

# ARM® Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for ARMv8-A



# ARM Architecture Reference Manual Supplement

## The Scalable Vector Extension (SVE), for ARMv8-A

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

### Release Information

The following changes have been made to this document.

Change History			
Date	Issue	Confidentiality	Change
31 March 2017	A.a	Non-Confidential Beta	Beta release
21 August 2017	A.b	Non-Confidential EAC	EAC release

### Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement specifically covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the ARM trademark usage guidelines <http://www.arm.com/about/trademarks/guidelines/index.php>.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.  
ARM Limited. Company 02557590 registered in England.  
110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20327

In this document, where the term ARM is used to refer to the company it means “ARM or any of its subsidiaries as appropriate”.

### Confidentiality Status

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

**Product Status**

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

**Web Address**

<http://www.arm.com>

**Limitations of A.b**

SVE mandates that the ARMv8.3-CompNum complex number support feature is available in all implemented instruction sets. This feature is not documented in this supplement but will be documented in a future release of the *ARM<sup>®</sup> Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.



# Contents

## ARM Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for ARMv8-A

	<b>Preface</b>	
	About this book .....	viii
	Using this book .....	ix
	Conventions .....	x
	Additional reading .....	xi
	Feedback .....	xii
<b>Chapter 1</b>	<b>Introduction</b>	
	1.1 About the SVE supplement .....	1-14
	1.2 About the Scalable Vector Extension .....	1-15
	1.3 Terminology .....	1-16
<b>Chapter 2</b>	<b>SVE Application Level Programmers' Model</b>	
	2.1 Registers .....	2-20
	2.2 Process state, PSTATE .....	2-24
	2.3 SVE half-precision floating-point support .....	2-26
<b>Chapter 3</b>	<b>SVE System Level Programmers' Model</b>	
	3.1 Exception model .....	3-28
	3.2 Configurable Vector length .....	3-29
<b>Chapter 4</b>	<b>Memory Model</b>	
	4.1 SVE Memory Model .....	4-32
<b>Chapter 5</b>	<b>SVE Instruction Set</b>	
	5.1 SVE assembler language .....	5-36

	5.2	Instruction set overview .....	5-37
<b>Chapter 6</b>		<b>System Registers</b>	
	6.1	System registers .....	6-64
<b>Chapter 7</b>		<b>SVE Debug</b>	
	7.1	Self-hosted debug .....	7-68
	7.2	External debug .....	7-69
<b>Chapter 8</b>		<b>SVE Performance Monitors Extension</b>	
	8.1	Introduction .....	8-72
	8.2	New performance monitor events .....	8-73
	8.3	Existing ARMv8-A PMU events affected by SVE .....	8-74
<b>Appendix A</b>		<b>Recommended SVE PMU events</b>	
	A.1	List of recommended PMU events .....	A-76
	A.2	Interesting combinations of SVE events .....	A-104
	A.3	Instruction categories .....	A-106

# Preface

This preface introduces the *ARM® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for ARMv8-A*. It contains the following sections:

- *About this book* on page viii.
- *Using this book* on page ix.
- *Conventions* on page x.
- *Additional reading* on page xi.
- *Feedback* on page xii.

## About this book

This book is the *ARM® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for ARMv8-A*. This book describes the changes and additions to the ARMv8-A AArch64 architecture that are introduced by SVE, and therefore must be read in conjunction with the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

## Using this book

This book is a supplement to the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* (DDI0487), and is intended to be used with it. The ARMv8-A ARM is the definitive source of information about ARMv8-A.

It is assumed that the reader is familiar with the ARMv8-A architecture.

This book is organized into the following chapters:

### Chapter 1 Introduction

Read this for an overview of the SVE extension and definitions of key terminology. It outlines the key features of SVE and introduces the terminology used to describe the extension.

### Chapter 2 SVE Application Level Programmers' Model

Read this for a description of the SVE Application Level Programmers' Model. This section must be read in conjunction with the section titled *The AArch64 Application Level Programmers' Model* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### Chapter 3 SVE System Level Programmers' Model

Read this for a description of the SVE System Level Programmers' Model. This section must be read in conjunction with the section titled *The AArch64 System Level Programmers' Model* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### Chapter 4 Memory Model

Read this for a description of the SVE Memory Model. This section must be read in conjunction with the sections titled *The AArch64 Application Level Memory Model* and *The AArch64 System Level Memory Model* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### Chapter 5 SVE Instruction Set

Read this for a description of the SVE Instruction Set Architecture. This section must be read in conjunction with the section titled *The AArch64 Instruction Set* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### Chapter 6 System Registers

Read this for a description of the new SVE System registers and the pre-existing AArch64 System registers that are modified by SVE. This section must be read in conjunction with the section titled *AArch64 System Register Descriptions* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### Chapter 7 SVE Debug

Read this for a description of the SVE additions to the ARMv8-A AArch64 Debug Architecture. This section must be read in conjunction with the sections titled *AArch64 Self-hosted Debug and Debug State* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### Chapter 8 SVE Performance Monitors Extension

Read this for a description of the SVE additions to the ARMv8-A AArch64 Performance Monitors Extension. This section must be read in conjunction with the section titled *The Performance Monitor Extension* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### Appendix A Recommended SVE PMU events

Read this for a list of the recommended PMU events for SVE and their descriptions.

## Conventions

The following sections describe conventions that this book can use:

- [Typographical conventions](#).
- [Numbers](#).

### Typographical conventions

The following table describes the typographical conventions:

Typographical conventions	
Style	Purpose
<i>italic</i>	Introduces special terminology, and denotes citations.
<b>bold</b>	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.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
SMALL CAPITALS	Used for a few terms that have specific technical meanings, and are included in the glossary.
Colored text	Indicates a link. This can be: <ul style="list-style-type: none"><li>• A URL, for example <a href="http://developer.arm.com">http://developer.arm.com</a>.</li><li>• A cross-reference, that includes the page number of the referenced information if it is not on the current page.</li><li>• A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term.</li></ul>

### 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`.

## Additional reading

This section lists relevant publications from ARM and third parties.

See [developer.arm.com](http://developer.arm.com), <http://developer.arm.com>, and the Infocenter, <http://infocenter.arm.com>, for access to ARM documentation.

### ARM publications

- *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* (ARM DDI 0487).
- *System Register XML for ARMv8.3.*
- *A64 ISA XML for ARMv8.3.*

## Feedback

ARM welcomes feedback on its documentation.

### Feedback on this book

If you have comments on the content of this book, send an e-mail to [errata@arm.com](mailto:errata@arm.com). Give:

- The title.
- The number, ARM DDI 0584A.b.
- The page numbers to which your comments apply.
- 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.

---

# Chapter 1

## Introduction

This chapter provides an introduction to the Scalable Vector Extension for the ARMv8-A architecture. This chapter contains the following sections:

- *About the SVE supplement* on page 1-14.
- *About the Scalable Vector Extension* on page 1-15.
- *Terminology* on page 1-16.

## 1.1 About the SVE supplement

This supplement must be read with the *ARM<sup>®</sup> Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*. Together, the manual and this supplement provide a full description of the ARMv8-A architecture, including the Scalable Vector Extension.

In general, this supplement describes only the architectural changes that are introduced by the Scalable Vector Extension.

This supplement does not contain any detailed instruction descriptions, pseudocode, or System register descriptions. Instead, this information is provided in a separate format, with links to this information appearing throughout the supplement.

## 1.2 About the Scalable Vector Extension

The *Scalable Vector Extension* (SVE) is an optional extension to the ARMv8-A architecture, with a base requirement of ARMv8.2-A. SVE complements and does not replace AArch64 Advanced SIMD and floating-point functionality. If SVE is implemented, all SVE instructions are mandatory and the ARMv8.2-FP16 half-precision floating-point and the ARMv8.3-CompNum complex number instructions must be implemented.

SVE is defined for the AArch64 Execution state only, and adds:

- Support for wide Vector and Predicate registers.
- A set of instructions that operate on wide vectors and predicates.
- Some minor additions to the configuration and identification registers.

The key features that SVE provides are:

- Scalable vector length. See [Vector registers on page 2-20](#).
- Predication. See [Predicate registers on page 2-21](#).
- Gather-load and scatter-store. See [Load, store, and prefetch instructions on page 5-37](#).
- Software-managed speculative vectorization. See [First Fault Register, FFR on page 2-22](#).

## 1.3 Terminology

The following is an alphabetical list of key terminology and phrases that are used throughout this supplement.

### Active element

An Active element is a Vector element or Predicate element that has been identified, by the value of the corresponding element of an instruction's Governing predicate being TRUE, as a source register element or destination register element to be used by the instruction. If an instruction is unpredicated, all of the Vector elements or Predicate elements are implicitly treated as active.

### Constructive instruction encoding

A Constructive instruction encoding is an instruction encoding where the destination register is encoded independently of the source registers.

### Destructive instruction encoding

A Destructive instruction encoding is an instruction encoding where one of the source registers is also used as the destination register.

### Element number

For a given element size of N bits, elements within a vector or Predicate register are numbered with element[0] always representing bits[(N-1):0], element[1] always representing bits[(2N-1):N], and so on, of the vector or Predicate register. See [Figure 2-1 on page 2-20](#) for more information.

### First active element

The First active element of a vector is defined as the first element, in order of increasing Element number, that is an Active element.

### First-fault load

SVE provides a First-fault option for some SVE vector load instructions. This option causes memory access faults to be suppressed if they do not occur as a result of the First active element of the vector. Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded. See [First Fault Register, FFR on page 2-22](#) for more information about the FFR.

### Gather-load

Gather-load is a mechanism that allows the elements of a vector to be read from non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode. See [Load, store, and prefetch instructions on page 5-37](#) for more information.

### Governing predicate

A Governing predicate defines the Active elements and Inactive elements of the source and destination registers for the corresponding instruction.

### Inactive element

An Inactive element is a Vector element or Predicate element that has been identified, by the value of the corresponding element of an instruction's Governing predicate being FALSE, as an unused source register element or destination register element for the associated instruction.

### Last active element

The Last active element of a vector or Predicate register is defined as the last element, in order of increasing Element number, that is an Active element.

### Merging predication

When a Predicated instruction specifies Merging predication, the Inactive elements of the destination register remain unchanged.

### Non-fault load

SVE provides a Non-fault option for some SVE vector load instructions. This option causes all memory access faults to be suppressed. Instead, the FFR is updated to indicate which of the active Vector elements were not successfully loaded. See [First Fault Register, FFR on page 2-22](#) for more information about the FFR.

### Predicated instruction

An instruction is said to be predicated if the instruction specifies a Governing predicate register.

### Predicate

A one-dimensional array of predicate elements of the same size. The predicate element size of 1, 2, 4, or 8 bits is specified independently by each instruction, and is one-eighth the size of the corresponding vector element.

### Predicate element

The lowest-numbered bit of each Predicate element holds the Boolean value of that element, where 1 represents TRUE and 0 represents FALSE.

### Predicate register

An SVE Predicate register, P0-P15, having a length that is a multiple of 16 bits, in the range 16 to 256, inclusive.

### Prefixed instruction

The instruction that immediately follows a MOVPRFX instruction in program order.

### Scalar base register

A Scalar base register refers to an AArch64 general-purpose register, X0-X30, or the current stack pointer, SP.

### Scalar index register

A Scalar index register refers to an AArch64 general-purpose register, X0-X30, or for certain instructions, XZR.

### Scatter-store

Scatter-store is a mechanism that allows the elements of a vector to be written to non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode. See [Load, store, and prefetch instructions on page 5-37](#) for more information.

### SIMD

Single Instruction, Multiple Data. A SIMD instruction performs the same operation on multiple Vector elements or Predicate elements in parallel.

### Speculative vectorization

Speculative vectorization refers to the process of performing a vector operation on one or more Vector elements before the termination condition of the unvectorized loop has been resolved.

### Vector

A one-dimensional array of vector elements of the same size and data type. The vector element size of 8, 16, 32, 64, or 128 bits, and the data type, is specified independently by each instruction.

### Vector length

The accessible width of the SVE Vector registers at the current Exception level, as constrained by the [ZCR\\_EL1](#), [ZCR\\_EL2](#), and [ZCR\\_EL3](#) System registers. All Vector registers at the same Exception level have the same Vector length. The accessible width of the SVE Predicate registers and FFR is one-eighth of the Vector length.

**Vector register**

An SVE Vector register, Z0-Z31, having a length that is a multiple of 128 bits, in the range 128 to 2048, inclusive.

**Zeroing predication**

When a Predicated instruction specifies Zeroing predication, the Inactive elements of the destination register are set to zero.

# Chapter 2

## SVE Application Level Programmers' Model

This chapter introduces the SVE Application Level Programmers' model. This chapter contains the following sections:

- [Registers on page 2-20.](#)
- [Process state, PSTATE on page 2-24.](#)
- [SVE half-precision floating-point support on page 2-26.](#)

## 2.1 Registers

### 2.1.1 Vector registers

SVE includes 32 scalable Vector registers, Z0-Z31. The SVE Vector registers are all of equal size, where the size is an IMPLEMENTATION DEFINED multiple of 128 bits, up to an architectural maximum of 2048 bits. Each Vector register can be subdivided into a number of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit Vector elements. The Vector element size for a given instruction is encoded in the opcode of the instruction. If the order in which operations are performed on Vector elements has observable significance, then the Vector elements must be processed in order of increasing Element number.

Bits[127:0] of the SVE Vector registers, Z0-Z31, are shared with the AArch64 SIMD&FP registers, V0-V31, so that Vn maps to Zn[127:0], as shown in Figure 2-1. If the accessible SVE Vector length at the current Exception level is greater than 128 bits, then any AArch64 instruction that writes to V0-V31 sets all the accessible bits above bit[127] of the corresponding SVE Vector register to zero. See [Configurable Vector length on page 3-29](#) for more information.

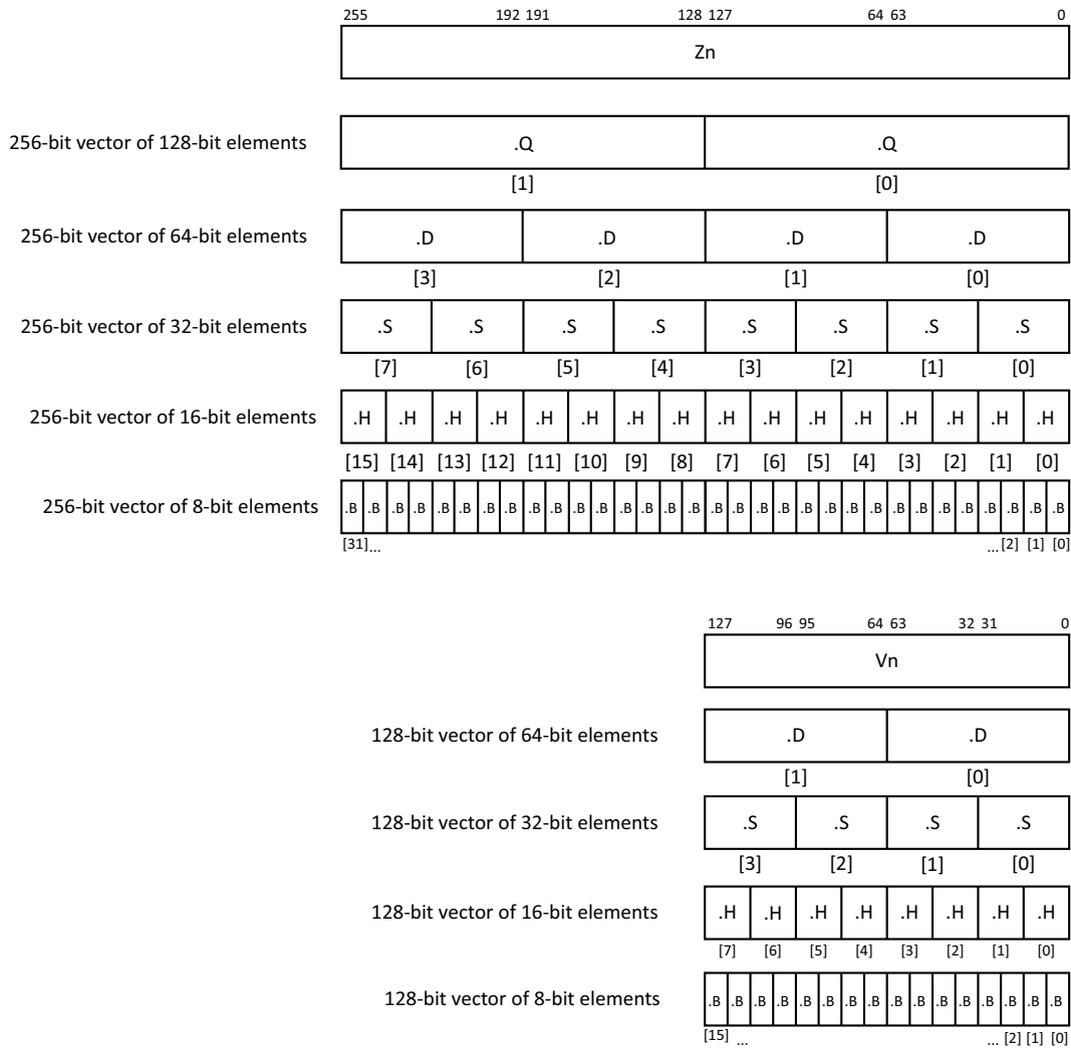


Figure 2-1 SVE and SIMD&FP vectors in AArch64 state

## 2.1.2 Predicate registers

SVE includes 16 scalable Predicate registers, P0-P15. Each Predicate register holds one bit per byte of a Vector register, meaning that each Predicate register is one-eighth of the size of a Vector register. Therefore, each Predicate register is an IMPLEMENTATION DEFINED multiple of 16 bits. Each Predicate register can be subdivided into a number of 1, 2, 4, or 8-bit elements, where each Predicate element corresponds to a Vector element. If the lowest-numbered bit of a Predicate element has a value of 1, the Predicate element is TRUE, otherwise it is FALSE. For all instructions other than those listed in [Predicate permute on page 5-60](#), the other bits of the Predicate element are IGNORED on reads and set to zero on writes. A Predicate element value with zeroes in all bits except the lowest-numbered bit is said to be in canonical form.

See [Predicate operations on page 5-51](#) for descriptions of instructions that operate on Predicate registers.

### Governing predicate

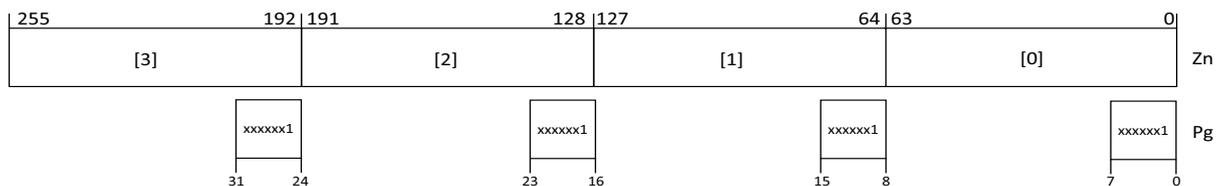
Where an instruction supports predication, it is known as a Predicated instruction. The Predicate register that is used to determine the Active elements of a Predicated instruction is known as the Governing predicate for that instruction. Predicated instructions can only use P0-P7 as the Governing predicate.

When a Governing predicate element is TRUE, then the corresponding vector or predicate register element is Active and is processed by the instruction, otherwise it is Inactive and takes no part in the operation of the instruction.

When a Predicated instruction writes to a vector or predicate result register, either:

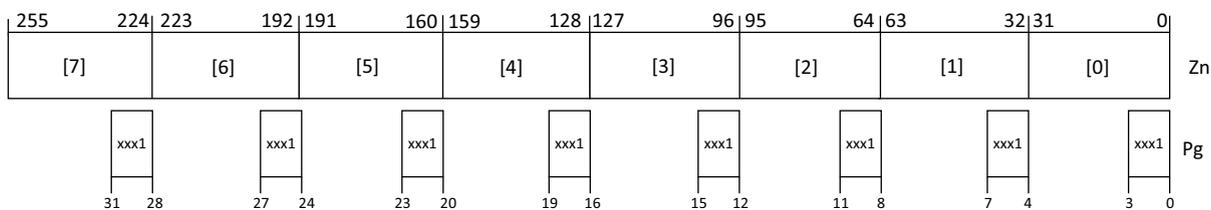
- The Inactive elements in the destination vector are set to zero. This is known as Zeroing predication.
- The Inactive elements in the destination vector retain their previous value. This is known as Merging predication.

[Figure 2-2](#) and [Figure 2-3](#) show the relationship between a 256-bit implementation of an SVE Vector register,  $Z_n$ , and the associated 32-bit Governing predicate register,  $P_g$ . [Figure 2-2](#) shows an SVE Vector register of four 64-bit elements, with an associated Governing predicate register of four 8-bit elements. In this case, the lowest-numbered bit of each Predicate element is 1, indicating that all elements of the Vector register are Active.



**Figure 2-2 256-bit vector, 4x64-bit packed elements**

[Figure 2-3](#) shows an SVE Vector register of eight 32-bit elements, with an associated Governing predicate register of eight 4-bit elements. In this case, the lowest-numbered bit of each Predicate element is 1, indicating that all elements of the Vector register are Active.



**Figure 2-3 256-bit vector, 8x32-bit packed elements**

Figure 2-4 shows a property of SVE predicates that allows a Governing predicate register to be interpreted differently when used for different Vector element sizes. As defined in *Predicate registers* on page 2-21, an SVE Predicate register contains one bit per byte of the corresponding SVE Vector register and the Predicate elements are numbered to match the equivalent Vector elements.

Figure 2-4 shows Za, a 256-bit vector of 32-bit elements, where the values of the 4-bit Governing predicate elements indicate that the even-numbered Vector elements are Active and the odd-numbered Vector elements are Inactive. When the same Governing predicate register is used for Zb, a 256-bit vector of 64-bit elements, Pg is interpreted differently. Now, the values of the 8-bit Governing predicate elements indicate that all 64-bit elements of Zb are Active.

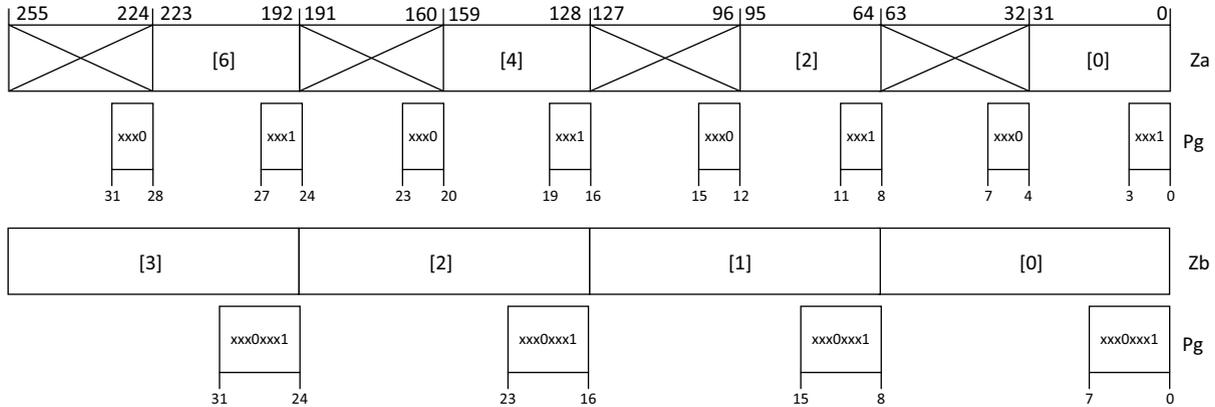


Figure 2-4 Governing predicate interpretation for different vector organizations

### 2.1.3 First Fault Register, FFR

The First Fault Register, FFR, is a dedicated register that captures the cumulative fault status of a sequence of SVE First-fault and Non-fault vector load instructions. The format of the FFR is the same as the Predicate registers. Bits in FFR are initialized to TRUE using the [SETFFR](#) instruction, and are indirectly cleared to FALSE as a result of an unsuccessful load of the corresponding, or lower-numbered, Vector element. After a sequence of one or more SVE First-fault or Non-fault loads that follow a [SETFFR](#) instruction, the FFR contains a sequence of zero or more TRUE elements followed by zero or more FALSE elements. Bits in the FFR are never set to TRUE as a result of a vector load instruction, therefore the TRUE elements in the FFR indicate the shortest sequence of consecutive elements that could contain valid data loaded from memory.

The only instructions that can read FFR are:

- [RDFFR](#)
- [RDFFRS](#)

The only instructions that can directly write FFR are:

- [WRFFR](#)
- [SETFFR](#)

The FFR is a Special-purpose register as defined in the section titled *Special-purpose registers* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*. This means that all direct and indirect reads and writes to the FFR appear to occur in program order relative to other instructions, without the need for explicit synchronization.

### 2.1.4 Scalar registers

Certain SVE instructions generate a scalar result that is written to an AArch64 general-purpose register or to element[0] of a Vector register. If an SVE instruction generates a scalar result that is narrower than the maximum destination register width, the upper bits of the destination register are set to zero.

---

**Note**

See *Registers in AArch64 Execution state* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* for more information about the AArch64 general-purpose and SIMD&FP registers.

---

## 2.2 Process state, PSTATE

SVE overloads the AArch64 PSTATE condition flags. See the section titled *Process state, PSTATE* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* for more information about the PSTATE condition flags. The condition flags can be set either by an explicit test of a Predicate register or based on the result of an SVE predicate-generating and flag-setting instruction. Where present, a Governing predicate determines which Predicate elements are to be tested. Some instructions use the condition flags to signal different events, but the most common SVE interpretations of the condition flags are shown in [Table 2-1](#).

**Table 2-1 SVE condition flags**

Flag	SVE Name	SVE Interpretation
N	First	Has a value of 1 if the First active element was TRUE, otherwise has a value of 0.
Z	None	Has a value of 1 if no Active element was TRUE, otherwise has a value of 0.
C	Not last	Has a value of 0 if the Last active element was TRUE, otherwise has a value of 1.
V	-	Cleared to 0 by the SVE flag-setting instructions, except <a href="#">CTERMEO</a> and <a href="#">CTERMNE</a> .

The SVE assembler syntax defines a new set of condition code aliases. The condition code aliases and their associated meanings are described in [Table 2-2](#).

