The A64 instruction set

Version 1.0

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
03 March 2017	0100	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

http://www.arm.com

Contents

1 The A64 instruction set

One of the most significant changes introduced in the ARMv8-A architecture was the addition of an instruction set for AArch64, called A64. This instruction set contains features similar to the existing AArch32 (ARMv7-A) 32-bit instruction set.

The addition of A64 provides access to 64-bit wide integer registers and data operations, and the ability to use 64-bit sized pointers to memory. The A64 instructions execute in the AArch64 Execution state. ARMv8-A also includes the original ARM® instruction set, now called A32, and the Thumb® (T32) instruction set.

Programmers writing at the application level might never need to write code in assembly language. However, assembly code can be useful in cases when highly optimized code is required. This is the case when writing compilers, or where using low level features not directly available in C is required, for example:

- Portions of boot code.
- Device drivers.
- Operating system development.

Reading assembly code can be helpful for debugging C, particularly to understand the mapping between assembly instructions and C statements.

1.1 Instruction mnemonics

The A64 instruction set overloads instruction mnemonics. That is, it distinguishes between the different forms of an instruction, based on the operand register names that are used. For example, the following ADD instructions all have different forms, but you only have to remember one instruction and the assembler automatically chooses the correct encoding, based on the operands used.

```
ADD W0, W1, W2              // add 32-bit registers
ADD X0, X1, X2              // add 64-bit registers
ADD X0, X1, W2, SXTW        // add sign extended 32-bit register to 64-bit
                            // extended register
ADD X0, X1, #42             // add immediate to 64-bit register
ADD V0.8H, V1.8H, V2.8H     // NEON 16-bit add, in each of 8 lanes
```

Distinguishing between 32-bit and 64-bit A64 instructions

Most integer instructions in the A64 instruction set have two forms, which operate on either 32-bit or 64-bit values within the 64-bit general-purpose register file.

When looking at the register name that the instruction uses:

- If the register name starts with X, it is a 64-bit register.
- If the register name starts with W, it is a 32-bit register.

When a 32-bit register form is selected:

- Right shifts and rotates inject at bit 31, instead of bit 63.
- The condition flags, where set by the instruction, are computed from the lower 32 bits.
- Writes to the W register set bits [63:32] of the X register to zero.

This distinction applies even when the results of a 32-bit register form would be indistinguishable from the lower 32 bits computed by the equivalent 64-bit register form. For example, A64 includes separate 32-bit and 64-bit register forms of the `ORR` instructions. A 32-bit bitwise `ORR` could just as easily be performed using a 64-bit `ORR` and ignoring the top 32 bits of the result.

2 Data processing instructions

These are the fundamental arithmetic and logical operations of the processor and operate on values in general-purpose registers, or on a register and an immediate value. A64 deals naturally with 64-bit signed and unsigned data types by offering more concise and efficient ways of manipulating 64-bit integers. This can be advantageous for all languages that provide 64-bit integers such as C or Java. Multiply instructions can be considered special cases of these instructions.

Data processing instructions mostly use one destination register and two source operands. The general format can be considered as the instruction, followed by the operands, as follows:

```
Instruction Rd, Rn, Operand2
```

Where

`Rd` is the destination register.

`Rn` is the register that is operated on.

The use of R here indicates that the registers can be either X or W registers.

`Operand2` might be a register, a modified register, or an immediate value.

Data processing operations include:

- Arithmetic and logical operations.
- Move and shift operations.
- Instructions for sign and zero extension.
- Bit and bitfield manipulation.
- Conditional comparison and data processing.

2.1 Arithmetic and logical operations

Some of the available operations are shown in the following table.

Type	Instructions
Arithmetic	ADD, SUB, ADC, SBC, NEG
Logical	AND, BIC, ORR, ORN, EOR, EON
Comparison	CMP, CMN, TST
Move	MOV, MVN

Some of these instructions also have an `S` suffix, indicating that the instruction sets flags.

Of the above instructions, those taking the suffix include `ADDS`, `SUBS`, `ADCS`, `SBCS`, `ANDS`, and `BICS`. There are other flag setting instructions, notably `CMP`, `CMN`, and `TST`, but these do not take an `S` suffix.

The operations ADC and SBC perform additions and subtractions that also use the carry condition flag as an input.

```
ADC{S}: Rd = Rn + Rm + C

SBC{S}: Rd = Rn - Rm - 1 + C
```

For example:

```
Loop
ADD W0, W1, W2, LSL #3        // W0 = W1 + (W2 << 3)
SUBS X0, X4, X3, ASR #2       // X0 = X4 - (X3 >> 2), set flags
BGE Loop                      // Loop if not negative
MOV X0, X1                    // Copy X1 to X0
CMP W3, W4                    // Set flags based on W3 - W4
ADD W0, W5, #27               // W0 = W5 + 27
```

The logical operations are the same as the corresponding Boolean operators operating on individual bits of the register.

For example:

```
AND X2, X2, X1

ORR W2, W2, W5

EOR W7, W7, W6

BIC X0, X0, X1
```

The BIC (Bitwise bit Clear) instruction (`BIC Rd, Rn, Operand2`):

- Inverts `Operand2`.
- Performs a bitwise AND with `Rn`.
- Stores the result in `Rd`.

For example, to clear bit [11] of register X0, use:

```
MOV X1, #0x800
BIC X0, X0, X1
```

`ORN Rd, Rn, Operand2` inverts `Operand2` and ORs with `Rd`.

`EON Rd, Rn, Operand2` inverts `Operand2` and EORs with `Rd`.

The comparison instructions only modify the flags and have no other effect. The range of immediate values for these instructions is 12 bits, and the immediate value can be shifted 12 bits to the left.

2.2 Multiply instructions

The multiply instructions that are provided are broadly similar to those in ARMv7-A, but have the ability to perform 64-bit multiplies in a single instruction.

Instruction	Description
MADD	Multiply add
MNEG	Multiply negate
MSUB	Multiply subtract
MUL	Multiply
SMADDL	Signed multiply-add long
SMNEGL	Signed multiply-negate long
SMSUBL	Signed multiply subtract long
SMULH	Signed multiply returning high half
SMULL	Signed multiply long
UMADDL	Unsigned multiply-add long
UMNEGL	Unsigned multiply-negate long
UMSUBL	Unsigned multiply subtract long
UMULH	Unsigned multiply returning high half
UMULL	Unsigned multiply long

There are multiply instructions that operate on 32-bit or 64-bit values and return a result of the same size as the operands. For example, two 64-bit registers can be multiplied to produce a 64-bit result with the MUL instruction. Any overflow is ignored and the result is taken as the lower 64 bits.

```
MUL X0, X1, X2              // X0 = X1 × X2
```

There is also the ability to add or subtract an accumulator value in a third source register, using the MADD or MSUB instructions.

