



Learn the architecture - A64 Instruction Set Architecture

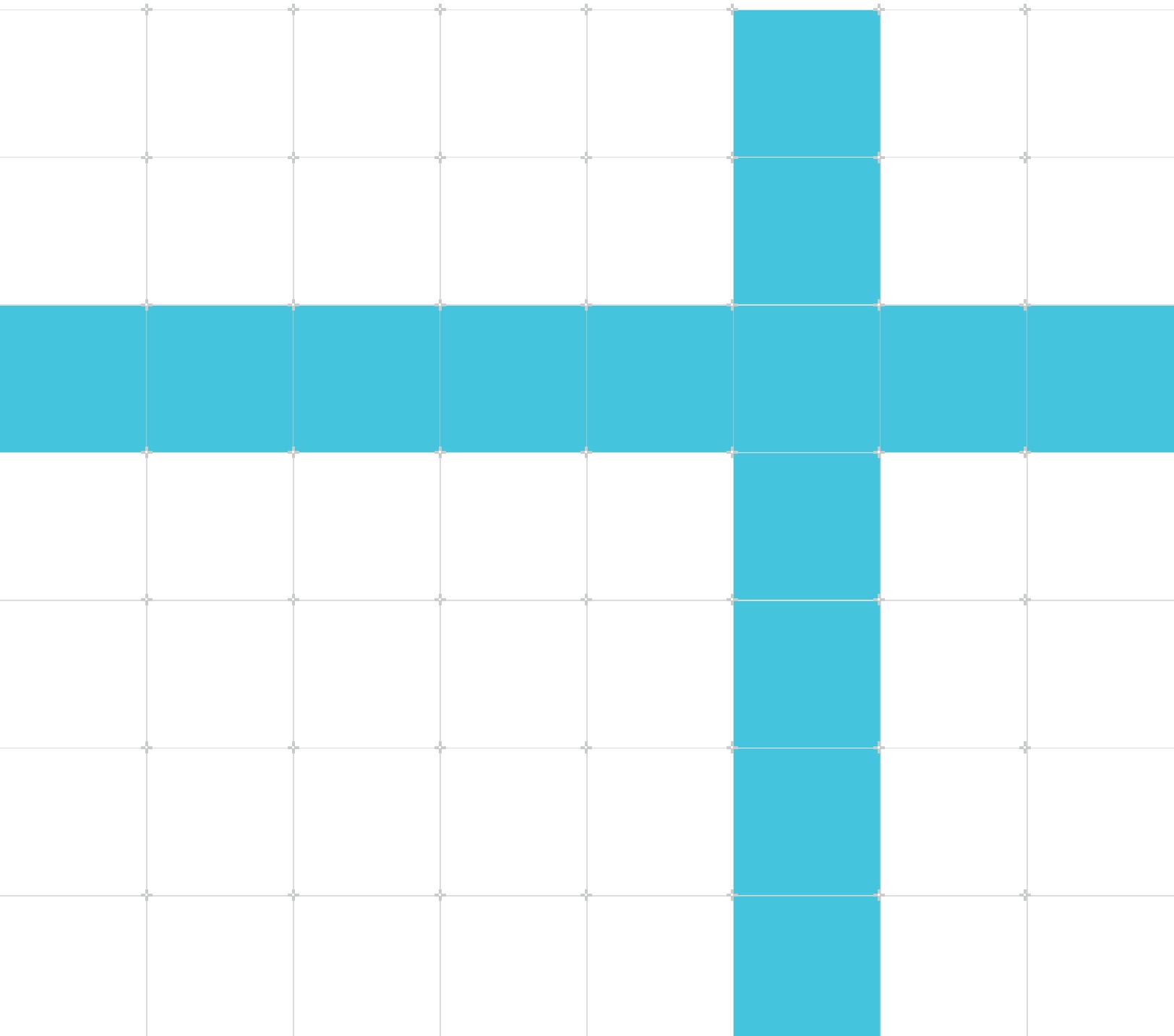
1.3

Guide

Non-Confidential

Issue 02

Copyright © 2022–2025 Arm Limited (or its affiliates). 102374_0103_02_en
All rights reserved.



Learn the architecture - A64 Instruction Set Architecture Guide

Copyright © 2022–2025 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0103-02	25 September 2025	Non-Confidential	Updates to support 2025 extensions
0103-01	19 September 2025	Non-Confidential	Updates to Loads and stores - load pair and store pair chapter
0102-02	2 December 2024	Non-Confidential	Updates to images
0102-01	5 December 2023	Non-Confidential	Updates to floating point chapter
0101-02	14 June 2023	Non-Confidential	Fix typo in Loads and stores chapter
0101-01	3 November 2022	Non-Confidential	Updates to vector matrix data, floating point and Related information chapters
0100-03	14 July 2022	Non-Confidential	Fix typo in “Logical and integer arithmetic instruction format” diagram
0100-02	30 June 2022	Non-Confidential	Fix typo in TBZ instruction in Program flow - loops and decisions
0100-01	25 April 2022	Non-Confidential	Initial release

Proprietary Notice

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

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

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

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

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

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

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 this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm's trademark usage

guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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.....	7
2. Why you should care about the ISA.....	8
3. Instruction sets in the Arm architecture.....	9
4. Instruction set resources.....	10
5. Simple sequential execution.....	11
6. Registers in AArch64 - general-purpose registers.....	12
7. Registers in AArch64 - other registers.....	14
8. Registers in AArch64 - system registers.....	15
9. Data processing - arithmetic and logic operations.....	17
10. Data processing - floating point.....	19
10.1 Support for 8-bit and 16-bit floating point.....	19
10.2 Is floating point support optional?.....	20
11. Data processing - bit manipulation.....	21
12. Data processing - extension and saturation.....	23
12.1 Test yourself.....	24
12.2 Sub-register-sized integer data processing.....	24
13. Data processing - format conversion.....	26
14. Data processing - vector and matrix data.....	28
15. Loads and stores.....	30
16. Loads and stores - size.....	31
17. Loads and stores - zero and sign extension.....	32

17.1 Task.....	33
18. Loads and stores - addressing.....	34
19. Loads and stores - load pair and store pair.....	38
20. Loads and stores - using floating point registers.....	40
21. Loads and stores - specialist instructions.....	41
21.1 Load-Acquire and Release.....	41
21.2 The 64-byte atomic load and stores.....	41
21.3 Loads and stores to optimize memcpy() style operations.....	42
22. Program flow.....	43
23. Program flow - loops and decisions.....	44
23.1 Unconditional branch instructions.....	44
23.2 Conditional branch instructions.....	44
23.3 Test yourself.....	45
24. Program flow - generating condition code.....	46
24.1 Test yourself.....	47
25. Program flow - conditional select instructions.....	48
26. Function calls.....	50
27. Procedure Call Standard.....	51
28. System calls.....	53
29. Check your knowledge.....	54
30. Related information.....	55
31. Next steps.....	56

1. Overview

An Instruction Set Architecture (ISA) is part of the abstract model of a computer. It defines how software controls the processor.

The Arm ISA allows you to write software and firmware that conforms to the Arm specifications. This means that, if your software or firmware conforms to the specifications, any Arm-based processor will execute it in the same way.

This guide introduces the A64 instruction set, used in the 64-bit Armv8-A architecture, also known as AArch64.

We will not cover every single instruction in this guide. All instructions are detailed in the [Arm Architecture Reference Manual](#). Instead, we will introduce the format of the instructions, the different types of instruction, and how code written in assembler can interact with compiler-generated code.

At the end of this guide, you can [Check your knowledge](#). You will have learned about the main classes of instructions, the syntax of data-processing instructions, and how the use of W and X registers affects instructions. The key outcome that we hope you will learn from this guide is to be able to explain how generated assembler code maps to C statements, when given a C program and the compiler output for it. Finally, this guide will show you how to write a function in assembler that can be called from C.

2. Why you should care about the ISA

As developers, you may not need to write directly in assembler in our day-to-day role. However, assembler is still important in some areas, such as the first stage boot software or some low-level kernel activities.

Even if you are not writing assembly code directly, understanding what the instruction set can do, and how the compiler makes use of those instructions, can help you to write more efficient code. It can also help you to understand the output of the compiler. This can be useful when debugging.

3. Instruction sets in the Arm architecture

Armv8-A supports three instruction sets: A32, T32 and A64.

The A64 instruction set is used when executing in the AArch64 Execution state. It is a fixed-length 32-bit instruction set. The 64 in the name refers to the use of this instruction by the AArch64 Execution state. It does not refer to the size of the instructions in memory.

The A32 and T32 instruction sets are also referred to as Arm and Thumb, respectively. These instruction sets are used when executing in the AArch32 Execution state. In this guide, we do not cover the A32 and T32 instruction sets. To find out more about these instruction sets, see [Related information](#).

4. Instruction set resources

Each version of the Arm architecture has its own Arm Architecture Reference Manual, which can be found on the Arm Developer website. Every Arm ARM provides a detailed description of each instruction, including:

- Encoding - the representation of the instruction in memory.
- Arguments - inputs to the instruction.
- Pseudocode - what the instruction does, as expressed in Arm pseudocode language.
- Restrictions - when the instruction cannot be used, or the exceptions it can trigger.

The instruction descriptions for A64 are also available in XML and HTML. The XML and HTML formats are useful if you need to refer to the instructions often. The XML and HTML formats can be found on the Arm Developer website. You can find a link in [Related information](#). The XML can be downloaded as a compressed archive and the HTML can be viewed and searched using a web browser.



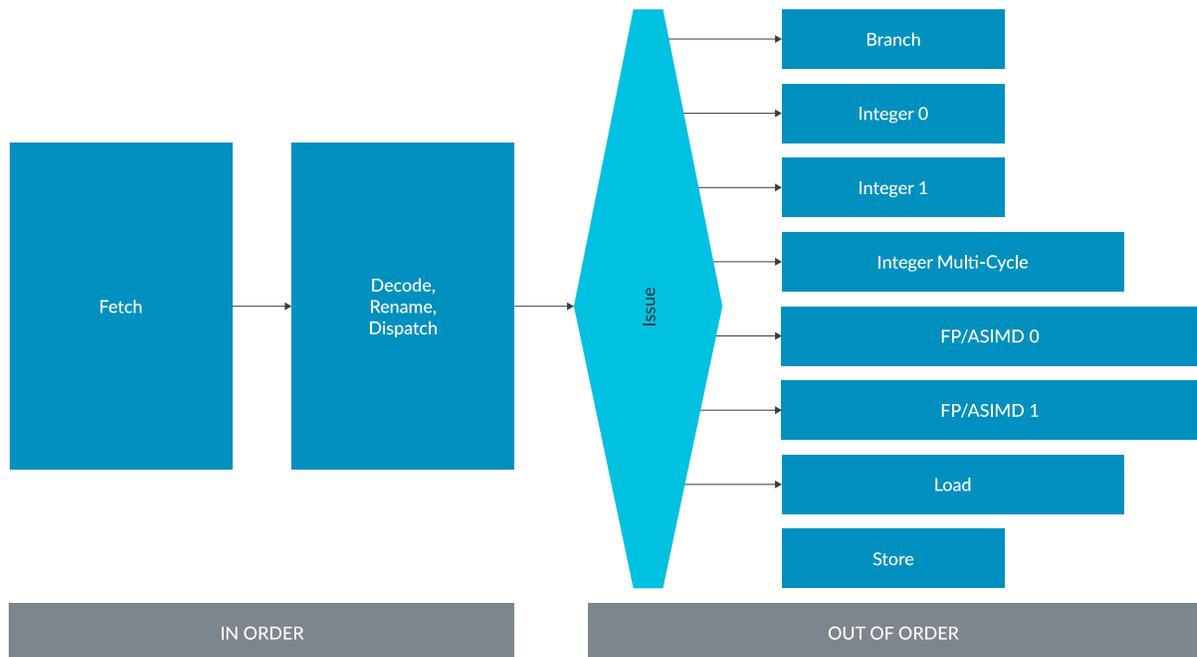
The information in the XML/HTML and the Arm Architecture Reference Manual are taken from the same source but may be formatted slightly differently.