**Table 2-2 Predicate condition flags**

Condition test	AArch64 name	SVE alias	SVE interpretation
Z == 1	EQ	NONE	No Active elements were TRUE
Z == 0	NE	ANY	An Active element was TRUE
C == 1	HS/CS	NLAST	The Last active element was not TRUE
C == 0	LO/CC	LAST	The Last active element was TRUE
N == 1	MI	FIRST	The First active element was TRUE
N == 0	PL	NFRST	The First active element was not TRUE
C == 1 && Z == 0	HI	PMORE	An Active element was TRUE but not the Last active element
C == 0    Z == 1	LS	PLAST	The Last active element was TRUE or none were TRUE
V == 1	VS	-	CTERM comparison failed, but end of partition reached
V == 0	VC	-	CTERM comparison succeeded, or end of partition not reached
N == V	GE	TCONT	CTERM termination condition not detected
N != V	LT	TSTOP	CTERM termination condition detected

———— **Note** —————

For Predicated instructions, it is the Governing predicate for the instruction that determines the Active and Inactive elements in the predicate source and destination registers. Any unpredicated SVE flag-setting instructions have an implicit Governing predicate, with all elements set to TRUE. This means that all elements in the predicate source and destination registers are considered Active for the purpose of setting the condition flags.

		[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	0						
15																
Pd.H	x	0	x	1	x	0	x	1	x	1	x	1	x	0	x	0
Pg	x	0	x	0	x	0	x	1	x	1	x	1	x	1	x	0
15																0

**Figure 2-5 First active element**

In [Figure 2-5](#), Pd.H is a Predicate register containing eight 2-bit Predicate elements and Pg is the Governing predicate. In this case, element[1] of Pd.H is the First active element, not element[0].

## 2.3 SVE half-precision floating-point support

SVE inherits the following behaviors from ARMv8.2-FP16:

- The half-precision instructions are subject to the same floating-point exception traps and enables as apply to the equivalent SVE single-precision or double-precision instructions.
- FPCR.FZ has no effect on the half-precision instructions.
- FPCR.FZ16 enables Flush-to-zero mode for all of the half-precision instructions, but not for conversions between half-precision and single or double-precision.
- A half-precision value that is flushed to zero as a result of FPCR.FZ16 will not generate an Input Denormal exception that sets FPSR.IDC to 1.

SVE half-precision floating-point instructions support only IEEE 754-2008 half-precision format and ignore the value of the FPCR.AHP bit, behaving as if it has an Effective value of 0.

# Chapter 3

## SVE System Level Programmers' Model

This chapter introduces the SVE System Level Programmers' model. This chapter contains the following sections:

- [Exception model](#) on page 3-28
- [Configurable Vector length](#) on page 3-29

## 3.1 Exception model

SVE adds hierarchical trap and enable controls at EL3, EL2, and EL1. These controls are implemented using the following System register fields:

- [CPTR\\_EL3.EZ](#).
- [CPTR\\_EL2.TZ](#), when [HCR\\_EL2.E2H](#) == 0.
- [CPTR\\_EL2.ZEN](#), when [HCR\\_EL2.E2H](#) == 1.
- [CPACR\\_EL1.ZEN](#).

### 3.1.1 SVE exception class

SVE defines the 0b011001 Exception class value, in [ESR\\_ELx.EC](#). The 0b011001 value is reported for exceptions that are due to attempted execution of SVE instructions and MRS/MSR instructions that access the ZCR\_ELx System registers when trapped due to the controls described in *Exception model*.

See [ESR\\_ELx](#) for more details.

### 3.1.2 SVE floating-point exception traps

SVE floating-point instructions only generate floating-point exceptions in response to floating-point operations performed on Active elements, but these are otherwise consistent with the behaviors described in the section titled *Floating-point exceptions and exception traps* in the *ARM<sup>®</sup> Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### 3.1.3 MOVPRFX exception behavior

For detailed information about the SVE [MOVPRFX](#) (predicated) and [MOVPRFX](#) (unpredicated) instructions, see *Move prefix on page 5-60*. When a MOVPRFX instruction is used as described in *Move prefix on page 5-60* and execution of the pair of instructions generates a synchronous exception or causes entry to Debug state, then the restart address that is recorded in [ESR\\_ELx](#) or DLR\_EL0 is a CONSTRAINED UNPREDICTABLE choice of:

- If the MOVPRFX instruction has caused no change to the architectural state, the address of the MOVPRFX instruction.
- Otherwise, the address of the Prefixed instruction.

Irrespective of the address that is recorded in [ESR\\_ELx](#) or DLR\_EL0, if the Prefixed instruction generates an Instruction Abort due to an MMU fault or synchronous External abort and the MOVPRFX does not generate an Instruction Abort, then the appropriate ESR\_ELx, FAR\_ELx, or HPFAR\_EL2 registers will record the syndrome information and address that is associated with the erroneous Prefixed instruction fetch and not the MOVPRFX instruction fetch. If both instruction fetches would cause an Instruction abort, then the address of the MOVPRFX instruction is recorded in the appropriate FAR\_ELx register.

### 3.1.4 Synchronous memory faults

See *Memory fault handling on page 4-33* and *Constrained unpredictable memory accesses on page 4-34* for more information on SVE fault handling behavior.

### 3.1.5 Asynchronous exception behavior

It is IMPLEMENTATION DEFINED whether SVE instructions can be interrupted by asynchronous exceptions. An interrupted SVE instruction on return from an asynchronous exception will restart and cannot resume.

## 3.2 Configurable Vector length

Privileged Exception levels can use the [ZCR\\_EL1.LEN](#), [ZCR\\_EL2.LEN](#), and [ZCR\\_EL3.LEN](#) System register fields to constrain the Vector length that is accessible by that Exception level and by less privileged Exception levels. See *SVE System registers* on page 6-65 for more information about the ZCR\_ELx registers. SVE requires that an implementation must allow the Vector length to be constrained to any power of two that is less than the maximum implemented Vector length, but also permits an implementation to allow the Vector length to be constrained to multiples of 128 that are not a power of two. It is IMPLEMENTATION DEFINED which of the permitted multiples of 128 are supported. See [Table 3-1](#) for more information.

If an unsupported Vector length is requested in ZCR\_ELx, the implementation is required to select the largest supported Vector length that is less than the requested length. This does not alter the value of ZCR\_ELx.LEN.

**Table 3-1 Configurable Vector lengths**

Maximum configurable Vector length	Additionally configurable Vector lengths	
	Required	Permitted
128	-	-
256	128	-
384	128, 256	-
512	128, 256	384
640	128, 256, 512	384
768	128, 256, 512	384, 640
896	128, 256, 512	384, 640, 768
1024	128, 256, 512	384, 640, 768, 896
1152	128, 256, 512, 1024	384, 640, 768, 896
1280	128, 256, 512, 1024	384, 640, 768, 896, 1152
1408	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280
1536	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408
1664	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408, 1536
1792	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664
1920	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664, 1792
2048	128, 256, 512, 1024	384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664, 1792, 1920



# Chapter 4

## Memory Model

This chapter introduces the changes to the ARMv8-A memory model introduced by SVE. This chapter contains the following sections:

- *Atomicity* on page 4-32.
- *Alignment support* on page 4-32.
- *Endian support* on page 4-32.
- *Memory ordering* on page 4-32.
- *Memory fault handling* on page 4-33.
- *Device memory* on page 4-34
- *Constrained unpredictable memory accesses* on page 4-34

## 4.1 SVE Memory Model

### 4.1.1 Atomicity

SVE vector loads and stores are performed as a sequence of element accesses.

Further to the rules relating to the atomicity of SIMD loads and stores in the section titled *Atomicity in the ARM architecture* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*, the following behaviors are specific to accesses generated by SVE loads and stores:

- For predicated SVE vector element or structure loads, where an element address is aligned to the size of the element in memory, that access is treated as a single-copy atomic read.
- For predicated SVE vector element or structure stores, where an element address is aligned to the size of the element in memory, that access is treated as a single-copy atomic write.
- Unpredicated loads and stores of a Vector or Predicate register are regarded as a stream of byte accesses. Single-copy atomicity of any access that is larger than a byte, within the series of byte accesses, is not guaranteed by the architecture.

### 4.1.2 Alignment support

Further to the rules relating to alignment of SIMD loads and stores in the sections titled *Alignment Support* and *Memory types and attributes* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*, the following behaviors are specific to accesses generated by SVE loads or stores when alignment checking is enabled. Alignment checking is enabled when SCTL<sub>R</sub>\_EL<sub>x</sub>.A has a value of 1 at the current Exception level or because the access is to any type of Device memory:

- For predicated SVE vector element and structure loads and stores:
  - Alignment checks are based on the memory element access size, not the Vector element size. See [Table 5-1 on page 5-37](#) for more information on SVE memory element access sizes.
  - Inactive elements cannot cause an Alignment fault.
- For unpredicated SVE Vector register loads or stores, the base address is checked for 16-byte alignment.
- For unpredicated SVE Predicate register loads or stores, the base address is checked for 2-byte alignment.

Where an SVE load or store uses the current stack pointer, SP, as the base address, and stack alignment checking is enabled in SCTL<sub>R</sub>\_EL<sub>x</sub> at the current Exception level, then the stack pointer is checked for 16-byte alignment even when there are no Active elements to be transferred.

### 4.1.3 Endian support

Further to the rules relating to the byte and element order of SIMD loads and stores in the section titled *Data endianness* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*, the following behaviors are specific to accesses generated by SVE loads and stores:

- For predicated SVE vector element and structure loads and stores, the data size that is used for endianness conversions is the memory element access size, not the Vector element size.
- For unpredicated SVE Vector register loads and stores, the vector is treated as containing byte elements that are transferred in increasing Element number order without any endianness conversion.
- For unpredicated SVE Predicate register loads and stores, the predicate is treated as if each 8 predicate bits in increasing Element number order are held in a byte that is transferred without any endianness conversion.
- The endian conversion for SVE loads occurs before any sign-extension or zero-extension into a Vector element. For SVE stores, the endian conversion occurs after any truncation from the Vector element to the memory element access size.

### 4.1.4 Memory ordering

The ARMv8 memory model described in the section titled *Definition of the ARMv8 memory model* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* is relaxed for reads and writes generated by SVE load and store instructions as follows:

- An address dependency between two reads generated by SVE vector load instructions does not contribute to the Dependency-ordered-before relation.

- For a given observer, a pair of reads from the same location is not required to satisfy the internal visibility requirement if at least one of the reads was generated by an SVE load instruction.
- A single SVE vector store instruction that generates multiple writes to the same location ensures that those writes appear in the Coherence order for that location, in order of increasing Vector element number. No other ordering restrictions apply to memory effects generated by the same SVE vector store instruction.

For all SVE instructions that load or store Vector registers or Predicate registers, there is no requirement for the memory system beyond the PE to be able to identify the size of the elements that are accessed by that load or store instruction, and, except for multiple writes to the same location, the order in which elements and registers are accessed is not architecturally defined. This applies to accesses to Normal memory and accesses to Device memory. See the section titled *Memory types and attributes* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* for more information.

#### 4.1.5 Memory fault handling

The following mechanism is used to report a synchronous fault that is generated by a memory access that was performed as a result of an SVE load or store instruction. That is:

- The appropriate [ESR\\_ELx](#).EC field is updated with 0b100100 or 0b100101, depending on the Exception level from which the fault occurred.
- Depending on the Exception level handling the fault, the FAR\_EL1, FAR\_EL2, FAR\_EL3, or HPFAR\_EL2 System register is updated with the lowest address applicable to the Active element that the fault is reported against.
- A Data Abort exception is then taken.

Where multiple faults arise from different addresses that are generated by the same instruction, the architecture does not prioritize between the different faults.

The SVE load and store instructions can generate a sequence of accesses that might not be completed as a result of an exception being taken during that sequence of accesses. On return from such an exception, a load or store instruction that has not been architecturally executed is restarted, meaning that one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location that has been changed between the accesses, or that might be sensitive to the number of accesses.

SVE vector load and store instructions that generate a fault obey the sections titled *Definition of a precise exception* and *Effect of Data Aborts* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*, with the following SVE-specific modifications:

- For SVE predicated vector stores, memory locations that are associated with Active elements that do not generate a fault are set to an UNKNOWN value. Memory locations that are associated with Inactive elements, or with Active elements that do generate a fault, are preserved.
- For SVE predicated vector loads that are not a First-fault or Non-fault load, all elements in the destination Vector registers return an UNKNOWN value, irrespective of any predication, unless the destination register is a Vector register that is also used as a base or index register by the instruction, in which case the original value of the register is preserved.
- For SVE Non-fault vector loads, an exception is not taken and the ESR\_ELx exception syndrome and FAR\_ELx and HPFAR\_EL2 fault address registers are not updated. When a Non-fault memory element access generates a fault or is suppressed for any other reason, the FFR Predicate elements starting from that element number, up to and including the highest-numbered element, are set to FALSE. Since an FFR Predicate element is never set to TRUE by an SVE vector load instruction, the fault indications are cumulative. Following execution of an SVE Non-fault vector load, each destination Vector element contains either:
  - An UNKNOWN value, if the corresponding FFR element is FALSE.
  - The value read from memory, if the corresponding FFR element is TRUE and it was an Active element.
  - Zero, if the corresponding FFR element is TRUE and it was an Inactive element.
- For SVE First-fault vector loads, memory accesses due to the First active element are handled in the same way as for SVE predicated vector loads. If a memory access due to the First active element does not generate a fault, then the other elements are handled in the same way as for an SVE Non-fault vector load.

---

**Note**

---

The term *fault* in this section includes any of the following results of a data access performed as a result of the execution of SVE vector load and store instructions:

- MMU fault.
- Synchronous External abort, including synchronous parity or ECC error.
- Watchpoint debug event

Furthermore, an implementation is permitted to suppress the read of any Active element for an SVE Non-fault vector load, and any Active element other than the First active element for an SVE First-fault vector load.

---

#### 4.1.6 Device memory

For accesses by SVE instructions to a memory location with any Device memory type, the following additional exceptions apply:

- SVE vector prefetch instructions are guaranteed not to access Device memory.
- SVE Non-fault vector load instructions are guaranteed not to access Device memory, and an attempt by any Active element to access Device memory is suppressed and reported in the FFR as described in [Memory fault handling on page 4-33](#).
- SVE First-fault vector load instructions can access Device memory only for the First active element. If that access does not generate a fault, then an attempt by any other Active element to access Device memory is suppressed and reported in the FFR as described in [Memory fault handling on page 4-33](#).
- Hardware speculation of data accesses performed to a Device memory location is not permitted by the architecture, with the following exceptions:
  - Explicit reads that are performed by an SVE load instruction are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a naturally-aligned 64-byte window that contains at least 1 byte that is explicitly accessed by the instruction.
  - Explicit reads that are performed by an SVE non-temporal load instruction from memory locations with the Gathering attributes are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a naturally-aligned 128-byte window that contains at least 1 byte that is explicitly accessed by the instruction.

#### 4.1.7 CONSTRAINED UNPREDICTABLE memory accesses

The sections titled *Crossing a page boundary with different memory types or Shareability attributes* and *Crossing a peripheral boundary with a Device access* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* define CONSTRAINED UNPREDICTABLE behaviors associated with memory accesses due to loads and stores. These behaviors also apply to SVE vector loads and stores as follows:

- An SVE unpredicated contiguous load or store instruction has the described CONSTRAINED UNPREDICTABLE behaviors if it accesses an address range that crosses a boundary between memory types, Shareability attributes, or peripherals.
- An SVE predicated contiguous load or store instruction has the described CONSTRAINED UNPREDICTABLE behaviors only if there are accesses associated with Active elements on both sides of a boundary between different memory types, Shareability attributes, or peripherals.
- An SVE predicated non-contiguous gather-load or scatter-store instruction has the described CONSTRAINED UNPREDICTABLE behaviors only if there is a memory access associated with any Active element that crosses a boundary between different memory types, Shareability attributes, or peripherals.
- Memory addresses associated with Inactive elements cannot trigger CONSTRAINED UNPREDICTABLE behaviors.

# Chapter 5

## **SVE Instruction Set**

This chapter introduces the SVE instruction set. This chapter contains the following sections:

- [SVE assembler language on page 5-36.](#)
- [Instruction set overview on page 5-37.](#)

## 5.1 SVE assembler language

The SVE assembler language extends the A64 assembler language, as described in the section titled *Structure of the A64 assembler language* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*, as follows:

- SVE Vector register names Z0–Z31 and Predicate register names P0–P15 are added.
- The number of elements in a Vector or Predicate register is not specified as part of a Vector register shape qualifier. For example, Z1.B is used rather than V1.16B.
- An element size qualifier is not required for the Governing predicate, Pg, except in the cases where the element size cannot be inferred from the source and destination element sizes. However, an assembler must accept a Predicate element size qualifier, if provided, and check it for consistency with the other operands.
- Where appropriate, Predicated instructions must indicate whether the inactive destination Vector elements are to undergo Zeroing predication or Merging predication. The type of predication is indicated by use of a qualifier suffix to the Governing predicate, where:
  - Pg/Z indicates Zeroing predication.
  - Pg/M indicates Merging predication.

Some instructions identify Active and Inactive elements, but do not write to a destination Vector register. For these instructions, the Governing predicate operand is used with no zeroing or merging qualifier.

- Many SVE instructions have Destructive instruction encodings. To avoid ambiguity, the assembler language frequently uses a constructive notation for these instructions, where the destination register is repeated in the appropriate source operand position.
- The AArch64 load/store address syntax is extended to allow for Vector register operands within the address specifier. See [Load, store, and prefetch instructions on page 5-37](#) for more information.
- A set of SVE aliases is defined for the AArch64 condition codes. See [Table 2-2 on page 2-24](#) for further details.

## 5.2 Instruction set overview

### 5.2.1 Introduction

SVE adds a set of instructions to the existing ARMv8-A A64 instruction set. For details on the A64 instruction set, see the section titled *The A64 instruction set* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*. For a detailed listing of the instructions that are introduced by SVE, see the [SVE instruction index](#). The SVE instructions break down into the following functional groups:

- [Load, store, and prefetch instructions](#).
- [Integer operations](#).
- [Vector address calculation](#).
- [Bitwise operations](#).
- [Floating-point operations](#).
- [Predicate operations](#).
- [Move operations](#).
- [Reduction operations](#).

The following sections provide an overview of these functional groups. For detailed information on each instruction, see the individual instruction descriptions.

### 5.2.2 Load, store, and prefetch instructions

SVE vector load and store instructions transfer data in memory to or from elements of one or more vector or predicate transfer registers. SVE also includes vector prefetch instructions that provide read and write hints to the memory system.

For SVE predicated load, store, and prefetch instructions, the memory element access size and type that is associated with each Vector element is specified by a suffix to the instruction mnemonic, independently of the element size of the transfer registers. For example, LD1SH. [Table 5-1](#) shows the supported instruction suffixes for SVE load, store, and prefetch instructions.

**Table 5-1 SVE memory element access instruction suffixes**

Instruction suffix	Memory element access size and type
B	Unsigned byte
H	Unsigned halfword or half-precision floating-point
W	Unsigned word or single-precision floating-point
D	Unsigned doubleword or double-precision floating-point
SB	Signed byte
SH	Signed halfword
SW	Signed word

The element size of the transfer registers is always greater than or equal to the memory element access size. When the element size of the transfer registers is strictly greater than the memory element access size, then these are referred to as unpacked data accesses. In the case of unpacked data accesses:

- For load instructions, each element access is sign-extended or zero-extended to fill the Vector element, according to its size and type in [Table 5-1](#).
- For store instructions, each Vector element is truncated to the element access size.

Where the Vector element size and the memory element access size are the same, then these are referred to as packed data accesses. Signed access types are not supported for packed data accesses. Packed and unpacked access sizes and types relate to the Vector element size of the transfer registers, as defined in [Table 5-2](#).

**Table 5-2 Relationship between Vector element size and memory element access size and type**

Vector element	Packed access suffix	Unpacked access suffixes
.B	B	-
.H	H	B, SB
.S	W	H, SH, B, SB
.D	D	W, SW, H, SH, B, SB

**Note**

For gather-load and scatter-store instructions, the vector element size can only be .S or .D. This means that any non-contiguous memory element access of less than a word is unpacked. Non-contiguous memory element accesses of a word can be either packed or unpacked, depending on the Vector element size.

Load, store, and prefetch instructions consist of the following:

- [Predicated single vector contiguous element accesses](#).
- [Predicated multiple vector contiguous structure load/store on page 5-39](#).
- [Predicated non-contiguous element accesses on page 5-40](#).
- [Predicated replicating element loads on page 5-40](#).
- [Unpredicated vector register load/store on page 5-41](#).
- [Unpredicated predicate register load/store on page 5-41](#).

All predicated load instructions zero the Inactive elements of the destination vector, except for Non-fault loads and First-fault loads when the corresponding FFR element is FALSE.

Prefetch instructions provide hints to hardware and do not change architectural state. Therefore, a Governing predicate for a prefetch instruction provides an additional hint which indicates the memory locations to be prefetched. Prefetch instructions require an additional <prfop> operand to be specified. SVE prefetch instructions support all of the <prfop> listed in the section titled *Prefetch memory* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*, except for the PLI prefetch operand types.

Load, store, and prefetch instructions that multiply a Scalar index register or an index Vector element by the memory element access size specify a shift type, followed by a shift amount in bits. The shift type can be one of LSL, SXTW, or UXTW. The shift amount is always  $\text{Log}_2$  of the memory element access size, in bytes. The shift amount defaults to zero when the memory element access size is a byte. The shift type of LSL must be omitted if the shift amount is omitted.

SVE load, store, and prefetch instructions do not support pre-indexed or post-indexed addressing.

**Predicated single vector contiguous element accesses**

Predicated contiguous load and store instructions access memory locations starting from an address that is defined by a Scalar base register plus either:

- A Scalar index register.
- An immediate index value that is in the range -8 to 7, inclusive. This defaults to zero if omitted.

Prefetch instructions address contiguous memory locations in a similar manner, with the index being either:

- A Scalar index register.
- An immediate index value that is in the range of -32 to 31, inclusive. This defaults to zero if omitted.

For this group of instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of Vector elements, irrespective of predication, and then multiplied by the memory element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.
- The Scalar index register value is multiplied by the memory element access size in bytes. The index value is incremented by one after each element access, but the Scalar index register is not updated by the instruction.
- Load, LD1, and store, ST1, instructions support both packed and unpacked data accesses, with a Scalar index register or an immediate index value.
- First-fault load, LDFF1, instructions support both packed and unpacked data accesses, with a Scalar index register that defaults to XZR if omitted.
- Non-fault load, LDNF1, instructions support both packed and unpacked data accesses, with an immediate index value.
- Non-temporal load, LDNT1, and store, STNT1, instructions support only packed data accesses, with a Scalar index register or an immediate index value.
- Prefetch, PRF, instructions support only packed data accesses, with a Scalar index register or an immediate index value.
- When alignment checking is enabled for loads and stores, the value of the base address register must be aligned to the memory element access size.

Supported addressing modes	Assembler syntax
Scalar base + scalar index	[<Xn SP>, <Xm>{, LSL #<sh>}]
Scalar base + immediate index	[<Xn SP>{, #<simm>, MUL VL}]

### Predicated multiple vector contiguous structure load/store

Structure load, LD2, LD3, LD4, instructions read N consecutive memory locations to the same-numbered element in each of the N vector transfer registers, where N = 2, 3, or 4, respectively. Structure store, ST2, ST3, ST4, instructions write from the same-numbered element in each of the N consecutive vector transfer registers to N consecutive memory locations. The starting address is defined by a Scalar base register plus either:

- A Scalar index register.
- An immediate index that is a multiple of N, in the range  $-8 \times N$  to  $7 \times N$ , inclusive. This defaults to zero if omitted.

For this group of instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of Vector elements, irrespective of predication, and then multiplied by the element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.
- The Scalar index register value is multiplied by the element access size in bytes. Following each element access, the index value is incremented by one but the instruction does not update the Scalar index register.
- Each Predicate element applies to a single structure in memory, or equivalently to the same Element number within each of the two, three, or four transferred Vector registers.
- These instructions support packed data accesses only.
- When alignment checking is enabled for loads and stores, the base address must be aligned to the element access size.

Supported addressing modes	Assembler syntax
Scalar base + scalar index	[<Xn SP>, <Xm>{, LSL #<sh>}]
Scalar base + immediate index	[<Xn SP>{, #<simm>, MUL VL}]

### Predicated non-contiguous element accesses

Predicated non-contiguous element accesses address non-contiguous memory locations that are specified by either:

- A Scalar base register plus a vector of indices or offsets.
- A vector of base addresses plus an immediate byte offset. The immediate byte offset is a multiple of the memory element access size, in the range 0 to 31 times the memory element access size, inclusive, and defaults to zero if omitted.

For this group of instructions:

- Vector registers used as part of the address must specify a Vector element size of 32 bits or 64 bits, .S or .D. For load and store instructions, the transfer register must specify the same Vector element size.
- If the index Vector register contains 32-bit index values then the lowest 32 bits of each index Vector element can either be zero-extended or sign-extended to 64 bits.
- For load and store instructions, the index Vector elements are then optionally multiplied by the memory element access size, in bytes, if a shift amount is specified. For prefetch instructions the index Vector elements are always multiplied by the memory element access size, in bytes.
- Load, LD1, store, ST1, and First-fault load, LDFF1, instructions support packed and unpacked data accesses. Prefetch, PRF, instructions only specify the memory element access size.
- When alignment checking is enabled for loads and stores, the computed virtual address of each element must be aligned to the memory element access size.

<b>Assembler syntax</b>		
<b>Supported addressing modes</b>	<b>64-bit elements</b>	<b>32-bit elements</b>
Scalar base + 64-bit vector index	[<Xn SP>, <Zm>.D{, LSL #<sh>}]	-
Scalar base + 32-bit vector index	[<Xn SP>, <Zm>.D, (S U)XTW{ #<sh>}]	[<Xn SP>, <Zm>.S, (S U)XTW{ #<sh>}]
Vector base + immediate offset	[<Zn>.D{, #<uimm>}]	[<Zn>.S{, #<uimm>}]

### Predicated replicating element loads

The load and replicate instructions read one or more contiguous memory locations starting from an address that is defined by a Scalar base register plus either:

- A Scalar index register.
- An immediate byte offset. This defaults to zero if omitted.

For this group of instructions:

- The LD1R instructions load a single element value and replicate it into all Active elements of the destination vector. These instructions support packed and unpacked data accesses. These instructions use an immediate byte offset that is a multiple of the memory element access size, in the range 0 to 63 times the memory element access size, inclusive.
- The LD1RQ instructions load a predicated 128-bit quadword vector segment from contiguous element values and replicate that segment into all segments of the destination vector. These instructions support only packed data accesses. These instructions can use a Scalar index register that is multiplied by the memory element access size, or an immediate byte offset that is a multiple of 16, in the range of -128 to 112, inclusive.
- When alignment checking is enabled for loads and stores, the base address must be aligned to the memory element access size.

