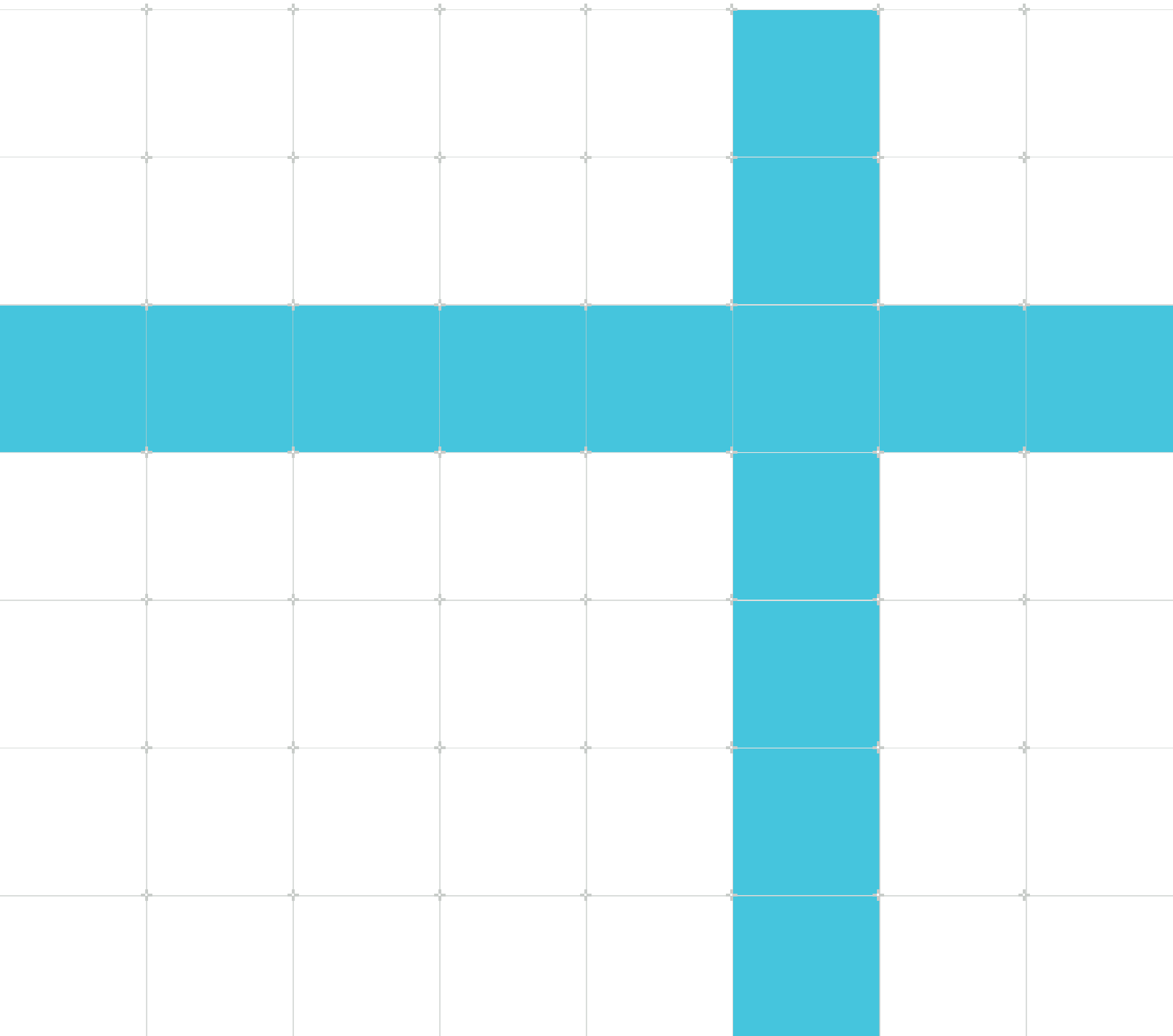
arm

Introduction to SVE

Version 1.0

Non-Confidential

Issue 02
102476_0100_02_en



Introduction to SVE

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	18 January 2022	Non-Confidential	Initial release
0100-02	6 January 2023	Non-Confidential	Fix error in Figure 3-4 Per lane predication merging

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

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

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

This guide is a short introduction to the Scalable Vector Extension (SVE) for the Arm AArch64 architecture. In this guide, you can learn about the concept and main features of SVE, the application domains of SVE, and how SVE compares to Neon. We also describe how to develop a program for an SVE-enabled target.

1.1 Before you begin

This article assumes you are already familiar with the following concepts:

- Single Instruction Multi Data (SIMD)
- Neon

If you are not familiar with these concepts, read: Introducing Neon for Armv8-A

2. Introducing SVE

This section introduces the Scalable Vector Extension (SVE) of the Arm AArch64 architecture.

