ARM® Architecture Reference Manual Supplement

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

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

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

Release Information

The following changes have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 March 2017</td>
<td>A.a</td>
<td>Non-Confidential Beta</td>
<td>Beta release</td>
</tr>
</tbody>
</table>

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

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

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

LES-PRE-20327

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

Confidentiality Status

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

The information in this document is for a Beta product, that is a product under development.

Web Address

http://www.arm.com

Limitations of A.a

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

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

Preface
About this book ........................................................................................................ viii
Using this book ........................................................................................................ ix
Conventions ............................................................................................................... x
Additional reading ................................................................................................... xi
Feedback .................................................................................................................. xii

Chapter 1  Introduction
1.1 About the SVE supplement ............................................................................ 1-14
1.2 About the Scalable Vector Extension .......................................................... 1-15
1.3 Terminology .................................................................................................... 1-16

Chapter 2  SVE Programmers' Model
2.1 Registers ......................................................................................................... 2-20
2.2 Process state, PSTATE .................................................................................. 2-24
2.3 SVE half-precision floating-point .................................................................. 2-26
2.4 Exception model ............................................................................................. 2-27

Chapter 3  SVE Memory Model
3.1 SVE Memory Model ..................................................................................... 3-30

Chapter 4  SVE Instruction Set
4.1 SVE assembler language ................................................................................. 4-34
4.2 Instruction set overview ................................................................................ 4-35
Preface

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

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

This book describes 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), version B.a, 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 architecture.

The information in this manual is organized into the following chapters:

**Chapter 1 Introduction**
Read this for an overview of the SVE extension. It outlines the key features of SVE and introduces the terminology used to describe the extension.

**Chapter 2 SVE Programmers’ Model**
Read this for a description of the SVE Programmers’ Model. This section must be read in conjunction with the corresponding sections in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile.

**Chapter 3 SVE Memory Model**
Read this for a description of the SVE Memory Model. This section must be read in conjunction with the corresponding sections in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile.

**Chapter 4 SVE Instruction Set**
Read this for a description of the SVE Instruction Set Architecture. This section must be read in conjunction with the corresponding sections in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile.

**Chapter 5 SVE System Registers**
Read this for a description of the new SVE System registers and the existing AArch64 System registers that are modified by SVE. This section must be read in conjunction with the corresponding sections in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile.

**Chapter 6 SVE Debug**
Read this for a description of the SVE the additions to the ARMv8-A AArch64 Debug Architecture. This section must be read in conjunction with the corresponding sections in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile.

**Chapter 7 SVE Performance Monitors Extension**
Read this for a description of the SVE the additions to the ARMv8-A AArch64 Performance Monitors Extension. This section must be read in conjunction with the corresponding sections in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile.

**Appendix A Flag-setting instructions**
Read this for a description of the SVE condition flag-setting instructions.

**Appendix B Predicate-setting instructions**
Read this for a description of the SVE predicate-setting instructions.
Conventions

The following sections describe conventions that this book can use:

• Typographical conventions.
• Numbers.

Typographical conventions

The following table describes the typographical conventions:

<table>
<thead>
<tr>
<th>Style</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>italic</td>
<td>Introduces special terminology, and denotes citations.</td>
</tr>
<tr>
<td>bold</td>
<td>Denotes signal names, and is used for terms in descriptive lists, where appropriate.</td>
</tr>
</tbody>
</table>
| 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 the Infocenter http://developer.arm.com, for access to ARM documentation.

ARM publications

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.a.
• The page numbers to which your comments apply.
• A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.
Chapter 1

Introduction

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

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

This supplement must be read with the most recent issue of 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.

*About the Scalable Vector Extension on page 1-15* introduces 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 instruction descriptions 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 extension to the ARMv8-A architecture profile. SVE is defined for the AArch64 Execution state only, and adds:

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

The key features that SVE provides are:

- Scalable vector length. See Vector registers on page 2-20.
- Predication. See Predicate registers on page 2-21.
- Gather-load and scatter-store. See Load, store, and prefetch instructions on page 4-36.
- Fault-tolerant speculative vectorization. See First Fault Register, FFR on page 2-23.

Note

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

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

Active element
An active element is a vector 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 or predicate register elements are implicitly treated as active.

Canonical predicate register value
A canonical predicate register value contains 0 in all bits except for the lowest-numbered bit of each predicate element.

Constructive instruction
A constructive instruction is an instruction where the destination register is defined independently of the source registers. Therefore, the original contents of the source registers are not modified by the execution of the instruction.

Destructive instruction
A destructive instruction is an instruction where one of the source registers also acts as the destination register. Therefore, the contents of the source register, when the instruction begins execution, are replaced by the result of the instruction when the instruction completes execution.

Element number
Elements within a vector or predicate register are numbered with element[0] always representing the lowest-numbered bits of the vector register. See Figure 2-1 on page 2-20 for more information.

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

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 or offsets where the addresses are constructed according to the addressing mode. See Load, store, and prefetch instructions on page 4-36 for more information.

Governing predicate
A governing predicate defines the active and inactive elements of the source vector or predicate registers and destination vector or predicate register.

Inactive element
An inactive element is a vector or predicate register element that has been identified, by the value of the corresponding element of an instruction’s governing predicate being FALSE, as an unused source register element or destination register element for the associated instruction.

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

Merging predication
When a predicated instruction specifies merging predication, the inactive elements of the result vector remain unchanged.

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

Predicate element
An SVE predicate register can be subdivided into 1-bit, 2-bit, 4-bit, or 8-bit elements which are one eighth of the specified vector element size. Only the lowest-numbered bit of each predicate element provides the Boolean value of the element, where 1 represents TRUE and 0 represents FALSE. Other bits in the predicate element are IGNORED on reads and set to zero on writes.

**Predicate register**
A one-dimensional array of predicate elements. A predicate register contains one bit for every eight bits in a vector register and therefore has a width that is a multiple of 16 bits, in the range 16-256 bits, inclusive.

**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 4-36 for more information.

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

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

**Vector element**
An SVE vector register can be subdivided into 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements. The SVE vector element size is specified independently by each instruction.

**Vector length (VL)**
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. Unless otherwise stated, the vector length is a multiple of 128 bits in the range of 128-2048 bits, inclusive. All vector registers have the same vector length.

**Vector register**
A vector register is a one-dimensional array of vector elements.

**Zeroing predication**
When a predicated instruction specifies zeroing predication, the inactive elements of the result vector are set to zero.
1 Introduction
1.3 Terminology
Chapter 2
SVE Programmers’ Model

This chapter introduces the SVE Programmers’ model. This chapter contains the following sections:

• *Registers* on page 2-20.
• *Process state, PSTATE* on page 2-24.
• *SVE half-precision floating-point* on page 2-26.
• *Exception model* on page 2-27.
2.1 Registers

2.1.1 Vector registers

SVE introduces 32 scalable vector registers, Z0-Z31. In any implementation, the vector registers are all the same size. The size of every vector register 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 width for a given instruction is encoded in the opcode of the instruction.

Bits [127:0] of the SVE vector registers, Z0-Z31, are shared with the AArch64 SIMD&FP registers, V0-V31, so that Vn is equal to Zn[127:0], as shown in Figure 2-1. If VL is greater than 128 bits, then any AArch64 scalar Advanced SIMD and floating-point instruction that writes to V0-V31 will zero bits[VL - 1:128] of the corresponding SVE vector register that are accessible at the current Exception level. See Configurable vector lengths on page 2-21 for more information.

