Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension

Document number	DDI 0584
Document quality	EAC
Document version	B.a
Document confidentiality	Non-confidential
Document build information	Printed on: May 24, 2021.

Release information

Date	Version	Changes
2021/May/24	Non-Confidential EAC B.a	• SVE as EAC • SVE2 as EAC
2021/Jan/22	Non-Confidential EAC A.i	• EAC maintenance release including FEAT_AFP
2020/Jul/17	Non-Confidential EAC A.h	• EAC maintenance release including BFloat and Matrix Multiplication instructions
2020/Feb/21	Non-Confidential EAC A.g	• Updated EAC release incorporating BFloat and Matrix Multiplication instructions
2019/Jul/05	Non-Confidential EAC A.f	• EAC Maintenance release
2018/Oct/31	Non-Confidential EAC A.e	• EAC Maintenance release
2017/Dec/21	Non-Confidential EAC A.d	• EAC Maintenance release
2017/Dec/15	Non-Confidential EAC A.c	• EAC Maintenance release
2017/Aug/21	Non-Confidential EAC A.b	• EAC release
2017/Mar/31	Non-Confidential Beta A.a	• Beta

Product Status

This manual covers multiple versions of the architecture. The content relating to both *SVE* and *SVE2* is at EAC quality.

EAC quality means that:

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

Contents

Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension

Preface

About this supplement

This supplement is the *Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension*.

This supplement describes the changes and additions introduced by *SVE* to the Armv8-A architecture.

This supplement also describes the changes and additions introduced by *SVE2* to the Armv9-A architecture.

For *SVE*, this supplement is to be read in conjunction with all of the following documents:

- Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile.
- System Register XML for Armv8.
- A64 ISA XML for Armv8.

Together, these documents provide a full description of the Armv8-A Scalable Vector Extension.

For *SVE2*, this supplement is to be read in conjunction with all of the following documents:

- Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile.
- Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile.
- System Register XML for Armv9.
- A64 ISA XML for Armv9.

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

This supplement does not contain any detailed instruction descriptions, pseudocode, XML, or System register descriptions. This information is provided in a separate format. Links to this information are included throughout the supplement.

This supplement is organized into parts:

- SVE Application level programmers' model

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

- SVE System level programmer's model

  Describes how the *PE* at a system level is altered by the implementation of *SVE*.

- SVE Memory Model

  Describes the extensions made for *SVE* to the Arm memory model.

- SVE instruction set

  Describes the extensions made for *SVE* to the Arm instruction set.

- SVE Debug

  Describes how the Arm v8Debug exception model has been extended for *SVE*.

- SVE Performance Monitor Usage

  Lists interesting combinations of *SVE* events

- SVE instruction categories

  Lists *SVE* instructions grouped by the type of instruction.

- Glossary

  Defines terms used in this document that have a specialized meaning.

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

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

`monospace`

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

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

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

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

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

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

Numbers

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

Pseudocode descriptions

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

Asterisks in instruction mnemonics

Some behavior descriptions in this manual apply to a group of similar instructions that start with the same characters. In these situations, an * might be inserted at the end of a series of characters as a wildcard. For example, a reference to LDNF1* means that the behavior applies to all of the following instructions:

- LDNF1B
- LDNF1D
- LDNF1H
- LDNF1SB
- LDNF1SH
- LDNF1SW
- LDNF1W

The * is inclusive, so ADR* includes both the ADR instruction and the ADRP instruction.

In order to make sure it is convenient to find instructions grouped in this way, this manual uses the * at the end of the instruction name only and never at the start or in the middle of an instruction name.

Assembler syntax descriptions

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

Rules-based writing

Rules-based writing differs significantly from the traditional style used in Arm technical documents. Rules-based writing has short statements with unique identifiers.

Requirements are referred to as rules and other information is referred to as information statements. Rules-based architectural documents describe requirements of the architecture and information statements provide additional information. Structured rules also follow a specific structure and specify some keyword terms. The following is an example of a rule, followed by an information statement.

R_{WPBT} If a document has structured rules, by default all rules statements have a specific structure.

I_{NBPP} For architectural specifications the writing style for outcomes is deliberately different from the usual Arm style.

All rules and information statements have unique IDs. IDs start with a designator, followed by a unique string of 4 or 5 SMALL CAPITAL consonants. If the designator is an ‘R’ it is a rule. If the designator is an ‘I’, it is an information statement.

All rules have a specific structure. Information statements may take any structure.

Rules generally start with any conditions that make the rule applicable. These conditions have a limited set of introductory phrases:

- Conditions that begin with if are used to make rules conditional on a state and tend to last for a while.
- Conditions that begin with when are used to make rules conditional on an event happening.
- Conditions that begin with for are used to make a rule apply to a part of the system.

There are three forms of the if-statements:

- If indicates the condition is sufficient to cause the action but might not be necessary. “If X then Y” means the same as “If X then Y happens, but if not X then Y might still happen”.
- Only if indicates the condition is necessary but might not be sufficient. “Only if X then Y” means the same as “If X then Y might happen, if not X then Y cannot happen”.
- If and only if indicates the condition is both necessary and sufficient. “If and only if X then Y” means the same as “If X then Y happens, if not X then Y cannot happen”.

There are also three forms of the when-statements:

- When indicates the condition is sufficient to cause the action but might not be necessary.
- Only when indicates the condition is necessary but might not be sufficient.
- When and only when indicates the condition is both necessary and sufficient.

There may be many preconditions in a rule.

The next part of a rule is either an actor or a subject. When a specific action by a specific entity is defined, the rule will be written in active voice and will have an actor. If the action is performed by an IMPLEMENTATION DEFINED entity, then the rule will be written in passive voice and the rule will have a subject, which is something that is acted on by the action in the rule statement. The action to be carried out follows the actor or subject and is required. The object(s) of the action, followed by the outcome of the rule, are both optional and may be present after the action.

Additional reading

This section lists publications by Arm and by third parties.

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

Arm publications

- *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile* (ARM DDI 0487).
- *Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile* (ARM DDI 0608).
- *System Register XML for Armv8*.
- *A64 ISA XML for Armv8*.
- *System Register XML for Armv9*.
- *A64 ISA XML for Armv9*.

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 (Arm® Architecture Reference Manual Supplement, The Scalable Vector Extension).
- The number (DDI 0584 B.a).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Progressive Terminology Commitment

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

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

Chapter 1
Introduction

1.1 About the Scalable Vector Extension

$\mathrm{I}_{\mathrm{TKDFM}}$ The *Scalable Vector Extension* (SVE) includes the following functionality:

- Configurable vector length, from 128 bits up to 2048 bits.
- Predication and the required predicate registers.
- Instructions that operate on variable size vectors and predicates.
- Gather-load and scatter-store.
- Support for software-managed speculative vectorization.
- System registers and fields to configure the *SVE* vector length and traps.
- Minor additions to the configuration and identification registers.

$\mathrm{I}_{\mathrm{VKNHX}}$ *SVE* complements the AArch64 *Advanced SIMD and floating-point* functionality. *SVE* does not replace the AArch64 *Advanced SIMD and floating-point* functionality.

$\mathrm{I}_{\mathrm{LGLPZ}}$ *Scalable Vector Extension* version two (SVE2) is a superset of *SVE* that incorporates functionality similar to Advanced SIMD, as well as other enhancements. In this document, unless stated otherwise, when *SVE* is used, the behavior also applies to *SVE2*.

$\mathrm{R}_{\mathrm{JLGXX}}$ *SVE* is supported in AArch64 state only.

$\mathrm{I}_{\mathrm{FMGMH}}$ The *Scalable Vector Extension* (SVE) is identified as FEAT_SVE.

$\mathrm{I}_{\mathrm{GYSYC}}$ The *Scalable Vector Extension version 2* (SVE2) is identified as FEAT_SVE2.

1.1.1 Features that affect SVE

R$_{XZBNC}$ If *SVE* is implemented, all of the following features are also implemented:

- FEAT_FCMA.
- FEAT_FP16.

R$_{YKCSW}$ The following list summarizes whether generic architectural features that affect *SVE* are OPTIONAL or mandatory:

- FEAT_BF16 BFloat16 instructions are OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations.
- FEAT_I8MM matrix multiplication instructions are OPTIONAL in Armv8.2 implementations and mandatory in Armv8.6 implementations.

1.1.2 Features within SVE

I$_{HXYLC}$

- FEAT_F32MM matrix multiplication instructions are OPTIONAL for *SVE* in Armv8.2.
- FEAT_F64MM matrix multiplication instructions are OPTIONAL for *SVE* in Armv8.2.

I$_{NZHVT}$ The following list summarizes the OPTIONAL *SVE2* features:

- FEAT_SVE_AES Scalable Vector AES instructions.
- FEAT_SVE_BitPerm Scalable Vector Bit Permute instructions.
- FEAT_SVE_PMULL128 Scalable Vector PMULL (128-bit result) instructions.
- FEAT_SVE_SHA3 Scalable Vector RAX1 instruction.
- FEAT_SVE_SM4 Scalable Vector SM4 instructions.

1.2 Register disambiguation

I$_{LJSKW}$ In some sections of this manual, registers are referred to by a generic name because the description applies to multiple *Exception levels*, and therefore at a particular *Exception level* the register names need to take the appropriate *Exception level* suffix, _EL0, _EL1, _EL2, or _EL3. The following table disambiguates the generic names of some *System registers* by *Exception level*:

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

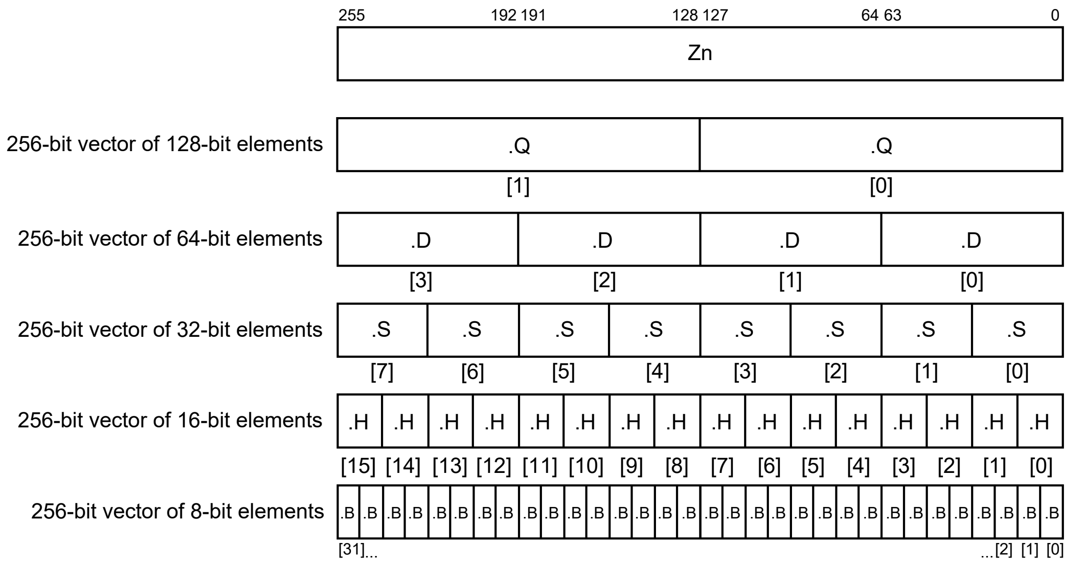
Chapter 2
SVE Application level programmers' model

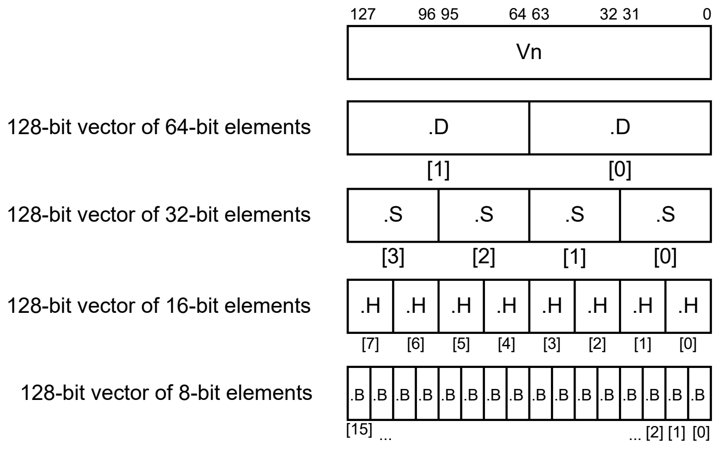
2.1 SVE-specific registers

2.1.1 SVE Vector registers

$\mathrm{R_{FBGSJ}}$	*SVE* has 32 scalable vector registers named Z0-Z31.
$\mathrm{R_{WJNYD}}$	All *SVE* scalable vector registers are the same size.
$\mathrm{R_{KCWQB}}$	The size of an *SVE* scalable vector register is an IMPLEMENTATION DEFINED multiple of 128 bits.
$\mathrm{R_{KJSDQ}}$	The maximum size of an *SVE* scalable vector register is 2048 bits.
$\mathrm{R_{RXPHX}}$	The minimum size of an *SVE* scalable vector register is 128 bits.
$\mathrm{I_{GKWYJ}}$	Unless stated otherwise in an instruction description, *SVE* instructions treat an *SVE* scalable vector register as containing one or more vector elements that are equal in size.
$\mathrm{I_{CDKJQ}}$	Unless stated otherwise in an instruction description, vector elements can be processed in parallel by *SVE* instructions.
$\mathrm{R_{KHDBN}}$	When an *SVE* scalable vector register is divided into vector elements by an instruction, the size of the vector elements is encoded in the opcode of the instruction. The size of the vector elements is 8, 16, 32, 64, or 128 bits.
$\mathrm{R_{CJZLM}}$	When the order of operations performed by an *SVE* instruction on vector or predicate elements has observable significance, elements are processed in increasing element number order.

I_{DBZRX} The layouts of an *SVE* 256-bit vector register and a SIMD&FP vector in AArch64 state are:


255
192 191
128 127
64 63
0
Zn
256-bit vector of 128-bit elements
.Q [1]
.Q [0]
256-bit vector of 64-bit elements
.D [3] .D [2] .D [1] .D [0]
256-bit vector of 32-bit elements
.S [7] .S [6] .S [5] .S [4] .S [3] .S [2] .S [1] .S [0]
256-bit vector of 16-bit elements
.H [15] .H [14] .H [13] .H [12] .H [11] .H [10] .H [9] .H [8] .H [7] .H [6] .H [5] .H [4] .H [3] .H [2] .H [1] .H [0]
256-bit vector of 8-bit elements
.B
[31]...
...[2] [1] [0]


127
96 95
64 63
32 31
0
Vn
128-bit vector of 64-bit elements
.D [1] .D [0]
128-bit vector of 32-bit elements
.S [3] .S [2] .S [1] .S [0]
128-bit vector of 16-bit elements
.H [7] .H [6] .H [5] .H [4] .H [3] .H [2] .H [1] .H [0]
128-bit vector of 8-bit elements
.B
[15] ...
... [2] [1] [0]

R_{YDXCP} Bits[127:0] of each of the *SVE* scalable vector registers, Z0-Z31, hold the correspondingly numbered AArch64 SIMD&FP register, V0-V31.

R_{WKYLB} When the accessible *SVE* vector length at the current *Exception level* is greater than 128 bits, any AArch64 instruction that writes to V0-V31 sets all the accessible bits above bit [127] of the corresponding *SVE* scalable vector register to zero.

See also:

- 2.1.2 *SVE predicate registers*.
- 3.2 *Configurable vector length*.
- 4.2 *Atomicity*.

2.1.2 SVE predicate registers

R_{DCWFB} *SVE* has 16 scalable predicate registers named P0-P15.

R_{NLGZS} Each *SVE* predicate register holds one bit for each byte of a vector register.

R_{NKRJV} The size of an *SVE* predicate register is an IMPLEMENTATION DEFINED multiple of 16 bits.

R_{MFPXG} The maximum size of an *SVE* predicate register is 256 bits.

R_{BBTXX} The minimum size of an *SVE* predicate register is 16 bits.

R_{XVRKX} Unless stated otherwise in the instruction description, *SVE* instructions treat an *SVE* predicate register as containing one or more predicate elements of equal size.

R_{XMPLM} Each predicate register can be subdivided into a number of 1, 2, 4, or 8-bit elements.

R_{NSXCV} Each predicate element in a predicate register corresponds to a vector element.

R_{XCZQR} When a predicate register is divided into predicate elements by an instruction, the size of the predicate elements is encoded in the opcode of the instruction.

R_{DNMFH} If the lowest-numbered bit of a predicate element is 1, the value of the predicate element is TRUE.

R_{HRPMD} If the lowest-numbered bit of a predicate element is 0, the value of the predicate element is FALSE.

R_{HBMLS} For all *SVE* instructions, if all of the following are true, all bits except the lowest-numbered bit of each predicate element are ignored on reads:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

R_{LTGQC} For all *SVE* instructions, if all of the following are true, all bits except the lowest-numbered bit of each predicate element are set to zero on writes:

- The instructions are not used to move and permute predicate elements.
- The instructions are not predicate logical operations.

See also:

- 2.3 *Predication*.
- 5.2.6.3 *Predicate logical operations* This section contains a list of instructions used to perform bitwise logical operations on predicate registers that operate on all bits of the register.
- 5.2.7.3 *Predicate permute* This section contains a list of instructions used to used to move and permute predicate elements.

2.1.3 First Fault Register, FFR

R_{TRLWH} *SVE* has a dedicated First Fault Register named FFR.

I_{XPLQW} The FFR captures the cumulative fault status of a sequence of *SVE* First-fault and Non-fault vector load instructions.

R_{CPQQN} The FFR and the predicate registers have the same size and format.

I_{PBWPM} The FFR is a Special-purpose register.

R_{CGHCK} All accessible bits in the FFR are initialized to 1 by using the SETFFR instruction.

R_{WZJVT} Bits in the FFR are indirectly set to 0 as a result of a suppressed access or fault generated in response to an *Active element* of an *SVE* First-fault or Non-fault vector load.

R_{BZLJG} Bits in the FFR are never set to 1 as a result of a vector load instruction.

I_{XLZQY} After a sequence of one or more *SVE* First-fault or Non-fault loads that follow a SETFFR instruction, the FFR contains a sequence of zero or more TRUE elements, followed by zero or more FALSE elements.

I_{TQMTV} The TRUE elements in the FFR indicate the shortest sequence of consecutive elements that could contain valid data loaded from memory.

RGHFRQ The only instructions that directly read the FFR are:

- RDFFR.
- RDFFRS.

R_LHBRN The only instructions that directly write the FFR are:

- WRFFR.
- SETFFR.

R_XXMMP All direct and indirect reads and writes to the FFR occur in program order relative to other instructions, without explicit synchronization.

See also:

- 2.1.2 *SVE predicate registers*.
- 3.1.1 *Synchronous memory faults*.

2.1.4 SVE writes to scalar registers

I_ZDLGD Certain *SVE* instructions generate a scalar result that is written to an AArch64 general-purpose register or to element[0] of a vector register.

R_HNVTM When an *SVE* instruction generates a scalar result of width N bits, the instruction places the result in bits [N-1:0] of the destination register.

R_QCLSH When an instruction generates a scalar result of width N bits, and N is less than the maximum accessible destination register width RW, the instruction sets to zero bits [RW-1:N] of the destination register.

See also:

- *Registers in AArch64 Execution state* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*

2.2 SVE floating-point support

I_{XKSFG} Unless otherwise specified, *SVE* floating-point instructions follow the normal Armv8 floating-point behaviors in the *Advanced SIMD and floating-point support* section of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

2.2.1 Half-precision floating-point support

I_{GQVFD} Except as specified in this section, *SVE* half-precision floating-point instructions honor the descriptions in the *Half-precision floating-point format* section of the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

R_{RWYCB} *SVE* half-precision floating-point instructions ignore the value of the FPCR.AHP bit and behave as if the bit has an *Effective value* of 0.

I_{DLWTQ} The *SVE* half-precision floating-point instructions support only the IEEE 754-2008 half-precision format.

