Arm® Architecture Reference Manual Supplement

The Scalable Vector Extension (SVE), for Armv8-A

arm

ARM DDI 0584A.g (ID022020)

Arm Architecture Reference Manual Supplement
The Scalable Vector Extension (SVE), for Armv8-A

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 release
21 August 2017	A.b	Non-Confidential	EAC release
15 December 2017	A.c	Non-Confidential	EAC maintenance release
21 December 2017	A.d	Non-Confidential	EAC maintenance release
31 October 2018	A.e	Non-Confidential	EAC maintenance release
5 July 2019	A.f	Non-Confidential	EAC maintenance release
21 Feb 2020	A.g	Non-Confidential	Updated EAC release incorporating the BFloat and Matrix Multiplication instructions

Proprietary Notice

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

LES-PRE-20349

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

Note

- The term Arm can refer to versions of the Arm architecture, for example Armv8 refers to version 8 of the Arm architecture. The context makes it clear when the term is used in this way.
- This document describes only the Armv8-A architecture profile. For the behaviors required by the previous version of this architecture profile, ARMv7-A, see the *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

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`

Contents

Arm Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for Armv8-A

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

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 *SVE 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
italic	Introduces special terminology, and denotes citations.
bold	Denotes signal names, and is used for terms in descriptive lists, where appropriate.
`monospace`	Used for assembler syntax descriptions, pseudocode, and source code examples. Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.
<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: • A URL, for example `http://developer.arm.com`. • A cross-reference, that includes the page number of the referenced information if it is not on the current page. • A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term.

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, https://developer.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 Armv8.6*.
- *A64 ISA XML for Armv8.6*.

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`. Give:

- The title.
- The number, ARM DDI 0584A.g.
- 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:

1.1 About the SVE supplement

This supplement must be read with the *Arm® 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:

- ARMv8.2-FP16 half-precision floating-point must be implemented.
- 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 *Configurable vector length* on page 3-30.
- Predication. See *Predicate registers* on page 2-21.
- Gather-load and scatter-store. See *Load, store, and prefetch instructions* on page 5-41.
- Software-managed speculative vectorization. See *First Fault Register, FFR* on page 2-22.

1.2.1 Features within SVE

If SVE is implemented:

- ARMv8.2-BF16 BFloat16 instructions are OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations. If SVE and AArch64 Advanced SIMD are both implemented, they must agree on the presence of ARMv8.2-BF16.
- ARMv8.2-I8MM matrix multiplication instructions are OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations. If SVE and AArch64 Advanced SIMD are both implemented, they must agree on the presence of ARMv8.2-I8MM.
- ARMv8.2-F32MM and ARMv8.2-F64MM matrix multiplication instructions are OPTIONAL for SVE in Armv8.2.

For more information, see the section titled *Features added to the Armv8.2 extension in later releases* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.

1.3 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.

1.4 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.

First active element

The First active element of a vector or predicate register is defined as the lowest numbered element 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.

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 highest numbered element that is an Active element.

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.

1.5 Register disambiguation

In some sections of this manual, registers are referred to by a general name, where the description applies to more than one context. This is because the description applies to multiple Exception levels, and therefore at a particular Exception level the register names need to take the appropriate Exception level suffix, _EL0, _EL1, _EL2, or _EL3.

1.5.1 Register name disambiguation by Exception level

Table 1-1 disambiguates the general names of the registers by Execution state.

Table 1-1 Disambiguation of System registers by Exception level

General form	EL0	EL1	EL2	EL3
ELR_ELx	-	ELR_EL1	ELR_EL2	ELR_EL3
ESR_ELx	-	ESR_EL1	ESR_EL2	ESR_EL3
FAR_ELx	-	FAR_EL1	FAR_EL2	FAR_EL3
SCTLR_ELx	-	SCTLR_EL1	SCTLR_EL2	SCTLR_EL3
ZCR_ELx	-	ZCR_EL1	ZCR_EL2	ZCR_EL3

Chapter 2
SVE Application Level Programmers' Model

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

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 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 3-30 for more information.

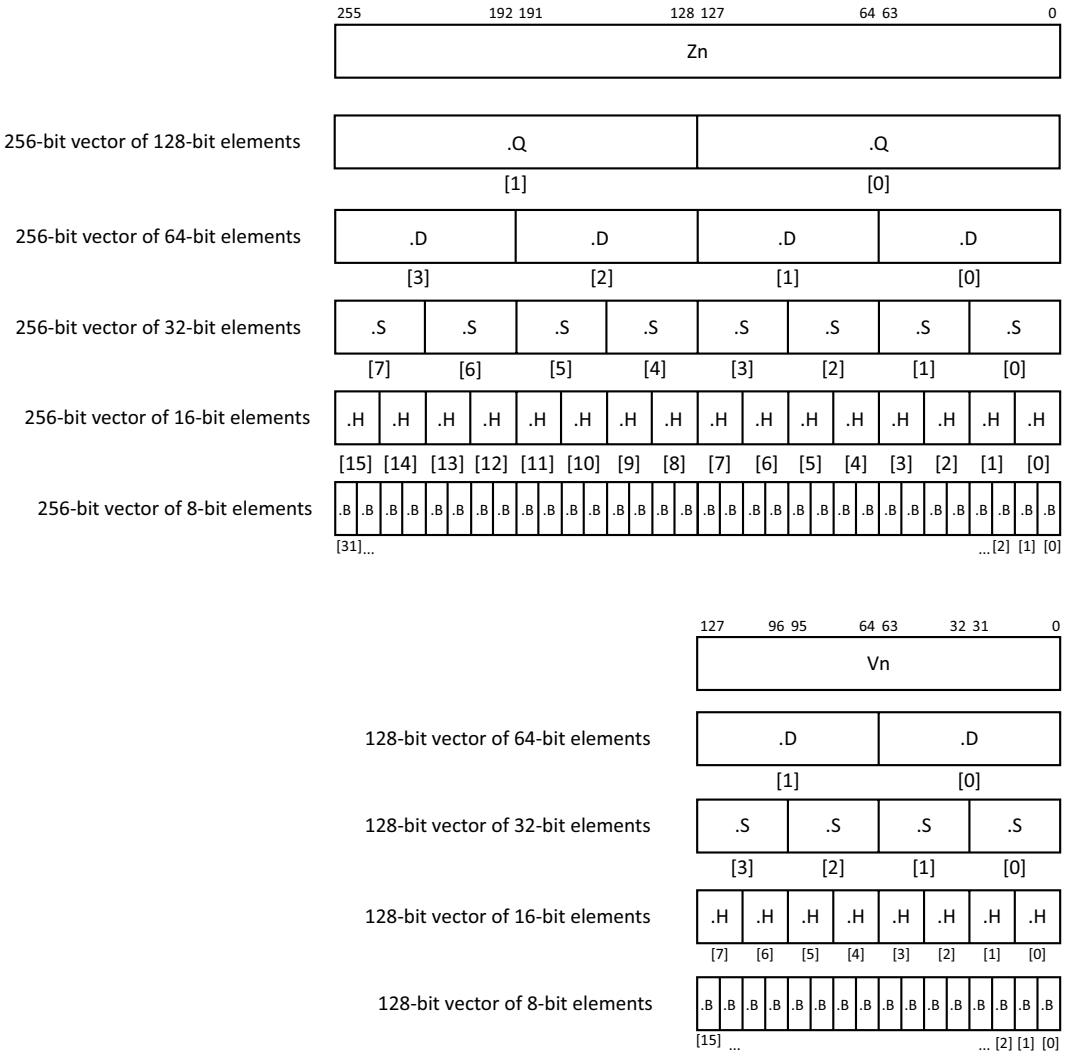

Figure 2-1 SVE and SIMD&FP vectors in AArch64 state

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 5-64, 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 5-57 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. Many predicated instructions can only use P0-P7 as the Governing predicate, refer to the individual instruction descriptions for details.