<b>Supported addressing modes</b>	<b>Assembler syntax</b>
Scalar base + scalar index	[<Xn SP>, <Xm>{, LSL #<sh>}]
Scalar base + immediate offset	[<Xn SP>{, #<imm>}]

### Unpredicated vector register load/store

The unpredicated Vector register load, LDR, and store, STR, instructions transfer a single Vector register from or to memory locations that are specified by a Scalar base register plus an immediate index value that is in the range -256 to 255, inclusive. The immediate index value defaults to zero if omitted. For this group of instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the current Vector register length in bytes.
- The data transfer is performed as a contiguous stream of byte accesses in ascending element order, without endianness conversion.
- When alignment checking is enabled for loads and stores, the base address must be 16-byte aligned.

Supported addressing mode	Assembler syntax
Scalar base + immediate index	[<Xn SP>{, #<sim>, MUL VL}]

### Unpredicated predicate register load/store

The unpredicated Predicate register load, LDR, and store, STR, instructions transfer a single Predicate register from or to memory locations that are specified by a Scalar base register plus an immediate index value that is in the range -256 to 255, inclusive. The immediate index value defaults to zero if omitted. For this group of instructions:

- The immediate index value is a predicate index, not an element index. The immediate index value is multiplied by the current Predicate register length, in bytes.
- The data transfer is performed as a contiguous stream of byte accesses, each byte containing 8 consecutive predicate bits, in ascending bit and element order, without endian conversion.
- When alignment checking is enabled for loads and stores, the base address must be 2-byte aligned.

Supported addressing mode	Assembler syntax
Scalar base + immediate index	[<Xn SP>{, #<sim>, MUL VL}]

## 5.2.3 Vector move operations

### Element move and broadcast

These instructions copy data from scalar registers, immediate values, and other vectors to selected Vector elements. The copied data might be in an integer or floating-point format.

**Table 5-3 Element move and broadcast instructions**

Mnemonic	Instruction	See
CPY	Copy signed integer immediate to Vector elements	<a href="#">CPY</a>
	Copy general-purpose register to Vector elements	<a href="#">CPY</a>
	Copy SIMD&FP scalar register to Vector elements	<a href="#">CPY</a>
DUP	Broadcast signed immediate to Vector elements	<a href="#">DUP</a>
	Broadcast general-purpose register to Vector elements	<a href="#">DUP</a>
FCPY	Copy 8-bit floating-point immediate to Vector elements	<a href="#">FCPY</a>
FDUP	Broadcast 8-bit floating-point immediate to Vector elements	<a href="#">FDUP</a>
FMOV	Move floating-point +0.0 to Vector elements (unpredicated)	<a href="#">FMOV</a>
	Move floating-point +0.0 to Vector elements (predicated)	<a href="#">FMOV</a>

**Table 5-3 Element move and broadcast instructions (continued)**

Mnemonic	Instruction	See
	Move 8-bit floating-point immediate to Vector elements (unpredicated)	<a href="#">FMOV</a>
	Move 8-bit floating-point immediate to Vector element (predicated)	<a href="#">FMOV</a>
MOV	Move signed integer immediate to Vector elements (unpredicated)	<a href="#">MOV</a>
	Move signed integer immediate to Vector elements (predicated)	<a href="#">MOV</a>
	Move general-purpose register to Vector elements (unpredicated)	<a href="#">MOV</a>
	Move general-purpose register to Vector elements (predicated)	<a href="#">MOV</a>
	Move SIMD&FP scalar register to Vector elements (unpredicated)	<a href="#">MOV</a>
	Move SIMD&FP scalar register to Vector elements (predicated)	<a href="#">MOV</a>
	Move Vector register (unpredicated)	<a href="#">MOV</a>
	Move Vector register (predicated)	<a href="#">MOV</a>
SEL	Select Vector elements from two vectors	<a href="#">SEL</a>

## 5.2.4 Integer operations

The following instructions operate on signed or unsigned integer data within a vector.

### Integer arithmetic

These instructions perform arithmetic operations on a source vector containing integer element values, and for binary operations, either a second source vector of integer values, or an immediate value.

**Table 5-4 Integer arithmetic instructions**

Mnemonic	Instruction	See
ABS	Absolute value	<a href="#">ABS</a>
ADD	Add vectors (predicated)	<a href="#">ADD</a>
	Add vectors (unpredicated)	<a href="#">ADD</a>
	Add immediate	<a href="#">ADD</a>
CNOT	Logically invert Boolean condition	<a href="#">CNOT</a>
MAD	Multiply-add, writing to the multiplicand register	<a href="#">MAD</a>
MLA	Multiply-add, writing to the addend register	<a href="#">MLA</a>
MLS	Multiply-subtract, writing to the addend register	<a href="#">MLS</a>
MSB	Multiply-subtract, writing to the multiplicand register	<a href="#">MSB</a>
MUL	Multiply by immediate	<a href="#">MUL</a>
	Multiply vectors	<a href="#">MUL</a>
NEG	Negate	<a href="#">NEG</a>
SABD	Signed absolute difference	<a href="#">SABD</a>
SDIV	Signed divide	<a href="#">SDIV</a>
SDIVR	Signed reverse divide	<a href="#">SDIVR</a>

Table 5-4 Integer arithmetic instructions (continued)

Mnemonic	Instruction	See
SMAX	Signed maximum with immediate	<a href="#">SMAX</a>
	Signed maximum vectors	<a href="#">SMAX</a>
SMIN	Signed minimum with immediate	<a href="#">SMIN</a>
	Signed minimum vectors	<a href="#">SMIN</a>
SMULH	Signed multiply returning high half	<a href="#">SMULH</a>
SQADD	Signed saturating add immediate	<a href="#">SQADD</a>
	Signed saturating add vectors	<a href="#">SQADD</a>
SQSUB	Signed saturating subtract immediate	<a href="#">SQSUB</a>
	Signed saturating subtract vectors	<a href="#">SQSUB</a>
SUB	Subtract immediate	<a href="#">SUB</a>
	Subtract vectors (predicated)	<a href="#">SUB</a>
	Subtract vectors (unpredicated)	<a href="#">SUB</a>
SUBR	Reversed subtract from immediate	<a href="#">SUBR</a>
	Reversed subtract vectors	<a href="#">SUBR</a>
SXTB	Signed byte extend	<a href="#">SXTB</a>
SXTH	Signed halfword extend	<a href="#">SXTH</a>
SXTW	Signed word extend	<a href="#">SXTW</a>
UABD	Unsigned absolute difference	<a href="#">UABD</a>
UDIV	Unsigned divide	<a href="#">UDIV</a>
UDIVR	Unsigned reversed divide	<a href="#">UDIVR</a>
UMAX	Unsigned maximum with immediate	<a href="#">UMAX</a>
	Unsigned maximum vectors	<a href="#">UMAX</a>
UMIN	Unsigned minimum with immediate	<a href="#">UMIN</a>
	Unsigned minimum vectors	<a href="#">UMIN</a>
UMULH	Unsigned multiply returning high half	<a href="#">UMULH</a>
UQADD	Unsigned saturating add immediate	<a href="#">UQADD</a>
	Unsigned saturating add vectors	<a href="#">UQADD</a>
UQSUB	Unsigned saturating subtract immediate	<a href="#">UQSUB</a>
	Unsigned saturating subtract vectors	<a href="#">UQSUB</a>
UXTB	Unsigned byte extend	<a href="#">UXTB</a>
UXTH	Unsigned halfword extend	<a href="#">UXTH</a>
UXTW	Unsigned word extend	<a href="#">UXTW</a>

### Integer dot product

The integer partial dot product instructions delimit the source vectors into groups of four 8-bit or 16-bit integer elements, called quadruplets. Within each group of four elements, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting widened products are summed and added to the 32-bit or 64-bit element of the accumulator and destination vector that aligns with the group of four elements in the first source vector.

The indexed forms of these instructions specify a single, numbered, group of four elements within each 128-bit segment of the second source vector as the multiplier for all the groups of four elements within the corresponding 128-bit segment of the first source vector.

**Table 5-5 Integer dot product instructions**

Mnemonic	Instruction	See
SDOT	Signed dot product by vector	<a href="#">SDOT</a>
	Signed dot product by indexed quadtuplets	<a href="#">SDOT</a>
UDOT	Unsigned dot product by vector	<a href="#">UDOT</a>
	Unsigned dot product by indexed quadtuplets	<a href="#">UDOT</a>

### Integer comparisons

These instructions compare Active elements in the first source vector with the corresponding elements in a second vector or with an immediate value. The Boolean result of each comparison is placed in the corresponding element of the destination predicate. Inactive elements in the destination Predicate register are set to zero. All integer comparisons set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero.

The wide element variants of the compare instructions allow a packed vector of narrower elements to be compared with wider 64-bit elements. These instructions treat the second source vector as having a fixed 64-bit doubleword element size and compare each narrow element of the first source vector with the corresponding vertically-aligned wide element of the second source vector. For example, if the first source vector contained 8-bit byte elements, then 8-bit element[0] to element[7] of the first source vector are compared with 64-bit element[0] of the second source vector, 8-bit element[8] to element[15] with 64-bit element[1], and so on. All 64 bits of the wide elements are significant for the comparison, with the narrow elements being sign-extended or zero-extended to 64 bits as appropriate for the type of comparison.

**Table 5-6 Integer comparison instructions**

Mnemonic	Instruction	See
CMPEQ	Compare signed equal to immediate	<a href="#">CMPEQ</a>
	Compare signed equal to wide elements	<a href="#">CMPEQ</a>
	Compare signed equal to vector	<a href="#">CMPEQ</a>
CMPGE	Compare signed greater than or equal to immediate	<a href="#">CMPGE</a>
	Compare signed greater than or equal to wide elements	<a href="#">CMPGE</a>
	Compare signed greater than or equal to vector	<a href="#">CMPGE</a>
CMPGT	Compare signed greater than immediate	<a href="#">CMPGT</a>
	Compare signed greater than wide elements	<a href="#">CMPGT</a>
	Compare signed greater than vector	<a href="#">CMPGT</a>
CMPHI	Compare unsigned higher than immediate	<a href="#">CMPHI</a>
	Compare unsigned higher than wide elements	<a href="#">CMPHI</a>
	Compare unsigned higher than vector	<a href="#">CMPHI</a>
CMPHS	Compare unsigned higher than or same as immediate	<a href="#">CMPHS</a>
	Compare unsigned higher than or same as wide elements	<a href="#">CMPHS</a>
	Compare unsigned higher than or same as vector	<a href="#">CMPHS</a>
CMPLE	Compare signed less than or equal to immediate	<a href="#">CMPLE</a>
	Compare signed less than or equal to wide elements	<a href="#">CMPLE</a>
	Compare signed less than or equal to vector	<a href="#">CMPLE</a>
CMPLO	Compare unsigned lower than immediate	<a href="#">CMPLO</a>
	Compare unsigned lower than 64-bit wide elements	<a href="#">CMPLO</a>

Table 5-6 Integer comparison instructions (continued)

Mnemonic	Instruction	See
	Compare unsigned lower than vector	<a href="#">CMPLO</a>
CMPLS	Compare unsigned lower or same as immediate	<a href="#">CMPLS</a>
	Compare unsigned lower or same as wide elements	<a href="#">CMPLS</a>
	Compare unsigned lower or same as vector	<a href="#">CMPLS</a>
CMPLT	Compare signed less than immediate	<a href="#">CMPLT</a>
	Compare signed less than wide elements	<a href="#">CMPLT</a>
	Compare signed less than vector	<a href="#">CMPLT</a>
CMPNE	Compare not equal to immediate	<a href="#">CMPNE</a>
	Compare not equal to wide elements	<a href="#">CMPNE</a>
	Compare not equal to vector	<a href="#">CMPNE</a>

### 5.2.5 Vector address calculation

These instructions compute vectors of addresses and addresses of vectors. This includes instructions to add a multiple of the current Vector length or Predicate register length, in bytes, to a general-purpose register.

The [ADR](#) instruction is an integer arithmetic operation that is used to calculate a vector of 64-bit or 32-bit addresses. The destination register elements are computed by the addition of the corresponding elements in the source registers, with an optional sign or zero extension and optional bitwise left shift of 1-3 bits applied to the final operands. This can be considered as the addition of a vector base and a scaled vector index.

32-bit addresses are computed by the addition of a 32-bit base and a scaled 32-bit unsigned index.

64-bit addresses are computed by one of:

- Addition of a 64-bit base and a scaled 64-bit unsigned index.
- Addition of a 64-bit base and a scaled, zero-extended 32-bit index.
- Addition of a 64-bit base and a scaled, sign-extended 32-bit index.

Table 5-7 Vector address calculation instructions

Mnemonic	Instruction	See
ADDVL	Add multiple of Vector length, in bytes, to scalar register	<a href="#">ADDVL</a>
ADDPL	Add multiple of Predicate register length, in bytes, to scalar register	<a href="#">ADDPL</a>
ADR	Compute vector of addresses	<a href="#">ADR</a>
RDVL	Read multiple of Vector register length, in bytes, to scalar register	<a href="#">RDVL</a>

## 5.2.6 Bitwise operations

### Bitwise logical operations

These instructions perform bitwise logical operations on vectors. Where operations are unpredicated, the operations are independent of the element size.

**Table 5-8 Bitwise logical operations**

Mnemonic	Instruction	See
AND	Bitwise AND vectors (predicated)	<a href="#">AND</a>
	Bitwise AND vectors (unpredicated)	<a href="#">AND</a>
	Bitwise AND with immediate	<a href="#">AND</a>
BIC	Bitwise clear with vector (predicated)	<a href="#">BIC</a>
	Bitwise clear with vector (unpredicated)	<a href="#">BIC</a>
	Bitwise clear using immediate	<a href="#">BIC</a>
DUPM	Broadcast bitmask immediate to vector (unpredicated)	<a href="#">DUPM</a>
EON	Bitwise exclusive OR with inverted immediate	<a href="#">EON</a>
EOR	Bitwise exclusive OR vectors (predicated)	<a href="#">EOR</a>
	Bitwise exclusive OR vectors (unpredicated)	<a href="#">EOR</a>
	Bitwise exclusive OR with immediate	<a href="#">EOR</a>
MOV	Move bitmask immediate to vector	<a href="#">MOV</a>
	Move Vector register	<a href="#">MOV</a>
NOT	Bitwise invert vector	<a href="#">NOT</a>
ORN	Bitwise OR with inverted immediate	<a href="#">ORN</a>
ORR	Bitwise OR vectors (predicated)	<a href="#">ORR</a>
	Bitwise OR vectors (unpredicated)	<a href="#">ORR</a>
	Bitwise OR with immediate	<a href="#">ORR</a>

### Bitwise shift, reverse, and count

Bitwise shifts, reversals, and counts within Vector elements.

Shift counts saturate at the number of bits per element, rather than being used modulo the element size. If modulo behavior is required, then the modulus must be computed separately.

The wide element variants of the bitwise shift instructions allow a packed vector of narrower elements to be shifted by wider 64-bit shift amounts. These instructions treat the second source vector as having a fixed 64-bit doubleword element size and shift each narrow element of the first source vector by the corresponding vertically-aligned wide element of the second source vector. For example, if the first source vector contained 8-bit byte elements, then 8-bit element[0] to element[7] of the first vector are shifted by 64-bit element[0] of the second source vector, 8-bit element [8] to element[15] by 64-bit element[1], and so on. All 64 bits of the wide shift amount are significant.

**Table 5-9 Bitwise shift, permute, and count instructions**

Mnemonic	Instruction	See
ASR	Arithmetic shift right by immediate (predicated)	<a href="#">ASR</a>
	Arithmetic shift right by immediate (unpredicated)	<a href="#">ASR</a>
	Arithmetic shift right by wide elements (predicated)	<a href="#">ASR</a>
	Arithmetic shift right by wide elements (unpredicated)	<a href="#">ASR</a>
	Arithmetic shift right by vector	<a href="#">ASR</a>

**Table 5-9 Bitwise shift, permute, and count instructions (continued)**

<b>Mnemonic</b>	<b>Instruction</b>	<b>See</b>
ASRD	Arithmetic shift right for divide by immediate	<a href="#">ASRD</a>
ASRR	Reversed arithmetic shift right by vector	<a href="#">ASRR</a>
CLS	Count leading sign bits	<a href="#">CLS</a>
CLZ	Count leading zero bits	<a href="#">CLZ</a>
CNT	Count nonzero bits.	<a href="#">CNT</a>
LSL	Logical shift left by immediate (predicated)	<a href="#">LSL</a>
	Logical shift left by immediate (unpredicated)	<a href="#">LSL</a>
	Logical shift left by wide elements (predicated)	<a href="#">LSL</a>
	Logical shift left by wide elements (unpredicated)	<a href="#">LSL</a>
	Logical shift left by vector	<a href="#">LSL</a>
LSLR	Reversed logical shift left by vector	<a href="#">LSLR</a>
LSR	Logical shift right by immediate (predicated)	<a href="#">LSR</a>
	Logical shift right by immediate (unpredicated)	<a href="#">LSR</a>
	Logical shift right by wide elements (predicated)	<a href="#">LSR</a>
	Logical shift right by wide elements (unpredicated)	<a href="#">LSR</a>
	Logical shift right by vector	<a href="#">LSR</a>
LSRR	Reversed logical shift right by vector	<a href="#">LSRR</a>
RBIT	Reverse bits	<a href="#">RBIT</a>

## 5.2.7 Floating-point operations

The following instructions operate on floating-point data within a vector.

### Floating-point arithmetic

These instructions perform arithmetic operations on vectors containing floating-point element values.

**Table 5-10 Floating-point arithmetic instructions**

<b>Mnemonic</b>	<b>Instruction</b>	<b>See</b>
FABD	Floating-point absolute difference	<a href="#">FABD</a>
FABS	Floating-point absolute value	<a href="#">FABS</a>
FADD	Floating-point add immediate	<a href="#">FADD</a>
	Floating-point add (predicated)	<a href="#">FADD</a>
	Floating-point add (unpredicated)	<a href="#">FADD</a>
FDIV	Floating-point divide	<a href="#">FDIV</a>
FDIVR	Floating-point reversed divide	<a href="#">FDIVR</a>
FMAX	Floating-point maximum with immediate	<a href="#">FMAX</a>
	Floating-point maximum vectors	<a href="#">FMAX</a>
FMAXNM	Floating-point maximum number with immediate	<a href="#">FMAXNM</a>
	Floating-point maximum number vectors	<a href="#">FMAXNM</a>
FMIN	Floating-point minimum with immediate	<a href="#">FMIN</a>
	Floating-point minimum vectors	<a href="#">FMIN</a>
FMINNM	Floating-point minimum number with immediate	<a href="#">FMINNM</a>

**Table 5-10 Floating-point arithmetic instructions (continued)**

Mnemonic	Instruction	See
	Floating-point minimum number vectors	<a href="#">FMINNM</a>
FMUL	Floating-point multiply by immediate	<a href="#">FMUL</a>
	Floating-point multiply vectors (predicated)	<a href="#">FMUL</a>
	Floating-point multiply vectors (unpredicated)	<a href="#">FMUL</a>
FMULX	Floating-point multiply-extended	<a href="#">FMULX</a>
FNEG	Floating-point negate	<a href="#">FNEG</a>
FRECPE	Floating-point reciprocal estimate	<a href="#">FRECPE</a>
FRECPS	Floating-point reciprocal step	<a href="#">FRECPS</a>
FRECPX	Floating-point reciprocal exponent	<a href="#">FRECPX</a>
FRSQRTE	Floating-point reciprocal square root estimate	<a href="#">FRSQRTE</a>
FRSQRTS	Floating-point reciprocal square root step	<a href="#">FRSQRTS</a>
FSCALE	Floating-point adjust exponent by vector	<a href="#">FSCALE</a>
FSQRT	Floating-point square root	<a href="#">FSQRT</a>
FSUB	Floating-point subtract immediate	<a href="#">FSUB</a>
	Floating-point subtract vectors (predicated)	<a href="#">FSUB</a>
	Floating-point subtract vectors (unpredicated)	<a href="#">FSUB</a>
FSUBR	Floating-point reversed subtract from immediate	<a href="#">FSUBR</a>
	Floating-point reversed subtract vectors	<a href="#">FSUBR</a>

### Floating-point multiply accumulate

These instructions perform floating-point fused multiply-add or multiply-subtract operations and their negated forms. There are two groups of these instructions, as follows:

- Instructions where the result of the operation is written to the addend register.
  - Supported instructions are: FMLA, FMLS, FNMLA, FNMLS.
- Instructions where the result of the operation is written to the multiplicand register.
  - Supported instructions are: FMAD, FMSB, FNMA, FNMSB.

**Table 5-11 Floating-point multiply accumulate instructions**

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add vectors, writing to the addend register	<a href="#">FMLA</a>
FMLS	Floating-point fused multiply-subtract vectors, writing to the addend register	<a href="#">FMLS</a>
FNMLA	Floating-point negated fused multiply-add vectors, writing to the addend register	<a href="#">FNMLA</a>
FNMLS	Floating-point negated fused multiply-subtract vectors, writing to the addend register	<a href="#">FNMLS</a>
FMAD	Floating-point fused multiply-add vectors, writing to the multiplicand register	<a href="#">FMAD</a>

**Table 5-11 Floating-point multiply accumulate instructions (continued)**

Mnemonic	Instruction	See
FMSB	Floating-point fused multiply-subtract vectors, writing to the multiplicand register	<a href="#">FMSB</a>
FNMAD	Floating-point negated fused multiply-add vectors, writing to the multiplicand register	<a href="#">FNMAD</a>
FNMSB	Floating-point negated fused multiply-subtract vectors, writing to the multiplicand register	<a href="#">FNMSB</a>

### Floating-point complex arithmetic

These instructions perform arithmetic on vectors containing floating-point complex numbers as interleaved pairs of elements, where the even-numbered elements contain the real components and the odd-numbered elements contain the imaginary components.

The FCADD instructions rotate the complex numbers in the second source vector by 90 degrees or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, before adding active pairs of elements to the corresponding elements of the first source vector in a destructive manner.

The FCMLA instructions perform a transformation of the operands to allow the creation of multiply-add or multiply-subtract operations on complex numbers by combining two of the instructions. The transformations performed are as follows:

- The complex numbers in the second source vector, considered in polar form, are rotated by 0 degrees or 180 degrees before multiplying by the duplicated real components of the first source vector.
- The complex numbers in the second source vector, considered in polar form, are rotated by 90 degrees or 270 degrees before multiplying by the duplicated imaginary components of the first source vector.

The resulting products are then added to the corresponding components of the destination and addend vector, without intermediate rounding. Two FCMLA instructions can be used as follows:

```
FCMLA Zda.S, Pg/M, Zn.S, Zm.S, #A
...
FCMLA Zda.S, Pg/M, Zn.S, Zm.S, #B
```

For example, some meaningful combinations of A and B are:

- A=0, B=90. In this case, the two vectors of complex numbers in Zn and Zm are multiplied and the products are added to the complex numbers in Zda.
- A=0, B=270. In this case, the conjugates of the complex numbers in Zn are multiplied by the complex numbers in Zm and the products are added to the complex numbers in Zda.
- A=180, B=270. In this case, the two vectors of complex numbers in Zn and Zm are multiplied and the products are subtracted from the complex numbers in Zda.
- A=180, B=90. In this case, the conjugates of the complex numbers in Zn are multiplied by the complex numbers in Zm and the products are subtracted from the complex numbers in Zda.

#### ————— **Note** —————

The lack of intermediate rounding can give unexpected results in certain cases relative to a traditional sequence of independent multiply, add, and subtract instructions.

In addition, when using these instructions, the behavior of calculations such as  $(\infty+\infty i)$  multiplied by  $(0+i)$  is  $(NaN+NaNi)$  using these instructions, rather than the result expected by ISO C, which is complex  $\infty$ .

The expectation is that these instructions are only used in situations where the effect of differences in the rounding and handling of infinities are not material to the calculation.

**Table 5-12 Floating-point complex arithmetic instructions**

Mnemonic	Instruction	See
FCADD	Floating-point complex add with rotate	<a href="#">FCADD</a>
FCMLA	Floating-point complex multiply-add with rotate	<a href="#">FCMLA</a>

### Floating-point rounding and conversion

These instructions change floating-point size and precision, round floating-point to integral floating-point with explicit rounding mode, and convert floating-point to or from integer format.

**Table 5-13 Floating-point rounding and conversion instructions**

Mnemonic	Instruction	See
FCVT	Floating-point convert precision	<a href="#">FCVT</a>
FCVTZS	Floating-point convert to signed integer, rounding toward zero	<a href="#">FCVTZS</a>
FCVTZU	Floating-point convert to unsigned integer, rounding toward zero	<a href="#">FCVTZU</a>
FRINTA	Floating-point round to integral value, to nearest with ties away from zero	<a href="#">FRINTA</a>
FRINTI	Floating-point round to integral value, using the current rounding mode	<a href="#">FRINTI</a>
FRINTM	Floating-point round to integral value, toward minus infinity	<a href="#">FRINTM</a>
FRINTN	Floating-point round to integral value, to nearest with ties to even	<a href="#">FRINTN</a>
FRINTP	Floating-point round to integral value, toward plus infinity	<a href="#">FRINTP</a>
FRINTX	Floating-point round to integral value exact, using the current rounding mode	<a href="#">FRINTX</a>
FRINTZ	Floating-point round to integral value, toward zero	<a href="#">FRINTZ</a>
SCVTF	Signed integer convert to floating-point	<a href="#">SCVTF</a>
UCVTF	Unsigned integer convert to floating-point	<a href="#">UCVTF</a>

### Floating-point comparisons

These instructions compare active floating-point element values in the first source vector with corresponding elements in the second vector or with the immediate value +0.0. The Boolean result of each comparison is placed in the corresponding element of the destination predicate. Inactive elements in the destination Predicate register are set to zero. Floating-point vector comparisons do not set the condition flags.

**Table 5-14 Floating-point comparison instructions**

Mnemonic	Instruction	See
FACGE	Floating-point absolute compare greater than or equal	<a href="#">FACGE</a>
FACGT	Floating-point absolute compare greater than	<a href="#">FACGT</a>
FACLE	Floating-point absolute compare less than or equal	<a href="#">FACLE</a>
FACT	Floating-point absolute compare less than	<a href="#">FACT</a>
FCMEQ	Floating-point compare equal to zero	<a href="#">FCMEQ</a>

**Table 5-14 Floating-point comparison instructions (continued)**

Mnemonic	Instruction	See
	Floating-point compare equal to vector	<a href="#">FCMEQ</a>
FCMGE	Floating-point compare greater than or equal to zero	<a href="#">FCMGE</a>
	Floating-point compare greater than or equal to vector	<a href="#">FCMGE</a>
FCMGT	Floating-point compare greater than zero	<a href="#">FCMGT</a>
	Floating-point compare greater than vector	<a href="#">FCMGT</a>
FCMLE	Floating-point compare less than or equal to zero	<a href="#">FCMLE</a>
	Floating-point compare less than or equal to vector	<a href="#">FCMLE</a>
FCMLT	Floating-point compare less than zero	<a href="#">FCMLT</a>
	Floating-point compare less than vector	<a href="#">FCMLT</a>
FCMNE	Floating-point compare not equal to zero	<a href="#">FCMNE</a>
	Floating-point compare not equal to vector	<a href="#">FCMNE</a>
FCMUO	Floating-point unordered vectors	<a href="#">FCMUO</a>

### Floating-point transcendental acceleration

The floating-point transcendental instructions accelerate calculations of sine, cosine, and exponential functions for vectors containing floating-point element values.