2.2.2 Single-precision floating-point support

I_{YRYHT} For more information on single-precision floating-point support, see the *Single-precision floating-point format* section in the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

2.2.3 Double-precision floating-point support

I_{QQLZQ} For more information on double-precision floating-point support, see the *Double-precision floating-point format* section in the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

2.2.4 BFloat16 floating-point support

R_{QBLKC} The BFloat16 instructions are only supported if ID_AA64ZFR0_EL1.BF16 is not 0.

I_{CPMZW} For more information on BFloat16 support, see the *BFloat16 floating-point format* section in the *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*.

2.3 Predication

I_{VKHDR} If an instruction supports predication, it is known as a predicated instruction.

I_{FNFRN} The predicate operand that is used to determine the *Active elements* of a predicated instruction is known as the *Governing predicate*.

I_{SVYXB} An instruction that does not have a *Governing predicate* operand and implicitly treats all other vector and predicate elements as Active is known as an unpredicated instruction.

I_{KNKBN} Many predicated instructions can only use P0-P7 as the *Governing predicate*.

R_{LZVFJ} When a *Governing predicate* element is TRUE, the corresponding element in other vector or predicate operands is an *Active element*.

R_{CNFLG} When a *Governing predicate* element is FALSE, the corresponding element in other vector or predicate operands is an *Inactive element*.

R_{CBYJH} Predicated instructions process *Active elements*.

R_{LDXSF} Predicated instructions do not process *Inactive elements*.

R_{GJLPZ} Unpredicated instructions process all elements in their vector or predicate operands.

R_{WLQBD} When a predicated instruction writes to a vector destination register or a predicate destination register, one of the following happens:

- The *Inactive elements* in the destination register are set to zero.
- The *Inactive elements* in the destination register retain their previous value.

I_{QBHRN} Zeroing predication is performed when the *Inactive elements* in the destination register are set to zero.

I_{YPYRF} Merging predication is performed when *Inactive elements* in the destination register retain their previous value.

2.4 Process state, PSTATE N, Z, C and V Condition flags

I_{WYXLS} Process state, or *PSTATE*, is an abstraction of process state information. This section describes the *SVE*-specific use of *PSTATE*.

I_{YZYCQ} PSTATE N, Z, C and V Condition flags can be updated by any of the following:

- An *SVE* instruction that generates a predicate result and updates the PSTATE N, Z, C and V Condition flags based on the value of the result.
- An *SVE* instruction that updates the PSTATE N, Z, C and V Condition flags based on the value in its predicate source register or FFR:
  - PTEST
  - RDFFRS (predicated)
- An *SVE* instruction that updates the PSTATE N, Z, C and V Condition flags based on the values in its general-purpose source registers:
  - CTERMEQ
  - CTERMNE

R_{TPXTF} When setting the PSTATE N, Z, C and V Condition flags for *SVE* predicated flag-setting instructions, the instruction's *Governing predicate* determines which predicate elements are considered Active.

R_{QJBRW} When setting the PSTATE N, Z, C and V Condition flags for *SVE* unpredicated flag-setting instructions, all predicate elements are considered Active.

R_{ZMRXC} Unless otherwise specified in an instruction description, the *SVE* flag-setting instructions update the PSTATE N, Z, C and V Condition flags as follows:

Flag	*SVE* Name	*SVE* interpretation
N	First	Set to 1 if the *First active element* was TRUE, otherwise cleared to 0.
Z	None	Cleared to 0 if any *Active element* was TRUE, otherwise set to 1.
C	Not last	Cleared to 0 if the *Last active element* was TRUE, otherwise set to 1.
V	-	Cleared to 0.

I_{KSXVR} For convenience, the *SVE* assembler syntax defines an alternative set of *SVE* condition code aliases for use with AArch64 conditional instructions, as follows:

Condition test	AArch64 name	*SVE* alias	*SVE* interpretation
Z == 1	EQ	NONE	All *Active elements* were FALSE or there were no *Active elements*
Z == 0	NE	ANY	An *Active element* was TRUE.
C == 1	HS/CS	NLAST	The *Last active element* was FALSE or there were no *Active elements*.
C == 0	LO/CC	LAST	The *Last active element* was TRUE.
N == 1	MI	FIRST	The *First active element* was TRUE.
N == 0	PL	NFRST	The *First active element* was FALSE or there were no *Active elements*.

Condition test	AArch64 name	*SVE* alias	*SVE* interpretation
C == 1 && Z == 0	HI	PMORE	An *Active element* was TRUE, but the *Last active element* was FALSE.
C == 0 || Z == 1	LS	PLAST	The *Last active element* was TRUE, or all *Active elements* were FALSE, or there were no *Active elements*.
V == 1	VS	-	CTERM comparison failed, but end of partition reached.
V == 0	VC	-	CTERM comparison succeeded, or end of partition not reached.
N == V	GE	TCONT	CTERM termination condition not detected.
N != V	LT	TSTOP	CTERM termination condition detected.

See also:

- 2.3 *Predication*

2.5 Data independent timing of SVE and SVE2 data-processing instructions

I_{FDNVB} If *SVE2* is not implemented, the data independent timing control introduced by FEAT_DIT does not affect the timing properties of *SVE* instructions. However, if *SVE2* is implemented, the addition of cryptographic instructions in *SVE2* requires that when PSTATE.DIT is 1, the timing properties of certain *SVE* and *SVE2* instructions are affected.

R_{VPSLG} If *SVE2* is implemented and PSTATE.DIT is 1, the data independent timing of the following subset of *SVE* and *SVE2* instructions is affected in all of the following ways:

- For unpredicated *SVE* and *SVE2* instructions, the instruction timing is independent of all of the following:
  - The data values supplied in any of its operand registers.
  - The values of the NZCV flags.
- For predicated *SVE* and *SVE2* instructions, the instruction timing is independent of all of the following:
  - The data values supplied in any of its operand registers when its *Governing predicate* register contains the same value for each execution.
  - The values of the NZCV flags.
- For predicated *SVE* and *SVE2* instructions, the architecture does not mandate that the instruction timing is independent of the value of the *Governing predicate*.
- For the *SVE* `SEL` instruction, the instruction timing is independent of all of the following:
  - The data values supplied in any of its operand registers.
  - The values of the NZCV flags

I_{KXCSP} When using the predicated instructions, it is the programmer's responsibility to use a *Governing predicate* that does not reflect the values of the data being operated on.

R_{FQVYY} The *Operational information* section of an *SVE* instruction description indicates whether or not that instruction honors the PSTATE.DIT control. If the *Operational information* section of an *SVE* instruction description does not mention PSTATE.DIT or if the section does not exist, then the instruction timing is not affected by PSTATE.DIT.

Chapter 3
SVE System level programmers' model

3.1 Exception model

I QNHYT *SVE* adds hierarchical trap and enable controls at EL3, EL2, and EL1:

- CPTR_EL3.EZ.
- CPTR_EL2.TZ, when HCR_EL2.E2H == 0.
- CPTR_EL2.ZEN, when HCR_EL2.E2H == 1.
- CPACR_EL1.ZEN.

I FYMMP *SVE* defines the `0b011001` exception class value in ESR_ELx.EC. The `0b011001` exception class value is for exceptions that are due to attempted execution of *SVE* instructions and MRS/MSR accesses to the ZCR_ELx registers that are trapped by CPACR_EL1, CPTR_EL2 or CPTR_EL3.

R VQCKK Predicated *SVE* floating-point instructions only generate floating-point exceptions in response to floating-point operations performed on *Active elements*.

R RWVTR When a MOVPRFX instruction pairs legally with another instruction and the execution of the pair generates a synchronous exception, the return address that is stored in ELR_ELx is one of the following:

- When the MOVPRFX instruction did not cause a change to the architectural state, the address of the MOVPRFX instruction is stored.
- When the MOVPRFX instruction caused a change to the architectural state, the address of the prefixed instruction is stored.

R XRWVD When a MOVPRFX instruction pairs legally with another instruction and the execution of the pair causes entry to *Debug state*, the return address that is stored in DLR_EL0 is one of the following:

- When the MOVPRFX instruction did not cause a change to the architectural state, the address of the MOVPRFX instruction is stored.
- When the MOVPRFX instruction caused a change to the architectural state, the address of the prefixed instruction is stored.

R_{TPRKM} When a MOVPRFX instruction pairs illegally with another instruction and execution of the pair generates a synchronous exception, the return address recorded in ELR_ELx is a CONSTRAINED UNPREDICTABLE choice one of the following:

- The address of the MOVPRFX instruction.
- The address of the prefixed instruction.

R_{JVNGC} When a MOVPRFX instruction pairs illegally with another instruction and execution of the pair causes entry to *Debug state*, the return address recorded in DLR_EL0 is a CONSTRAINED UNPREDICTABLE choice one of the following:

- The address of the MOVPRFX instruction.
- The address of the prefixed instruction.

R_{CRRPM} When a 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 address of the prefixed instruction is recorded in the appropriate FAR_ELx or HPFAR_EL2 register and the address of the MOVPRFX instruction is recorded in the appropriate ELR_ELx register.

R_{ZJYDX} When a prefixed instruction generates an Instruction Abort due to an MMU fault or synchronous *External abort* and the MOVPRFX also generates an Instruction Abort, then the address of the MOVPRFX instruction is recorded in the appropriate FAR_ELx or HPFAR_EL2 register and the appropriate ELR_ELx register.

See also:

- ESR_EL1
- ESR_EL2
- ESR_EL3
- ELR_EL1
- ELR_EL2
- ELR_EL3
- FAR_EL1
- FAR_EL2
- FAR_EL3
- MOVPRFX (predicated)
- MOVPRFX (unpredicated)
- ZCR_EL1
- ZCR_EL2
- ZCR_EL3
- 5.2.7.5 *Move prefix*

3.1.1 Synchronous memory faults

I_{YLCFS} In this section, the term *Memory fault* refers to the detection of an erroneous condition or debug event as a result of performing a data memory access for an *SVE* load or store instruction.

R_{SKNTR} When an *SVE* load or store instruction results in a data memory access, the detection of any of the following conditions is considered to be a *Memory fault*:

- MMU fault.
- Alignment fault, excluding the *SP* alignment fault.
- Synchronous External abort, including synchronous parity error or ECC error.
- Watchpoint debug event.

For more details see *VMSAv8-64 memory aborts* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.

I_{LDTYT} The detection or generation of a *Memory fault* by an *SVE* load or store instruction may or may not cause a synchronous Data Abort or Watchpoint exception to be taken.

R_{LVCNH} Unless otherwise specified in this section, *SVE* vector load and store instructions that detect a *Memory fault* cause a Data Abort or Watchpoint exception to be taken, as described in 3.1.1.1 *Data Abort and Watchpoint exceptions*.

R_{FPVTT} A *Memory fault* detected for a memory location that can only be accessed by an *Inactive element* of a predicated *SVE* vector load or store instruction is ignored and does not cause a Data Abort or Watchpoint exception to be taken by that instruction.

3.1.1.1 Data Abort and Watchpoint exceptions

R_{KJPTS} Unless otherwise specified in this section, a Data Abort and Watchpoint exception caused by an *SVE* load or store instruction follows the behaviors described in sections *Effect of Data Aborts and Watchpoints*, *Exception entry*, and *Synchronous exception types, routing and priorities* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.

I_{DGSNC} *SVE* load and store instructions can generate a sequence of single-copy atomic memory accesses that might not be completed due to a *Memory fault* causing a Data Abort or Watchpoint exception to be taken during the memory access sequence.

R_{ZKBRX} When the execution of an *SVE* load or store instruction detects multiple *Memory faults* caused by different single-copy atomic memory accesses, the *Memory faults* are not prioritized by the architecture.

I_{MSVYK} When an *SVE* load or store instruction that has not been architecturally executed is restarted after an exception return, any memory locations that it accessed before taking the exception might be accessed again. Therefore, *SVE* load or store instructions might perform multiple accesses to memory locations that do not cause a *Memory fault* but which are sensitive to the number of accesses, or have been modified between the accesses.

R_{ZXNXT} When execution of an *SVE* load instruction causes a Data Abort or Watchpoint exception to be taken and the destination is not a vector register that is also used as a base or index register by the instruction, then all elements of the destination register become UNKNOWN.

R_{SNJQR} When execution of an *SVE* load instruction causes a Data Abort or Watchpoint exception to be taken and the destination is a vector register that is also used as a base or index register by the instruction, then all elements of the destination vector register are restored to their original value prior to execution of the load instruction.

R_{DWYCY} When execution of an *SVE* predicated vector store instruction causes a Data Abort or Watchpoint exception to be taken, one or more of the following occurs:

- Memory locations that are associated with *Active elements* and which do not generate a *Memory fault* become UNKNOWN.
- Memory locations that are associated with *Active elements* and which generate a *Memory fault* are unchanged.
- Memory locations that are only associated with *Inactive elements* are unchanged.

3.1.1.2 First-fault and Non-fault loads

I_{JZBGW} When a memory access performed for the *First active element* of an *SVE* First-fault vector load instruction detects a *Memory fault*, this causes a synchronous exception to be taken as described in 3.1.1.1 *Data Abort and Watchpoint exceptions*.

I_{DXBNG} When a memory access performed for the *First active element* of an *SVE* First-fault vector load instruction does not detect a *Memory fault*, the other elements are handled in the same way as the elements of an *SVE* Non-fault vector load instruction.

R_{JKGYJ} A Data Abort or Watchpoint exception is not generated when a *Memory fault* is detected by a memory access performed for any of the following elements:

- Any *Active element* of an *SVE* Non-fault vector load.
- Any *Active element* of an *SVE* First-fault vector load except for the *First active element*.

R_{MNKNV} The *PE* can choose to suppress a memory access performed for any of the following elements:

- Any *Active element* of an *SVE* Non-fault vector load.
- Any *Active element* of an *SVE* First-fault vector load except for the *First active element*.

R_{YFTRN} When a memory access performed for any of the following elements detects a *Memory fault* or is suppressed for any other reason, the FFR predicate elements starting from that element number, up to and including the highest-numbered element, are set to FALSE:

- Any *Active element* of an *SVE* Non-fault vector load.
- Any *Active element* of an *SVE* First-fault vector load except for the *First active element*.

I_{BMQVT} An FFR predicate element is never set to TRUE by an *SVE* vector load, therefore the fault indications are cumulative.

R_{NGFTJ} After an *SVE* Non-fault vector load or First-fault vector load is executed, each destination vector element contains one of the values listed in the following table:

Corresponding FFR element	Vector element status	Content of destination vector element
FALSE	Active	Each byte of the element contains an independently CONSTRAINED UNPREDICTABLE choice of one of the following: • 0. • The previous value of that byte in the destination vector register. • If and only if all of the following apply, the value read from memory: – The memory access for that byte was not an access to any type of Device memory. – The memory access for that byte does not return information that cannot be accessed at the current or a lower level of privilege.
FALSE	Inactive	A CONSTRAINED UNPREDICTABLE choice of: • 0. • The previous value of that vector element.
TRUE	Active	The value read from memory.
TRUE	Inactive	0.

R_{WCHSR} In the previous table, watchpoints are not a mechanism for preventing access to memory.

See also:

- *VMSAv8-64 memory aborts* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.
- *Effect of Data Aborts and Watchpoints* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*
- *Exception entry* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.
- *Synchronous exception types, routing and priorities* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.

3.1.2 Asynchronous exceptions

$\mathrm{R}_{\mathrm{TFLTX}}$ Permitting *SVE* instructions to be interrupted by asynchronous exceptions is IMPLEMENTATION DEFINED.

$\mathrm{R}_{\mathrm{WFMZK}}$ When returning from an asynchronous exception, an interrupted *SVE* instruction is restarted and cannot resume at the point the instruction was interrupted.

$\mathrm{I}_{\mathrm{QRQJP}}$ For Data Aborts taken asynchronously, refer to *Effect of Data Aborts and Watchpoints* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*.

3.2 Configurable vector length

I_{NWYBP} Privileged *Exception levels* can use the ZCR_ELx.LEN *System register* fields to constrain the vector length at that *Exception level* and at less privileged *Exception levels*.

R_{PVRSF} An implementation allows the vector length to be constrained to any power of two that is less than the maximum implemented vector length.

R_{RYQYY} An implementation is permitted to allow the vector length to be constrained to multiples of 128 that are not a power of two. It is IMPLEMENTATION DEFINED which of the permitted multiples of 128 are supported.

I_{CPZLW} The following table shows the *SVE* configurable vector lengths:

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

R_{MMCTJ} When the values in ZCR_ELx.LEN configure an unsupported vector length, the implementation is required to select the largest supported vector length that is less than the configured vector length. This does not alter the values in ZCR_ELx.LEN.

R_{PXZTM} If executing at an *Exception level* that is constrained to use a vector length that is less than the maximum implemented vector length, the bits beyond the constrained length of the vector registers, predicate registers, or FFR are inaccessible.

R_{NLYDK} When taking an exception from an *Exception level* that is more constrained to a target *Exception level* that is less constrained, the previously inaccessible bits that become accessible have one of the following:

- A value of zero.
- The value that they had before executing at the more constrained vector length.

The choice between these options is IMPLEMENTATION DEFINED and can vary dynamically.

R_{TQVGX} When the size of the maximum vector length is increased by writing a larger value to ZCR_ELx.LEN, the previously inaccessible bits that become accessible have one of the following:

- A value of zero.
- The value that they had before executing at the more constrained vector size.

The choice between these options is IMPLEMENTATION DEFINED and can vary dynamically.

R_{DMBPN} If both floating-point and *SVE* instructions are disabled, trapped, or not available at all *Exception levels* below the target *Exception level*, for the current *Security state*, the accessible *SVE* register state at the target *Exception level* is preserved.

R_{KXKNK} If *SVE* instructions are disabled or trapped at ELx, or not available because that *Exception level* is in AArch32 state, then for all purposes other than a direct read, the ZCR_ELx.LEN field has an Effective value of 0, which implies an *SVE* vector length of 128 bits.

See also:

- ZCR_EL1
- ZCR_EL2
- ZCR_EL3

Chapter 4
SVE Memory Model

4.1 SVE memory model

I_{CZFSY} *SVE* predicated memory operations have a vector element size and a memory element access size. The vector element size specifies the data that is read from and written to the vector. The memory element access size specifies the amount of data that is read from and written to the memory.

I_{TJQJF} The vector element size and the memory element access size do not need to have the same value.

I_{LGGHH} For each memory element, there is an associated element address.

4.2 Atomicity

R_{HGFJZ} Atomicity rules for *SIMD* load and store instructions apply to *SVE* load and store instructions.

I_{XPQKZ} Additional rules apply to the atomicity of memory accesses performed by *SVE* load and store instructions.

R_{NNHQB} *SVE* predicated load and store instructions are performed as a sequence of memory element accesses.

R_{HLHZV} When an *SVE* predicated load or store instruction uses an element address that is aligned to the specified memory element access size, the related element memory access is performed as a single-copy atomic access.

R_{JJFKG} *SVE* unpredicated load and store instructions are performed as a sequence of byte accesses.

R_{SPYMV} *SVE* unpredicated load and store instructions do not guarantee that any access larger than a byte will be performed as a single-copy atomic access.

See also:

- *Atomicity* in the *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile*

4.3 Alignment support

R~PQQGP~ Alignment rules for *SIMD* load and store instructions apply to *SVE* load and store instructions.

I~KHHZF~ Additional rules apply to the alignment of memory accesses performed by *SVE* load and store instructions.

R~ZTJZY~ For predicated *SVE* vector element and structure load or store instructions, alignment checks are based on the memory element access size, not on the vector element size.

R~XFVKZ~ For predicated *SVE* vector element and structure load or store instructions, *Inactive elements* cannot cause an *Alignment fault*.

R~SCXJP~ For unpredicated *SVE* vector register load or store instructions, the base address is checked for 16-byte alignment.

R~CWKLF~ For unpredicated *SVE* predicate register load or store instructions, the base address is checked for 2-byte alignment.

R~DLDVL~ If *SP* alignment checking is enabled in SCTLR_ELx at the current *Exception level* and an *SVE* predicated load or store instruction with any *Active elements* uses the current *SP* as the base address, then the *SP* register is checked for 16-byte alignment.

R~FNCJX~ If *SP* alignment checking is enabled in SCTLR_ELx at the current *Exception level* and an *SVE* predicated load or store instruction with no *Active elements* uses the current *SP* as the base address, then it is CONSTRAINED UNPREDICTABLE whether the *SP* register is checked for 16-byte alignment.