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 destination register, either:

- The Inactive elements in the destination are set to zero. This is known as zeroing predication.
- The Inactive elements in the destination 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, Zn, and the associated 32-bit Governing predicate register, Pg. 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.

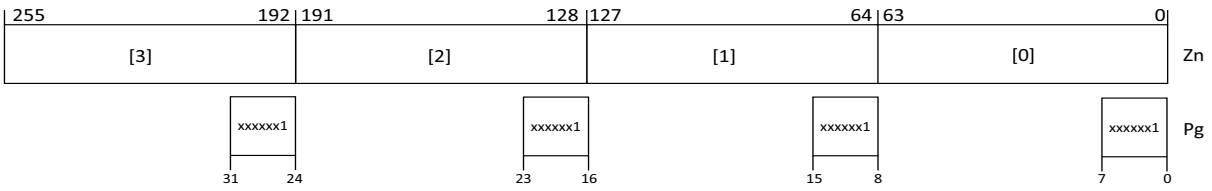

Figure 2-2 256-bit vector, 4x64-bit packed elements

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.

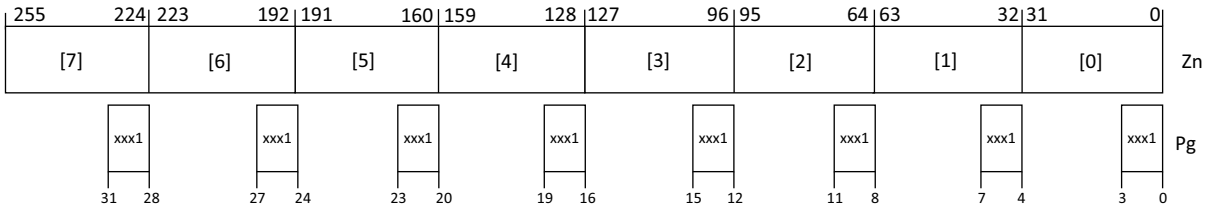

Figure 2-3 256-bit vector, 8x32-bit packed elements

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.

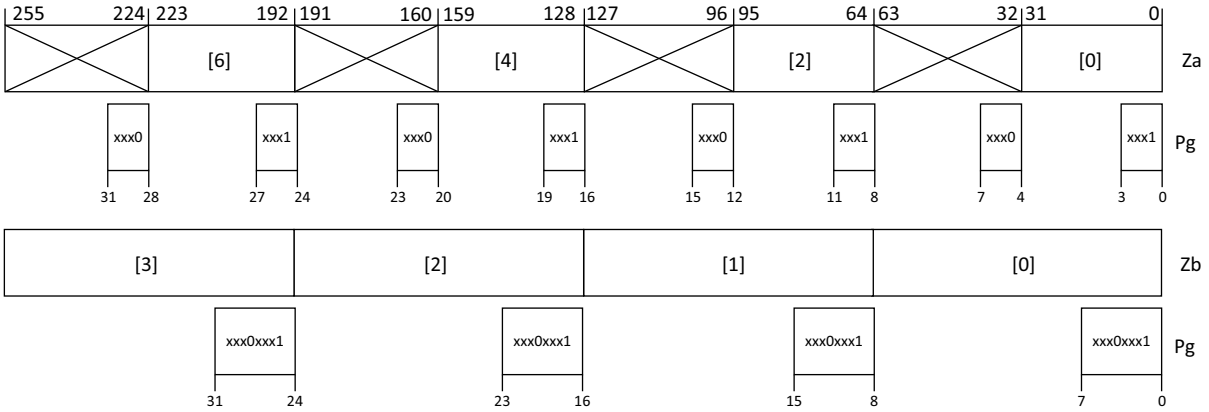

Figure 2-4 Governing predicate interpretation for different vector organizations

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 read the FFR are:

- RDFFR
- RDFFRS

The only instructions that directly write the 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.

See *Synchronous memory faults* on page 3-28 for more information on SVE First-fault and Non-fault loads.

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 CTERMEQ and CTERMNE.

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 no Active elements 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.

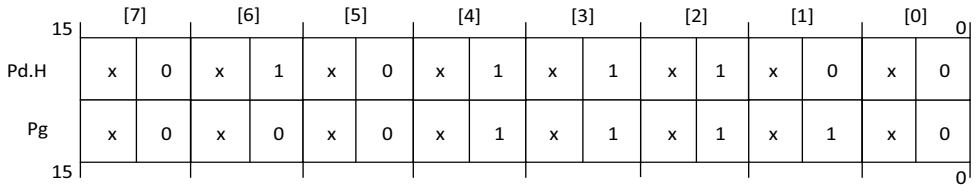

Figure 2-5 First active element

In Figure 2-5 on page 2-24, `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].

Chapter 3
SVE System Level Programmers' Model

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

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® 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 5-65*. When a `MOVPRFX` instruction is used as described in *Move prefix on page 5-65* 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 ELR_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 ELR_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

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_ELx, 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 one of the following:
  - If the corresponding FFR element is FALSE and it was an Active element, then each byte of the element has an independently CONSTRAINED UNPREDICTABLE choice of either zero, the previous value of that byte in the destination vector register, or the value read from memory if and only if the access for that byte was not to any type of Device memory, and does not return information that cannot be accessed at the current or a lower level of privilege.

    Note

    For the avoidance of doubt, a watchpoint is not a mechanism for preventing access to memory.

  - If the corresponding FFR element is FALSE and it was an Inactive element, a CONSTRAINED UNPREDICTABLE choice of either zero, or the previous value of that vector element.
  - If the corresponding FFR element is TRUE and it was an Active element, the value read from memory.
  - If the corresponding FFR element is TRUE and it was an Inactive element, zero.
- 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.
- Alignment fault, excluding the SP alignment 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.

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 at that Exception level and at less privileged Exception levels. 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.

Table 3-1 Configurable vector lengths

	Additionally configurable vector lengths	
Maximum configurable vector length	**Required**	**Permitted**
128	128	-
256	128, 256	-
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

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.

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 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.

If SVE instructions are disabled or trapped at ELx, or not available because 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.

