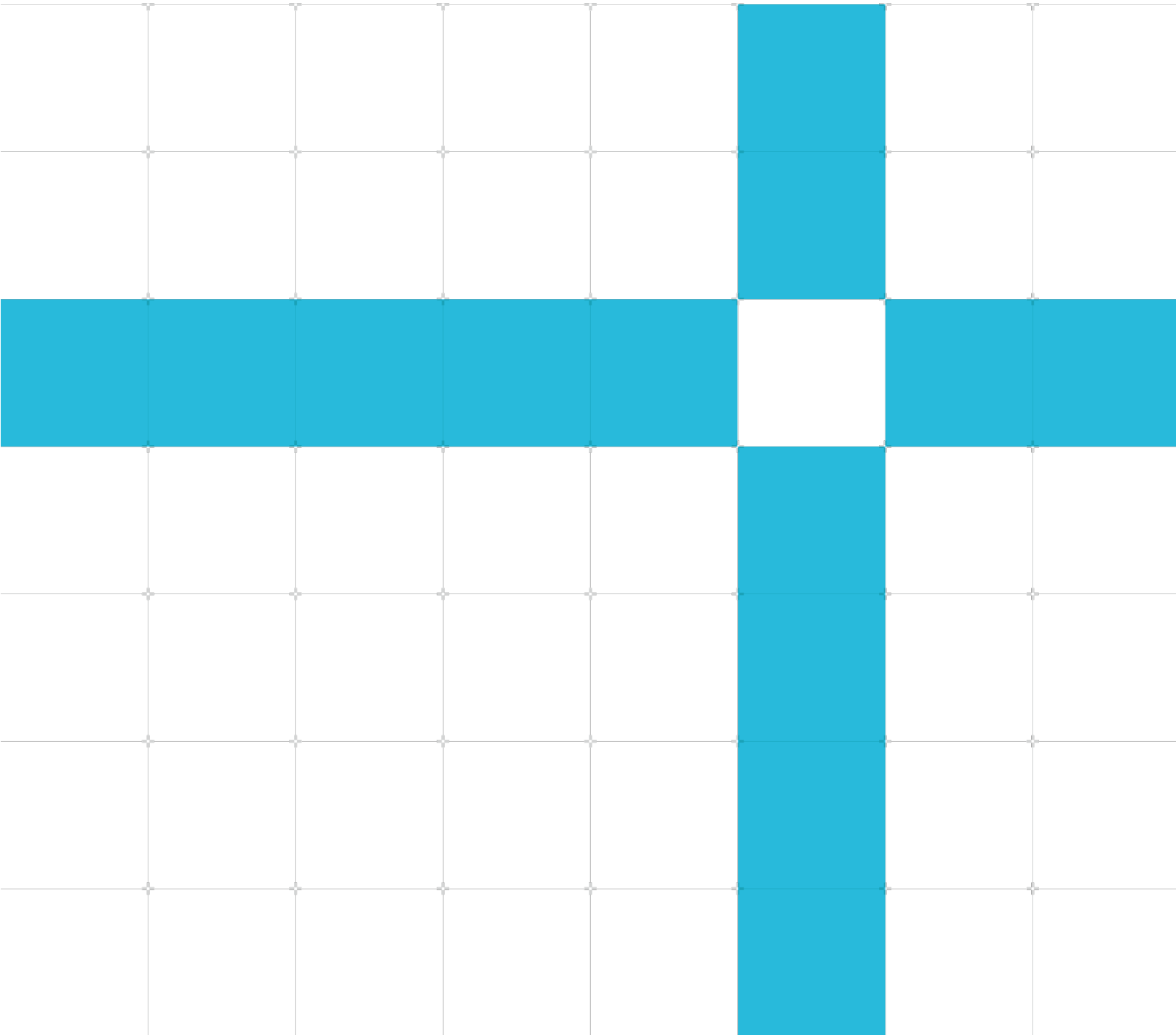
arm

Coding for Neon

Non-Confidential

Issue 0101
102159



Coding for Neon

Release information

Document history

Issue	Date	Confidentiality	Change
01	05 July 2020	Non-Confidential	First release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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.