Following the development of the Neon architecture extension, which has a fixed 128-bit vector length for the instruction set, Arm designed the Scalable Vector Extension (SVE) as a next-generation SIMD extension to AArch64. SVE allows flexible vector length implementations with a range of possible values in CPU implementations. The vector length can vary from a minimum of 128 bits up to a maximum of 2048 bits, at 128-bit increments. The SVE design guarantees that the same application can run on different implementations that support SVE, without the need to recompile the code. SVE improves the suitability of the architecture for High Performance Computing (HPC) and Machine Learning (ML) applications, which require very large quantities of data processing.

SVE introduces the following key features:

- Scalable vectors
- Per-lane predication
- Gather-load and scatter-store
- Speculative vectorization
- Horizontal and serialized vector operations

These features help vectorize and optimize loops when you process large datasets.

SVE is not an extension nor the replacement of the Neon instruction set. SVE is redesigned for better data parallelism for HPC and ML.

The SVE architecture supplement is available.

3. SVE architecture fundamentals

This section introduces the basic architecture features of SVE.

SVE is based on a set of scalable vectors. SVE adds the following registers:

- 32 scalable vector registers, `z0-z31`
- 16 scalable predicate registers, `p0-p15`
- One First Fault predicate Register (FFR)
- Scalable vector system control registers `ZCR_Elx`

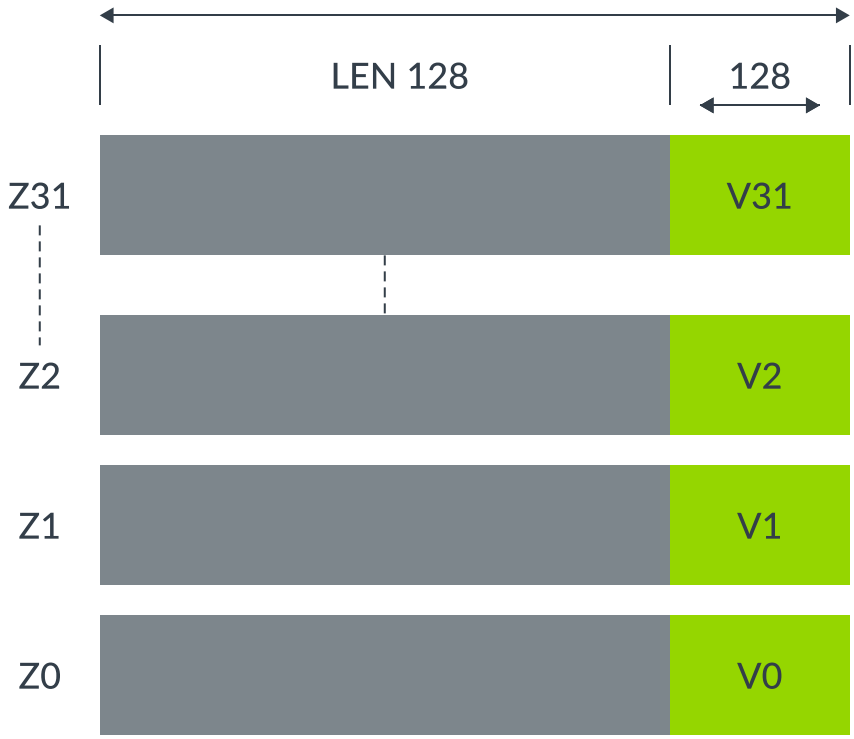
Let us look at each of these registers in turn.

3.1 Scalable vector registers z0-z31

The scalable vector registers `z0-z31` can be implemented with 128-2048 bits in microarchitectures. The bottom 128 bits are shared with the fixed 128-bit `v0-v31` vectors of Neon.

The figure below shows the scalable vector registers `z0-z31`:

Figure 3-1: zo-z31-registers



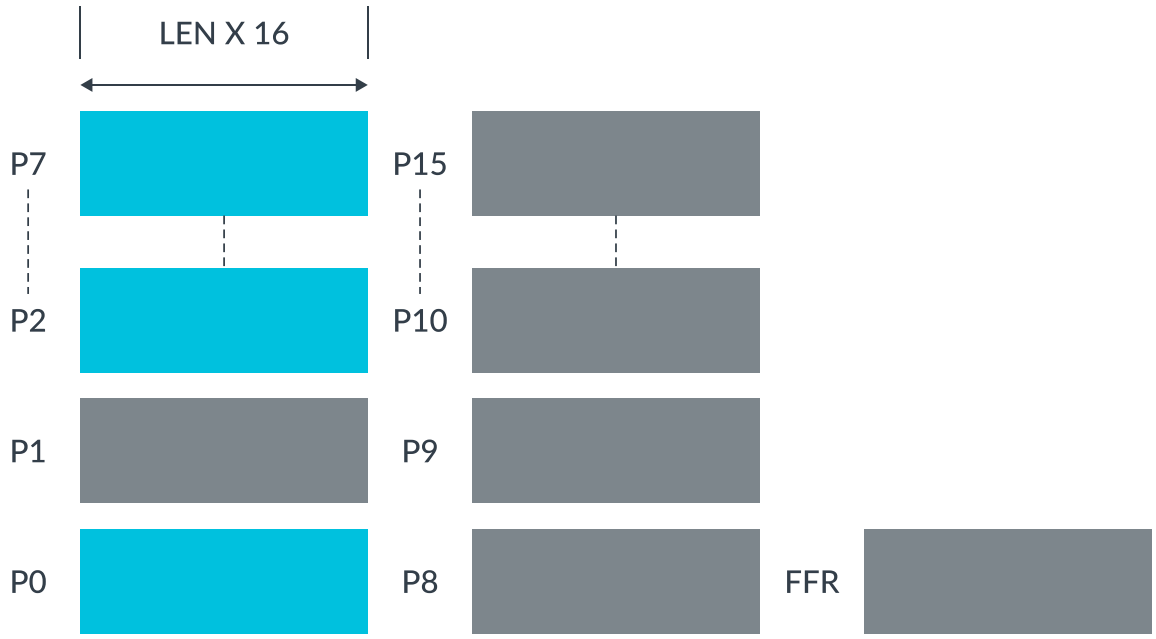
The scalable vectors:

- Can hold 64, 32, 16, and 8-bit elements
- Support integer, double-precision, single-precision, and half-precision floating-point elements
- Are configurable with the vector length for each Exception Level (EL)

3.2 Scalable predicate registers P0-P15

To govern which active elements are involved in the operations, the predicate registers are used in many SVE instructions as masks, which also gives flexibility to the vector operation. The figure below shows the scalable predicate registers `P0-P15`:

Figure 3-2: Scalable predicate registers p0-p15



The predicate registers are usually used as bit masks for data operations, where:

- Each predicate register is 1/8 of the `Zx` length.
- `P0-P7` are governing predicates for load, store, and arithmetic.
- `P8-P15` are extra predicates for loop management.
- First Fault Register (FFR) is a special predicate register, which is set by the first-fault load and store instructions, to indicate how successful the load and store operation for each element is. FFR is designed to support speculative memory accesses which make the vectorization, in many situations, easier and safer.

The predicate registers can also be used as operands in various SVE instructions.

3.3 Configurable vector length

Within the maximum implemented vector length, it is also possible to configure the length of the vector for each Exception level through the `ZCR_Elx` registers. The length implementation and configuration need to meet the minimum requirements from the AArch64 SVE Supplement, so that either of the following conditions are met:

- An implementation must allow the vector length to be constrained to any power of two.
- An implementation allows the vector length to be constrained to multiples of 128 that are not a power of two.

Privileged Exception levels can use the `LEN` fields of the scalable vector control registers `ZCR_El1`, `ZCR_El2`, and ZCR_El3 to constrain the vector length at that Exception level and at less privileged Exception levels:

Figure 3-3: Scalable vector control registers

The scalable vector system control registers indicate the SVE implementation features:

- The `ZCR_Elx.LEN` field is for the vector length of the current and lower exception levels.
- Most bits are currently reserved for future use.

3.4 SVE assembly syntax

SVE assembly syntax format is composed of operation code, destination register, predicate register (if the instruction supports predicate masks), and input operators. The following instruction examples show the detail of this format.

Example 1:

```
LDFF1D {<Zt>.D}, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]
```

Where:

- <Zt> are the vectors, Z0-Z31
- <Zt>.D and <Zm>.D specify the element types of the destination and operand vectors and do not need to specify the element numbers
- <Pg> are the predicates, P0-P15
- <Pg>/Z is the zeroing predication
- <Zm> specifies the offset of the address mode for the gather-load

Example 2:

```
ADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Where:

- M is the merging predication
- <Zdn> is both the destination register and one of the input operators. The instruction syntax shows <Zdn> at both places for your convenience. In assembly encoding, they are encoded once, for simplification.

Example 3