See also:

- *Alignment support* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*
- *Memory types and attributes* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*
- SCTLR_EL1
- SCTLR_EL2
- SCTLR_EL3
- *SP alignment checking* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*

4.4 Data endianness

R_{VDGQK} Rules on byte and element order of SIMD load and store instructions apply to *SVE* load and store instructions.

I_{RFQJP} Additional rules apply to the data endianness of memory accesses performed by *SVE* load and store instructions.

R_{CNKCL} For predicated *SVE* vector element and structure load and store instructions, an endianness conversion is performed using the memory element access size. The size of the vector element is not used in endianness conversion.

R_{QHXPL} For unpredicated *SVE* vector register load and store instructions, the vector byte elements are transferred in increasing element number order without any endianness conversion.

R_{RWLXY} For unpredicated *SVE* predicate register load and store instructions, each 8 bits from the predicate are transferred as a byte in increasing element number order without any endianness conversion.

R_{YGSBQ} When an *SVE* load instruction is executed, endianness conversion occurs before any sign-extension or zero-extension into a vector element.

R_{KYRQW} When an *SVE* store instruction is executed, endianness conversion occurs after any truncation from the vector element to the memory element access size.

See also:

- *Data endianness* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*

4.5 Memory ordering

$\mathrm{I}_{\mathrm{CTNGV}}$ The Armv8 memory model is relaxed for reads and writes generated by *SVE* load and store instructions.

$\mathrm{R}_{\mathrm{QLJPC}}$ When two reads generated by *SVE* vector load instructions have an address dependency, the dependency does not contribute to the dependency-ordered-before relation.

$\mathrm{R}_{\mathrm{YMBMZ}}$ When a pair of reads access the same location, and at least one of the reads is generated by an *SVE* load instruction, for a given observer, the pair of reads is not required to satisfy the internal visibility requirement.

$\mathrm{R}_{\mathrm{CJHWV}}$ When a single *SVE* vector store instruction generates multiple writes to the same location, the instruction ensures that these writes appear in the coherence order for that location, in order of increasing vector element number. No other ordering restrictions apply to memory effects generated by the same *SVE* store instruction.

$\mathrm{R}_{\mathrm{LVXTJ}}$ If a single *SVE* load instruction generates multiple reads, the order in which the reads for different elements and registers appear is not architecturally defined.

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

$\mathrm{I}_{\mathrm{CCWGN}}$ For any *SVE* load or store instruction that generates multiple single-copy atomic accesses to Normal or Device memory, there is no requirement for the memory system beyond the *PE* to be able to identify the single-copy atomic memory element access sizes.

See also:

- *Definition of the Armv8 memory model*, *Arm® Architecture Reference Manual*
- *Memory types and attributes*, *Arm® Architecture Reference Manual*

4.6 Device memory

R_{XLLJZ} All rules applying to *Device memory* accesses by *Advanced SIMD* and floating-point load and store instructions apply to *Device memory* access by *SVE* load and store instructions.

I_{YHWJT} Additional rules apply to *Device memory* access by *SVE* load and store instructions.

R_{NYMWH} When an *SVE* vector prefetch instruction is executed, any resulting memory read is guaranteed not to access *Device memory*.

R_{SBHLD} When an *SVE* Non-fault vector load is executed or for any element from a First-fault load except the *First active element*, the resulting memory reads will not access *Device memory*.

R_{TMVNR} When an *SVE* Non-fault vector load instruction is executed, an attempt by any *Active element* to access *Device memory* is suppressed and reported in the FFR.

R_{SFBKQ} When an *SVE* First-fault vector load instruction is executed, any memory read performed for the *First active element* can access *Device memory*.

R_{BHNQN} When an *SVE* First-fault vector load instruction is executed, an attempt by any *Active element* other than the *First active element* to access *Device memory* is suppressed and is reported in the FFR.

R_{VHCGD} Hardware speculation of data accesses performed to a *Device memory* location is not permitted unless stated otherwise.

R_{QHHHG} For reads, including hardware speculation, that are performed by an *SVE* unpredicated load instruction, all of the following are true:

- For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction, any byte in the window can be accessed by the instruction.
- All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by the instruction.

R_{SCGGF} For reads, including hardware speculation, that are performed by an *SVE* predicated load instruction that is not a non-temporal load, all of the following are true:

- For any 64-byte window aligned to 64 bytes containing at least 1 byte that is explicitly accessed by an *Active element* of the instruction, any byte in the window can be accessed by the instruction.
- All bytes accessed by the instruction will be in a 64-byte window aligned to 64 bytes that contains at least 1 byte that is explicitly accessed by an *Active element* of the instruction.

R_{JWYBD} For reads, including hardware speculation, that are performed by an *SVE* predicated non-temporal load instruction from memory locations with the Gathering attributes, all of the following are true:

- For any 128-byte window aligned to 128 bytes containing at least 1 byte that is explicitly accessed by an *Active element* of the instruction, any byte in the window can be accessed by the instruction.
- All bytes accessed by the instruction are in a 128-byte window aligned to 128 bytes that contains at least 1 byte that is explicitly accessed by an *Active element* of the instruction.

R_{QBLMZ} Any access to Device memory performed by an *SVE* load or store instruction is relaxed such that it might behave as if:

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

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

See also:

- 2.1.3 *First Fault Register, FFR*
- 3.1.1 *Synchronous memory faults*

4.7 CONSTRAINED UNPREDICTABLE memory accesses

I_{XZBLN} CONSTRAINED UNPREDICTABLE behaviors that are associated with memory accesses due to loads and stores also apply to *SVE* vector load and store instructions.

I_{XFHKS} The CONSTRAINED UNPREDICTABLE behaviors referred to in this section are defined in the *Crossing a page boundary with different memory types or Shareability attributes* and *Crossing a peripheral boundary with a Device access* sections of the the *Arm® Architecture Reference Manual, Armv8-A*. architecture profile.

R_{SPHTR} When an *SVE* unpredicated contiguous load or store instructions accesses an address range that crosses a boundary between memory types, *Shareability* attributes, or peripherals, the instruction has CONSTRAINED UNPREDICTABLE behaviors associated with the cross boundary memory access.

R_{WPBDN} When an *SVE* predicated contiguous load or store instruction performs memory accesses that are associated with *Active elements* on both sides of a boundary between different memory types, *Shareability* attributes, or peripherals, the instruction has CONSTRAINED UNPREDICTABLE behaviors associated with the cross boundary memory access.

R_{TGZYT} When an *SVE* predicated non-contiguous load or store instruction performs a memory access that is associated with an *Active element* that crosses a boundary between memory types, *Shareability* attributes, or peripherals, the instruction has CONSTRAINED UNPREDICTABLE behaviors associated with the cross boundary memory access.

R_{FQLTC} Memory addresses that are associated with *Inactive elements* cannot trigger CONSTRAINED UNPREDICTABLE behaviors.

R_{FDHZH} If *SVE* vector loads and stores trigger a CONSTRAINED UNPREDICTABLE behavior that then generates an alignment fault, the fault is handled the same as any other synchronous memory fault caused by an *SVE* load or store instruction.

See also:

- *Crossing a page boundary with different memory types*, *Arm® Architecture Reference Manual, Armv8-A*
- *Crossing a peripheral boundary with a Device access*, *Arm® Architecture Reference Manual, Armv8-A*
- 3.1.1 *Synchronous memory faults*

Chapter 5
SVE instruction set

5.1 SVE assembler language

The *SVE* assembler language extends the A64 assembler language.

I$_{RFMVH}$ The additions are:

- *SVE* adds vector register names `Z0`-`Z31` and predicate register names `P0`-`P15`.
- The number of elements in a vector or predicate register is not specified as part of a vector register shape qualifier. For example, `Z1.B` is used rather than `V1.16B`.
- An element size qualifier is not required for the *Governing predicate* (`Pg`) except where the element size cannot be inferred from the source and destination element sizes. However, when a predicate element size qualifier is provided it is accepted by an assembler and checked for consistency with the other operands.
- When appropriate, predicated instructions 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*:

  – `Pg/Z` - zeroing predication.
  – `Pg/M` - merging predication.

  Some instructions identify *Active elements* and *Inactive elements*, but do not write to a destination vector register. For these instructions, the *Governing predicate* operand is used with no zeroing or merging qualifier.
- Many *SVE* instructions have destructive instruction encodings. To avoid ambiguity, a constructive notation is used by the assembler language for these instructions. The destination register is repeated in the appropriate source operand position.

- *SVE* load/store addresses have a syntax that allows vector register operands within the address specified.
- A set of *SVE* aliases is defined for the AArch64 condition codes.

See also:

- *Structure of the A64 assembler language* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*
- 2.4 *Process state, PSTATE N, Z, C and V Condition flags*.

5.2 SVE ISA functional groups

I_{NSZQX} This section describes the functional groups for *SVE* instructions.

This section does not include *SVE2* functional groups. For information on *SVE2* functional groups, see 5.3 *SVE2 ISA functional groups*.

I_{KXTHC} *SVE* adds a set of instructions to the existing ARMv8-A A64 instruction set. 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.

See also:

- *The A64 instruction set* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*

5.2.1 Load, store, and prefetch instructions

I_{QZSSG} *SVE* vector load and store instructions transfer data in memory to or from elements of one or more vector or predicate transfer registers. *SVE* also includes vector prefetch instructions that provide read and write hints to the memory system. For *SVE* predicated load, store, and prefetch instructions, the memory element access size and type that is associated with each vector element is specified by a suffix to the instruction mnemonic, independently of the element size of the transfer registers. For example, LD1SH. The following table shows the supported instruction suffixes for *SVE* load, store, and prefetch instructions:

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

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

- For load instructions, each element access is sign-extended or zero-extended to fill the vector element, according to its memory element access size and type.
- For store instructions, each vector element is truncated to the memory element access size.

Where the vector element size and the memory element access size are the same, then these are referred to as packed data accesses. Signed access types are not supported for packed data accesses. Packed and unpacked access sizes and types relate to the vector element size of the transfer registers, as defined in the following table:

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

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

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

- *5.2.1.1 Predicated single vector contiguous element accesses*
- *5.2.1.4 Predicated replicating element loads*
- *5.2.1.2 Predicated multiple vector contiguous structure load/store*
- *5.2.1.3 Predicated non-contiguous element accesses*
- *5.2.1.4 Predicated replicating element loads*
- *5.2.1.5 Unpredicated vector register load/store*
- *5.2.1.6 Unpredicated predicate register load/store*

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

Prefetch instructions provide hints to hardware and do not change architectural state. Therefore, a *Governing predicate* for a prefetch instruction provides an additional hint which indicates the memory locations to be prefetched. Prefetch instructions require a prefetch operation specifier. *SVE* prefetch instructions support all of the prefetch operations except for the PLI prefetch operand types.

Load, store, and prefetch instructions that multiply a scalar index register or an index vector element by the memory element access size specify a shift type, followed by a shift amount in bits. The shift type can be one of LSL, SXTW, or UXTW. The shift amount is always Log2 of the memory element access size, in bytes. The shift amount defaults to zero when the memory element access size is a byte, and the shift size can be omitted. The shift type of LSL must be omitted if the shift amount is omitted.

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

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

5.2.1.1 Predicated single vector contiguous element accesses

I_{SKTWK} Predicated contiguous load and store instructions access memory locations starting from an address that is defined by a scalar base register plus either:

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

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

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

For this group of *SVE* instructions:

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

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

5.2.1.2 Predicated multiple vector contiguous structure load/store

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

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

For this group of *SVE* instructions:

- The immediate index value is a vector index, not an element index. The immediate index value is multiplied by the number of vector elements, irrespective of predication, and then multiplied by the memory element access size in bytes. The resulting offset is incremented following each element access by the memory element access size.
- The scalar index register value is multiplied by the memory element access size in bytes. Following each element access, the index value is incremented by one but the instruction does not update the scalar index register.
- Each predicate element applies to a single structure in memory, or equivalently to the same element number within each of the two, three, or four transferred vector registers.
- These instructions support packed data accesses only.

- When alignment checking is enabled for loads and stores, the base address must be aligned to the element access size.

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

5.2.1.3 Predicated non-contiguous element accesses

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

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

For this group of *SVE* instructions:

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

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

5.2.1.4 Predicated replicating element loads

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

- A scalar index register.
- An immediate byte offset.

This defaults to zero if omitted.

For this group of *SVE* instructions:

- The single element load and replicate instructions, LD1RB, LD1RD, LD1RH, LD1RSB, LD1RSH, LD1RSW, and LD1RW, load a single element value and replicate it into all *Active elements* of the destination vector.

These instructions support packed and unpacked data accesses. These instructions use an immediate byte offset that is a multiple of the memory element access size, in the range 0 to 63 times the memory element access size, inclusive.

- The 128-bit quadword load and replicate instructions, LD1RQB, LD1RQD, LD1RQH, and LD1RQW, load a predicated 128-bit quadword vector segment from contiguous element values and replicate that segment into all segments of the destination vector. These instructions support only packed data accesses. These instructions can use a scalar index register that is multiplied by the memory element access size, or an immediate byte offset that is a multiple of 16, in the range of -128 to 112, inclusive.
- When alignment checking is enabled, the base address must be aligned to the memory element access size.

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

5.2.1.5 Unpredicated vector register load/store

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

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

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

5.2.1.6 Unpredicated predicate register load/store

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

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

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

5.2.2 Vector move operations

5.2.2.1 Element move and broadcast

I_{JSBCJ} 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.

Mnemonic	Instruction	See
CPY	Copy signed integer immediate to vector elements	CPY (immediate)
	Copy general-purpose register to vector elements	CPY (scalar)
	Copy SIMD&FP scalar register to vector elements	CPY (SIMD&FP scalar)
DUP	Broadcast signed immediate to vector elements	DUP (immediate)
	Broadcast general-purpose register to vector elements	DUP (scalar)
FCPY	Copy 8-bit floating-point immediate to vector elements	FCPY
FDUP	Broadcast 8-bit floating-point immediate to vector elements	FDUP
FMOV	Move floating-point +0.0 to vector elements (unpredicated)	FMOV (zero, unpredicated)
	Move floating-point +0.0 to vector elements (predicated)	FMOV (zero, predicated)
	Move 8-bit floating-point immediate to vector elements (unpredicated)	FMOV (immediate, unpredicated)
	Move 8-bit floating-point immediate to vector element (predicated)	FMOV (immediate, predicated)
MOV	Move signed integer immediate to vector elements (unpredicated)	MOV (immediate, unpredicated)
	Move signed integer immediate to vector elements (predicated)	MOV (immediate, predicated)
	Move general-purpose register to vector elements (unpredicated)	MOV (scalar, unpredicated)
	Move general-purpose register to vector elements (predicated)	MOV (scalar, predicated)
	Move SIMD&FP scalar register to vector elements (unpredicated)	MOV (SIMD&FP scalar, unpredicated)
	Move SIMD&FP scalar register to vector elements (predicated)	MOV (SIMD&FP scalar, predicated)
	Move vector register (unpredicated)	MOV (scalar, unpredicated)
	Move vector register (predicated)	MOV (vector, predicated)
SEL	Select vector elements from two vectors	SEL (vectors)

5.2.3 Integer operations

I_{JZWFQ} The following instructions operate on signed or unsigned integer data within a vector.

5.2.3.1 Integer arithmetic

I_{FQSMH} For binary operations, these instructions perform arithmetic operations on a source vector containing integer element values, and a second source vector of either integer element values or an immediate value. For ternary operations, these instructions perform arithmetic operations on a source vector containing integer element values, a second source vector of either integer element values or an immediate value, and a third source vector containing integer element values.

Mnemonic	Instruction	See
ABS	Absolute value	ABS
ADD	Add vectors (predicated)	ADD (vectors, predicated)
	Add vectors (unpredicated)	ADD (vectors, unpredicated)
	Add immediate	ADD (immediate)
CNOT	Logically invert Boolean condition	CNOT
MAD	Multiply-add, writing to the multiplicand register	MAD
MLA	Multiply-add, writing to the addend register	MLA (vectors)
MLS	Multiply-subtract, writing to the addend register	MLS (vectors)
MSB	Multiply-subtract, writing to the multiplicand register	MSB
MUL	Multiply by immediate	MUL (immediate)
	Multiply vectors	MUL (vectors, predicated)
NEG	Negate	NEG
SABD	Signed absolute difference	SABD
SDIV	Signed divide	SDIV
SDIVR	Signed reversed divide	SDIVR
SMAX	Signed maximum with immediate	SMAX (immediate)
	Signed maximum vectors	SMAX (vectors)
SMIN	Signed minimum with immediate	SMIN (immediate)
	Signed minimum vectors	SMIN (vectors)
SMULH	Signed multiply returning high half	SMULH (predicated)
SQADD	Signed saturating add immediate	SQADD (immediate)
	Signed saturating add vectors	SQADD (vectors, unpredicated)
SQSUB	Signed saturating subtract immediate	SQSUB (immediate)
	Signed saturating subtract vectors	SQSUB (vectors, unpredicated)
SUB	Subtract immediate	SUB (immediate)
	Subtract vectors (predicated)	SUB (vectors, predicated)
	Subtract vectors (unpredicated)	SUB (vectors, unpredicated)
SUBR	Reversed subtract from immediate	SUBR (immediate)
	Reversed subtract vectors	SUBR (vectors)
SXTB	Signed byte extend	SXTB, SXTH, SXTW
SXTH	Signed halfword extend	SXTB, SXTH, SXTW
SXTW	Signed word extend	SXTB, SXTH, SXTW
UABD	Unsigned absolute difference	UABD
UDIV	Unsigned divide	UDIV
UDIVR	Unsigned reversed divide	UDIVR

Mnemonic	Instruction	See
UMAX	Unsigned maximum with immediate	UMAX (immediate)
	Unsigned maximum vectors	UMAX (vectors)
UMIN	Unsigned minimum with immediate	UMIN (immediate)
	Unsigned minimum vectors	UMIN (vectors)
UMULH	Unsigned multiply returning high half	UMULH (predicated)
UQADD	Unsigned saturating add immediate	UQADD (immediate)
	Unsigned saturating add vectors	UQADD (vectors, unpredicated)
UQSUB	Unsigned saturating subtract immediate	UQSUB (immediate)
	Unsigned saturating subtract vectors	UQSUB (vectors, unpredicated)
UXTB	Unsigned byte extend	UXTB, UXTH, UXTW
UXTH	Unsigned halfword extend	UXTB, UXTH, UXTW
UXTW	Unsigned word extend	UXTB, UXTH, UXTW

5.2.3.2 Integer dot product

I_{VRLQK} 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 first source vector are multiplied by the corresponding elements in the second source vector. The resulting widened products are summed and added to the 32-bit or 64-bit element of the accumulator and destination vector that aligns with the group of four elements in the first source vector.

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

The SUDOT and USDOT instructions are only supported ID_AA64ZFR0_EL1.I8MM is 1. The SUDOT and USDOT instructions only support groups of 8-bit elements.

Mnemonic	Instruction	See
SDOT	Signed dot product by vector	SDOT (vectors)
	Signed dot product by indexed elements	SDOT (indexed)
SUDOT	Signed by unsigned integer dot product by in dexed elements	SUDOT
UDOT	Unsigned dot product by vector	UDOT (vectors)
	Unsigned dot product by indexed elements	UDOT (indexed)
USDOT	Unsigned by signed integer dot product	USDOT (vectors)
	Unsigned by signed integer dot product by in dexed elements	USDOT (indexed)

5.2.3.3 Integer matrix multiply operations

I_{ZTKYV} These instructions facilitate matrix multiplication and include integer matrix multiply-accumulate instructions.

The matrix multiplication instructions that are supported depend on the value of ID_AA64ZFR0_EL1.I8MM. The following table displays the supported instructions if ID_AA64ZFR0_EL1.I8MM is 1.

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

- The first source vector 2x8 8-bit matrix is organized in row-by-row order.
- The second source vector 8x2 8-bit matrix is organized in a column-by-column order.
- The destination vector 2x2 32-bit matrix is organized in row-by-row order.

One matrix multiplication is performed per vector segment of 128 bits and accumulated into the destination vector segment.

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

5.2.3.4 Integer comparisons

I_{ZTCWQ} 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 FALSE. All integer comparisons set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero.

