADD Rdn, PC; CMP Rn, PC; MOV Rd, PC; BX PC; BLX PC. – 0b010001xx1xxxx111: ADD PC, Rm; CMP PC, Rm; MOV PC, Rm. This pattern also covers UNPREDICTABLE cases with BLX Rn. These instructions are always UNDEFINED, regardless of whether they would pass or fail the condition code check that applies to them as a result of being in an IT block. It is IMPLEMENTATION DEFINED whether the IT instruction is treated as: • A 16-bit instruction, that can only be followed by another 16-bit instruction. • The first half of a 32-bit instruction. This means that, for the situations that are UNDEFINED, either the second 16-bit instruction or the 32-bit instruction is UNDEFINED. An implementation might vary dynamically as to whether IT is treated as a 16-bit instruction or the first half of a 32-bit instruction.

If an instruction in an active IT block that would be disabled by this field sets this field to 1 then behavior is CONSTRAINED UNPREDICTABLE. For more information see Changes to an ITD control by an instruction in an IT block .

ITD is optional, but if it is implemented in the SCTLR_EL2 then it must also be implemented in the SCTLR_EL1, HSCTLR, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement ITD, access to this field is **RAZ/WI**.

When EL0 can only use AArch64, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

Bit [0]

Reserved, RES1.

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

ITD, bit [7]

IGNORED.

Otherwise:

RES0

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults under certain conditions at EL2, and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, EL0.

The following instructions generate an Alignment fault if all bytes being accessed are not within a single 16-byte quantity, aligned to 16 bytes for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STLUR, STLURH, and STLURH

nAA	Meaning
0b0	Unaligned accesses by the specified instructions generate an Alignment fault.
0b1	Unaligned accesses by the specified instructions do not generate an Alignment fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

CP15BEN, bit [5]

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

CP15BEN, bit [5]

System instruction memory barrier enable. Enables accesses to the DMB, DSB, and ISB System instructions in the (coproc==0b1111) encoding space from EL0:

CP15BEN	Meaning
0b0	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is UNDEFINED.
0b1	EL0 using AArch32: EL0 execution of the CP15DMB, CP15DSB, and CP15ISB instructions is enabled.

CP15BEN is optional, but if it is implemented in the SCTLR_EL2 then it must also be implemented in the SCTLR_EL1, HSCTLR, and SCTLR.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When EL0 can only use AArch64, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1

Bit [0]

Reserved, RES0.

When EL0 is capable of using AArch32, HCR_EL2.E2H == 1 and HCR_EL2.TGE == 0

CP15BEN, bit [5]

IGNORED.

Otherwise:

RES1

SA0, bit [4]

When HCR_EL2.E2H == 1 and HCR_EL2.TGE == 1:

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL2 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Data access Cacheability control, for accesses at EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, EL0

C	Meaning
0b0	The following are Non-cacheable for all levels of data and unified cache: • Data accesses to Normal memory from EL2. • When HCR_EL2.{E2H, TGE} != {1, 1}, Normal memory accesses to the EL2 translation tables. • When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}: – Data accesses to Normal memory from EL0. – Normal memory accesses to the EL2&0 translation tables.
0b1	This control has no effect on the Cacheability of: • Data access to Normal memory from EL2. • When HCR_EL2.{E2H, TGE} != {1, 1}, Normal memory accesses to the EL2 translation tables. • When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}: – Data accesses to Normal memory from EL0. – Normal memory accesses to the EL2&0 translation tables.

This bit has no effect on the EL3 translation regime.

When EL2 is disabled in the current Security state or HCR_EL2.{E2H, TGE} != {1, 1}, this bit has no effect on the EL2&0 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to 0b0.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2 and, when EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, EL0.

A	Meaning
0b0	Alignment fault checking disabled when executing at EL2. When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, alignment fault checking disabled when executing at EL0. Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.
0b1	Alignment fault checking enabled when executing at EL2. When EL2 is enabled in the current Security state and HCR_EL2.{E2H, TGE} == {1, 1}, alignment fault checking enabled when executing at EL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

If FEAT_MOPS is implemented, SETG* instructions have an alignment check regardless of the value of the A bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU enable for EL2 or EL2&0 stage 1 address translation.

M	Meaning
0b0	When HCR_EL2.{E2H, TGE} != {1, 1}, EL2 stage 1 address translation disabled. When HCR_EL2.{E2H, TGE} == {1, 1}, EL2&0 stage 1 address translation disabled. See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.
0b1	When HCR_EL2.{E2H, TGE} != {1, 1}, EL2 stage 1 address translation enabled. When HCR_EL2.{E2H, TGE} == {1, 1}, EL2&0 stage 1 address translation enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to `0b0`.