5. Simple sequential execution

The Arm architecture describes instructions following a Simple Sequential Execution (SSE) model. This means that the processor behaves as if the processor fetched, decoded and executed one instruction at a time, and in the order in which the instructions appeared in memory.

In practice, modern processors have pipelines that can execute multiple instructions at once, and may do so out of order. This diagram shows an example pipeline for an Arm Cortex processor:

Figure 5-1: Example Cortex pipeline



You will remember that the architecture is a functional description. This means that it does not specify how an individual processor works. Each processor must behave consistently with the simple sequential execution model, even if it is reordering instructions internally.

6. Registers in AArch64 - general-purpose registers

Most A64 instructions operate on registers. The architecture provides 31 general purpose registers. Each register can be used as a 64-bit X register (X0..X30), or as a 32-bit W register (W0..W30). These are two separate ways of looking at the same register. For example, this register diagram shows that W0 is the bottom 32 bits of X0, and W1 is the bottom 32 bits of X1:

Figure 6-1: Register diagram



For data processing instructions, the choice of `x` or `w` determines the size of the operation. Using `x` registers will result in 64-bit calculations, and using `W` registers will result in 32-bit calculations. This example performs a 32-bit integer addition:

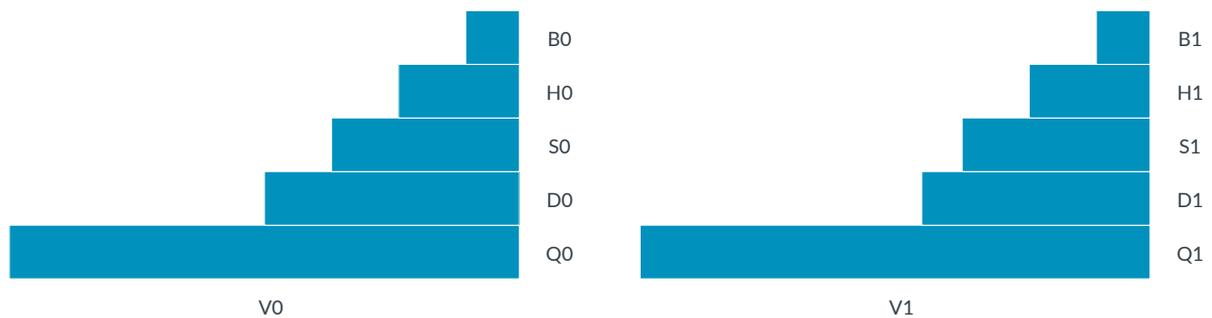
```
ADD W0, W1, W2
```

This example performs a 64-bit integer addition:

```
ADD X0, X1, X2
```

When a `w` register is written, as seen in the example above, the top 32 bits of the 64-bit register are zeroed.

There is a separate set of 32 registers used for floating point and vector operations. These registers are 128-bit, but like the general-purpose registers, can be accessed in several ways. `Bx` is 8 bits, `Hx` is 16 bits and so on to `Qx` which is 128 bits.

Figure 6-2: Register diagram

The name you use for the register determines the size of the calculation. This example performs a 32-bit floating point addition:

```
FADD S0, S1, S2
```

This example performs a 64-bit floating point addition:

```
FADD D0, D1, D2
```

These registers can also be referred to as *v* registers. When the *v* form is used, the register is treated as being a vector. This means that it is treated as though it contains multiple independent values, instead of a single value. This example performs vector floating point addition:

```
FADD V0.2D, V1.2D, V2.2D
```

This example performs vector integer addition:

```
ADD V0.2D, V1.2D, V2.2D
```

We will look at vector instructions in more detail later in this guide.

7. Registers in AArch64 - other registers

Here are some other registers in the A64 that you should know about:

- The zero registers, `xZR` and `wZR`, always read as 0 and ignore writes.
- You can use the stack pointer (`sP`) as the base address for loads and stores. You can also use the stack pointer with a limited set of data-processing instructions, but it is not a regular general purpose register. Armv8-A has multiple stack pointers, and each one is associated with a specific Exception level. When `sP` is used in an instruction, it means the current stack pointer. The guide to the exception model explains how the stack pointer is selected.
- X30 is used as the Link Register and can be referred to as `LR`. Separate registers, `ELR_ELx`, are used for returning from exceptions. This is discussed in more detail in the guide to the exception model.
- The Program Counter (`PC`) is not a general-purpose register in A64, and it cannot be used with data processing instructions. The PC can be read using:

```
ADR Xd, .
```

The `ADR` instruction returns the address of a label, calculated based on the current location. Dot (.) means 'here', so the shown instruction is returning the address of itself. This is equivalent to reading the `PC`. Some branch instructions, and some load/store operations, implicitly use the value of the `PC`.



Note

In the A32 and T32 instruction sets, the `PC` and `sP` are general purpose registers. This is not the case in A64 instruction set.

8. Registers in AArch64 - system registers

As well as general purpose registers, the architecture defines system registers. System registers are used to configure the processor and to control systems such as the MMU and exception handling.

System registers cannot be used directly by data processing or load/store instructions. Instead, the contents of a system register need to be read into an x register, operated on, and then written back to the system register. There are two specialist instructions for accessing system registers:

```
MRS      Xd, <system register>
```

reads the system register into xd.

```
MSR      <system register>, Xn
```

writes xn to the system register.

System registers are specified by name, for example SCTLR_EL1:

```
MRS      X0, SCTLR_EL1
```

reads SCTLR_EL1 into x0.

System register names end with _ELx. The _ELx specifies the minimum privilege necessary to access the register. For example:

```
SCTLR_EL1
```

requires EL1 or higher privilege.

```
SCTLR_EL2
```

requires EL2 or higher privilege.

```
SCTLR_EL3
```

requires EL3 privilege

Attempting to access the register with insufficient privilege results in an exception.

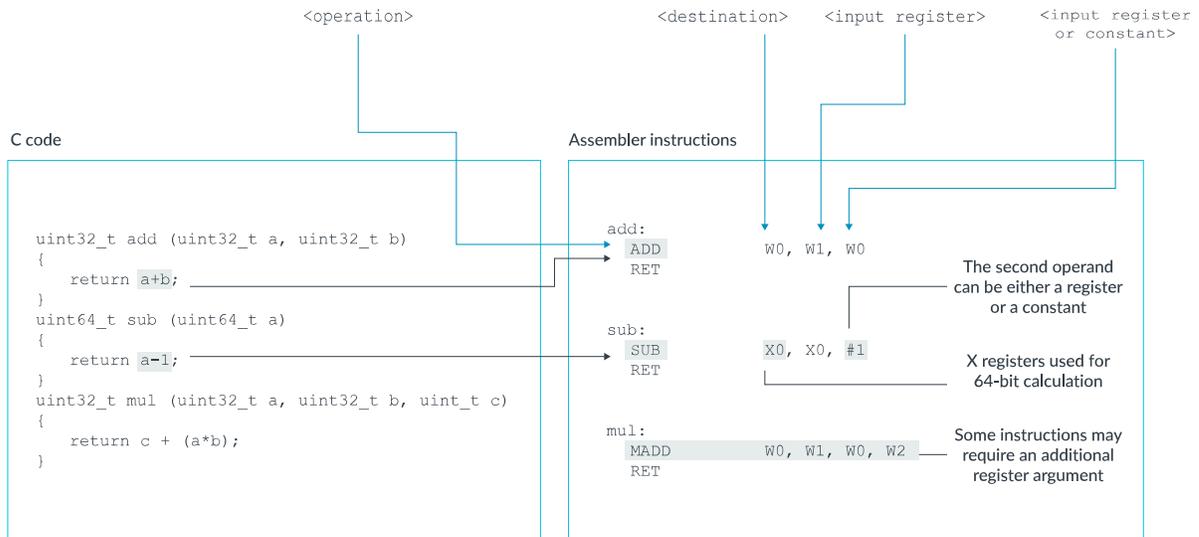


Sometimes you will see _EL12 or _EL01. These are used as part of virtualization. Refer to the guide on virtualization for more information.

9. Data processing - arithmetic and logic operations

The basic format of logical and integer arithmetic instructions is:

Figure 9-1: Logical and integer arithmetic instruction format



The parts of the instruction are as follows:

- Operation defines what the instruction does. For example, ADD does addition and AND performs a logical AND. An S can be added to the operation to set flags. For example, ADD becomes ADDS. This s tells the processor to update the ALU flags based on the result of instruction. We discuss ALU flags in the section on generating condition code.
- Destination is the destination of the instruction is always a register, and specifies where the result of the operation is placed. Most instructions have a single destination register. A few instructions have two destination registers. When the destination is a W register, the upper 32 bits of the corresponding X register are set to 0.
- Operand 1 will always be a register. This is the first input to the instruction.
- Operand 2 will be a register or a constant, and is the second input to the instruction. When operand 2 is a register, it may include an optional shift. When operand 2 is a constant, it is encoded within the instruction itself. This means that the range of constants available is limited.

You should be aware of a couple of special cases, such as the MOV and MVN instructions. MOV moves a constant, or the contents of another register, into the register specified as the destination.