Web Address

www.arm.com

Contents

1 Overview

Arm Neon technology is a 64-bit or 128-bit hybrid Single Instruction Multiple Data (SIMD) architecture that is designed to accelerate the performance of multimedia and signal processing applications. These applications include the following:

- Video encoding and decoding
- Audio encoding and decoding
- 3D graphics processing
- Speech processing
- Image processing

This guide provides information about how to write SIMD code for Neon using assembly language.

This guide will grow and evolve over time. When complete, the guide will cover getting started with Neon, using it efficiently, and hints and tips for more experienced coders.

This first installment of the guide begins by looking at memory operations, and how to use the flexible load and store with permute instructions.

More installments will follow.

2 Load and store: example RGB conversion

This section considers an example task of converting RGB data to BGR color data.

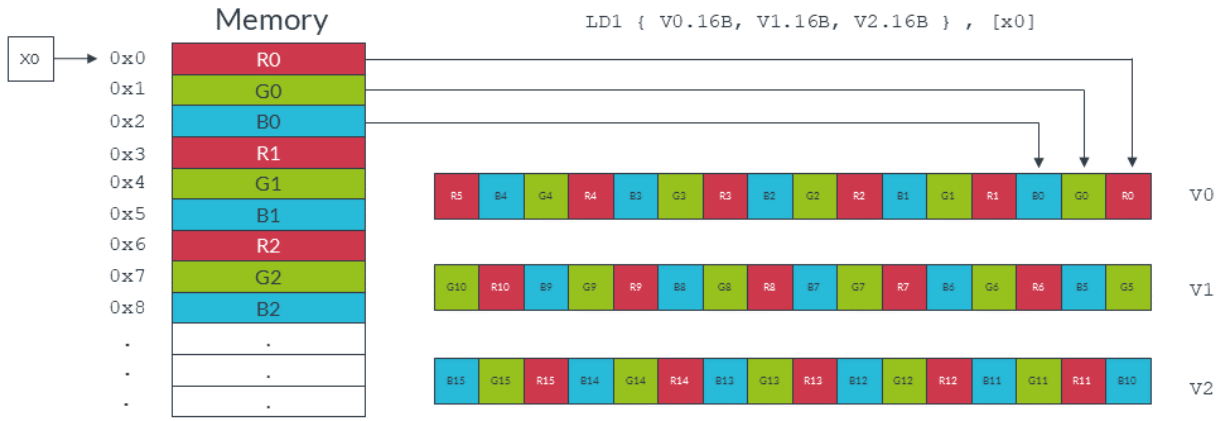
In a 24-bit RGB image, the pixels are arranged in memory as R, G, B, R, G, B, and so on. You want to perform a simple image-processing operation, like switching the red and blue channels. How can you do this efficiently using Neon?

Using a load that pulls RGB data items sequentially from memory into registers makes swapping the red and blue channels awkward.

Consider the following instruction, which loads RGB data one byte at a time from memory into consecutive lanes of three Neon registers:

```
LD1 { V0.16B, V1.16B, V2.16B }, [x0]
```

The following diagram shows the operation of this instruction:

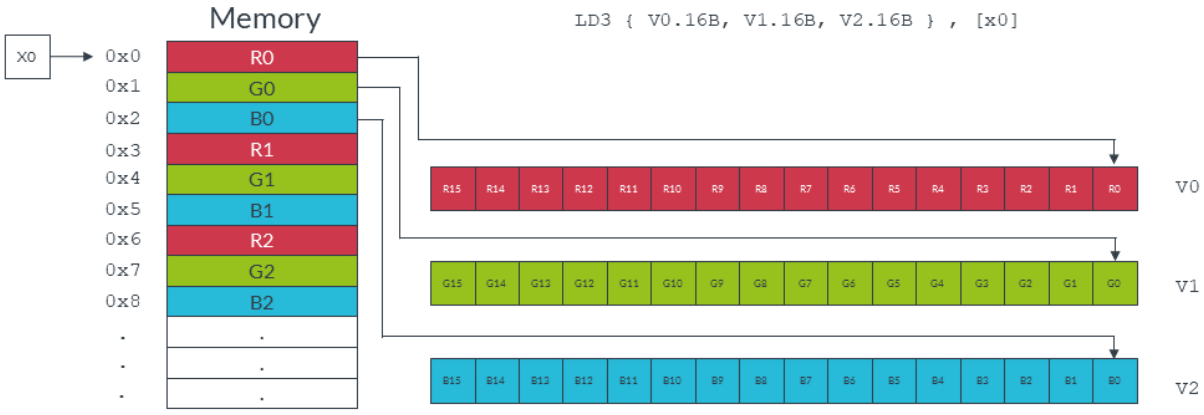


Code to swap channels based on this input would be complicated. We would need to mask different lanes to obtain the different color components, then shift those components and recombine. The resulting code is unlikely to be efficient.

Neon provides structure load and store instructions to help in these situations. These instructions pull in data from memory and simultaneously separate the loaded values into different registers. For this example, you can use the `LD3` instruction to separate the red, green, and blue data values into different Neon registers as they are loaded:

```
LD3 { V0.16B, V1.16B, V2.16B }, [x0]
```

The following diagram shows how the above instruction separates the different data channels:



The red and blue values can now be switched easily using the `MOV` instruction to copy the entire vector. Finally, we write the data back to memory, with reinterleaving, using the `ST3` store instruction.

A single iteration of this RGB to BGR switch can be coded as follows:

```
LD3  { V0.16B, V1.16B, V2.16B }, [x0], #48  // 3-way interleaved load from
                                            // address in X0, post-incremented
                                            // by 48

MOV    V3.16B, V0.16B                       // Swap V0 -> V3

MOV    V0.16B, V2.16B                       // Swap V2 -> V0

MOV    V2.16B, V3.16B                       // Swap V3 -> V2
                                            // (net effect is to swap V0 and V2)

ST3  { V0.16B, V1.16B, V2.16B }, [x1], #48  // 3-way interleaved store to address
                                            // in X1, post-incremented by 48
```

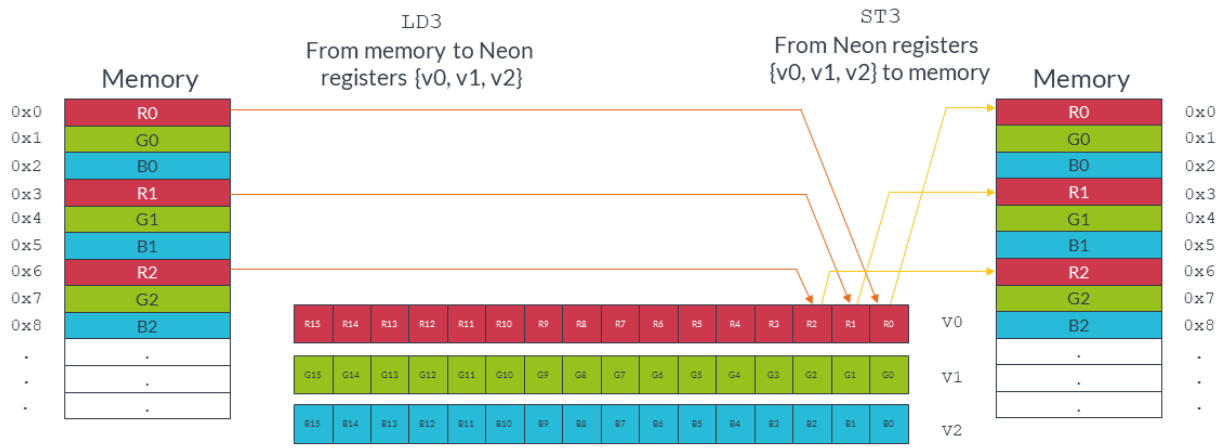
Each iteration of this code does the following:

- Loads from memory 16 red bytes into `V0`, 16 green bytes into `V1`, and 16 blue bytes into `V2`.
- Increments the source pointer in `X0` by 48 bytes ready for the next iteration. The increment of 48 bytes is the total number of bytes that we read into all three registers, so 3 x 16 bytes in total.
- Swaps the vector of red values in `V0` with the vector of blue values in `V2`, using `V3` as an intermediary.
- Stores the data in `V0`, `V1`, and `V2` to memory, starting at the address that is specified by the destination pointer in `X1`, and increments the pointer.

3 Load and store: data structures

Neon structure load instructions read data from memory into 64-bit Neon registers, with optional deinterleaving.

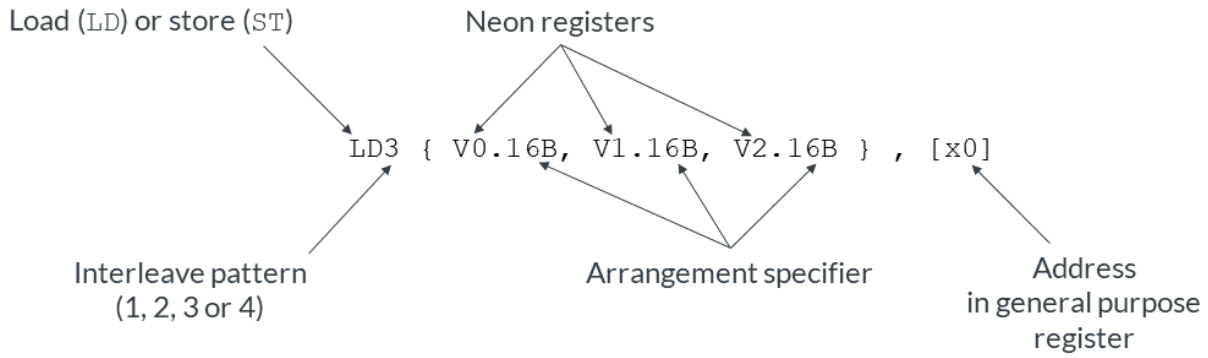
Structure store instructions work similarly, reinterleaving data from registers before writing it to memory, as shown in the following diagram:



3.1 Syntax

The structure load and store instructions follow a consistent syntax.

The following diagram shows the general syntax of the structure load and store instructions:



This instruction syntax has the following format:

- An instruction mnemonic, with two parts:
  - The operation, either `LD` for loads or `ST` for stores.
  - A numeric interleave pattern specifying the number of elements in each structure.

- A set of 64-bit Neon registers to be read or written. A maximum of four registers can be listed, depending on the interleave pattern. Each entry in the set of Neon registers has two parts:
  - The Neon register name, for example `V0`.
  - An arrangement specifier. This indicates the number of bits in each element and the number of elements that can fit in the Neon vector register. For example, `16B` indicates that each element is one byte (`B`), and each vector is a 128-bit vector containing 16 elements.
- A general-purpose register containing the location to access in memory. The address can be updated after the access.

3.2 Interleave pattern

Neon provides instructions to load and store interleaved structures containing from one to four equally sized elements. Elements are the standard Neon-supported widths of 8 (B), 16 (H), 32 (S), or 64 (D) bits.

- `LD1` is the simplest form. It loads one to four registers of data from memory, with no deinterleaving. You can use `LD1` to process an array of non-interleaved data.
- `LD2` loads two or four registers of data, deinterleaving even and odd elements into those registers. You can use `LD2` to separate stereo audio data into left and right channels.
- `LD3` loads three registers and deinterleaves. You can use `LD3` to split RGB pixel data into separate color channels.
- `LD4` loads four registers and deinterleaves. You can use `LD4` to process ARGB image data.