The trigonometric instructions accelerate the calculation of a polynomial series approximation for the sine and cosine functions. The exponential instruction accelerates the polynomial series calculation of the exponential function.

**Table 5-15 Floating-point transcendental acceleration instructions**

Mnemonic	Instruction	See
FTMAD	Floating-point trigonometric multiply-add coefficient	<a href="#">FTMAD</a>
FTSMUL	Floating-point trigonometric starting value	<a href="#">FTSMUL</a>
FTSSEL	Floating-point trigonometric select coefficient	<a href="#">FTSSEL</a>
FEXPA	Floating-point exponential accelerator	<a href="#">FEXPA</a>

### Floating-point indexed multiplies

These instructions multiply all floating-point elements within each 128-bit segment of the first source vector by the single numbered element within the corresponding segment of the second source vector. For the FMLA and FMLS instructions, the products are destructively added or subtracted from the corresponding elements of the addend and destination vector, without intermediate rounding.

**Table 5-16 Floating-point index multiply instructions**

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add by indexed elements	<a href="#">FMLA</a>
FMLS	Floating-point fused multiply-subtract by indexed elements	<a href="#">FMLS</a>
FMUL	Floating-point multiply by indexed elements	<a href="#">FMUL</a>

## 5.2.8 Predicate operations

These instructions relate to operations that manipulate the Predicate registers.

Some of these instructions are insensitive to the Predicate element size and specify an explicit byte element size qualifier, `.B`, but an assembler must accept any qualifier, or none.

### Predicate initialization

These instructions initialize Predicate elements.

Predicate elements can be initialized to be FALSE, or to be TRUE when their Element number is less than:

- A fixed number of elements, `VL1` to `VL256`.
- The largest power of two elements, `POW2`.
- The largest multiple of three or four elements, `MUL3` or `MUL4`.
- The number of accessible elements, `ALL`, which is implicitly a multiple of two.

Unspecified or out of range constraint encodings generate a predicate with values that are all FALSE and do not cause an Undefined Instruction exception.

**Table 5-17 Predicate initialization instructions**

Mnemonic	Instruction	See
PFALSE	Set all Predicate elements to FALSE	<a href="#">PFALSE</a>
PTRUE	Initialize Predicate elements from named constraint	<a href="#">PTRUE</a>
PTRUES	Initialize Predicate elements from named constraint, setting the condition flags	<a href="#">PTRUES</a>

### Predicate move operations

These instructions copy data from Predicate elements. These instructions operate on a fixed, 1-bit Predicate element size, so the Governing predicate for the flag-setting instructions must be in canonical form.

**Table 5-18 Predicate move instructions**

Mnemonic	Instruction	See
SEL	Select Predicate elements from two predicates	<a href="#">SEL</a>
MOV	Move Predicate elements (predicated, merging)	<a href="#">MOV</a>
	Move Predicate elements (predicated, zeroing)	<a href="#">MOV</a>
	Move Predicate elements (unpredicated)	<a href="#">MOV</a>
MOVS	Move Predicate elements, setting the condition flags (predicated)	<a href="#">MOVS</a>
	Move Predicate elements, setting the condition flags (unpredicated)	<a href="#">MOVS</a>

### Predicate logical operations

These instructions perform bitwise logical operations on Predicate registers that operate on all bits of the register, implying a fixed, 1-bit Predicate element size. The flag-setting variants set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero. Inactive elements in the destination Predicate register are set to zero, except for PTEST which does not specify a destination register. Because these instructions operate with a fixed, 1-bit element size, the Governing predicate for the flag-setting instructions must be in canonical form.

**Table 5-19 Predicate logical operation instructions**

Mnemonic	Instruction	See
AND	Bitwise AND predicates	<a href="#">AND</a>
ANDS	Bitwise AND predicates, setting the condition flags	<a href="#">ANDS</a>
BIC	Bitwise clear predicates	<a href="#">BIC</a>

**Table 5-19 Predicate logical operation instructions (continued)**

Mnemonic	Instruction	See
BICS	Bitwise clear predicates, setting the condition flags	<a href="#">BICS</a>
EOR	Bitwise exclusive OR predicates	<a href="#">EOR</a>
EORS	Bitwise exclusive OR predicates, setting the condition flags	<a href="#">EORS</a>
NAND	Bitwise NAND predicates	<a href="#">NAND</a>
NANDS	Bitwise NAND predicates, setting the condition flags	<a href="#">NANDS</a>
NOR	Bitwise NOR predicates	<a href="#">NOR</a>
NORS	Bitwise NOR predicates, setting the condition flags	<a href="#">NORS</a>
NOT	Bitwise invert predicate	<a href="#">NOT</a>
NOTS	Bitwise invert predicate, setting the condition flags	<a href="#">NOTS</a>
ORN	Bitwise OR inverted predicate	<a href="#">ORN</a>
ORNS	Bitwise OR inverted predicate, setting the condition flags	<a href="#">ORNS</a>
ORR	Bitwise OR predicates	<a href="#">ORR</a>
ORRS	Bitwise OR predicates, setting the condition flags	<a href="#">ORRS</a>
PTEST	Test predicate value, setting the condition flags	<a href="#">PTEST</a>

### FFR predicate handling

These instructions work with SVE First-fault and Non-fault loads using the FFR to determine which elements have been successfully loaded and which remain to be loaded on a subsequent iteration. The RDIFFRS instruction sets the N, Z, and C condition flags based on the predicate result, and sets the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the Governing predicate used with the predicated RDIFFR and RDIFFRS instructions must be in canonical form.

**Table 5-20 FFR predicate handling instructions**

Mnemonic	Instruction	See
RDIFFR	Return predicate of successfully loaded elements (unpredicated)	<a href="#">RDIFFR</a>
	Return predicate of successfully loaded elements (predicated)	<a href="#">RDIFFR</a>
RDIFFRS	Return predicate of successfully loaded elements, setting the condition flags (predicated)	<a href="#">RDIFFRS</a>
SETFFR	Initialize the First-fault register to all TRUE	<a href="#">SETFFR</a>
WRFFR	Write a Predicate register to the First-fault register	<a href="#">WRFFR</a>

### Predicate counts

These instructions count either the number of Active predicate elements that are set to TRUE, or the number of elements implied by a named predicate constraint. The count can be placed in a general-purpose register, or used to increment or decrement a vector or general-purpose register.

Signed or unsigned saturating variants handle cases where, for example, an increment might cause a vectorized scalar loop index to overflow and therefore never satisfy a loop termination condition that compares it with a limit that is close to the maximum integer value.

The named predicate constraint limits the number of elements to:

- A fixed number of elements, VL1 to VL256.
- The largest power of two elements, POW2.

- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, implicitly a multiple of two.

Unspecified or out of range predicate constraint encodings generate a zero element count and do not cause an Undefined Instruction exception.

**Table 5-21 Predicate count instructions**

<b>Mnemonic</b>	<b>Instruction</b>	<b>See</b>
CNTB	Set scalar to multiple of 8-bit predicate constraint element count	<a href="#">CNTB</a>
CNTH	Set scalar to multiple of 16-bit predicate constraint element count	<a href="#">CNTH</a>
CNTW	Set scalar to multiple of 32-bit predicate constraint element count	<a href="#">CNTW</a>
CNTD	Set scalar to multiple of 64-bit predicate constraint element count	<a href="#">CNTD</a>
CNTP	Set scalar to the number of Active predicate elements that are TRUE	<a href="#">CNTP</a>
DECB	Decrement scalar by multiple of 8-bit predicate constraint element count	<a href="#">DECB</a>
DECH	Decrement scalar by multiple of 16-bit predicate constraint element count	<a href="#">DECH</a>
	Decrement vector by multiple of 16-bit predicate constraint element count	<a href="#">DECH</a>
DECW	Decrement scalar by multiple of 32-bit predicate constraint element count	<a href="#">DECW</a>
	Decrement vector by multiple of 32-bit predicate constraint element count	<a href="#">DECW</a>
DECD	Decrement scalar by multiple of 64-bit predicate constraint element count	<a href="#">DECD</a>
	Decrement vector by multiple of 64-bit predicate constraint element count	<a href="#">DECD</a>
DECP	Decrement scalar by the number of predicate elements that are TRUE	<a href="#">DECP</a>
	Decrement vector by the number of Active predicate elements that are TRUE	<a href="#">DECP</a>
INCB	Increment scalar by multiple of 8-bit predicate constraint element count	<a href="#">INCB</a>
INCH	Increment scalar by multiple of 16-bit predicate constraint element count	<a href="#">INCH</a>
	Increment vector by multiple of 16-bit predicate constraint element count	<a href="#">INCH</a>
INCW	Increment scalar by multiple of 32-bit predicate constraint element count	<a href="#">INCW</a>
	Increment vector by multiple of 32-bit predicate constraint element count	<a href="#">INCW</a>
INCD	Increment scalar by multiple of 64-bit predicate constraint element count	<a href="#">INCD</a>
	Increment vector by multiple of 64-bit predicate constraint element count	<a href="#">INCD</a>

Table 5-21 Predicate count instructions (continued)

Mnemonic	Instruction	See
INCP	Increment scalar by the number of predicate elements that are TRUE	<a href="#">INCP</a>
	Increment vector by the number of predicate elements that are TRUE	<a href="#">INCP</a>
SQDECB	Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count	<a href="#">SQDECB</a>
SQDECH	Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count	<a href="#">SQDECH</a>
	Signed saturating decrement vector by multiple of 16-bit predicate constraint element count	<a href="#">SQDECH</a>
SQDECW	Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.	<a href="#">SQDECW</a>
	Signed saturating decrement vector by multiple of 32-bit predicate constraint element count	<a href="#">SQDECW</a>
SQDECD	Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count	<a href="#">SQDECD</a>
	Signed saturating decrement vector by multiple of 64-bit predicate constraint element count	<a href="#">SQDECD</a>
SQDECP	Signed saturating decrement scalar the number of predicate elements that are TRUE	<a href="#">SQDECP</a>
	Signed saturating decrement vector by the number of predicate elements that are TRUE	<a href="#">SQDECP</a>
SQINCB	Signed saturating increment scalar by multiple of 8-bit predicate constraint element count	<a href="#">SQINCB</a>
SQINCH	Signed saturating increment scalar by multiple of 16-bit predicate constraint element count	<a href="#">SQINCH</a>
	Signed saturating increment vector by multiple of 16-bit predicate constraint element count	<a href="#">SQINCH</a>
SQINCW	Signed saturating increment scalar by multiple of 32-bit predicate constraint element count	<a href="#">SQINCW</a>
	Signed saturating increment vector by multiple of 32-bit predicate constraint element count	<a href="#">SQINCW</a>
SQINCD	Signed saturating increment scalar by multiple of 64-bit predicate constraint element count	<a href="#">SQINCD</a>
	Signed saturating increment vector by multiple of 64-bit predicate constraint element count	<a href="#">SQINCD</a>
SQINCP	Signed saturating increment scalar by the number of predicate elements that are TRUE	<a href="#">SQINCP</a>
	Signed saturating increment vector by the number of predicate elements that are TRUE	<a href="#">SQINCP</a>
UQDECB	Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count	<a href="#">UQDECB</a>
UQDECH	Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count	<a href="#">UQDECH</a>
	Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count	<a href="#">UQDECH</a>

**Table 5-21 Predicate count instructions (continued)**

<b>Mnemonic</b>	<b>Instruction</b>	<b>See</b>
UQDECW	Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count	<a href="#">UQDECW</a>
	Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count	<a href="#">UQDECW</a>
UQDECD	Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count	<a href="#">UQDECD</a>
	Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count	<a href="#">UQDECD</a>
UQDECP	Unsigned saturating decrement scalar by the number of predicate elements that are TRUE	<a href="#">UQDECP</a>
	Unsigned saturating decrement vector by the number of predicate elements that are TRUE	<a href="#">UQDECP</a>
UQINCB	Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count	<a href="#">UQINCB</a>
UQINCH	Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count	<a href="#">UQINCH</a>
	Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count	<a href="#">UQINCH</a>
UQINCW	Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count	<a href="#">UQINCW</a>
	Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count	<a href="#">UQINCW</a>
UQINCD	Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count	<a href="#">UQINCD</a>
	Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count	<a href="#">UQINCD</a>
UQINCP	Unsigned saturating increment scalar by the number of predicate elements that are TRUE	<a href="#">UQINCP</a>
	Unsigned saturating increment vector by the number of predicate elements that are TRUE	<a href="#">UQINCP</a>

### Loop control

These instructions control counted vector loops and vector loops with data-dependent termination conditions.

These instructions create a loop partition predicate with Active elements set to TRUE up to the point where the loop should terminate, and FALSE thereafter. Two loop concepts are supported:

### Simple loops

For simple counted loops, the `WHILE` instructions compare an incrementing value from their first scalar source register with their second scalar register, for each destination Predicate element. The result is a Predicate with elements set to `TRUE` while the comparison is true, and `FALSE` thereafter. The condition flags are unconditionally set to control a subsequent conditional branch.

**Table 5-22 Simple counted loop instructions**

Mnemonic	Instruction	See
<code>WHILELE</code>	While incrementing signed scalar less than or equal to scalar	<a href="#">WHILELE</a>
<code>WHILELO</code>	While incrementing unsigned scalar lower than scalar	<a href="#">WHILELO</a>
<code>WHILELS</code>	While incrementing unsigned scalar lower than or the same as scalar	<a href="#">WHILELS</a>
<code>WHILELT</code>	While incrementing signed scalar less than scalar	<a href="#">WHILELT</a>

### Data-dependent loops

For data-dependent termination conditions, it is necessary to convert the result of a vector comparison into a loop partition predicate. The new partition truncates the current vector partition immediately before or after the first active `TRUE` comparison. The condition flags are optionally set to control a subsequent conditional branch.

The `BRKA` instructions set active destination Predicate elements to `TRUE` up to and including the first active `TRUE` element in their source Predicate register, setting subsequent elements to `FALSE`.

The `BRKB` instructions set active destination Predicate elements to `TRUE` up to but excluding the first active `TRUE` element in their source Predicate register, setting subsequent elements to `FALSE`.

The `BRKPA` and `BRKPB` instructions propagate the result of a previous `BRKB` or `BRKPB` instruction, by setting their destination Predicate register to all `FALSE` if the Last active element of their first source Predicate register is not `TRUE`, but otherwise generate the destination predicate from their second source predicate as described for the `BRKA` and `BRKB` instructions.

The `BRKN` instructions propagate the result of a previous `BRKB` or `BRKPB` instruction by setting the destination Predicate register to all `FALSE` if the Last active element of their first source Predicate register is not `TRUE`, but otherwise leave the destination predicate unchanged. The destination and second source predicate must have been created by another instruction, such as `RDFFR` or `WHILE`.

These instructions operate on a fixed, 1-bit Predicate element size, so the Governing predicate must be in canonical form.

**Table 5-23 Data-dependent loop instructions**

Mnemonic	Instruction	See
<code>BRKA</code>	Break after the first true condition	<a href="#">BRKA</a>
<code>BRKAS</code>	Break after the first true condition, setting the condition flags	<a href="#">BRKAS</a>
<code>BRKB</code>	Break before the first true condition	<a href="#">BRKB</a>
<code>BRKBS</code>	Break before the first true condition, setting the condition flags	<a href="#">BRKBS</a>
<code>BRKN</code>	Propagate break to next partition	<a href="#">BRKN</a>
<code>BRKNS</code>	Propagate break to next partition, setting the condition flags	<a href="#">BRKNS</a>
<code>BRKPA</code>	Break after the first true condition, propagating from previous partition	<a href="#">BRKPA</a>

**Table 5-23 Data-dependent loop instructions (continued)**

Mnemonic	Instruction	See
BRKPAS	Break after the first true condition, propagating from previous partition, setting the condition flags	<a href="#">BRKPAS</a>
BRKPB	Break before the first true condition, propagating from the previous partition	<a href="#">BRKPB</a>
BRKPBS	Break before the first true condition, propagating from the previous partition, setting the condition flags	<a href="#">BRKPBS</a>

### Serialized operations

These instructions permit Active elements within a vector to be processed sequentially without unpacking the vector. The condition flags are unconditionally set to control a subsequent conditional branch.

The PFIRST instruction operates with a fixed, 1-bit Predicate element size, so its Governing predicate must be in canonical form.

**Table 5-24 Serialized operation instructions**

Mnemonic	Instruction	See
PFIRST	Set the First active element to TRUE	<a href="#">PFIRST</a>
PNEXT	Find next Active element	<a href="#">PNEXT</a>
CTERMEQ	Compare and terminate loop when equal	<a href="#">CTERMEQ</a>
CTERMNE	Compare and terminate loop when not equal	<a href="#">CTERMNE</a>

## 5.2.9 Move operations

### Element permute and shuffle

These instructions move data between different Vector elements, or between Vector elements and scalar registers. These instructions perform the following operations:

- Conditionally extract the Last active element of a vector or the following element.
  - The supported instructions are: CLASTA, CLASTB.
- Unconditionally extract the Last active element of a vector or the following element.
  - The supported instructions are: LASTA, LASTB.
- Variable permute instructions where the permutation is determined by the values in a Predicate register or a table of element index values.
  - The supported instructions are: COMPACT, SPLICE, TBL.
- Fixed permute instructions where the form of the permutation is encoded in the instruction.
  - The supported instructions are: DUP, EXT, INSR, REV, REVB, REVH, REVW, SUNPKHI, SUNPKLO, TRN1, TRN2, UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, ZIP2.

**Table 5-25 Element permute and shuffle instructions**

Mnemonic	Instruction	See
CLASTA	Conditionally extract element after last to general-purpose register	<a href="#">CLASTA</a>
	Conditionally extract element after last to SIMD&FP scalar	<a href="#">CLASTA</a>
	Conditionally extract element after last to vector	<a href="#">CLASTA</a>
CLASTB	Conditionally extract Last active element to general-purpose register	<a href="#">CLASTB</a>



**Table 5-26 Unpacking instructions (continued)**

Mnemonic	Instruction	See
UUNPKLO	Unpack and zero-extend elements from low half of vector	<a href="#">UUNPKLO</a>
PUNPKHI	Unpack and widen elements from high half of predicate	<a href="#">PUNPKHI</a>
PUNPKLO	Unpack and widen elements from low half of predicate	<a href="#">PUNPKLO</a>

### Predicate permute

These instructions are used to move and permute Predicate elements. These instructions generally mirror the fixed vector permutes to allow predicates to follow their data. The permutes move all of the bits in a Predicate element, not just the least-significant bit.

**Table 5-27 Predicate permute instructions**

Mnemonic	Instruction	See
REV	Reverse all elements in predicate	<a href="#">REV</a>
TRN1	Interleave even elements from two predicates	<a href="#">TRN1</a>
TRN2	Interleave odd elements from two predicates	<a href="#">TRN2</a>
UZP1	Select even elements from two predicates	<a href="#">UZP1</a>
UZP2	Select odd elements from two predicates	<a href="#">UZP2</a>
ZIP1	Interleave elements from low halves of two predicates	<a href="#">ZIP1</a>
ZIP2	Interleave elements from high halves of two predicates	<a href="#">ZIP2</a>

### Index vector generation

The INDEX instruction initializes a vector horizontally by setting its first element to an integer value, and then repeatedly incrementing it by a second integer value to generate the subsequent elements. Each integer value can be specified as a signed immediate or a general-purpose register.

**Table 5-28 Index vector generation instructions**

Mnemonic	Instruction	See
INDEX	Create index vector starting from and incremented by immediates	<a href="#">INDEX</a>
	Create index vector starting from immediate and incremented by general-purpose register	<a href="#">INDEX</a>
	Create index vector starting from general-purpose register and incremented by immediate	<a href="#">INDEX</a>
	Create index vector starting from and incremented by general-purpose registers	<a href="#">INDEX</a>

### Move prefix

The [MOVPRFX](#) (predicated) instruction is a predicated vector move that can be combined with a predicated Destructive instruction that immediately follows it, in program order, to create a single constructive operation, or to convert an instruction with Merging predication to use Zeroing predication.

The [MOVPRFX](#) (unpredicated) instruction is an unpredicated vector move that can be combined with a predicated and unpredicated Destructive instruction that immediately follows it, in program order, to create a single constructive operation.

The Prefixed instruction that immediately follows a MOVPRFX instruction in program order must be an SVE instruction where:

- The destination register field implicitly specifies one of the source operands, which means that it is a destructive binary or ternary vector operation or unary operation with Merging predication, excluding MOVPRFX.
- The destination register is the same as the MOVPRFX destination register.
- The Prefixed instruction does not use the MOVPRFX destination register in any of its other source register fields, even if it has a different name but refers to the same architectural register state. For example, Z1, V1, and D1 all refer to the same architectural register.
- If the MOVPRFX instruction is predicated, then the Prefixed instruction is predicated using the same Governing predicate register, and the maximum encoded element size is the same as the MOVPRFX element size, excluding the fixed-size 64-bit elements of the wide elements form of bitwise shift and integer compare operations.
- If the MOVPRFX instruction is unpredicated, then the Prefixed instruction can use any Governing predicate register and element size, or it can be unpredicated. A predicated MOVPRFX cannot be used with an unpredicated instruction.

Otherwise, the use of a MOVPRFX instruction has a CONSTRAINED UNPREDICTABLE result, with the following permitted behaviors for the pair of instructions:

- Either or both instructions can execute with their individually described effects.
- Either instruction can generate an Undefined Instruction exception.
- Either or both instructions can execute as a NOP.
- The second instruction can execute with an UNKNOWN value for any of its source registers.
- Any register that is written by either or both instructions can be set to an UNKNOWN value.
- A control flow instruction that writes the PC can set the PC to an UNKNOWN value.

**Table 5-29 Move prefix instructions**

Mnemonic	Instruction	See
MOVPRFX	Move prefix (predicated)	<a href="#">MOVPRFX</a>
	Move prefix (unpredicated)	<a href="#">MOVPRFX</a>

## 5.2.10 Reduction operations

### Horizontal reductions

These instructions perform arithmetic horizontally across Active elements of a single source vector and deliver a scalar result.

The floating-point horizontal accumulating sum instruction, FADDA, operates strictly in order of increasing Element number across a vector, using the scalar destination register as a source for the initial value of the accumulator. This preserves the original program evaluation order where non-associativity is required.

The other floating-point reductions calculate their result using a recursive pair-wise algorithm that does not preserve the original program order, but permits increased parallelism for code that does not require strict order of evaluation.

Integer reductions are fully associative, and the order of evaluation is not specified by the architecture.

**Table 5-30 Horizontal reduction instructions**

Mnemonic	Instruction	See
ANDV	Bitwise AND reduction, treating Inactive elements as all ones	<a href="#">ANDV</a>
EORV	Bitwise XOR reduction, treating Inactive elements as zero	<a href="#">EORV</a>
FADDA	Floating-point add strictly-ordered reduction, accumulating in scalar, ignoring Inactive elements	<a href="#">FADDA</a>

**Table 5-30 Horizontal reduction instructions (continued)**

<b>Mnemonic</b>	<b>Instruction</b>	<b>See</b>
FADDV	Floating-point add recursive reduction, treating Inactive elements as +0.0	<a href="#">FADDV</a>
FMAXNMV	Floating-point maximum number recursive reduction, treating Inactive elements as the default NaN	<a href="#">FMAXNMV</a>
FMAXV	Floating-point maximum recursive reduction, treating Inactive elements as negative infinity	<a href="#">FMAXV</a>
FMINNMV	Floating-point minimum number recursive reduction, treating Inactive elements as the default NaN	<a href="#">FMINNMV</a>
FMINV	Floating-point minimum recursive reduction, treating Inactive elements as positive infinity	<a href="#">FMINV</a>
ORV	Bitwise OR reduction, treating Inactive elements as zero	<a href="#">ORV</a>
SADDV	Signed add reduction, treating Inactive elements as zero	<a href="#">SADDV</a>
SMAXV	Signed maximum reduction, treating Inactive elements as the minimum signed integer	<a href="#">SMAXV</a>
SMINV	Signed minimum reduction, treating Inactive elements as the maximum signed integer	<a href="#">SMINV</a>
UADDV	Unsigned add reduction, treating Inactive elements as zero	<a href="#">UADDV</a>
UMAXV	Unsigned maximum reduction, treating Inactive elements as zero	<a href="#">UMAXV</a>
UMINV	Unsigned minimum reduction, treating Inactive elements as the maximum unsigned integer	<a href="#">UMINV</a>

# Chapter 6

## System Registers

This chapter introduces the ARMv8-A System registers that affect SVE as well as the new System registers that are specific to SVE. This chapter contains the following sections:

- [AArch64 System registers affected by SVE](#) on page 6-64.
- [SVE System registers](#) on page 6-65.

## 6.1 System registers

The following sections describe the AArch64 System registers that affect SVE and the AArch64 System registers specific to SVE. For general information on AArch64 System registers, see the section titled *AArch64 System register descriptions* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### 6.1.1 AArch64 System registers affected by SVE

Table 6-1 lists the modifications to the AArch64 System registers due to SVE.

———— **Note** —————

The Description column in Table 6-1 provides a general description of the behavior of the fields relating to SVE. For the strict definition of these fields, follow the links to the appropriate XML or HTML content.

**Table 6-1 AArch64 System registers**

Register	Change	Description
<a href="#">ID_AA64PFR0_EL1</a>	Defines bits[35:32] as the SVE field.	The SVE field indicates whether the Scalable Vector Extension is implemented.
<a href="#">CPACR_EL1</a>	Defines bits[17:16] as the ZEN field.	The ZEN field enables access to SVE functionality from EL1 and EL0.
<a href="#">CPTR_EL2</a>	When <a href="#">HCR_EL2.E2H</a> == 0, defines bit[8] as TZ.	The TZ field traps access to SVE functionality from EL2 and Non-secure EL1&0 to EL2.
	When <a href="#">HCR_EL2.E2H</a> == 1, defines bits[17:16] as the ZEN field.	The ZEN field enables access to SVE functionality from EL2 and Non-secure EL1&0.
<a href="#">CPTR_EL3</a>	Defines bit[8] as EZ.	The EZ field enables access to SVE functionality from EL0, EL1, EL2, and EL3.
<a href="#">TCR_EL1</a>	Defines bit[54] as NFD1 and bit[53] as NFD0.	The NFD1 and NFD0 fields disable translation table walks caused by certain elements of the SVE First-fault and Non-fault vector load instructions from EL0 for translations using TTBR1_EL1 or TTBR0_EL1, respectively.
<a href="#">TCR_EL2</a>	When <a href="#">HCR_EL2.E2H</a> == 1, defines bit[54] as NFD1 and bit[53] as NFD0.	The NFD1 and NFD0 fields disable translation table walks caused by certain elements of the SVE First-fault and Non-fault vector load instructions from EL0 for translations using TTBR1_EL2 or TTBR0_EL2, respectively.
<a href="#">EDPFR</a>	Defines bits[35:32] as the SVE field.	The SVE field indicates whether the Scalable Vector Extension is implemented for debug.
<a href="#">ESR_ELx</a>	New exception class, 0b011001, added to the EC field description.	The new EC code, 0b011001, identifies accesses to SVE functionality when disabled or trapped by <a href="#">CPACR_EL1</a> , <a href="#">CPTR_EL2</a> , or <a href="#">CPTR_EL3</a> .