MOV and MVN only require a single input operand, which can be either a register or a constant, as shown here:

```
MOV      X0, #1
```

sets: $x0 = 1$

```
MVN     W0, W1
```

sets: $w0 = \sim w1$

10. Data processing - floating point

Floating-point operations follow the same format as integer data-processing instructions and use floating-point registers. Like with the integer data-processing instructions, the size of the operation determines the size of the register that is used. The operation part of a floating-point instruction always starts with an F. For example, this instruction sets $H0 = H1 / H2$ with half precision:

```
FDIV      H0, H1, H2
```

This instruction sets $S0 = S1 + S2$ with single precision:

```
FADD      S0, S1, S2
```

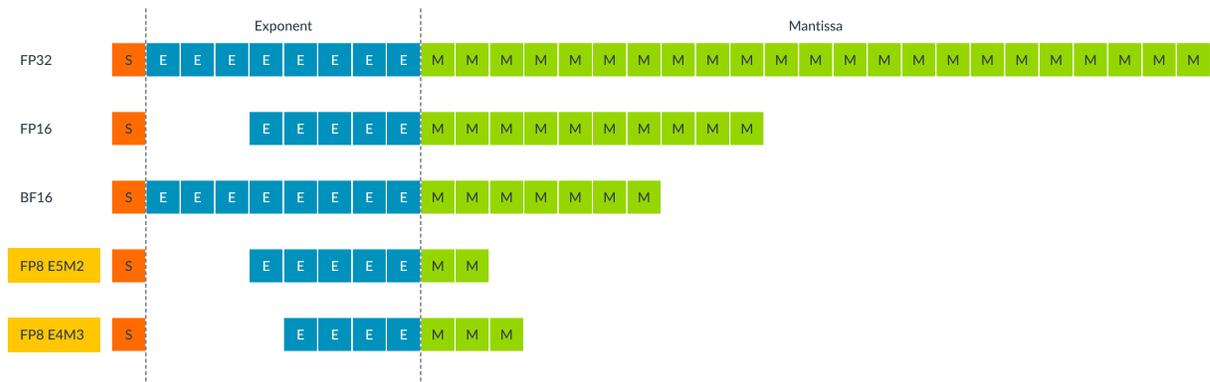
This instruction sets $D0 = D1 - D2$ with double precision:

```
FSUB      D0, D1, D2
```

10.1 Support for 8-bit and 16-bit floating point

The Arm architecture includes support for 16-bit and 8-bit floating-point formats.

Figure 10-1: Differences between data formats



Armv8.2-A added support for 16-bit floating point, FP16 (FEAT_FP16). Support for FP16 is optional from Armv8.2-A, becoming mandatory if SVE or SVE2 is implemented, meaning that it is effectively mandatory in Armv9-A.

BFloat16, often abbreviated to BF16, is an alternative 16-bit floating-point storage format. BF16 is optionally supported from Armv8.2-A (FEAT_BF16), but becomes mandatory in Arm8.6-A and Armv9.1-A. BF16 has recently emerged as a format tailored specifically to high-performance processing of Neural Networks (NNs)

The difference between the FP16 and BF16 formats is how the bits are divided between the exponent and fraction, as shown in the diagram above. BF16 has the same exponent range as FP32, but with fewer bits for the fraction. This makes converting between BF16 and FP32 simpler, as BF16 is effectively a lower precision version of FP32.

For more information on BF16, see [AWS Graviton3 featuring Arm Neoverse V1 is up to 1.8x faster than x86 for deep learning inference workloads](#) blog.

In 2022 Arm, Intel, and Nvidia announced their collaboration on FP8, an interchange format that allows software ecosystems to share NN models easily and support the continuous advancement of AI computing capabilities. This became the [OCP8 standard](#) and support was introduced to the architecture in Armv9.2-A (FEAT_FP8, FEAT_FP8xxx).

FP8 introduces a pair of formats: E5M2 and E4M3. These two formats give different trade-offs between precision and range.

When working with FP8, which format (E5M2 and E4M3) is used is selected by fields in the FPMR register. Different formats can be selected for the different inputs to an instruction, allowing for efficient working with datasets in different formats.

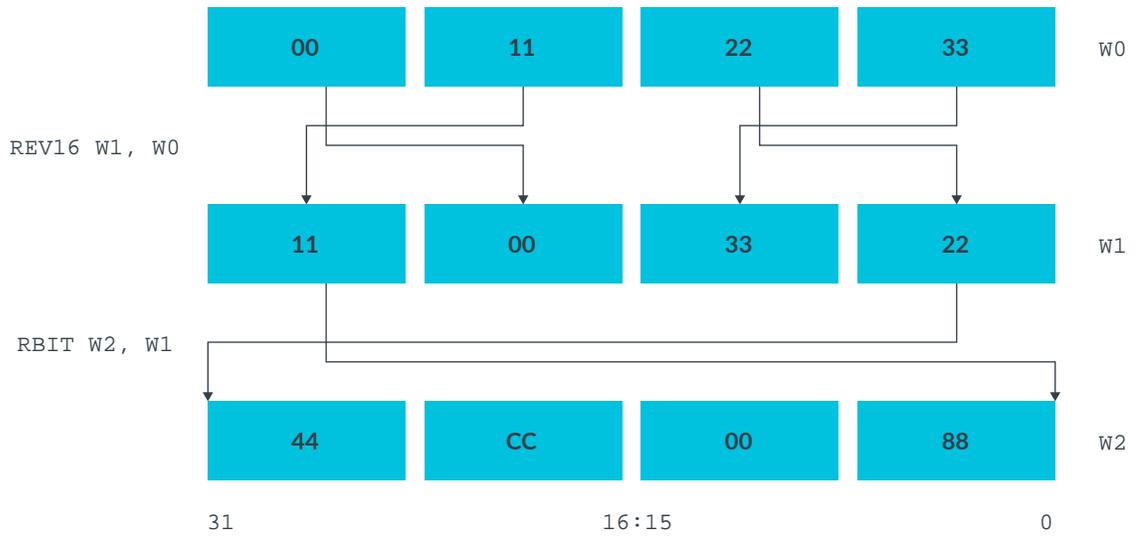
To support other small size data formats, Armv9.6-A and Armv9.7-A introduced Lookup Table (LUT) instructions (FEAT_LUT, SVE2.3 and SME2.3). These instructions allow software to efficiently convert from 2-, 4-, or 6-bit formats to other formats for processing.

10.2 Is floating point support optional?

No. Support for floating point is mandatory in Armv8-A. The architecture specifies that it is required whenever a rich operating system, such as Linux, is used.

You are technically permitted to omit floating point support, if you are running an entirely proprietary software stack. Most toolchains, including GCC and Arm Compiler 6, will assume floating point support.

Figure 11-2: Instructions that reverse byte or bit order

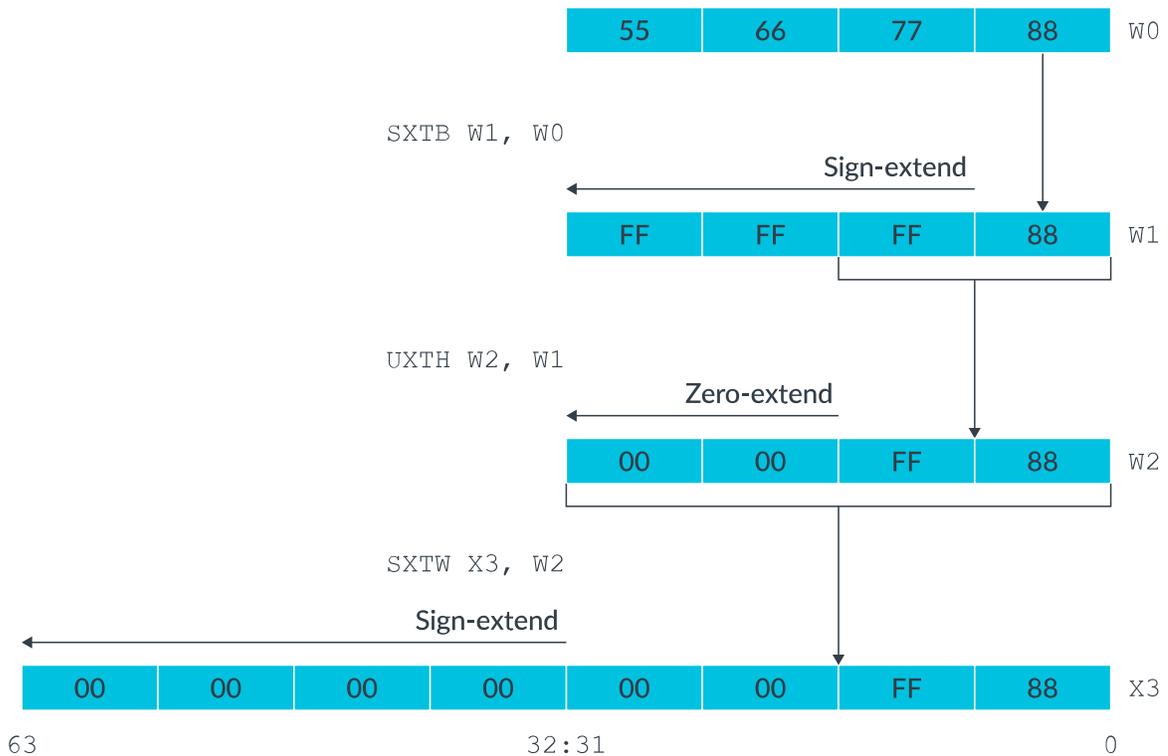


`REV16` and `RBIT` are particularly useful when you are handling data that is in a different endianness.

12. Data processing - extension and saturation

Sometimes it is necessary to convert data from one size to another. The `SXTx` (sign extend) and `UXTx` (unsign extend) instructions are available for this conversion. In this conversion, the `x` determines the size of the data being extended, as shown in this figure:

Figure 12-1: Data size conversion example



In the first instruction `SXTB`, the `B` means byte. It takes the bottom byte of `w0` and sign extends it to 32 bits.

`UXTH` is an unsigned extension of a halfword (`H`). It takes the bottom 16 bits of `W1` and zero extends it to 32 bits.

The first two examples have `w` registers as a destination, meaning the extension is to 32 bits. The third example has an `x` register, meaning the sign extension is to 64 bits.

12.1 Test yourself

Write an instruction to sign extend a byte in W5 to 64 bits, placing the result in X7.

```
SXTB X7, W5
```

12.2 Sub-register-sized integer data processing

Some instructions perform saturating arithmetic. This means that if the result is larger or smaller than the destination can hold, then the result is set to the largest or smallest value of the destination's integer range.

The data-processing instructions can efficiently handle 32-bit data and 64-bit data. In practice, you often see saturation instructions when handling sub-register calculations. Sub-register calculations are calculations of 16 bits or 8 bits. The following examples show sub-register calculations in C and the generated assembler code.

Example 1

C code:

```
uint32_t add32(uint32_t a, uint32_t b) {  
    return a+b;  
}
```

Generated assembler:

```
add32:  
    ADD W0, W1, W0  
    RET
```

Example 2

C code:

```
uint16_t uadd16(uint16_t a, uint16_t b) {  
    return a+b;  
}
```

Generated assembler:

```
uadd16:  
    AND W8, W1, #0xffff  
    ADD W0, W8, W0, UXTH  
    RET
```

Example 3

C code:

```
int16_t sadd16(int16_t a, int16_t b) {  
    return a+b;  
}
```

Generated assembler:

```
sadd16:  
    SXTH    W8, W1  
    ADD     W0, W8, W0, SXTH  
    RET
```

In the first example, the 32-bit addition maps onto *w* registers and therefore can be handled easily.