The store instructions `ST1`, `ST2`, `ST3`, and `ST4` support the same options, but interleave the data from registers before writing them to memory.

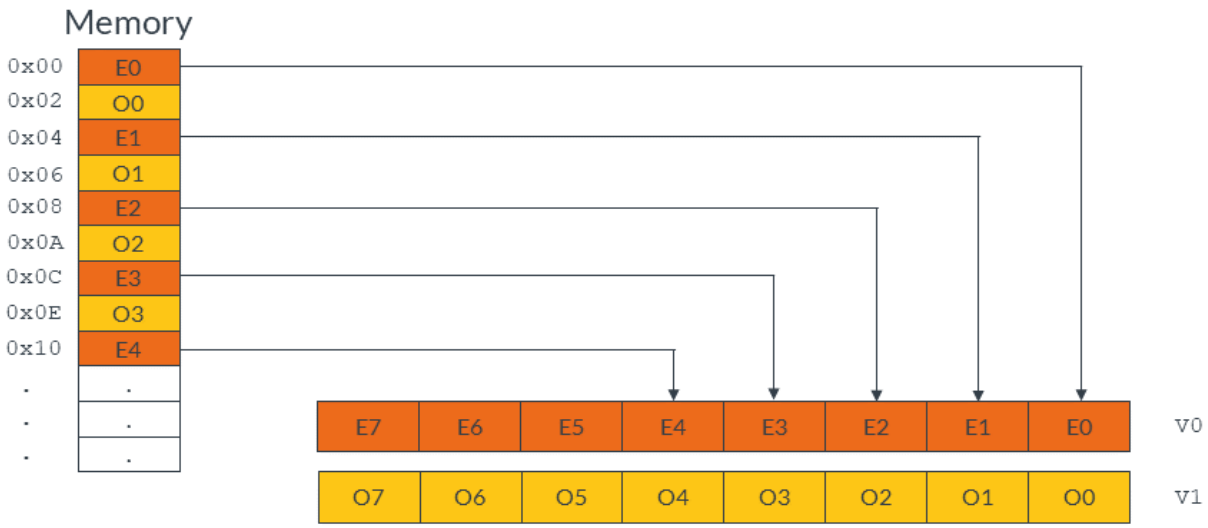
3.3 Element types

Loads and stores interleave elements based on the size that is specified to the instruction.

For example, consider the following instruction:

```
LD2 {V0.8H, V1.8H}, [X0]
```