The MNEG instruction can be used to negate the result, for example:

```
MNEG X0, X1, X2             // X0 = -(X1 × X2)
```

Also, there are a range of multiply instructions that produce a long result, that is, multiplying two 32-bit numbers and generating a 64-bit result. There are both signed and unsigned variants of these long multiplies (UMULL, SMULL). There are also options to accumulate a value from another register (UMADDL, SMADDL) or to negate (UMNEGL, SMNEGL).

Including 32-bit and 64-bit multiply with optional accumulation gives a result size that is the same size as the operands. Again, any overflow is ignored and the result is taken as the lower 64 bits of the calculation:

- 32 ± (32 × 32) gives a 32-bit result.

- 64 ± (64 × 64) gives a 64-bit result.
- ± (32 × 32) gives a 32-bit result.
- ± (64 × 64) gives a 64-bit result.

Both signed and unsigned widening multiply with accumulation give a single 64-bit result:

- 64 ± (32 × 32) gives a 64-bit result.
- ± (32 × 32) gives a 32-bit result.

A 64 × 64 to 128-bit multiplication has the potential to generate a result that is larger than 64 bits. This causes the result register of the standard `MUL` instruction to overflow. In this case, the signed/unsigned multiply high instructions (`SMULH`, `UMULH`) captures the bits of the result lost by the overflow in another register.

For example, the operation `0xFFFF_FFFF_FFFF_FFFF` x `0x2` would give the answer `0x1_FFFF_FFFF_FFFF_FFFE` which will not fit into a single 64 bit register.

```
MOV     X0, #0xFFFFFFFFFFFFFFFF
MOV     X1, #0x2

MUL     X2, X0, X1      // X2 = 0xFFFF_FFFF_FFFF_FFFE
UMULH   X3, X0, X1      // X3 = 0x0000_0000_0000_0001
```

Note

You cannot directly multiply a 32-bit W register by a 64-bit X register.

2.3 Divide instructions

ARMv8-A supports signed and unsigned division of 32-bit and 64-bit sized values.

Instruction	Description
`SDIV`	Signed divide
`UDIV`	Unsigned divide

For example:

```
UDIV W0, W1, W2         // W0 = W1 / W2 (unsigned, 32-bit divide)

SDIV X0, X1, X2         // X0 = X1 / X2 (signed, 64-bit divide)
```

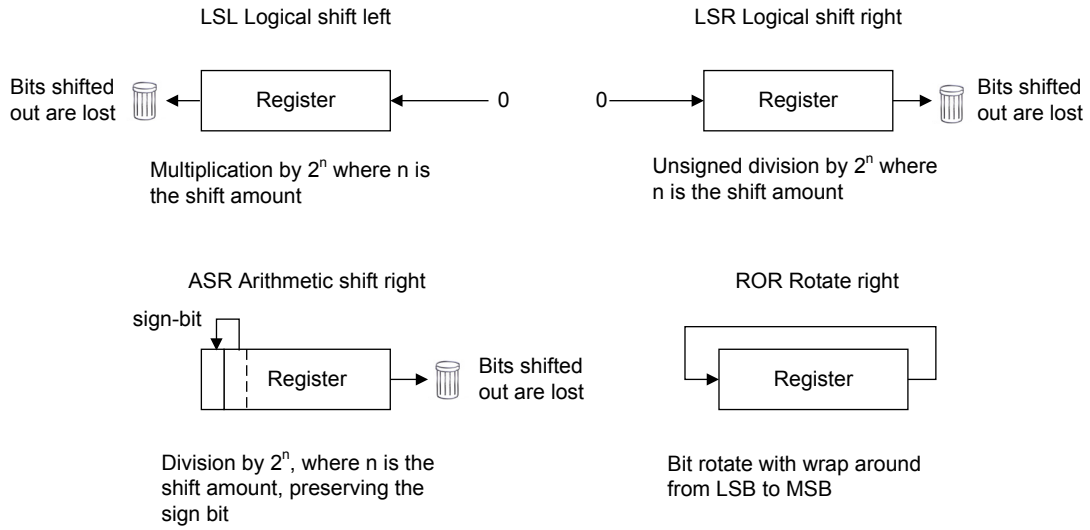
Overflow and divide-by-zero are not trapped:

- Any integer division by zero returns zero.
- Overflow can only occur in `SDIV`:
    - `INT_MIN / -1` returns `INT_MIN`, where `INT_MIN` is the smallest negative number that can be encoded in the registers that are used for the operation. The result is always rounded towards zero, as in most C/C++ dialects.

2.4 Shift operations

The following instructions move the bit patterns within a register to the left or to the right:

- Logical Shift Left (LSL). The LSL instruction performs multiplication by a power of 2.
- Logical Shift Right (LSR). The LSR instruction performs division by a power of 2.
- Arithmetic Shift Right (ASR). The ASR instruction performs division by a power of 2, preserving the sign bit.
- Rotate right (ROR). The ROR instruction performs a bitwise rotation, wrapping the bits rotated from the LSB into the MSB.



The register that is specified for a shift can be 32-bit or 64-bit. The amount to be shifted can be an immediate, that is, up to (`Register size - 1`), or by a register where the value is taken only from the bottom five (modulo-32) or six (modulo-64) bits.

2.5 Shift operators

The Arithmetic (shifted register) instructions apply an optional shift operator to the operand register value before performing the arithmetic operation. They can be used in some arithmetic instructions to allow constant multiplication or division and bit pattern manipulation on the Operand. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit [63] or bit[31].

The general form of these is

```
<operation> Rd, Rs, Operand<operation> #imm
```

For example

```
MOV X0,X0,LSL #8

MOV W0,W0,ASR #2
```

The following table contains arithmetic instructions that can contain shifted operands:

Instruction	Description
ADD	Add
ADDS	Add and set flags
SUB	Subtract
SUBS	Subtract and set flags
CMN	Compare negative
CMP	Compare
NEG	Negate
NEGS	Negate and set flags

The shift operators LSL, ASR, and LSR have the same names and perform the same actions as shift instructions. Shift operators take a source register and shift it left or right by the specified number of bits, with optional sign extension. Omitting the shift operator means that there is no shift. The shift operations are mostly the same as they are in A32 and T32.

The shift amount is encoded in the instruction (and is therefore constant). A significant difference from A32 is that there are no register-shifted-by-register forms. Such operations are possible in A64, but as in T32, they are standalone instructions with slightly different syntax.

For all shift modifiers, the size of the result is the same as the size of the source. There is no implicit widening or narrowing. These modifiers are the A64 equivalent of the flexible operand that is often called Operand2 in A32 and T32.

The general form of these operators is

```
<operation> #imm
```

Where

`operation` is one of the modifiers, and

`#imm` is optional and defaults to 0.

For example

```
SUB X0, X1, X2, LSR #0          //Subtract a shifted register
                                // X0 = X1 - ((uint-t)x2 >> 0)

ADD X5, X6, #10, LSL #12        //Add a shifted immediate
                                // X5 = X6 + (10 >> 12)
```

The following table shows the shift operators;

Instruction	Description
LSL	Immediate left shift amount in the range 0 to (Register size - 1).
ASR	Arithmetic right shift amount in the range 0 to (Register size - 1).
LSR	Immediate right shift amount in the range 0 to(Register size - 1).