If floating-point and SVE are both disabled, trapped, or not available 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.

Note

The ZCR_ELx.LEN field constrains the SVE vector length to be no greater than (LEN+1)×128 bits, at Exception level ELx and below. See the ZCR_EL1, ZCR_EL2, and ZCR_EL3 System register descriptions for more details.

An indirect read of ZCR_EL1.LEN, ZCR_EL2.LEN, or ZCR_EL3.LEN appears to occur in program order relative to a direct write of the same register, without the need for explicit synchronization.

Chapter 4
SVE Memory Model

This chapter introduces the changes to the Armv8-A memory model introduced by SVE. This chapter contains the following sections:

4.1 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.

4.2 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 SCTLR_ELx.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 5-41 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 SCTLR_ELx 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.

4.3 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.

4.4 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.
- If an address dependency exists between two memory reads, and an SVE non-temporal vector load instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

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.

4.4.1 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 *Synchronous memory faults* on page 3-28.
- 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 *Synchronous memory faults* on page 3-28.
- Hardware speculation of data accesses performed to a Device memory location is not permitted by the architecture, with the following exceptions:
  - Reads that are performed by an SVE unpredicated 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.
  - Reads that are performed by an SVE predicated load instruction are permitted to access bytes that are not explicitly accessed by an Active element of 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 an Active element of the instruction.
  - 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 an Active element of 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 an Active element of the instruction.

4.5 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 the described CONSTRAINED UNPREDICTABLE behaviors.
- Where the CONSTRAINED UNPREDICTABLE behavior is to generate an Alignment fault then this is handled consistently with the handling of an Alignment fault described in *Synchronous memory faults* on page 3-28.

Chapter 5
SVE Instruction Set

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

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 5-41 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 memory 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 5-43.
- *Predicated non-contiguous element accesses* on page 5-44.
- *Predicated replicating element loads* on page 5-44.
- *Unpredicated vector register load/store* on page 5-45.
- *Unpredicated predicate register load/store* on page 5-45.

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 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.

When included as part of the assembler syntax for an instruction, `MUL VL` indicates that the specified immediate index value is multiplied by the size of the addressed vector or predicate in memory, measured in bytes, irrespective of predication. For a detailed description of the meaning of this assembler syntax for each instruction, see the appropriate subsection below.

Note

When used in pseudocode, the symbol `VL` represents the vector length, measured in bits.

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.

Predicated contiguous prefetch instructions address 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×N to 7×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 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. 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.

	Assembler syntax	
Supported addressing modes	64-bit elements	32-bit elements
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.

Supported addressing modes	Assembler syntax
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>{, #<simm>, 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>{, #<simm>, 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	CPY
	Copy general-purpose register to vector elements	CPY
	Copy SIMD&FP scalar register to vector elements	CPY
DUP	Broadcast signed immediate to vector elements	DUP
	Broadcast general-purpose register to vector elements	DUP
FCPY	Copy 8-bit floating-point immediate to vector elements	FCPY
FDUP	Broadcast 8-bit floating-point immediate to vector elements	FDUP
FMOV	Move floating-point +0.0 to vector elements (unpredicated)	FMOV
	Move floating-point +0.0 to vector elements (predicated)	FMOV
	Move 8-bit floating-point immediate to vector elements (unpredicated)	FMOV
	Move 8-bit floating-point immediate to vector element (predicated)	FMOV
MOV	Move signed integer immediate to vector elements (unpredicated)	MOV
	Move signed integer immediate to vector elements (predicated)	MOV
	Move general-purpose register to vector elements (unpredicated)	MOV
	Move general-purpose register to vector elements (predicated)	MOV
	Move SIMD&FP scalar register to vector elements (unpredicated)	MOV
	Move SIMD&FP scalar register to vector elements (predicated)	MOV
	Move vector register (unpredicated)	MOV
	Move vector register (predicated)	MOV
SEL	Select vector elements from two vectors	SEL

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	ABS
ADD	Add vectors (predicated)	ADD
	Add vectors (unpredicated)	ADD
	Add immediate	ADD
CNOT	Logically invert Boolean condition	CNOT
MAD	Multiply-add, writing to the multiplicand register	MAD

Table 5-4 Integer arithmetic instructions (continued)

Mnemonic	Instruction	See
MLA	Multiply-add, writing to the addend register	MLA
MLS	Multiply-subtract, writing to the addend register	MLS
MSB	Multiply-subtract, writing to the multiplicand register	MSB
MUL	Multiply by immediate	MUL
	Multiply vectors	MUL
NEG	Negate	NEG
SABD	Signed absolute difference	SABD
SDIV	Signed divide	SDIV
SDIVR	Signed reverse divide	SDIVR
SMAX	Signed maximum with immediate	SMAX
	Signed maximum vectors	SMAX
SMIN	Signed minimum with immediate	SMIN
	Signed minimum vectors	SMIN
SMULH	Signed multiply returning high half	SMULH
SQADD	Signed saturating add immediate	SQADD
	Signed saturating add vectors	SQADD
SQSUB	Signed saturating subtract immediate	SQSUB
	Signed saturating subtract vectors	SQSUB
SUB	Subtract immediate	SUB
	Subtract vectors (predicated)	SUB
	Subtract vectors (unpredicated)	SUB
SUBR	Reversed subtract from immediate	SUBR
	Reversed subtract vectors	SUBR
SXTB	Signed byte extend	SXTB
SXTH	Signed halfword extend	SXTH
SXTW	Signed word extend	SXTW
UABD	Unsigned absolute difference	UABD
UDIV	Unsigned divide	UDIV
UDIVR	Unsigned reversed divide	UDIVR
UMAX	Unsigned maximum with immediate	UMAX
	Unsigned maximum vectors	UMAX
UMIN	Unsigned minimum with immediate	UMIN
	Unsigned minimum vectors	UMIN
UMULH	Unsigned multiply returning high half	UMULH
UQADD	Unsigned saturating add immediate	UQADD
	Unsigned saturating add vectors	UQADD
UQSUB	Unsigned saturating subtract immediate	UQSUB
	Unsigned saturating subtract vectors	UQSUB
UXTB	Unsigned byte extend	UXTB
UXTH	Unsigned halfword extend	UXTH
UXTW	Unsigned word extend	UXTW

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 quadtuplets. 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	SDOT
	Signed dot product by indexed elements	SDOT
SUDOT	Signed by unsigned integer dot product by indexed elements[a]	SUDOT
UDOT	Unsigned dot product by vector	UDOT
	Unsigned dot product by indexed elements	UDOT
USDOT	Unsigned by signed integer dot product[a]	USDOT
	Unsigned by signed integer dot product by indexed elements[a]	USDOT

a. This instruction is only supported when ID_AA64ZFR0_EL1.I8MM is set to 1.

Integer matrix multiply operations

These instructions facilitate matrix multiplication and include integer matrix multiply-accumulate instructions.