## 6.1.2 SVE System registers

The AArch64 System registers specific to SVE are outlined in [Table 6-2](#).

**Table 6-2 SVE System registers**

Register	Description
<a href="#">ID_AA64ZFR0_EL1</a>	SVE feature ID register 0
<a href="#">ZCR_EL1</a>	SVE control register to constrain the accessible Vector length at EL1 and EL0
<a href="#">ZCR_EL2</a>	SVE control register to constrain the accessible Vector length at EL2 and Non-secure EL1 and EL0
<a href="#">ZCR_EL3</a>	SVE control register to constrain the accessible Vector length at EL3, EL2, EL1, and EL0

Writes to ZCR\_ELx are self-synchronizing and appear to occur in program order relative to other instructions.

When executing at an Exception level that is constrained to use a Vector length that is less than the maximum implemented Vector length, then the bits beyond the constrained length of the Vector registers, Predicate registers, and FFR are inaccessible. On taking an exception from an Exception level that is more constrained to a target Exception level that is less constrained, or on writing a larger value to ZCR\_ELx.LEN, then the previously inaccessible bits of these registers that become accessible must have a value of either zero or the value they had before executing at the more constrained size. The choice between these options is IMPLEMENTATION DEFINED and can vary dynamically. The values of the upper bits, in this case, are UNKNOWN.

If SVE instructions are disabled or trapped at ELx, or if that Exception level is in AArch32 state, then for all purposes other than a direct read, the ZCR\_ELx.LEN field has an Effective value of 0, which implies an SVE Vector length of 128 bits.

However, if floating-point and SVE are both disabled or trapped at all Exception levels below the target Exception level, in the current Security state, then the accessible SVE register state at the target Exception level is preserved.



# Chapter 7

## SVE Debug

This chapter introduces the additions to ARMv8-A AArch64 debug due to SVE. This chapter contains the following sections:

- [Self-hosted debug on page 7-68.](#)
- [External debug on page 7-69.](#)

## 7.1 Self-hosted debug

SVE defines behaviors that permit debugging of SVE instructions when using the debug exception model described in the section titled *AArch64 Self-hosted Debug* in *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### 7.1.1 Watchpoints

The following memory accesses can trigger watchpoints:

- Non-speculative, single-copy atomic accesses performed as a result of an Active element by an SVE predicated vector load or store instruction.
- Byte accesses performed by an SVE unpredicated register load or store instruction.

For all SVE predicated vector load and store instructions, watchpoint debug events can only be generated by address matches that occur due to active Vector elements.

SVE Non-fault vector loads do not generate watchpoint debug events, but any Active element access that matches a configured watchpoint is suppressed and reported in the FFR as described in [Memory fault handling on page 4-33](#).

SVE First-fault vector loads can generate a watchpoint debug event only for the First active element. If that access does not generate a fault or a watchpoint debug event, then any other Active element access that matches a configured watchpoint is suppressed and reported in the FFR as described in [Memory fault handling on page 4-33](#).

### 7.1.2 MOVPRFX instruction debug behavior

For debugging purposes, the [MOVPRFX](#) (predicated) and the [MOVPRFX](#) (unpredicated) instructions have predictable behavior when used with breakpoints and single-step execution:

- It is permitted to use MOVPRFX to prefix an A64 BRK or HLT instruction.
- A hardware breakpoint is only predictable if it is programmed with the address of the initial MOVPRFX instruction, and not the address of the Prefixed instruction.
- A single step when the instruction to be stepped is a permitted use of MOVPRFX can either step over the pair of instructions, or step over only the MOVPRFX instruction, as described in [MOVPRFX exception behavior on page 3-28](#).

## 7.2 External debug

SVE architectural state can be accessed using external debug features while in Debug state, as described in the section titled *External Debug* in *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

### 7.2.1 SVE instructions that are changed in Debug state

The list of instructions that are contained in the section titled *A64 instructions that are changed in Debug state* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* is extended to include the following SVE instruction:

**Table 7-1 SVE instructions that are changed in Debug state**

Instruction	Change in Debug state
CMPNE (immediate form, byte element size only)	This instruction has unchanged behavior in Debug state with respect to the SVE vector and predicate source and destination registers, but is architecturally defined to set DSPSR_ELO and DLR_ELO to UNKNOWN values.

### 7.2.2 SVE instructions that are unchanged in Debug state

The list of instructions that are contained in the section titled *A64 instructions that are unchanged in Debug state* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* is extended to include the following SVE instructions:

- [RDVL](#)
- [CPY](#) (immediate form, with Zeroing predication, byte element size, and a shift amount of 0 only)
- [PTRUE](#) (with ALL constraint and byte element size only)
- [RDFFR](#) (unpredicated)
- [WRFRR](#) (unpredicated)
- [EXT](#)
- [INSR](#) (scalar)
- [DUP](#) (scalar)

### 7.2.3 SVE instructions that are CONSTRAINED UNPREDICTABLE in Debug state

All SVE instructions, other than those listed in *SVE instructions that are changed in Debug state* and *SVE instructions that are unchanged in Debug state*, are CONSTRAINED UNPREDICTABLE in Debug state, with the following permissible behaviors:

- The instruction generates an Undefined Instruction exception.
- The instruction executes as a NOP.
- If the instruction modifies PSTATE, it sets DLR\_ELO and DSPSR\_ELO to UNKNOWN values.
- If the instruction is similar to an SVE instruction that has defined behavior in Debug state, it executes as that instruction. Such instructions are described in *SVE instructions that are changed in Debug state* and *SVE instructions that are unchanged in Debug state*.
- The instruction has the same behavior as in Non-debug state.



# Chapter 8

## SVE Performance Monitors Extension

This chapter introduces the changes that are made to the ARMv8-A Performance Monitor Extension by the Scalable Vector Extension. This chapter contains the following sections:

- [Introduction on page 8-72.](#)
- [New performance monitor events on page 8-73.](#)
- [Existing ARMv8-A PMU events affected by SVE on page 8-74.](#)

## 8.1 Introduction

This chapter defines the changes made to the ARMv8-A Performance Monitor Extension by SVE. For more information about the ARMv8-A Performance Monitor Extension, see the sections titled *The Performance Monitor Extension* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

SVE reserves PMU event numbers in the range 0x8000 to 0x80FF. Unless otherwise stated, the behavior of SVE PMU events is defined for AArch64 state only. In AArch32 state, it is IMPLEMENTATION DEFINED which instructions and operations are counted.

All SVE PMU events share the same IMPLEMENTATION DEFINED definition of *speculatively executed* as is defined in the section titled *PMU events and event numbers* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*.

Implementations of SVE that include the Performance Monitor Extension are architecturally required to include at least one of [SVE\\_INST\\_RETIRED](#) and [SVE\\_INST\\_SPEC](#).

### ———— Note ————

ARM strongly recommends that the [SVE\\_INST\\_RETIRED](#) event is implemented.

As well as these required events, ARM recommends that certain other events are implemented. For more information on the recommended SVE PMU events, see [List of recommended PMU events on page A-76](#). An implementation is permitted to:

- Modify the definition of an event to better correspond to the implementation.
- Not use some, or many, of these events.

The instructions and operations that are counted by each event can be described by reference to the instruction categories in [Instruction categories on page A-106](#), with the following exceptions:

- It is IMPLEMENTATION DEFINED whether operations due to any of the instructions listed in [Data movement instructions on page A-106](#), [Floating-point conversions on page A-111](#), or [Floating-point or integer instructions on page A-112](#), will be counted
  - As integer operations.
  - As floating-point operations.
  - As neither integer operations or floating-point operations.

However, they must not be counted as both integer operations and floating-point operations.

- Unless otherwise stated, a reference to Advanced SIMD or SVE instructions refers to all the instructions listed in [Instruction categories on page A-106](#) under the corresponding subheadings. This includes data-processing, predicate handling, load, store, and prefetch instructions.
- A reference to Advanced SIMD scalar instructions refers to Advanced SIMD instructions that would be counted for the ARMv8 DP\_SPEC event, and to Advanced SIMD instructions that only read element[0] of their source vectors, and can write a non-zero result only to element[0] of their destination vector.
- It is IMPLEMENTATION DEFINED whether a reference to Advanced SIMD instructions includes the instructions listed in [Cryptographic instructions on page A-114](#) that would be counted by the ARMv8 CRYPTO\_SPEC event. If they are counted as Advanced SIMD instructions, then it is IMPLEMENTATION DEFINED whether individual Cryptographic instructions are counted as SIMD or Advanced SIMD scalar instructions.
- Except for events 0x807C - 0x807F, it is IMPLEMENTATION DEFINED whether an SVE MOVPRFX instruction, or microarchitectural operations ( $\mu$ -ops) due to a MOVPRFX instruction, are counted. This can vary dynamically for each execution of the same instruction.
- The terms *Operation* and *Speculatively executed* are broad enough to permit such events to count only retired, architecturally executed instructions. However, the [UOP\\_SPEC](#), [ASE\\_UOP\\_SPEC](#), [SVE\\_UOP\\_SPEC](#), [ASE\\_SVE\\_UOP\\_SPEC](#) and [SIMD\\_UOP\\_SPEC](#) events are explicitly defined to count speculative execution on both correct and false execution paths of  $\mu$ -ops that are due to architectural instructions.

## 8.2 New performance monitor events

### 8.2.1 Required SVE PMU events

Implementations of SVE that include the Performance Monitor Extension are architecturally required to include at least one of the events listed in [Table 8-1](#).

**Table 8-1 New SVE PMU events**

Event number	Event type	Event mnemonic
0x8002	Architectural, Required	<a href="#">SVE_INST_RETIRED</a>
0x8006	Microarchitectural, Required	<a href="#">SVE_INST_SPEC</a>

#### PMU event descriptions

##### 0x8002, [SVE\\_INST\\_RETIRED](#), SVE instructions architecturally executed

This event counts architecturally executed SVE instructions. It is IMPLEMENTATION DEFINED whether this event counts the instructions listed in [Non-SIMD SVE instructions on page A-113](#).

##### 0x8006, [SVE\\_INST\\_SPEC](#), SVE operations speculatively executed

This event counts speculatively executed operations due to SVE instructions. It is IMPLEMENTATION DEFINED whether it counts operations due to the instructions listed in [Non-SIMD SVE instructions on page A-113](#).

### 8.3 Existing ARMv8-A PMU events affected by SVE

The following ARMv8-A PMU events also count SVE instructions and operations. These events are described in Table 8-2. All other ARMv8-A PMU events do not count SVE instructions and operations. See the section titled *Common Event Numbers* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* for further information on these events.

**Table 8-2 Existing PMU events affected by SVE**

Event number	Event mnemonic	SVE clarification
0x0006	LD_RETIRED	Counts architecturally executed SVE load instructions.
0x0007	ST_RETIRED	Counts architecturally executed SVE store instructions.
0x0008	INST_RETIRED	Counts architecturally executed SVE instructions. It is IMPLEMENTATION DEFINED whether MOVPREX is counted by this event.
0x000F	UNALIGNED_LDST_RETIRED	Counts architecturally executed SVE load and store instructions that access at least one unaligned element address that would generate an alignment fault when Alignment fault checking is enabled.
0x0013	MEM_ACCESS	Counts memory reads and writes as a result of SVE load and store instructions. The number of accesses generated by each SVE instruction is IMPLEMENTATION DEFINED.
0x001B	INST_SPEC	Counts speculatively executed SVE operations. It is IMPLEMENTATION DEFINED whether MOVPREX is counted by this event.
0x0066	MEM_ACCESS_RD	Similar to MEM_ACCESS, but only counts reads.
0x0067	MEM_ACCESS_WR	Similar to MEM_ACCESS but only counts writes.
0x0068	UNALIGNED_LD_SPEC	Counts speculatively executed SVE load operations that access at least one unaligned element address.
0x0069	UNALIGNED_ST_SPEC	Counts speculatively executed SVE store operations that access at least one unaligned element address.
0x006A	UNALIGNED_LDST_SPEC	Counts speculatively executed SVE load and store operations that access at least one unaligned element address.
0x0070	LD_SPEC	Counts speculatively executed SVE load operations.
0x0071	ST_SPEC	Counts speculatively executed SVE store operations.
0x0072	LDST_SPEC	Counts speculatively executed SVE load and store operations.

# Appendix A

## Recommended SVE PMU events

This section contains a list of the recommended PMU events for SVE and contains the following section:

- [List of recommended PMU events on page A-76](#)

## A.1 List of recommended PMU events

Table [Table A-1](#) describes the recommended PMU events for SVE implementations.

**Table A-1 Recommended PMU events**

Event number	Event type	Event mnemonic
0x8000	Architectural	<a href="#">SIMD_INST_RETIRED</a>
0x8001	Architectural	<a href="#">ASE_INST_RETIRED</a>
0x8002	Architectural. Required event	See <a href="#">SVE_INST_RETIRED</a>
0x8003	Architectural	<a href="#">ASE_SVE_INST_RETIRED</a>
0x8004	Microarchitectural	<a href="#">SIMD_INST_SPEC</a>
0x8005	Microarchitectural	<a href="#">ASE_INST_SPEC</a>
0x8006	Microarchitectural. Required event	See <a href="#">SVE_INST_SPEC</a>
0x8007	Microarchitectural	<a href="#">ASE_SVE_INST_SPEC</a>
0x8008	Microarchitectural	<a href="#">UOP_SPEC</a>
0x8009	Microarchitectural	<a href="#">ASE_UOP_SPEC</a>
0x800A	Microarchitectural	<a href="#">SVE_UOP_SPEC</a>
0x800B	Microarchitectural	<a href="#">ASE_SVE_UOP_SPEC</a>
0x800C	Microarchitectural	<a href="#">SIMD_UOP_SPEC</a>
0x800D	Reserved	Reserved for future expansion
0x800E	Microarchitectural	<a href="#">SVE_MATH_SPEC</a>
0x800F	Reserved	Reserved for future expansion
0x8010	Microarchitectural	<a href="#">FP_SPEC</a>
0x8011	Microarchitectural	<a href="#">ASE_FP_SPEC</a>
0x8012	Microarchitectural	<a href="#">SVE_FP_SPEC</a>
0x8013	Microarchitectural	<a href="#">ASE_SVE_FP_SPEC</a>
0x8014	Microarchitectural	<a href="#">FP_HP_SPEC</a>
0x8015	Microarchitectural	<a href="#">ASE_FP_HP_SPEC</a>
0x8016	Microarchitectural	<a href="#">SVE_FP_HP_SPEC</a>
0x8017	Microarchitectural	<a href="#">ASE_SVE_FP_HP_SPEC</a>
0x8018	Microarchitectural	<a href="#">FP_SP_SPEC</a>
0x8019	Microarchitectural	<a href="#">ASE_FP_SP_SPEC</a>
0x801A	Microarchitectural	<a href="#">SVE_FP_SP_SPEC</a>
0x801B	Microarchitectural	<a href="#">ASE_SVE_FP_SP_SPEC</a>
0x801C	Microarchitectural	<a href="#">FP_DP_SPEC</a>
0x801D	Microarchitectural	<a href="#">ASE_FP_DP_SPEC</a>
0x801E	Microarchitectural	<a href="#">SVE_FP_DP_SPEC</a>
0x801F	Microarchitectural	<a href="#">ASE_SVE_FP_DP_SPEC</a>
0x8020	Microarchitectural	<a href="#">FP_DIV_SPEC</a>
0x8021	Microarchitectural	<a href="#">ASE_FP_DIV_SPEC</a>
0x8022	Microarchitectural	<a href="#">SVE_FP_DIV_SPEC</a>
0x8023	Microarchitectural	<a href="#">ASE_SVE_FP_DIV_SPEC</a>
0x8024	Microarchitectural	<a href="#">FP_SQRT_SPEC</a>
0x8025	Microarchitectural	<a href="#">ASE_FP_SQRT_SPEC</a>
0x8026	Microarchitectural	<a href="#">SVE_FP_SQRT_SPEC</a>

Table A-1 Recommended PMU events (continued)

Event number	Event type	Event mnemonic
0x8027	Microarchitectural	ASE_SVE_FP_SQRT_SPEC
0x8028	Microarchitectural	FP_FMA_SPEC
0x8029	Microarchitectural	ASE_FP_FMA_SPEC
0x802A	Microarchitectural	SVE_FP_FMA_SPEC
0x802B	Microarchitectural	ASE_SVE_FP_FMA_SPEC
0x802C	Microarchitectural	FP_MUL_SPEC
0x802D	Microarchitectural	ASE_FP_MUL_SPEC
0x802E	Microarchitectural	SVE_FP_MUL_SPEC
0x802F	Microarchitectural	ASE_SVE_FP_MUL_SPEC
0x8030	Microarchitectural	FP_ADDSUB_SPEC
0x8031	Microarchitectural	ASE_FP_ADDSUB_SPEC
0x8032	Microarchitectural	SVE_FP_ADDSUB_SPEC
0x8033	Microarchitectural	ASE_SVE_FP_ADDSUB_SPEC
0x8034	Microarchitectural	FP_RECPE_SPEC
0x8035	Microarchitectural	ASE_FP_RECPE_SPEC
0x8036	Microarchitectural	SVE_FP_RECPE_SPEC
0x8037	Microarchitectural	ASE_SVE_FP_RECPE_SPEC
0x8038	Microarchitectural	FP_CVT_SPEC
0x8039	Microarchitectural	ASE_FP_CVT_SPEC,
0x803A	Microarchitectural	SVE_FP_CVT_SPEC
0x803B	Microarchitectural	ASE_SVE_FP_CVT_SPEC
0x803C	Microarchitectural	SVE_FP_AREDUCE_SPEC
0x803D	Microarchitectural	ASE_FP_PREDUCE_SPEC
0x803E	Microarchitectural	SVE_FP_VREDUCE_SPEC
0x803F	Microarchitectural	ASE_SVE_FP_VREDUCE_SPEC
0x8040	Microarchitectural	INT_SPEC
0x8041	Microarchitectural	ASE_INT_SPEC
0x8042	Microarchitectural	SVE_INT_SPEC
0x8043	Microarchitectural	ASE_SVE_INT_SPEC
0x8044	Microarchitectural	INT_DIV_SPEC
0x8045	Microarchitectural	INT_DIV64_SPEC
0x8046	Microarchitectural	SVE_INT_DIV_SPEC
0x8047	Microarchitectural	SVE_INT_DIV64_SPEC
0x8048	Microarchitectural	INT_MUL_SPEC
0x8049	Microarchitectural	ASE_INT_MUL_SPEC
0x804A	Microarchitectural	SVE_INT_MUL_SPEC
0x804B	Microarchitectural	ASE_SVE_INT_MUL_SPEC
0x804C	Microarchitectural	INT_MUL64_SPEC
0x804D	Microarchitectural	SVE_INT_MUL64_SPEC
0x804E	Microarchitectural	INT_MULH64_SPEC
0x804F	Microarchitectural	SVE_INT_MULH64_SPEC
0x8050 - 0x8057	Reserved	Reserved for future expansion

Table A-1 Recommended PMU events (continued)

Event number	Event type	Event mnemonic
0x8058	Microarchitectural	NONFP_SPEC
0x8059	Microarchitectural	ASE_NONFP_SPEC
0x805A	Microarchitectural	SVE_NONFP_SPEC
0x805B	Microarchitectural	ASE_SVE_NONFP_SPEC
0x805C	Reserved	Reserved for future expansion
0x805D	Microarchitectural	ASE_INT_VREDUCE_SPEC
0x805E	Microarchitectural	SVE_INT_VREDUCE_SPEC
0x805F	Microarchitectural	ASE_SVE_INT_VREDUCE_SPEC
0x8060	Microarchitectural	SVE_PERM_SPEC
0x8061	Microarchitectural	SVE_PERM_IGRANULE_SPEC
0x8062	Microarchitectural	SVE_PERM_XGRANULE_SPEC
0x8063	Microarchitectural	SVE_PERM_VARIABLE_SPEC
0x8064	Microarchitectural	SVE_XPIPE_SPEC
0x8065	Microarchitectural	SVE_XPIPE_Z2R_SPEC
0x8066	Microarchitectural	SVE_XPIPE_R2Z_SPEC
0x8067	Microarchitectural	SVE_PGEN_NVEC_SPEC
0x8068	Microarchitectural	SVE_PGEN_SPEC
0x8069	Microarchitectural	SVE_PGEN_FLG_SPEC
0x806A	Microarchitectural	SVE_PGEN_CMP_SPEC
0x806B	Microarchitectural	SVE_PGEN_FCM_SPEC
0x806C	Microarchitectural	SVE_PGEN_LOGIC_SPEC
0x806D	Microarchitectural	SVE_PPERM_SPEC
0x806E	Microarchitectural	SVE_PSCAN_SPEC
0x806F	Microarchitectural	SVE_PCNT_SPEC
0x8070	Microarchitectural	SVE_PLOOP_WHILE_SPEC
0x8071	Microarchitectural	SVE_PLOOP_TEST_SPEC
0x8072	Microarchitectural	SVE_PLOOP_ELTS_SPEC
0x8073	Microarchitectural	SVE_PLOOP_TERM_SPEC
0x8074	Microarchitectural	SVE_PRED_SPEC
0x8075	Microarchitectural	SVE_PRED_EMPTY_SPEC
0x8076	Microarchitectural	SVE_PRED_FULL_SPEC
0x8077	Microarchitectural	SVE_PRED_PARTIAL_SPEC
0x8078	Microarchitectural	SVE_UNPRED_SPEC
0x8079 - 0x807B	Reserved	Reserved for future expansion
0x807C	Microarchitectural	SVE_MOVPRFX_SPEC
0x807D	Microarchitectural	SVE_MOVPRFX_Z_SPEC
0x807E	Microarchitectural	SVE_MOVPRFX_M_SPEC
0x807F	Microarchitectural	SVE_MOVPRFX_U_SPEC
0x8080	Microarchitectural	SVE_LDST_SPEC
0x8081	Microarchitectural	SVE_LD_SPEC
0x8082	Microarchitectural	SVE_ST_SPEC
0x8083	Microarchitectural	SVE_PRF_SPEC

Table A-1 Recommended PMU events (continued)

Event number	Event type	Event mnemonic
0x8084	Microarchitectural	ASE_SVE_LDST_SPEC
0x8085	Microarchitectural	ASE_SVE_LD_SPEC
0x8086	Microarchitectural	ASE_SVE_ST_SPEC
0x8087	Microarchitectural	PRF_SPEC
0x8088	Microarchitectural	BASE_LDST_REG_SPEC
0x8089	Microarchitectural	BASE_LD_REG_SPEC
0x808A	Microarchitectural	BASE_ST_REG_SPEC
0x808B	Microarchitectural	BASE_PRF_SPEC
0x808C	Microarchitectural	FPASE_LDST_REG_SPEC
0x808D	Microarchitectural	FPASE_LD_REG_SPEC
0x808E	Microarchitectural	FPASE_ST_REG_SPEC
0x808F	Reserved	Reserved for future expansion
0x8090	Microarchitectural	SVE_LDST_REG_SPEC
0x8091	Microarchitectural	SVE_LDR_REG_SPEC
0x8092	Microarchitectural	SVE_STR_REG_SPEC
0x8093	Reserved	Reserved for future expansion
0x8094	Microarchitectural	SVE_LDST_PREG_SPEC
0x8095	Microarchitectural	SVE_LDR_PREG_SPEC
0x8096	Microarchitectural	SVE_STR_PREG_SPEC
0x8097	Reserved	Reserved for future expansion
0x8098	Microarchitectural	SVE_LDST_ZREG_SPEC
0x8099	Microarchitectural	SVE_LDR_ZREG_SPEC
0x809A	Microarchitectural	SVE_STR_ZREG_SPEC
0x809B	Reserved	Reserved for future expansion
0x809C	Microarchitectural	SVE_LDST_CONTIG_SPEC
0x809D	Microarchitectural	SVE_LD_CONTIG_SPEC
0x809E	Microarchitectural	SVE_ST_CONTIG_SPEC
0x809F	Microarchitectural	SVE_PRF_CONTIG_SPEC
0x80A0	Microarchitectural	SVE_LDSTNT_CONTIG_SPEC
0x80A1	Microarchitectural	SVE_LDNT_CONTIG_SPEC
0x80A2	Microarchitectural	SVE_STNT_CONTIG_SPEC
0x80A3	Reserved	Reserved for future expansion
0x80A4	Microarchitectural	ASE_SVE_LDST_MULTI_SPEC
0x80A5	Microarchitectural	ASE_SVE_LD_MULTI_SPEC
0x80A6	Microarchitectural	ASE_SVE_ST_MULTI_SPEC
0x80A7	Reserved	Reserved for future expansion
0x80A8	Microarchitectural	SVE_LDST_MULTI_SPEC
0x80A9	Microarchitectural	SVE_LD_MULTI_SPEC
0x80AA	Microarchitectural	SVE_ST_MULTI_SPEC
0x80AB	Reserved	Reserved for future expansion
0x80AC	Microarchitectural	SVE_LDST_NONCONTIG_SPEC
0x80AD	Microarchitectural	SVE_LD_GATHER_SPEC

Table A-1 Recommended PMU events (continued)

Event number	Event type	Event mnemonic
0x80AE	Microarchitectural	SVE_ST_SCATTER_SPEC
0x80AF	Microarchitectural	SVE_PRF_GATHER_SPEC
0x80B0	Microarchitectural	SVE_LDST64_NONCONTIG_SPEC
0x80B1	Microarchitectural	SVE_LD64_GATHER_SPEC
0x80B2	Microarchitectural	SVE_ST64_SCATTER_SPEC
0x80B3	Microarchitectural	SVE_PRF64_GATHER_SPEC
0x80B4	Microarchitectural	ASE_SVE_UNALIGNED_LDST_SPEC
0x80B5	Microarchitectural	ASE_SVE_UNALIGNED_LD_SPEC
0x80B6	Microarchitectural	ASE_SVE_UNALIGNED_ST_SPEC
0x80B7	Reserved	Reserved for future expansion
0x80B8	Microarchitectural	ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC
0x80B9	Microarchitectural	ASE_SVE_UNALIGNED_CONTIG_LD_SPEC
0x80BA	Microarchitectural	ASE_SVE_UNALIGNED_CONTIG_ST_SPEC
0x80BB	Reserved	Reserved for future expansion
0x80BC	Microarchitectural	SVE_LDFF_SPEC
0x80BD	Microarchitectural	SVE_LDFF_FAULT_SPEC
0x80BE - 0x80BF	Reserved	Reserved for future expansion
0x80C0	Microarchitectural	FP_SCALE_OPS_SPEC
0x80C1	Microarchitectural	FP_FIXED_OPS_SPEC
0x80C2	Microarchitectural	FP_HP_SCALE_OPS_SPEC
0x80C3	Microarchitectural	FP_HP_FIXED_OPS_SPEC
0x80C4	Microarchitectural	FP_SP_SCALE_OPS_SPEC
0x80C5	Microarchitectural	FP_SP_FIXED_OPS_SPEC
0x80C6	Microarchitectural	FP_DP_SCALE_OPS_SPEC
0x80C7	Microarchitectural	FP_DP_FIXED_OPS_SPEC
0x80C8	Microarchitectural	INT_SCALE_OPS_SPEC
0x80C9	Microarchitectural	INT_FIXED_OPS_SPEC
0x80CA	Microarchitectural	LDST_SCALE_OPS_SPEC
0x80CB	Microarchitectural	LDST_FIXED_OPS_SPEC
0x80CC	Microarchitectural	LD_SCALE_OPS_SPEC
0x80CD	Microarchitectural	LD_FIXED_OPS_SPEC
0x80CE	Microarchitectural	ST_SCALE_OPS_SPEC
0x80CF	Microarchitectural	ST_FIXED_OPS_SPEC
0x80D0 - 0x80D9	Reserved	Reserved for future expansion
0x80DA	Microarchitectural	LDST_SCALE_BYTES_SPEC
0x80DB	Microarchitectural	LDST_FIXED_BYTES_SPEC
0x80DC	Microarchitectural	LD_SCALE_BYTES_SPEC
0x80DD	Microarchitectural	LD_FIXED_BYTES_SPEC
0x80DE	Microarchitectural	ST_SCALE_BYTES_SPEC
0x80DF	Microarchitectural	ST_FIXED_BYTES_SPEC
0x80E0 - 0x80FF	Reserved	Reserved for future expansion

## A.1.1 Definitions

### CSIZE

Container size, in bits, that corresponds to the largest non-overlapping SVE or Advanced SIMD vector element size or scalar register size that is encoded in the instruction opcode. This excludes the 64-bit elements of the wide element variants of the SVE bitwise shift and integer compare instructions that overlap the narrower source and destination elements.