Accessing SCTLR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SCTLR_EL2 or SCTLR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SCTLR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SCTLR_EL2;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SCTLR_EL2;
```

MSR SCTLR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.NV == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    SCTLR_EL2 = X[t, 64];
elsif PSTATE.EL == EL3 then
    SCTLR_EL2 = X[t, 64];
```

MRS <Xt>, SCTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TRVM == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGRTR_EL2.SCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        X[t, 64] = NVMem[0x110];
    else
        X[t, 64] = SCTLR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        X[t, 64] = SCTLR_EL2;
    else
        X[t, 64] = SCTLR_EL1;
elsif PSTATE.EL == EL3 then
    X[t, 64] = SCTLR_EL1;
```

MSR SCTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn
        ↪== '1') && HFGWTR_EL2.SCTLR_EL1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
        NVMem[0x110] = X[t, 64];
    else
        SCTLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SCTLR_EL2 = X[t, 64];
    else
        SCTLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL3 then
    SCTLR_EL1 = X[t, 64];
```

E3.2.18 EDDEVID1, External Debug Device ID register 1

The EDDEVID1 characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

Attributes

EDDEVID1 is a 32-bit register.

Field descriptions

The EDDEVID1 bit assignments are:

Bits [31:8]

Reserved, RES0.

HSR, bits [7:4]

Indicates support for the External Debug Halt tus egister, HSR. Defined values are:

Hs	Meaning
b0000	EDHSR not implemented, and the PE follows behaviors consistent with all of the EDHSR fields having a zero value.
0b0001	EDHSR implemented.

All othe alue re rese ed.

Wh FEAT_D ugv8p2 is not implemented, the only permitted value is 0b0000.

PCSH t, bits [3:0]

Indicates the fset applied to PC samples returned by reads of EDPCSR. Permitted values of this field in Armv8 are:

PCSROffset	Meaning
0b0000	EDPCSR not implemented.
0b0010	EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register

space, as indicated by the value of PMU.PMDEVID.PCSample.

Accessing EDDEVID1

Accesses to this register use the following encodings in the external debug interface:

EDDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC4	EDDEVID1

This interface is accessible as follows:

- When FEAT_DoPD is not implemented or IsCorePowered() access to this register is **RO**.
- Otherwise access to this register returns an ERROR.

Chapter E4
Glossary terms

Effective Non-streaming SVE vector length

The Non-streaming SVE vector length in bits at the current Exception level, is an implementation-supported power of two up to the Maximum implemented Non-streaming SVE vector length, further constrained by ZCR_ELx at the current and higher Exception levels.

Effective Streaming SVE vector length

The Streaming SVE vector length in bits at the current Exception level is an implementation-supported power of two from 128 up to the Maximum implemented Streaming SVE vector length, further constrained by SMCR_ELx at the current and higher Exception levels.

Effective SVE vector length

The vector length in bits that applies to the execution of SVE instructions at the current Exception level is the Effective Streaming SVE vector length when the PE is in Streaming SVE mode, otherwise it is the Effective Non-streaming SVE vector length.

Illegal

Describes an implemented instruction whose attempted execution by a PE when `PSTATE.SM` and `PSTATE.ZA` are not in the required state causes an SME illegal instruction exception to be taken, unless its execution at the current Exception level is prevented by a higher priority configurable trap or enable.

Legal

Describes an implemented instruction that can be executed by a PE when `PSTATE.SM` and `PSTATE.ZA` are in the required state, unless its execution at the current Exception level is prevented by a configurable trap or enable.

Maximum implemented Non-streaming SVE vector length

The maximum Non-streaming SVE vector length in bits supported by the implementation.

Maximum implemented Streaming SVE vector length

The maximum Streaming SVE vector length in bits supported by the implementation.

NSVL

Effective Non-streaming SVE vector length.

Scalable Matrix Extension

Defines architectural state capable of holding two-dimensional matrix tiles, and a Streaming SVE mode which supports execution of SVE2 instructions with a vector length that matches the tile width, along with instructions that accumulate the outer product of two vectors into a tile, as well as load, store, and move instructions that transfer a vector to or from from a tile row or column.

Scalable Matrix Extension version 2

Extends the Scalable Matrix Extension by adding data-processing instructions with multi-vector operands and a multi-vector predication mechanism, a lookup table feature, a binary outer product instruction, and a range prefetch hint.

SMCU

Streaming Mode Compute Unit.

SME

Scalable Matrix Extension.

SME2

Scalable Matrix Extension version 2.

Streaming execution

Execution of instructions by a PE while that PE is in Streaming SVE mode.

Streaming Mode Compute Unit

Where more than one PE shares resources for Streaming execution of SVE and SME instructions, those shared resources are called the Streaming Mode Compute Unit (SMCU).

Streaming SVE mode

An execution mode that supports a substantial subset of the SVE2 instruction set and architectural state with a vector length that matches the width of SME tiles, which may be different from the vector length when the PE is not in Streaming SVE.

Streaming SVE register state

The registers *Z0-Z31*, *P0-P15*, and *FFR* that are accessed by SVE and SME instructions when the PE is in Streaming SVE mode.

SVL

Effective Streaming SVE vector length.

VL

Effective SVE vector length.

ZA array

A two-dimensional array of $[SVL_B \times SVL_B]$ bytes contained within the *ZA storage*.

ZA array vector

A vector that is SVL bits in size within the *ZA array*.

ZA storage

The architectural register state added by SME that is capable of holding two-dimensional matrix tiles.

ZA tile

A square, two-dimensional sub-array of the *ZA array*.

ZA tile slice

A one-dimensional set of horizontally or vertically contiguous elements within a *ZA tile*.

ZT0 register

The 512-bit architectural register added by SME2 that consists of up to sixteen 32-bit table entries, each containing an 8-bit, 16-bit, or 32-bit element.