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

Mnemonic	Instruction	See
CMPEQ	Compare signed equal to immediate	CMP<cc> (immediate)
	Compare signed equal to wide elements	CMP<cc> (wide elements)
	Compare signed equal to vector	CMP<cc> (vectors)
CMPGE	Compare signed greater than or equal to immediate	CMP<cc> (immediate)
	Compare signed greater than or equal to wide elements	CMP<cc> (wide elements)
	Compare signed greater than or equal to vector	CMP<cc> (vectors)
CMPGT	Compare signed greater than immediate	CMP<cc> (immediate)
	Compare signed greater than wide elements	CMP<cc> (wide elements)
	Compare signed greater than vector	CMP<cc> (vectors)

Mnemonic	Instruction	See
CMPHI	Compare unsigned higher than immediate	CMP<cc> (immediate)
	Compare unsigned higher than wide elements	CMP<cc> (wide elements)
	Compare unsigned higher than vector	CMP<cc> (vectors)
CMPHS	Compare unsigned higher than or same as immediate	CMP<cc> (immediate)
	Compare unsigned higher than or same as wide elements	CMP<cc> (wide elements)
	Compare unsigned higher than or same as vector	CMP<cc> (vectors)
CMPLE	Compare signed less than or equal to immediate	CMP<cc> (immediate)
	Compare signed less than or equal to wide elements	CMP<cc> (wide elements)
	Compare signed less than or equal to vector	CMP<cc> (vectors)
CMPLO	Compare unsigned lower than immediate	CMP<cc> (immediate)
	Compare unsigned lower than 64-bit wide elements	CMP<cc> (wide elements)
	Compare unsigned lower than vector	CMP<cc> (vectors)
CMPLS	Compare unsigned lower or same as immediate	CMP<cc> (immediate)
	Compare unsigned lower or same as wide elements	CMP<cc> (wide elements)
	Compare unsigned lower or same as vector	CMP<cc> (vectors)
CMPLT	Compare signed less than immediate	CMP<cc> (immediate)
	Compare signed less than wide elements	CMP<cc> (wide elements)
	Compare signed less than vector	CMP<cc> (vectors)
CMPNE	Compare not equal to immediate	CMP<cc> (immediate)
	Compare not equal to wide elements	CMP<cc> (wide elements)
	Compare not equal to vector	CMP<cc> (vectors)

5.2.3.5 Vector address calculation

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

The ADR instruction is an integer arithmetic operation that is used to calculate a vector of 64-bit or 32-bit addresses.

The ADR destination vector elements are computed by the addition of the corresponding elements in the source vectors, with an optional sign or zero extension and optional bitwise left shift of 1-3 bits applied to the final operands. This can be considered as the addition of a vector base and a scaled vector index.

The ADR instruction computes a vector of 32-bit addresses by the addition of a 32-bit base and a scaled 32-bit unsigned index.

The ADR instruction computes a vector of 64-bit addresses by one of:

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

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

5.2.4 Bitwise operations

5.2.4.1 Bitwise logical operations

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

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

5.2.4.2 Bitwise shift, reverse, and count

I$_{ZMLFJ}$ Bitwise shifts, reversals, and counts within vector elements.

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

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

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

5.2.5 Floating-point operations

I$_{JXJHX}$ The following instructions operate on floating-point data within a vector. For more information, see 2.2 *SVE floating-point support*.

5.2.5.1 Floating-point arithmetic

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

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

5.2.5.2 Floating-point multiply accumulate

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

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

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

5.2.5.3 Floating-point complex arithmetic

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

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

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

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

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

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

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

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

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

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

In addition, when using these instructions, the behavior of calculations such as $(\infty+\infty)$ multiplied by (0+i) is (NaN+NaNi), rather than the result expected by ISO C, which is complex ∞. The expectation is that these instructions are only used in situations where the effect of differences in the rounding and handling of infinities are not material to the calculation.

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

5.2.5.4 Floating-point rounding and conversion

I_{TMXZV} 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.

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

5.2.5.5 Floating-point comparisons

I_KBYDX 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 FALSE. Floating-point vector comparisons do not set the condition flags.

Mnemonic	Instruction	See
FACGE	Floating-point absolute compare greater than or equal	FAC<cc>
FACGT	Floating-point absolute compare greater than	FAC<cc>
FACLE	Floating-point absolute compare less than or equal	FACLE
FACLT	Floating-point absolute compare less than	FACLT
FCMEQ	Floating-point compare equal to zero	FCM<cc> (zero)
	Floating-point compare equal to vector	FCM<cc> (vectors)
FCMGE	Floating-point compare greater than or equal to zero	FCM<cc> (zero)
	Floating-point compare greater than or equal to vector	FCM<cc> (vectors)
FCMGT	Floating-point compare greater than zero	FCM<cc> (zero)
	Floating-point compare greater than vector	FCM<cc> (vectors)
FCMLE	Floating-point compare less than or equal to zero	FCM<cc> (zero)
	Floating-point compare less than or equal to vector	FCM<cc> (vectors)
FCMLT	Floating-point compare less than zero	FCM<cc> (zero)
	Floating-point compare less than vector	FCM<cc> (vectors)
FCMNE	Floating-point compare not equal to zero	FCM<cc> (zero)
	Floating-point compare not equal to vector	FCM<cc> (vectors)
FCMUO	Floating-point unordered vectors	FCM<cc> (vectors)

5.2.5.6 Floating-point transcendental acceleration

I_LRHVT 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.

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

5.2.5.7 Floating-point indexed multiplies

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

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

5.2.5.8 Floating-point matrix multiply operations

I$_{QSQWL}$ These instructions facilitate matrix multiplication and include floating-point matrix multiply-accumulate instructions, and companion instructions that support data rearrangements in vector registers as required by the double-precision matrix multiply-accumulate instructions.

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

- The first source vector 2x2 matrix is organized in row-by-row order.
- The second source vector 2x2 matrix is organized in a column-by-column order.
- The destination vector 2x2 matrix is organized in row-by-row order.

One matrix multiplication is performed per vector segment and accumulated into the destination vector segment. For the double-precision matrix multiply-accumulate instructions, the vector segment length and minimum vector length is 256 bits. Double-precision matrix multiply-accumulate instructions are not supported when the vector length is 128 bits. For the single-precision matrix multiply-accumulate instruction, the vector segment length is 128 bits.

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

I$_{TJXGZ}$ The following table shows the floating-point matrix multiplication instructions and companion instructions that are supported if ID_AA64ZFR0_EL1.F64MM is 1:

Mnemonic	Instruction	See
FMMLA	Floating-point matrix multiply-accumulate into 2x2 matrix (double-precision)	FMMLA
LD1ROB	Contiguous load and replicate thirty-two bytes, scalar plus scalar	LD1ROB (scalar plus scalar)
	Contiguous load and replicate thirty-two bytes, scalar plus immediate	LD1ROB (scalar plus immediate)
LD1ROD	Contiguous load and replicate four doublewords, scalar plus scalar	LD1ROD (scalar plus scalar)
	Contiguous load and replicate four doublewords, scalar plus immediate	LD1ROD (scalar plus immediate)

Mnemonic	Instruction	See
LD1ROH	Contiguous load and replicate sixteen halfwords, scalar plus scalar	LD1ROH (scalar plus scalar)
	Contiguous load and replicate sixteen halfwords, scalar plus immediate	LD1ROH (scalar plus immediate)
LD1ROW	Contiguous load and replicate eight words, scalar plus scalar	LD1ROW (scalar plus scalar)
	Contiguous load and replicate eight words, scalar plus immediate	LD1ROW (scalar plus immediate)
TRN1, TRN2	Interleave even or odd 128-bit elements from two vectors	TRN1, TRN2 (vectors)
UZP1, UZP2	Concatenate even or odd 128-bit elements from two vectors	UZP1, UZP2 (vectors)
ZIP1, ZIP2	Interleave 128-bit elements from two half vectors	ZIP1, ZIP2 (vectors)

I_{PTVQV} The following table shows the floating-point matrix multiplication instructions and companion instructions that are supported if ID_AA64ZFR0_EL1.F32MM is 1:

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

5.2.5.9 BFloat16 floating-point multiply instructions

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

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

The BFloat16 matrix multiply-accumulate instructions delimit source and destination vectors into segments. Within each segment:

- The first source vector 2x4 BF16 matrix is organized in row-by-row order.
- The second source vector 4x2 BF16 matrix is organized in a column-by-column order.
- The destination vector 2x2 single-precision matrix is organized in row-by-row order.

One matrix multiplication is performed per vector segment of 128 bits and accumulated into the destination vector.

The BFloat16 instructions are only supported if ID_AA64ZFR0_EL1.BF16 is 1.

Mnemonic	Instruction	See
BFDOT	BFloat16 floating-point dot product by vector	BFDOT (vectors)
	BFloat16 floating-point dot product by indexed elements	BFDOT (indexed)
BFMMLA	BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix	BFMMLA

Mnemonic	Instruction	See
BFMLALB	BFloat16 floating-point widening multiply accumulate long bottom by vector	BFMLALB (vectors)
	BFloat16 floating-point widening multiply accumulate long bottom by indexed elements	BFMLALB (indexed)
BFMLALT	BFloat16 floating-point widening multiply accumulate long top by vector	BFMLALT (vectors)
	BFloat16 floating-point widening multiply accumulate long bottom by indexed elements	BFMLALT (vectors)

5.2.6 Predicate operations

I_{LDPLK} These instructions perform operations that manipulate the predicate registers.

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

5.2.6.1 Predicate initialization

I_{RNWTM} These instructions initialize predicate elements.

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

- A fixed number of elements from the following range: VL1-VL8, VL16, VL32, VL64, VL128 or VL256.
- The largest power of two elements, POW2.
- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, which is implicitly a multiple of two.

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

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

5.2.6.2 Predicate move operations

I_{QFBXK} These instructions operate on all bits of the predicate registers, implying a fixed, 1-bit predicate element size. The flag-setting variants set the N, Z, and C condition flags based on the predicate result, and set the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the flag-setting variants should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

Mnemonic	Instruction	See
SEL	Select predicate elements from two predicates	SEL (predicates)

Mnemonic	Instruction	See
MOV	Move predicate elements (predicated, merging)	MOV (predicate, predicated, merging)
	Move predicate elements (predicated, zeroing)	MOV (predicate, predicated, zeroing)
	Move predicate elements (unpredicated)	MOV (predicate, unpredicated)
MOVS	Move predicate elements, setting the condition flags (predicated)	MOVS (predicated)
	Move predicate elements, setting the condition flags (unpredicated)	MOVS (unpredicated)

5.2.6.3 Predicate logical operations

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

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

5.2.6.4 FFR predicate handling

I_{LTYHB} These instructions work with *SVE* First-fault and Non-fault loads using the FFR to determine which elements have been successfully loaded and which remain to be loaded on a subsequent iteration. The RDFFRS instruction sets the N, Z, and C condition flags based on the predicate result, and sets the V flag to zero. Because these instructions operate with a fixed, 1-bit element size, the *Governing predicate* for the RDFFRS instruction should be in the canonical form for a predicate element size in order to generate a meaningful set of condition flags for that element size.

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

5.2.6.5 Predicate counts

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

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

The named predicate constraint limits the number of elements to:

- A fixed number of elements from the following range: VL1-VL8, VL16, VL32, VL64, VL128 or VL256.
- The largest power of two elements, POW2.
- The largest multiple of three or four elements, MUL3 or MUL4.
- The number of accessible elements, ALL, implicitly a multiple of two.

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

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

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

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

Mnemonic	Instruction	See
	Unsigned saturating increment vector by the number of predicate elements that are TRUE	UQINCP (vector)

5.2.6.6 Loop control

I_{CSCJP} These instructions control counted vector loops and vector loops with data-dependent termination conditions.

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

5.2.6.6.1 Simple loops

I_{FXMJD} An up-counting WHILE instruction that increments the value of the first scalar operand and compares the value with a second, fixed scalar operand. The instruction generates a destination predicate with all of the following characteristics:

- The predicate elements starting from the lowest numbered element are true while the comparison is true.
- The predicate elements thereafter, up to the highest numbered element, are false when the comparison becomes false.

All 32 bits or 64 bits of the scalar operands are significant for the purposes of comparison. The full 32-bit or 64-bit value of the first operand is incremented by 1 for each destination predicate element, irrespective of the element size. The first general-purpose register operand is not updated.

If all of the following occur, a comparison can never fail, resulting in an all-true predicate:

- The comparison includes an equality test.
- The second scalar operand is equal to the maximum integer value of the selected size and type of comparison.

The N, Z, C, and V condition flags are unconditionally set to control a subsequent conditional branch.

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

5.2.6.6.2 Data-dependent loops

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

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

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

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

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

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

5.2.6.7 Serialized operations

I$_{ZVHQH}$ These instructions permit *Active elements* within a vector to be processed sequentially without unpacking the vector. The condition flags are unconditionally set to control a subsequent conditional branch.

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

5.2.7 Move operations

5.2.7.1 Element permute and shuffle

I$_{DNKML}$ These instructions move data between different vector elements, or between vector elements and scalar registers.

These instructions perform the following operations:

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

Mnemonic	Instruction	See
CLASTA	Conditionally extract element after the *Last active element* to general-purpose register	CLASTA (scalar)
	Conditionally extract element after the *Last active element* to SIMD&FP scalar	CLASTA (SIMD&FP scalar)
	Conditionally extract element after the *Last active element* to vector	CLASTA (vectors)
CLASTB	Conditionally extract *Last active element* to general-purpose register	CLASTB (scalar)
	Conditionally extract *Last active element* to SIMD&FP scalar	CLASTB (SIMD&FP scalar)
	Conditionally extract *Last active element* to vector	CLASTB (vectors)
LASTA	Extract element after the *Last active element* to general-purpose register	LASTA (scalar)
	Extract element after the *Last active element* to SIMD&FP scalar	LASTA (SIMD&FP scalar)
LASTB	Extract *Last active element* to general-purpose register	LASTB (scalar)
	Extract *Last active element* to SIMD&FP scalar	LASTB (SIMD&FP scalar)
COMPACT	Shuffle *Active elements* of vector to the right and fill with zeros	COMPACT
SPLICE	Splice two vectors under predicate control	SPLICE
TBL	Programmable table lookup using vector of element indexes	TBL
DUP	Broadcast indexed vector element	DUP
EXT	Extract vector from pair of vectors	EXT
INSR	Insert general-purpose register into shifted vector	INSR (scalar)
	Insert SIMD&FP scalar register into shifted vector	INSR (SIMD&FP scalar)
MOV	Move indexed element or SIMD&FP scalar to vector (unpredicated)	MOV (SIMD&FP scalar, unpredicated)
	Move SIMD&FP scalar register to vector elements (predicated)	MOV (SIMD&FP scalar, predicated)
REV	Reverse all elements in vector	REV (vector)
REVB	Reverse 8-bit bytes in elements	REVB, REVH, REVW

Mnemonic	Instruction	See
REVH	Reverse 16-bit halfwords in elements	REVB, REVH, REVW
REVW	Reverse 32-bit words in elements	REVB, REVH, REVW
TRN1	Interleave even elements from two vectors	TRN1, TRN2 (vectors)
TRN2	Interleave odd elements from two vectors	TRN1, TRN2 (vectors)
UZP1	Concatenate even elements from two vectors	UZP1, UZP2 (vectors)
UZP2	Concatenate odd elements from two vectors	UZP1, UZP2 (vectors)
ZIP1	Interleave elements from low halves of two vectors	ZIP1, ZIP2 (vectors)
ZIP2	Interleave elements from high halves of two vectors	ZIP1, ZIP2 (vectors)

5.2.7.2 Unpacking instructions

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

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

5.2.7.3 Predicate permute

I_{QWFQC} These instructions are used to move and permute predicate elements. These instructions generally mirror the fixed vector permutes to allow predicates to follow their data. The permutes move all of the bits in a predicate element, not just the canonical bits.

Mnemonic	Instruction	See
REV	Reverse all elements in predicate	REV
TRN1	Interleave even elements from two predicates	TRN1, TRN2 (predicates)
TRN2	Interleave odd elements from two predicates	TRN1, TRN2 (predicates)
UZP1	Select even elements from two predicates	UZP1, UZP2 (predicates)
UZP2	Select odd elements from two predicates	UZP1, UZP2 (predicates)
ZIP1	Interleave elements from low halves of two predicates	ZIP1, ZIP2 (predicates)

Mnemonic	Instruction	See
ZIP2	Interleave elements from high halves of two predicates	ZIP1, ZIP2 (predicates)

5.2.7.4 Index vector generation

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

Mnemonic	Instruction	See
INDEX	Create index vector starting from and incremented by immediates	INDEX (immediates)
	Create index vector starting from immediate and incremented by general-purpose register	INDEX (immediate, scalar)
	Create index vector starting from general-purpose register and incremented by immediate	INDEX (scalar, immediate)
	Create index vector starting from and incremented by general-purpose registers	INDEX (scalars)

5.2.7.5 Move prefix

$\mathrm{I}_{\mathrm{TFSHM}}$ The MOVPRFX (predicated) instruction is a predicated vector move that can be combined with a predicated destructive instruction that immediately follows it, in program order, to create a single constructive operation, or to convert an instruction with merging predication to use zeroing predication.

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

The *Operational information* section of an *SVE* instruction description indicates whether or not an instruction can be predictably prefixed by a MOVPRFX instruction. If the *Operational information* of an *SVE* instruction description does not mention MOVPRFX or if the section does not exist, then the instruction cannot be predictably prefixed by a MOVPRFX instruction.

The prefixed instruction that immediately follows a MOVPRFX instruction in program order must be an *SVE* instruction that can be predictably prefixed by a MOVPRFX instruction, or an A64 HLT instruction, or an A64 BRK instruction. For an *SVE* instruction that can be predictably prefixed by a MOVPRFX instruction, all of the following apply:

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

- If the MOVPRFX instruction is unpredicated, then the prefixed instruction can use any *Governing predicate* register and element size, or it can be unpredicated. A predicated MOVPRFX cannot be used with an unpredicated instruction.

If the instruction that follows a MOVPRFX instruction is not an *SVE* instruction that can be predictably prefixed by a MOVPRFX instruction, the two instructions behave in one of the following CONSTRAINED UNPREDICTABLE ways:

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

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

Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that, for performance reasons, software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

R_{HLFJD} When a MOVPRFX instruction is executed, except for PMU events SVE_MOVPRFX_SPEC, SVE_MOVPRFX_Z_SPEC, SVE_MOVPRFX_M_SPEC, and SVE_MOVPRFX_U_SPEC, `0x807C`-`0x807F`, it is IMPLEMENTATION DEFINED for each execution of the instruction whether or not any Performance Monitor counts the instruction. This can vary dynamically for each execution of the same instruction.

R_{WWTRW} When a microarchitectural operation is executed because of a MOVPRFX instruction, except for PMU events SVE_MOVPRFX_SPEC, SVE_MOVPRFX_Z_SPEC, SVE_MOVPRFX_M_SPEC, and SVE_MOVPRFX_U_SPEC, `0x807C`-`0x807F`, it is IMPLEMENTATION DEFINED for each execution of the operation whether or not the Performance Monitor counts the operation. This can vary dynamically for each execution of the same instruction.

5.2.8 Reduction operations

5.2.8.1 Horizontal reductions

I_{PWNDL} These instructions perform arithmetic horizontally across *Active elements* of a single source vector and deliver a scalar result.

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

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

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

Mnemonic	Instruction	See
ANDV	Bitwise AND reduction, treating *Inactive elements* as all ones	ANDV

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

5.3 SVE2 ISA functional groups

5.3.1 Down-counting Loops

$\mathrm{I}_{\mathrm{TXYXY}}$ A down-counting WHILE instruction decrements the value of the first scalar operand and compares the value with a second, fixed scalar operand. The instruction generates a destination predicate with all of the following characteristics:

- The predicate elements starting from the highest-numbered element are true while the comparison remains true.
- The predicate elements thereafter, down to the lowest-numbered element, are false when the comparison becomes false.

All 32 bits or 64 bits of the scalar operands are significant for the purposes of comparison. The full 32-bit or 64-bit value of the first operand is decremented by 1 for each destination predicate element, irrespective of the element size. The first general-purpose register operand is not updated.