The current stack pointer, SP or WSP, cannot be used with this class of instructions.

2.6 Byte and bitfield manipulation instructions

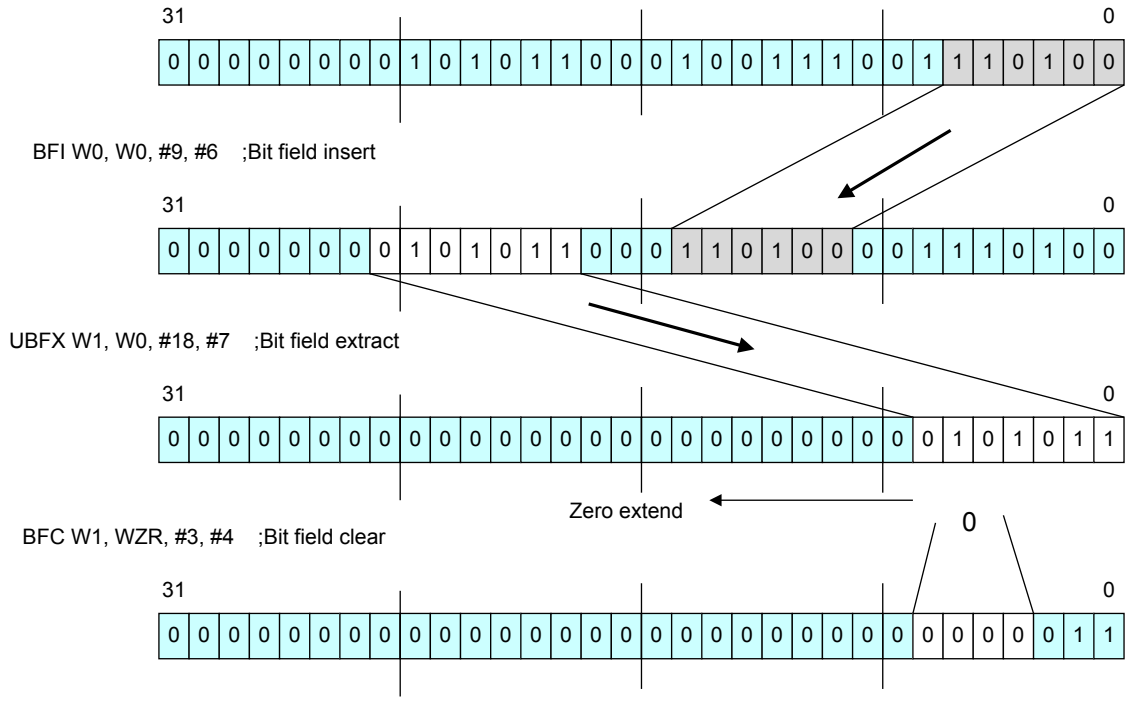
There are instructions that extend a byte, halfword, or word to register size (although only SXTW operates on a word). These instructions exist in both signed (SXTB, SXTH, SXTW) and unsigned (UXTB, UXTH) variants and are aliases to the appropriate bitfield manipulation instruction. The source is always a W register. The destination register (except for SXTW) is either an X or a W register.

For example:

```
SXTB X0, W1                 // Sign extend the least significant byte of
                            // register
                            // W1 from 8-bits to 64-bit by repeating the
                            //leftmost bit of the byte.
```

Bitfield instructions include Bit Field Insert (BFI), and signed and unsigned Bit Field Extract ((S/U)BFX). There are extra bitfield instructions too, such as BFXIL (Bit Field Extract and Insert Low), UBFIZ (Unsigned Bit Field Insert in Zero), and SBFIZ (Signed Bit Field Insert in Zero).

It is also possible to extend and shift the Operand register of an ADD, SUB, CMN, or CMP instruction and the index register of a Load or Store instruction. This results in efficient implementation of array index calculations using a 64-bit array pointer and 32-bit array index, as shown in the following figure:

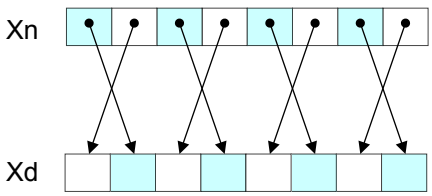


Note

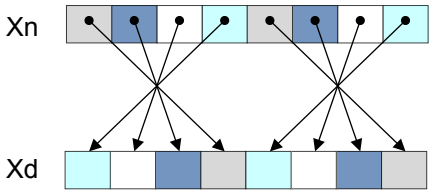
- There are also `BFM`, `UBFM`, and `SBFM` instructions.
- These are Bit Field Move instructions.
- However, these instructions do not need to be used explicitly, as aliases are provided for all cases.
- These aliases are the bitfield operations already described: (`S/U`)`XT`(`B/H/W/X`), `ASR/LSL/LSR` immediate, `BFI`, `BFXIL`, `SBFIZ`, `SBFX`, `UBFIZ`, and `UBFX`.

There are other bit manipulation instructions:

- `CLZ` Count leading zero bits in a register. Similarly, the same byte manipulation instructions:
- `RBIT` Reverse all bits.
- `REV` Reverse the byte order of a register.
- `REV16` Reverse the byte order of each halfword in a register.



- `REV32` Reverse the byte order of each word in a register.



These operations can be performed on either word (32-bit) or doubleword (64-bit) sized registers, except for REV32, which applies only to 64-bit registers.

Extend operators

The extended register instructions provide an optional sign-extension or zero-extension of a portion of the operand register value, followed by an optional left shift by a constant amount of 1-4, inclusive.

The general form of these is:

```
<operation> Rd, Rs, Operand<extend operation> #imm
```

The following table lists arithmetic (shifted register) instructions:

Instruction	Description
ADD	Add
ADDS	Add and set flags
SUB	Subtract
SUBS	Subtract and set flags
CMN	Compare negative
CMP	Compare

The extend operators take a sequence of consecutive bits from the source register, then sign or zero-extend it to make it the required size. Again, the operators have the same names and perform the same actions as the instruction described in byte and bitfield manipulation instructions, but with some differences. For example, there are no {S/U}XTW or {S/U}XTX instructions.

The result size is implied by the context. Most of these extend operations exist in A32 and T32, but they are more flexible in A64 and can often be used as an operand modifier.

Several contexts also allow extend modes to take an extra immediate left shift (like LSL). This shift has a limited range (of 0-4 bits), and it applies after the extend operation. For example, SXTB #2 means sign extend from the 8-bit source, then shift it left by 2 bits.

Because extend operands can take a shift, UXTX, and sometimes UXTW, are functionally identical to LSL for shifts 0-4. These are actually aliases in a few corner cases where extend modes are available but shift modes are not. If the shift amount is not specified, the default shift amount is zero.