The matrix multiplication instructions that are supported depend on the values of ID_AA64ZFR0_EL1.I8MM, as shown in Table 5-6.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:

- The first source vector matrix is organized in row-by-row order.
- The second source vector matrix is organized in a column-by-column order.
- The destination vector matrix is organized in row-by-row order.

One matrix multiplication is performed per vector segment. For other matrix multiply instructions, the vector segment length is 128 bits.

Table 5-6 Instructions supported when ID_AA64ZFR0_EL1.I8MM == 1

Mnemonic	Instruction	See
SMMLA	Widening signed 8-bit integer matrix multiply-accumulate into 2x2 matrix	SMMLA
UMMLA	Widening unsigned 8-bit integer matrix multiply-accumulate into 2x2 matrix	UMMLA
USMMLA	Widening mixed sign 8-bit integer matrix multiply-accumulate into 2x2 matrix	USMMLA

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 5-7 Integer comparison instructions

Mnemonic	Instruction	See
CMPEQ	Compare signed equal to immediate	CMPEQ
	Compare signed equal to wide elements	CMPEQ
	Compare signed equal to vector	CMPEQ
CMPGE	Compare signed greater than or equal to immediate	CMPGE
	Compare signed greater than or equal to wide elements	CMPGE
	Compare signed greater than or equal to vector	CMPGE
CMPGT	Compare signed greater than immediate	CMPGT
	Compare signed greater than wide elements	CMPGT
	Compare signed greater than vector	CMPGT
CMPHI	Compare unsigned higher than immediate	CMPHI
	Compare unsigned higher than wide elements	CMPHI
	Compare unsigned higher than vector	CMPHI
CMPHS	Compare unsigned higher than or same as immediate	CMPHS
	Compare unsigned higher than or same as wide elements	CMPHS
	Compare unsigned higher than or same as vector	CMPHS
CMPLE	Compare signed less than or equal to immediate	CMPLE
	Compare signed less than or equal to wide elements	CMPLE
	Compare signed less than or equal to vector	CMPLE
CMPLO	Compare unsigned lower than immediate	CMPLO
	Compare unsigned lower than 64-bit wide elements	CMPLO
	Compare unsigned lower than vector	CMPLO
CMPLS	Compare unsigned lower or same as immediate	CMPLS
	Compare unsigned lower or same as wide elements	CMPLS
	Compare unsigned lower or same as vector	CMPLS
CMPLT	Compare signed less than immediate	CMPLT
	Compare signed less than wide elements	CMPLT
	Compare signed less than vector	CMPLT
CMPNE	Compare not equal to immediate	CMPNE
	Compare not equal to wide elements	CMPNE
	Compare not equal to vector	CMPNE

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 5-8 Vector address calculation instructions

Mnemonic	Instruction	See
ADDVL	Add multiple of vector length, in bytes, to scalar register	ADDVL
ADDPL	Add multiple of predicate register length, in bytes, to scalar register	ADDPL
ADR	Compute vector of addresses	ADR
RDVL	Read multiple of vector register length, in bytes, to scalar register	RDVL

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 5-9 Bitwise logical operations

Mnemonic	Instruction	See
AND	Bitwise AND vectors (predicated)	AND
	Bitwise AND vectors (unpredicated)	AND
	Bitwise AND with immediate	AND
BIC	Bitwise clear with vector (predicated)	BIC
	Bitwise clear with vector (unpredicated)	BIC
	Bitwise clear using immediate	BIC
DUPM	Broadcast bitmask immediate to vector (unpredicated)	DUPM
EON	Bitwise exclusive OR with inverted immediate	EON
EOR	Bitwise exclusive OR vectors (predicated)	EOR
	Bitwise exclusive OR vectors (unpredicated)	EOR
	Bitwise exclusive OR with immediate	EOR
MOV	Move bitmask immediate to vector	MOV
	Move vector register	MOV
NOT	Bitwise invert vector	NOT
ORN	Bitwise OR with inverted immediate	ORN
ORR	Bitwise OR vectors (predicated)	ORR
	Bitwise OR vectors (unpredicated)	ORR
	Bitwise OR with immediate	ORR

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 5-10 Bitwise shift, permute, and count instructions

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

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 5-11 Floating-point arithmetic instructions

Mnemonic	Instruction	See
FABD	Floating-point absolute difference	FABD
FABS	Floating-point absolute value	FABS
FADD	Floating-point add immediate	FADD
	Floating-point add (predicated)	FADD

Table 5-11 Floating-point arithmetic instructions (continued)

Mnemonic	Instruction	See
	Floating-point add (unpredicated)	FADD
FDIV	Floating-point divide	FDIV
FDIVR	Floating-point reversed divide	FDIVR
FMAX	Floating-point maximum with immediate	FMAX
	Floating-point maximum vectors	FMAX
FMAXNM	Floating-point maximum number with immediate	FMAXNM
	Floating-point maximum number vectors	FMAXNM
FMIN	Floating-point minimum with immediate	FMIN
	Floating-point minimum vectors	FMIN
FMINNM	Floating-point minimum number with immediate	FMINNM
	Floating-point minimum number vectors	FMINNM
FMUL	Floating-point multiply by immediate	FMUL
	Floating-point multiply vectors (predicated)	FMUL
	Floating-point multiply vectors (unpredicated)	FMUL
FMULX	Floating-point multiply-extended	FMULX
FNEG	Floating-point negate	FNEG
FRECPE	Floating-point reciprocal estimate	FRECPE
FRECPS	Floating-point reciprocal step	FRECPS
FRECPX	Floating-point reciprocal exponent	FRECPX
FRSQRTE	Floating-point reciprocal square root estimate	FRSQRTE
FRSQRTS	Floating-point reciprocal square root step	FRSQRTS
FSCALE	Floating-point adjust exponent by vector	FSCALE
FSQRT	Floating-point square root	FSQRT
FSUB	Floating-point subtract immediate	FSUB
	Floating-point subtract vectors (predicated)	FSUB
	Floating-point subtract vectors (unpredicated)	FSUB
FSUBR	Floating-point reversed subtract from immediate	FSUBR
	Floating-point reversed subtract vectors	FSUBR

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, FNMAD, FNMSB.

Table 5-12 Floating-point multiply accumulate instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add vectors, writing to the addend	FMLA
FMLS	Floating-point fused multiply-subtract vectors, writing to the addend	FMLS
FNMLA	Floating-point negated fused multiply-add vectors, writing to the addend	FNMLA

Table 5-12 Floating-point multiply accumulate instructions (continued)

Mnemonic	Instruction	See
FNMLS	Floating-point negated fused multiply-subtract vectors, writing to the addend	FNMLS
FMAD	Floating-point fused multiply-add vectors, writing to the multiplicand	FMAD
FMSB	Floating-point fused multiply-subtract vectors, writing to the multiplicand	FMSB
FNMAD	Floating-point negated fused multiply-add vectors, writing to the multiplicand	FNMAD
FNMSB	Floating-point negated fused multiply-subtract vectors, writing to the multiplicand	FNMSB

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 (∞+∞i) multiplied by (0+i) is (NaN+NaNi), rather than the result expected by ISO C, which is complex ∞.