If all of the following occur, a comparison can never fail, resulting in an all-true predicate:

- The comparison includes an equality test.
- The second scalar operand is equal to the minimum integer value of the selected size and type of comparison.

The following are the *SVE2* down-counting WHILE instructions:

- WHILEGE. While decrementing signed 32-bit or 64-bit scalar greater than or equal to scalar.
- WHILEGT. While decrementing signed 32-bit or 64-bit scalar greater than scalar.
- WHILEHI. While decrementing unsigned 32-bit or 64-bit scalar higher than scalar.
- WHILEHS. While decrementing unsigned 32-bit or 64-bit scalar higher or same as scalar.

5.3.2 Constructive multiply

$\mathrm{I}_{\mathrm{BSYCX}}$ SVE2 includes the following constructive, three-operand versions of the integer multiply instructions:

- MUL (vectors, unpredicated). Multiply vectors (unpredicated).
- SMULH (unpredicated). Signed multiply returning high half (unpredicated).
- UMULH (unpredicated). Unsigned multiply returning high half (unpredicated).

5.3.3 Uniform DSP operations

$\mathrm{I}_{\mathrm{QDSMK}}$ The uniform DSP instructions are based on AArch64 Advanced SIMD instructions with the same mnemonic. The instructions operate on fixed-point operands and produce results with a uniform element size. The operation of an instruction might include one or more of rounding, halving, saturation, and accumulation.

The following are the SVE2 uniform DSP instructions:

- SABA. Signed absolute difference and accumulate.
- SHADD. Signed halving addition.
- SHSUB. Signed halving subtract.
- SHSUBR. Signed halving subtract reversed vectors.
- SLI. Shift left and insert (immediate).
- SQABS. Signed saturating absolute value.
- SQADD (vectors, predicated). Signed saturating addition (predicated).
- SQDMULH (vectors). Signed saturating doubling multiply high (unpredicated).
- SQNEG. Signed saturating negate.
- SQRDCMLAH (vectors). Signed saturating rounding doubling multiply-add high to accumulator (unpredicated).

- SQRDMLSH (vectors). Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated).
- SQRDMULH (vectors). Signed saturating rounding doubling multiply high (unpredicated).
- SQRSHL. Signed saturating rounding shift left by vector (predicated).
- SQRSHLR. Signed saturating rounding shift left reversed vectors (predicated).
- SQSHL (immediate). Signed saturating shift left by immediate.
- SQSHL (vectors). Signed saturating shift left by vector (predicated).
- SQSHLR. Signed saturating shift left reversed vectors (predicated).
- SQSHLU. Signed saturating shift left unsigned by immediate.
- SQSUB (vectors, predicated). Signed saturating subtraction (predicated).
- SQSUBR. Signed saturating subtraction reversed vectors (predicated).
- SRHADD. Signed rounding halving addition.
- SRI. Shift right and insert (immediate).
- SRSHL. Signed rounding shift left by vector (predicated).
- SRSHLR. Signed rounding shift left reversed vectors (predicated).
- SRSHR. Signed rounding shift right by immediate.
- SRSRA. Signed rounding shift right and accumulate (immediate).
- SSRA. Signed shift right and accumulate (immediate).
- SUQADD. Signed saturating addition of unsigned value.
- UABA. Unsigned absolute difference and accumulate.
- UHADD. Unsigned halving addition.
- UHSUB. Unsigned halving subtract.
- UHSUBR. Unsigned halving subtract reversed vectors.
- UQADD (vectors, predicated). Unsigned saturating addition (predicated).
- UQRSHL. Unsigned saturating rounding shift left by vector (predicated).
- UQRSHLR. Unsigned saturating rounding shift left reversed vectors (predicated).
- UQSHL (immediate). Unsigned saturating shift left by immediate.
- UQSHL (vectors). Unsigned saturating shift left by vector (predicated).
- UQSHLR. Unsigned saturating shift left reversed vectors (predicated).
- UQSUB (vectors, predicated). Unsigned saturating subtraction (predicated).
- UQSUBR. Unsigned saturating subtraction reversed vectors (predicated).
- URECPE. Unsigned reciprocal estimate (predicated).
- URHADD. Unsigned rounding halving addition.
- URSHL. Unsigned rounding shift left by vector (predicated).
- URSHLR. Unsigned rounding shift left reversed vectors (predicated).
- URSHR. Unsigned rounding shift right by immediate.
- URSQRTE. Unsigned reciprocal square root estimate (predicated).
- URSRA. Unsigned rounding shift right and accumulate (immediate).
- USQADD. Unsigned saturating addition of signed value.
- USRA. Unsigned shift right and accumulate (immediate).

5.3.4 Widening DSP operations

I_{ZCKQT} The widening DSP instructions are based on AArch64 Advanced SIMD instructions with similar mnemonics. The instructions operate on fixed-point values and produce results that are twice the width of some or all of the inputs. The instructions read the narrow inputs from either the even-numbered (bottom) or odd-numbered (top) source elements and place each result in the double-width destination elements that overlap the narrow source elements. The widening DSP operations are unpredicated and constructive.

The following are the SVE2 widening DSP instructions:

- SABALB. Signed absolute difference and accumulate long (bottom).
- SABALT. Signed absolute difference and accumulate long (top).
- SABDLB. Signed absolute difference long (bottom).
- SABDLT. Signed absolute difference long (top).

- SADDLB. Signed add long (bottom).
- SADDLT. Signed add long (top).
- SADDWB. Signed add wide (bottom).
- SADDWT. Signed add wide (top).
- SMLALB (vectors). Signed multiply-add long to accumulator (bottom).
- SMLALT (vectors). Signed multiply-add long to accumulator (top).
- SMLSLB (vectors). Signed multiply-subtract long from accumulator (bottom).
- SMLSLT (vectors). Signed multiply-subtract long from accumulator (top).
- SMULLB (vectors). Signed multiply long (bottom).
- SMULLT (vectors). Signed multiply long (top).
- SQDMLALB (vectors). Signed saturating doubling multiply-add long to accumulator (bottom).
- SQDMLALT (vectors). Signed saturating doubling multiply-add long to accumulator (top).
- SQDMLSLB (vectors). Signed saturating doubling multiply-subtract long from accumulator (bottom).
- SQDMLSLT (vectors). Signed saturating doubling multiply-subtract long from accumulator (top).
- SQDMULLB (vectors). Signed saturating doubling multiply long (bottom).
- SQDMULLT (vectors). Signed saturating doubling multiply long (top).
- SSHLLB. Signed shift left long by immediate (bottom).
- SSHLLT. Signed shift left long by immediate (top).
- SSUBLB. Signed subtract long (bottom).
- SSUBLT. Signed subtract long (top).
- SSUBWB. Signed subtract wide (bottom).
- SSUBWT. Signed subtract wide (top).
- UABALB. Unsigned absolute difference and accumulate long (bottom).
- UABALT. Unsigned absolute difference and accumulate long (top).
- UABDLB. Unsigned absolute difference long (bottom).
- UABDLT. Unsigned absolute difference long (top).
- UADDLB. Unsigned add long (bottom).
- UADDLT. Unsigned add long (top).
- UADDWB. Unsigned add wide (bottom).
- UADDWT. Unsigned add wide (top).
- UMLALB (vectors). Unsigned multiply-add long to accumulator (bottom).
- UMLALT (vectors). Unsigned multiply-add long to accumulator (top).
- UMLSLB (vectors). Unsigned multiply-subtract long from accumulator (bottom).
- UMLSLT (vectors). Unsigned multiply-subtract long from accumulator (top).
- UMULLB (vectors). Unsigned multiply long (bottom).
- UMULLT (vectors). Unsigned multiply long (top).
- USHLLB. Unsigned shift left long by immediate (bottom).
- USHLLT. Unsigned shift left long by immediate (top).
- USUBLB. Unsigned subtract long (bottom).
- USUBLT. Unsigned subtract long (top).
- USUBWB. Unsigned subtract wide (bottom).
- USUBWT. Unsigned subtract wide (top).

5.3.5 Narrowing DSP operations

ITTJZR The narrowing DSP instructions are based on AArch64 Advanced SIMD instructions with similar mnemonics. The instructions operate on fixed-point values and produce results that are half the width of the inputs. The instructions read wide source elements and place each result in one of the following:

- The overlapped even-numbered (bottom) half-width destination elements. The odd-numbered destination elements are set to zero.
- The overlapped odd-numbered (top) half-width destination elements. The even-numbered destination elements are unchanged, which means that the instructions are implicitly merging operations.

The narrowing DSP operations are unpredicated and constructive.

The following are the SVE2 narrowing DSP instructions:

- ADDHNB. Add narrow high part (bottom).
- ADDHNT. Add narrow high part (top).
- RADDHNB. Rounding add narrow high part (bottom).
- RADDHNT. Rounding add narrow high part (top).
- RSHRNB. Rounding shift right narrow by immediate (bottom).
- RSHRNT. Rounding shift right narrow by immediate (top).
- RSUBHNB. Rounding subtract narrow high part (bottom).
- RSUBHNT. Rounding subtract narrow high part (top).
- SHRNB. Shift right narrow by immediate (bottom).
- SHRNT. Shift right narrow by immediate (top).
- SQRSHRNB. Signed saturating rounding shift right narrow by immediate (bottom).
- SQRSHRNT. Signed saturating rounding shift right narrow by immediate (top).
- SQRSHRUNB. Signed saturating rounding shift right unsigned narrow by immediate (bottom).
- SQRSHRUNT. Signed saturating rounding shift right unsigned narrow by immediate (top).
- SQSHRNB. Signed saturating shift right narrow by immediate (bottom).
- SQSHRNT. Signed saturating shift right narrow by immediate (top).
- SQSHRUNB. Signed saturating shift right unsigned narrow by immediate (bottom).
- SQSHRUNT. Signed saturating shift right unsigned narrow by immediate (top).
- SUBHNB. Subtract narrow high part (bottom).
- SUBHNT. Subtract narrow high part (top).
- UQRSHRNB. Unsigned saturating rounding shift right narrow by immediate (bottom).
- UQRSHRNT. Unsigned saturating rounding shift right narrow by immediate (top).
- UQSHRNB. Unsigned saturating shift right narrow by immediate (bottom).
- UQSHRNT. Unsigned saturating shift right narrow by immediate (top).

5.3.6 Unary narrowing operations

I_{SMSNH} The unary narrowing instructions are unpredicated and do not write to the source register. The instructions read elements from the source vector and saturate each value to the half-width destination element size. The instructions place the narrow results in one of the following:

- The overlapped even-numbered (bottom) half-width elements. The odd-numbered elements are set to zero.
- The overlapped odd-numbered (top) half-width elements. The even-numbered elements are unchanged.

Non-saturating (truncating) conversions can be performed using existing *SVE* instructions such as shifts, masks, and permutes.

The following are the *SVE2* unary narrowing instructions:

- SQXTNB. Signed saturating extract narrow (bottom).
- SQXTNT. Signed saturating extract narrow (top).
- SQXTUNB. Signed saturating unsigned extract narrow (bottom).
- SQXTUNT. Signed saturating unsigned extract narrow (top).
- UQXTNB. Unsigned saturating extract narrow (bottom).
- UQXTNT. Unsigned saturating extract narrow (top).

5.3.7 Non-widening pairwise arithmetic

I_{XKYJD} The non-widening pairwise arithmetic instructions operate on pairs of adjacent elements in each source vector and produce a result element that is the same size as a single input element. The results from the first and second source vectors are interleaved, so that the source and result elements overlap. The result is destructively placed in the first source vector. The AArch64 Advanced SIMD instructions do not interleave the results from the first and second source vectors.

Predication applies to the destination vector. The even-numbered predicate elements enable an operation on a pair of elements in the first source vector. The odd-numbered predicate elements enable an operation on a pair of elements in the second source vector.

Inactive elements in the destination vector register are not modified.

The following are the SVE2 non-widening pairwise arithmetic instructions:

- ADDP. Add pairwise.
- FADDP. Floating-point add pairwise.
- FMAXNMP. Floating-point maximum number pairwise.
- FMAXP. Floating-point maximum pairwise.
- FMINNMP. Floating-point minimum number pairwise.
- FMINP. Floating-point minimum pairwise.
- SMAXP. Signed maximum pairwise.
- SMINP. Signed minimum pairwise.
- UMAXP. Unsigned maximum pairwise.
- UMINP. Unsigned minimum pairwise.

5.3.8 Widening pairwise arithmetic

I_{MHJXQ} The widening pairwise arithmetic instructions operate on pairs of adjacent elements in a single source vector and produce a double-width result element that is accumulated into the destination vector.

Inactive elements in the destination vector register are not modified.

The following are the SVE2 widening pairwise arithmetic instructions:

- SADALP. Signed add and accumulate long pairwise.
- UADALP. Unsigned add and accumulate long pairwise.

5.3.9 Bitwise ternary logical instructions

I_{NWVWW} The bitwise ternary logical instructions enable complex bit processing codes to be accelerated using multiple bitwise logical operations in a shorter instruction sequence. All of the following operations are supported by the bitwise ternary logical instructions:

- The BCAX instruction combines a ternary bitwise clear with an exclusive OR.
- The EOR3 instruction provides a ternary exclusive OR.
- The XAR instruction combines an exclusive OR with rotation by a constant amount.
- The bitwise select instructions (BSL, BSL1N, BSL2N, and NBSL) can be used with other bitwise logical instructions to generate all 256 possible bitwise combinations of three input bits using at most three instructions.

The bitwise ternary logical instructions are unpredicated.

The following are the *SVE2* bitwise ternary logical instructions:

- BCAX. Bitwise clear and exclusive OR.
- BSL. Bitwise select.
- BSL1N. Bitwise select with first input inverted.
- BSL2N. Bitwise select with second input inverted.
- EOR3. Bitwise exclusive OR of three vectors.
- NBSL. Bitwise inverted select.
- XAR. Bitwise exclusive OR and rotate right by immediate.

5.3.10 Large integer arithmetic

I_YKWKM The large integer arithmetic instructions aid the processing of large integers in vector registers by maintaining multiple carry chains that are interleaved in accumulator vectors.

A large integer arithmetic instruction takes as input all of the following:

- Either the even-numbered (bottom) or odd-numbered (top) elements of the first source vector.
- A 1-bit carry input from the least-significant bit of the odd-numbered elements of the second source vector.

The inputs to the instruction are added to or subtracted from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

The following are the SVE2 large integer arithmetic instructions:

- ADCLB. Add with carry long (bottom).
- ADCLT. Add with carry long (top).
- SBCLB. Subtract with carry long (bottom).
- SBCLT. Subtract with carry long (top).

5.3.11 Multiplication by indexed elements

I_ZXGXY The multiplication by indexed elements instructions take all integer elements in each 128-bit vector segment of the first source vector and multiplies them by the indexed element in the corresponding segment of the second source vector.

The products might be destructively added to or subtracted from the corresponding elements of an addend vector.

The second source vector elements are specified using an immediate index that selects the same element position in each 128-bit vector segment. The index range is 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the element size.

The following are the SVE2 multiplication by indexed elements instructions:

- MLA (indexed). Multiply-add to accumulator (indexed).
- MLS (indexed). Multiply-subtract from accumulator (indexed).
- MUL (indexed). Multiply (indexed).
- SMLALB (indexed). Signed multiply-add long to accumulator (bottom, indexed).
- SMLALT (indexed). Signed multiply-add long to accumulator (top, indexed).
- SMLSLB (indexed). Signed multiply-subtract long from accumulator (bottom, indexed).
- SMLSLT (indexed). Signed multiply-subtract long from accumulator (top, indexed).
- SMULLB (indexed). Signed multiply long (bottom, indexed).
- SMULLT (indexed). Signed multiply long (top, indexed).
- SQDMLALB (indexed). Signed saturating doubling multiply-add long to accumulator (bottom, indexed).
- SQDMLALT (indexed). Signed saturating doubling multiply-add long to accumulator (top, indexed).
- SQDMLSLB (indexed). Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed).
- SQDMLSLT (indexed). Signed saturating doubling multiply-subtract long from accumulator (top, indexed).
- SQDMULH (indexed). Signed saturating doubling multiply high (indexed).
- SQDMULLB (indexed). Signed saturating doubling multiply long (bottom, indexed).
- SQDMULLT (indexed). Signed saturating doubling multiply long (top, indexed).
- SQRDMLAH (indexed). Signed saturating rounding doubling multiply-add high to accumulator (indexed).
- SQRDMLSH (indexed). Signed saturating rounding doubling multiply-subtract high from accumulator (indexed).
- SQRDMULH (indexed). Signed saturating rounding doubling multiply high (indexed).
- UMLALB (indexed). Unsigned multiply-add long to accumulator (bottom, indexed).
- UMLALT (indexed). Unsigned multiply-add long to accumulator (top, indexed).

- UMLSLB (indexed). Unsigned multiply-subtract long from accumulator (bottom, indexed).
- UMLSLT (indexed). Unsigned multiply-subtract long from accumulator (top, indexed).
- UMULLB (indexed). Unsigned multiply long (bottom, indexed).
- UMULLT (indexed). Unsigned multiply long (top, indexed).

5.3.12 Complex integer arithmetic

I_{PBSDF} Complex integer arithmetic instructions operate on signed integer complex numbers in vectors containing the following interleaved element pairs:

- The even-numbered elements contain the real parts of the complex numbers.
- The odd-numbered elements contain the imaginary parts of the complex numbers.

5.3.12.1 Uniform complex integer arithmetic

I_{CCVPK} The uniform complex integer arithmetic instructions operate on vectors containing integral complex numbers. The instructions operate on complex numbers that are in polar form.

The CADD instructions rotate the complex numbers in the second source vector before adding element pairs to the corresponding elements of the first source vector, in a destructive manner. The rotation direction is 90 degrees or 270 degrees from the positive real axis towards the positive imaginary axis.

The CMLA instructions transform the operands to enable multiply-add or multiply-subtract operations on complex numbers by combining two of the instructions. The following transformations are done:

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

The resulting products are added to the corresponding components of the destination and addend vector.

Two CMLA instructions can be used as follows:

```
CMLA Zda.S, Zn.S, Zm.S, #A CMLA Zda.S, Zn.S, Zm.S, #B
```

Some meaningful combinations of A and B are:

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

The CMLA (indexed) indexed form uses a single complex number in each 128-bit segment of the second source vector as the multiplier for all complex numbers in the corresponding first source vector segment. The complex numbers in the second source vector are specified using an immediate index that selects the same complex number position in each 128-bit vector segment. The index range is 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the complex number size.

The following are the *SVE2* uniform complex integer arithmetic instructions:

- CADD. Complex integer add with rotate.
- CMLA (vectors). Complex integer multiply-add with rotate.
- CMLA (indexed). Complex integer multiply-add with rotate (indexed).
- SQCADD. Saturating complex integer add with rotate.
- SQRDCMLAH (vectors). Saturating rounding doubling complex integer multiply-add high with rotate.

- SQRDCMLAH (indexed). Saturating rounding doubling complex integer multiply-add high with rotate (indexed).

5.3.12.2 Widening complex integer arithmetic

I_{JPXSW} The widening complex integer instructions deinterleave the real and imaginary components of integral complex numbers, and generate complex result components that have a higher numeric precision than the input values. The instructions differ from other complex instructions that process the real and imaginary components of complex numbers and write the complex result components to the destination.

The following instructions are useful when generating the widened components of the result of a complex multiply-add:

- SQDMLALBT: the imaginary results.
- SQDMLSLT: the real results.
- SQDMLALB: the conjugate real results.
- SQDMLSLBT: the conjugate imaginary results.

The following instructions are useful when generating the widened components of the result of a complex addition (X + jY) or (X - jY), given complex numbers X and Y.

- SADDLBT: the imaginary results when computing (X + jY) or real values when computing (X - jY).
- SSUBLBT: the real results when computing (X + jY) or imaginary values when computing (X - jY).

The following are the *SVE2* widening complex integer instructions:

- SADDLBT. Signed add long (bottom + top).
- SQDMLALBT. Signed saturating doubling multiply-add long to accumulator (bottom × top).
- SQDMLSLBT. Signed saturating doubling multiply-subtract long from accumulator (bottom × top).
- SSUBLBT. Signed subtract long (bottom - top).
- SSUBLTB. Signed subtract long (top - bottom).