![Figure 2-1 SVE and Advanced SIMD vectors in AArch64 state](image-url)
Configurable vector lengths

Privileged Exception levels can use the ZCR_EL1.LEN, ZCR_EL2.LEN, and ZCR_EL3.LEN System register fields to constrain the vector length that is accessible by less privileged Exception levels. See New AArch64 System registers on page 5-48 for more information about the ZCR_ELx registers. SVE requires that an implementation must allow the vector length to be constrained to any power of two that is less than the maximum 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. See Table 2-1 for more information.

If an unsupported vector length is requested in ZCR_ELx, the implementation is required to round the request down to the largest supported vector length that is less than the requested length.

Table 2-1 Configurable vector lengths

<table>
<thead>
<tr>
<th>Maximum vector length</th>
<th>Required</th>
<th>Permitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>-</td>
</tr>
<tr>
<td>384</td>
<td>128, 256</td>
<td>-</td>
</tr>
<tr>
<td>512</td>
<td>128, 256</td>
<td>384</td>
</tr>
<tr>
<td>640</td>
<td>128, 256, 512</td>
<td>384</td>
</tr>
<tr>
<td>768</td>
<td>128, 256, 512</td>
<td>384, 640</td>
</tr>
<tr>
<td>896</td>
<td>128, 256, 512</td>
<td>384, 640, 768</td>
</tr>
<tr>
<td>1024</td>
<td>128, 256, 512</td>
<td>384, 640, 768, 896</td>
</tr>
<tr>
<td>1152</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896</td>
</tr>
<tr>
<td>1280</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152</td>
</tr>
<tr>
<td>1408</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280</td>
</tr>
<tr>
<td>1536</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280, 1408</td>
</tr>
<tr>
<td>1664</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280, 1408, 1536</td>
</tr>
<tr>
<td>1792</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664</td>
</tr>
<tr>
<td>1920</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664, 1792</td>
</tr>
<tr>
<td>2048</td>
<td>128, 256, 512, 1024</td>
<td>384, 640, 768, 896, 1152, 1280, 1408, 1536, 1664, 1792, 1920</td>
</tr>
</tbody>
</table>

2.1.2 Predicate registers

SVE introduces 16 scalable predicate registers, P0-P15. Each predicate register holds one bit per byte of a vector register, meaning 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 number corresponds to the equivalent vector element number. If the lowest-numbered bit of a predicate element has a value of 1, the predicate element is TRUE, otherwise it is FALSE. All 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.

A predicate register can be set to a fixed pattern using the PTRUE and PFALSE instructions. Predicate elements can also be set as the result of a predicated instruction, such as a vector compare, or a bitwise logical predicate operation. See Predicate-setting instructions for a full list of the predicate-setting instructions.

Governing predicate

Where an instruction supports predication, it is known as a predicated instruction. The predicate register that is used to determine the active elements of a predicated instruction is known as the governing predicate for that instruction. Predicated instructions can only use P0-P7 as the governing predicate. All of P0-P15 can be accessed by predicate generation and predicate manipulation instructions.
When a governing predicate element is TRUE, then the corresponding vector or predicate register element is active and is processed by the instruction, otherwise it is inactive and takes no part in the operation of the instruction.

When a predicated instruction writes to a vector or predicate result register, either:
• The inactive elements in the destination vector are set to zero. This is known as zeroing predication.
• The inactive elements in the destination vector retain their previous value. This is known as merging predication. See Predication support on page 4-45 for more information about 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 register element is 1, indicating that all elements of the vector register are active.

Figure 2-4 on page 2-23 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 on page 2-23 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.
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 first-fault and non-fault vector load instructions. The format of the FFR is the same as the predicate registers.

The FFR can be initialized to all ones by the `SETFFR` instruction and can be copied to a numbered predicate register by the `RDFFR` and `RDFFRS` instructions. The `WRFFR` instruction provides a mechanism to restore a saved FFR value.

2.1.4 Scalar registers

Certain SVE instructions generate a scalar result that is written to an AArch64 general-purpose register or to the lowest element of a vector register. If an SVE instruction generates a scalar result that is narrower than the maximum 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 by an instruction that generates a predicate result. See Flag-setting instructions on page A-60 for a full list of flag-setting instructions. In either case, a governing predicate determines which predicate elements are to be tested. The SVE interpretation of the predicate condition flags is shown in Table 2-2.

### Table 2-2 SVE condition flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>SVE Name</th>
<th>SVE Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>First</td>
<td>Set to 1 if the first active predicate element is TRUE.</td>
</tr>
<tr>
<td>Z</td>
<td>None</td>
<td>Set to 1 if no active predicate element is TRUE.</td>
</tr>
<tr>
<td>C</td>
<td>Not last</td>
<td>Cleared to 0 if the last active predicate element is TRUE.</td>
</tr>
<tr>
<td>V</td>
<td>-</td>
<td>Cleared to 0 by the SVE flag-setting instructions, except CTERMNE and CTERMEQ.</td>
</tr>
</tbody>
</table>

SVE assembler syntax defines a new set of condition code aliases. The condition code aliases and their associated meanings are described in Table 2-3.

### Table 2-3 Predicate condition flags

<table>
<thead>
<tr>
<th>Condition test</th>
<th>AArch64 name</th>
<th>SVE alias</th>
<th>SVE Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z==1</td>
<td>EQ</td>
<td>NONE</td>
<td>No active elements are TRUE</td>
</tr>
<tr>
<td>Z==0</td>
<td>NE</td>
<td>ANY</td>
<td>An active element is TRUE</td>
</tr>
<tr>
<td>C==1</td>
<td>HS/CS</td>
<td>NLAST</td>
<td>The last active element is not TRUE</td>
</tr>
<tr>
<td>C==0</td>
<td>LO/CC</td>
<td>LAST</td>
<td>The last active element is TRUE</td>
</tr>
<tr>
<td>N==1</td>
<td>MI</td>
<td>FIRST</td>
<td>The first active element is TRUE</td>
</tr>
<tr>
<td>N==0</td>
<td>PL</td>
<td>NFRST</td>
<td>The first active element is not TRUE</td>
</tr>
<tr>
<td>V==1</td>
<td>VS</td>
<td>-</td>
<td>Used by CTERM, otherwise always FALSE</td>
</tr>
<tr>
<td>V==0</td>
<td>VC</td>
<td>-</td>
<td>Used by CTERM, otherwise always TRUE</td>
</tr>
<tr>
<td>C==1 &amp;&amp; Z==0</td>
<td>H1</td>
<td>PMORE</td>
<td>An active element is TRUE but not the last element</td>
</tr>
<tr>
<td>C==0</td>
<td></td>
<td>Z==1</td>
<td>LS</td>
</tr>
<tr>
<td>N==V</td>
<td>GE</td>
<td>TCONT</td>
<td>CTERM termination condition not detected</td>
</tr>
<tr>
<td>N!=V</td>
<td>LT</td>
<td>TSTOP</td>
<td>CTERM termination condition detected</td>
</tr>
</tbody>
</table>

---

### Note

Most SVE flag-setting instructions are themselves predicated. See Predicate-setting instructions on page B-62 for more information. In these cases, it is the governing predicate for the instruction that determines the active and inactive elements in the predicate source and destination registers. Any unpredicated flag-setting instructions have an implicit governing predicate, with all elements set to TRUE, meaning 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 16-bit predicate register containing eight 2-bit predicate elements, that corresponds to a 128-bit vector register with eight 16-bit vector elements. In this case, element[1] of Pd is the first active element, however, its value is FALSE.
2.3 SVE half-precision floating-point

SVE half-precision floating-point instructions support only IEEE 754-2008 half-precision format and ignore the value of the FPCR.AHP bit, treating it as zero. This includes conversions between half-precision and single-precision or double-precision. This is a change from AArch64 Advanced SIMD and floating-point behavior.