For the 16-bit examples, an extra instruction is necessary. The third example takes the 16-bit inputs, extends them to 32 bits, and then performs the addition. The sequence converts the 16-bit input to 32 bits, using:

```
SXTH    W8, W1
```

Then, this instruction performs the addition and saturates the result to signed 16 bits:

```
ADD     W0, W8, W0, SXTH
```

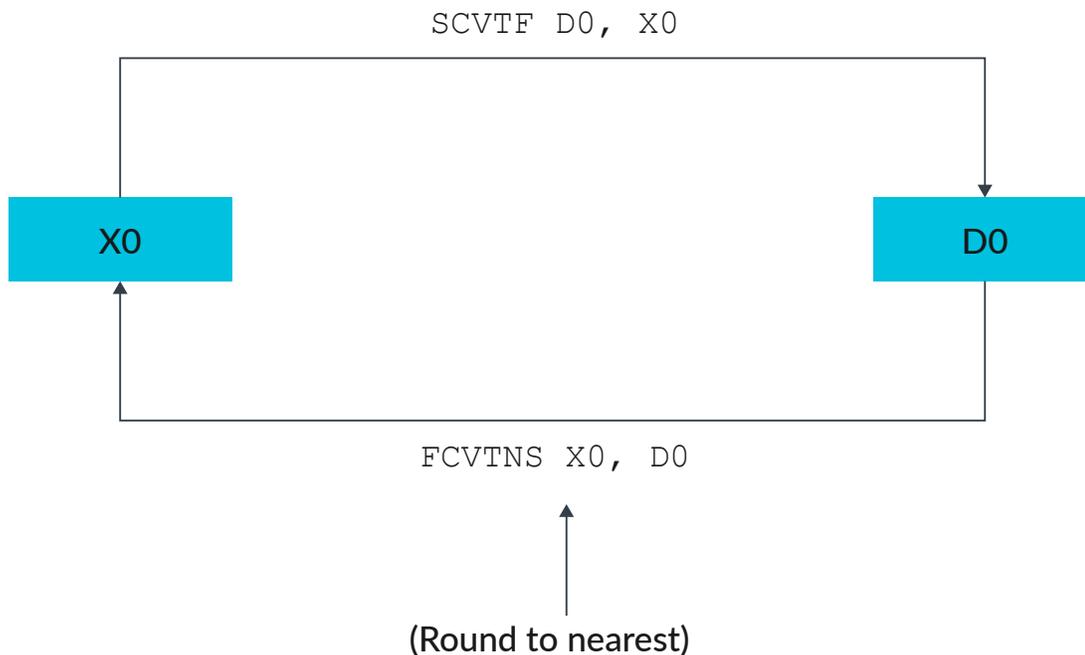
Adding `, sxth` to the end of the operand list of the `ADD` operation causes the result to use saturating arithmetic. Because the destination is a *w* register, the `ADD` instruction saturates to a 16-bit integer range.

13. Data processing - format conversion

We have seen that the `MOV` and `MVN` instructions copy the value from one register to another. Similarly, `FMOV` can be used to copy between floating-point and general purpose registers.

However, using `FMOV` copies the literal bit pattern between the registers. There are also instructions that can convert to the closest representation, as this figure shows:

Figure 13-1: FMOV example



In this example, imagine that `x0` contains the value 2 (positive integer 2):

```
X0 = 0x0000_0000_0000_0002
```

Then, the following sequence is executed:

```
FMOV D0, X0
SCVTF D1, X0
```

Both instructions “copy” `x0` into a `D` register. However, the results are quite different:

```
D0 = 0x0000_0000_0000_0002 = 9.88131e-324  
D1 = 0x4000_0000_0000_0002 = 2.0
```

The `FMov` copied the literal bit pattern, which is a very different value when interpreted as a floating-point value. The `scvTF` converted the value in `x0` to the closest equivalent in floating-point.

Similarly, `FCvTxx` can be used to convert a floating-point value to its closest integer representation. In this instance, different values of `xx` control the rounding mode used.

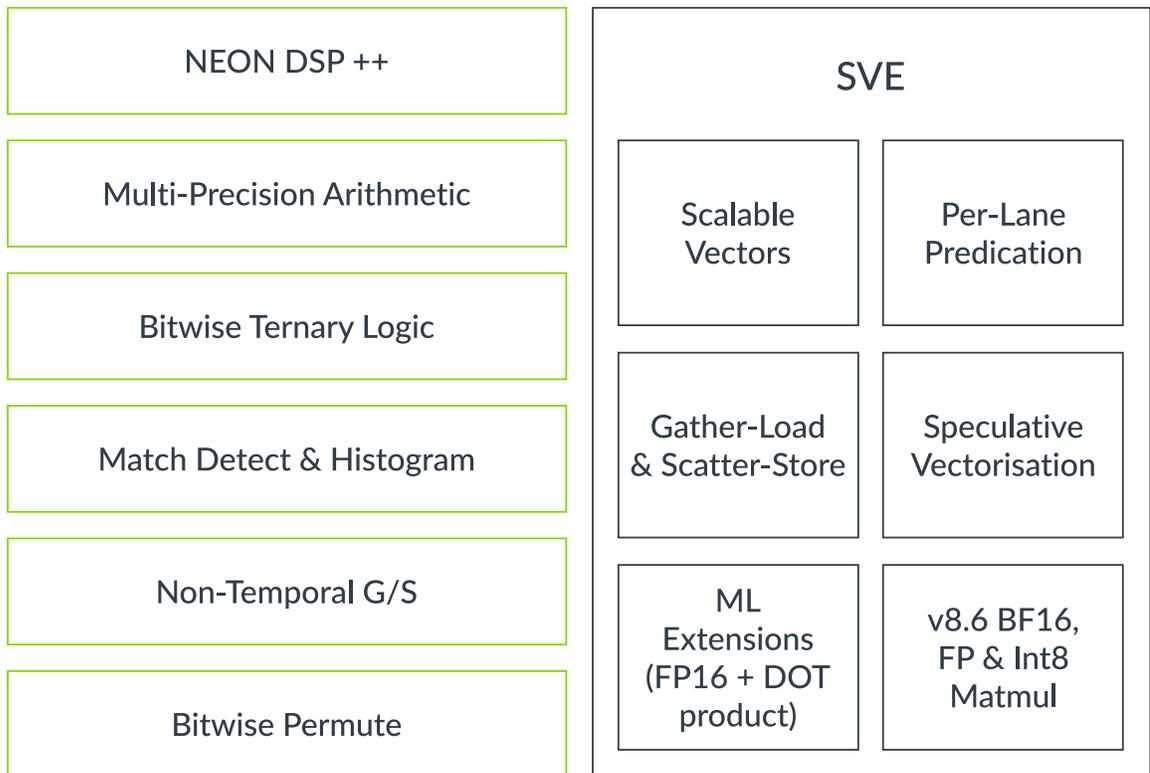
14. Data processing - vector and matrix data

The A64 architecture also provides support for vector data processing. The two types of vector processing available are:

- Advanced SIMD, which is also known as NEON.
- Scalable Vector Extension (SVE and SVE2). SVE was introduced in Armv8-A and was optimised for HPC workloads. Armv9-A introduced SVE2, which extends the base SVE to enable more use cases.

Figure 14-1: SVE and SVE2

SVE2



We will cover both types of vector processing in a later guide on vector programming.



The name Advanced SIMD derives from the existence of SIMD instructions that operated on regular 32-bit general-purpose registers in Armv6. In Armv7, the term Advanced SIMD was used for instructions that could operate on 128-bit vectors. The Armv6 style instructions do not exist in A64, but the naming convention remains.

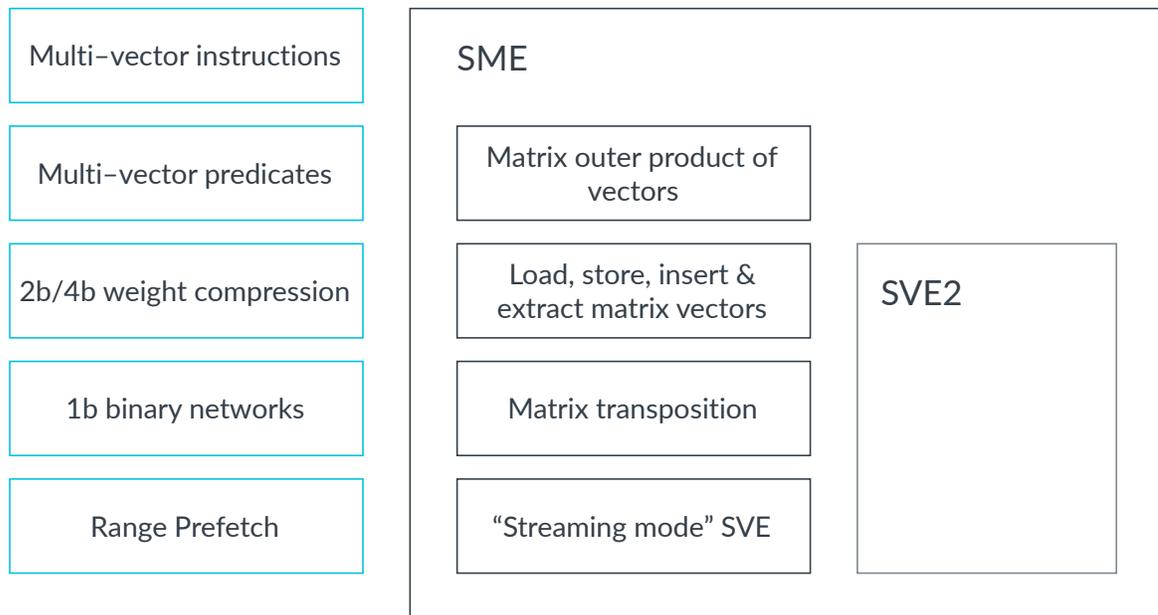
Armv9-A also introduced the optional Scalable Matrix Extensions (SME and SME2). SME builds on SVE2, adding new capabilities to efficiently process matrices. Key features include:

- Matrix tile storage
- Load, store, insert, and extract tile vectors, including on-the-fly transposition
- Outer product of SVE vectors
- Streaming SVE mode

SME provides outer-product instructions to accelerate matrix operations. SME2 significantly extends the capabilities with instructions for multi-vector operations, multi-vector predicates, range prefetches and 2b/4b weight compression.

Figure 14-2: SME and SME2

SME2



The new instructions enable SME2 to accelerate more workloads than the original SME. Including `GEMV`, Non-Linear Solvers, Small and Sparse Matrices, and Feature Extraction or tracking.