### MSIZE

Memory element access size, in bits, that corresponds to a load or store instruction mnemonic suffix, where B=8, H=16, W=32, or D=64. When an instruction mnemonic suffix is not specified, the memory access size is implied by the scalar transfer register size or SIMD transfer register element size.

### VL

The Vector length, in bits.

## A.1.2 Recommended PMU event descriptions

### 0x8000, SIMD\_INST\_RETIRED, SIMD instructions architecturally executed

This event counts architecturally executed SIMD instructions as follows:

- SVE instructions excluding the instructions listed in *Non-SIMD SVE instructions on page A-113*.
- Advanced SIMD instructions excluding the Advanced SIMD scalar instructions.

### 0x8001, ASE\_INST\_RETIRED, Advanced SIMD instructions architecturally executed

This event counts architecturally executed Advanced SIMD instructions. It is IMPLEMENTATION DEFINED whether this event counts Advanced SIMD scalar instructions.

### 0x8003, ASE\_SVE\_INST\_RETIRED, Advanced SIMD and SVE instructions architecturally executed

This event counts architecturally executed instructions that would be counted for ASE\_INST\_RETIRED or SVE\_INST\_RETIRED.

### 0x8004, SIMD\_INST\_SPEC, SIMD operations speculatively executed

This event counts speculatively executed operations due to:

- SVE instructions excluding the instructions listed in *Non-SIMD SVE instructions on page A-113*
- Advanced SIMD instructions excluding Advanced SIMD scalar instructions.

### 0x8005, ASE\_INST\_SPEC, Advanced SIMD operations speculatively executed

This event counts speculatively executed operations due to Advanced SIMD instructions. It is IMPLEMENTATION DEFINED whether this event counts operations due to Advanced SIMD scalar instructions.

### 0x8007, ASE\_SVE\_INST\_SPEC, Advanced SIMD and SVE operations speculatively executed

This event counts speculatively executed operations that would be counted for ASE\_INST\_SPEC or SVE\_INST\_SPEC.

**0x8008, UOP\_SPEC, microarchitectural operations speculatively executed**

This event counts all speculatively executed microarchitectural operations, irrespective of the IMPLEMENTATION DEFINED interpretation of *operation speculatively executed*.

**0x8009, ASE\_UOP\_SPEC, Advanced SIMD microarchitectural operations speculatively executed**

This event counts speculatively executed microarchitectural operations due to Advanced SIMD instructions, irrespective of the IMPLEMENTATION DEFINED interpretation of *operation speculatively executed*. It is IMPLEMENTATION DEFINED whether it counts microarchitectural operations due to Advanced SIMD scalar instructions.

**0x800A, SVE\_UOP\_SPEC, SVE micro-operation speculatively executed**

This event counts speculatively executed microarchitectural operations due to SVE instructions, irrespective of the IMPLEMENTATION DEFINED interpretation of *operation speculatively executed*. It is IMPLEMENTATION DEFINED whether this event counts microarchitectural operations due to the instructions listed in [Non-SIMD SVE instructions on page A-113](#).

**0x800B, ASE\_SVE\_UOP\_SPEC, Advanced SIMD and SVE microarchitectural operations speculatively executed**

This event counts speculatively executed microarchitectural operations as described for [SVE\\_UOP\\_SPEC](#) and [ASE\\_UOP\\_SPEC](#).

**0x800C, SIMD\_UOP\_SPEC, SIMD micro-operation speculatively executed**

This event counts speculatively executed microarchitectural operations irrespective of the IMPLEMENTATION DEFINED interpretation of *operation speculatively executed*, due to:

- SVE instructions excluding the instructions listed in [Non-SIMD SVE instructions on page A-113](#).
- Advanced SIMD instructions excluding Advanced SIMD scalar instructions.

**0x800E, SVE\_MATH\_SPEC, SVE math accelerator operation speculatively executed**

This event counts speculatively executed math function operations due to the SVE FTSMUL, FTMAD, FTSSSEL, and FEXPA instructions.

**0x8010, FP\_SPEC, floating-point operation speculatively executed**

This event counts speculatively executed operations due to scalar, Advanced SIMD, and SVE instructions listed in [Floating-point instructions on page A-110](#) and optionally [Floating-point conversions on page A-111](#) and [Floating-point or integer instructions on page A-112](#).

---

**Note**

---

This event differs from the ARMv8-A VFP\_SPEC event which does not count SIMD operations.

---

**0x8011, ASE\_FP\_SPEC, Advanced SIMD floating-point operation speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD instructions listed in [Floating-point instructions on page A-110](#) and optionally [Floating-point conversions on page A-111](#) and [Floating-point or integer instructions on page A-112](#).

**0x8012, SVE\_FP\_SPEC, SVE floating-point operation speculatively executed**

This event counts speculatively executed operations due to SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112.

**0x8013, ASE\_SVE\_FP\_SPEC, Advanced SIMD and SVE floating-point operations speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD and SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112.

**0x8014, FP\_HP\_SPEC, half-precision floating-point operation speculatively executed**

This event counts speculatively executed operations due to scalar, Advanced SIMD, and SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is half-precision.

**0x8015, ASE\_FP\_HP\_SPEC, Advanced SIMD half-precision floating-point operation speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is half-precision.

**0x8016, SVE\_FP\_HP\_SPEC, SVE half-precision floating-point operation speculatively executed**

This event counts speculatively executed operations due to SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is half-precision.

**0x8017, ASE\_SVE\_FP\_HP\_SPEC, Advanced SIMD and SVE floating-point half-precision operations speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD and SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is half-precision.

**0x8018, FP\_SP\_SPEC, single-precision floating-point operation speculatively executed**

This event counts speculatively executed operations due to scalar, Advanced SIMD, and SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is single-precision.

**0x8019, ASE\_FP\_SP\_SPEC, Advanced SIMD single-precision floating-point operations speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is single-precision.

**0x801A, SVE\_FP\_SP\_SPEC, SVE single-precision floating-point operation speculatively executed**

This event counts speculatively executed operations due to SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is single-precision.

**0x801B, ASE\_SVE\_FP\_SP\_SPEC, Advanced SIMD and SVE single-precision floating-point operations speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD and SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is single-precision.

**0x801C, FP\_DP\_SPEC, double-precision floating-point operation speculatively executed**

This event counts speculatively executed operations due to scalar, Advanced SIMD, and SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is double-precision.

**0x801D, ASE\_FP\_DP\_SPEC, Advanced SIMD double-precision floating-point operation speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is double-precision.

**0x801E, SVE\_FP\_DP\_SPEC, SVE double-precision floating-point operation speculatively executed**

This event counts speculatively executed operations due to SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is double-precision.

**0x801F, ASE\_SVE\_FP\_DP\_SPEC, Advanced SIMD and SVE double-precision floating-point operations speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD and SVE instructions listed in *Floating-point instructions* on page A-110 and optionally *Floating-point conversions* on page A-111 and *Floating-point or integer instructions* on page A-112, where the largest type is double-precision.

**0x8020, FP\_DIV\_SPEC, floating-point divide operation speculatively executed**

This event counts speculatively executed floating-point divide operations.

**0x8021, ASE\_FP\_DIV\_SPEC, Advanced SIMD floating-point divide operation speculatively executed**

This event counts speculatively executed Advanced SIMD floating-point divide operations.

**0x8022, SVE\_FP\_DIV\_SPEC, SVE floating-point divide operation speculatively executed**

This event counts speculatively executed SVE floating-point divide operations.

**0x8023, ASE\_SVE\_FP\_DIV\_SPEC, Advanced SIMD and SVE floating-point divide operations speculatively executed**

This event counts speculatively executed Advanced SIMD and SVE floating-point divide operations.

**0x8024, FP\_SQRT\_SPEC, floating-point square root operation speculatively executed**

This event counts speculatively executed floating-point square-root operations.

**0x8025, ASE\_FP\_SQRT\_SPEC, Advanced SIMD floating-point square root operation speculatively executed**

This event counts speculatively executed Advanced SIMD floating-point square-root operations.

**0x8026, SVE\_FP\_SQRT\_SPEC, SVE floating-point square root operation speculatively executed**

This event counts speculatively executed SVE floating-point square-root operations.

**0x8027, ASE\_SVE\_FP\_SQRT\_SPEC, Advanced SIMD and SVE floating-point square-root operations speculatively executed**

This event counts speculatively executed Advanced SIMD and SVE floating-point square-root operations.

**0x8028, FP\_FMA\_SPEC, floating-point FMA operation speculatively executed**

This event counts speculatively executed floating-point fused multiply-add and multiply-subtract operations due to the following instructions:

- Scalar: FMADD, FMSUB, FMADD, FNMSUB.
- Advanced SIMD: FCMLA, FMLA, FMLS.
- SVE: FCMLA, FMAD, FMLA, FMLS, FMSB, FMAD, FNMLA, FNMLS, FNMSB, FTMAD.

**0x8029, ASE\_FP\_FMA\_SPEC, Advanced SIMD floating-point FMA operation speculatively executed**

This event counts speculatively executed floating-point fused multiply-add and multiply-subtract operations due to the Advanced SIMD FCMLA, FMLA, and FMLS instructions.

**0x802A, SVE\_FP\_FMA\_SPEC, SVE floating-point FMA operation speculatively executed**

This operation speculatively executed floating-point fused multiply-add and multiply-subtract operations due to the SVE FCMLA, FMAD, FMLA, FMLS, FMSB, FMAD, FNMLA, FNMLS, FNMSB, and FTMAD instructions.

**0x802B, ASE\_SVE\_FP\_FMA\_SPEC, Advanced SIMD and SVE floating-point FMA operations speculatively executed**

This event counts speculatively executed floating-point fused multiply-add and multiply-subtract operations due to the following instructions:

- Advanced SIMD: FCMLA, FMLA, FMLS.
- SVE: FCMLA, FMAD, FMLA, FMLS, FMSB, FMAD, FNMLA, FNMLS, FNMSB, FTMAD.

**0x802C, FP\_MUL\_SPEC, floating-point multiply operation speculatively executed**

This event counts speculatively executed floating-point multiply operations due to the scalar, Advanced SIMD, and SVE FMUL and FMULX instructions and the SVE FTSMUL instruction.

**0x802D, ASE\_FP\_MUL\_SPEC, Advanced SIMD floating-point multiply operation speculatively executed**

This event counts speculatively executed floating-point multiply operations due to the scalar and Advanced SIMD `FMUL` and `FMULX` instructions.

**0x802E, SVE\_FP\_MUL\_SPEC, SVE floating-point multiply operation speculatively executed**

This event counts speculatively executed floating-point multiply operations due to the SVE `FMUL`, `FMULX`, and `FTSMUL` instructions.

**0x802F, ASE\_SVE\_FP\_MUL\_SPEC, Advanced SIMD and SVE floating-point multiply operations speculatively executed**

This event counts speculatively executed floating-point multiply operations due to the Advanced SIMD and SVE `FMUL` and `FMULX` instructions and the SVE `FTSMUL` instruction.

**0x8030, FP\_ADDSUB\_SPEC, floating-point add or subtract operation speculatively executed**

This event counts speculatively executed floating-point add and subtract operations due to the scalar, Advanced SIMD, and SVE `FADD` and `FSUB` instructions, and the Advanced SIMD and SVE `FABD` instructions.

**0x8031, ASE\_FP\_ADDSUB\_SPEC, Advanced SIMD floating-point add or subtract operation speculatively executed**

This event counts speculatively executed floating-point add and subtract operations due to the Advanced SIMD `FABD`, `FADD`, and `FSUB` instructions.

**0x8032, SVE\_FP\_ADDSUB\_SPEC, SVE floating-point add or subtract operation speculatively executed**

This event counts speculatively executed floating-point add and subtract operations due to the SVE `FABD`, `FADD`, and `FSUB` instructions.

**0x8033, ASE\_SVE\_FP\_ADDSUB\_SPEC, Advanced SIMD and SVE floating-point add and subtract operations speculatively executed**

This event counts speculatively executed floating-point add and subtract operations due to the Advanced SIMD and SVE `FABD`, `FADD`, and `FSUB` instructions.

**0x8034, FP\_RECPE\_SPEC, Floating-point reciprocal estimate operations speculatively executed**

This event counts speculatively executed floating-point reciprocal estimate operations due to the Advanced SIMD scalar, Advanced SIMD vector, and SVE `FRECPE` and `FRSQRT` instructions.

**0x8035, ASE\_FP\_RECPE\_SPEC, Advanced SIMD floating-point reciprocal estimate operations speculatively executed**

This event counts speculatively executed floating-point reciprocal estimate operations due to the Advanced SIMD vector `FRECPE` and `FRSQRT` instructions.

**0x8036, SVE\_FP\_RECPE\_SPEC, SVE floating-point reciprocal estimate operations speculatively executed**

This event counts speculatively executed floating-point reciprocal estimate operations due to the SVE `FRECPE` and `FRSQRT` instructions.

**0x8037, ASE\_SVE\_FP\_RECPE\_SPEC, Advanced SIMD and SVE floating-point reciprocal estimate operations speculatively executed**

This event counts speculatively executed floating-point reciprocal estimate operations due to Advanced SIMD vector and SVE FRECPE and FRSQRTE instructions.

**0x8038, FP\_CVT\_SPEC, floating-point convert operation speculatively executed**

This event counts speculatively executed floating-point convert operations due to the scalar, Advanced SIMD, and SVE floating-point conversion instructions listed in *Floating-point conversions on page A-111*.

**0x8039, ASE\_FP\_CVT\_SPEC, Advanced SIMD floating-point convert operation speculatively executed**

This event counts speculatively executed floating-point convert operations due to the Advanced SIMD floating-point conversion instructions listed in *Floating-point conversions on page A-111*.

**0x803A, SVE\_FP\_CVT\_SPEC, SVE floating-point convert operation speculatively executed**

This event counts speculatively executed floating-point convert operations due to the SVE floating-point conversion instructions listed in *Floating-point conversions on page A-111*.

**0x803B, ASE\_SVE\_FP\_CVT\_SPEC, Advanced SIMD and SVE floating-point convert operations speculatively executed**

This event counts speculatively executed floating-point convert operations due to the Advanced SIMD and SVE floating-point conversion instructions listed in *Floating-point conversions on page A-111*.

**0x803C, SVE\_FP\_AREDUCE\_SPEC, SVE floating-point accumulating reduction operations speculatively executed**

This event counts speculatively executed floating-point accumulating reduction operations due to the SVE FADDA instruction.

**0x803D, ASE\_FP\_PREDUCE\_SPEC, Advanced SIMD floating-point pairwise add step operations speculatively executed**

This event counts speculatively executed floating-point pairwise add operations due to the Advanced SIMD FADDP instruction.

**0x803E, SVE\_FP\_VREDUCE\_SPEC, SVE floating-point vector reduction operation speculatively executed**

This event counts speculatively executed floating-point tree-wise reduction operations due to the SVE FADDV, FMAXNMV, FMAXV, FMINNMV, and FMINV instructions.

**0x803F, ASE\_SVE\_FP\_VREDUCE\_SPEC, Advanced SIMD and SVE floating-point vector reduction operations speculatively executed**

This event counts speculatively executed floating-point reduction operations due to the Advanced SIMD and SVE FMAXNMV, FMAXV, FMINNMV, and FMINV instructions, the Advanced SIMD FADDP instruction, and the SVE FADDV instruction.

**0x8040, INT\_SPEC, integer operations speculatively executed**

This event counts speculatively executed integer arithmetic operations due to scalar, Advanced SIMD, and SVE data-processing instructions listed in *Integer instructions on page A-107* and optionally *Floating-point conversions on page A-111* and *Floating-point or integer instructions on page A-112*.

**0x8041, ASE\_INT\_SPEC, Advanced SIMD integer operations speculatively executed**

This event counts speculatively executed integer arithmetic operations due to Advanced SIMD data-processing instructions listed in *Integer instructions on page A-107* and optionally *Floating-point conversions on page A-111* and *Floating-point or integer instructions on page A-112*.

**0x8042, SVE\_INT\_SPEC, SVE integer operations speculatively executed**

This event counts speculatively executed integer arithmetic operations due to SVE data-processing instructions listed in *Integer instructions on page A-107* and optionally *Floating-point conversions on page A-111* and *Floating-point or integer instructions on page A-112*.

**0x8043, ASE\_SVE\_INT\_SPEC, Advanced SIMD and SVE integer operations speculatively executed**

This event counts speculatively executed integer arithmetic operations due to Advanced SIMD and SVE data-processing instructions listed in *Integer instructions on page A-107* and optionally *Floating-point conversions on page A-111* and *Floating-point or integer instructions on page A-112*.

**0x8044, INT\_DIV\_SPEC, integer divide operation speculatively executed**

This event counts speculatively executed scalar and SVE integer divide operations due to the SDIV and UDIV instructions.

**0x8045, INT\_DIV64\_SPEC, 64-bit integer divide operation speculatively executed**

This event counts speculatively executed scalar and SVE integer divide operations due to the SDIV and UDIV instructions with 64-bit operands or vector elements.

**0x8046, SVE\_INT\_DIV\_SPEC, SVE integer divide operation speculatively executed**

This event counts speculatively executed SVE integer divide operations due to the SVE SDIV and UDIV instructions.

**0x8047, SVE\_INT\_DIV64\_SPEC, SVE 64-bit integer divide operation speculatively executed**

This event counts speculatively executed SVE integer divide operations due to the SVE SDIV and UDIV instructions with 64-bit vector elements.

**0x8048, INT\_MUL\_SPEC, integer multiply operation speculatively executed**

This event counts speculatively executed integer multiply operations due to the following instructions:

- Scalar: MADD, MSUB, MUL, SMADDL, SMULH, UMADDL, UMULH.
- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, SMLAL, SMLS, SMULL, SQMLAL, SQDMLS, SQDMULH, SQDMULL, SQRDMLAH, SQRDMLSH, SQRDMULH, UMLAL, UMLS, UMULL.
- SVE: MAD, MLA, MLS, MSB, MUL, SMULH, UMULH.

**0x8049, ASE\_INT\_MUL\_SPEC, Advanced SIMD integer multiply operation speculatively executed**

This event counts speculatively executed integer multiply operations due to the following Advanced SIMD instructions: MLA, MLS, MUL, PMUL, PMULL, SMLAL, SMLS, SMULL, SQMLAL, SQDMLS, SQDMULH, SQDMULL, SQRDMLAH, SQRDMLSH, SQRDMULH, UMLAL, UMLS, UMULL.

**0x804A, SVE\_INT\_MUL\_SPEC, SVE integer multiply operation speculatively executed**

This event counts speculatively executed integer multiply operations due to the following SVE instructions: MAD, MLA, MLS, MSB, MUL, SMULH, UMULH.

**0x804B, ASE\_SVE\_INT\_MUL\_SPEC, Advanced SIMD and SVE integer multiply operations speculatively executed**

This event counts speculatively executed integer multiply operations due to the following instructions:

- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, SMLAL, SMLSL, SMULL, SQDMLAL, SQDMLSL, SQDMULH, SQDMULL, SQRDMLAH, SQRDMLSH, SQRDMULH, UMLAL, UMLSL, UMULL.
- SVE: MAD, MLA, MLS, MSB, MUL, SMULH, UMULH.

**0x804C, INT\_MUL64\_SPEC, integer 64x64 multiply operation speculatively executed**

This event counts speculatively executed integer multiply operations returning a 64-bit result for the following instructions:

- Scalar: MADD, MSUB, MUL, SMADDL, SMULH, UMADDL, UMULH.
- SVE: MAD, MLS, MLA, MSB, MUL, SMULH, UMULH.

**0x804D, SVE\_INT\_MUL64\_SPEC, SVE integer 64-bit multiply operation speculatively executed**

This event counts speculatively executed integer multiply operations returning a 64-bit result for the following SVE instructions: MAD, MLA, MLS, MSB, MUL, SMULH, UMULH.

**0x804E, INT\_MULH64\_SPEC, integer 64-bit multiply returning high part operation speculatively executed**

This event counts speculatively executed widening integer multiply operations returning a 64-bit result for the scalar and SVE SMULH and UMULH instructions.

**0x804F, SVE\_INT\_MULH64\_SPEC, SVE integer 64-bit multiply high part operations speculatively executed**

This event counts speculatively executed widening integer multiply operations returning a 64-bit result for the SVE SMULH and UMULH instructions.

**0x8058, NONFP\_SPEC, Non floating-point operations speculatively executed**

This event counts speculatively executed operations due to the following instructions:

- Scalar instructions that would be counted by the ARMv8 DP\_SPEC event.
- Advanced SIMD data processing instructions defined in the section titled *Data processing - SIMD and floating-point* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* that would not be counted by FP\_SPEC.
- SVE instructions with vector source or destination registers that would not be counted by FP\_SPEC.

**0x8059, ASE\_NONFP\_SPEC, Advanced SIMD non-floating-point operations speculatively executed**

This event counts speculatively executed operations due to Advanced SIMD data processing instructions defined in the section titled *Data processing - SIMD and floating-point* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* that would not be counted by ASE\_FP\_SPEC.

**0x805A, SVE\_NONFP\_SPEC, SVE non-floating-point operations speculatively executed**

This event counts speculatively executed operations due to SVE instructions with vector source or destination registers that would not be counted by SVE\_FP\_SPEC.

**0x805B, ASE\_SVE\_NONFP\_SPEC, Advanced SIMD and SVE non-floating-point operations speculatively executed**

This event counts speculatively executed operations due to the following instructions:

- Advanced SIMD data-processing instructions defined in the section titled *Data processing - SIMD and floating-point* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile* that would not be counted by [ASE\\_SVE\\_FP\\_SPEC](#).
- SVE instructions with vector source or destination registers that would not be counted by [ASE\\_SVE\\_FP\\_SPEC](#).

**0x805D, ASE\_INT\_VREDUCE\_SPEC, Advanced SIMD integer reduction operation speculatively executed**

This event counts speculatively executed across-vector and pairwise integer reduction operations due to the Advanced SIMD SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADDVL, UMAXV, and UMINV instructions.

**0x805E, SVE\_INT\_VREDUCE\_SPEC, SVE integer reduction operation speculatively executed**

This event counts speculatively executed across-vector integer reduction operations due to the following SVE instructions: ANDV, EORV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, and UMINV instructions.

**0x805F, ASE\_SVE\_INT\_VREDUCE\_SPEC, Advanced SIMD and SVE integer reduction operations speculatively executed**

This event counts speculatively executed across-vector and pairwise integer reduction operations due to the following instructions:

- Advanced SIMD: SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADDLV, UMAXV, and UMINV.
- SVE: ANDV, EORV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, UMINV.

**0x8060, SVE\_PERM\_SPEC, SVE permute operation speculatively executed**

This event counts speculatively executed vector or predicate permute operations due to the following SVE instructions: CLASTA, CLASTB, CPY, COMPACT, DUP, EXT, INSR, LASTA, LASTB, PUNPKHI, PUNPKLO, REV, REV16, REV32, REV64, SPLICE, SUNPKHI, SUNPKLO, TBL, TRN1, TRN2, UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, and ZIP2.

**0x8061, SVE\_PERM\_IGRANULE\_SPEC, SVE intra-granule permute operations speculatively executed**

This event counts speculatively executed vector or predicate permute operations within a 128-bit vector granule or 16-bit predicate granule for the following SVE instructions: REV16, REV32, REV64, TRN1, TRN2.

**0x8062, SVE\_PERM\_XGRANULE\_SPEC, SVE cross-granule permute operations speculatively executed**

This event counts speculatively executed vector or predicate permute operations that can cross between 128-bit vector granules or 16-bit predicate granules for the following SVE instructions: CLASTA, CLASTB, CPY, COMPACT, DUP, EXT, INSR, LASTA, LASTB, PUNPKHI, PUNPKLO, REV, SPLICE, SUNPKHI, SUNPKLO, TBL, UNPKHI, UNPKLO, UZP1, UZP2, ZIP1, and ZIP2.