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

- The half-precision instructions are subject to precisely 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.
2.4 Exception model

SVE adds hierarchical trap controls at EL3, EL2, and EL1. This is implemented using the following System registers:

- CPTR_EL3.ZE.
- CPTR_EL2.TZ, or CPTR_EL2.ZEN when HCR_EL2.E2H == 1.
- CPACR_EL1.ZEN.

2.4.1 SVE exception class

SVE defines the 0b011001 exception class value, in ESR_ELx.EC. The 0b011001 value is returned for exceptions due to attempted execution of SVE instructions and MRS/MSR instructions that access the ZCR_ELx System registers when trapped by CPACR_EL1.ZEN, CPTR_EL2.TZ, CPTR_EL2.ZEN, or CPTR_EL3.ZE.

See ESR_ELx for more details.

2.4.2 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.

2.4.3 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.

2.4.4 MOVPRFX exception behavior

For detailed information about the SVE MOVPRFX (predicated) and MOVPRFX (unpredicated) instructions, see Move prefix on page 4-44. When a MOVPRFX instruction is used correctly and execution of the pair of instructions generates a synchronous exception or causes entry to Debug state, then the restart address that is recorded in ESR_ELx or DLR_EL0 can be either:

- The address of the MOVPRFX instruction if it has caused no change to the architectural state.
- The address of the prefixed instruction with the MOVPRFX copy operation having been architecturally performed.

The choice of the above options can vary dynamically. Irrespective of the choice that is made, 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 and FAR or HPFAR registers will record the syndrome information and address that is associated with the erroneous prefixed instruction fetch and not the MOVPRFX instruction. If both instruction fetches would cause an Instruction abort, then the lower address is reported in the FAR register.
2 SVE Programmers' Model
2.4 Exception model
Chapter 3
SVE Memory Model

This chapter introduces the SVE Memory Model. This chapter contains the following sections:

- Atomicity on page 3-30.
- Alignment support on page 3-30.
- Endian support on page 3-30.
- Memory ordering on page 3-30.
- Memory fault handling on page 3-31.
3.1 SVE Memory Model

3.1.1 Atomicity

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

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

- For predicated SVE vector element or structure loads, where an element address is aligned to the size of the element in memory, that element 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 element 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. No atomicity between accesses to different bytes is ensured by the architecture.

3.1.2 Alignment support

Further to the rules relating to alignment of SIMD loads and stores in the sections titled Alignment Support and Memory types and attributes in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile the following behaviors are specific to accesses generated by SVE loads or stores when alignment checking is enabled. Alignment checking is enabled when SCTLR_ELx.A is 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 or stores:
  - Alignment checks are based on the size of the accessed element in memory, not the vector element size or the overall access size.
  - Inactive elements do not 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.

3.1.3 Endian support

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

- For predicated SVE vector element and structure loads and stores, the data size that is used for endianness conversions is the size in memory of the data to be transferred, 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.

3.1.4 Memory ordering

The SVE load and store instructions relax some of the rules relating to the observation order of reads or writes to memory described in the section titled Ordering and Observability in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile. The relaxations are as follows:

- Where an address dependency exists between two reads generated by SVE vector load instructions, in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order.
- In the absence of any other barrier mechanism to achieve order, two reads of the same memory location by the same observer that were generated by different load instructions can be observed in any order if one or both of the reads were generated by an SVE load instruction.
• Multiple reads generated by the same SVE load instruction, by the same observer, can be observed in any order, even if they are to the same address.
• Multiple writes that are generated by the same SVE store instruction, by the same observer, can be observed in any order, with the exception that writes by the same instruction to the same memory location must be observed in order of increasing vector element number.

For accesses by SVE instructions to a memory location with any Device memory type, the following additional exceptions apply:
• An explicit access to Device memory due to an SVE instruction can be performed for:
  — Any byte of an unpredicated vector load or store instruction.
  — Any byte of an unpredicated predicate register load or store instruction.
  — Only the active elements of a predicated vector load or store instruction that is not a first-fault vector load or non-fault vector load.
  — Only the first active element of a first-fault vector load instruction.
• An explicit access to Device memory due to an SVE prefetch or non-fault vector load instruction is never performed externally.
• Hardware speculation of data accesses performed to a Device memory location are not permitted by the architecture, with the following exceptions:
  — Explicit reads that are performed by an SVE load instruction are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a naturally aligned 64-byte window that contains at least 1 byte that is explicitly accessed by the instruction.
  — Explicit reads that are performed by an SVE non-temporal load instruction from memory locations with the Gathering attributes are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a naturally aligned 128-byte window that contains at least 1 byte that is explicitly accessed by the instruction.
• For all SVE instructions that load or store vector 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 these load or store instructions, and the order in which the registers are accessed is not architecturally defined. This applies to accesses to Normal memory and accesses to Device memory. See the section titled Normal Memory in the ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile for more information.

### 3.1.5 Memory fault handling

SVE uses the standard ARMv8-A Data Abort mechanism to report a synchronous fault that is generated by a memory access that was performed as a result of an SVE load or store instruction. That is:
• The appropriate ESR_ELx EC field is updated with 0b100100 or 0b100101, depending on the Exception level from which the fault occurred.
• Depending on the Exception level handling the fault, the FAR_EL1, FAR_EL2, FAR_EL3, or HPFAR_EL2 System register is updated with the lowest address applicable to the active element that the fault is reported against.

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 be abandoned as a result of an exception being taken during that sequence of accesses. On return from the exception the instruction is restarted, meaning that one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

SVE vector load and store instructions that give rise to 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 predicated SVE vector stores, memory locations that are associated with active elements that do not detect a fault are set to an UNKNOWN value. Memory locations that are associated with inactive elements or active elements that do detect a fault are preserved. A Data Abort exception is then taken.
For non-speculative, predicated SVE vector loads that are not first-fault or non-fault loads, all elements in the destination vector registers are set to an UNKNOWN value, irrespective of any predication, unless the destination register is a vector register that is also used as a base or offset register by the instruction in which case the original value of the register is preserved. A Data Abort exception is then taken.

For non-fault SVE loads, all accesses are speculative. The ESR_ELx exception syndrome and FAR_ELx and HPFAR_EL2 fault address registers are unchanged and no exception is taken. Instead, the FFR elements starting from the lowest-numbered element that gave rise to a fault, up to and including the highest-numbered element, are set to 0. If the final value of the lowest-numbered bit in an FFR element is 0, then the corresponding destination vector element is set to an UNKNOWN value. Otherwise, the destination vector element is set to the value read from memory, if it is active, or to zero if it is inactive. If any active element address references Device memory, then it updates the FFR and the destination vector in the same way as for a fault and the read is not performed externally.

For first-fault SVE loads, memory accesses due to the first active element are non-speculative and allow accesses to Device memory. Faults generated by the first active element are handled as described for a non-fault load, with no exception taken and the FFR is updated as described.

--- Note ---

The term fault in this section includes any MMU fault, Synchronous external abort including synchronous parity or ECC faults, and watchpoint debug events resulting from a data access generated by the execution of SVE vector load and store instructions. Furthermore, an implementation is permitted to suppress the read of any active element for a non-fault vector load, and any active element other than the first active element for a first-fault vector load. Any suppressed element read is reported as failed using the FFR, in the same manner as the reporting of faulting element reads.
Chapter 4
SVE Instruction Set

This chapter introduces the SVE instruction set. The Instruction set overview section of this chapter provides an overview of a subset of the instructions that are introduced by SVE. This chapter contains the following sections:

• SVE assembler language on page 4-34.
• Instruction set overview on page 4-35.
4.1 SVE assembler language

The SVE assembler language follows the AArch64 Advanced SIMD 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, with the following exceptions:

- Scalable vector register names Z0-Z31 and scalable 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, Z1.8 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 data operand 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 are destructive. To avoid ambiguity, the assembler language frequently uses a three-operand constructive notation, 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 4-36 for more information.
- A set of SVE aliases is defined for the AArch64 condition codes. See Table 2-3 on page 2-24 for further details.
4.2 Instruction set overview

4.2.1 Introduction

SVE adds a set of instructions to the existing ARMv8-A AArch64 instruction set. For details on the AArch64 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.