15. Loads and stores

The basic load and store operations are: `LDR` (load) and `STR` (store). These operations transfer a single value between memory and the general-purpose registers. The syntax for these instructions is:

```
LDR<Sign><Size>  <Destination>, [<address>]  
STR<Size>        <Source>, [<address>]
```

16. Loads and stores - size

The size of the load or store is determined by the register type `x` or `w` and the `size` field. `x` is used for 64 bits and `w` is used for 32 bits. For example, this instruction loads 32 bits from address into `w0`:

```
LDR    w0, [<address>]
```

This instruction loads 64 bits from `address` into `x0`:

```
LDR    x0, [<address>]
```

The `size` field allows you to load a sub-register sized quantity of data. For example, this instruction stores the bottom byte (`B`) of `w0` to `address`:

```
STRB   w0, [<address>]
```

This instruction stores the bottom halfword (`H`) of `w0` to `address`:

```
STRH   w0, [<address>]
```

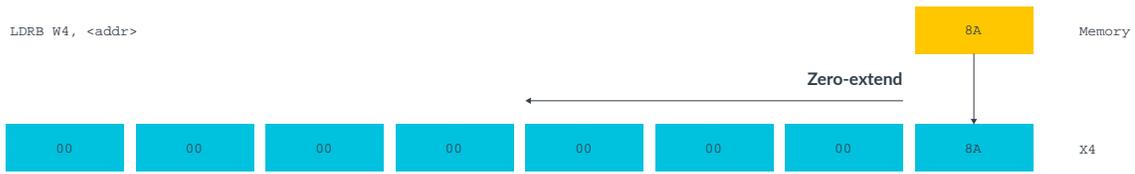
Finally, this instruction stores the bottom word (`w`) of `x0` to `address`:

```
STRW   x0, [<address>]
```

17. Loads and stores - zero and sign extension

By default, when a sub-register-sized quantity of data is loaded, the rest of the register is zeroed, as shown in this figure:

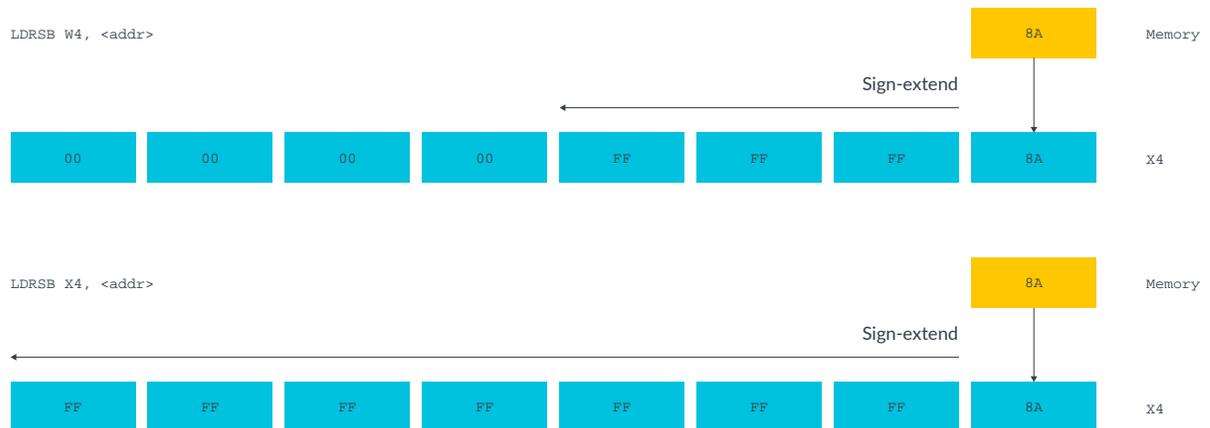
Figure 17-1: Zeroed register example



Remember that whenever a w register is written, the top half of the x register is zeroed.

Adding an s to the operation causes the value to be sign extended instead. How far the size extension goes depends on whether the target is a w or x register, as shown in this figure:

Figure 17-2: Size extension example



17.1 Task

If the byte at address `0x8000` contains the value `0x1F`, what would be the result of `LDRSB x4`?

`LDRSB` is performing a byte load with sign extension to 64 bits. The most significant bit of the loaded value will be replicated to fill the 64-bit register. The loaded value, `0x1F`, has its top bit clear. Therefore, the value in `x4` will be `0x0000_0000_0000_001F`.

18. Loads and stores - addressing

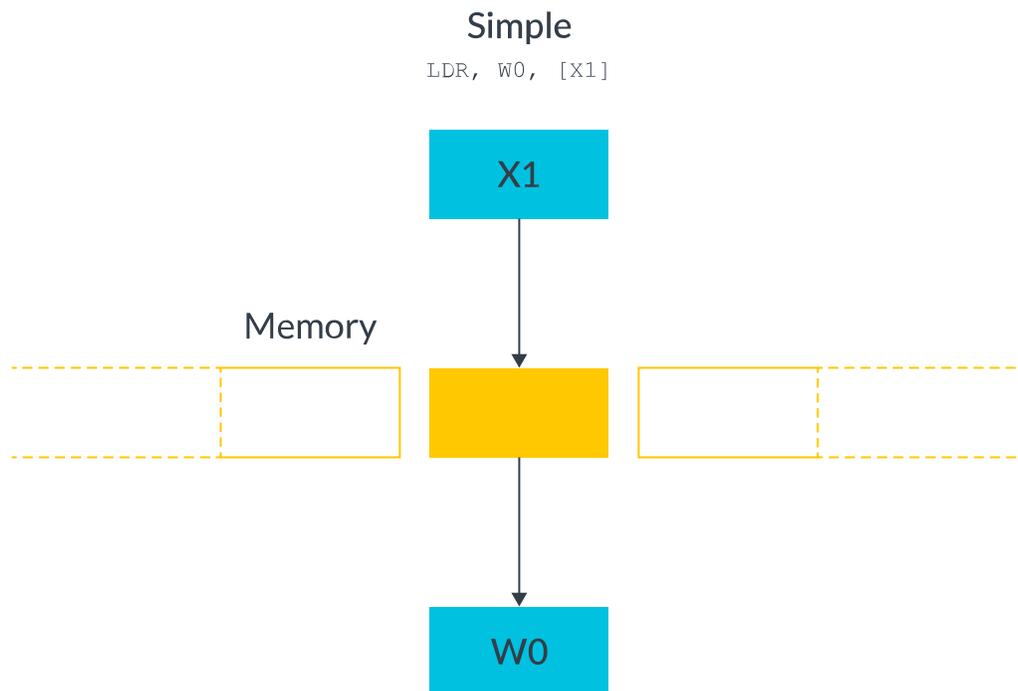
The addresses for load and store instructions appear within the square brackets, as shown in this example:

```
LDR W0, [X1]
```

There are several addressing modes that define how the address is formed.

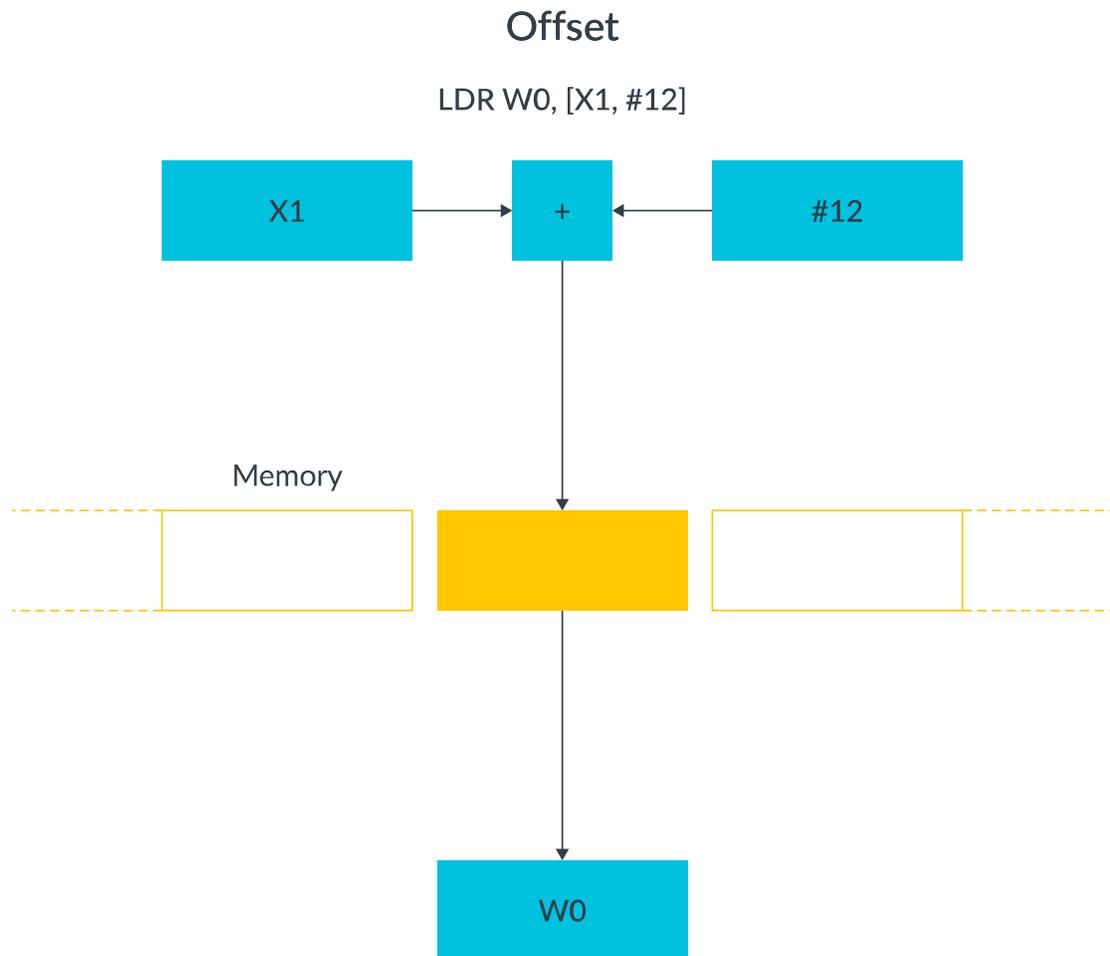
- Base register - The simplest form of addressing is a single register. Base register is an x register that contains the full, or absolute, virtual address of the data being accessed, as you can see in this figure:

Figure 18-1: Base register example



- Offset addressing modes - An offset can be applied optionally to the base address, as you can see in this figure:

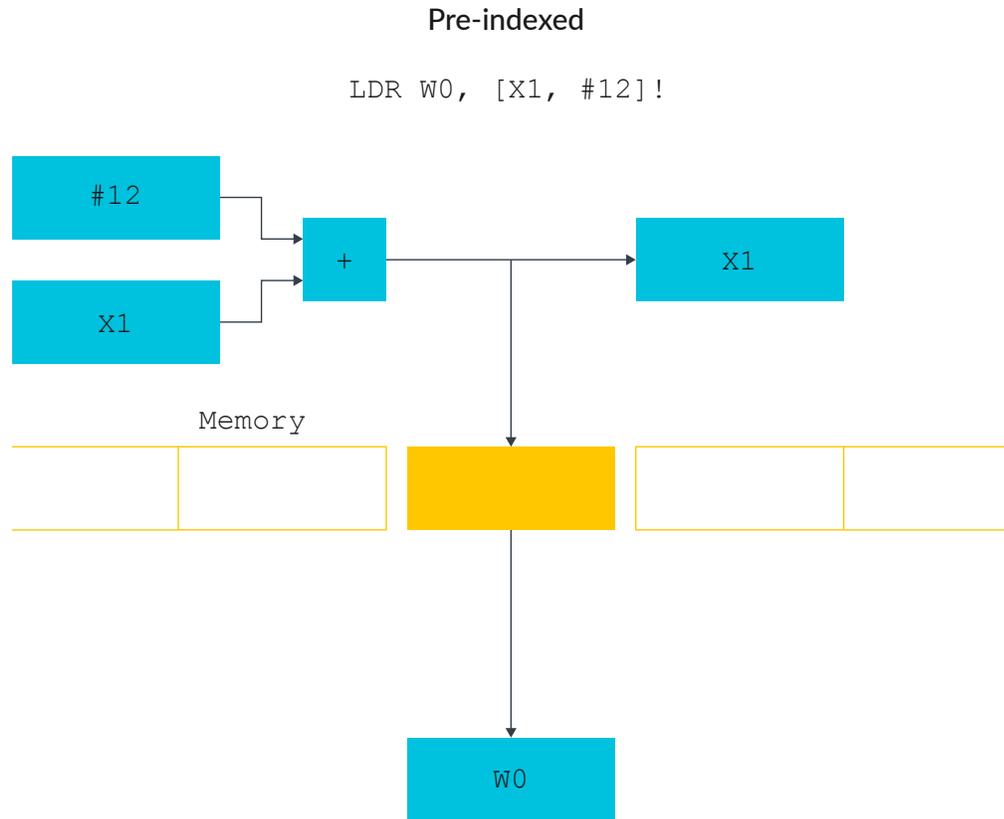
Figure 18-2: Offset example



In the preceding figure, `x1` contains the base address and `#12` is a byte offset from that address. This means that the accessed address is `x1+12`. The offset can be either a constant or another register. This type of addressing might be used for structs, for example. The compiler maintains a pointer to the base of struct using the offset to select different members.

- Pre-index addressing modes - In the instruction syntax, pre-indexing is shown by adding an exclamation mark `!` after the square brackets, as this figure shows:

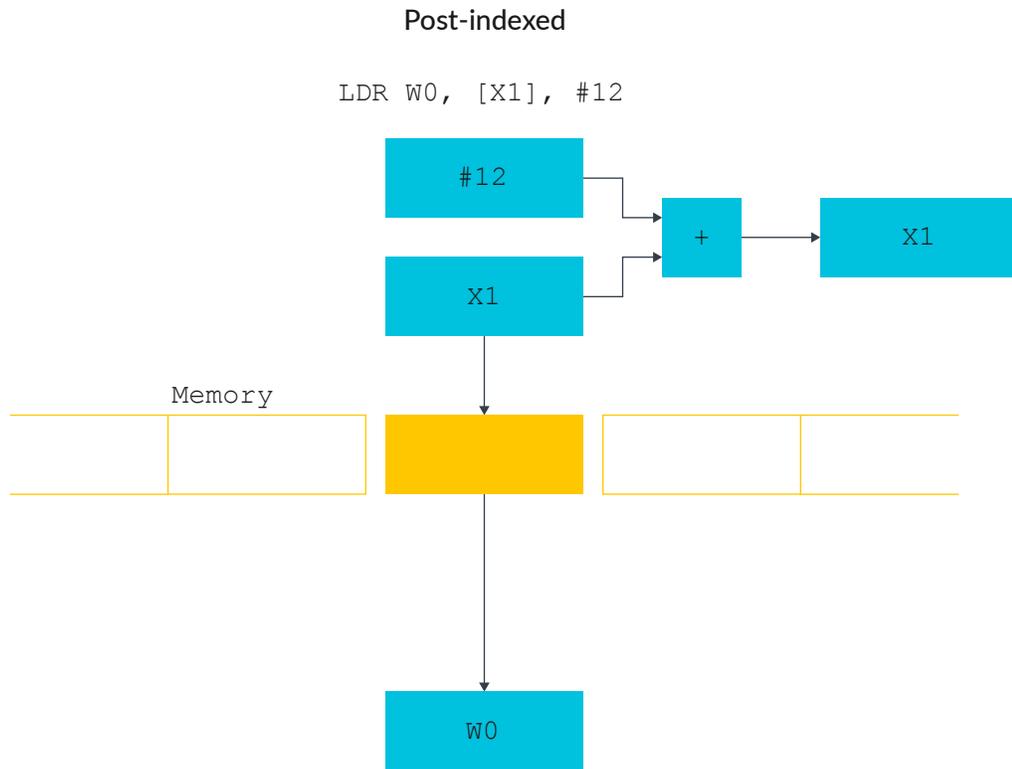
Figure 18-3: Pre-indexing example



Pre-indexed addressing is like offset addressing, except that the base pointer is updated as a result of the instruction. In the preceding figure, `x1` would have the value `x1+12` after the instruction has completed.

- Post-index addressing modes - With post-index addressing, the value is loaded from the address in the base pointer, and then the pointer is updated, as this figure shows:

Figure 18-4: Post-indexing example



Post-index addressing is useful for popping off the stack. The instruction loads the value from the location pointed at by the stack pointer, and then moves the stack pointer on to the next full location in the stack.

19. Loads and stores - load pair and store pair

So far, we have discussed the load and store of a single register. A64 also has load pair (LDP) and store pair (STP) instructions.

These instructions transfer two registers to and from memory. Registers are processed in operand order, from left-to-right. That is, the first register operand is loaded from, or stored to, the first address. The second register operand is loaded from, or stored to, the next consecutive address.

Consider the following examples.

The first instruction loads [x0] into w3, and loads [x0 + 4] into w7:

```
LDP    w3, w7, [x0]
```

This second instruction stores D0 to [x4] and stores D1 to [x4 + 8]:

```
STP    D0, D1, [x4]
```

Load and store pair instructions are often used for pushing, and popping off the stack. This first instruction pushes x0 and x1 onto the stack:

```
STP    x0, x1, [SP, #-16]!
```

This second instruction pops x0 and x1 from the stack:

```
LDP    x0, x1, [SP], #16
```

Remember that in AArch64 the stack-pointer must be 128-bit aligned.

Ordering, Alignment and Atomicity

Load and store pair instructions are treated as performing two separate accesses. The architecture does not guarantee the order that accesses are performed in. If the order of the accesses matters, then load or store pair instructions are not suitable.

The alignment requirement is based on the access size, not the total size.

For example:

```
LDP w3, w7, [x0]
```

This instruction is treated as performing two 32-bit accesses. Therefore, to be aligned the address must be 4-byte aligned, even though the total number of bytes being loaded is 8.

If the target address is aligned, each access performed by the load or store pair instruction will be single-copy atomic.

If `FEAT_LSE2` is implemented, then in some cases the entire operation can be single-copy atomic.

Using load or store pair instructions with MMIO

Load pair and store pair instructions are not typically suitable for interacting with MMIO registers. System software programmers should not use load pair or store pair instructions, or have the compiler generate them, when accessing MMIO registers.

MMIO device registers often only support a limited set of accesses sizes. A common requirement is that accesses to an MMIO register is register-sized, aligned and single-copy atomic. For example, most Generic Interrupt Controller version 3 (GICv3) MMIO registers are 32-bit in size and only support 32-bit aligned accesses.

When interacting with memory-mapped device registers, the order of accesses is often also important.

Using load pair or store pair instructions with MMIO device registers can lead to unexpected and undesirable outcomes. For example, accessing two adjacent registers using a `LDP` instruction could lead to the accesses not being in the expected order. Similarly, if accessing a single 64-bit register with a `LDP` with a `W` registers, this could appear as two accesses to the device which might not be supported.

Another consideration of using the pair instructions with MMIO is fault behaviour. The syndrome information provided when a load pair or store pair instruction faults might not contain enough information to reliably reconstruct which of the two parts of the instruction caused the fault.

20. Loads and stores - using floating point registers

Loads and stores can also be carried out using the floating-point registers, as we will see here. The first instruction loads 64-bits from `[x0]` into `D1`:

```
LDR      D1, [X0]
```

This second instruction stores 128-bits from `Q0` to `[x0 + x1]`:

```
STR      Q0, [X0, X1]
```

Finally, this instruction loads a pair of 128-bit values from `x5`, then increments `x5` by 256:

```
LDP      Q1, Q3, [X5], #256
```

There are some restrictions:

- The size is specified by the register type only.
- There is no option to sign extend loads.
- The address must still be an x register.

Load and stores using floating-point registers can be found in unexpected cases. It is common for `memcpy()` type routines to use them. This is because the wider register means that fewer iterations are needed. Just because your code does not use floating-point values, don't assume that you won't need to use the floating-point registers.

21. Loads and stores - specialist instructions

The A64 instruction set also includes some load and store instructions for more specialist use cases.

21.1 Load-Acquire and Release

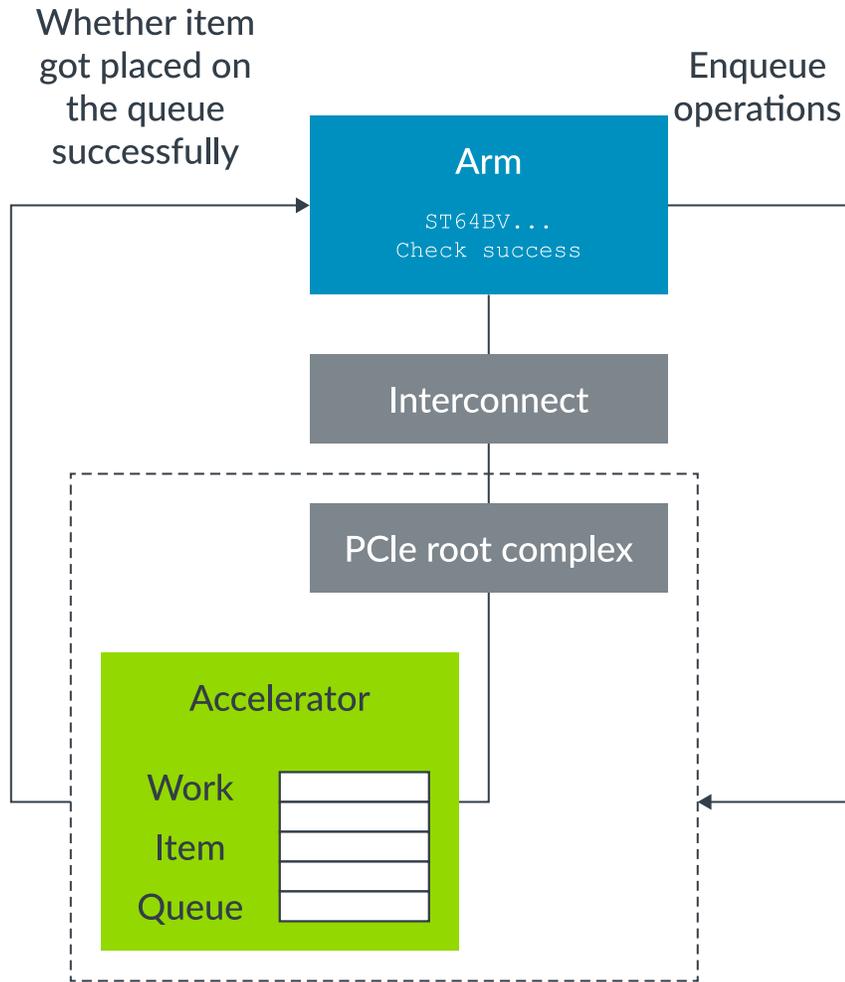
These are load and stores instructions with implicit memory barriers.