Recall the format of the ADD instruction

```
ADD X0, X1, W2, SXTW                // add sign extended 32-bit register to
                                    // 64-bit extended register
```

In this case, the extension is applied to the operand, and can have the following values:

Instruction	Description
UXTB	Extracts a byte value from a register and zero extends it to the size of the register.
UXTH	Extracts a halfword value from a register and zero extends it to the size of the register.
LSL or UXTW	Extracts a word value from a register and zero extends it to the size of the register.
UXTX	Use the whole 64-bit register.
SXTB	Extracts a byte value from a register and zero extends it to the size of the register.
SXTH	Extracts a halfword value from a register and zero extends it to the size of the register.
SXTW	Extracts a word value from a register and zero extends it to the size of the register.
SXTX	Use the whole 64-bit register.

- When the shift amount is also zero, then both the operator and the shift amount can be omitted.
- For 64-bit register instructions, the additional operators UXTX and SXTX use all 64 bits of the operand register with an optional shift. In that case, ARM recommends UXTX as the operator.
- When at least one register is SP, ARM recommends use of the LSL operator, rather than UXTX.
- For 32-bit register instructions, the operators UXTW and SXTW both use all 32 bits of the operand register with an optional shift. Again, ARM recommends UXTW as the operator.
- When one register is WSP, ARM recommends use of the LSL operator, rather than UXTW.

```
In 64-bit register instructions, the final register operand is written as Wm for all
except the UXTX/LSL and SXTX extend operators.
```

For example:

```
LDR W10, [X11, W12, SXTW]           // Load from an array using a signed
                                    //index
                                    // W10 = *(uint32_t*)(X11 + (int32_t)W12)
```

2.7 Conditional instructions

There are four status flags, Zero (Z), Negative (N), Carry (C) and Overflow (V). The following table indicates the value of these bits for flag setting operations.

Note

The condition flags (NZCV) and the condition codes are the same in A32 and T32.

Flag	Name	Description
N	Negative	Set to the same value as bit[31] of the result. For a 32-bit signed integer, bit[31] being set indicates that the value is negative.
Z	Zero	Set to 1 if the result is zero, otherwise it is set to 0.
C	Carry	Set to the carry-out value from result, or to the value of the last bit shifted out from a shift operation.
V	Overflow	Set to 1 if signed overflow or underflow occurred, otherwise it is set to 0.

The C flag is set if the result of an unsigned operation overflows the result register.

The V flag operates in the same way as the C flag, but for signed operations.

Code	Encoding	Meaning (when set by CMP)	Meaning (when set by FCMP)	Condition flags
EQ	0b0000	Equal to.	Equal to.	Z =1
NE	0b0001	Not equal to.	Unordered, or not equal to.	Z = 0
CS	0b0010	Carry set (identical to HS).	Greater than, equal to, or unordered (identical to HS).	C = 1
HS	0b0010	Greater than, equal to (unsigned)(identical to CS).	Greater than, equal to, or unordered (identical to CS).	C = 1
CC	0b0011	Carry clear (identical to LO).	Less than (identical to LO).	C = 0
LO	0b0011	Unsigned less than (identical to CC).	Less than (identical to CC).	C = 0
MI	0b0100	Minus, Negative.	Less than.	N = 1
PL	0b0101	Positive or zero.	Greater than, equal to, or unordered.	N = 0
VS	0b0110	Signed overflow.	Unordered. (At least one argument was NaN).	V = 1
VC	0b0111	No signed overflow.	Not unordered. (No argument was NaN).	V = 0
HI	0b1000	Greater than (unsigned).	Greater than or unordered.	(C = 1) &&(Z = 0)
LS	0b1001	Less than or equal to (unsigned).	Less than or equal to.	(C = 0) || (Z= 1)

GE	0b1010	Greater than or equal to (signed).	Greater than or equal to.	N==V
LT	0b1011	Less than (signed).	Less than or unordered.	N!=V
GT	0b1100	Greater than (signed).	Greater than.	(Z==0) && (N==V)
LE	0b1101	Less than or equal to (signed).	Less than, equal to or unordered.	(Z==1) ||(N!=V)
AL	0b1110	Always executed.	Default. Always executed.	Any
NV	0b1111	Always executed.	Always executed.	Any

Note

A64 adds NV (0b1111), though it behaves the same as its complement, AL (0b1110). This is different in ARMv7-A A32.

There are a small set of conditional data processing instructions. These instructions are unconditionally executed but use the condition flags (N,Z, C,V) as an extra input to the instruction. For example:

```
MRS  X1, NZCV          // copy N, Z, C, and V flags into general-purpose x1
MOV  X2, #0x30000000
BIC  X1,X1,X2          // clears the C and V flags (bits 29,28)
ORR  X1,X1,#0xC0000000 // sets the N and Z flags (bits 31,30)
MSR  NZCV, X1          // copy x1 back into NZCV register to update the
                       // condition flags
```

This set has been provided to replace common usage of conditional execution in ARM code.

Instructions types that read the condition flags are:

Add/subtract with carry — The traditional ARM instructions, for example, for multi-precision arithmetic and checksums.

Conditional select with optional increment, negate, or invert — Conditionally select between one source register and a second incremented, negated, inverted, or unmodified source register.

Note

These are the most common uses of single conditional instructions in A32 and T32. Typical uses include conditional counting or calculating the absolute value of a signed quantity.

Conditional operations

A64 only enables conditional execution on branch instructions. This is in contrast to A32 and T32 where most instructions can be used with a condition code. The A64 instructions are:

Conditional select (move)

- CSEL Selects between two registers, based on a condition. Unconditional instructions, followed by a conditional select, can replace short conditional sequences.

For example:

```
CSEL W1,W2,W3,EQ              // Return W2 to W0 if W3 and W2 are equal.
```

- CSINC Selects between two registers, based on a condition. Return the first source register or the operand register that is incremented by one. This is useful for controlling program loops.

  For example:

  ```
  CSINC X0, X1, X0, NE        // Set the return register X0 to X1 if Zero
                              // flag clear, else increment X0
  ```

- CSINV Selects between two registers, based on a condition. Return the first source register or the inverted operand register.

- CSNEG Selects between two registers, based on a condition. Return the first source register or the negated operand register.

Conditional set

Conditionally selects between 0 and 1 (CSET) or 0 and -1 (CSETM). Used, for example, to set the condition flags as a Boolean value or mask in a general register.

Conditional compare

CMP and CMN set the condition flags to the result of a comparison if the original condition is true. If not true, the conditional flags are set to a specified condition flag state. The conditional compare instruction is useful for expressing nested or compound comparisons.

Conditional select and conditional compare are also available for floating-point registers using the FCSEL and FCCMP instructions.

For example:

```
CINC X0, X0, LS               // If less than or same (LS) then X0 = X0 + 1
CSET W0, EQ                   // If the previous comparison was equal
                              // (Z=1) then W0 = 1, else W0 = 0
CSETM X0, NE                  // If not equal then X0 = -1, else X0 = 0
```

This class of instructions provides a powerful way to avoid the use of branches or conditionally executed instructions. Compilers, or assembly programmers, might adopt a technique of performing the operations for both branches of an if-then-else statement. Then the correct result is selected at the end.