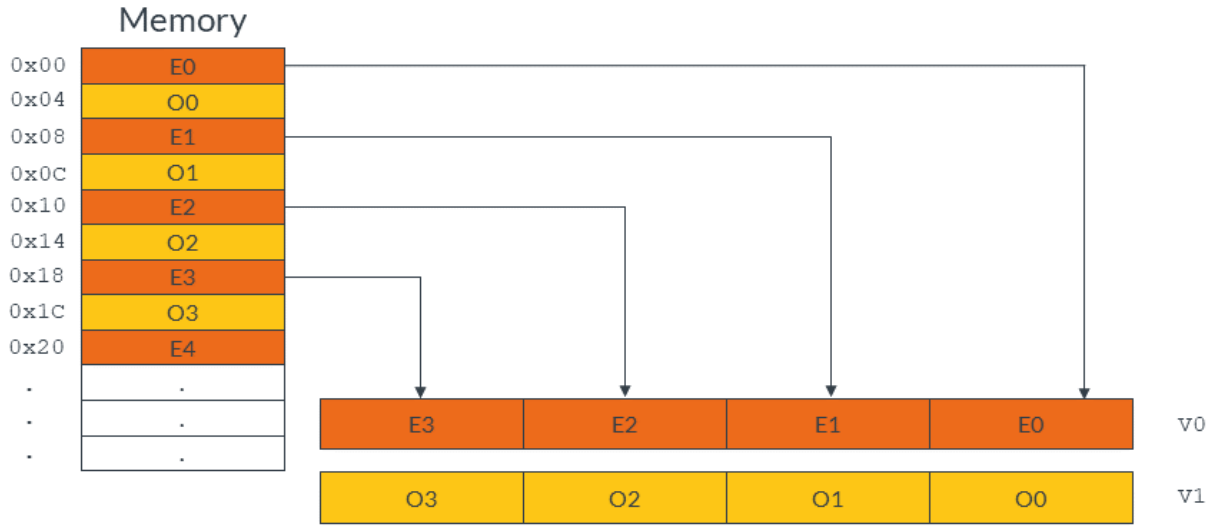
This instruction loads two Neon registers with deinterleaved data starting from the memory address in `X0`. The `8H` in the arrangement specifier indicates that each element is a 16-bit halfword (`H`), and each Neon register is loaded with eight elements. This instruction therefore results in eight 16-bit elements in the first register `V0`, and eight 16-bit elements in the second register `V1`. Adjacent pairs (even and odd) are separated to each register, as shown in the following diagram:



The following instruction uses the arrangement specifier `4S`, changing the element size to 32-bits:

```
LD2 {V0.4S, V1.4S}, [X0]
```

Changing the element size to 32-bits loads the same amount of data, but now only four elements make up each vector, as shown in the following diagram:



Element size also affects endianness handling. In general, if you specify the correct element size to the load and store instructions, bytes are read from memory in the appropriate order. This means that the same code works on little-endian systems and big-endian systems.

Finally, element size has an impact on pointer alignment. Alignment to the element size generally gives better performance, and it might be a requirement of your target operating system. For example, when loading 32-bit elements, align the address of the first element to at least 32-bits.

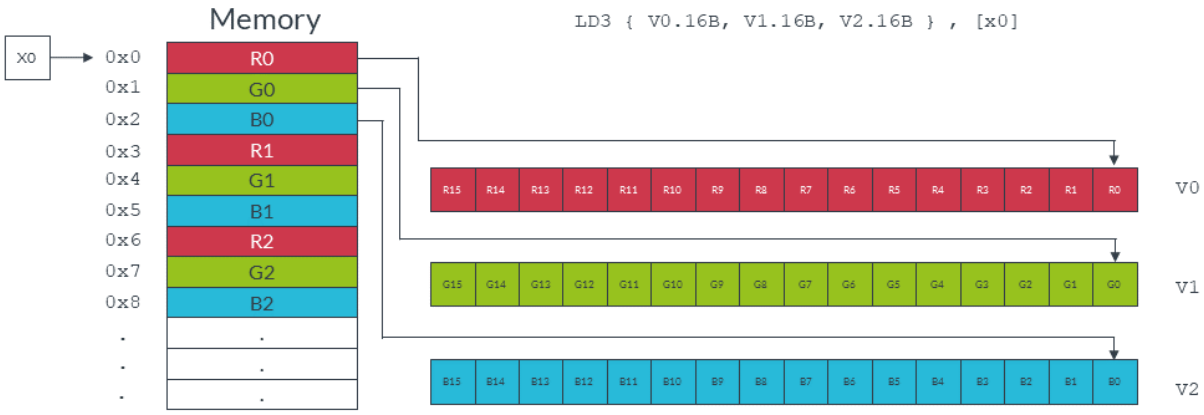
3.4 Single or multiple elements

In addition to loading multiple elements, structure loads can also read single elements from memory with deinterleaving. Data can either be replicated to all lanes of a Neon register, or inserted into a single lane, leaving the other lanes intact.