**0x8063, SVE\_PERM\_VARIABLE\_SPEC, SVE programmable permute operations speculatively executed**

This event counts speculatively executed variable vector permute operations due to the following SVE instructions: CLASTA, CLASTB, COMPACT, LASTA, LASTB, SPLICE, and TBL.

**0x8064, SVE\_XPIPE\_SPEC, SVE cross-pipe operations speculatively executed**

This event counts speculatively executed cross-pipeline transfer operations due to the following SVE instructions: CLASTA (scalar), CLASTB (scalar), CNTP, CPY (scalar), DECP (scalar), DUP (scalar), INCP (scalar), INDEX (immediate, scalar), INDEX (scalar, immediate), INDEX (scalar, scalar), INSR (scalar), LASTA (scalar), LASTB (scalar), SQDECP (scalar), SQINCP (scalar), UQDECP (scalar), UQDECP (scalar), WHILE<cc>.

**0x8065, SVE\_XPIPE\_Z2R\_SPEC, SVE vector to scalar cross-pipe operations speculatively executed**

This event counts speculatively executed vector to general-purpose scalar cross-pipeline transfer operations due to the following SVE instructions: CLASTA (scalar), CLASTB (scalar), CNTP, DECP (scalar), INCP (scalar), LASTA (scalar), LASTB (scalar), SQDECP (scalar), SQINCP (scalar), UQDECP (scalar), UQDECP (scalar).

**0x8066, SVE\_XPIPE\_R2Z\_SPEC, SVE scalar to vector cross-pipe operations speculatively executed**

This event counts speculatively executed general-purpose scalar to vector cross-pipeline transfer operations due to the following SVE instructions: CPY (scalar), DUP (scalar), INDEX (immediate, scalar), INDEX (scalar, immediate), INDEX (scalar, scalar), INSR (scalar), WHILE<cc>.

**0x8067, SVE\_PGEN\_NVEC\_SPEC, SVE predicate-only operations speculatively executed**

This event counts speculatively executed predicate-generating operations that do not read Vector registers due to the following SVE instructions: AND (predicates), ANDS, BIC (predicates), BICS, BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, EOR (predicates), EORS, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), ORRS, PFALSE, PFIRST, PNEXT, PTRUE, PTRUES, PUNPKHI, PUNPKLO, RDIFFR, RDIFFRS, REV (predicate), SEL (predicates), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), WHILE<cc>, ZIP1 (predicates), ZIP2 (predicates).

**0x8068, SVE\_PGEN\_SPEC, SVE predicate generating operations speculatively executed**

This event counts speculatively executed predicate-generating operations due to the following SVE instructions: AND (predicates), ANDS, BIC (predicates), BICS, BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, CMP<cc>, EOR (predicates), EORS, FAC<cc>, FCM<cc>, NAND, NANDS, NOR, NORS, ORN (predicates), ORNS, ORR (predicates), ORRS, PFALSE, PFIRST, PNEXT, PTRUE, PTRUES, PUNPKHI, PUNPKLO, RDIFFR, RDIFFRS, REV (predicate), SEL (predicates), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), WHILE<cc>, ZIP1 (predicates), ZIP2 (predicates).

**0x8069, SVE\_PGEN\_FLG\_SPEC, SVE predicate flag setting operations speculatively executed**

This event counts speculatively executed predicate-generating operations that set condition flags, due to the following SVE instructions: ANDS, BICS, BRKAS, BRKBS, BRKNS, BRKPAS, BRKPBS, CMP<cc>, EORS, NANDS, NORS, ORNS, ORRS, PFIRST, PNEXT, PTRUES, RDIFFRS, WHILE<cc>.

**0x806A, SVE\_PGEN\_CMP\_SPEC, SVE vector compare operations speculatively executed**

This event counts speculatively executed vector compare operations due to the following SVE instructions: CMP<cc>, FAC<cc>, FCM<cc>.

**0x806B, SVE\_PGEN\_FCM\_SPEC, SVE floating-point vector compare operations speculatively executed**

This event counts speculatively executed vector floating-point compare operations, due to the following SVE instructions: FAC<cc>, FCM<cc>.

**0x806C, SVE\_PGEN\_LOGIC\_SPEC, SVE predicate logical operations speculatively executed**

This event counts speculatively executed predicate logical operations, due to the following SVE instructions: AND (predicates), ANDS (predicates), BICS, EOR (predicates), EORS, NAND, NANDS, NOR, NORs, ORN (predicates), ORNS, ORR (predicates), ORRS.

**0x806D, SVE\_PPERM\_SPEC, SVE predicate permute operations speculatively executed**

This event counts speculatively executed predicate permute operations, due to the following SVE instructions: PUNPKHI, PUNPKLO, REV (predicate), TRN1 (predicates), TRN2 (predicates), UZP1 (predicates), UZP2 (predicates), ZIP1 (predicates), ZIP2 (predicates).

**0x806E, SVE\_PSCAN\_SPEC, SVE predicate scan operations speculatively executed**

This event counts speculatively executed predicate scanning and generation operations, due to the following SVE instructions: BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, PFIRST, PNEXT.

**0x806F, SVE\_PCNT\_SPEC, SVE predicate count operations speculatively executed**

This event counts speculatively executed predicate population count operations, due to the following SVE instructions: CNTP, DECP, INCP, SQDECP, SQINCP, UQDECP, UQINCP.

**0x8070, SVE\_PLOOP\_WHILE\_SPEC, SVE predicate loop while operations speculatively executed**

This event counts speculatively executed counted predicate generation operations, due to the following SVE instructions: WHILELE, WHILELO, WHILELS, WHILELT.

**0x8071, SVE\_PLOOP\_TEST\_SPEC, SVE predicate loop test operations speculatively executed**

This event counts speculatively executed loop predicate test operations, due to the following SVE instructions: BRKAS, BRKBS, BRKNS, BRKPAS, BRKPBS, WHILELE, WHILELO, WHILELS, WHILELT.

**0x8072, SVE\_PLOOP\_ELTS\_SPEC, SVE predicate loop elements speculatively executed**

This event counts speculatively executed loop predicate generation operations, due to the following SVE instructions: WHILELE, WHILELO, WHILELS, WHILELT. This event increments the counter by  $(128 \div \text{CSIZE})$ .

**———— Note ————**

This counter must be multiplied by  $(\text{VL} \div 128)$  to determine the number of vector elements speculatively processed by while loops.

**0x8073, SVE\_PLOOP\_TERM\_SPEC, SVE predicate loop termination speculatively executed**

This event counts speculatively executed loop-terminating predicate generation operations due to the following SVE instructions:

- WHILELE, WHILELO, WHILELS, WHILELT, which set PSTATE.N to 0.
- BRKAS, BRKBS, BRKNS, BRKPAS, BRKPBS, which set PSTATE.C to 1.
- CTERM EQ and CTERM NE, which set PSTATE.N to 1 and PSTATE.V to 0.

**0x8074, SVE\_PRED\_SPEC, SVE predicated operations speculatively executed**

This event counts speculatively executed SIMD data-processing and load/store operations due to SVE instructions with a Governing predicate operand that determines the Active elements.

**0x8075, SVE\_PRED\_EMPTY\_SPEC, SVE predicated operations with no active predicates speculatively executed**

This event counts speculatively executed SIMD data-processing and load/store operations due to SVE instructions with a Governing predicate in which all elements are FALSE.

**0x8076, SVE\_PRED\_FULL\_SPEC, SVE predicated operations speculatively executed with all active predicates**

This event counts speculatively executed SIMD data-processing and load/store operations due to SVE instructions with a Governing predicate in which all elements are TRUE.

**0x8077, SVE\_PRED\_PARTIAL\_SPEC, SVE predicated operations speculatively executed with partially active predicates**

This event counts speculatively executed SIMD data-processing and load/store operations due to SVE instructions with a Governing predicate in which elements are neither all TRUE nor all FALSE.

**0x8078, SVE\_UNPRED\_SPEC, SVE unpredicated operations speculatively executed**

This event counts speculatively executed SIMD data-processing and load/store operations due to SVE instructions without a Governing predicate.

**0x807C, SVE\_MOVPRFX\_SPEC, SVE MOVPRFX operations speculatively executed**

This event counts speculatively executed operations due to MOVPRFX instructions, whether or not they were fused with the Prefixed instruction.

**0x807D, SVE\_MOVPRFX\_Z\_SPEC, SVE MOVPRFX zeroing predication operations speculatively executed**

This event counts speculatively executed operations due to MOVPRFX instructions using Zeroing predication, whether or not they were fused with the Prefixed instruction.

**0x807E, SVE\_MOVPRFX\_M\_SPEC, SVE MOVPRFX merging predication operations speculatively executed**

This event counts speculatively executed operations due to MOVPRFX instructions using Merging predication, whether or not they were fused with the Prefixed instruction.

**0x807F, SVE\_MOVPRFX\_U\_SPEC, SVE MOVPRFX unfused operations speculatively executed**

This event counts speculatively executed operations due to MOVPRFX instructions that were not fused with the Prefixed instruction.

**0x8080, SVE\_LDST\_SPEC, SVE load/store/prefetch operations speculatively executed**

This event counts speculatively executed operations that read from, write to, or prefetch memory due to SVE instructions.

**0x8081, SVE\_LD\_SPEC, SVE load operations speculatively executed**

This event counts speculatively executed operations that read from memory due to SVE load instructions.

**0x8082, SVE\_ST\_SPEC, SVE store operations speculatively executed**

This event counts speculatively executed operations that write to memory due to SVE store instructions.

**0x8083, SVE\_PRF\_SPEC, SVE prefetch operations speculatively executed**

This event counts speculatively executed operations that prefetch memory due to SVE prefetch instructions.

**0x8084, ASE\_SVE\_LDST\_SPEC, Advanced SIMD and SVE load/store operations speculatively executed**

This event counts speculatively executed operations that read from or write to memory due to SVE and Advanced SIMD instructions, or any instructions that prefetch memory.

**0x8085, ASE\_SVE\_LD\_SPEC, Advanced SIMD and SVE load operations speculatively executed**

This event counts speculatively executed operations that read from memory due to SVE and Advanced SIMD load instructions.

**0x8086, ASE\_SVE\_ST\_SPEC, Advanced SIMD and SVE store operations speculatively executed**

This event counts speculatively executed operations that write to memory due to SVE and Advanced SIMD store instructions.

**0x8087, PRF\_SPEC, Prefetch operations speculatively executed**

This event counts speculatively executed prefetch operations due to scalar PRFM and SVE PRF instructions.

**0x8088, BASE\_LDST\_REG\_SPEC, General-purpose register load/store/prefetch operations speculatively executed**

This event counts speculatively executed operations that read from memory to a general-purpose register, write a general-purpose register to memory, or prefetch memory due to the PRFM instruction. It is IMPLEMENTATION DEFINED whether operations due to the DC ZVA instruction are counted.

**0x8089, BASE\_LD\_REG\_SPEC, General-purpose register load operations speculatively executed**

This event counts speculatively executed operations that read from memory due to an instruction that loads a general-purpose register.

**0x808A, BASE\_ST\_REG\_SPEC, General-purpose register store operations speculatively executed**

This event counts speculatively executed operations that write to memory due to an instruction that stores a general-purpose register. It is IMPLEMENTATION DEFINED whether operations due to the DC ZVA instruction are counted.

**0x808B, BASE\_PRF\_SPEC, General-purpose register prefetch operations speculatively executed**

This event counts speculatively executed operations that prefetch memory due to the PRFM instruction.

**0x808C, FPASE\_LDST\_REG\_SPEC, Floating-point and Advanced SIMD register load/store operations speculatively executed**

This event counts speculatively executed operations that read from or write to memory, due to scalar SIMD&FP LDR, LDP, STR, and STP instructions or Advanced SIMD LD1, LD1R, and ST1 instructions.

**0x808D, FPASE\_LD\_REG\_SPEC, Floating-point and Advanced SIMD register load operations speculatively executed**

This event counts speculatively executed operations that read from memory, due to scalar SIMD&FP LDR and LDP instructions or Advanced SIMD LD1 and LD1R instructions.

**0x808E, FPASE\_ST\_REG\_SPEC, Floating-point and Advanced SIMD register store operations speculatively executed**

This event counts speculatively executed operations that write to memory, due to scalar SIMD&FP STR and STP instructions or Advanced SIMD ST1 instructions.

**0x8090, SVE\_LDST\_REG\_SPEC, SVE unpredicated load/store register operations speculatively executed**

This event counts speculatively executed operations that read from memory or write to memory due to SVE LDR and STR instructions.

**0x8091, SVE\_LDR\_REG\_SPEC, SVE unpredicated load register operations speculatively executed**

This event counts speculatively executed operations that read from memory due to an SVE LDR instruction.

**0x8092, SVE\_STR\_REG\_SPEC, SVE unpredicated store register operations speculatively executed**

This event counts speculatively executed operations that write to memory due to an SVE STR instruction.

**0x8094, SVE\_LDST\_PREG\_SPEC, SVE load/store Predicate register operations speculatively executed**

This event counts speculatively executed operations that read from memory or write to memory due to SVE LDR (predicate) and STR (predicate) instructions.

**0x8095, SVE\_LDR\_PREG\_SPEC, SVE load Predicate register operations speculatively executed**

This event counts speculatively executed operations that read from memory due to an SVE LDR (predicate) instruction.

**0x8096, SVE\_STR\_PREG\_SPEC, SVE store Predicate register operations speculatively executed**

This event counts speculatively executed operations that write to memory due to an SVE STR (predicate) instruction.

**0x8098, SVE\_LDST\_ZREG\_SPEC, SVE load/store Vector register operations speculatively executed**

This event counts speculatively executed operations that read from memory or write to memory due to SVE LDR (vector) and STR (vector) instructions.

**0x8099, SVE\_LDR\_ZREG\_SPEC, SVE load Vector register operations speculatively executed**

This event counts speculatively executed operations that read from memory due to an SVE LDR (vector) instruction.

**0x809A, SVE\_STR\_ZREG\_SPEC, SVE store Vector register operations speculatively executed**

This event counts speculatively executed operations that write to memory due to an SVE STR (vector) instruction.

**0x809C, SVE\_LDST\_CONTIG\_SPEC, SVE contiguous load/store/prefetch element operations speculatively executed**

This event counts speculatively executed operations that read from, write to, or prefetch memory due to an SVE predicated single vector contiguous element load, store, or prefetch instruction. Operations due to SVE load and replicate LD1R and LD1RQ instructions are also counted.

**0x809D, SVE\_LD\_CONTIG\_SPEC, SVE contiguous load element operations speculatively executed**

This event counts speculatively executed operations that read from memory due to SVE predicated single vector contiguous element load instructions. Operations due to SVE load and replicate LD1R and LD1RQ instructions are also counted.

**0x809E, SVE\_ST\_CONTIG\_SPEC, SVE contiguous store element operations speculatively executed**

This event counts speculatively executed operations that write to memory due to SVE predicated single vector contiguous element store instructions.

**0x809F, SVE\_PRF\_CONTIG\_SPEC, SVE contiguous prefetch element operations speculatively executed**

This event counts speculatively executed operations that prefetch memory due to an SVE predicated single contiguous element prefetch instruction.

**0x80A0, SVE\_LDSTNT\_CONTIG\_SPEC, SVE non-temporal contiguous load/store element operations speculatively executed**

This event counts speculatively executed operations that read from memory or write to memory with a non-temporal hint due to an SVE non-temporal contiguous element load or store instruction.

**0x80A1, SVE\_LDNT\_CONTIG\_SPEC, SVE non-temporal contiguous load element operations speculatively executed**

This event counts speculatively executed operations that read from memory with a non-temporal hint due to an SVE non-temporal contiguous element load instruction.

**0x80A2, SVE\_STNT\_CONTIG\_SPEC, SVE non-temporal contiguous store element operations speculatively executed**

This event counts speculatively executed operations that write to memory with a non-temporal hint due to an SVE non-temporal contiguous element store instruction.

**0x80A4, ASE\_SVE\_LDST\_MULTI\_SPEC, Advanced SIMD and SVE contiguous load/store multiple vector operations speculatively executed**

This event counts speculatively executed operations that read from memory or write to memory due to an SVE or Advanced SIMD multiple vector contiguous structure load and store instruction.

**0x80A5, ASE\_SVE\_LD\_MULTI\_SPEC, Advanced SIMD and SVE contiguous load multiple vector operations speculatively executed**

This event counts speculatively executed operations that read from memory due to SVE and Advanced SIMD multiple vector contiguous structure load instructions.

**0x80A6, ASE\_SVE\_ST\_MULTI\_SPEC, Advanced SIMD and SVE contiguous store multiple vector operations speculatively executed**

This event counts speculatively executed operations that write to memory due to SVE and Advanced SIMD multiple vector contiguous structure store instructions.

**0x80A8, SVE\_LDST\_MULTI\_SPEC, SVE contiguous load/store multiple vector operations speculatively executed**

This event counts speculatively executed operations that read from memory or write to memory due to SVE multiple vector contiguous structure load and store instructions.

**0x80A9, SVE\_LD\_MULTI\_SPEC, SVE contiguous load multiple vector operations speculatively executed**

This event counts speculatively executed operations that read from memory due to SVE multiple vector contiguous structure load instructions.

**0x80AA, SVE\_ST\_MULTI\_SPEC, SVE contiguous store multiple vector operations speculatively executed**

This event counts speculatively executed operations that write to memory due to SVE multiple vector contiguous structure store instructions.

**0x80AC, SVE\_LDST\_NONCONTIG\_SPEC, SVE non-contiguous load/store/prefetch operations speculatively executed**

This event counts speculatively executed operations that read from, write to, or prefetch memory due to SVE non-contiguous gather-load, scatter-store, and gather-prefetch instructions.

**0x80AD, SVE\_LD\_GATHER\_SPEC, SVE gather-load operations speculatively executed**

This event counts speculatively executed operations that read from memory due to SVE non-contiguous gather-load instructions.

**0x80AE, SVE\_ST\_SCATTER\_SPEC, SVE scatter-store operations speculatively executed**

This event counts speculatively executed operations that write to memory due to SVE non-contiguous scatter-store instructions.

**0x80AF, SVE\_PRF\_GATHER\_SPEC, SVE gather-prefetch operations speculatively executed**

This event counts speculatively executed operations that prefetch memory due to SVE non-contiguous gather-prefetch instructions.

**0x80B0, SVE\_LDST64\_NONCONTIG\_SPEC, SVE 64-bit non-contiguous load/store/prefetch operations speculatively executed**

This event counts speculatively executed operations that read from, write to, or prefetch memory due to SVE non-contiguous gather-load, scatter-store, and gather-prefetch instructions with 64-bit vector elements in the address.

**0x80B1, SVE\_LD64\_GATHER\_SPEC, SVE 64-bit gather-load operations speculatively executed**

This event counts speculatively executed operations that read from memory due to SVE non-contiguous gather-load instructions with 64-bit vector elements in the address.

**0x80B2, SVE\_ST64\_SCATTER\_SPEC, SVE 64-bit scatter-store operations speculatively executed**

This event counts speculatively executed operations that write to memory due to SVE non-contiguous scatter-store instructions with 64-bit vector elements in the address.

**0x80B3, SVE\_PRF64\_GATHER\_SPEC, SVE 64-bit gather-prefetch operations speculatively executed**

This event counts speculatively executed operations that prefetch memory due to SVE non-contiguous gather-prefetch instructions with 64-bit vector elements in the address.

**0x80B4, ASE\_SVE\_UNALIGNED\_LDST\_SPEC, Advanced SIMD and SVE unaligned accesses**

This event counts memory read and write accesses due to SVE and Advanced SIMD load and store instructions where:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather/scatter or single element address is not aligned to the memory element access size, in bytes.

This event also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

**0x80B5, ASE\_SVE\_UNALIGNED\_LD\_SPEC, Advanced SIMD and SVE unaligned read accesses**

This event counts memory read and write accesses due to SVE and Advanced SIMD load instructions where:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather/scatter or single element address is not aligned to the memory element access size, in bytes.

This event also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

**0x80B6, ASE\_SVE\_UNALIGNED\_ST\_SPEC, Advanced SIMD and SVE unaligned write accesses**

This event counts memory read and write accesses due to SVE and Advanced SIMD store instructions where:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather/scatter or single element address is not aligned to the memory element access size, in bytes.

This event also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

**0x80B8, ASE\_SVE\_UNALIGNED\_CONTIG\_LDST\_SPEC, Advanced SIMD and SVE unaligned contiguous accesses**

This event counts memory read and write accesses due to SVE and Advanced SIMD contiguous load and store instructions where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

This event also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

**0x80B9, ASE\_SVE\_UNALIGNED\_CONTIG\_LD\_SPEC, Advanced SIMD and SVE unaligned contiguous read accesses**

This event counts memory read accesses due to SVE and Advanced SIMD contiguous load instructions where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

This event also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

**0x80BA, ASE\_SVE\_UNALIGNED\_CONTIG\_ST\_SPEC, Advanced SIMD and SVE unaligned contiguous write accesses**

This event counts memory write accesses due to SVE and Advanced SIMD contiguous store instructions where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

This event also counts unaligned accesses if they are subsequently converted into multiple aligned accesses.

**0x80BC, SVE\_LDFD\_SPEC, SVE First-fault load operations speculatively executed**

This event counts speculatively executed memory read operations due to SVE First-fault and Non-fault load instructions.

**0x80BD, SVE\_LDFD\_FAULT\_SPEC, SVE First-fault load operations speculatively executed which set FFR bit to 0**

This event counts speculatively executed memory read operations due to SVE First-fault and Non-fault load instructions that write 0 to at least one bit in FFR.

**0x80C0, FP\_SCALE\_OPS\_SPEC, Scalable floating-point element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [SVE\\_FP\\_SPEC](#), except that it is IMPLEMENTATION DEFINED whether operations due to instructions other than those listed in *Floating-point arithmetic (SVE)* on page A-111 are counted. The counter is incremented by  $(128 \div \text{CSIZE})$  and by twice that amount for operations that would also be counted by [SVE\\_FP\\_FMA\\_SPEC](#).

**0x80C1, FP\_FIXED\_OPS\_SPEC, Non-scalable floating-point element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [FP\\_SPEC](#) but not by [SVE\\_FP\\_SPEC](#), and it is IMPLEMENTATION DEFINED whether operations due to instructions other than those listed in *Floating-point arithmetic (scalar)* on page A-110 and *Floating-point arithmetic (Advanced SIMD)* on page A-110 are counted. The counter is incremented by the specified number of elements for Advanced SIMD operations or by 1 for scalar operations, and by twice those amounts for operations that would also be counted by [FP\\_FMA\\_SPEC](#).

**0x80C2, FP\_HP\_SCALE\_OPS\_SPEC, Scalable half-precision floating-point element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [SVE\\_FP\\_HP\\_SPEC](#), except that it is IMPLEMENTATION DEFINED whether operations due to instructions other than those listed in *Floating-point arithmetic (SVE)* on page A-111 are counted. The counter is incremented by 8, or by 16 for operations that would also be counted by [SVE\\_FP\\_FMA\\_SPEC](#).

**0x80C3, FP\_HP\_FIXED\_OPS\_SPEC, Non-scalable half-precision floating-point element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [FP\\_HP\\_SPEC](#) but not by [SVE\\_FP\\_HP\\_SPEC](#), and it is IMPLEMENTATION DEFINED whether operations due to instructions other than those listed in [Floating-point arithmetic \(scalar\) on page A-110](#) and [Floating-point arithmetic \(Advanced SIMD\) on page A-110](#) are counted. The counter is incremented by the number of 16-bit elements for Advanced SIMD operations, or by 1 for scalar operations, and by twice those amounts for operations that would also be counted by [FP\\_FMA\\_SPEC](#).

**0x80C4, FP\_SP\_SCALE\_OPS\_SPEC, Scalable single-precision floating-point element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [SVE\\_FP\\_SP\\_SPEC](#), except that it is IMPLEMENTATION DEFINED whether operations other than those listed in [Floating-point arithmetic \(SVE\) on page A-111](#) are counted. The counter is incremented by 4, or by 8 for operations that would also be counted by [SVE\\_FP\\_FMA\\_SPEC](#).

**0x80C5, FP\_SP\_FIXED\_OPS\_SPEC, Non-scalable single-precision floating-point element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [FP\\_SP\\_SPEC](#) but not by [SVE\\_FP\\_SP\\_SPEC](#), and it is IMPLEMENTATION DEFINED whether due to instructions other than those listed in [Floating-point arithmetic \(scalar\) on page A-110](#) and [Floating-point arithmetic \(Advanced SIMD\) on page A-110](#) are counted. The counter is incremented by the number of 32-bit elements for Advanced SIMD operations, or by 1 for scalar operations, and by twice those amounts for operations that would also be counted by [FP\\_FMA\\_SPEC](#).

**0x80C6, FP\_DP\_SCALE\_OPS\_SPEC, Scalable double-precision floating-point element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [SVE\\_FP\\_DP\\_SPEC](#), except that it is IMPLEMENTATION DEFINED whether operations due to instructions other than those listed in [Floating-point arithmetic \(SVE\) on page A-111](#) are counted. The counter is incremented by 2, or by 4 for operations that would also be counted by [SVE\\_FP\\_FMA\\_SPEC](#).

**0x80C7, FP\_DP\_FIXED\_OPS\_SPEC, Non-scalable double-precision floating-point element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [FP\\_DP\\_SPEC](#) but not by [SVE\\_FP\\_DP\\_SPEC](#), and it is IMPLEMENTATION DEFINED whether operations due to instructions other than those listed in [Floating-point arithmetic \(scalar\) on page A-110](#) and [Floating-point arithmetic \(Advanced SIMD\) on page A-110](#) are counted. The counter is incremented by 2 for Advanced SIMD operations, or by 1 for scalar operations, and by twice those amounts for operations that would also be counted by [FP\\_FMA\\_SPEC](#).

**0x80C8, INT\_SCALE\_OPS\_SPEC, Scalable integer element operations speculatively executed**

This event counts speculatively executed operations that would be counted by [SVE\\_INT\\_SPEC](#). The counter is incremented by  $(128 \div \text{CSIZE})$ , and by twice that amount for operations due to the SVE MAD, MLA, and MSB instructions.

**0x80C9, INT\_FIXED\_OPS\_SPEC, Non-scalable integer element operations speculatively executed**

This event counts speculatively executed operations that would be counted by `INT_SPEC` but not by `SVE_INT_SPEC`. The counter is incremented by the specified number of elements for Advanced SIMD operations, or by 1 for scalar operations, and by twice those amounts for operations due to:

- Scalar: `MADD`, `MSUB`, `SMADDL`, `SMSUBL`, `UMADD`, `UMSUBL`.
- Advanced SIMD: `MLA`, `MLS`, `SMLAL`, `SMLS`, `SQDMLAL`, `SQDMLS`, `SQRDMLAH`, `SQRDMLSH`, `UMLAL`, `UMLS`.