For example, consider the C code:

```
if (i == 0) r = r + 2; else r = r - 1;
```

This might produce assembly code similar to:

```
CMP W0, #0                    // if (i == 0)
SUB W2, W1, #1                // r = r - 1
ADD W1, W1, #2                // r = r + 2
CSEL W1, W1, W2, EQ           // select between the two results
```

3 Memory access instructions

The ARMv8-A architecture is a load/store architecture, like all previous ARM architectures. This means that data processing instructions do not operate directly on data in memory. The data must first be loaded into registers, modified, and then stored to memory. An instruction must specify an address, the size of data to be transferred, and a source or destination register. There are also Load and Store instructions that provide extra control of how memory is accessed.

Memory instructions can access Normal memory in an unaligned fashion. This is not supported by exclusive accesses, load acquire or store release variants. If unaligned accesses are not wanted, they can be configured to be faulted; however, most real operating systems (including Linux) do not do this. Unaligned accesses can be useful, and many user-space programs assume that they are possible.

3.1 Load instruction format

The general form of a Load instruction is:

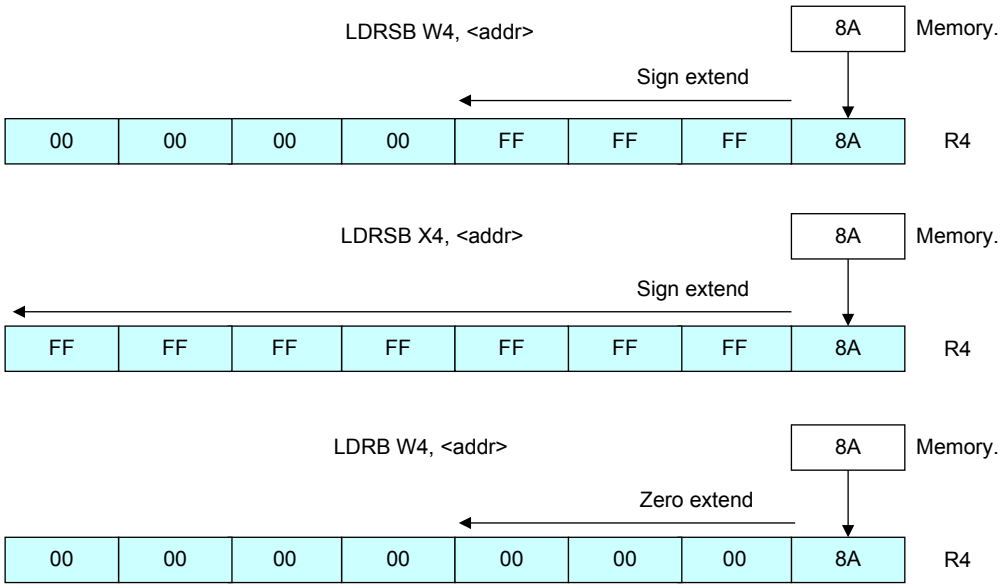
```
LDR Rt, <addr>
```

For loads into integer registers, you can choose a size to load. For example, to load a size smaller than the specified register value, append one of the following suffixes to the LDR instruction:

- `LDRB` (8-bit, zero extended).
- `LDRSB` (8-bit, sign extended).
- `LDRH` (16-bit, zero extended).
- `LDRSH` (16-bit, sign extended).
- `LDRSW` (32-bit, sign extended).

There are also unscaled offset forms such as `LDUR<type>`. Programmers do not normally have to use the `LDUR` form explicitly, because most assemblers can select the appropriate version, based on the offset used.

You do not need to specify a zero-extended load to an X register, because writing to a W register effectively zero extends to the entire register width, as shown in the following figure:



3.2 Store instruction format

The general form of a Store instruction is:

```
STR Rn, <addr>
```

There are also unscaled offset forms such as `STUR<type>`. Programmers do not normally have to use the `STUR` form explicitly, as most assemblers can select the appropriate version, based on the offset used.

The size to be stored might be smaller than the register. You can specify this by adding a `B` or `H` suffix to the `STR`. It is always the least significant part of the register that is stored in such a case.

3.3 SIMD (NEON) and floating-point scalar loads and stores

Load and Store instructions can also access SIMD and floating-point registers. In this case, the size is determined only by the register being loaded or stored. which can be any of the B, H, S, D, or Q registers. The memory bits read or written in each case is summarized in the following tables.

For Load instructions:

Load	Xn	Wn	Qn	Dn	Sn	Hn	Bn
LDR	64	32	128	64	32	16	9
LDP	128	64	256	128	64	-	-
LDRB	-	8	-	-	-	-	-
LDRH	-	16	-	-	-	-	-
LDRSB	8	8	-	-	-	-	-
LDRSH	16	16	-	-	-	-	-

LDRSW	32	-	-	-	-	-	-

For Store instructions:

Store	Xn	Wn	Qn	Dn	Sn	Hn	Bn
STR	64	32	128	64	32	16	8
STP	128	64	256	128	64	-	-
STRB	-	8	-	-	-	-	-
STRH	-	16	-	-	-	-	-

No sign-extension options are available for loads into FP/SIMD registers. Addresses for such loads are still specified using the general-purpose registers.

For example:

```
LDR D0, [X0, X1]            // Loads register D0 with the doubleword at the
                            // memory address pointed to by X0 plus X1.
```

Floating-point and scalar SIMD Loads and Stores use the same addressing modes as integer Loads and Stores.

For example:

```
LD1R { V1.8B }, [X1]
```

3.4 Specifying the address for a Load or Store instruction

The addressing modes available to A64 are similar to those in A32 and T32. There are some additional restrictions and some new features, but the addressing modes available to A64 will not be surprising to someone familiar with A32 or T32.

In A64, the base register of an address operand must always be an X register. However, several instructions support zero-extension or sign-extension so that a 32-bit offset can be provided as a W register.

Offset modes

Offset addressing modes add an immediate value or an optionally modified register value to a 64-bit base register to generate an address.

Example instruction	Description
`LDR X0, [X1]`	Load from the address in X1
`LDR X0, [X1, #8]`	Load from address X1 + 8
`LDR X0, [X1, X2]`	Load from address X1 + X2
`LDR X0, [X1, X2, LSL, #3]`	Load from address X1 + (X2 << 3)
`LDR X0, [X1, W2, SXTW]`	Load from address X1 + sign extend(W2)
`LDR X0, [X1, W2, SXTW, #3]`	Load from address X1 + (sign extend(W2) << 3)

Table 12 Offset addressing modes

When specifying a shift or extension option, the shift amount can be either 0 (the default) or $\log_2$ of the access size in bytes (so that Rn << <shift> multiplies Rn by the access size). This supports common array-indexing operations.

```
// A C example showing accesses that a compiler is
// likely to generate.

void example_dup(int32_t a[], int32_t length)
   {
 int32_t first = a[0];              // LDR W3, [X0]
 for (int32_t i = 1; i < length; i++)
          {
          a[i] = first;                    // STR W3, [X0, W2, SXTW, #2]
          }
}
```

Index modes

Index modes are similar to offset modes, but they also update the base register. The syntax is the same as in A32 and T32, but the set of operations is more restrictive. Usually, only immediate offsets can be provided for index modes.