5.3.12.3 Complex integer dot product

I_{THWWJ} The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. The complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector. The wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element that overlaps all four of the elements that form a pair of complex number values in the first source vector.

Each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates four wide real sums or four wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180, or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, considered in polar form, by applying the following transformations before the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector accumulates the imaginary parts of a complex conjugate dot product.

The indexed form of the instruction selects a single complex number pair within each 128-bit segment of the second source vector to multiply with all complex number pairs within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index which selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment, encoded in 1 or 2 bits depending on the size of the complex number pair.

Each complex number is represented in a vector register as an even and odd pair of elements. The real part is the even-numbered element and the imaginary part is the odd-numbered element.

The following are the *SVE2* complex integer dot product instructions:

- CDOT (vectors). Complex integer dot product.
- CDOT (indexed). Complex integer dot product (indexed).

5.3.13 Floating-point extra conversions

I_{DRDQS} The floating-point extra conversion instructions convert to and from fully packed vectors of narrower floating-point elements.

The FCVTLT instruction converts the top or odd-numbered narrow floating-point vector elements to wider elements of the next higher precision. The conversion is similar to what is done by the widening integer instructions.

The FCVTNT and FCVXNT instructions convert wider floating-point vector elements to the top or odd-numbered narrower elements of the next lower precision. The conversion is similar to what is done by the narrowing integer instructions.

The FCVTXNT and FCVTX instructions convert from double-precision to fully packed half-precision in two narrowing steps, double-precision to single-precision and then single-precision to half-precision. The two-step conversion is done without an intermediate rounding error by using von Neumann rounding, which rounds an inexact mantissa to an odd value.

The existing *SVE* FCVT instructions implement the corresponding widening and narrowing conversions on the bottom or even-numbered half-width elements.

The following are the *SVE2* floating-point extra conversion instructions:

- FCVTLT. Floating-point up convert long (top, predicated).
- FCVTNT. Floating-point down convert narrow (top, predicated).
- FCVTX. Floating-point down convert, rounding to odd (predicated).
- FCVTXNT. Floating-point down convert, rounding to odd (top, predicated).

5.3.14 Floating-point widening multiply-accumulate

I_{FYLWD} The floating-point widening multiply-accumulate instructions multiply the even-numbered or odd-numbered half-precision elements of the two source vectors and then destructively add or subtract the single-precision intermediate products. Intermediate rounding is not done. The result is placed into the overlapping single-precision elements of the addend vector.

The instructions implicitly convert the half-precision inputs to single-precision and can be used to mitigate the impact of round-off errors when accumulating half-precision floating-point values over many iterations.

The instructions are unpredicated and preserve the multiplier and multiplicand source vectors.

The following are the *SVE2* floating-point widening multiply-accumulate instructions:

- FMLALB (vectors). Floating-point fused multiply-add long to accumulator (bottom).
- FMLALB (indexed). Floating-point fused multiply-add long to accumulator (bottom, indexed).
- FMLALT (vectors). Floating-point fused multiply-add long to accumulator (top).
- FMLALT (indexed). Floating-point fused multiply-add long to accumulator (top, indexed).
- FMLSLB (vectors). Floating-point fused multiply-subtract long from accumulator (bottom).

- FMLSLB (indexed). Floating-point fused multiply-subtract long from accumulator (bottom, indexed).
- FMLSLT (vectors). Floating-point fused multiply-subtract long from accumulator (top).
- FMLSLT (indexed). Floating-point fused multiply-subtract long from accumulator (top, indexed).

5.3.15 Floating-point integer binary logarithm

I_{CXRFN} The floating-point integer binary logarithm instruction returns the signed integer base 2 logarithm of each floating-point input element |x| after normalization.

The instruction produces the unbiased exponent of x used in the representation of the floating-point value. For positive x, x = significand × 2^{exponent}.

The integer results are placed in elements of the destination vector, which have the same width as the floating-point input elements:

- If x is normal, the result is the base 2 logarithm of x.
- If x is subnormal, the result corresponds to the normalized representation.
- If x is infinite, the result is $2^{(\text{esize-1})}$-1.
- If x is ±0.0 or NaN, the result is -$2^{(\text{esize-1})}$.

Inactive elements in the destination vector register are not modified.

The following is the *SVE2* floating-point integer binary logarithm instruction:

- FLOGB. Floating-point base 2 logarithm as integer.

5.3.16 Cross-lane match detect

I$_{\text{XMDQH}}$ This section includes instructions that detect or count matching elements within another vector, or within a 128-bit vector segment.

5.3.16.1 Vector Histogram Count

I$_{\text{KRLFR}}$ The vector histogram count instructions create vector histograms.

- HISTSEG compares each 8-bit byte element in the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector. The instruction counts the matching elements and places the result in the corresponding destination vector element. The instruction is unpredicated.
- HISTCNT compares each active 32-bit or 64-bit element in the first source vector with all elements in the second source vector that have an element number less than or equal to the *Active element* in the first source vector. The number of matching elements is counted and the result is placed in the corresponding destination vector element. *Inactive elements* in the destination vector are set to zero. *Inactive elements* in the second source vector do not cause a match.

The following are the *SVE2* vector histogram count instructions:

- HISTCNT. Count matching elements in vector.
- HISTSEG. Count matching elements in vector segments.

5.3.16.2 Character match

I$_{\text{ZQYVT}}$ The character match instructions can be used to scan each 128-bit segment of the second source vector for an 8-bit or 16-bit character string from the first source vector.

The MATCH and NMATCH instructions compare each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. When the first source character matches any (`MATCH`) or does not match any (`NMATCH`) character in the second segment, a true value is placed in the corresponding destination predicate element, otherwise a false value is placed in the destination predicate element. *Inactive elements* in the destination predicate register are set to zero. The instruction sets the First (N), None (Z), !Last (C) condition flags based on the predicate result, and sets the V flag to zero.

The following are the *SVE2* character match instructions:

- MATCH. Detect any matching elements, setting the condition flags.
- NMATCH. Detect no matching elements, setting the condition flags.

5.3.16.3 Contiguous conflict detection

I_{QDLRX} The contiguous conflict detection instructions check two addresses for a conflict or overlap between address ranges that could result in a loop-carried dependency through memory. The address range has the form [addr,addr+*VL*÷8], where *VL* is the accessible vector length in bits. A conflict can occur when contiguous load and store instructions use these addresses within the same loop iteration.

The instructions generate a predicate with elements that are true when the addresses cannot conflict within the same iteration, and false thereafter. The instructions set the First (N), None (Z), !Last (C) condition flags based on the predicate result, and the V flag is set to zero.

The following are the *SVE2* contiguous conflict detection instructions:

- WHILERW. While free of read-after-write conflicts.
- WHILEWR. While free of write-after-read/write conflicts.

5.3.17 Bit permutation

I_{HBNNG} The bit permutation instructions are optional. The bit permutation instructions are configured by the ID_AA64ZFR0_EL1.BitPerm bit. The instructions can be used to scatter, gather, or separate a set of bits within each first source vector element under the control of a bit mask or sieve in the corresponding second source vector elements. The instructions are unpredicated.

The BDEP instruction scatters the lowest-numbered contiguous bits within each first source vector element to the bit positions that are indicated by non-zero bits in the corresponding mask element of the second source vector. The order of the bits is preserved. The bits corresponding to a zero mask bit are set to zero.

The BEXT instruction gathers bits in each first source vector element from the bit positions that are indicated by non-zero bits in the corresponding mask element of the second source vector. The bits are gathered to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order. The remaining higher-numbered bits are set to zero.

The BGRP instruction selects bits from each first source vector element and groups them into the corresponding destination element, using a corresponding mask element in the second source vector, as follows:

- The non-zero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the lowest-numbered contiguous bits of the destination element.
- The zero bits in the mask element select the bit positions from the corresponding first source vector element. The selected bits are gathered into the highest-numbered contiguous bits of the destination element.

The bit order within each group is preserved.

The following are the optional *SVE2* bit permutation instructions:

- BDEP. Scatter lower bits into positions selected by bitmask.
- BEXT. Gather lower bits from positions selected by bitmask.
- BGRP. Group bits to right or left as selected by bitmask.

5.3.18 Polynomial arithmetic

I_{HYNTW} The polynomial arithmetic instructions support polynomial arithmetic over [0, 1], where exclusive-OR takes the place of addition. The instructions can be used in applications such as CRC calculations, AES-GCM, elliptic curve cryptography, Diffie-Hellman key exchange, and others.

The PMUL and widening PMULL instructions perform a polynomial multiplication over [0, 1]. The PMULL instructions read the source operands from either the even-numbered (bottom) or odd-numbered (top) narrow elements. Each double-width result is placed in the destination elements that overlap the narrow source elements.

The interleaving bitwise exclusive-OR instructions operate on the even-numbered (bottom) elements of the first source vector register and the odd-numbered (top) elements of the second source vector register. The result is either placed in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged, or placed in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged.

These instructions are unpredicated.

The following are the *SVE2* polynomial arithmetic instructions:

- EORBT. Interleaving exclusive-OR (bottom, top).
- EORTB. Interleaving exclusive-OR (top, bottom).
- PMUL. Polynomial multiply vectors (unpredicated).
- PMULLB. Polynomial multiply long (bottom).
- PMULLT. Polynomial multiply long (top).

5.3.19 Vector concatenation

I_{FKWXL} The vector concatenation instructions have new constructive versions that are introduced in SVE2 that preserve both of the source operands. In the constructive versions of the instruction, only the first source vector register number is encoded, which requires the source vectors to be in consecutively numbered registers (modulo 32).

The following are the SVE2 vector concatenation instructions:

- EXT. Extract vector from pair of vectors.
- SPLICE. Splice two vectors under predicate control.

5.3.20 Extended table lookup/permute

I_{PHTRC} The *SVE2* extended table lookup instructions, TBL and TBX enable the construction of table lookups or programmable vector permutes where the table consists of two or more vector registers.

Because the index values can select any element in a vector, the instructions are not naturally vector length agnostic.

The following are the *SVE2* extended table lookup instructions:

- TBL. Programmable table lookup in two vector table (zeroing).
- TBX. Programmable table lookup in single vector table (merging).

5.3.21 Non-temporal gather/scatter

I_{RSDSD} The non-temporal gather load and scatter store instructions provide a hint to the memory system that the data structure being accessed has a low reuse frequency. The memory system can use the hint to avoid retaining the data or evicting more frequently-used data from the caches.

These instructions support a single addressing mode consisting of 64-bit or 32-bit vector base addresses plus an unscaled 64-bit scalar offset that defaults to the zero register (XZR). Other addressing modes can be constructed using extra instructions.

The following are the *SVE2* non-temporal gather load and scatter store instructions:

- LDNT1B (vector plus scalar). Gather load non-temporal unsigned bytes.

- LDNT1D (vector plus scalar). Gather load non-temporal unsigned doublewords.
- LDNT1H (vector plus scalar). Gather load non-temporal unsigned halfwords.
- LDNT1SB. Gather load non-temporal signed bytes.
- LDNT1SH. Gather load non-temporal signed halfwords.
- LDNT1SW. Gather load non-temporal signed words.
- LDNT1W (vector plus scalar). Gather load non-temporal unsigned words.
- STNT1B (vector plus scalar). Scatter store non-temporal bytes.
- STNT1D (vector plus scalar). Scatter store non-temporal doublewords.
- STNT1H (vector plus scalar). Scatter store non-temporal halfwords.
- STNT1W (vector plus scalar). Scatter store non-temporal words.

5.3.22 Cryptography support

I_{YKKMY} Implementation of cryptography acceleration instructions is optional and controlled by the ID_AA64ZFR0_EL1.{SM4, SHA3, AES} bit fields. Implementation of the instructions requires consistency is maintained with the existing Armv8 cryptographic functionality support, as follows:

- If none of the *SVE2* cryptographic instructions are implemented, then the Armv8 AES, SHA1, and SHA256 instructions and the Armv8.4 SHA512, SHA3, SM3, and SM4 instructions can be implemented.
- If the *SVE2* SHA3 instructions are implemented, then implementation of the Armv8.4 SHA3 instructions is required.
- If the *SVE2* SM4 instructions are implemented, then implementation of the Armv8.4 SM4 instructions is required, but implementing any of the following instructions is optional:
  - The Armv8 AES, SHA1, and SHA256 instructions.
  - The Armv8.4 SHA512 and SHA3 instructions.
- If the *SVE2* AES instructions are implemented, then implementation of the Armv8 AES instructions is required, but implementing any of the Armv8 SHA256, SHA512, SHA3, SM3, and SM4 instructions is optional.
- If all of the *SVE2* cryptographic instructions are implemented, then implementation of the equivalent Armv8 and Armv8.4 instructions is required.

5.3.22.1 AES-128 instructions

I_{FYSXP} AES-128 is a 128-bit block cipher that is computed using a combination of linear XOR operations, the use of rotations by fixed values, and a set of 8-bit non-linear substitutions.

The following instructions accelerate a single encryption round:

- The AESE instruction reads a 16-byte state array from each 128-bit segment of the first source vector and a round key from the corresponding 128-bit segment of the second source vector. A single round of the *AddRoundKey()*, *SubBytes()*, and *ShiftRows()* transformations, in accordance with the AES standard, is applied to each state array.
- The AESMC instruction reads a 16-byte state array from each 128-bit segment of the source register and performs a single round of the *MixColumns()* transformation on each state array in accordance with the AES standard.

The following instructions accelerate a single decryption round:

- The AESD instruction reads a 16-byte state array from each 128-bit segment of the first source vector and a round key from the corresponding 128-bit segment of the second source vector. A single round of the *AddRoundKey()*, *InvSubBytes()*, and *InvShiftRows()* transformations in accordance with the AES standard, is applied to each state array.
- The AESIMC instruction reads a 16-byte state array from each 128-bit segment of the source register and performs a single round of the *InvMixColumns()* transformation on each state array in accordance with the AES standard.

Each updated state array is destructively placed in the corresponding segment of the first source vector. The AES instructions are unpredicated.

The following are the *SVE2* AES-128 instructions:

- AESD. AES single round decryption.
- AESE. AES single round encryption.
- AESIMC. AES inverse mix columns.
- AESMC. AES mix columns.

5.3.22.2 SHA-3 instructions

I TFMTS The SHA-3 instructions accelerate the SHA-3 hash algorithm.

The SHA-3 hash is based on a running digest of 1600 bits, arranged as a five by five array of 64-bit values. The instructions map the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is done on these registers during a round of the SHA-3 hash calculation.

Two or more parallel SHA-3 hash calculations are combined as a SIMD operation, where one calculation operates on the 0th 64-bit element of each vector, and the other calculation operates on the first 64-bit element of each vector. The SIMD operation is useful for the fast parallel hash algorithm recently introduced into the SHA-3 standard that allows a single input stream to be computed using multiple SHA-3 hashes in parallel.

The SHA3 instructions are unpredicated.

The only specialized *SVE2* SHA-3 instruction is RAX1.

See also:

- *5.3.9 Bitwise ternary logical instructions*

5.3.22.3 SM4 instructions

I MRLTX SM4 is the standard Chinese symmetric encryption algorithm which can be accelerated using a similar approach to that used for AES.

SM4 is a 128-bit wide block cipher that is computed using a combination of linear XOR operations, the use of fixed-value rotations, and a set of 8-bit non-linear substitutions.

- The SM4E instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, and four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each input data block is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector.
- The SM4EKEY instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, and four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector.

The SM4 instructions are unpredicated.

The following are the *SVE2* SM4 instructions:

- SM4E. SM4 encryption and decryption.
- SM4EKEY. SM4 key updates.

Chapter 6
SVE Debug

6.1 Self-hosted debug

I_{XLMZB} *SVE* extends the AArch64 self-hosted debug exception model.

6.1.1 SVE Watchpoint exceptions

R_{ZRWTZ} For *SVE* predicated vector load or store instructions which are not First-fault vector loads or Non-fault vector loads, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element*, a *Watchpoint debug event* is generated.

R_{GLRCD} For *SVE* predicated vector load or store instructions, if the instruction performs an access due to an *Inactive element*, a *Watchpoint debug event* is not generated.

R_{DYGMS} For *SVE* Non-fault vector load instructions, when the instruction performs an access, a *Watchpoint debug event* is not generated.

R_{LKKQG} For *SVE* Non-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element*, the access is reported in the FFR.

R_{TLSCX} For *SVE* First-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to the *First active element*, a *Watchpoint event* is generated.

R_{XBBLW} For *SVE* First-fault vector load instructions, when the instruction performs a non-speculative single-copy atomic access matching a configured watchpoint due to an *Active element* that is not the *First active element*, a *Watchpoint debug event* is not generated and the access is reported in the FFR.

R_{CKZFP} Watchpoints are not a mechanism for preventing access to memory.

I$_{ZHXGG}$ For *SVE* Non-fault and First-fault vector load instructions that do not generate a *Watchpoint debug event*, an access that matches a configured watchpoint can return the data and set the appropriate FFR elements to FALSE.

See also:

- 3.1.1 *Synchronous memory faults*

6.1.2 MOVPRFX instruction behavior in self-hosted debug

I$_{JZVZB}$ A MOVPRFX instruction can legally prefix a BRK or HLT instruction.

R$_{BHPGY}$ If a hardware breakpoint is programmed with the address of a legal MOVPRFX instruction, when any of the following events occur, the hardware breakpoint generates a Breakpoint exception:

- The MOVPRFX instruction is committed for execution.
- The combined MOVPRFX and Prefixed instruction is committed for execution.

R$_{WLYTJ}$ If a hardware breakpoint is programmed with the address of an illegal MOVPRFX instruction or a Prefixed instruction, when any of the MOVPRFX instruction and Prefixed instruction are committed for execution, it is CONSTRAINED UNPREDICTABLE whether or not the hardware breakpoint generates a Breakpoint exception.

R$_{BMYNV}$ If a single-step is performed for a MOVPRFX instruction, it is CONSTRAINED UNPREDICTABLE whether the *PE* steps over the pair of instructions or steps over only the MOVPRFX instruction.

See also:

- 3.1 *Exception model*
- 5.2.7.5 *Move prefix*

6.2 External debug

R_{NFJLQ} If the *PE* is in *Debug state*, the *SVE* architectural state can be accessed.

See also:

- *External Debug* in the *ARM® Architecture Reference Manual, ARMv8-A, for ARMv8-A architecture profile*

6.2.1 Instructions in Debug state

R_{TGSHS} If the *PE* is in *Debug state*, all of the following apply:

- The following *SVE* instructions have the same behavior as in Non-debug state:
  - RDVL.
  - CPY (immediate, zeroing) with byte element size and a shift amount of 0.
  - PTRUE with ALL constraint and byte element size.
  - RDFFR (unpredicated).
  - WRFFR.
  - EXT.
  - INSR (scalar).
  - DUP (scalar).
- CMPNE (immediate) with byte element size has the same behavior as in Non-debug state, but also sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values.
- For *SVE* instructions not listed in the first two bullets, their behavior is CONSTRAINED UNPREDICTABLE. The behaviors which can occur are:
  - The instruction generates an Undefined Instruction exception.
  - The instruction executes as a NOP.
  - If the instruction modifies PSTATE, it sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values.
  - If the instruction reads PSTATE condition flags, it uses an UNKNOWN value for the condition flag.
  - The instruction has the same behavior as in Non-debug state.

Chapter 7
SVE Performance Monitor Usage

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

7.1 Interesting combinations of SVE events

7.1.1 Scalar-equivalent operations

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

FP_SCALE_OPS_SPEC × VL ÷ 128 + FP_FIXED_OPS_SPEC

A summary of these event pairs is given below. Combined multiply-add and multiply-subtract instructions are counted as two operations per element.