Note

This beta quality release does not include clickable links between listed instructions and the corresponding definitions. This issue is expected to be addresses in a future release.

4.2.2 Instruction aliases

The architected SVE instruction aliases are detailed in Table 4-1.

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIC (immediate, unpredicated)</td>
<td>Bitwise clear bits using immediate</td>
<td>AND (immediate)</td>
</tr>
<tr>
<td>CMPLE (vectors)</td>
<td>Compare signed less than or equal to vector, setting the condition flags</td>
<td>CMP&lt;cc&gt; (vectors)</td>
</tr>
<tr>
<td>CMPLE0 (vectors)</td>
<td>Compare unsigned lower than vector, setting the condition flags</td>
<td>CMP&lt;cc&gt; (vectors)</td>
</tr>
<tr>
<td>CMPLE0S (vectors)</td>
<td>Compare unsigned lower than or same as vector, setting the condition flags</td>
<td>CMP&lt;cc&gt; (vectors)</td>
</tr>
<tr>
<td>CMPLE0L (vectors)</td>
<td>Compare signed less than or vector, setting the condition flags</td>
<td>CMP&lt;cc&gt; (vectors)</td>
</tr>
<tr>
<td>EOR (unpredicted)</td>
<td>Bitwise exclusive OR with inverted immediate</td>
<td>EOR (immediate)</td>
</tr>
<tr>
<td>FACLE</td>
<td>Floating-point absolute compare less than or equal</td>
<td>FAC&lt;cc&gt;</td>
</tr>
<tr>
<td>FACLT</td>
<td>Floating-point absolute compare less than</td>
<td>FAC&lt;cc&gt;</td>
</tr>
<tr>
<td>FOMLE (vectors)</td>
<td>Floating-point compare less than or equal to vector</td>
<td>FOM&lt;cc&gt; (vectors)</td>
</tr>
<tr>
<td>FOMLT (vectors)</td>
<td>Floating-point compare less than vector</td>
<td>FOM&lt;cc&gt; (vectors)</td>
</tr>
<tr>
<td>FMOV (immediate, predicated)</td>
<td>Move 8-bit floating-point immediate to vector elements</td>
<td>FCPY</td>
</tr>
<tr>
<td>FMOV (immediate, unpredicated)</td>
<td>Move 8-bit floating-point immediate to vector elements</td>
<td>FDUP</td>
</tr>
<tr>
<td>FMOV (zero, predicated)</td>
<td>Move floating-point 0 to vector elements</td>
<td>CPY</td>
</tr>
<tr>
<td>FMOV (zero, unpredicated)</td>
<td>Move floating-point 0 to vector elements</td>
<td>DUP (immediate)</td>
</tr>
<tr>
<td>MOV (bitmask, immediate, unpredicated)</td>
<td>Move logical bitmask immediate to vector</td>
<td>DUPM</td>
</tr>
<tr>
<td>MOV (immediate, predicated)</td>
<td>Move signed integer immediate to vector elements</td>
<td>CPY (immediate)</td>
</tr>
</tbody>
</table>
4.2.3 Load, store, and prefetch instructions

SVE vector load and store instructions transfer data to or from elements of one or more vector or predicate 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 specified for the vector register being transferred. Table 4-2 shows the supported instruction suffixes for SVE load, store, and prefetch instructions.

<table>
<thead>
<tr>
<th>Instruction suffix</th>
<th>Element access size and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unsigned 8-bit byte</td>
</tr>
<tr>
<td>H</td>
<td>Unsigned 16-bit halfword or half-precision floating-point</td>
</tr>
<tr>
<td>W</td>
<td>Unsigned 32-bit word or single-precision floating-point</td>
</tr>
<tr>
<td>D</td>
<td>Unsigned 64-bit doubleword or double-precision floating-point</td>
</tr>
<tr>
<td>SB</td>
<td>Signed 8-bit byte</td>
</tr>
<tr>
<td>SH</td>
<td>Signed 16-bit halfword</td>
</tr>
<tr>
<td>SW</td>
<td>Signed 32-bit word</td>
</tr>
</tbody>
</table>

The element size that is specified for the register being transferred is always greater than or equal to the element access size. When the element size specified for the register being transferred is strictly greater than the 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 in Table 4-2.
- For store instructions, each vector element is truncated to the element access size.

---

Table 4-2 SVE memory element access instruction suffixes

<table>
<thead>
<tr>
<th>Instruction suffix</th>
<th>Element access size and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unsigned 8-bit byte</td>
</tr>
<tr>
<td>H</td>
<td>Unsigned 16-bit halfword or half-precision floating-point</td>
</tr>
<tr>
<td>W</td>
<td>Unsigned 32-bit word or single-precision floating-point</td>
</tr>
<tr>
<td>D</td>
<td>Unsigned 64-bit doubleword or double-precision floating-point</td>
</tr>
<tr>
<td>SB</td>
<td>Signed 8-bit byte</td>
</tr>
<tr>
<td>SH</td>
<td>Signed 16-bit halfword</td>
</tr>
<tr>
<td>SW</td>
<td>Signed 32-bit word</td>
</tr>
</tbody>
</table>

The element size that is specified for the register being transferred is always greater than or equal to the element access size. When the element size specified for the register being transferred is strictly greater than the 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 in Table 4-2.
- For store instructions, each vector element is truncated to the element access size.
Where the vector element size and the 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, as defined in Table 4-3.

<table>
<thead>
<tr>
<th>Vector element</th>
<th>Packed access</th>
<th>Unpacked access</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
<td>-</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>B, SB</td>
</tr>
<tr>
<td>S</td>
<td>W</td>
<td>H, SH, B, SB</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>W, SW, H, SH, B, SB</td>
</tr>
</tbody>
</table>

In the following descriptions, a scalar base register refers to a AArch64 general-purpose register, X0-X30, or the current stack pointer, SP. A scalar index register refers only to an AArch64 general-purpose register.

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

- Predicated single vector contiguous element accesses.
- Predicated multiple vector contiguous structure load/store on page 4-38.
- Predicated non-contiguous element accesses on page 4-38.
- Predicated replicating element loads on page 4-38.
- Unpredicated vector register load/store on page 4-39.
- Unpredicated predicate register load/store on page 4-39.

All predicated load instructions zero the inactive elements of the destination vector.

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.

**Predicated single vector contiguous element accesses**

Predicated contiguous load and store instructions access contiguous memory locations starting from an address that is defined by a scalar base register plus either a scalar index register or an immediate index value in the range -8 to 7, inclusive. Prefetch instructions address contiguous memory locations in a similar manner, with indexing from either a scalar register or an immediate index in the range -32 to 31, inclusive.

- The immediate index value is multiplied by the total vector access size, which is the number of accessed elements in memory, ignoring predication, multiplied by the element access size in bytes.
- The scalar index register value is multiplied by the element access size in bytes.
- 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 element access size.

**Supported addressing modes**

<table>
<thead>
<tr>
<th>Addressing modes</th>
<th>Assembler syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar base + scalar index</td>
<td>[&lt;Xn</td>
</tr>
<tr>
<td>Scalar base + immediate offset</td>
<td>[&lt;Xn</td>
</tr>
</tbody>
</table>
Predicated multiple vector contiguous structure load/store

Structure load, LD2, LD3, LD4, and store, ST2, ST3, ST4, instructions transfer two, three, or four vector registers from or to contiguous structures of two, three, or four fields in memory, starting from an address that is defined by a scalar base register plus a scalar index register or an immediate index value in the range -8 to 7, inclusive.