There are two variants:

- Pre-index modes, which apply the offset before accessing the memory.
- Post-index modes, which apply the offset after accessing the memory.

Offset addressing modes are shown in the following table:

Example instruction	Description
LDR X0, [X1, #8]!	Pre-index: Update X1 first (to X1 + #8), then load from the new address
LDR X0, [X1], #8	Post-index: Load from the unmodified address in X1 first, then update X1 (to X1 + #8)
STP X0, X1, [SP, #-16]!	Push X0 and X1 to the stack.
LDP X0, X1, [SP], #16	Pop X0 and X1 off the stack.

These options map cleanly onto some common C operations:

```
// C example showing accesses that a compiler is likely to generate.

void example_strcpy(char * dst, const char * src)
{
char c;
do {
    c = *(src++);                       // LDRB W2, [X1], #1
    *(dst++) = c;                       // STRB W2, [X0], #1
   }
while (c != '\0');
}
```

3.5 Literal pools

A literal pool is an area of constant data held within the code section, typically after the end of a function and before the start of another. The pools are not executed, but their data can be accessed from surrounding code using PC-relative memory addresses. Literal pools are often used to encode constant values that do not fit into a simple move-immediate instruction.

A64 has another addressing mode specifically for accessing literal pools. In A32 and T32, the PC can be read like a general-purpose register, so a literal pool can be accessed simply by specifying PC as the base register.

PC-relative loads

In AArch64, the PC is not accessible and is treated differently to general-purpose registers; there is a special addressing mode for load instructions that accesses a PC-relative address. This special-purpose addressing mode has a much greater range than the PC-relative loads in A32 and T32 could achieve, so there can be fewer, more widely spaced, literal pools. The use of PC-relative modes is shown in the following table:

Example instruction	Description
`LDR W0, <label>`	Load 4 bytes from <label> into W0a
`LDR X0, <label>`	Load 8 bytes from <label> into X0
`LDRSW X0, <label>`	Load 4 bytes from <label> and sign-extend into X0
`LDR S0, <label>`	Load 4 bytes from <label> into S0
`LDR D0, <label>`	Load 8 bytes from <label> into D0
`LDR Q0, <label>`	Load 16 bytes from <label> into Q0

Note

`<label>` must be 4-byte-aligned for all variants.

3.6 Accessing multiple memory locations

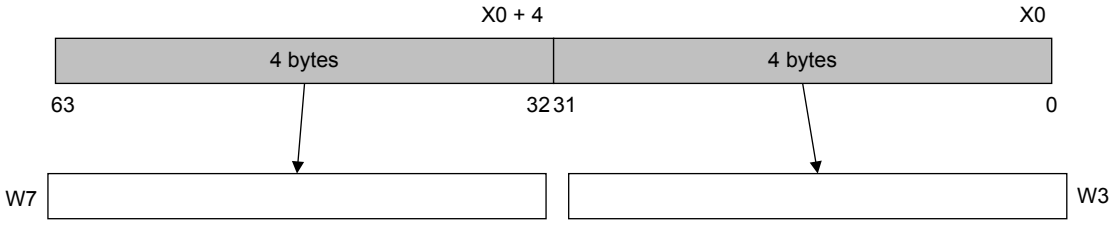
A64 provides the Load Pair (`LDP`) and Store Pair (`STP`) instructions. Data is read or written to or from adjacent memory locations. The addressing mode options that are provided for these instructions are more restrictive than for other memory access instructions. `LDP` and `STP` instructions can only use a base register with a scaled 7-bit signed immediate value, with optional pre- or post-increment. Unaligned accesses are possible for `LDP` and `STP`, unlike the 32-bit `LDRD` and `STRD` as shown in the following table:

Load and Store pair	Description
`LDP W3, W7, [X0]`	`W3 = W3 + X0` `W7 = W7 + (X0 + 4)`

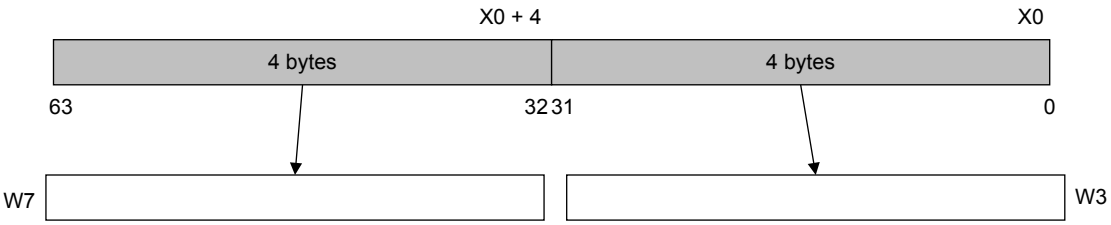
`LDP X8, X2, [X0, #0x10]!`	`X0 = X0 + 0x10`
	`X8 =(X0)`
	`X2 =(X0 + 8)`

`LDP X8, X2, [X0], #0x10`	Loads doubleword at address X0 into X8 and the doubleword at address X11 + 8 into D2 and adds 0x10 to X11.
	`X8 = X0`
	`X2 = (X11 + 8)`
	`X11 = X11 + 0x10`

`LDPSW X3, X4, [X0]`	Loads word at address X0 into X3 and word at address X0 + 4 into X4, and sign extends both to doubleword size.
`STP X9, X8, [X4]`	Stores the doubleword in X9 to address X4 and stores the doubleword in X8 to address X4 + 8.



LDP W3, W7 [X0]



LDP X8, X2, [X0 + #0x10]!

3.7 Unprivileged access

The A64 `LDTR` and `STTR` instructions perform an unprivileged Load or Store:

- At EL0, EL2 or EL3, they behave as normal Loads or Stores.
- When executed at EL1, they behave as if they were executed at privilege level EL0. They can be used to de-reference pointers that are provided with system calls, allowing OS to ensure that only data accessible to the application is accessed.

3.8 Prefetching memory

Prefetch from Memory (`PRFM`) provides a hint to the memory system that data from a particular address will be used by the program soon. The effect of this hint typically results in data or instructions being loaded into one of the caches.

The instruction syntax is:

```
PRFM <prfop>, <addr> | label
```

Where `prfop` is a concatenation of the following options:

`Type`	`PLD` or `PST` (prefetch for load or store).
`Target`	`L1`, `L2`, or `L3` (which cache to target).
`Policy`	`KEEP` or `STRM` (keep in cache, or streaming data).

For example, `PLDL1KEEP`.

3.9 Non-temporal load and store pair

ARMv8-A includes the LDNP and STNP instructions, or non-temporal load and store. These perform a read or write of a pair of register values, and give a hint to the memory system that caching is not useful for this data. The hint does not prohibit memory system activity such as caching of the address, preload, or gathering, but simply indicates that caching is unlikely to increase performance.