**0x80CA, LDST\_SCALE\_OPS\_SPEC, Scalable load/store element operations speculatively executed**

This event counts speculatively executed memory read and write operations, due to the SVE predicated vector load and store instructions, excluding the replicating `LD1R` and `LD1RQ` instructions. For each instruction, the counter is incremented by  $(128 \div \text{CSIZE})$ , multiplied by the number of transferred Vector register.

**0x80CB, LDST\_FIXED\_OPS\_SPEC, Non-scalable load/store element operations speculatively executed**

This event counts speculatively executed memory read and write operations as follows:

- Loading or storing a single scalar register increments the counter by 1.
- Loading or storing a pair of scalar registers increments the counter by 2.
- An atomic store instruction increments the counter by 1.
- An atomic load instruction increments the counter by 2.
- SVE and Advanced SIMD `LD1R` instructions increment the counter by 1.
- SVE `LD1RQ` instructions increment the counter by  $(128 \div \text{CSIZE})$ .
- Advanced SIMD `LD[1-4]` and `ST[1-4]` instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.

**0x80CC, LD\_SCALE\_OPS\_SPEC, Scalable load element operations speculatively executed**

This event counts speculatively executed memory read operations, due to SVE predicated vector load instructions, excluding the replicating `LD1R` and `LD1RQ` instructions. For each instruction, the counter is incremented by  $(128 \div \text{CSIZE})$ , multiplied by the number of transferred registers.

**0x80CD, LD\_FIXED\_OPS\_SPEC, Non-scalable load element operations speculatively executed**

This event counts speculatively executed memory read operations as follows:

- Loading a single scalar register increments the counter by 1.
- Loading a pair of scalar registers increments the counter by 2.
- An atomic load instruction increments the counter by 1.
- SVE and Advanced SIMD `LD1R` instructions increment the counter by 1.
- SVE `LD1RQ` instructions increment the counter by  $(128 \div \text{CSIZE})$ .
- Advanced SIMD `LD[1-4]` instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.

**0x80CE, ST\_SCALE\_OPS\_SPEC, Scalable store element operations speculatively executed**

This event counts speculatively executed memory write operations, due to SVE predicated vector store instructions. For each instruction, the counter is incremented by  $(128 \div \text{CSIZE})$ , multiplied by the number of transferred registers.

**0x80CF, ST\_FIXED\_OPS\_SPEC, Non-scalable store element operations speculatively executed**

This event counts speculatively executed memory write operations as follows:

- Storing a single scalar register increments the counter by 1.
- Storing a pair of scalar registers increments the counter by 2.
- An atomic store instruction increments the counter by 1.
- Advanced SIMD ST[1-4] instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.

**0x80DA, LDST\_SCALE\_BYTES\_SPEC, Scalable load/store bytes speculatively executed**

This event counts bytes speculatively read or written due to SVE vector load and store instructions, excluding the replicating LD1R and LD1RQ instructions. For each instruction, the counter is incremented by  $(16 \div (\text{CSIZE} \div \text{MSIZE}))$ , multiplied by the number of transferred Vector registers.

**0x80DB, LDST\_FIXED\_BYTES\_SPEC, Non-scalable load/store bytes speculatively executed**

This event counts bytes speculatively read and written as follows:

- Non-SVE LDR and STR instructions increment the counter by  $(\text{MSIZE} \div 8)$ .
- LDP, LDNP, STP, and STNP instructions increment the counter by  $2 \times (\text{MSIZE} \div 8)$ .
- Atomic store instructions increment the counter by  $(\text{MSIZE} \div 8)$ .
- Atomic load instructions increment the counter by  $2 \times (\text{MSIZE} \div 8)$ .
- SVE and Advanced SIMD LD1R instructions increment the counter by  $(\text{MSIZE} \div 8)$ .
- SVE LD1RQ instructions increment the counter by 16.
- Advanced SIMD LD[1-4] and ST[1-4] instructions increment the counter by the number of bytes being transferred per register multiplied by the number of registers transferred.

**0x80DC, LD\_SCALE\_BYTES\_SPEC, Scalable load bytes speculatively executed**

This event counts bytes speculatively read due to SVE vector load instructions, excluding the replicating LD1R and LD1RQ instructions. For each instruction, the counter is incremented by  $(16 \div (\text{CSIZE} \div \text{MSIZE}))$ , multiplied by the number of transferred Vector registers.

**0x80DD, LD\_FIXED\_BYTES\_SPEC, Non-scalable load bytes speculatively executed**

This event counts bytes speculatively read as follows:

- Non-SVE LDR instructions increment the counter by  $(\text{MSIZE} \div 8)$ .
- LDP and LDNP instructions increment the counter by  $(2 \times (\text{MSIZE} \div 8))$ .
- Atomic load instructions increment the counter by  $(\text{MSIZE} \div 8)$ .
- SVE and Advanced SIMD LD1R instructions increment the counter by  $(\text{MSIZE} \div 8)$ .
- SVE LD1RQ instructions increment the counter by 16.
- Advanced SIMD LD[1-4] instructions increment the counter by the number of bytes read per register multiplied by the number of registers transferred.

**0x80DE, ST\_SCALE\_BYTES\_SPEC, Scalable store bytes speculatively executed**

This event counts bytes speculatively written due to SVE vector store instructions. For each instruction, the counter is incremented by  $(16 \div (\text{CSIZE} \div \text{MSIZE}))$ , multiplied by the number of transferred Vector registers.

**0x80DF, ST\_FIXED\_BYTES\_SPEC, Non-scalable store bytes speculatively executed**

This event counts bytes speculatively written as follows:

- Non-SVE STR instructions increment the counter by  $(MSIZE \div 8)$ .
- STP and STNP instructions increment the counter by  $2 \times (MSIZE \div 8)$ .
- Atomic load and store instructions increment the counter by  $(MSIZE \div 8)$ .
- Advanced SIMD ST[1-4] instructions increment the counter by the number of bytes read per register multiplied by the number of registers transferred.

## A.2 Interesting combinations of SVE events

### A.2.1 Scalar-equivalent operations

The number of speculatively executed operations performed on individual scalar values, assuming that all SVE Vector elements are active, can be determined from a pair of event counters. For example, the total number of individual floating-point operations performed can be computed as follows:

$$FP\_SCALE\_OPS\_SPEC \times VL \div 128 + FP\_FIXED\_OPS\_SPEC$$

A summary of these event pairs is given in [Table A-2](#). Note that combined multiply-add and multiply-subtract instructions are counted as two operations per element.

**Table A-2 Total operation count pairs**

Operation type	Scalable operations	Fixed width operations
Floating-point operations (any precision)	FP_SCALE_OPS_SPEC	FP_FIXED_OPS_SPEC
Half-precision floating-point operations	FP_HP_SCALE_OPS_SPEC	FP_HP_FIXED_OPS_SPEC
Single-precision floating-point operations	FP_SP_SCALE_OPS_SPEC	FP_SP_FIXED_OPS_SPEC
Double-precision floating-point operations	FP_DP_SCALE_OPS_SPEC	FP_DP_FIXED_OPS_SPEC
Integer operations (any size)	INT_SCALE_OPS_SPEC	INT_FIXED_OPS_SPEC
Load/store accesses (any size)	LDST_SCALE_OPS_SPEC	LDST_FIXED_OPS_SPEC
Load accesses (any size)	LD_SCALE_OPS_SPEC	LD_FIXED_OPS_SPEC
Store accesses (any size)	ST_SCALE_OPS_SPEC	ST_FIXED_OPS_SPEC

### A.2.2 Bytes loaded and stored

The number of bytes speculatively loaded from memory or stored to memory, assuming that all SVE Vector elements are active, can be determined from a pair of event counters. For example, the total number of bytes loaded from memory can be computed as follows:

$$LD\_SCALE\_BYTES\_SPEC \times VL \div 128 + LD\_FIXED\_BYTES\_SPEC$$

**Table A-3 Total byte count pairs**

Operation type	Scalable operations	Fixed width operations
Load/store byte count	LDST_SCALE_BYTES_SPEC	LDST_FIXED_BYTES_SPEC
Load byte count	LD_SCALE_BYTES_SPEC	LD_FIXED_BYTES_SPEC
Store byte count	ST_SCALE_BYTES_SPEC	ST_FIXED_BYTES_SPEC

### A.2.3 Overall vector utilization

ARM does not recommend the accumulation of an active predicate population count, or predicate weight, by every predicated SVE instruction. However, the vector utilization can be estimated using one or more of the ratios of events shown in [Table A-4](#) and the result used to adjust SVE event counters that ignore the predicate weight.

**Table A-4 Vector utilization ratios**

Utilization rate	Ratio
All predicates active	$\text{SVE\_PRED\_FULL\_SPEC} \div \text{SVE\_PRED\_SPEC}$
Partial predicates active	$\text{SVE\_PRED\_PARTIAL\_SPEC} \div \text{SVE\_PRED\_SPEC}$
No predicates active	$\text{SVE\_PRED\_EMPTY\_SPEC} \div \text{SVE\_PRED\_SPEC}$

Regions of code generating a high frequency of [SVE\\_PRED\\_EMPTY\\_SPEC](#) events might indicate where the addition of a `B.NONE` conditional branch around a block of predicated code would avoid executing instructions that frequently generate no useful result.

### A.2.4 Vector loop efficiency

The effectiveness with which sequential or scalar source loops are vectorized can be estimated using ratios of the [SVE\\_PLOOP\\_\\*\\_SPEC](#) predicated loop events, as shown in [Table A-5](#).

**Table A-5 Vector loop efficiency ratios**

Vector loop metric	Ratio
Source level iterations per loop	$\text{SVE\_PLOOP\_ELTS\_SPEC} \div \text{SVE\_PLOOP\_TERM\_SPEC}$
Vectorized iterations per loop	$\text{SVE\_PLOOP\_TEST\_SPEC} \div \text{SVE\_PLOOP\_TERM\_SPEC}$
Parallelism per vector loop	$\text{SVE\_PLOOP\_ELTS\_SPEC} \div \text{SVE\_PLOOP\_TEST\_SPEC}$

## A.3 Instruction categories

### A.3.1 Data movement instructions

#### Data movement (scalar)

- FCSEL
- FMOV (immediate)
- FMOV (general)
- FMOV (register)

#### Data movement (Advanced SIMD)

- DUP
- EXT
- FMOV (vector, immediate)
- INS
- SMOV
- TBL
- TBX
- TRN1, TRN2
- UMOV
- UZP1, UZP2
- XTN, XTN2
- ZIP1, ZIP2

#### Data movement (SVE)

- CLASTA, CLASTB
- COMPACT
- CPY (scalar)
- CPY (immediate) [MOV (immediate, predicated) alias]
- DUP (scalar)
- DUP (immediate) [MOV (immediate, unpredicated) alias]
- EXT
- FCPY [FMOV (immediate, predicated) alias]
- FDUP [FMOV (immediate, unpredicated) alias]
- INDEX
- INSR
- LASTA, LASTB
- MOVPRFX
- REV (vector)
- SEL (vectors)
- SPLICE
- SUNPKHI, SUNPKLO
- TBL
- TRN1, TRN2 (vectors)
- UUNPKHI, UUNPKLO
- UZP1, UZP2 (vectors)
- ZIP1, ZIP2 (vectors)

## A.3.2 Integer instructions

### Integer (scalar)

#### *Integer uniform arithmetic (scalar)*

- ADC, ADCS
- ADD, ADDS
- CCMN, CCMP
- CSINC, CSINV, CSNEG
- MADD, MSUB
- SBC, SBCS
- SDIV, UDIV
- SMULH, UMULH
- SUB, SUBS
- ADR, ADRP

#### *Integer widening arithmetic*

- SMADDL, SMSUBL, UMADDL, UMSUBL

#### *Integer bitwise operations (scalar)*

- AND, ANDS, BIC, BICS, EOR, EON, ORR, ORN
- ASRV, LSLV, LSRV, RORV
- BFM, SBFM, UBFM
- CLS, CLZ
- EXTR
- RBIT, REV, REV16, REV32

### Integer (Advanced SIMD)

#### *Integer uniform arithmetic (Advanced SIMD)*

- ABS, NEG
- ADD, SUB
- MLA, MLS
- MUL, PMUL
- SABA, UABA
- SABD, UABD
- SDOT, UDOT
- SHADD, SHSUB, SRHADD, UHADD, UHSUB, URHADD
- SMAX, SMIN, UMAX, UMIN
- SQABS, SQNEG
- SQADD, SQSUB, SUQADD, UQADD, USQADD, UQSUB
- SQDMULH, SQRDMULH
- SQRDMLAH, SQRDMLSH
- URECPE, URSQRTE
- USRA

**Integer widening arithmetic (Advanced SIMD)**

- SABAL, SABAL2, UABAL, UABAL2
- SABDL, SABDL2, UABDL, UABDL2
- SADDL, SADDL2, UADDL, UADDL2
- SADDW, SADDW2, UADDW, UADDW2
- SMLAL, SMLAL2, UMLAL, UMLAL2
- SMLS, SMLS2, UMLS, UMLS2
- SMULL, SMULL2, UMULL, UMULL2, PMULL, PMULL2
- SQDMULL, SQDMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2
- SHLL, SHLL2, SSHLL, SSHLL2, USHLL, USHLL2
- SSUBL, SSUBL2, USUBL, USUBL2
- SSUBW, SSUBW2, USUBW, USUBW2
- UXTL, UXTL2

**Integer narrowing arithmetic (Advanced SIMD)**

- ADDHN, ADDHN2, RADDHN, RADDHN2
- SUBHN, SUBHN2, RSUBHN, RSUBHN2
- SHRN, SHRN2, RSHRN, RSHRN2
- SQRSHRN, SQRSHRN2, SQRSHRUN, SQRSHRUN2, UQRS
- SQSHRN, SQSHRN2, SQSHRUN, SQSHRUN2, UQSHRN, UQSHRN2
- SQXTN, SQXTN2, SQXTUN, SQXTUN2, UQXTN, UQXTN2

**Integer bitwise operations (Advanced SIMD)**

- AND, BIC, EOR, ORN, ORR
- BIF, BIT, BSL
- CLS, CLZ, CNT
- MOVI, MVNI
- NOT
- RBIT, REV16, REV32, REV64
- SHL, SRSHL, URSHL
- SRSHR, URSHR
- SRSRA, SSRA, URSRA
- SLI, SRI
- SQRSHL, QSQSHL, QSQSHLU, UQRSHL, UQSHL
- SSSL, USHL
- SSHR, USHR

**Integer comparisons (Advanced SIMD)**

- CMEQ, CMGE, CMGT, CMHI, CMHS, CMLE, CMLT, CMTST

**Integer reductions (Advanced SIMD)**

- ADDP, ADDV
- SADALP, UADALP
- SADDLP, SADDLV, UADDLP, UADDLV
- SMAXP, SMAXV, UMAXP, UMAXV

- SMINP, SMINV, UMINP, UMINV

## **Integer (SVE)**

### ***Integer uniform arithmetic (SVE)***

- ABS, NEG
- ADD, SUB, SUBR
- ADR
- CNOT
- MAD, MSB
- MLA, MLS
- MUL
- SABD, UABD
- SDIV, SDIVR, UDIV, UDIVR
- SDOT, UDOT
- SMAX, SMIN, UMAX, UMIN
- SMULH, UMULH
- SQADD, SQSUB, UQADD, UQSUB
- SXTB, SXTH, SXTW, UXTB, UXTH, UXTW

### ***Integer bitwise operations (SVE)***

- AND, BIC, EON, EOR, ORN, ORR (vectors)
- ASR, ASRR
- ASRD
- CLS, CLZ, CNT
- DUPM
- LSL, LSLR
- LSR, LSRR
- NOT (vector)
- RBIT, REVB, REVH, REVW

### ***Integer comparisons (SVE)***

- CMPEQ, CMPGE, CMPGT, CMPHI, CMPHS, CMPL, CMPLO, CMPLS, CMPLT, CMPNE

### ***Integer reductions (SVE)***

- ANDV, EORV, ORV
- SADDV, UADDV
- SMAXV, UMAXV
- SMINV, UMINV

**Element count and increment vector (SVE)**

- DECH, DECW, DECD (vector)
- INCH, INCW, INCD (vector)
- SQDECH, SQDECW, SQDECD (vector)
- SQINCH, SQINCW, SQINCD (vector)
- UQDECH, UQDECW, UQDECD (vector)
- UQINCH, UQINCW, UQINCD (vector)

**A.3.3 Floating-point instructions****Floating-point (scalar)****Floating-point arithmetic (scalar)**

- FADD, FSUB (scalar)
- FDIV (scalar)
- FMADD, FMSUB, FNMADD, FNMSUB (scalar)
- FMUL, FNMUL (scalar)
- FSQRT (scalar)

**Floating-point miscellaneous (scalar)**

- FMAX, FMAXNM (scalar)
- FMIN, FMINNM (scalar)
- FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ (scalar)

**Floating-point comparisons (scalar)**

- FCMP, FCMPE

**Floating-point (Advanced SIMD)****Floating-point arithmetic (Advanced SIMD)**

- FABD, FADD, FSUB (vector)
- FCADD, FCMLA
- FDIV (vector)
- FMLA, FMLS
- FMUL, FMULX (vector)
- FRECPX, FRSQRTS
- FSQRT (vector)

**Floating-point miscellaneous (Advanced SIMD)**

- FMAX, FMAXNM (vector)
- FMIN, FMINNM (vector)
- FRECPX
- FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ (vector)

**Floating-point comparisons (Advanced SIMD)**

- FACGE, FACGT
- FCMEQ, FCMGE, FCMGT, FCMLE, FCMLT

**Floating-point reductions (Advanced SIMD)**

- FADDP
- FMAXNMP, FMAXP
- FMAXNMV, FMAXV
- FMINNMP, FMINP
- FMINNMV, FMINV

**Floating-point (SVE)****Floating-point arithmetic (SVE)**

- FABD, FADD, FSUB, FSUBR
- FCADD, FCMLA
- FDIV, FDIVR
- FMAD, FMAD, FNMSB, FMSB
- FMLA, FMLS, FNMLA, FNMLS
- FMUL, FMULX
- FRECP5, FRSQRT5
- FSCALE
- FSQRT
- FTMAD, FTSMUL

**Floating-point miscellaneous (SVE)**

- FMAX, FMAXNM
- FMIN, FMINNM
- FRECPX
- FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ

**Floating-point comparisons (SVE)**

- FACGE, FACGT, FACLE, FACLT
- FCMEQ, FCMGE, FCMGT, FCMLE, FCMLT, FCMNE, FCMUO

**Floating-point reductions (SVE)**

- FADDA, FADDV
- FMAXNMV, FMAXV
- FMINNMV, FMINV

**A.3.4 Floating-point conversions****Float↔Float convert (scalar)**

- FCVT

**Float↔Float convert (Advanced SIMD)**

- FCVTL, FCVTL2
- FCVTN, FCVTN2
- FCVTXN, FCVTXN2

**Float↔Float convert (SVE)**

- FCVT

**Float↔Int convert (scalar)**

- FCVTAS, FCVTMS, FCVTNS, FCVTPS, FCVTZS (scalar)
- FCVTAU, FCVTMU, FCVTNU, FCVTPU, FCVTZU (scalar)
- FJCVTZS
- SCVTF, UCVTF (scalar)

**Float↔Int convert (Advanced SIMD)**

- FCVTAS, FCVTMS, FCVTNS, FCVTPS, FCVTZS (vector)
- FCVTAU, FCVTMU, FCVTNU, FCVTPU, FCVTZU (vector)
- SCVTF, UCVTF (vector)

**Float↔Int convert (SVE)**

- FCVTZS, FCVTZU
- SCVTF, UCVTF

**A.3.5 Floating-point or integer instructions****Floating-point or integer arithmetic (scalar)**

- FABS, FNEG (scalar)

**Floating-point or integer arithmetic (Advanced SIMD)**

- FABS, FNEG (vector)
- FRECPE, FRSQRTE

**Floating-point or integer arithmetic (SVE)**

- FABS, FNEG
- FRECPE, FRSQRTE
- FEXPA, FTSSSEL

### A.3.6 Non-SIMD SVE instructions

#### Element count and increment scalar (SVE)

- ADDPL, ADDVL, RDVL
- CNTB, CNTH, CNTW, CNTD
- DECB, DECH, DECW, DECD (scalar)
- INCB, INCH, INCW, INCD (scalar)
- SQDECB, SQDECH, SQDECW, SQDECD (scalar)
- SQINCB, SQINCH, SQINCW, SQINCD (scalar)
- UQDECB, UQDECH, UQDECW, UQDECD (scalar)
- UQINCB, UQINCH, UQINCW, UQINCD (scalar)

#### Compare and terminate (SVE)

- CTERM EQ, CTERMNE

### A.3.7 Predicate instructions

#### Predicate move (SVE)

- PFALSE, PTRUE, PTRUES
- PUNPKHI, PUNPKLO
- RDIFFR, RDIFFRS (predicated)
- RDIFFR, SETFFR, WRFFR (unpredicated)
- REV (predicate)
- SEL (predicates)
- TRN1, TRN2 (predicates)
- UZP1, UZP2 (predicates)
- ZIP1, ZIP2 (predicates)

#### Predicate counted loop (SVE)

- WHILELE, WHILELO, WHILELS, WHILELT

#### Predicate bitwise logical operations (SVE)

- AND, ANDS (predicates)
- BIC, BICS (predicates)
- EOR, EORS (predicates)
- NAND, NANDS
- NOR, NORS
- NOT, NOTS (predicate)
- ORN, ORNS (predicates)
- ORR, ORRS (predicates)
- PTEST

**Predicate scan (SVE)**

- BRKA, BRKAS, BRKB, BRKBS
- BRKN, BRKNS
- BRKPA, BRKPAS, BRKPB, BRKPBS
- PFIRST,

**Predicate count and increment scalar (SVE)**

- CNTP, DECP, INCP (scalar)
- SQDECP, SQINCP (scalar)
- UQDECP, UQINCP (scalar)

**Predicate count and increment vector (SVE)**

- DECP, INCP (vector)
- SQDECP, SQINCP (vector)
- UQDECP, UQINCP (vector)

**A.3.8 Cryptographic instructions****Cryptographic (Advanced SIMD)**

- AESD, AESE
- AESIMC, AESMC
- SHA1C, SHA1H, SHA1M, SHA1P
- SHA1SU0, SHA1SU1
- SHA256H, SHA256H2
- SHA256SU0, SHA256SU1

**A.3.9 Load/store/prefetch instructions****Load/store (Advanced SIMD & FP scalar)****Contiguous elements load/store (Advanced SIMD)**

- LD1 (multiple structures)
- ST1 (multiple structures)

**Contiguous structures load/store (Advanced SIMD)**

- LD2, LD3, LD4 (multiple structures)
- ST2, ST3, ST4 (multiple structures)

**Single element/structure load/store (Advanced SIMD)**

- LD1, LD2, LD3, LD4 (single structure)
- ST1, ST2, ST3, ST4 (single structure)

**Single element/structure replicating load (Advanced SIMD)**

- LD1R, LD2R, LD3R, LD4R

**Register load/store (Advanced SIMD & FP scalar)**

- LDNP (SIMD&FP)
- LDP (SIMD&FP)
- LDR (SIMD&FP)
- LDUR (SIMD&FP)
- STNP (SIMD&FP)
- STP (SIMD&FP)
- STR (SIMD&FP)
- STUR (SIMD&FP)

**Load/store/prefetch (SVE)****Contiguous elements load/store/prefetch (SVE)**

- LD1B, LD1H, LD1W, LD1D, LD1SB, LD1SH, LD1SW (scalar, immediate)
- LD1B, LD1H, LD1W, LD1D, LD1SB, LD1SH, LD1SW (scalars)
- LDFF1B, LDFF1H, LDFF1W, LDFF1D, LDFF1SB, LDFF1SH, LDFF1SW (scalars)
- LDNF1B, LDNF1H, LDNF1W, LDNF1D, LDNF1SB, LDNF1SH, LDNF1SW
- LDNT1B, LDNT1H, LDNT1W, LDNT1D (scalar, immediate)
- LDNT1B, LDNT1H, LDNT1W, LDNT1D (scalars)
- PRFB, PRFH, PRFW, PRFD (scalar, immediate)
- PRFB, PRFH, PRFW, PRFD (scalars)
- ST1B, ST1H, ST1W, ST1D (scalar, immediate)
- ST1B, ST1H, ST1W, ST1D (scalars)
- STNT1B, STNT1H, STNT1W, STNT1D (scalar, immediate)
- STNT1B, STNT1H, STNT1W, STNT1D (scalars)

**Contiguous structures load/store (SVE)**

- LD2B, LD2H, LD2W, LD2D (scalar, immediate)
- LD2B, LD2H, LD2W, LD2D (scalars)
- LD3B, LD3H, LD3W, LD3D (scalar, immediate)
- LD3B, LD3H, LD3W, LD3D (scalars)
- LD4B, LD4H, LD4W, LD4D (scalar, immediate)
- LD4B, LD4H, LD4W, LD4D (scalars)
- ST2B, ST2H, ST2W, ST2D (scalar, immediate)
- ST2B, ST2H, ST2W, ST2D (scalars)
- ST3B, ST3H, ST3W, ST3D (scalar, immediate)
- ST3B, ST3H, ST3W, ST3D (scalars)
- ST4B, ST4H, ST4W, ST4D (scalar, immediate)
- ST4B, ST4H, ST4W, ST4D (scalars)

**Gather/scatter load/store/prefetch (SVE)**

- LD1B, LD1H, LD1W, LD1D, LD1SB, LD1SH, LD1SW (vector, immediate)
- LD1B, LD1H, LD1W, LD1D, LD1SB, LD1SH, LD1SW (scalar, vector)
- LDFF1B, LDFF1H, LDFF1W, LDFF1D, LDFF1SB, LDFF1SH, LDFF1SW (vector, immediate)
- LDFF1B, LDFF1H, LDFF1W, LDFF1D, LDFF1SB, LDFF1SH, LDFF1SW (scalar, vector)
- PRFB, PRFH, PRFW, PRFD (vector, immediate)
- PRFB, PRFH, PRFW, PRFD (scalar, vector)
- ST1B, ST1H, ST1W, ST1D (vector, immediate)
- ST1B, ST1H, ST1W, ST1D (scalar, vector)

**Single element load and replicate (SVE)**

- LD1RB, LD1RH, LD1RW, LD1RD, LD1RSB, LD1RSH, LD1RSW

**Single quadword load and replicate (SVE)**

- LD1RQB, LD1RQH, LD1RQW, LD1RQD (scalar, immediate)
- LD1RQB, LD1RQH, LD1RQW, LD1RQD (scalars)

**Register load/store (SVE)**

- LDR (predicate)
- LDR (vector)
- STR (predicate)