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 5-13 Floating-point complex arithmetic instructions

Mnemonic	Instruction	See
FCADD	Floating-point complex add with rotate	FCADD
FCMLA	Floating-point complex multiply-add with rotate	FCMLA

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 5-14 Floating-point rounding and conversion instructions

Mnemonic	Instruction	See
BFCVT	Floating-point down convert to BFloat16 format	BFCVT
BFCVTNT	Floating-point down convert and narrow to BFloat16 format (top, predicated)	BFCVTNT
FCVT	Floating-point convert precision	FCVT
FCVTZS	Floating-point convert to signed integer, rounding toward zero	FCVTZS
FCVTZU	Floating-point convert to unsigned integer, rounding toward zero	FCVTZU
FRINTA	Floating-point round to integral value, to nearest with ties away from zero	FRINTA
FRINTI	Floating-point round to integral value, using the current rounding mode	FRINTI
FRINTM	Floating-point round to integral value, toward minus infinity	FRINTM
FRINTN	Floating-point round to integral value, to nearest with ties to even	FRINTN
FRINTP	Floating-point round to integral value, toward plus infinity	FRINTP
FRINTX	Floating-point round to integral value exact, using the current rounding mode	FRINTX
FRINTZ	Floating-point round to integral value, toward zero	FRINTZ
SCVTF	Signed integer convert to floating-point	SCVTF
UCVTF	Unsigned integer convert to floating-point	UCVTF

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 5-15 Floating-point comparison instructions

Mnemonic	Instruction	See
FACGE	Floating-point absolute compare greater than or equal	FACGE
FACGT	Floating-point absolute compare greater than	FACGT
FACLE	Floating-point absolute compare less than or equal	FACLE
FACLT	Floating-point absolute compare less than	FACLT
FCMEQ	Floating-point compare equal to zero	FCMEQ
	Floating-point compare equal to vector	FCMEQ
FCMGE	Floating-point compare greater than or equal to zero	FCMGE
	Floating-point compare greater than or equal to vector	FCMGE
FCMGT	Floating-point compare greater than zero	FCMGT

Table 5-15 Floating-point comparison instructions (continued)

Mnemonic	Instruction	See
	Floating-point compare greater than vector	FCMGT
FCMLE	Floating-point compare less than or equal to zero	FCMLE
	Floating-point compare less than or equal to vector	FCMLE
FCMLT	Floating-point compare less than zero	FCMLT
	Floating-point compare less than vector	FCMLT
FCMNE	Floating-point compare not equal to zero	FCMNE
	Floating-point compare not equal to vector	FCMNE
FCMUO	Floating-point unordered vectors	FCMUO

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 5-16 Floating-point transcendental acceleration instructions

Mnemonic	Instruction	See
FTMAD	Floating-point trigonometric multiply-add coefficient	FTMAD
FTSMUL	Floating-point trigonometric starting value	FTSMUL
FTSSEL	Floating-point trigonometric select coefficient	FTSSEL
FEXPA	Floating-point exponential accelerator	FEXPA

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 5-17 Floating-point index multiply instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add by indexed elements	FMLA
FMLS	Floating-point fused multiply-subtract by indexed elements	FMLS
FMUL	Floating-point multiply by indexed elements	FMUL

Floating-point matrix multiply operations

Instructions to facilitate matrix multiplication include floating-point matrix multiply-accumulate instructions, and companion instructions that support data rearrangements in vector registers as required by some of the matrix multiply instructions.

The floating-point matrix multiplication instructions and companion instructions that are supported depend on the values of:

- ID_AA64ZFR0_EL1.F64MM, as shown in Table 5-18 on page 5-56.
- ID_AA64ZFR0_EL1.F32MM, as shown in Table 5-19 on page 5-56.

The matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:

- The first source vector matrix is organized in row-by-row order.
- The second source vector matrix is organized in a column-by-column order.
- The destination vector matrix is organized in row-by-row order.

One matrix multiplication is performed per vector segment. For the double-precision matrix multiply instructions, the vector segment length and minimum vector length is 256 bits. For other matrix multiply instructions, the vector segment length is 128 bits.

The floating-point matrix multiply-accumulate instructions perform all arithmetic with IEEE 754 compliant numerical behaviors and observe the FPCR controls. These are all detailed in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile* in the following sections:

- *Single-precision floating-point format*.
- *Double-precision floating-point format.*
- *Advanced SIMD and floating-point support.*
- *FPCR, Floating-point Control Register*.

For the floating-point matrix multiply-accumulate instructions, the order of accumulations is architecturally defined. The multiplications and additions are not fused, so intermediate rounding is performed after every multiplication and every addition.

Table 5-18 Instructions supported when ID_AA64ZFR0_EL1.F64MM == 1

Mnemonic	Instruction	See
FMMLA	Floating-point matrix multiply-accumulate into 2x2 matrix (double-precision)	FMMLA
LD1ROB	Contiguous load and replicate thirty-two bytes, scalar plus scalar	LD1ROB
	Contiguous load and replicate thirty-two bytes, scalar plus immediate	LD1ROB
LD1ROD	Contiguous load and replicate four doublewords, scalar plus scalar	LD1ROD
	Contiguous load and replicate four doublewords, scalar plus immediate	LD1ROD
LD1ROH	Contiguous load and replicate sixteen halfwords, scalar plus scalar	LD1ROH
	Contiguous load and replicate sixteen halfwords, scalar plus immediate	LD1ROH
LD1ROW	Contiguous load and replicate eight words, scalar plus scalar	LD1ROW
	Contiguous load and replicate eight words, scalar plus immediate	LD1ROW
TRN1, TRN2	Interleave even or odd 128-bit elements from two vectors	TRN1, TRN2
UZP1, UZP2	Concatenate even or odd 128-bit elements from two vectors	UZP1, UZP2
ZIP1, ZIP2	Interleave 128-bit elements from two half vectors	ZIP1, ZIP2

Table 5-19 Instructions supported when ID_AA64ZFR0_EL1.F32MM == 1

Mnemonic	Instruction	See
FMMLA	Floating-point matrix multiply-accumulate into 2x2 matrix (single-precision)	FMMLA

BFloat16 floating-point support

BFloat16, or BF16, is a 16-bit floating-point storage format defined by the Arm architecture such that it inherits many of its properties and behaviors from the IEEE 754 single-precision format. For more information, see the section titled *BFloat16 floating-point format* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.

BFloat16 floating-point instructions

All of these instructions perform an implicit conversion of vectors of BF16 input values to IEEE 754 single-precision floating-point format. In addition, the BFDOT and BFMMLA instructions perform an N-way dot-product calculation that accumulates the products into a vector of single-precision accumulators.

All of these instructions perform arithmetic with fixed numeric behaviors. For more information, see the section titled *BFloat16 floating-point format* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.