Non-temporal loads and stores relax the memory ordering requirements. In the following example, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

```
LDR X0, [X3]
LDNP X2, X1, [X0]                  // External observers might not observe
                                   // the loads in order (where all
                                   // observers are in the same Inner
                                   //Shareable domain.
```

For an external observer to observe the loads in order, an explicit barrier must be used:

```
LDR X0, [X3]
DMB ishld                     // Barrier forces previous memory accesses to
                              // complete before further accesses.
LDNP X2, X1, [X0]
```

3.10 Memory access atomicity

An aligned memory access, using a single general-purpose register, is guaranteed to be atomic. (An atomic load-modify-store memory operation cannot be interrupted.) Load pair and store pair instructions to a pair of general-purpose registers, using an aligned memory address are guaranteed to appear as two individual atomic accesses. Unaligned accesses are not atomic, as they typically require two separate accesses. Also, SIMD and floating-point memory accesses are not guaranteed to be atomic.

3.11 Memory barrier instructions

Both ARMv7-A and ARMv8-A provide support for different barrier operations.

- Data Memory Barrier (`DMB`). This forces all earlier-in-program-order memory accesses to become globally visible before any subsequent accesses.

  For example:

  ```
  LDR X0, [X3]
  DMB ishld              // Barrier instruction forces previous memory accesses
                         // to complete before further accesses.
  LDNP X2, X1, [X0]
  ```

- Data Synchronization Barrier (`DSB`). All pending loads and stores, cache maintenance instructions, are completed before program execution continues. A `DSB` behaves like a `DMB`, but is more restrictive. Code execution stalls, whereas `DMB` only stalls memory access.

- Instruction Synchronization Barrier (ISB). This instruction flushes the core pipeline and prefetch buffers, causing instructions after the ISB to be fetched (or refetched) from cache or memory.

ARMv8-A introduces one-sided barriers, which are associated with the Release Consistency model. These are called Load-Acquire (LDAR) and Store-Release (STLR) instructions and are address-based synchronization primitives. Only base register addressing is supported for these instructions, no offsets, or other kinds of indexed addressing are provided.

3.12 Synchronization primitives

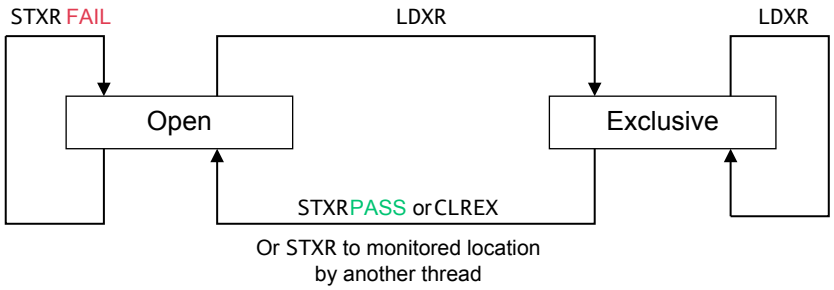
ARMv7-A and ARMv8-A architectures both provide support for exclusive memory accesses. In A64, this is achieved with the Load/Store exclusive (LDXR/STXR) pair.

There are also Load Acquire/Store Release instructions (LDAXR and STLXR).

The LDXR instruction loads a value from a memory address and attempts to claim an exclusive lock on the address. The Store-Exclusive instruction then writes a new value to that location, but only if the lock was successfully obtained and held.

The LDXR/STXR pairing is used to construct standard synchronization primitives such as spinlocks. A paired set of LDXRP and STXRP instructions is provided, to allow code to atomically update a location that spans two registers. Byte, halfword, word, and doubleword options are available. Like the Load Acquire/Store Release pairing, only base register addressing, without any offsets, is supported.

In hardware, the core includes a device that is named the local monitor. This monitor observes the address bus. When the core performs an exclusive load access, it records that fact. When it performs an exclusive store, it checks that a previous exclusive load was performed and, if this was not the case, fails the exclusive store. The architecture enables individual implementations to determine the level of checking performed by the monitor, as shown in the following figure:



The CLREX instruction clears the monitors, but unlike in ARMv7-A, exception entry or return also does the same thing. The monitor can also be cleared in error, for example by cache evictions or other reasons that are not directly related to the application. Software must avoid having any explicit memory accesses, system control register updates, or cache maintenance instructions between paired LDXR and STXR instructions.

4 Flow control

A64 instructions generally provide longer offsets, both for PC-relative branches and for offset addressing.

The increased branch range makes it easier to manage inter-section jumps. Dynamically generated code is placed on the heap so it can, in practice, be located anywhere. It is much easier for the runtime system to manager inter-section jumps with fewer requirements to insert veneers (extra code to reset the program counter).

The need for literal pools (blocks of data that are embedded in the code stream) has long been a feature of ARM instruction sets. This still exists in A64. However, the larger PC-relative load offset helps considerably with the management of literal pools, making it possible to use one per compilation unit. This removes the need to manufacture locations for multiple pools in long code sequences.

The A64 instruction set provides several kinds of branch instructions, as shown in the following table. For simple relative branches, that is, those to an offset from the current address, the B instruction is used.

Unconditional simple relative branches can branch backward or forward up to 128MB from the current program counter location. Conditional simple relative branches, where a condition code is appended to the B, have a smaller range of ±1MB.

Calls to subroutines, where the return address must be stored in the link register (X30), use the `BL` instruction. This does not have a conditional version. `BL` behaves as a `B` instruction with the additional effect of storing the return address, which is the address of the instruction after the `BL`, in register X30.

In addition to these PC-relative instructions, A64 includes two absolute branches. The `BR Xn` instruction performs an absolute branch to the address in Xn while `BLR Xn` has the same effect, but also stores the return address in X30 (the link register). The `RET` instruction behaves like `BR Xn`, but it hints to branch prediction logic that it is a function return. `RET` branches to the address in X30 by default, though other registers can be specified.

Branch instructions are shown in the following table:

Branch instructions	
B (offset)	Program relative branch forward or back 128MB. A conditional version, for example B.EQ, has a 1MB range.
BL (offset)	As B but store the return address in X30, and hint to branch prediction logic that this is a function call.
BR Xn	Absolute branch to address in Xn.
BLR Xn	As BR but store the return address in X30, and hint to branch prediction logic that this is a function call.
RET{Xn}	As BR, but hint to branch prediction logic that this is a function return. Returns to the address in X30 by default, but a different register can be specified.
Conditional branch instructions	
B.cond	Conditional branch forward or back. For example B.EQ, has a 1MB range. The branch is only taken if the condition is true (for the current value of NZCV).

A64 includes some special conditional branches. These allow improved code density because an explicit comparison is not necessary.

```
CBZ Rt, label              // Compare and branch if zero

CBNZ Rt, label             // Compare and branch if not zero
```

Both instructions compare the source register, either 32-bit or 64-bit, with zero and then perform a conditional branch. The branch offset has a range of ± 1MB. These instructions do not read or write the condition code flags (NZCV).