For example, the following instruction loads a single three-element data structure from the memory address pointed to by `X0`, then replicates that data into all lanes of three Neon registers:

```
LD3R  { V0.16B, V1.16B, V2.16B } , [x0]
```

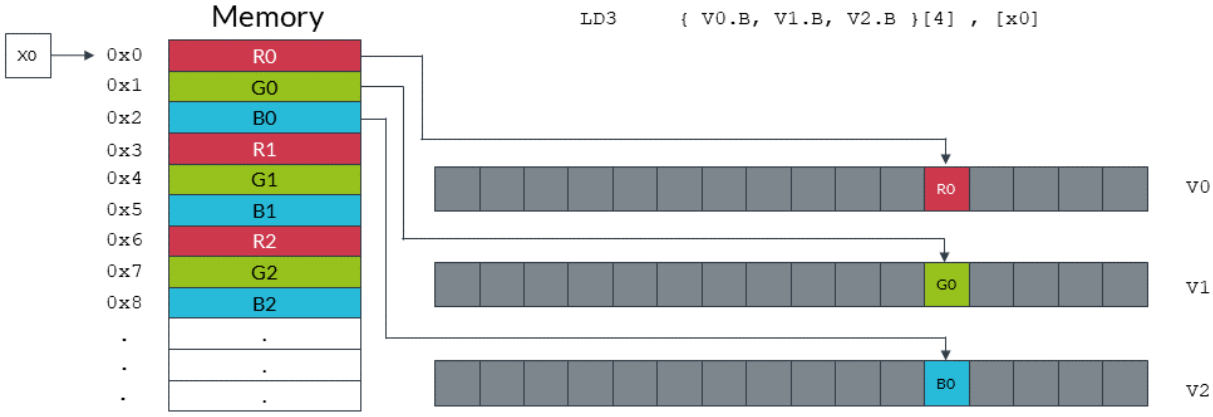
The following diagram shows the operation of this instruction:



By contrast, the following instruction loads a single three-element data structure into a single lane of three Neon registers, leaving the other lanes intact:

```
LD3     { V0.B, V1.B, V2.B }[4] , [x0]
```

The following diagram shows the operation of this instruction. This form of the load instruction is useful when you need to construct a vector from data scattered in memory.



Stores are similar, providing support for writing single or multiple elements with interleaving.

3.5 Addressing

Structure load and store instructions support three formats for specifying addresses:

- Register (no offset): `[Xn]`

  This is the simplest form. Data is loaded and stored to the address that is specified by `Xn`.

- Register with post-index, immediate offset: `[Xn], #imm`

  Use this form to update the pointer in `Xn` after loading or storing, ready to load or store the next elements.

  The immediate increment value `#imm` must be equal to the number of bytes that is read or written by the instruction.

  For example, the following instruction loads 48 bytes of data, using three registers, each containing 16 x 1 byte data elements. This means that the immediate increment is 48:

  ```
  LD3     { V0.16B, V1.16B, V2.16B }, [x0], #48
  ```

  However, the next example loads 32 bytes of data, using two registers, each containing 2 x 8 byte data elements. This means that the immediate increment is 32:

  ```
  LD2     { V0.2D, V1.2D}, [x0], #32
  ```

- Register with post-index, register offset: `[Xn], Xm`

  After the memory access, increment the pointer by the value in register `Xm`. This form is useful when reading or writing groups of elements that are separated by fixed widths, for example when reading a vertical line of data from an image.

3.6 Other types of loads and stores

This guide only deals with structure loads and stores. However, Neon also provides other types of load and store instruction, including:

- `LDR` and `STR` to load and store single Neon registers.
- `LDP` and `STP` to load or store pairs of Neon registers.

For more details on supported load and store operations, see the Arm Architecture Reference Manual.

Detailed cycle timing information for the instructions can be found in the Technical Reference Manual for each core.

4 Related information

Here are some resources related to material in this guide:

- Neon Programmer's Guide for Armv8-A
- SIMD ISAs on Arm Developer
- Armv8-A Neon optimization presentation video