Table 5-20 BFloat16 floating-point instructions

Mnemonic	Instruction	See
BFDOT	BFloat16 floating-point dot product by vector	BFDOT
	BFloat16 floating-point dot product by indexed elements	BFDOT
BFMMLA	BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix	BFMMLA
BFMLALB	BFloat16 floating-point widening multiply accumulate long bottom by vector	BFMLALB
	BFloat16 floating-point widening multiply accumulate long bottom by indexed elements	BFMLALB
BFMLALT	BFloat16 floating-point widening multiply accumulate long top by vector	BFMLALT
	BFloat16 floating-point widening multiply accumulate long bottom by indexed elements	BFMLALT

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 5-21 Predicate initialization instructions

Mnemonic	Instruction	See
PFALSE	Set all predicate elements to FALSE	PFALSE
PTRUE	Initialize predicate elements from named constraint	PTRUE
PTRUES	Initialize predicate elements from named constraint, setting the condition flags	PTRUES

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 5-22 Predicate move instructions

Mnemonic	Instruction	See
SEL	Select predicate elements from two predicates	SEL
MOV	Move predicate elements (predicated, merging)	MOV
	Move predicate elements (predicated, zeroing)	MOV
	Move predicate elements (unpredicated)	MOV
MOVS	Move predicate elements, setting the condition flags (predicated)	MOVS
	Move predicate elements, setting the condition flags (unpredicated)	MOVS

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 5-23 Predicate logical operation instructions

Mnemonic	Instruction	See
AND	Bitwise AND predicates	AND
ANDS	Bitwise AND predicates, setting the condition flags	ANDS
BIC	Bitwise clear predicates	BIC
BICS	Bitwise clear predicates, setting the condition flags	BICS
EOR	Bitwise exclusive OR predicates	EOR
EORS	Bitwise exclusive OR predicates, setting the condition flags	EORS
NAND	Bitwise NAND predicates	NAND
NANDS	Bitwise NAND predicates, setting the condition flags	NANDS
NOR	Bitwise NOR predicates	NOR
NORS	Bitwise NOR predicates, setting the condition flags	NORS
NOT	Bitwise invert predicate	NOT
NOTS	Bitwise invert predicate, setting the condition flags	NOTS
ORN	Bitwise OR inverted predicate	ORN
ORNS	Bitwise OR inverted predicate, setting the condition flags	ORNS
ORR	Bitwise OR predicates	ORR
ORRS	Bitwise OR predicates, setting the condition flags	ORRS
PTEST	Test predicate value, setting the condition flags	PTEST

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 RDFFRS 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 RDFFR and RDFFRS instructions must be in canonical form.

Table 5-24 FFR predicate handling instructions

Mnemonic	Instruction	See
RDFFR	Return predicate of successfully loaded elements (unpredicated)	RDFFR
	Return predicate of successfully loaded elements (predicated)	RDFFR
RDFFRS	Return predicate of successfully loaded elements, setting the condition flags (predicated)	RDFFRS
SETFFR	Initialize the First-fault register to all TRUE	SETFFR
WRFFR	Write a predicate register to the First-fault register	WRFFR

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 5-25 Predicate count instructions

Mnemonic	Instruction	See
CNTB	Set scalar to multiple of 8-bit predicate constraint element count	CNTB
CNTH	Set scalar to multiple of 16-bit predicate constraint element count	CNTH
CNTW	Set scalar to multiple of 32-bit predicate constraint element count	CNTW
CNTD	Set scalar to multiple of 64-bit predicate constraint element count	CNTD
CNTP	Set scalar to the number of Active predicate elements that are TRUE	CNTP
DECB	Decrement scalar by multiple of 8-bit predicate constraint element count	DECB
DECH	Decrement scalar by multiple of 16-bit predicate constraint element count	DECH
	Decrement vector by multiple of 16-bit predicate constraint element count	DECH
DECW	Decrement scalar by multiple of 32-bit predicate constraint element count	DECW
	Decrement vector by multiple of 32-bit predicate constraint element count	DECW
DECD	Decrement scalar by multiple of 64-bit predicate constraint element count	DECD
	Decrement vector by multiple of 64-bit predicate constraint element count	DECD
DECP	Decrement scalar by the number of predicate elements that are TRUE	DECP

Table 5-25 Predicate count instructions (continued)

Mnemonic	Instruction	See
	Decrement vector by the number of Active predicate elements that are TRUE	DECP
INCB	Increment scalar by multiple of 8-bit predicate constraint element count	INCB
INCH	Increment scalar by multiple of 16-bit predicate constraint element count	INCH
	Increment vector by multiple of 16-bit predicate constraint element count	INCH
INCW	Increment scalar by multiple of 32-bit predicate constraint element count	INCW
	Increment vector by multiple of 32-bit predicate constraint element count	INCW
INCD	Increment scalar by multiple of 64-bit predicate constraint element count	INCD
	Increment vector by multiple of 64-bit predicate constraint element count	INCD
INCP	Increment scalar by the number of predicate elements that are TRUE	INCP
	Increment vector by the number of predicate elements that are TRUE	INCP
SQDECB	Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count	SQDECB
SQDECH	Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count	SQDECH
	Signed saturating decrement vector by multiple of 16-bit predicate constraint element count	SQDECH
SQDECW	Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.	SQDECW
	Signed saturating decrement vector by multiple of 32-bit predicate constraint element count	SQDECW
SQDECD	Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count	SQDECD
	Signed saturating decrement vector by multiple of 64-bit predicate constraint element count	SQDECD
SQDECP	Signed saturating decrement scalar the number of predicate elements that are TRUE	SQDECP
	Signed saturating decrement vector by the number of predicate elements that are TRUE	SQDECP
SQINCB	Signed saturating increment scalar by multiple of 8-bit predicate constraint element count	SQINCB
SQINCH	Signed saturating increment scalar by multiple of 16-bit predicate constraint element count	SQINCH
	Signed saturating increment vector by multiple of 16-bit predicate constraint element count	SQINCH
SQINCW	Signed saturating increment scalar by multiple of 32-bit predicate constraint element count	SQINCW
	Signed saturating increment vector by multiple of 32-bit predicate constraint element count	SQINCW
SQINCD	Signed saturating increment scalar by multiple of 64-bit predicate constraint element count	SQINCD
	Signed saturating increment vector by multiple of 64-bit predicate constraint element count	SQINCD
SQINCP	Signed saturating increment scalar by the number of predicate elements that are TRUE	SQINCP
	Signed saturating increment vector by the number of predicate elements that are TRUE	SQINCP

Table 5-25 Predicate count instructions (continued)

Mnemonic	Instruction	See
UQDECB	Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count	UQDECB
UQDECH	Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count	UQDECH
	Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count	UQDECH
UQDECW	Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count	UQDECW
	Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count	UQDECW
UQDECD	Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count	UQDECD
	Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count	UQDECD
UQDECP	Unsigned saturating decrement scalar by the number of predicate elements that are TRUE	UQDECP
	Unsigned saturating decrement vector by the number of predicate elements that are TRUE	UQDECP
UQINCB	Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count	UQINCB
UQINCH	Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count	UQINCH
	Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count	UQINCH
UQINCW	Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count	UQINCW
	Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count	UQINCW
UQINCD	Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count	UQINCD
	Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count	UQINCD
UQINCP	Unsigned saturating increment scalar by the number of predicate elements that are TRUE	UQINCP
	Unsigned saturating increment vector by the number of predicate elements that are TRUE	UQINCP