Operation type	Scalable operations	Fixed width operations
Floating-point operations (any precision)	FP_SCALE_OPS_SPEC	FP_FIXED_OPS_SPEC
Half-precision floating-point operations	FP_HP_SCALE_OPS_SPEC	FP_HP_FIXED_OPS_SPEC
Single-precision floating-point operations	FP_SP_SCALE_OPS_SPEC	FP_SP_FIXED_OPS_SPEC
Double-precision floating-point operation	FP_DP_SCALE_OPS_SPEC	FP_DP_FIXED_OPS_SPEC
Integer operations (any size)	INT_SCALE_OPS_SPEC	INT_FIXED_OPS_SPEC
Load/store accesses (any size)	LDST_SCALE_OPS_SPEC	LDST_FIXED_OPS_SPEC
Load accesses (any size)	LD_SCALE_OPS_SPEC	LD_FIXED_OPS_SPEC
Store accesses (any size)	ST_SCALE_OPS_SPEC	ST_FIXED_OPS_SPEC

7.1.2 Bytes loaded and stored

I_{NYJRH} The number of bytes speculatively loaded from memory or stored to memory, assuming that all *SVE* vector elements are active, can be determined from a pair of event counters. For example, the total number of bytes loaded from memory can be computed as follows:

LD_SCALE_BYTES_SPEC × VL ÷ 128 + LD_FIXED_BYTES_SPEC

A summary of the total byte count pairs is as follows:

Operation type	Scalable operations	Fixed width operations
Load/store byte count	LDST_SCALE_BYTES_SPEC	LDST_FIXED_BYTES_SPEC
Load byte count	LD_SCALE_BYTES_SPEC	LD_FIXED_BYTES_SPEC
Store byte count	ST_SCALE_BYTES_SPEC	ST_FIXED_BYTES_SPEC

7.1.3 Overall vector utilization

I_{MVTZM} Vector utilization rates for *SVE* events which ignore the number of Active elements can be estimated by adjusting them using the following ratios:

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

7.1.4 Vector loop efficiency

I_{MXYQS} The effectiveness with which sequential or scalar source loops are vectorized can be estimated using ratios of the SVE_PLOOP_*_SPEC predicated loop events, as shown in the following table:

Vector loop metric	Ratio
Source level iterations per loop	SVE_PLOOP_ELTS_SPEC ÷ SVE_PLOOP_TERM_SPEC
Vectorized iterations per loop	SVE_PLOOP_TEST_SPEC ÷ SVE_PLOOP_TERM_SPEC
Parallelism per vector loop	SVE_PLOOP_ELTS_SPEC ÷ SVE_PLOOP_TEST_SPEC

Chapter 8
SVE instruction categories

I_{HDLNY} The lists in this chapter include only *SVE* instructions, they do not include *SVE2* instructions.

8.1 Data movement instructions

8.1.1 Data movement (scalar)

I$_{GGCPB}$ All of the following are data movement (scalar) instructions:

- FCSEL.
- FMOV (scalar, immediate).
- FMOV (general).
- FMOV (register).

8.1.2 Data movement (Advanced SIMD)

I$_{BMXMV}$ All of the following are data movement (*Advanced SIMD*) instructions:

- DUP (element).
- DUP (general).
- EXT.
- FMOV (vector, immediate).
- INS (element).
- INS (general).
- SMOV.
- TBL.
- TBX.
- TRN1.
- TRN2.
- UMOV.
- UZP1.
- UZP2.
- XTN, XTN2.
- ZIP1.
- ZIP2.

8.1.3 Data movement (SVE)

I$_{ZFWZB}$ All of the following are data movement (*SVE*) instructions:

- CLASTA (scalar).
- CLASTA (SIMD&FP scalar).
- CLASTA (vectors).
- CLASTB (scalar).
- CLASTB (SIMD&FP scalar).
- CLASTB (vectors).
- COMPACT.
- CPY (scalar).
- CPY (immediate).
- DUP (scalar).
- DUP (immediate).
- EXT.
- FCPY.
- FDUP.
- INDEX (immediate, scalar).
- INDEX (immediates).

- INDEX (scalar, immediate).
- INDEX (scalars).
- INSR (scalar).
- INSR (SIMD&FP scalar).
- LASTA (scalar).
- LASTA (SIMD&FP scalar).
- LASTB (scalar).
- LASTB (SIMD&FP scalar).
- MOVPRFX (predicated).
- MOVPRFX (unpredicated).
- REV (vector).
- SEL (vectors).
- SPLICE.
- SUNPKHI, SUNPKLO.
- TBL.
- TRN1, TRN2 (vectors)
- UUNPKHI, UUNPKLO.
- UZP1, UZP2 (vectors).
- ZIP1, ZIP2 (vectors).

8.2 Integer instructions

8.2.1 Integer (scalar)

8.2.1.1 Integer uniform arithmetic (scalar)

I_{HMLWT} All of the following are integer uniform arithmetic (scalar) instructions:

- ADC.
- ADCS.
- ADD (extended register).
- ADD (immediate).
- ADD (shifted register).
- ADDS (extended register).
- ADDS (immediate).
- ADDS (shifted register).
- CCMN (immediate).
- CCMN (register).
- CCMP (immediate).
- CCMP (register).
- CSINC.
- CSINV.
- CSNEG.
- MADD.
- MSUB.
- SBC.
- SBCS.
- SDIV.
- UDIV.
- SMULH.
- UMULH.
- SUB (extended register).
- SUB (immediate).
- SUB (shifted register).
- SUBS (extended register).
- SUBS (immediate).
- SUBS (shifted register).
- ADR.
- ADRP.

8.2.1.2 Integer widening arithmetic

I_{ZNGLP} All of the following are integer widening arithmetic instructions:

- SMADDL.
- SMSUBL.
- UMADDL.
- UMSUBL.

8.2.1.3 Integer bitwise operations (scalar)

I_{GTHXC} All of the following are integer bitwise operations (scalar) instructions:

- AND (immediate)

- AND (shifted register)
- ANDS (immediate)
- ANDS (shifted register)
- BIC (shifted register).
- BICS (shifted register).
- EOR (immediate).
- EOR (shifted register).
- EON (shifted register).
- ORR (immediate).
- ORR (shifted register).
- ORN (shifted register).
- ASRV.
- LSLV.
- LSRV.
- RORV.
- BFM.
- SBFM.
- UBFM.
- CLS.
- CLZ.
- EXTR.
- RBIT.
- REV.
- REV16.
- REV32.

8.2.2 Integer (Advanced SIMD)

8.2.2.1 Integer uniform arithmetic (Advanced SIMD)

I_{QBHGF} All of the following are integer uniform arithmetic (*Advanced SIMD*) instructions:

- ABS.
- NEG (vector).
- ADD (vector).
- SUB (vector).
- MLA (by element).
- MLA (vector).
- MLS (by element).
- MLS (vector).
- MUL (by element).
- MUL (vector).
- PMUL.
- SABA.
- UABA.
- SABD.
- UABD.
- SDOT (by element).
- SDOT (vector).
- UDOT (by element).
- UDOT (vector).
- SHADD.
- SHSUB.
- SRHADD.

- UHADD.
- UHSUB.
- URHADD.
- SMAX.
- SMIN.
- UMAX.
- UMIN.
- SQABS.
- SQNEG.
- SQADD.
- SQSUB.
- SUQADD.
- UQADD.
- USQADD.
- UQSUB.
- SQDMULH (by element).
- SQDMULH (vector).
- SQRDMULH (by element).
- SQRDMULH (vector).
- SQRDMLAH (by element).
- SQRDMLAH (vector).
- SQRDMLSH (by element).
- SQRDMLSH (vector).
- URECPE.
- URSQRTE.
- USRA.

8.2.2.2 Integer widening arithmetic (Advanced SIMD)

$\mathrm{I}_{\mathrm{CGVJC}}$ All of the following are integer widening arithmetic (*Advanced SIMD*) instructions:

- SABAL, SABAL2.
- UABAL, UABAL2.
- SABDL, SABDL2.
- UABDL, UABDL2.
- SADDL, SADDL2.
- UADDL, UADDL2.
- SADDW, SADDW2.
- UADDW, UADDW2.
- SMLAL, SMLAL2 (by element).
- SMLAL, SMLAL2 (vector).
- UMLAL, UMLAL2 (by element).
- UMLAL, UMLAL2 (vector).
- SMLSL, SMLSL2 (by element).
- SMLSL, SMLSL2 (vector).
- UMLSL, UMLSL2 (by element).
- UMLSL, UMLSL2 (vector).
- SMULL, SMULL2 (by element).
- SMULL, SMULL2 (vector).
- UMULL, UMULL2 (by element).
- UMULL, UMULL2 (vector).
- PMULL, PMULL2.
- SQDMULL, SQDMULL2 (by element).
- SQDMULL, SQDMULL2 (vector).
- SQDMLAL, SQDMLAL2 (by element).

- SQDMLAL, SQDMLAL2 (vector).
- SQDMLSL, SQDMLSL2 (by element).
- SQDMLSL, SQDMLSL2 (vector).
- SHLL, SHLL2.
- SSHLL, SSHLL2.
- USHLL, USHLL2.
- SSUBL, SSUBL2.
- USUBL, USUBL2.
- SSUBW, SSUBW2.
- USUBW, USUBW2.
- UXTL, UXTL2.

8.2.2.3 Integer narrowing arithmetic (Advanced SIMD)

I_{YFSMM} All of the following are integer narrowing arithmetic (*Advanced SIMD*) instructions:

- ADDHN, ADDHN2.
- RADDHN, RADDHN2.
- SUBHN, SUBHN2.
- RSUBHN, RSUBHN2.
- SHRN, SHRN2.
- RSHRN, RSHRN2.
- SQSHRN, SQSHRN2.
- SQSHRUN, SQSHRUN2.
- UQRSHRN, UQRSHRN2.
- UQSHRN, UQSHRN2.
- SQXTN, SQXTN2.
- SQXTUN, SQXTUN2.
- UQXTN, UQXTN2.

8.2.2.4 Integer bitwise operations (Advanced SIMD)

I_{DCGLC} All of the following are integer bitwise operations (*Advanced SIMD*) instructions:

- AND (vector).
- BIC (vector, immediate).
- BIC (vector, register).
- EOR (vector).
- ORN (vector).
- ORR (vector, immediate).
- ORR (vector, register).
- BIF.
- BIT.
- BSL.
- CLS (vector).
- CLZ (vector).
- CNT.
- MOVI.
- MVNI.
- NOT.
- RBIT (vector).
- REV16 (vector).
- REV32 (vector).
- REV64.
- SHL.

- SRSHL.
- URSHL.
- SRSHR.
- URSHR.
- SRSRA.
- SSRA.
- URSRA.
- SLI.
- SRI.
- SQRSHL.
- SQSHL (register).
- SQSHLU.
- UQRSHL.
- UQSHL (immediate).
- UQSHL (register).
- SSHL.
- USHL.
- SSHR.
- USHR.

8.2.2.5 Integer comparisons (Advanced SIMD)

I_{TVYLX} All of the following are integer comparisons (*Advanced SIMD*) instructions:

- CMEQ (register).
- CMEQ (zero).
- CMGE (register).
- CMGE (zero).
- CMGT (register).
- CMGT (zero).
- CMHI (register).
- CMHS (register).
- CMLE (zero).
- CMLT (zero).
- CMTST.

8.2.2.6 Integer reductions (Advanced SIMD)

I_{SNQKR} All of the following are integer reductions (*Advanced SIMD*) instructions:

- ADDP (scalar).
- ADDP (vector).
- ADDV (vector).
- SADALP.
- UADALP.
- SADDLP.
- SADDLV.
- UADDLP.
- UADDLV.
- SMAXP.
- SMAXV.
- UMAXP.
- UMAXV.
- SMINP.
- SMINV.

- UMINP.
- UMINV.

8.2.3 Integer (SVE)

8.2.3.1 Integer uniform arithmetic (SVE)

$\texttt{I}_{\texttt{GMVWX}}$ All of the following are integer uniform arithmetic (*SVE*) instructions:

- ABS.
- NEG.
- ADD (immediate).
- ADD (vectors, predicated).
- ADD (vectors, unpredicated).
- SUB (immediate).
- SUB (vectors, predicated).
- SUB (vectors, unpredicated).
- SUBR (immediate).
- SUBR (vectors).
- ADR.
- CNOT.
- MAD.
- MSB.
- MLA (indexed).
- MLA (vectors).
- MLS (indexed).
- MLS (vectors).
- MUL (immediate).
- MUL (indexed).
- MUL (vectors, predicated).
- MUL (vectors, unpredicated).
- SABD.
- UABD.
- SDIV.
- SDIVR.
- UDIV.
- UDIVR.
- SDOT (indexed).
- SDOT (vectors).
- UDOT (indexed).
- UDOT (vectors).
- SMAX (immediate).
- SMAX (vectors).
- SMIN (immediate).
- SMIN (vectors).
- UMAX (immediate).
- UMAX (vectors).
- UMIN (immediate).
- UMIN (vectors).
- SMULH (predicated).
- SMULH (unpredicated).
- UMULH (predicated).
- UMULH (unpredicated).
- SQADD (immediate).

- SQADD (vectors, predicated).
- SQADD (vectors, unpredicated).
- SQSUB (immediate).
- SQSUB (vectors, predicated).
- SQSUB (vectors, unpredicated).
- UQADD (immediate).
- UQADD (vectors, predicated).
- UQADD (vectors, unpredicated).
- UQSUB (immediate).
- UQSUB (vectors, predicated).
- UQSUB (vectors, unpredicated).
- SXTB, SXTH, SXTW.
- UXTB, UXTH, UXTW.

8.2.3.2 Integer bitwise operations (SVE)

I_{PNLPF} All of the following are integer bitwise operations (*SVE*) instructions:

- AND (vectors, predicated).
- AND (vectors, unpredicated).
- BIC (vectors, predicated).
- BIC (vectors, unpredicated).
- EON.
- EOR (vectors, predicated).
- EOR (vectors, unpredicated).
- ORN.
- ORR (vectors, predicated).
- ORR (vectors, unpredicated).
- ASR (immediate, predicated).
- ASR (immediate, unpredicated).
- ASR (vectors).
- ASR (wide elements, predicated).
- ASR (wide elements, unpredicated).
- ASRR.
- ASRD.
- CLS.
- CLZ.
- CNT.
- DUPM.
- LSL (immediate, predicated).
- LSL (immediate, unpredicated).
- LSL (vectors).
- LSL (wide elements, predicated).
- LSL (wide elements, unpredicated).
- LSLR.
- LSR (immediate, predicated).
- LSR (immediate, unpredicated).
- LSR (vectors).
- LSR (wide elements, predicated).
- LSR (wide elements, unpredicated).
- LSRR.
- NOT (vector).
- RBIT.
- REVB, REVH, REVW.

8.2.3.3 Integer comparisons (SVE)

I$_{\text{VYSCC}}$ All of the following are integer comparisons (*SVE*) instructions:

- CMP<cc> (immediate).
- CMP<cc> (vectors).
- CMP<cc> (wide elements).

8.2.3.4 Integer reductions (SVE)

I$_{\text{THTPR}}$ All of the following are integer reductions (*SVE*) instructions:

- ANDV.
- EORV.
- ORV.
- SADDV.
- UADDV.
- SMAXV.
- UMAXV.
- SMINV.
- UMINV.

8.2.3.5 Element count and increment vector (SVE)

I$_{\text{MPWZH}}$ All of the following are element count and increment vector (*SVE*) instructions:

- DECD, DECH, DECW (vector).
- INCD, INCH, INCW (vector).
- SQDECH (vector).
- SQDECW (vector).
- SQDECD (vector).
- SQINCH (vector).
- SQINCW (vector).
- SQINCD (vector).
- UQDECH (vector).
- UQDECW (vector).
- UQDECD (vector).
- UQINCH (vector).
- UQINCW (vector).
- UQINCD (vector).

8.3 Floating-point instructions

8.3.1 Floating-point (scalar)

8.3.1.1 Floating-point arithmetic (scalar)

I_{WLHZN} All of the following are floating-point arithmetic (scalar) instructions:

- FADD (scalar).
- FSUB (scalar).
- FDIV (scalar).
- FMADD.
- FMSUB.
- FNMADD.
- FNMSUB.
- FMUL (scalar).
- FNMUL (scalar).
- FSQRT (scalar).

8.3.1.2 Floating-point miscellaneous (scalar)

I_{DMZTD} All of the following are floating-point miscellaneous (scalar) instructions:

- FMAX (scalar).
- FMAXNM (scalar).
- FMIN (scalar).
- FMINNM (scalar).
- FRINTA (scalar).
- FRINTI (scalar).
- FRINTM (scalar).
- FRINTN (scalar).
- FRINTP (scalar).
- FRINTX (scalar).
- FRINTZ (scalar).

8.3.1.3 Floating-point comparisons (scalar)

I_{QYXGH} All of the following are floating-point comparisons (scalar) instructions:

- FCMP.
- FCMPE.

8.3.2 Floating-point (Advanced SIMD)

8.3.2.1 Floating-point arithmetic (Advanced SIMD)

I_{PTKPC} All of the following are floating-point arithmetic (*Advanced SIMD*) instructions:

- FABD.
- FADD (vector).
- FSUB (vector).
- FCADD.
- FCMLA.
- FCMLA (by element).

- FDIV (vector).
- FMLA (by element).
- FMLA (vector).
- FMLS (by element).
- FMLS (vector).
- FMUL (by element).
- FMUL (vector).
- FMULX.
- FMULX (by element).
- FRECPS.
- FRSQRTS.
- FSQRT (vector).

8.3.2.2 Floating-point miscellaneous (Advanced SIMD)

I GJCYK All of the following are floating-point miscellaneous (*Advanced SIMD*) instructions:

- FMAX (vector).
- FMAXNM (vector).
- FMIN (vector).
- FMINNM (vector).
- FRECPX.
- FRINTA (vector).
- FRINTI (vector).
- FRINTM (vector).
- FRINTN (vector).
- FRINTP (vector).
- FRINTX (vector).
- FRINTZ (vector).

8.3.2.3 Floating-point comparisons (Advanced SIMD)

I XVJNF All of the following are floating-point comparisons (*Advanced SIMD*) instructions:

- FACGE.
- FACGT.
- FCMEQ (register).
- FCMEQ (zero).
- FCMGE (register).
- FCMGE (zero).
- FCMGT (register).
- FCMGT (zero).
- FCMLE (zero).
- FCMLT (zero).

8.3.2.4 Floating-point reductions (Advanced SIMD)

I FNNRS All of the following are floating-point reductions (*Advanced SIMD*) instructions:

- FADDP (scalar).
- FADDP (vector).
- FMAXNMP (scalar).
- FMAXNMP (vector).
- FMAXP (scalar).
- FMAXP (vector).
- FMAXNMV.

- FMAXV.
- FMINNMP (scalar).
- FMINNMP (vector).
- FMINP (scalar).
- FMINP (vector).
- FMINNMV.
- FMINV.

8.3.3 Floating-point (SVE)

8.3.3.1 Floating-point arithmetic (SVE)

$\mathtt{I}_{\mathtt{BGBSK}}$ All of the following are floating-point arithmetic (*SVE*) instructions:

- FABD.
- FADD (immediate).
- FADD (vectors, predicated).
- FADD (vectors, unpredicated).
- FSUB (immediate).
- FSUB (vectors, predicated).
- FSUB (vectors, unpredicated).
- FSUBR (immediate).
- FSUBR (vectors).
- FCADD.
- FCMLA (indexed).
- FCMLA (vectors).
- FDIV.
- FDIVR.
- FMAD.
- FNMAD.
- FNMSB.
- FMSB.
- FMLA (indexed).
- FMLA (vectors).
- FMLS (indexed).
- FMLS (vectors).
- FNMLA.
- FNMLS.
- FMUL (immediate).
- FMUL (indexed).
- FMUL (vectors, predicated).
- FMUL (vectors, unpredicated).
- FMULX.
- FRECPS.
- FRSQRTS.
- FSCALE.
- FSQRT.
- FTMAD.
- FTSMUL.

8.3.3.2 Floating-point miscellaneous (SVE)

$\mathtt{I}_{\mathtt{QHKWQ}}$ All of the following are floating-point miscellaneous (*SVE*) instructions:

- FMAX (immediate).

- FMAX (vectors).
- FMAXNM (immediate).
- FMAXNM (vectors).
- FMIN (immediate).
- FMIN (vectors).
- FMINNM (immediate).
- FMINNM (vectors).
- FRECPX.
- FRINTA.
- FRINTI.
- FRINTM.
- FRINTN.
- FRINTP.
- FRINTX.
- FRINTZ.

