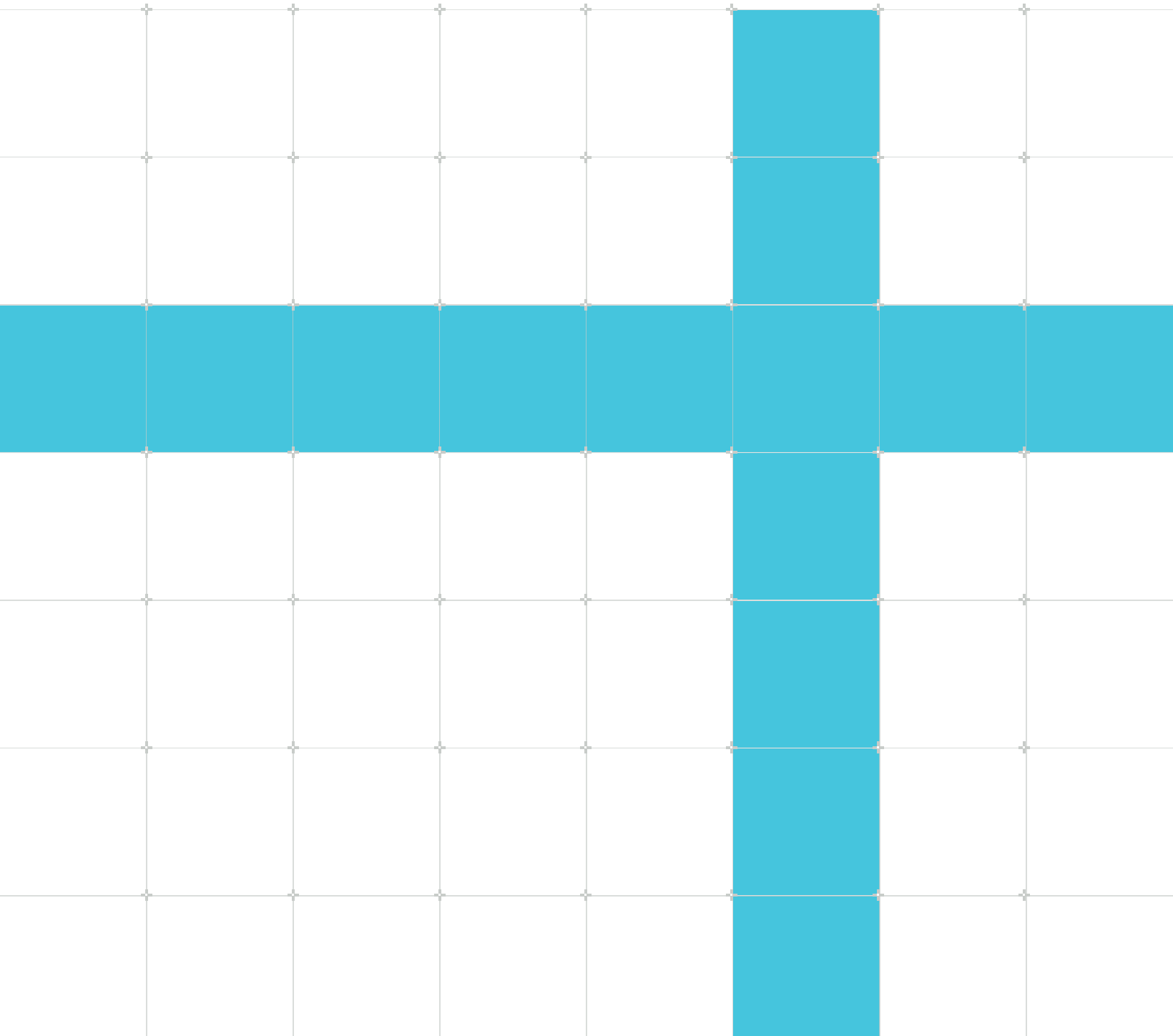
arm

Learn the architecture - Introducing Neon

Version 1.0

Issue 02
102474_0100_02_en



Learn the architecture - Introducing Neon

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	1 January 2020	Non-Confidential	Initial release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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.