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 5-26 Simple counted loop instructions

Mnemonic	Instruction	See
WHILELE	While incrementing signed scalar less than or equal to scalar	WHILELE
WHILELO	While incrementing unsigned scalar lower than scalar	WHILELO
WHILELS	While incrementing unsigned scalar lower than or the same as scalar	WHILELS
WHILELT	While incrementing signed scalar less than scalar	WHILELT

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 5-27 Data-dependent loop instructions

Mnemonic	Instruction	See
BRKA	Break after the first true condition	BRKA
BRKAS	Break after the first true condition, setting the condition flags	BRKAS
BRKB	Break before the first true condition	BRKB
BRKBS	Break before the first true condition, setting the condition flags	BRKBS
BRKN	Propagate break to next partition	BRKN
BRKNS	Propagate break to next partition, setting the condition flags	BRKNS
BRKPA	Break after the first true condition, propagating from previous partition	BRKPA
BRKPAS	Break after the first true condition, propagating from previous partition, setting the condition flags	BRKPAS
BRKPB	Break before the first true condition, propagating from the previous partition	BRKPB
BRKPBS	Break before the first true condition, propagating from the previous partition, setting the condition flags	BRKPBS

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 5-28 Serialized operation instructions

Mnemonic	Instruction	See
PFIRST	Set the First active element to TRUE	PFIRST
PNEXT	Find next Active element	PNEXT
CTERMEQ	Compare and terminate loop when equal	CTERMEQ
CTERMNE	Compare and terminate loop when not equal	CTERMNE

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 5-29 Element permute and shuffle instructions

Mnemonic	Instruction	See
CLASTA	Conditionally extract element after the Last active element to general-purpose register	CLASTA
	Conditionally extract element after the Last active element to SIMD&FP scalar	CLASTA
	Conditionally extract element after the Last active element to vector	CLASTA
CLASTB	Conditionally extract Last active element to general-purpose register	CLASTB
	Conditionally extract Last active element to SIMD&FP scalar	CLASTB
	Conditionally extract Last active element to vector	CLASTB
LASTA	Extract element after the Last active element to general-purpose register	LASTA
	Extract element after the Last active element to SIMD&FP scalar	LASTA
LASTB	Extract Last active element to general-purpose register	LASTB
	Extract Last active element to SIMD&FP scalar	LASTB
COMPACT	Shuffle Active elements of vector to the right and fill with zeros	COMPACT
SPLICE	Splice two vectors under predicate control	SPLICE
TBL	Programmable table lookup using vector of element indexes	TBL
DUP	Broadcast indexed vector element	DUP

Table 5-29 Element permute and shuffle instructions (continued)

Mnemonic	Instruction	See
EXT	Extract vector from pair of vectors	EXT
INSR	Insert general-purpose register into shifted vector	INSR
	Insert SIMD&FP scalar register into shifted vector	INSR
MOV	Move indexed element or SIMD&FP scalar to vector (unpredicated)	MOV
	Move SIMD&FP scalar register to vector elements (predicated)	MOV
REV	Reverse all elements in vector	REV
REVB	Reverse 8-bit bytes in elements	REVB
REVH	Reverse 16-bit halfwords in elements	REVH
REVW	Reverse 32-bit words in elements	REVW
TRN1	Interleave even elements from two vectors	TRN1
TRN2	Interleave odd elements from two vectors	TRN2
UZP1	Concatenate even elements from two vectors	UZP1
UZP2	Concatenate odd elements from two vectors	UZP2
ZIP1	Interleave elements from low halves of two vectors	ZIP1
ZIP2	Interleave elements from high halves of two vectors	ZIP2

Unpacking instructions

These instructions unpack half of the elements from the source vector register or predicate register, widen the unpacked elements to twice the width, and place the result in the destination register.

Table 5-30 Unpacking instructions

Mnemonic	Instruction	See
SUNPKHI	Unpack and sign-extend elements from high half of vector	SUNPKHI
SUNPKLO	Unpack and sign-extend elements from low half of vector	SUNPKLO
UUNPKHI	Unpack and zero-extend elements from high half of vector	UUNPKHI
UUNPKLO	Unpack and zero-extend elements from low half of vector	UUNPKLO
PUNPKHI	Unpack and widen elements from high half of predicate	PUNPKHI
PUNPKLO	Unpack and widen elements from low half of predicate	PUNPKLO

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 5-31 Predicate permute instructions

Mnemonic	Instruction	See
REV	Reverse all elements in predicate	REV
TRN1	Interleave even elements from two predicates	TRN1
TRN2	Interleave odd elements from two predicates	TRN2
UZP1	Select even elements from two predicates	UZP1

Table 5-31 Predicate permute instructions (continued)

Mnemonic	Instruction	See
UZP2	Select odd elements from two predicates	UZP2
ZIP1	Interleave elements from low halves of two predicates	ZIP1
ZIP2	Interleave elements from high halves of two predicates	ZIP2

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 5-32 Index vector generation instructions

Mnemonic	Instruction	See
INDEX	Create index vector starting from and incremented by immediates	INDEX
	Create index vector starting from immediate and incremented by general-purpose register	INDEX
	Create index vector starting from general-purpose register and incremented by immediate	INDEX
	Create index vector starting from and incremented by general-purpose registers	INDEX

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 identified as supporting MOVPRFX in its *Operational notes* section, or an A64 HLT instruction, or an A64 BRK instruction. For a prefixed SVE instruction all of the following apply:

- 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 5-33 Move prefix instructions

Mnemonic	Instruction	See
MOVPRFX	Move prefix (predicated)	MOVPRFX
	Move prefix (unpredicated)	MOVPRFX

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 5-34 Horizontal reduction instructions

Mnemonic	Instruction	See
ANDV	Bitwise AND reduction, treating Inactive elements as all ones	ANDV
EORV	Bitwise XOR reduction, treating Inactive elements as zero	EORV
FADDA	Floating-point add strictly-ordered reduction, accumulating in scalar, ignoring Inactive elements	FADDA
FADDV	Floating-point add recursive reduction, treating Inactive elements as +0.0	FADDV
FMAXNMV	Floating-point maximum number recursive reduction, treating Inactive elements as the default NaN	FMAXNMV
FMAXV	Floating-point maximum recursive reduction, treating Inactive elements as negative infinity	FMAXV
FMINNMV	Floating-point minimum number recursive reduction, treating Inactive elements as the default NaN	FMINNMV
FMINV	Floating-point minimum recursive reduction, treating Inactive elements as positive infinity	FMINV
ORV	Bitwise OR reduction, treating Inactive elements as zero	ORV
SADDV	Signed add reduction, treating Inactive elements as zero	SADDV
SMAXV	Signed maximum reduction, treating Inactive elements as the minimum signed integer	SMAXV
SMINV	Signed minimum reduction, treating Inactive elements the maximum signed integer	SMINV
UADDV	Unsigned add reduction, treating Inactive elements as zero	UADDV
UMAXV	Unsigned maximum reduction, treating Inactive elements as zero	UMAXV
UMINV	Unsigned minimum reduction, treating Inactive elements as the maximum unsigned integer	UMINV