21.2 The 64-byte atomic load and stores

A growing trend in enterprise systems is the introduction of accelerators that can be accessed using a 64-byte atomic loads or stores. These are used to add items to queues and can, in some cases, signal success or failure of the enqueue operation.

To support this new breed of accelerators, Armv8.7-A and Armv9.2-A add support for a 64-byte atomic load (`LD64B`) instruction and three store (`ST64Bx`) instructions are added to the architecture.

Figure 21-1: Architecture diagram



21.3 Loads and stores to optimize memcpy() style operations

The `memcpy()` / `memset()` family of functions are widely used across many workloads. It is therefore important that they run as efficiently as possible. To enable a standard optimized implementation of these functions, which will be efficient across different processor implementations, A64 includes the `CPYX` and `SETX` instructions.

22. Program flow

Ordinarily, a processor executes instructions in program order. This means that a processor executes instructions in the same order that they are set in memory. One way to change this order is to use branch instructions. Branch instructions change the program flow and are used for loops, decisions and function calls.

The A64 instruction set also has some conditional branch instructions. These are instructions that change the way they execute, based on the results of previous instructions.



Armv8.3-A and Armv8.5-A introduced instructions to protect against return-oriented programming and jump-oriented programming.

23. Program flow - loops and decisions

In this section, we will examine how loops and decisions let you change the flow of your program code using branch instructions. There are two types of branch instructions: unconditional and conditional.

23.1 Unconditional branch instructions

There are two types of unconditional branch instructions; **B** which means Branch and **BR** which means Branch with Register.

The unconditional branch instruction **B** <label> performs a direct, PC-relative, branch to <label>. The offset from the current PC to the destination is encoded within the instruction. The range is limited by the space available within the instruction to record the offset and is +/-128MB.

When you use **BR** <x_n>, **BR** performs an indirect, or absolute, branch to the address specified in x_n.

23.2 Conditional branch instructions

The conditional branch instruction **B.<cond>** <label> is the conditional version of the B instruction. The branch is only taken if the condition <cond> is true. The range is limited to +/-1MB.

The condition is tested against the **ALU** flags stored in **PSTATE** and needs to be generated by a previous instruction such as a compare (**CMF**).

```
CBZ <Xn> <label> and CBNZ <Xn> <label>
```

This instruction branches to <label> if x_n contains 0 (**CBZ**), and branches to label if x_n does not contain 0 (**CBNZ**).

```
TBZ <Xn>, #<imm>, <label> and TBNZ <Xn>, #<imm>, <label>
```

The **TBZ** and **TBNZ** instructions work in a similar way to the **CBZ** and **CBNZ** instructions, but test the single bit specified by <imm> rather than the entire register value.



Note

The direct, or PC-relative, branches store the offset to the destination within the instruction. The conditional branches have a smaller range. This is because some bits are needed to store the condition itself, which leaves fewer bits for the offset.

Mapping these on to what you might write in C, the following examples show how branches are used for loops and decisions.

Consider the following C code:

```
if (a == 5)
    b = 5;
```

Typical output from a compiler for the above C code might be as follows:

```
    CMP W0, #5
    B.NE skip
    MOV W8, #5
skip:
```

Consider the following C code:

```
while (a != 0)
{
    b = b + c;
    a = a - 1;
}
```

Typical output from a compiler for the above C code might be as follows:

```
loop:
    CBZ W8, skip
    ADD W9, W9, W10
    SUB W8, W8, #1
    B loop
skip:
```



Note

The labels shown in the output would not be created by a compiler. They are included here to aid readability.

23.3 Test yourself

There is no conditional indirect branch instruction in A64. How could you construct an instruction sequence that performs an absolute branch to the address in x1, if x5 contains 0?

There is no single correct answer, but something like this would be acceptable:

```
    CMP    X5, XZR    // Compare X5 with zero
    B.NE skip        // If X5!=0 branch past BR
    BR    X1
skip:
...
```

24. Program flow - generating condition code

In [Program flow - loops and decisions](#), we learned that the `<cond>` is tested against the `ALU` flags stored in `PSTATE`.

The `ALU` flags are set as a side effect of data-processing instructions. To recap, an `s` at the end of an operation causes the `ALU` flags to be updated. This is an example of an instruction in which the `ALU` flags are not updated:

```
ADD      X0, X1, X2
```

This is an example of an instruction in which the `ALU` flags are updated with the use of `s`:

```
ADDS    X0, X1, X2
```

This method allows software to control when the flags are updated or not updated. The flags can be used by subsequent condition instructions. Let's take the following code as an example:

```
SUBS    X0, X5, #1
AND     X1, X7, X9
B.EQ   label
```

The `SUBS` instruction performs a subtract and updates the `ALU` flags. Then the `AND` instruction performs an and operation, and does not update the `ALU` flags. Finally, the `B.EQ` instruction performs a conditional branch, using flags set as result of the subtract.

The flags are:

- `N` - Negative
- `C` - Carry
- `V` - Overflow
- `Z` - Zero

Let's take the `Z` flag as an example. If the result of the operation was zero, then the `Z` flag is set to 1. For example, here the `Z` flag will be set if `x5` is 1, otherwise it will be cleared:

```
SUBS    X0, X5, #1
```

The condition codes map on to these flags and come in pairs. Let's take `EQ` (equal) and `NE` (not equal) as an example, and see how they map to the `Z` flag:

The `EQ` code checks for `Z==1`. The `NE` code checks for `Z==0`.

Taking the following code as an example:

```
SUBS    w0, w7, w9           // w0 = w7 - w9
B.EQ    label
```

In the first line, we have a subtract operation. Because we used the `s` suffix, this subtract operation sets the `z` flag if the result is zero. In the final line, there is a branch to `label` if `z==1`.

If `w7==w9`, the result of the subtraction will be zero, and the `z` flag would have been set. Therefore, the branch to `label` will be taken if `w7` and `w9` are equal.

In addition to the regular data-processing instructions, other instructions are available that only update the `ALU` flags:

- `CMP` - Compare
- `TST` - Test

These instructions are aliases of other instructions. For example:

```
CMP X0, X7           //an alias of SUBS XZR, X0, X7
TST W5, #1           //an alias of ANDS WZR, W5, #1
```

By using the Zero register as a destination, we are discarding the result of the operation and only updating the `ALU` flags.

24.1 Test yourself

The examples we seen so far have used the `EQ` and `NE` conditions. Write a sequence that will set `x0` to 5 if the result of `(x5 - x6)` is negative. To do this, you will need to look up the full list of condition codes in the Arm Architecture Reference Manual.

There are several sequences you could write. Here is one example:

```
SUBS    XZR, X5, X6
B.PL    1a           //Branch is positive or zero
MOV     X0, #5       //Only executed if result was negative
1:
...
```

A conditional select instruction might be a better choice here.

25. Program flow - conditional select instructions

So far, we have seen examples that use branches to handle decisions. The A64 instruction set also provides conditional select instructions. In many cases, these instructions can be used as an alternative to branches.

There are many variants, but the basic form is:

```
CSEL      Xd, Xn, Xm, cond
```

This means that:

```
if cond then
    Xd = Xn
else
    Xd = Xm
```

You can see an example in this code:

```
CMP      W1, #0
CSEL     W5, W6, W7, EQ
```

Which gives the same result as:

```
if (W1==0) then
    W5 = W6
else
    W5 = W7
```

There are variants that combine another operation with the conditional select. For example, `CSINC` performs a select and addition:

```
CSINC    Xd, Xn, Xm, cond
```

This means that:

```
if cond then
    Xd = Xn
else
```

```
Xd = Xm + 1
```

To just conditionally increment, you could write:

```
CSINC      X0, X0, X0, cond
```

Which equates to:

```
if cond then
    X0 = X0
else
    X0 = X0 + 1
```

The architecture provides an alias, `CINC`, for this command. Note however that `CINC` inverts the logic of `CSINC`:

- `CSINC X0, X0, X0, cond` leaves `X0` unchanged if `cond` is true and increments `X0` if `cond` is false
- `CINC X0, X0, cond` increments `X0` if `cond` is true and leaves `X0` unchanged if `cond` is false

Compilers choose the most efficient method to implement the functionality in your program. Compilers will often use a conditional select for small `if ... else` statements performing simple operations, because conditional selects can be more efficient than branches.

Here are some simple `if ... else` examples that compare implementations using branches to equivalent implementations using conditional select instructions:

C	Branching	Conditional select
<code>if (a != 0) b = b + 1;</code>	<code>CMP W0, #0 B.EQ else ADD W1, W1, #1 else: ...</code>	<code>CMP W0, #0 CINC W1, W1, NE</code>
<code>if (a == 0) y = y + 1; else y = y - 1;</code>	<code>CMP W0, #0 B.NE else ADD W1, W1, #1 B end else: SUB W1, W1, #1 end: ...</code>	<code>CMP W0, #0 SUB W2, W1, #1 CSINC W1, W2, W1, NE</code>

In these types of examples, conditional selects have some advantages. The sequences are shorter and take the same number of instructions, regardless of the outcome.

Importantly, conditional selects also remove the need to branch. In modern processors, this kind of branch can be difficult for the branch prediction logic to predict correctly. A mispredicted branch can have a negative effect on performance, it is better that you remove branches where possible.

26. Function calls

When calling a function or sub-routine, we need a way to get back to the caller when finished. Adding an `L` to the `B` or `BR` instructions turns them into a branch with link. This means that a return address is written into `LR` (`x30`) as part of the branch.

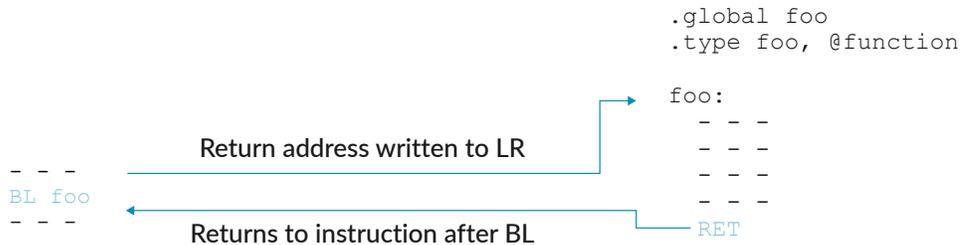


Note

The names `LR` and `x30` are interchangeable. An assembler, such as GNU GAS or `armclang`, will accept both.