- The immediate index value is multiplied by the total access size, which is the number of vector elements, ignoring predication, multiplied by the element size in bytes multiplied by the number of transferred registers.
- The scalar index register value is multiplied by the element access size in bytes.
- Each predicate element applies to a complete structure in memory, or equivalently to the same vector element position within each of the two, three, or four transferred vector registers.
- These instructions support packed data accesses only.
- When alignment checking is enabled, the base address must be aligned to the element access size.

Supported addressing modes

<table>
<thead>
<tr>
<th>Supported addressing modes</th>
<th>Assembler syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar base + scalar index</td>
<td>[&lt;Xn]</td>
</tr>
<tr>
<td>Scalar base + immediate offset</td>
<td>[&lt;Xn]</td>
</tr>
</tbody>
</table>

Predicated non-contiguous element accesses

Predicated non-contiguous element accesses access non-contiguous memory locations that are specified by a scalar base register plus a vector of offsets, or a vector base address plus an immediate byte offset that is a multiple of the element access size in the range 0-31 multiplied by the access size, inclusive.

- The element size that is specified for the register being transferred is 32 bits or 64 bits (.S or .D).
- Vector elements that are used to form an address are 32 bits or 64 bits wide and always match the element size that is specified for the register being transferred. If the offset vector element size is 32 bits, then each offset is zero-extended or sign-extended to 64 bits. 32-bit vector base address elements are always zero-extended.
- Offset vector elements are optionally multiplied by the element access size, in bytes, except for prefetch instructions where offset vector elements must be multiplied by the element access size, in bytes.
- Load, LD1, store, ST1, and first-fault load, LDFF1, instructions support packed and unpacked data accesses as defined in Table 4-3 on page 4-37.
- When alignment checking is enabled for loads and stores, the computed address of each element must be aligned to the element access size.

Assembler syntax

<table>
<thead>
<tr>
<th>Supported addressing modes</th>
<th>64-bit form</th>
<th>32-bit form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar base + vector offset</td>
<td>[&lt;Xn]</td>
<td>SP&gt;, &lt;Zm&gt;.D{, LSL #&lt;sh&gt;}]</td>
</tr>
<tr>
<td>Vector base + immediate offset</td>
<td>[&lt;Zn&gt;.D{, #&lt;uimm&gt;}]</td>
<td>[&lt;Zn&gt;.S{, #&lt;uimm&gt;}]</td>
</tr>
</tbody>
</table>

Predicated replicating element loads

Load and replicate instructions read one or more contiguous memory locations starting at a scalar base address plus a scalar index register or immediate byte offset, depending on the instruction.

- Load a single element value and replicate it into all active elements of the destination vector, LD1R, supporting packed and unpacked data accesses, with an immediate byte offset that is a multiple of the element access size, in the range 0-63 multiplied by the access size, inclusive.
- Load a predicated 128-bit quadword vector segment from contiguous element values and replicate that segment into all segments of the destination vector, LD1RQ, supporting only packed data accesses, using a scalar index register that is scaled by the 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, the base address must be aligned to the element access size.

<table>
<thead>
<tr>
<th>Supported addressing modes</th>
<th>Assembler syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar base + scalar index</td>
<td>[&lt;Xn</td>
</tr>
<tr>
<td>Scalar base + immediate offset</td>
<td>[&lt;Xn</td>
</tr>
</tbody>
</table>

**Unpredicated vector register load/store**

The unpredicated vector register load, LDR, and store, STR, instructions transfer a single vector register without predication from or to memory locations specified by a scalar base register plus an immediate index value in the range -256 to 255, inclusive.

• 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 elements in ascending element order, without endian conversion.
• When alignment checking is enabled, the base address must be 16-byte aligned.

<table>
<thead>
<tr>
<th>Supported addressing mode</th>
<th>Assembler syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar base + immediate offset</td>
<td>[&lt;Xn</td>
</tr>
</tbody>
</table>

**Unpredicated predicate register load/store**

The predicate register load, LDR, and store, STR, instructions transfer a single predicate register without predication from or to memory locations specified by a scalar base register plus an immediate index value in the range -256 to 255, inclusive.

• The immediate index value is multiplied by the current vector register length, in bytes, divided by 8.
• The data transfer is performed as a contiguous stream of bytes, each byte containing 8 consecutive predicate bits, in ascending bit order, without endian conversion.
• When alignment checking is enabled, the base address must be 2-byte aligned.

<table>
<thead>
<tr>
<th>Supported addressing mode</th>
<th>Assembler syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar base + immediate offset</td>
<td>[&lt;Xn</td>
</tr>
</tbody>
</table>

**4.2.4 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.

Reversed destructive instructions for non-commutative subtract and divide operations overwrite the subtrahend and divisor instead of the minuend and dividend.

The supported instructions are: ABS, ADD, CNOT, MAD, MLA, MLS, MSB, MUL, NEG, SABD, SDIV, SDIVR, SMAX, SMIN, SMULH, SQADD, SQSUB, SUB, SUBR, SXTB, SXTW, UABD, UDIV, UDIVR, UMAX, UMIN, UMULH, UQADD, UQSUB, UXTB, UXTW.
4SVE Instruction Set
4.2 Instruction set overview

Integer dot product

The integer partial dot product instructions delimit the source vectors into groups of four 8-bit or 16-bit integer elements. Within each group of four elements, the elements in the source vector are multiplied by the corresponding elements in the second source vector and 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 supported instructions are: SDOT, UDOT.

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 condition flags based on the predicate result.

The supported instructions are: CMP EQ, CMPGE, CMP GT, CMP HI, CMP H S, CMP LE, CMP LT, CMP NE.

4.2.5 Vector address calculation

These instructions compute vectors of addresses and addresses of vectors. This includes instructions to add a multiple of the current vector or predicate register size to a general-purpose register.

The supported instructions are: AD OPL, ADD VL, ADR, RD VL.

Note

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 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 offset.

32-bit values are computed by the addition of a 32-bit base and a scaled 32-bit unsigned offset.

64-bit values are computed by one of:

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

4.2.6 Bitwise operations

Bitwise logical operations

Bitwise logical operations on vectors. Where operations are unpredicated, the operations are element size agnostic.

The supported instructions are: AND, BIC, DUP M, EOR, MOV, NOT, ORR.

Bitwise shift, permute, and count

Bitwise shifts, permutes, 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.

Reversed vector shifts provide a destructive shift that overwrites the shift amount vector, instead of the vector being shifted.

The supported instructions are: ASR, AS RD, AS RR, CL S, CL Z, CNT, L S L, L SLR, L SR, L S RR, R B I T.
4.2.7 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.

Reversed destructive instructions for non-commutative subtract and divide operations overwrite the subtrahend and divisor instead of the minuend and dividend.

The supported instructions are: FABD, FABS, FADD, FDIV, FDIVR, FMAX, FMAXNM, FMIN, FMINNM, FMLA, FMLS, FMSB, FMUL, FMULX, FNEG, FNMAD, FNMLA, FNMLS, FNMSB, FRECPE, FRECPX, FRSQRT, FRQRTE, FRQRTS, FSQRT, FSUB, FSUBR.

**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, 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 FOPLA instructions perform a transformation of the operands to allow the creation of a multiply-add operation 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 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 FOPLA instructions can then be combined as follows:

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

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 Z 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, the behavior of calculations such as \((\infty+\infty i)\) multiplied by \((0+i)\) will be \((\text{NaN}+\text{NaN}i)\) using these instructions, rather than the result expected by ISO C, which is complex \(\infty\).