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:

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
ID_AA64PFR0_EL1	Defines bits[35:32] as the SVE field.	The SVE field indicates whether SVE is implemented.
CPACR_EL1	Defines bits[17:16] as the ZEN field.	The ZEN field enables access to SVE functionality from EL1 and EL0.
CPTR_EL2	When HCR_EL2.E2H == 0, defines bit[8] as TZ.	The TZ field traps access to SVE functionality from EL2 and Non-secure EL1&0 to EL2.
	When HCR_EL2.E2H == 1, defines bits[17:16] as the ZEN field.	The ZEN field enables access to SVE functionality from EL2 and Non-secure EL1&0.
CPTR_EL3	Defines bit[8] as EZ.	The EZ field enables access to SVE functionality from EL0, EL1, EL2, and EL3.
TCR_EL1	Defines bit[54] as NFD1 and bit[53] as NFD0.	The NFD1 and NFD0 fields disable stage 1 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.
TCR_EL2	When HCR_EL2.E2H == 1, defines bit[54] as NFD1 and bit[53] as NFD0.	The NFD1 and NFD0 fields disable stage 1 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.
EDPFR	Defines bits[35:32] as the SVE field.	The SVE field indicates whether SVE is implemented for debug.
ESR_ELx	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 CPACR_EL1, CPTR_EL2, or CPTR_EL3.

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
ID_AA64ZFR0_EL1	SVE feature ID register 0
ZCR_EL1	SVE control register to constrain the vector length at EL1 and EL0
ZCR_EL2	SVE control register to constrain the vector length at EL2 and Non-secure EL1 and EL0
ZCR_EL3	SVE control register to constrain the vector length at EL3, EL2, EL1, and EL0

Chapter 7
SVE Debug

This chapter introduces the additions to Armv8-A AArch64 debug due to SVE. This chapter contains the following sections:

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:

- Accesses performed as a result of an Active element by an SVE predicated vector store instruction.
- Accesses performed as a result of an Active element by an SVE predicated vector load instruction that is not a First-fault or Non-fault load.
- Accesses performed as a result of the First active element by an SVE First-fault vector load 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 elements.

SVE Non-fault vector loads do not generate watchpoint debug events, but any Active element access that matches a configured watchpoint is reported in the FFR as described in *Synchronous memory faults* on page 3-28.

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 reported in the FFR as described in *Synchronous memory faults* on page 3-28.

A watchpoint is not a mechanism for preventing access to memory. An SVE Non-fault or First-fault vector load that does not trigger a watchpoint can return the data from an access that matches a configured watchpoint, while setting the corresponding FFR element to FALSE.

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_EL0 and DLR_EL0 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)
- WRFFR (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_EL0 and DSPSR_EL0 to UNKNOWN values.
- If the instruction reads PSTATE condition flags, it uses an UNKNOWN value for the condition flag.
- 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:

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 *Recommended PMU events* on page A-80. 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-83, with the following exceptions:

- It is IMPLEMENTATION DEFINED whether operations due to any of the instructions listed in *Data movement instructions* on page A-83, *Floating-point conversions* on page A-88, or *Floating-point or integer instructions* on page A-89, 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-83 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-91 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 (μ–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 μ–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	SVE_INST_RETIRED
0x8006	Microarchitectural, Required	SVE_INST_SPEC

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-89.

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-89.

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 MOVPRFX 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 MOVPRFX 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:

A.1 Recommended PMU events

The section titled *PMU events and event numbers* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile* describes the recommended architectural and microarchitectural PMU events for SVE implementations.

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 × VL ÷ 128 + FP_FIXED_OPS_SPEC

A summary of these event pairs is given in Table A-1. Note that combined multiply-add and multiply-subtract instructions are counted as two operations per element.

Table A-1 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 × VL ÷ 128 + LD_FIXED_BYTES_SPEC

Table A-2 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-3 and the result used to adjust SVE event counters that ignore the predicate weight.

Table A-3 Vector utilization ratios

Utilization rate	Ratio
All predicates active	SVE_PRED_FULL_SPEC ÷ SVE_PRED_SPEC
Partial predicates active	SVE_PRED_PARTIAL_SPEC ÷ SVE_PRED_SPEC
No predicates active	SVE_PRED_EMPTY_SPEC ÷ 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-4.

Table A-4 Vector loop efficiency ratios

Vector loop metric	Ratio
Source level iterations per loop	SVE_PLOOP_ELTS_SPEC ÷ SVE_PLOOP_TERM_SPEC
Vectorized iterations per loop	SVE_PLOOP_TEST_SPEC ÷ SVE_PLOOP_TERM_SPEC
Parallelism per vector loop	SVE_PLOOP_ELTS_SPEC ÷ 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
- SMLSL, SMLSL2, UMLSL, UMLSL2
- 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
- SQSHRN, SQSHRN2, SQSHRUN, SQSHRUN2, UQRSHRN, UQRSHRN2
- 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, SQSHL, SQSHLU, UQRSHL, UQSHL
- SSHL, 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, CMPLE, 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)
- `FRECPS, 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, FNMAD, FNMSB, FMSB
- FMLA, FMLS, FNMLA, FNMLS
- FMUL, FMULX
- FRECPS, FRSQRTS
- 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, FTSSEL

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)

- `CTERMEQ, CTERMNE`

A.3.7 Predicate instructions

Predicate move (SVE)

- `PFALSE, PTRUE, PTRUES`
- `PUNPKHI, PUNPKLO`
- `RDFFR, RDFFRS` (predicated)
- `RDFFR, 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, LDNF1SH`
- `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)

Glossary

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.

Effective value

A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an Effective value:

- In some cases, the description of a control *a* specifies that when control *a* is active it causes a register control field *b* to be treated as having a fixed value for all purposes other than direct reads, or direct reads and direct writes, of the register containing control field *b*. When control *a* is active that fixed value is described as the Effective value of register control field *b*. For example, when the value of HCR.DC is 1, the Effective value of HCR.VM is 1, regardless of its actual value.

  In other cases, in some contexts a register control field *b* is not implemented or is not accessible, but behavior of the PE is as if control field *b* was implemented and accessible, and had a particular value. In this case, that value is the Effective value of register control field *b*.

  Note

  Where a register control field is introduced in a particular version of the architecture, and is not implemented in an earlier version of the architecture, typically it will have an Effective value in that earlier version of the architecture.

- Otherwise, the Effective value of a register control field is the value of that field.

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. See Figure 2-1 on page 2-20 for more information.

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 5-41 for more information.

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

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.

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

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 5-41 for more information.

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

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.