8.3.3.3 Floating-point comparisons (SVE)

I_{CCKRC} All of the following are floating-point comparisons (*SVE*) instructions:

- FACGE.
- FACGT.
- FACLE.
- FACLT.
- FCMEQ (vectors).
- FCMEQ (zero).
- FCMGE (vectors).
- FCMGE (zero).
- FCMGT (vectors).
- FCMGT (zero).
- FCMLE (vectors).
- FCMLE (zero).
- FCMGT (vectors).
- FCMGT (zero).
- FCMLT (vectors).
- FCMLT (zero).
- FCMNE (vectors).
- FCMNE (zero).
- FCMUO (vectors).

8.3.3.4 Floating-point reductions (SVE)

I_{RBLQR} All of the following are floating-point reductions (*SVE*) instructions:

- FADDA.
- FADDV.
- FMAXNMV.
- FMAXV.
- FMINNMV.
- FMINV.

8.4 Floating-point conversions

8.4.1 Float↔Float convert (scalar)

I_{YVFFQ} The following is a Floating-point convert (scalar) instruction:

- FCVT.

8.4.2 Float↔Float convert (Advanced SIMD)

I_{MFPKF} All of the following are Floating-point convert (Advanced SIMD) instructions:

- FCVTL, FCVTL2.
- FCVTN, FCVTN2.
- FCVTXN, FCVTXN2.

8.4.3 Float↔Float convert (SVE)

I_{WCKFQ} The following is a Floating-point convert (SVE) instruction:

- FCVT.

8.4.4 Float↔Int convert (scalar)

I_{PCMBH} All of the following are Floating-point integer convert (scalar) instructions:

- FCVTAS (scalar).
- FCVTMS (scalar).
- FCVTNS (scalar).
- FCVTPS (scalar).
- FCVTZS (scalar, fixed-point).
- FCVTZS (scalar, integer).
- FCVTAU (scalar).
- FCVTMU (scalar).
- FCVTNU (scalar).
- FCVTPU (scalar).
- FCVTZU (scalar, fixed-point).
- FCVTZU (scalar, integer).
- FJCVTZS.
- SCVTF (scalar, fixed-point).
- SCVTF (scalar, integer).
- UCVTF (scalar, fixed-point).
- UCVTF (scalar, integer).

8.4.5 Float↔Int convert (Advanced SIMD)

I_{SGHBH} All of the following are Floating-point integer convert (Advanced SIMD) instructions:

- FCVTAS (vector).
- FCVTMS (vector).
- FCVTNS (vector).
- FCVTPS (vector).

- FCVTZS (vector, fixed-point).
- FCVTZS (vector, integer).
- FCVTZS (vector, integer).
- FCVTAU (vector).
- FCVTMU (vector).
- FCVTNU (vector).
- FCVTPU (vector).
- FCVTZU (vector, fixed-point).
- FCVTZU (vector, integer).
- SCVTF (vector, fixed-point).
- SCVTF (vector, integer).
- UCVTF (vector, fixed-point).
- UCVTF (vector, integer)

8.4.6 Float↔Int convert (SVE)

I_{KDNBP} All of the following are Floating-point integer convert (SVE) instructions:

- FCVTZS.
- FCVTZU.
- SCVTF.
- UCVTF.

8.5 Floating-point or integer instructions

8.5.1 Floating-point or integer arithmetic (scalar)

I$_{YFHSZ}$ All of the following are Floating-point or integer arithmetic (scalar) instructions:

- FABS (scalar).
- FNEG (scalar).

8.5.2 Floating-point or integer arithmetic (Advanced SIMD)

I$_{JZJVJ}$ All of the following are Floating-point or integer arithmetic (Advanced SIMD) instructions:

- FABS (vector).
- FNEG (vector).
- FRECPE.
- FRSQRTE.

8.5.3 Floating-point or integer arithmetic (SVE)

I$_{XFWRK}$ All of the following are Floating-point or integer arithmetic (SVE) instructions:

- FABS.
- FNEG.
- FRECPE.
- FRSQRTE.
- FEXPA.
- FTSSEL.

8.6 Non-SIMD SVE instructions

8.6.1 Element count and increment scalar (SVE)

I~GVGGT~ All of the following are element count and increment scalar (*SVE*) instructions:

- ADDPL.
- ADDPL.
- RDVL.
- CNTB, CNTD, CNTH, CNTW.
- DECB, DECD, DECH, DECW (scalar).
- INCB, INCD, INCH, INCW (scalar).
- SQDECB.
- SQDECH (scalar).
- SQDECW (scalar).
- SQDECD (scalar).
- SQINCB.
- SQINCH (scalar).
- SQINCW (scalar).
- SQINCD (scalar).
- UQDECB.
- UQDECH (scalar).
- UQDECW (scalar).
- UQDECD (scalar).
- UQINCB.
- UQINCH (scalar).
- UQINCW (scalar).
- UQINCD (scalar).

8.6.2 Compare and terminate (SVE)

I~KHHTX~ The following is a compare and terminate (*SVE*) instruction:

- CTERMEQ, CTERMNE.

8.7 Predicate instructions

8.7.1 Predicate move (SVE)

I$_{HFNFD}$ All of the following are predicate move (*SVE*) instructions:

- PFALSE.
- PTRUE, PTRUES.
- PUNPKHI, PUNPKLO.
- RDFFR, RDFFRS (predicated).
- RDFFR (unpredicated).
- SETFFR.
- WRFFR.
- REV (predicate).
- SEL (predicates).
- TRN1, TRN2 (predicates).
- UZP1, UZP2 (predicates).
- ZIP1, ZIP2 (predicates).

8.7.2 Predicate counted loop (SVE)

I$_{YPLYS}$ All of the following are predicate counted loop (*SVE*) instructions:

- WHILELE.
- WHILELO.
- WHILELS.
- WHILELT.

8.7.3 Predicate bitwise logical operations (SVE)

I$_{JGDHG}$ All of the following are predicate bitwise logical operations (*SVE*) instructions:

- AND, ANDS (predicates).
- BIC, BICS (predicates).
- EOR, EORS (predicates).
- NAND, NANDS.
- NOR, NORS.
- NOT (predicate).
- NOTS.
- ORN, ORNS (predicates).
- ORR, ORRS (predicates).
- PTEST.

8.7.4 Predicate scan (SVE)

I$_{NKBWD}$ All of the following are predicate scan (*SVE*) instructions:

- BRKA, BRKAS.
- BRKB, BRKBS.
- BRKN, BRKNS.
- BRKPA, BRKPAS.
- BRKPB, BRKPBS.
- PFIRST.

8.7.5 Predicate count and increment scalar (SVE)

I$_{JRGKK}$ All of the following are predicate count and increment scalar (*SVE*) instructions:

- CNTP.
- DECP (scalar).
- INCP (scalar).
- SQDECP (scalar).
- SQINCP (scalar).
- UQDECP (scalar).
- UQINCP (scalar).

8.7.6 Predicate count and increment vector (SVE)

I$_{HQZKL}$ All of the following are predicate count and increment vector (*SVE*) instructions:

- DECP (vector).
- INCP (vector).
- SQDECP (vector).
- SQINCP (vector).
- UQDECP (vector).
- UQINCP (vector).

8.8 Cryptographic instructions

8.8.1 Cryptographic (Advanced SIMD)

I$_{GBYWL}$ All of the following are Cryptographic (Advanced SIMD) instructions:

- AESD.
- AESE.
- AESIMC.
- AESMC.
- SHA1C.
- SHA1H.
- SHA1M.
- SHA1P.
- SHA1SU0.
- SHA1SU1.
- SHA256H.
- SHA256H2.
- SHA256SU0.
- SHA256SU1.

8.9 Load/store/prefetch instructions

8.9.1 Load/store (Advanced SIMD and floating-point scalar)

8.9.1.1 Contiguous elements load/store (Advanced SIMD)

I_{VRXTK} All of the following are contiguous elements load/store (*Advanced SIMD*) instructions:

- LD1 (multiple structures).
- ST1 (multiple structures).

8.9.1.2 Contiguous structures load/store (Advanced SIMD)

I$_{LZRDQ}$ All of the following are contiguous structures load/store (*Advanced SIMD*) instructions:

- LD2 (multiple structures).
- LD3 (multiple structures).
- LD4 (multiple structures).
- ST2 (multiple structures).
- ST3 (multiple structures).
- ST4 (multiple structures).

8.9.1.3 Single element/structure load/store (Advanced SIMD)

I$_{KLWGR}$ All of the following are single element/structure load/store (*Advanced SIMD*) instructions:

- LD1 (single structure).
- LD2 (single structure).
- LD3 (single structure).
- LD4 (single structure).
- ST1 (single structure).
- ST2 (single structure).
- ST3 (single structure).
- ST4 (single structure).

8.9.1.4 Single element/structure replicating load (Advanced SIMD)

I$_{XQCVF}$ All of the following are single element/structure replicating load (*Advanced SIMD*) instructions:

- LD1R.
- LD2R.
- LD3R.
- LD4R.

8.9.1.5 Register load/store (Advanced SIMD and floating-point scalar)

I$_{LSJXF}$ All of the following are register load/store (*Advanced SIMD&FP* scalar) instructions:

- LDNP (SIMD&FP).
- LDP (SIMD&FP).
- LDR (immediate, SIMD&FP).
- LDR (literal, SIMD&FP).
- LDR (register, SIMD&FP).
- LDUR (SIMD&FP).
- STNP (SIMD&FP).

- STP (SIMD&FP).
- STR (immediate, SIMD&FP).
- STR (register, SIMD&FP).
- STUR (SIMD&FP).

8.9.2 Load/store/prefetch (SVE)

8.9.2.1 Contiguous elements load/store/prefetch (SVE)

I$_{NNZKF}$ All of the following are contiguous elements load/store/prefetch (*SVE*) instructions:

- LD1B (scalar plus immediate).
- LD1H (scalar plus immediate).
- LD1W (scalar plus immediate).
- LD1D (scalar plus immediate).
- LD1SB (scalar plus immediate).
- LD1SH (scalar plus immediate).
- LD1SW (scalar plus immediate).
- LD1B (scalar plus scalar).
- LD1H (scalar plus scalar).
- LD1W (scalar plus scalar).
- LD1D (scalar plus scalar).
- LD1SB (scalar plus scalar).
- LD1SH (scalar plus scalar).
- LD1SW (scalar plus scalar).
- LDFF1B (scalar plus scalar).
- LDFF1H (scalar plus scalar).
- LDFF1W (scalar plus scalar).
- LDFF1D (scalar plus scalar).
- LDFF1SB (scalar plus scalar).
- LDFF1SH (scalar plus scalar).
- LDFF1SW (scalar plus scalar).
- LDNF1B.
- LDNF1H.
- LDNF1W.
- LDNF1D.
- LDNF1SB.
- LDNF1SH.
- LDNT1B (scalar plus immediate).
- LDNT1H (scalar plus immediate).
- LDNT1W (scalar plus immediate).
- LDNT1D (scalar plus immediate).
- LDNT1B (scalar plus scalar).
- LDNT1H (scalar plus scalar).
- LDNT1W (scalar plus scalar).
- LDNT1D (scalar plus scalar).
- PRFB (scalar plus immediate).
- PRFH (scalar plus immediate).
- PRFW (scalar plus immediate).
- PRFH (scalar plus immediate).
- PRFB (scalar plus scalar).
- PRFH (scalar plus scalar).
- PRFW (scalar plus scalar).
- PRFD (scalar plus scalar).

- ST1B (scalar plus immediate).
- ST1H (scalar plus immediate).
- ST1W (scalar plus immediate).
- ST1D (scalar plus immediate).
- ST1B (scalar plus scalar).
- ST1H (scalar plus scalar).
- ST1W (scalar plus scalar).
- ST1D (scalar plus scalar).
- STNT1B (scalar plus immediate).
- STNT1H (scalar plus immediate).
- STNT1W (scalar plus immediate).
- STNT1D (scalar plus immediate).
- STNT1B (scalar plus scalar).
- STNT1H (scalar plus scalar).
- STNT1W (scalar plus scalar).
- STNT1D (scalar plus scalar).

8.9.2.2 Contiguous structures load/store (SVE)

I$_{QPNSV}$ All of the following are contiguous structures load/store (*SVE*) instructions:

- LD2B (scalar plus immediate).
- LD2H (scalar plus immediate).
- LD2W (scalar plus immediate).
- LD2D (scalar plus immediate).
- LD2B (scalar plus scalar).
- LD2H (scalar plus scalar).
- LD2W (scalar plus scalar).
- LD2D (scalar plus scalar).
- LD3B (scalar plus immediate).
- LD3H (scalar plus immediate).
- LD3W (scalar plus immediate).
- LD3D (scalar plus immediate).
- LD3B (scalar plus scalar).
- LD3H (scalar plus scalar).
- LD3W (scalar plus scalar).
- LD3D (scalar plus scalar).
- LD4B (scalar plus immediate).
- LD4H (scalar plus immediate).
- LD4W (scalar plus immediate).
- LD4D (scalar plus immediate).
- LD4B (scalar plus scalar).
- LD4H (scalar plus scalar).
- LD4W (scalar plus scalar).
- LD4D (scalar plus scalar).
- ST2B (scalar plus immediate).
- ST2H (scalar plus immediate).
- ST2W (scalar plus immediate).
- ST2D (scalar plus immediate).
- ST2B (scalar plus scalar).
- ST2H (scalar plus scalar).
- ST2W (scalar plus scalar).
- ST2D (scalar plus scalar).
- ST3B (scalar plus immediate).
- ST3H (scalar plus immediate).

- ST3W (scalar plus immediate).
- ST3D (scalar plus immediate).
- ST3B (scalar plus scalar).
- ST3H (scalar plus scalar).
- ST3W (scalar plus scalar).
- ST3D (scalar plus scalar).
- ST4B (scalar plus immediate).
- ST4H (scalar plus immediate).
- ST4W (scalar plus immediate).
- ST4D (scalar plus immediate).
- ST4B (scalar plus scalar).
- ST4H (scalar plus scalar).
- ST4W (scalar plus scalar).
- ST4D (scalar plus scalar).

8.9.2.3 Gather/scatter load/store/prefetch (SVE)

I_{CDKPW} All of the following are gather/scatter load/store/prefetch (*SVE*) instructions:

- LD1B (vector plus immediate).
- LD1H (vector plus immediate).
- LD1W (vector plus immediate).
- LD1D (vector plus immediate).
- LD1SB (vector plus immediate).
- LD1SH (vector plus immediate).
- LD1SW (vector plus immediate).
- LD1B (scalar plus vector).
- LD1H (scalar plus vector).
- LD1W (scalar plus vector).
- LD1D (scalar plus vector).
- LD1SB (scalar plus vector).
- LD1SH (scalar plus vector).
- LD1SW (scalar plus vector).
- LDFF1B (vector plus immediate).
- LDFF1H (vector plus immediate).
- LDFF1W (vector plus immediate).
- LDFF1D (vector plus immediate).
- LDFF1SB (vector plus immediate).
- LDFF1SH (vector plus immediate).
- LDFF1SW (vector plus immediate).
- LDFF1B (scalar plus vector).
- LDFF1H (scalar plus vector).
- LDFF1W (scalar plus vector).
- LDFF1D (scalar plus vector).
- LDFF1SB (scalar plus vector).
- LDFF1SH (scalar plus vector).
- LDFF1SW (scalar plus vector).
- PRFB (vector plus immediate).
- PRFH (vector plus immediate).
- PRFW (vector plus immediate).
- PRFD (vector plus immediate).
- PRFB (scalar plus vector).
- PRFH (scalar plus vector).
- PRFW (scalar plus vector).
- PRFD (scalar plus vector).

- ST1B (vector plus immediate).
- ST1H (vector plus immediate).
- ST1W (vector plus immediate).
- ST1D (vector plus immediate).
- ST1B (scalar plus vector).
- ST1H (scalar plus vector).
- ST1W (scalar plus vector).
- ST1D (scalar plus vector).

8.9.2.4 Single element load and replicate (SVE)

I_{WWRTF} All of the following are single element load and replicate (*SVE*) instructions:

- LD1RB.
- LD1RH.
- LD1RW.
- LD1RD.
- LD1RSB.
- LD1RSH.
- LD1RSW.

8.9.2.5 Single quadword load and replicate (SVE)

I_{SRJYM} All of the following are single quadword load and replicate (*SVE*) instructions:

- LD1RQB (scalar plus immediate).
- LD1RQH (scalar plus immediate).
- LD1RQW (scalar plus immediate).
- LD1RQD (scalar plus immediate).
- LD1RQB (scalar plus scalar).
- LD1RQH (scalar plus scalar).
- LD1RQW (scalar plus scalar).
- LD1RQD (scalar plus scalar).

8.9.2.6 Register load/store (SVE)

I_{GHZLB} All of the following are register load/store (*SVE*) instructions:

- LDR (predicate).
- LDR (vector).
- STR (predicate).

Chapter 9
Glossary

Active element

An Active element is a vector element or predicate element that is a source register element or destination register element used by an instruction. When the corresponding element in the instruction's *Governing predicate* is TRUE, the element is Active. If an instruction is unpredicated, all of the vector elements or predicate elements are implicitly Active.

Constructive instruction encoding

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

Destructive instruction encoding

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

Element number

For a given element size of N bits, elements within a vector or predicate register are numbered with element[0] always representing bits[(N-1):0], element[1] always representing bits[(2N-1):N], and so on. For more information, see the layout diagram in 2.1.1 *SVE Vector registers*.

First active element

The First active element of a vector or predicate register is the lowest numbered element that is an Active element.

First-fault load

SVE provides a First-fault option for some *SVE* vector load instructions. This option causes memory access faults to be suppressed if they do not occur as a result of the First active element of the vector. Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded. For more information, see 2.1.3 *First Fault Register, FFR*.

Gather-load

Gather-load is a mechanism that allows the elements of a vector to be read from non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode.

Governing predicate

The predicate register that is used to determine the Active elements of a predicated instruction is known as the *Governing predicate* for that instruction.

Inactive element

An Inactive element is a vector element or predicate element that is an unused source register element or destination register element for the associated instruction. When the corresponding element of an instruction's *Governing predicate* is FALSE, the element is inactive.

Last active element

The Last active element of a vector or predicate register is the highest numbered element that is an Active element.

Memory element

An item of data in memory that is transferred to or from a vector or predicate element by an *SVE* load or store instruction. Each memory element has an access size and a type. The memory element access size is specified by each load and store instruction independently of the vector element size.

Merging predication

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

Non-fault load

SVE provides a Non-fault option for some *SVE* vector load instructions. This option causes all memory access faults to be suppressed. Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded. For more information, see 2.1.3 *First Fault Register, FFR*.

Packed access

A memory access that is performed as a result of a load or store instruction for which the vector element size and the memory element size are the same.

Predicate

A one-dimensional array of predicate elements of the same size.

Predicate element

Individual subdivisions of a predicate register that can be 1, 2, 4, or 8 bits in size. The predicate element size is specified independently by each instruction and is always one-eighth the size of the corresponding vector element. The lowest-numbered bit of each predicate element holds the Boolean value of that element, where 1 represents TRUE and 0 represents FALSE.

Predicate register

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

Predicated instruction

An *SVE* instruction that has a *Governing predicate* operand, which determines the Active and Inactive elements for that instruction.

Prefixed instruction

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

Scalar base register

A scalar base register refers to an AArch64 *general-purpose register*, X0-X31, or the current stack pointer, SP.

Scalar index register

A scalar index register refers to an AArch64 *general-purpose register*, X0-X31, or for certain instructions, XZR.

Scatter-store

Scatter-store is a mechanism that allows the elements of a vector to be written to non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode.

SIMD

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

Unpacked access

A memory access that is performed as a result of a load or store instruction for which the vector element size is larger than the memory element size.

Unpredicated instruction

An *SVE* instruction that does not have a *Governing predicate* operand and implicitly treats all other vector and predicate elements as Active.

Vector

A one-dimensional array of vector elements of the same size and data type.

Vector element

Individual subdivisions of a vector register that can be 8, 16, 32, 64 or 128 bits in size. The vector element size and data type is specified independently by each instruction.

Vector length

The accessible width of the *SVE* vector registers at the current *Exception level*, as constrained by the ZCR_EL1, ZCR_EL2, and ZCR_EL3 *System registers*. All vector registers at the same *Exception level* have the same vector length. The accessible width of the *SVE* predicate registers and FFR is one-eighth of the vector length.

Vector register

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

Zeroing predication

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