There is a specialist function return instruction, `RET`. This performs an indirect branch to the address in the link register. Together, this means that we get:

Figure 26-1: RET



Note

The figure shows the function `foo()` written in GAS syntax assembler. The keyword `.global` exports the symbol and `.type` indicates that the exported symbol is a function.

Why do we need a special function return instruction? Functionally, `BR LR` would do the same job as `RET`. Using `RET` tells the processor that this is a function return. Most modern processors, and all Cortex-A processors, support branch prediction. Knowing that this is a function return allows processors to more accurately predict the branch.

Branch predictors guess the direction the program flow will take across branches. The guess is used to decide what to load into a pipeline with instructions waiting to be processed. If the branch predictor guesses correctly, the pipeline has the correct instructions and the processor does not have to wait for instructions to be loaded from memory.

27. Procedure Call Standard

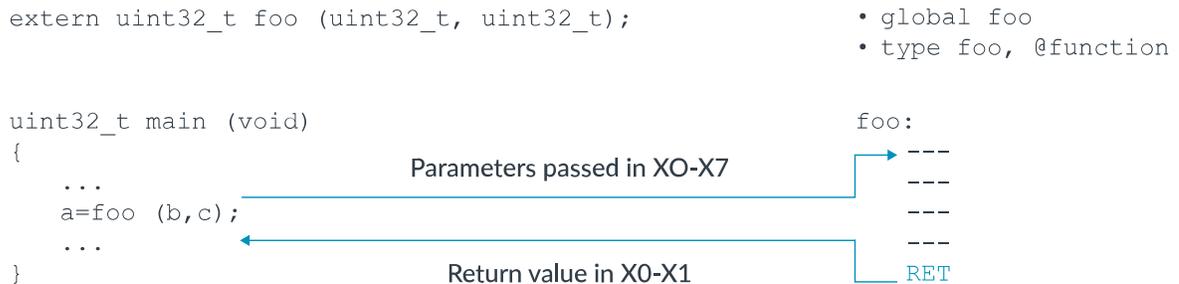
The Arm architecture places few restrictions on how general purpose registers are used. To recap, integer registers and floating-point registers are general purpose registers. However, if you want your code to interact with code that is written by someone else, or with code that is produced by a compiler, then you need to agree rules for register usage. For the Arm architecture, these rules are called the Procedure Call Standard, or PCS.

The PCS specifies:

- Which registers are used to pass arguments into the function.
- Which registers are used to return a value to the function doing the calling, known as the caller.
- Which registers the function being called, which is known as the callee, can corrupt.
- Which registers the callee cannot corrupt.

Consider a function `foo()`, being called from `main()`:

Figure 27-1: PCS example



The PCS says that the first argument is passed in `x0`, the second argument in `x1`, and so on up to `x7`. Any further arguments are passed on the stack. Our function, `foo()`, takes two arguments: `b` and `c`. Therefore, `b` will be in `w0` and `c` will be in `w1`.

Why `w` and not `x`? Because the arguments are a 32-bit type, and therefore we only need a `w` register.



In C++, `x0` is used to pass the implicit `this` pointer that points to the called function.

Next, the PCS defines which registers can be corrupted, and which registers cannot be corrupted. If a register can be corrupted, then the called function can overwrite without needing to restore, as this table of PCS register rules shows:

X0-X7	X8-X15	X16-X23	X24-X30
Parameter and Result Registers (x0-x7)	xR (x8)	IP0 (x16)	Callee-saved Registers (x24-x28)
-	Corruptible Registers (x9-x15)	IP1 (x17)	FP (x29)
-	-	PR (x18)	LR (x30)
-	-	Callee-saved Registers (x19-x23)	-

For example, the function `foo()` can use registers x0 to x15 without needing to preserve their values. However, if `foo()` wants to use x19 to x28 it must save them to stack first, and then restore from the stack before returning.

Some registers have special significance in the PCS:

- **xR** - This is an indirect result register. If `foo()` returned a struct, then the memory for struct would be allocated by the caller, `main()` in the earlier example. `xR` is a pointer to the memory allocated by the caller for returning the struct.
- **IP0** and **IP1** - These registers are intra-procedure-call corruptible registers. These registers can be corrupted between the time that the function is called and the time that it arrives at the first instruction in the function. These registers are used by linkers to insert veneers between the caller and callee. Veneers are small pieces of code. The most common example is for branch range extension. The branch instruction in A64 has a limited range. If the target is beyond that range, then the linker needs to generate a veneer to extend the range of the branch.
- **FP** - Frame pointer.
- **LR** - x30 is the link register (`LR`) for function calls.



Note

We previously introduced the `ALU` flags, which are used for conditional branches and conditional selects. The PCS says that the `ALU` flags do not need to be preserved across a function call.

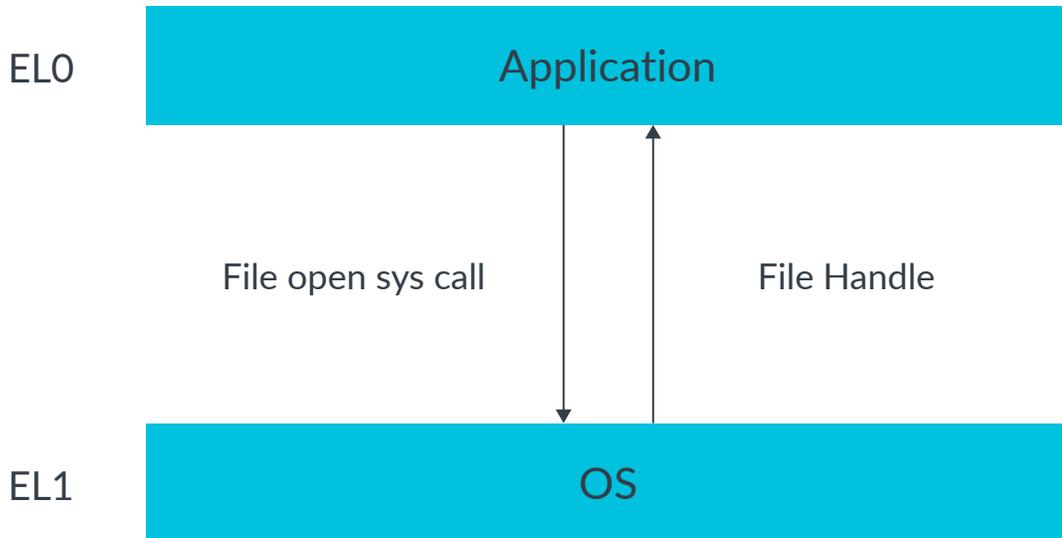
There is a similar set of rules for the floating-point registers:

D0-D7	D8-D15	D16-D23	D24-D31
Parameter and Result Registers (D0-D7)	Callee-saved Registers (D8-D15)	Callee-saved Registers (D16-D23)	Callee-saved Registers (D24-D31)

28. System calls

Sometimes it is necessary for software to request a function from a more privileged entity. This might happen when, for example, an application requests that the OS opens a file.

Figure 28-1: Application request example



In A64, there are special instructions for making such system calls. These instructions cause an exception, which allows controlled entry into a more privileged Exception level.

- `svc` Supervisor call causes an exception targeting EL1. Used by an application to call the OS.
- `hvc` Hypervisor call causes an exception targeting EL2. Used by an OS to call the hypervisor, not available at EL0.
- `smc` Secure monitor call causes an exception targeting EL3. Used by an OS or hypervisor to call the EL3 firmware, not available at EL0.

If an exception is executed from an Exception level higher than the target exception level, then the exception is taken to the current Exception level. This means that an `svc` at EL2 would cause exception entry to EL2. Similarly, an `hvc` at EL3 causes exception entry to EL3. This is consistent with the rule that an exception can never cause the processor to lose privilege.

29. Check your knowledge

Answer the following questions to check your knowledge.

According to the PCS, for the function `uint32_t foo(uint64_t a, uint64_t b)`, which registers will be used to pass in `a` and `b`, and to return the result?

`a` will be passed in `x0` and `b` will be passed in `x1`. The result will be returned in `w0`.

What instruction does a 64-bit floating-point addition of `D0` and `D1`, then places the result in `D5`?

`FADD D5, D0, D1`

What does this instruction do? `LDRSH W0, [X1, #8]`

It loads 16-bits from the address `x1+8` and sign extends the result to 32 bits, placing the result in `w0`.

What instruction is most commonly used for a function return?

`RET`

Which register is used to store the return address when calling a function?

`x30`, which can also be referred as `LR`.

What does executing an `svc` instruction in an application do?

It would cause an exception, leading to entry into the OS.

After executing `SUB W5, W8, W4`, what is in the upper 32 bits of `x5`?

0s (zeroes). Whenever a `w` register is written, the upper 32 bits of the mapped `x` register are zeroed.

30. Related information

Here are some resources related to material in this guide:

- [Vectors: Neon guide](#)
- [Vectors: SVE guide](#)
- [SVE examples: SVE Programming Examples](#)
- [Building an embedded image guide](#)
- [Arm architecture and reference manuals](#) for information on the extensions to Armv8.3-A and Armv8.5-A instruction sets, vector data-processing instructions, Advance SIMD and SVE
- [Arm community](#)
- [ARM Assembly Language](#)
- [Building an ELF Image for an Armv8-A Fixed Virtual Platform](#)
- [Changing Exception Level and Security State with an Armv8-A Fixed Virtual Platform](#)
- [Cortex-A Programmer's Guide](#)
- [Retargeting and Enabling Exceptions with an ELF Image](#)

Here are some resources related to topics in this guide:

Instruction set resources

- [Arm A64 instruction descriptions](#)

Procedure Call Standard

- [Procedure Call Standard \(PCS\) for the ARM 64-bit Architecture \(AArch64\)](#)

Useful links to training:

- [Introduction to Armv8-A](#)
- [Overview - ISA](#)
- [Architecture profiles](#)
- [What does architecture consist of?](#)

31. Next steps

Using the Arm Instruction Set Architecture (ISA), you can write software or firmware that any Arm-based processor will execute in the same way, if that software or firmware conforms to the Arm specifications. In this guide, we introduced the A64 instruction set, which is used in Armv8-A AArch64. We introduced the format of the instructions, the different types of instruction, and how code written in assembler can interact with compiler-generated code. We explained the main classes of instructions, the syntax of data-processing instructions, and how the use of `w` and `x` registers affects instructions.

Based on the material learned in this guide, you can explain how generated assembler code maps to C statements when given a C program and compiler output, and how to write a function in assembler that can be called from C. You will also understand how to find detailed descriptions for each instruction on the Arm Developer website, and concepts such as registers, data processing, program flow, and loads and stores.

To keep learning about the Armv8-A architecture, see more in our [series of guides](#).

To check your knowledge of A64 assembler, try the ISA lab exercises (coming soon). The lab exercises require the Arm DS-5, Ultimate Edition. A 30-day evaluation version is available and can be used to complete the exercises.