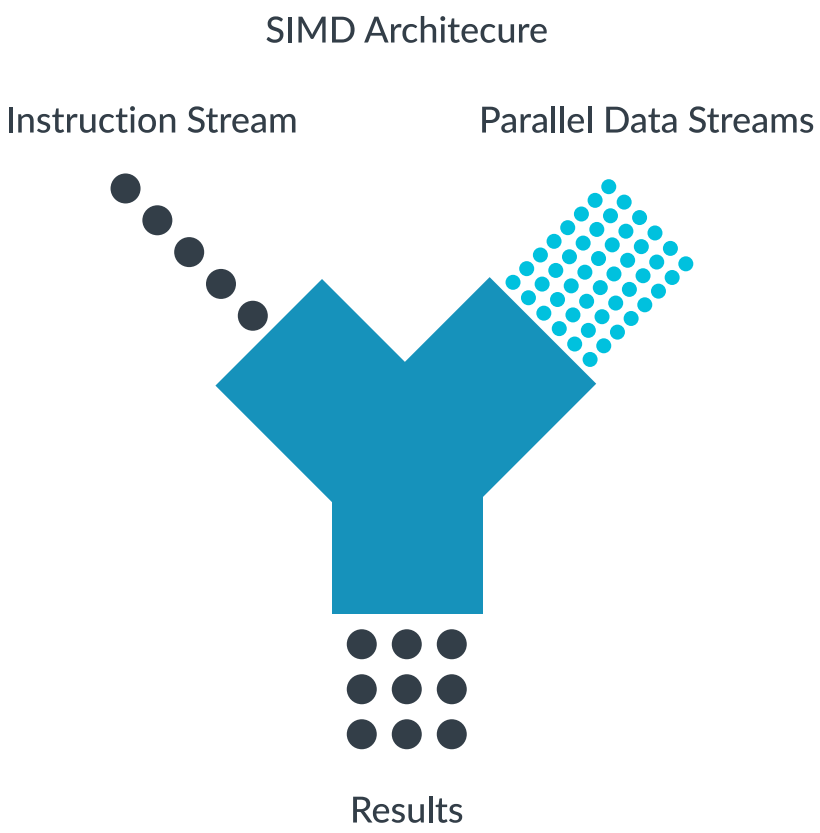
Contents

1. Overview

This guide introduces Arm Neon technology, the Advanced SIMD (Single Instruction Multiple Data) architecture extension for implementation of the Armv8-A or Armv8-R architecture profiles.

Neon technology provides a dedicated extension to the Instruction Set Architecture, providing additional instructions that can perform mathematical operations in parallel on multiple data streams.

Figure 1-1: Single instruction, multiple data Architecture



This can improve the multimedia user experience by accelerating audio and video encoding/decoding, user interface, 2D/3D graphics or gaming. Neon can also accelerate signal processing algorithms and functions to speed up applications such as audio and video processing, voice and facial recognition, computer vision and deep learning.

As a programmer, there are a number of ways you can make use of Neon technology:

- Neon-enabled open source libraries such as the Arm Compute Library provide one of the easiest ways to take advantage of Neon.

- Auto-vectorization features in your compiler can automatically optimize your code to take advantage of Neon.
- Neon intrinsics are function calls that the compiler replaces with appropriate Neon instructions. This gives you direct, low-level access to the exact Neon instructions you want, all from C/C++ code.
- For very high performance, hand-coded Neon assembler can be an alternative approach for experienced programmers.

1.1 Before you begin

If you are completely new to Arm technology, you can read the Cortex-A Series Programmer's Guide for general information about the Arm architecture and programming guidelines.

The information in this guide relates to Neon for Armv8. If you are developing for Armv7 devices, you might find version 1.0 of the Neon Programmer's Guide more appropriate for your needs.

If you are hand-coding in assembler for a specific device, refer to the Technical Reference Manual for that processor to see microarchitectural details that can help you maximize performance. For some processors, Arm also publishes a Software Optimization Guide which may be of use. For example, see the Arm Cortex-A75 Technical Reference Manual and the Arm Cortex-A75 Software Optimization Guide.

2. Data processing methodologies

When processing large sets of data, a major performance limiting factor is the amount of CPU time taken to perform data processing instructions. This CPU time depends on the number of instructions it takes to deal with the entire data set. And the number of instructions depends on how many items of data each instruction can process.