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

The supported instructions are: FCADD, FOPLA.
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.

The supported instructions are: FCVT, FCVTZS, FCVTZU, FRINTA, FRINTI, FRINTM, FRINTN, FRINTX, FRINTZ, SCVT, UCVT.

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.

The supported instructions are: FACGE, FACGT, FACLE, FACLT, FOMEQ, FOMGE, FOMGT, FOMLE, FOMLT, FOMNE, FOMU0.

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.

The supported instructions are: FTMAD, FTSMUL, FTSSEL, FEXPA.

4.2.8 Predicate operations

These instructions relate to operations that manipulate the predicate registers.

Predicate initialization

These instructions initialize predicate register elements.

Predicate elements can be initialized to be FALSE, or to be TRUE when the element number satisfies a named predicate constraint. If the constraint specifies more elements than are available at the current vector length, then all elements of the destination predicate are set to FALSE.

Unspecified or out of range constraint encodings generate an empty predicate and do not cause an Undefined Instruction exception.

Some of these instructions are insensitive to the predicate organization and specify an explicit 8-bit byte element size qualifier, but an assembler must accept any qualifier, or none.

The supported instructions are: PFALSE, PTRUE, PTRUES.

Predicate logical operations

These instructions perform bitwise logical operations that are element size agnostic, operating on all bits of the predicate. The flag-setting variants set the condition flags based on the predicate result. Inactive elements in the destination predicate register are set to 0.

These instructions are insensitive to the predicate organization and specify an explicit 8-bit byte element size qualifier, but an assembler must accept any qualifier, or none.

The supported instructions are: AND, ANDS, BIC, BICS, EOR, EORS, NAND, NANDS, NOR, NORS, ORN, ORNS, ORR, ORRS, PTEST.
FFR predicate handling

These instructions work with first-fault and non-fault loads using the FFR to determine which elements have been successfully loaded and which remain to be processed. The RDFFRS instruction sets the condition flags based on the predicate result.

These instructions are insensitive to the predicate organization and specify an explicit 8-bit byte element size qualifier, but an assembler must accept any qualifier, or none.

The supported instructions are: RDFFR, RDFFRS, SETFFR, WRFFR.

Predicate counts

These instructions count either the number of active elements in a predicate register, or the number of elements implied by a named predicate constraint operand. 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 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.

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

The supported instructions are: CNTB, CNTD, CNTH, CNTP, CNTW, DECB, DECD, DECH, DECP, DECK, INCB, INC, INCD, INCP, INCH, INCW, SQDECB, SQDECD, SQDECH, SQDECP, SQDECQ, SQINCB, SQINC, SQINCP, SQINOW, UQDECB, UQDECD, UQDECH, UQDECP, UQDECQ, UQINCB, UQINCD, UQINCH, UQINCP, UQINOW.

Loop control

Instructions that are used to control counted vector loops and vector loops with data-dependent termination conditions.

These instructions create a loop partition predicate that contains active elements up to the point where the loop should terminate, optionally setting the condition flags to control a conditional branch. Two loop concepts are supported:

- For simple counted loops, the WHILE instructions compare the incrementing value of their first scalar source vector with a second fixed scalar operand, for each element of the destination predicate. The result is a partition predicate with elements set to TRUE while the comparison is true, and FALSE thereafter.
  - The supported instructions are: WHILELE, WHILELO, WHILELS, WHILELT.

- For data-dependent termination conditions, it is necessary to convert the result of a vector comparison into a loop partition predicate. The partition truncates the current vector iteration immediately before or after the first TRUE comparison.
  - The BRKA and BRKB instructions set active destination predicate elements to TRUE up to but excluding (BRKB) or up to and including (BRKA) the first TRUE element in their source predicate, setting subsequent elements to FALSE.
  - The BRKPA and BRKPB instructions propagate a truncated partition from their first source predicate containing the result of a previous BRKB or BRKPB instruction to their destination predicate, but otherwise generate the destination predicate from their second source predicate as described for the BRKA and BRKB instruction.
  - The BRKN instructions propagate a truncated partition from their first source predicate containing the result of a previous BRKB or BRKPB instruction to the next partition predicate in the destination and second source register, but otherwise leave the destination predicate unchanged.
  - The supported instructions are: BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS.

The PFIRST instruction can be used to generate the initial predicate for a do-while loop where the first iteration must be performed on only the first element in the current partition.

The supported instruction is: PFIRST.
4 SVE Instruction Set
4.2 Instruction set overview

Serialized operations
These instructions permit active elements within a vector to be processed sequentially without packing or unpacking the vector.

The supported instructions are: PNEXT, CTERMEQ, CTERMNE.

4.2.9 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.

The supported instructions are: CPY, DUP, FCPY, FDUP, SEL.

Element permute and shuffle
These instructions move data between elements of a vector, or between vector elements and scalar registers.

- Unconditionally extract the last active element of a vector or the following element.
  - The supported instructions are: CLASTA, CLASTB.

- Conditionally 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 vector or predicate register.
  - The supported instructions are: COMPACT, SPLICE, TBL.

- Fixed permute instructions where the permutation is encoded in the instruction.
  - The supported instructions are: DUP, EXT, INSR, REV, REV8, REVH, REWH, SUNPKHI, SUNPKLO, TRN1, TRN2, UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, ZIP2.

Predicate move operations
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 supported instructions are: PUNPKHI, PUNPKLO, REV, TRN1, TRN2, UZP1, UZP2, ZIP1, ZIP2, SEL.

Index vector generation
The INDEX instruction initializes a vector horizontally in by settings 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.

The supported instructions are: INDEX.

Move prefix
The MOVPRFX (predicated) and MOVPRFX (unpredicated) instructions are vector move operations that also provide a hint to the implementation that the instruction can be combined with the destructive instruction that follows it in program order to create a single constructive operation, or to convert an instruction with merging predication to use zeroing predication.

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

- The destination register field implicitly specifies one of the source operands, which means that it is a destructive binary operation or unary operation with merging predication, excluding MOVPRFX.
4.2.10 Reduction operations

Horizontal reductions

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

The floating-point horizontal accumulating sum instruction, \texttt{FADDA}, operates strictly right to left across a vector in order of ascending element number, 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, meaning the order of evaluation is not specified by the architecture.

The supported instructions for integer reduction are: \texttt{ANDV}, \texttt{EORV}, \texttt{ORV}, \texttt{SADDV}, \texttt{SMAXV}, \texttt{SMINV}, \texttt{UADDV}, \texttt{UMAXV}, \texttt{UMINV}.

The supported instructions for floating-point reduction are: \texttt{FADDA}, \texttt{FADDV}, \texttt{FMAXNMV}, \texttt{FMAXV}, \texttt{FMINNMV}, \texttt{FMINV}.

4.2.11 Predication support

For predicated SVE instructions, inactive source vector elements are ignored and cannot cause a side-effect such as updating FPSR, or generating an exception. Inactive destination vector elements are either set to zero or remain unchanged. If the inactive destination vector elements are set to zero, this is known as zeroing predication. If the inactive destination vector elements are unchanged, this is known as merging predication.

The following instructions support both zeroing and merging predication:

- \texttt{BRKA}, \texttt{BRKB}.
- \texttt{CPY} (immediate).
- \texttt{MOVPRFX} (predicated).

The following instructions support zeroing predication only:

- \texttt{BRKN}, \texttt{BRKNS}, \texttt{BRKPA}, \texttt{BRKPAS}, \texttt{BRKPBR}, \texttt{BRKPPBS}.
- All predicated load instructions.

\textbf{Note}

\texttt{LDR} (vector) and \texttt{LDR} (predicate) do not support predication.