There are two similar test and branch instructions

```
TBZ Rt, bit, label         // Test and branch if Rt<bit> zero

TBNZ Rt, bit, label        // Test and branch if Rt<bit> is not zero
```

These instructions test the bit in the source register at the bit position that is specified by the bit value and branch conditionally depending on whether the bit is set or clear. The branch offset has a range of ±32kB. As with CBZ and CBNZ, they do not read or write the condition code flags.

5 System control and other instructions

A64 contains instructions that relate to:

- Exception handling.
- System register access.
- Debug.
- Hint instructions, which in many systems have power management applications.

5.1 Exception generating instructions

There are three instructions which cause an exception to be taken. These are used to make a call to code that runs in a higher Exception level in the OS (EL1), the Hypervisor (EL2), or Secure monitor (EL3):

```
SVC #imm16                    // Supervisor Call, allows application
// program to call the kernel (EL1).

HVC #imm16                    // Hypervisor call, allows OS code to call the
// hypervisor (EL2).

SMC #imm16                    // Secure Monitor call, allows the OS or
// hypervisor to call the Secure Monitor (EL3).
```

When the generated exception is taken in AArch64 the `#imm16` (immediate) value is made available to the handler in the Exception Syndrome Register.

5.2 Exception return instructions

The instruction that is used to generate an exception return depends on the Execution state in which the instruction is executed.

From AArch64 state

To generate an exception return, software executes the `ERET` instruction. Processor state is restored by copying SPSR_ELn to PSTATE. Execution then starts at the address that is held in the ELR_ELn register for the Exception level at which the `ERET` instruction was executed.

5.3 System register access

Two instructions are provided for system register access:

```
MRS Xt, <system register>     // This copies a system register into a
                              // general purpose register
```

For example

```
MRS X4, ELR_EL1              // Copies ELR_EL1 to X4
```

and

```
MSR <system register>, Xt    // This copies a general-purpose register
                             // into a system register
```

For example

```
MSR SPSR_EL1, X0             // Copies X0 to SPSR_EL1
```

Individual fields of PSTATE can also be accessed with MSR or MRS. For example, to select the stack pointer that is associated with EL0 or the current Exception level:

```
MSR SPSel, #imm              // For #imm value #0, SPSEL is set to 0
                             // selecting EL0 stack pointer
                             // For #imm value #1, SPSEL is set to 1
                             // selecting the current Exception level's
                             // stack pointer.
```

There are special forms of these instructions that can be used to clear or set individual exception mask bits.

```
MSR DAIFClr, #imm4
```

```
MSR DAIFSet, #imm4
```

5.4 Debug instructions

There are two debug related instructions:

There are two debug related instructions:

```
BRK #imm16                   // Generates a breakpoint Instruction
                             // exception. Sets ESR_ELx to EC value
                             // 0x3c, and sets ESR_ELx.ISS to #imm16


HLT #imm16                   // #imm is ignored by the ARM hardware but
                             // may be used to store information about
                             // the breakpoint for a debugger
```

5.5 Hint instructions

`HINT` instructions can legally be treated as a `NOP`, but they can have implementation-specific effects.

`HINT` has the syntax:

```
HINT #imm
```

Where:

`#imm` is a 7-bit unsigned immediate value identifying the type of instruction, as shown in the following table:

#imm	Instruction	Description
0	`NOP`	No operation - not guaranteed to take time to execute
1	`YIELD`	Hint that the current thread is performing a task that can be swapped out
2	`WFE`	Wait for Event
3	`WFI`	Wait for interrupt
4	`SEV`	Send Event
5	`SEVL`	Send Event Local

5.6 SIMD instructions

SIMD instructions (also called NEON instructions) have several changes, some of which are significant.

- AArch64 SIMD (NEON) instructions do not have the V prefix present in AArch32 NEON instructions.
- There is support for double precision floating-point, enabling C code using double precision floating-point to be vectorized.

There are now:

- Both vector floating-point and scalar floating-point instructions.
- Instructions to operate on scalar data that is stored in NEON registers.
- Instructions to insert and extract vector elements.
- Instructions for type conversion and saturating integer arithmetic.
- Instructions for normalization of floating-point values.
- Cross-lane instructions for vector reduction, summation, and taking the minimum or maximum value.
- Instructions to perform actions such as compare, add, find an absolute value, and negate, to operate on 64-bit integer elements.

5.7 Floating-point instructions

A64 provides a similar set of floating-point instructions to those of the ARMv7-A VFPv4 extension, which provides single and double precision mathematical operations on scalar floating-point values. There are several changes and new features:

- Floating-point comparisons set the condition flags (NZCV) directly. In A64, there is no need to explicitly transfer the comparison results from floating-point to integer flags.

- Instructions relating to the IEEE754-2008 standard, for example to calculate the minimum and maximum of a pair of numbers.
- A rounding mode can be explicitly specified when converting from integer to floating-point formats. It is no longer necessary to set the global FPCR flags when simple conversions are required in a particular rounding mode. Some of these options are also available to ARMv8-A A32 and T32.
- Instructions to support conversions between 64-bit integers and floating-point formats.
- In A64, floating-point operations involving integer types work directly on integer registers. There is no need to transfer integer values manually between floating-point and integer registers for conversion operations.

5.8 Cryptographic instructions

As the use of computer technology widens, security is becoming a concern. There are requirements to ensure privacy, to avoid information from getting into wrong hands and to enable secure and sound business practices.

There are two aspects to providing security in a modern computing system or device:

1. Reinforcing a computing device to avoid unauthorized access to data or compute resources of the system.
2. Ensuring data or information that is sent across a network remains protected in transit and is only accessible to its intended destination.

ARM TrustZone technology provides hardware mechanisms to ensure security at the node level. Various protocols and algorithms have been defined in software to protect information and prevent unauthorized access the sensitive data. Cryptography is the specific science of protecting information and cryptographic algorithms are widely used in security protocols.

There are four main aspects to a secure communication protocol.

Key exchange	The key exchange mechanism defines how keys used to encrypt and decrypt data can be exchanged
Entity authentication	Entity authentication helps assert the identity of the two parties communicating
Message transfer	This is the phase where the actual data is encrypted and transferred across the communication channel.
Message authentication	This refers to mechanisms employed to ensure that the data/information was not tampered with while it was in transit.

Symmetric block encryption algorithms like AES are used for encryption of data during the message transfer phase and hash algorithms like SHA are used for message authentication.

The cryptographic instructions are an optional extension for ARMv8-A. These cryptographic instructions are ‘helper’ functions that require software to ensure they carry out the full operation. They significantly improve performance on tasks such as AES encryption and SHA1 and SHA256 hashing.

Another important aspect of security is being sure that you are executing the correct code. For this, Trustzone provides a secure boot. A chain of trust is established, beginning with implicitly trusted hardware.

Each stage of boot uses the cryptographic helper functions to authenticate the next stage before executing code. Trusted vendors sign any code updates and the cryptographic instructions are also used to improve performance when checking these signatures.