2.1 Single Instruction Single Data

Most Arm instructions are Single Instruction Single Data (SISD). Each instruction performs its specified operation on a single data source. Processing multiple data items therefore requires multiple instructions. For example, to perform four addition operations requires four instructions to add values from four pairs of registers:

```
ADD w0, w0, w5
ADD w1, w1, w6
ADD w2, w2, w7
ADD w3, w3, w8
```

This method is relatively slow and it can be difficult to see how different registers are related. To improve performance and efficiency, media processing is often off-loaded to dedicated processors such as a Graphics Processing Unit (GPU) or Media Processing Unit which can process more than one data value with a single instruction.

If the values you are dealing with are smaller than the maximum bit size, that extra potential bandwidth is wasted with SISD instructions. For example, when adding 8-bit values together, each 8-bit value needs to be loaded into a separate 64-bit register. Performing large numbers of individual operations on small data sizes does not use machine resources efficiently because processor, registers, and data path are all designed for 64-bit calculations.

2.2 Single Instruction Multiple Data

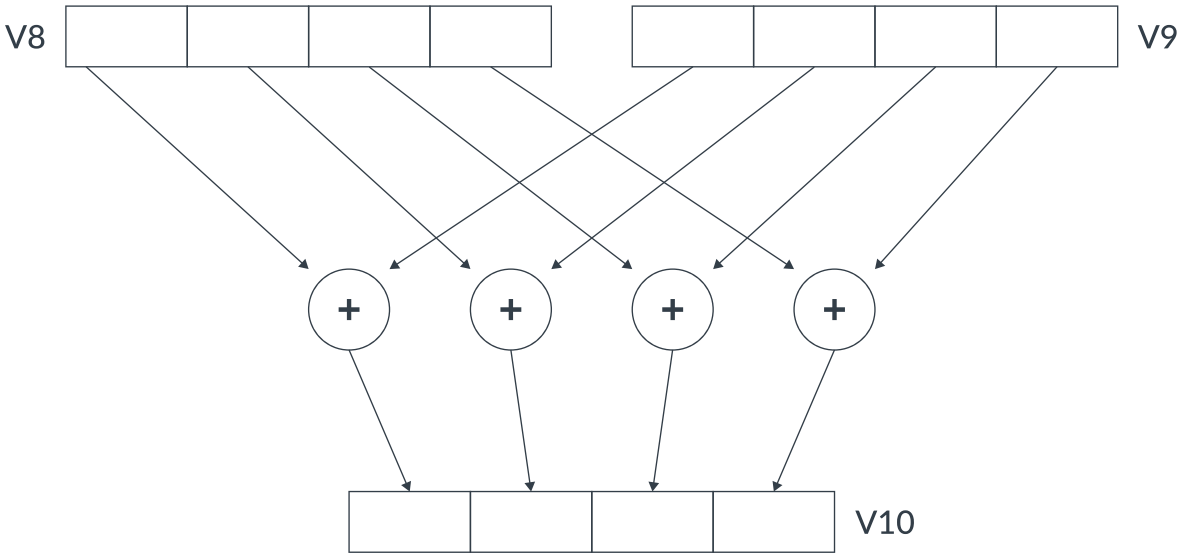
Single Instruction Multiple Data (SIMD) instructions perform the same operation simultaneously for multiple data items. These data items are packed as separate lanes in a larger register.

For example, the following instruction adds four pairs of single-precision (32-bit) values together. However, in this case, the values are packed into separate lanes in two pairs of 128-bit registers. Each lane in the first source register is then added to the corresponding lane in the second source register, before being stored in the same lane in the destination register:

```
ADD V10.4S, V8.4S, V9.4S
// This operation adds two 128-bit (quadword) registers, V8 and V9,
// and stores the result in V10.
// Each of the four 32-bit lanes in each register is added separately.
// There are no carries between the lanes.
```

This single instruction operates on all data values in the large register at the same time:

Figure 2-1: Single Instruction Multiple Data



Performing the four operations with a single SIMD instruction is faster than with four separate SISD instructions.

The diagram shows 128-bit registers each holding four 32-bit values, but other combinations are possible for Neon registers:

- Two 64-bit, four 32-bit, eight 16-bit, or sixteen 8-bit integer data elements can be operated on simultaneously using all 128 bits of a Neon register.
- Two 32-bit, four 16-bit, or eight 8-bit integer data elements can be operated on simultaneously using the lower 64 bits of a Neon register (in this case, the upper 64 bits of the Neon register are unused).

Note

The addition operations shown in the diagram are truly independent for each lane. Any overflow or carry from lane 0 does not affect lane 1, which is an entirely separate calculation.