- The predicated read FFR instruction:
  - \texttt{RDFFR} (predicated), \texttt{RDFFRS} (predicated).
Logical operations on predicate registers


- All vector compare instructions:
  - **CMP<cc>**, **FAC<cc>**, and **FCM<cc>** compare instructions.

The following instructions support merging predication only:

- Predicated vector data-processing instructions:
  - **CPY** (scalar) and **CPY** (SIMD&FP scalar).
Chapter 5
SVE System Registers

This chapter introduces the changes that are made to the existing ARMv8-A System registers by SVE as well as the new System registers that are introduced by SVE. This chapter contains the following sections:

-  Modified AArch64 System registers on page 5-48.
-  New AArch64 System registers on page 5-48.
5.1 SVE System registers

SVE modifies some of the existing AArch64 System registers and defines new AArch64 System registers. 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.

5.1.1 Modified AArch64 System registers

The existing AArch64 System registers that are modified by SVE are outlined in Table 5-1.

<table>
<thead>
<tr>
<th>Register</th>
<th>Change</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_AA64PFR0_EL1</td>
<td>Defines bits[35:32] as the SVE field.</td>
<td>The SVE field indicates whether the Scalable Vector Extension is implemented.</td>
</tr>
<tr>
<td>CPACR_EL1</td>
<td>Defines bits[17:16] as the ZEN field.</td>
<td>The ZEN field separately enables access to SVE functionality from EL1 and EL0.</td>
</tr>
<tr>
<td>CPTR_EL2</td>
<td>When HCR_EL2.E2H=0, defines bit[8] as TZ and when HCR_EL2.E2H=1, defines bit[17:16] as the ZEN field.</td>
<td>The TZ field traps access to SVE functionality from Non-secure EL1&amp;0 to EL2. The ZEN field separately enables access to SVE from EL2 and from EL0.</td>
</tr>
<tr>
<td>CPTR_EL3</td>
<td>Defines bit[8] as EZ.</td>
<td>The EZ field enables access to SVE functionality from EL0, EL1, EL2, and EL3.</td>
</tr>
<tr>
<td>TCR_EL1</td>
<td>Defines bit[54] as NFD1 and bit[53] as NFD0.</td>
<td>The NDF1 and NFD0 fields disable translation table walks caused by certain first-fault and non-fault vector load instructions.</td>
</tr>
<tr>
<td>EDPFR</td>
<td>Architecturally mapped to ID_AA64PFR0_EL1 and presents the same information, including the new SVE field.</td>
<td>-</td>
</tr>
<tr>
<td>ESR_ELx</td>
<td>New exception class, 0b011001, added to the EC field description.</td>
<td>The new EC code, 0b011001, identifies accesses to SVE functionality when disabled by CPACR_EL1, CPTR_EL2, or CPTR_EL3.</td>
</tr>
</tbody>
</table>

5.1.2 New AArch64 System registers

The new AArch64 System registers defined by SVE are outlined in Table 5-2.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_AA64ZFR0_EL1</td>
<td>SVE feature ID register 0</td>
</tr>
<tr>
<td>ZCR_EL1</td>
<td>SVE control register to constrain the accessible vector length at EL1 and EL0</td>
</tr>
<tr>
<td>ZCR_EL2</td>
<td>SVE control register to constrain the accessible vector length at EL2 and Non-secure EL1 and EL0</td>
</tr>
<tr>
<td>ZCR_EL3</td>
<td>SVE control register to constrain the accessible vector length at EL3, EL2, EL1, and EL0</td>
</tr>
</tbody>
</table>

Writes to ZCR_ELx are self-synchronizing and appear to occur in program order relative to other instructions.
When executing at an Exception level that is constrained to use a vector length that is smaller than the maximum implemented vector length, then the upper bits of the vector register, predicate registers, and FFR are inaccessible. On taking an exception from an Exception level that is more constrained to a target Exception level that is less constrained, or on writing a larger value to ZCR_ELx.LEN, then the previously inaccessible bits of these registers that become accessible must either be set to zero or be holding the same value that the same register had before executing at the more constrained size. The choice between these options is IMPLEMENTATION DEFINED and can vary dynamically. The values of the upper bits, in this case, must be regarded as UNKNOWN.

If a SIMD&FP register is written by an AArch64 floating-point or Advanced SIMD instruction while SVE instructions at the same Exception level would be trapped, then the SVE vector length is treated as 128 bits. If SVE is later enabled, then bit[128] and higher of the SVE vector register, where implemented, might contain zero or the original value of the register before SVE was disabled. The choice between these two options is IMPLEMENTATION DEFINED and can vary dynamically. The values of the upper bits, in this case, must be regarded as UNKNOWN.

However, if floating-point and SVE are both disabled at all Exception levels below the target Exception level, in the current Security state, then any SVE register state at the target Exception level is preserved.
Chapter 6
SVE Debug

This chapter introduces the changes that are made to ARMv8-A AArch64 debug by SVE. This chapter contains the following sections:

- *Self-hosted debug* on page 6-52.
- *External debug* on page 6-53.
6.1 Self-hosted debug

6.1.1 Watchpoints

For all vector loads and stores, Watchpoint debug events are only generated by address matches that occur due to active vector elements.

For first-fault vector loads:
- Watchpoint debug events are only generated by address matches that correspond to the first active element.
- If no Watchpoint debug event is generated corresponding to the first active element, and another active element matches a configured watchpoint, then the latter watchpoint is suppressed, and the corresponding element and all higher-numbered elements of the FFR are set to 0.

For non-fault vector loads:
- Debug watchpoint events are not generated. However, the corresponding FFR elements are set to 0 and the access is not performed.

A memory access that can trigger a watchpoint is any non-speculative, single-copy atomic access performed as a result of an active element by an SVE predicated vector load or store instruction, or any byte access performed by an SVE unpredicated register load or store instruction.

6.1.2 MOVRPFX 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 predictable 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 2-27.
6.2 External debug

6.2.1 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>
<thead>
<tr>
<th>Instruction</th>
<th>Change in Debug state</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPNE (immediate form, byte element size only)</td>
<td>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.</td>
</tr>
</tbody>
</table>

6.2.2 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 and byte element size only)
- PTRUE (ALL variant, with byte element size only)
- RDFFF (unpredicated)
- WRFFF (unpredicated)
- EXT
- TNSR (scalar)
- DUP (scalar)

6.2.3 SVE instructions that are CONSTRAINED UNPREDICTABLE in Debug state

All other SVE instructions are CONSTRAINED UNPREDICTABLE in Debug state, with the following permissible behaviors:

- The instruction executes as a NOP.
- If the instruction modifies PSTATE, it sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values.
- If the instruction is similar to an SVE instruction that is described in SVE instructions that are changed in Debug state or SVE instructions that are unchanged in Debug state that has defined behavior in Debug state, it executes as that instruction.
- The instruction has the same behavior as in Non-debug state.
6 SVE Debug
6.2 External debug
Chapter 7
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:

• New performance monitor events on page 7-56.
• Existing PMU events affected by SVE on page 7-57.
7.1 New performance monitor events

7.1.1 New SVE PMU events

Performance monitor event numbers 0x8000 - 0x80FF, in the extended event number space that is introduced in ARMv8.1, are reserved for use by SVE. The new PMU events that are defined by SVE are summarized in Table 7-1.