```
ORRS <Pd>.B, <Pg>.Z, <Pn>.B, <Pm>.B
```

Where:

- S is a new interpretation of predicate condition flags NZCV
- <Pg> a governing predicate acts a “bit mask” in the example operation.

3.5 SVE architecture features

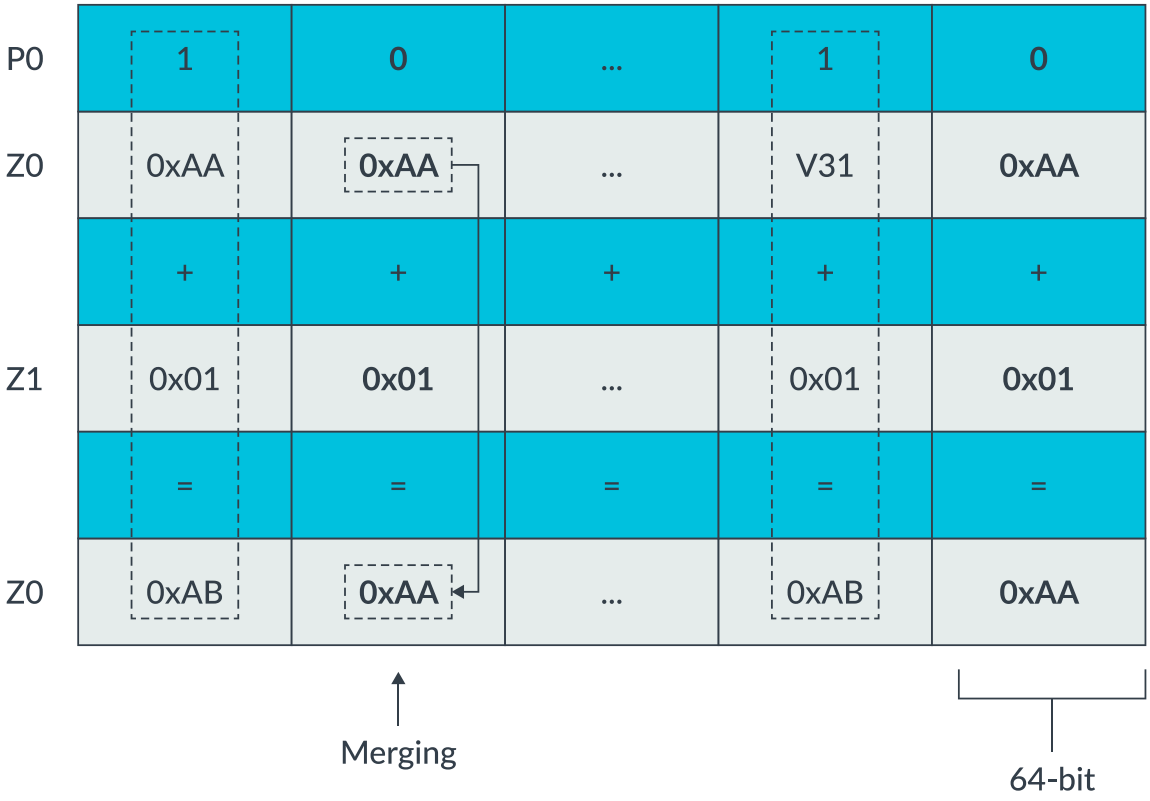
SVE includes the following key architecture features:

- Per-lane predication

To allow flexible operations on selected elements, SVE introduces 16 governing predicate registers, P0-P15, to indicate the valid operation on active lanes of the vectors. For example:

```
ADD Z0.D, P0/M, Z0.D, Z1.D  // Add the active elements Z0 and Z1 and put the result
 in Z0. P0 indicates which elements of the operands are active and inactive. 'M'
 after P0 refers to Merging, which indicates that the inactive element will be
 merged and as a result Z0 inactive element will remain its original value after the
 ADD operation. If it was 'Z' after P0, which refers to Zeroing, then the inactive
 element of the destination register will be zeroed after the operation.
```

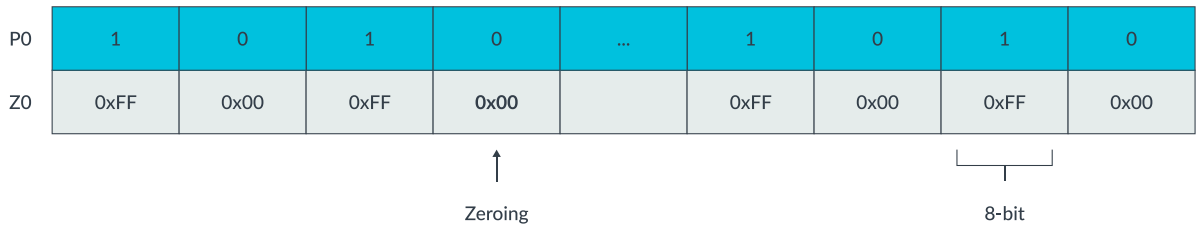
Figure 3-4: Per lane predication merging



If the predicate specification is `/z`, then the operation does zeroing to the results of the corresponding elements of the destination vector, where the predicate elements are zero. For example:

```
CPY Z0.B, P0/Z, #0xFF //Copy a signed integer 0xFF into Z0, where the inactive
 elements of Z0.B will be set to zero.
```

Figure 3-5: Per lane predication zeroing



Note

Not all instructions have predicate options.

Also, not all predicate operations have both merging and zeroing options. You must refer to the SVE Architecture Supplement for the specification details of each instruction.

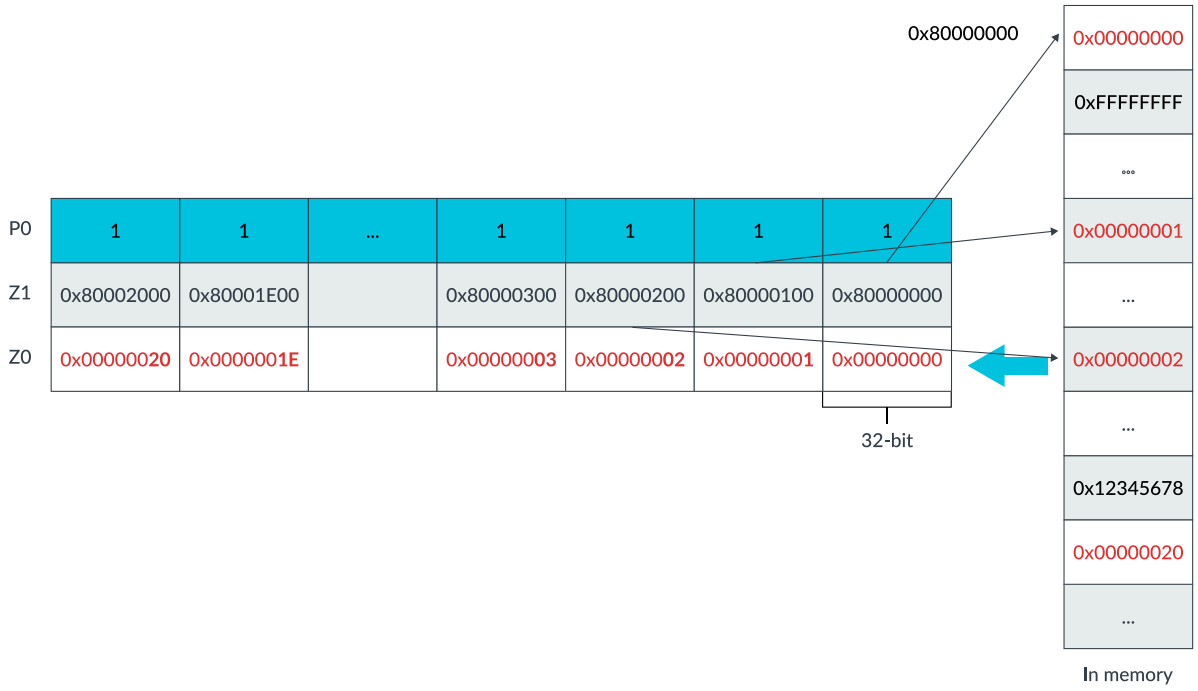
- Gather-load and scatter-store

The address mode in SVE allows the vector to be used as the base address and the offset in the Gather-load and Scatter-store instructions, which enables non-contiguous memory locations. For example:

```
LD1SB  Z0.S, P0/Z, [Z1.S]   // Gather load of signed bytes to active 32-bit elements
 of Z0 from memory addresses generated by 32-bit vector base Z1.
LD1SB  Z0.D, P0/Z, [X0, Z1.D]  // Gather load of signed bytes to active elements of
 Z0 from memory addresses generated by a 64-bit scalar base X0 plus vector index in
 Z1.D.
```

The following example shows the loading operation of `LD1SB Z0.S, P0/Z, [Z1.S]`, where `P0Z1` contains scattered addresses. After loading, the bottom byte of each `Z0.S` is updated with the fetched data from the scattered memory location.

Figure 3-6: Gather-load and scatter-store example

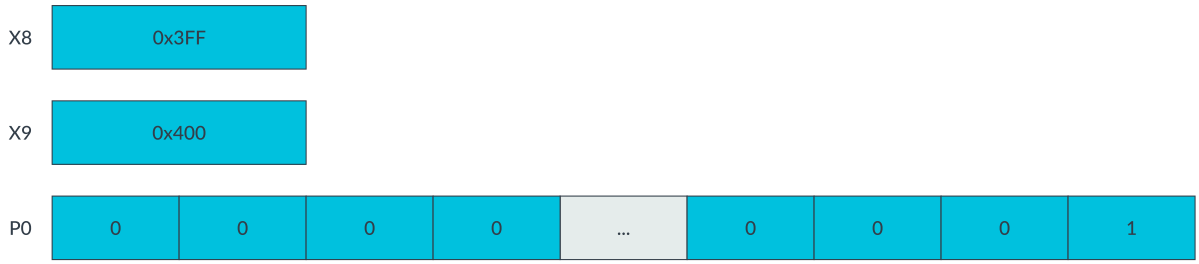


- Predicate-driven loop control and management

As a key feature of SVE, predication does not only give the flexibility of controlling individual elements of the vector operation, but also enables the predicate-driven loop control. Predicate-driven loop control and management make the loop control efficient and flexible. This feature removes the overhead of processing the extra loop heads and tails of partial vectors, by registering the active and inactive elements index in the predicate registers. Predicate-driven loop control and management means that, in the following loop iteration, only the active elements do the expected options. For example:

```
WHILELO P0.S, x8, x9  // Generate a predicate in P0 that starting from the lowest
 numbered element is true while the incrementing value of the first, unsigned scalar
 X8 operand is lower than the second scalar operand X9 and false thereafter, up to
 the highest numbered element.
B.FIRST Loop_start // B.FIRST (equivalent to B.MI) or B.NFRST (equivalent to B.PL)
 are often used to branch based on the above instruction test results of whether the
 first element of P0 is true or false as an ending or continue condition of a loop.
```

Figure 3-7: Predicate-driven loop control and management example



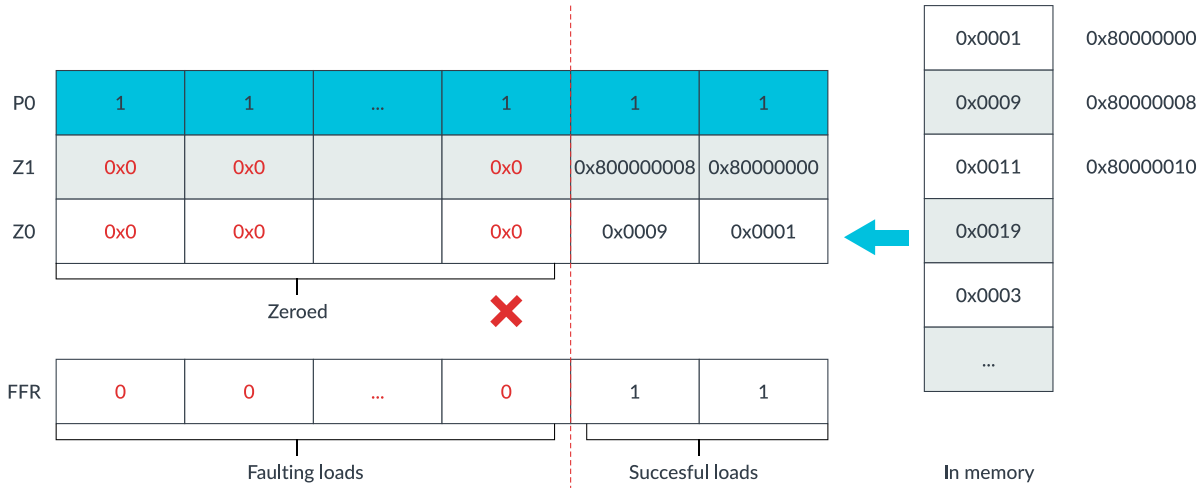
- Vector partitioning for software-managed speculation

Speculative loads can cause challenges to the memory read of a traditional vector, where if any fault occurs in some elements during the read, it is difficult to reverse the load operation and track which elements failed the loading. Neon does not allow speculative load. To allow speculative loads to vectors, for example `LDRFF`, SVE introduces the first-fault vector load instructions. To allow vector accesses to cross into invalid pages, SVE also introduces the First-Fault predicate Registers (FFRs). When loading to an SVE vector with first-fault vector load instructions, the FFR register updates with the load success or fail result for each element. When a load fault occurs, FFR immediately registers the corresponding element, registers the rest of the elements as 0 or false, and does not trigger an exception. Commonly, `RDFFR` instructions are used to read the FFR status. When the first element is false, `RDFFR` instructions finish the iterations. If the first element is true, `RDFFR` instructions continue the iterations. The length of FFR is the same as a predicate vector. The value can be initialized with `SETFFR` instruction. The following example uses `LDFF1D` to read from memory, and the FFR updates correspondingly:

```
LDFF1D Z0.D, P0/Z, [Z1.D, #0]  // Gather load with first-faulting behaviour of
 doublewords to active elements of Z0 from memory addresses generated by the vector
 base Z1 plus 0. Inactive elements will not read Device memory or signal faults and
 are set to zero in the destination vector. Successful loads from the valid memory
```

```
will set true to the elements in FFR. The first-faulting load will set false or 0
to the corresponding element and the rest of the elements in FFR.
```

Figure 3-8: Vector partitioning for software-managed speculation example

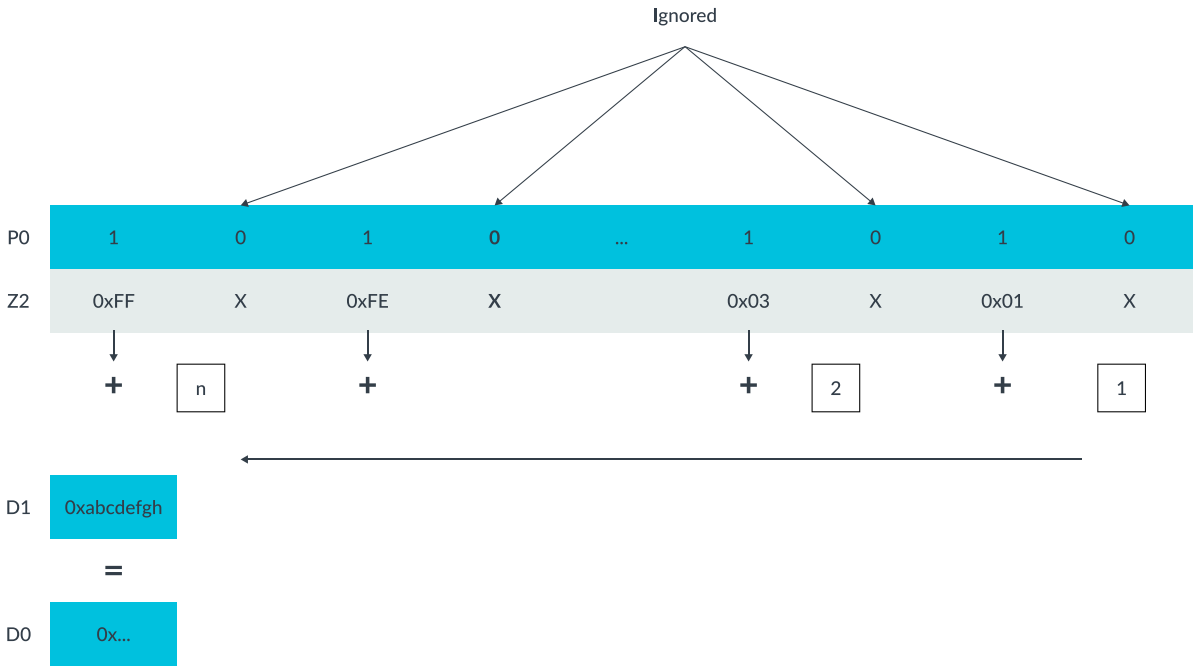


- Extended floating-point and horizontal reductions

To allow efficient reduction operations in a vector, and meet different requirements to the accuracy, SVE enhances floating-point and horizontal reduction operations. The instructions might have in-order (low to high) or tree-based (pairwise) floating-point reduction ordering, where the operation ordering might result in different rounding results. These operations trade-off repeatability and performance. For example:

```
FADDA  D0, P0/M, D1, Z2.D  // Floating-point add strictly-ordered reduction from low
 to high elements of the vector source, accumulating the result in a SIMD&FP scalar
 register. The example instruction adds D1 and all active elements of Z2.D and place
 the result in scalar register D0. Vector elements are process strictly in order
 from low to high, with the scalar source D1 providing the initial value. Inactive
 elements in the source vector are ignored. Whereas FADDV would perform a recursive
 pairwise reduction, and put the result in a scalar register.
```

Figure 3-9: Extended floating-point and horizontal reductions example



4. Programming with SVE

This section describes the software tools and libraries that support SVE application development. This section also describes how to develop your application for an SVE-enabled target and run it on SVE-enabled hardware, and describes how to run your application under SVE emulation on any Armv8-A-based hardware.

4.1 Software and libraries support

To build an SVE application, you must choose a compiler that supports SVE features, such as:

- Version 8.0+ of the GNU tools support SVE optimization for C/C++/Fortran.
- Arm Compiler for Linux, a native compiler for Arm Linux. Arm Compiler for Linux versions 18.0+ supports SVE code generation for C, C++, and Fortran code. Arm Compiler for Linux is part of the Arm Linux user-space tooling solution Arm Allinea Studio.
- Arm Compiler 6, a cross platform compiler for bare-metal application development, also supports SVE code generation from version 6.12. In addition to the compilers, you can also rely on some highly-optimized SVE libraries, such as:
- Arm Performance Libraries, a set of highly optimized math routines, can be linked to your application. Arm Performance Libraries versions 19.3+ support math libraries for SVE. Arm Performance Libraries is part of Arm Compiler for Linux.
- Other third-party math libraries.

4.2 How to program for SVE

There are a few ways to write or generate SVE code. In this section of the guide, we explore four methods of programming for SVE:

- Write SVE assembly code
- Program with SVE intrinsics
- Auto-vectorization
- Using SVE optimized libraries

Let us look at these four options in more detail.

4.3 Write assembly

You can write SVE instructions as inline assembly in your C/C++ code or as a complete function in assembler source. For example:

```
        .globl  subtract_arrays         // -- Begin function
        .p2align        2
        .type   subtract_arrays,@function
subtract_arrays:                // @subtract_arrays
        .cfi_startproc
// %bb.0:
        orr     w9, wzr, #0x400
        mov     x8, xzr
        whilelo p0.s, xzr, x9
.LBB0_1:                                // =>This Inner Loop Header: Depth=1
        ld1w    { z0.s }, p0/z, [x1, x8, lsl #2]
        ld1w    { z1.s }, p0/z, [x2, x8, lsl #2]
        sub     z0.s, z0.s, z1.s
        st1w    { z0.s }, p0, [x0, x8, lsl #2]
        incw    x8
        whilelo p0.s, x8, x9
        b.mi    .LBB0_1
// %bb.2:
        ret
.Lfunc_end0:
        .size   subtract_arrays, .Lfunc_end0-subtract_arrays
        .cfi_endproc T
```

If you are mixing functions that are written in a high-level language and in assembly, you must be familiar with Application Binary Interface (ABI) standard, as updated for SVE. The Procedure Call Standard for Arm Architecture (AAPCS) specifies the data types and register allocations and is most relevant to programming in assembly. The AAPCS requires that:

- `Z0-Z7` and `P0-P3` are used for passing the scalable vector parameters and results.
- `Z8-Z15` and `P4-P15` are callee-saved.
- All the other vector registers (`Z16-Z31`) are corruptible by the callee function, where the caller function is responsible for backing up and restoring them, when needed.

4.4 Use SVE instruction functions (instrinsics)

SVE intrinsics are functions supported by the compilers that can be replaced with corresponding instructions. Programmers can directly call the instruction functions in high-level languages like C and C++. The ACLE (Arm C Language Extension) for SVE defines which SVE instruction functions are available, their parameters and what they do. Compilers which support the ACLE can replace the intrinsics with mapped SVE instructions during the compilation. To use the ACLE intrinsics, you must include the header file “arm_sve.h”, which contains a list of vector types and instruction functions (for SVE) that can be used in C/C++. Each data type describes the size and datatype of the elements in the vector:

- `svint8_t svuint8_t`
- `svint16_t svuint16_t svfloat16_t`

- `svint32_t svuint32_t svfloat32_t`
- `svint64_t svuint64_t svfloat64_t`

For example, `svint64_t` represents a vector of 64-bit signed integers, and `svfloat16_t` represents a vector of half-precision floating-point numbers.

The following example C code has been manually optimized with SVE intrinsics:

```
//intrinsic_example.c
#include <arm_sve.h>
svuint64_t uaddlb_array(svuint32_t Zs1, svuint32_t Zs2)
{
    // widening add of even elements
    svuint64_t result = svaddlb(Zs1, Zs2);
    return result;
}
```

Source code, which includes arm_sve.h, can use the SVE vector types in the same way data types can be used for variable declaration and function parameters. To compile the code using Arm C/C++ Compiler, and target the Armv8-A architecture that supports SVE, use:

```
armclang -O3 -S -march=armv8-a+sve -o intrinsic_example.s intrinsic_example.c
```

This command generates the following assembly code:

```
//instrinsic_example.s
uaddlb_array:                           // @uaddlb_array
        .cfi_startproc
// %bb.0:
        uaddlb  z0.d, z0.s, z1.s
        ret
```

This example uses Arm Compiler for Linux 20.0.

4.5 Auto-vectorization

C/C++/Fortran compilers, for example the native Arm Compiler for Linux and GNU compilers for Arm platforms, support vectorizing C, C++, and Fortran loops using SVE instructions. To generate SVE code, select the appropriate compiler options. For example, when `armclang` uses the `-march=armv8-a+sve` option, the `armclang` also uses the default options `-fvectorize` and `-O2`. If you want to use the SVE-enabled version of the libraries, combine `-march=armv8-a+sve` with `-armpl=sve`. For more information about the compiler optimization options, refer to the compiler developer and reference guides, or the compiler man pages.

4.6 Use optimized libraries

Use libraries that are highly-optimized for SVE, for example Arm Performance Libraries and Arm Compute Library. Arm Performance Libraries contain highly-optimized implementations for BLAS, LAPACK, FFT, sparse linear algebra, and libamath-optimized mathematical functions. To be able to link any of the Arm Performance Libraries functions, you must install Arm Allinea Studio and include armpl.h in your code. To build the application with Arm Compiler for Linux and Arm Performance Libraries, you must specify `-armpl=<arg>` on the command line. If you use the GNU tools, you must include the Arm Performance Libraries installation path in the linker command line with `-L<armpl_install_dir>/lib`, and specify the GNU-equivalent to the Arm Compiler for Linux `armpl=<arg>` option, which is `-larmpl_lp64`. For more information, please reference to the Arm Performance Libraries Get started guide.

4.7 How to run an SVE application

If you do not have access to SVE hardware, you can use models or emulators to run your code. There are a few models and emulators to choose from:

- QEMU: Cross and native models, which support modeling on Arm AArch64 platforms with SVE
- Fast Models: Cross platform models, which support modeling Arm AArch64 platforms with SVE, running on x86-based hosts.
- Arm Instruction Emulator (ArmIE): Native AArch64 emulator, which supports the emulation of SVE instructions, and other new instructions, for future architectures.

5. Check your knowledge

The following questions will help you test your knowledge.

Which scalable vectors are introduced in SVE?

SVE introduces `Z0- Z31` vectors, `P0–P15` predicate registers, and an `FFR` predicate register.

How many bits can SVE vectors have?

`Z0- Z31` can be implemented from 128 bits up to 2048 bits wide. The length can be either a power of 2 number or a multiplication of 128.

What are the advantages of SVE compared to a traditional SIMD instruction set, for example Neon?

The advantages of SVE, compared to Neon, include:

- SVE programs can be vector-length agnostic; a single binary works on machines with different hardware vector lengths.
- SVE has more vectorization flexibility.
- SVE is designed for HPC and ML. Compared to Neon-based targets, SVE enables application performance advantages, even when the SVE-enabled targets use the same vector length (128-bit) as Neon targets.

6. Related information

Here are some resources that relate to the content in this guide:

- Arm architecture exploration tools
- Arm Architecture Reference Manual Supplement – The Scalable Vector Extension (SVE) for Armv8-A
- ACLE (Arm C Language Extensions (ACLE) for SVE
- Arm A64 Instruction Set Architecture: Future Architecture Technologies in the A architecture profile
- The Procedure Call Standard for Arm Architecture (AAPCS)
- Vector Function Application Binary Interface Specification for AArch64
- Server and HPC Linux user space software tooling: Arm Linux Compiler, Arm Performance Libraries
- Arm Instruction Emulator
- SVE Programmers Guide
- Arm SVE intrinsics coding considerations
- SVE and Neon coding compared
- Arm Community – Ask development questions and find articles and blogs on specific topics from Arm experts.
- Arm Compiler 6 for bare-metal images
- Fast models
- Neon resources
- QEMU