Media processors, such as used in mobile devices, often split each full data register into multiple sub-registers and perform computations on the sub-registers in parallel. If the processing for the data sets are simple and repeated many times, SIMD can give considerable performance improvements. It is particularly beneficial for digital signal processing or multimedia algorithms, such as:

- Audio, video, and image processing codecs.
- 2D graphics based on rectangular blocks of pixels.

- 3D graphics
- Color-space conversion.
- Physics simulations.

3. Fundamentals of Armv8 Neon technology

Armv8-A includes both 32-bit and 64-bit Execution states, each with their own instruction sets:

- AArch64 is the name used to describe the 64-bit Execution state of the Armv8-A architecture. In AArch64 state, the processor executes the A64 instruction set, which contains Neon instructions (also referred to as SIMD instructions). GNU and Linux documentation sometimes refers to AArch64 as ARM64.
- AArch32 describes the 32-bit Execution state of the Armv8-A architecture, which is almost identical to Armv7. In AArch32 state, the processor can execute either the A32 (called ARM in earlier versions of the architecture) or the T32 (Thumb) instruction set. The A32 and T32 instruction sets are backwards compatible with Armv7, including Neon instructions.

This guide will focus on Neon programming using A64 instructions for the AArch64 Execution state of the Armv8-A architecture.

If you want to write Neon code to run in the AArch32 Execution state of the Armv8-A architecture, you should refer to version 1.0 of the Neon Programmer's Guide.

3.1 Registers, vectors, lanes and elements

If you are familiar with the Armv8-A architecture profile, you will have noticed that in AArch64 state, Armv8 cores are a 64-bit architecture and use 64-bit registers, but the Neon unit uses 128-bit registers for SIMD processing.

This is possible because the Neon unit operates on a separate register file of 128-bit registers. The Neon unit is fully integrated into the processor and shares the processor resources for integer operation, loop control, and caching. This significantly reduces the area and power cost compared to a hardware accelerator. It also uses a much simpler programming model, since the Neon unit uses the same address space as the application.

The Neon register file is a collection of registers which can be accessed as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit registers.

The Neon registers contain vectors of elements of the same data type. The same element position in the input and output registers is referred to as a lane.

Usually each Neon instruction results in n operations occurring in parallel, where n is the number of lanes that the input vectors are divided into. Each operation is contained within the lane. There cannot be a carry or overflow from one lane to another.

The number of lanes in a Neon vector depends on the size of the vector and the data elements in the vector.

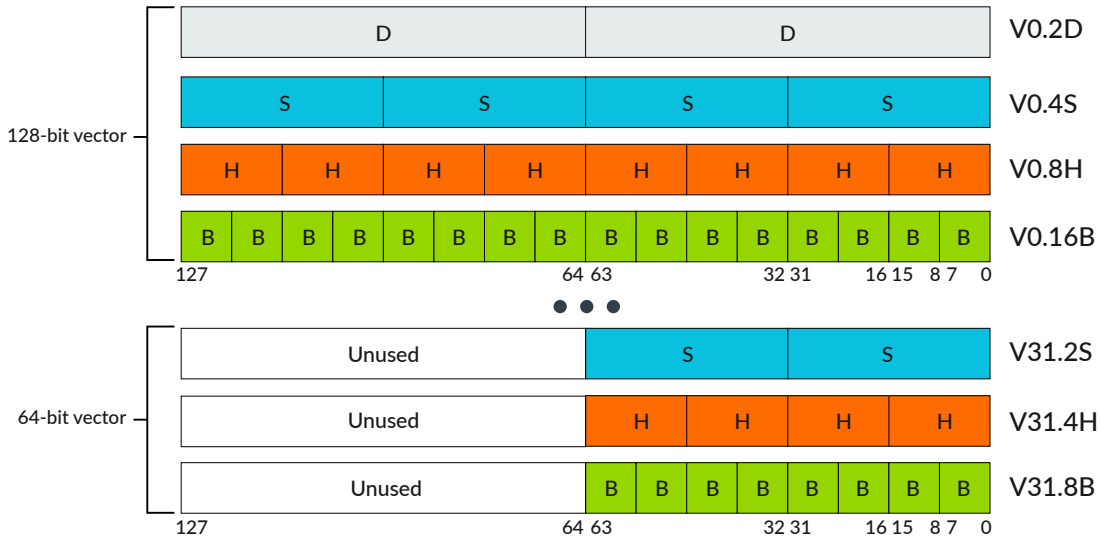
A 128-bit Neon vector can contain the following element sizes:

- Sixteen 8-bit elements (operand suffix `.16B`, where `B` indicates byte)
- Eight 16-bit elements (operand suffix `.8H`, where `H` indicates halfword)
- Four 32-bit elements (operand suffix `.4S`, where `S` indicates word)
- Two 64-bit elements (operand suffix `.2D`, where `D` indicates doubleword)

A 64-bit Neon vector can contain the following element sizes (with the upper 64 bits of the 128-bit register cleared to zero):

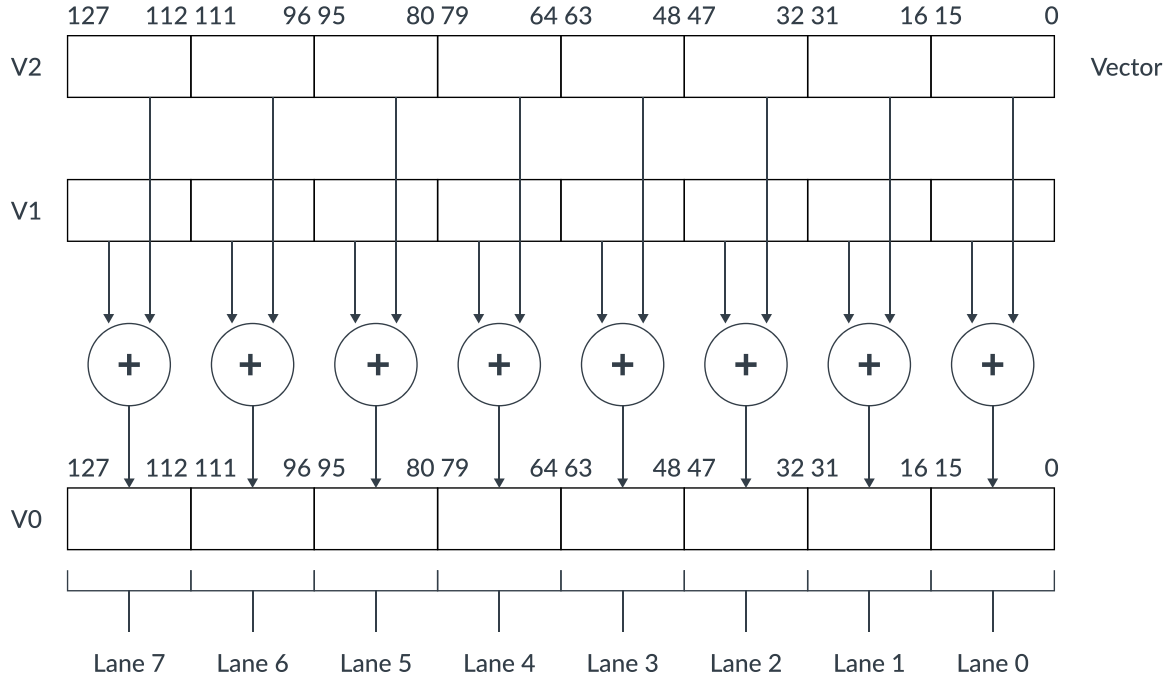
- Eight 8-bit elements (operand suffix `.8B`, where `B` indicates byte)
- Four 16-bit elements (operand suffix `.4H`, where `H` indicates halfword)
- Two 32-bit elements (operand suffix `.2S`, where `S` indicates word)