<table>
<thead>
<tr>
<th>Event number</th>
<th>Event type</th>
<th>Event mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8002</td>
<td>Architectural</td>
<td>SVE_INST_RETIRED</td>
<td>SVE instructions architecturally executed</td>
</tr>
<tr>
<td>0x8003</td>
<td>Architectural</td>
<td>SIMD_INST_RETIRED</td>
<td>AArch64 SIMD instructions architecturally executed</td>
</tr>
<tr>
<td>0x8006</td>
<td>Micro-architectural</td>
<td>SVE_INST_SPEC</td>
<td>SVE operations speculatively executed</td>
</tr>
<tr>
<td>0x8007</td>
<td>Micro-architectural</td>
<td>SIMD_INST_SPEC</td>
<td>AArch64 SIMD operations speculatively executed</td>
</tr>
</tbody>
</table>

Note
Implementations of SVE are architecturally required to include the following PMU events:
- At least one of SVE_INST_RETIRED and SVE_INST_SPEC.
- At least one of SIMD_INST_RETIRED and SIMD_INST_SPEC.

PMU event descriptions

0x8002, SVE_INST_RETIRED, SVE instructions architecturally executed
The counter increments when an SVE instruction is architecturally executed.
It is IMPLEMENTATION DEFINED whether the MOVPRFX instruction and instructions that do not directly access SVE vector or predicate registers are counted.

0x8003, SIMD_INST_RETIRED, AArch64 SIMD instructions architecturally executed
The counter increments when an SVE instruction is architecturally executed, as described in SVE_INST_RETIRED.
In AArch64 state, the counter also increments when instructions from the Advanced SIMD classes are architecturally executed. Instructions that are counted by DP_SPEC and CRYPTO_SPEC are not counted.
In AArch32 state, it is IMPLEMENTATION DEFINED which instructions are counted.

0x8006, SVE_INST_SPEC, SVE operations speculatively executed
The counter increments when an SVE operation is speculatively executed.
It is IMPLEMENTATION DEFINED whether the MOVPRFX instruction and operations that do not directly access SVE registers are counted.

0x8007, SIMD_INST_SPEC, AArch64 SIMD operations speculatively executed
The counter increments when an SVE operation is architecturally executed, as described in SVE_INST_SPEC.
In AArch64 state, the counter also increments when operations from the AArch64 Advanced SIMD classes are speculatively executed. Operations that are counted by DP_SPEC or CRYPTO_SPEC are not counted.
In AArch32 state, it is IMPLEMENTATION DEFINED which operations are counted.
### 7.2 Existing PMU events affected by SVE

The following ARMv8-A PMU events will also count SVE instructions and operations. These events are described in Table 7-2. All other ARMv8-A PMU events will not count SVE instructions and operations.

<table>
<thead>
<tr>
<th>Event number</th>
<th>Event mnemonic</th>
<th>Definition</th>
<th>SVE clarification</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0006</td>
<td>LD_RETIRED</td>
<td>Instruction architecturally executed, load</td>
<td>Counts architecturally executed SVE load instructions.</td>
</tr>
<tr>
<td>0x0007</td>
<td>ST_RETIRED</td>
<td>Instruction architecturally executed, store</td>
<td>Counts architecturally executed SVE store instructions.</td>
</tr>
<tr>
<td>0x0008</td>
<td>INST_RETIRED</td>
<td>Instruction architecturally executed</td>
<td>Counts architecturally executed SVE instructions. It is IMPLEMENTATION DEFINED whether MOVPRFX is counted by this event.</td>
</tr>
<tr>
<td>0x000F</td>
<td>UNALIGNED_LDST_RETIRED</td>
<td>Instruction architecturally executed, unaligned load or store</td>
<td>Counts architecturally executed SVE load or store instructions that access at least one unaligned element address that would generate an alignment fault or change the value of the FFR register when Alignment fault checking is enabled.</td>
</tr>
<tr>
<td>0x0013</td>
<td>MEM_ACCESS</td>
<td>Data memory access</td>
<td>Counts memory reads and writes as a result of SVE load or store instructions. The number of accesses generated by each SVE instruction is IMPLEMENTATION DEFINED.</td>
</tr>
<tr>
<td>0x001B</td>
<td>INST_SPEC</td>
<td>Operation speculatively executed</td>
<td>Counts speculatively executed SVE operations. It is IMPLEMENTATION DEFINED whether MOVPRFX is counted by this event.</td>
</tr>
<tr>
<td>0x0066</td>
<td>MEM_ACCESS_RD</td>
<td>Data memory access, read</td>
<td>Similar to MEM_ACCESS, but only counts reads.</td>
</tr>
<tr>
<td>0x0067</td>
<td>MEM_ACCESS_WR</td>
<td>Data memory access, write</td>
<td>Similar to MEM_ACCESS but only counts writes.</td>
</tr>
<tr>
<td>0x0068</td>
<td>UNALIGNED_LD_SPEC</td>
<td>Unaligned access, read</td>
<td>Counts speculatively executed SVE load operations that access at least one unaligned element address.</td>
</tr>
<tr>
<td>0x0069</td>
<td>UNALIGNED_ST_SPEC</td>
<td>Unaligned access, write</td>
<td>Counts speculatively executed SVE store operations that access at least one unaligned element address.</td>
</tr>
<tr>
<td>0x006A</td>
<td>UNALIGNED_LDST_SPEC</td>
<td>Unaligned access</td>
<td>Counts speculatively executed SVE load and store operations that access at least one unaligned element address.</td>
</tr>
<tr>
<td>0x0070</td>
<td>LD_SPEC</td>
<td>Operation speculatively executed, load</td>
<td>Counts speculatively executed SVE load operations.</td>
</tr>
<tr>
<td>0x0071</td>
<td>ST_SPEC</td>
<td>Operation speculatively executed, store</td>
<td>Counts speculatively executed SVE store operations.</td>
</tr>
<tr>
<td>0x0072</td>
<td>LDST_SPEC</td>
<td>Operation speculatively executed, store or store</td>
<td>Counts speculatively executed SVE load and store operations.</td>
</tr>
</tbody>
</table>
7 SVE Performance Monitors Extension
7.2 Existing PMU events affected by SVE
Appendix A
Flag-setting instructions

This appendix provides a list of the SVE instructions that can set condition flags and contains the following section:

• Flag-setting instructions on page A-60.
A.1 Flag-setting instructions

The flag-setting instructions are:

• Flag-setting logical operations on predicate registers:
  — ANDS
  — BICS
  — EORS
  — NANDS
  — NORS
  — ORNS
  — ORRS

• Flag-setting break instructions:
  — BRKAS
  — BRKBS
  — BRKNS
  — BRKPAS
  — BRKPBS

• Integer compare instructions, \texttt{CMP<cc>}

• Compare and terminate instructions, \texttt{CTERMEO} and \texttt{CTERMNE}.

• Predicate element operations:
  — PFIRST
  — PNEXT
  — PTEST
  — PTRUES

• Flag-setting FFR read, \texttt{RDFFRS}

• Counted loop instructions:
  — WHILELE
  — WHILELO
  — WHILELS
  — WHILELT
Appendix B
Predicate-setting instructions

This appendix provides a list of the SVE instructions that can set predicate registers and contains the following section:

• *Predicate-setting instructions* on page B-62.
B.1 Predicate-setting instructions

- AND, ANDS
- BIC, BICS
- BRAK, BRKAS
- BRKB, BRKBS
- BRKN, BRKNS
- BRKPA, BRKPS
- BRKPB, BRKPBS
- CMP<cc> (immediate, vectors, wide elements)
- EOR, EORS
- FAC<cc>
- FCM<cc> (vectors, zero)
- LDR
- NAND, NANDS
- NOR, NORS
- ORN, ORNS
- ORR, ORRS
- PFALSE
- PFIRST
- PNEXT
- PTRUE, PTRUES
- PUNPKHI, PUNPKLO
- ROFFR, ROFFRS
- REV
- SEL
- TRN1, TRN2
- UZP1, UZP2
- WHILELO, WHILELS, WHILELT
- ZIP1, ZIP2