Figure 3-1: 64-bit and 128-bit vectors



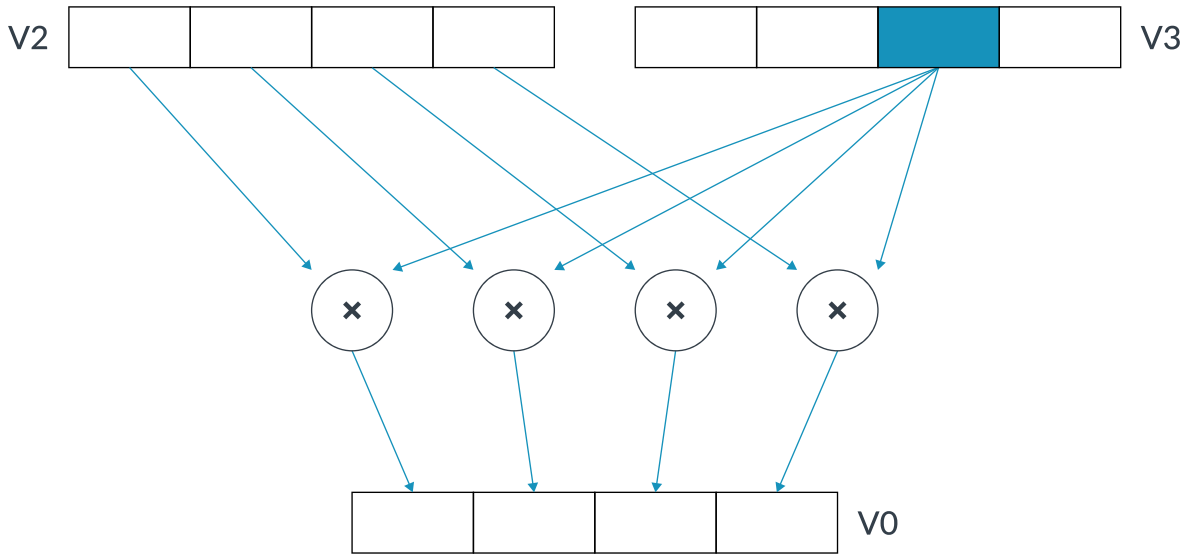
Elements in a vector are ordered from the least significant bit. That is, element 0 uses the least significant bits of the register. Let's look at an example of a Neon instruction. The instruction ADD V0.8H, V1.8H, V2.8H performs a parallel addition of eight lanes of 16-bit (8 x 16 = 128) integer elements from vectors in V1 and V2, storing the result in V0:

Figure 3-2: Order of elements in a vector



Some Neon instructions act on scalars together with vectors. Instructions which use scalars specify a lane index to refer to a specific element in a register. For example, the instruction `MUL V0.4S, V2.4S, V3.S[2]` multiplies each of the four 32-bit elements in `V2` by the 32-bit scalar value in lane 2 of `V3`, storing the result vector in `V0`.

Figure 3-3: Instructions using scalars



4. Check your knowledge

The following questions will help you test your knowledge.

What is the difference between Neon and the Advanced SIMD Architecture?

Neon is a brand name which refers to Arm's implementations of the Advanced SIMD Architecture. Although the two terms are often used interchangeably, Neon is not strictly speaking a feature of the Arm Architecture. Those looking to learn more about Neon from the Architecture Reference Manual or the Cortex-A Technical Reference Manuals, should therefore search for Advanced SIMD rather than Neon.

What does SIMD stand for and how can SIMD instructions speed up programs which use SISD instructions?

SIMD stands for Single Instruction Multiple Data. Since SIMD instructions can perform more operations than an equivalent SISD (Single Instruction Single Data) instruction, a program using SIMD instructions can process more data per instruction on average. If the execution time of the SIMD and SISD instructions are the same, the program will speed up.

What are the four basic ways one can use Neon?

Import a library using Neon, e.g. the Arm Compute Library or the Arm Performance Libraries. Use a compiler supporting Neon code generation, e.g. the Arm Compilers or GCC. Use the Neon Intrinsics in C or C++ code. Write Arm assembly which uses Advanced SIMD instruction set.

How many Neon registers are there in the AArch64 execution state, and how can they be divided into different lanes?

There are 32 128-bit registers, which an be divided into lanes which are 8, 16, 32, or 64 bits wide. These same registers can also be treated as 64-bit registers with the upper bits left unused.

5. Related information

Here are some resources related to material in this guide:

For definitive information about the SIMD instructions and registers, refer to the Arm Architecture Reference Manual for the Armv8-A architecture profile.

The ISA exploration tools provide descriptions in XML and HTML format for the A64 Instruction Set Architecture, including the SIMD instructions.

The Neon Intrinsics Reference provides information about Neon intrinsics - function calls that let C and C++ programmers code directly for Neon without writing assembly.

Arm Community has a useful article that introduces Neon: Arm Neon Programming Quick Reference.

5.1 Useful links to training:

Arm access is needed to view the trainings.

- Introduction to Armv8-A
- Overview ISA
- Arm's other architectures

6. Next step

The Optimizing C Code with Neon Intrinsics tutorial provides an excellent place to start for newcomers to Neon programming. The tutorial describes how to use Neon intrinsics by examining an example which processes a 24-bit RGB image with interleaved pixel data.