Chapter 6
The A64 instruction set

Many programmers writing at the application level do not need to write code in assembly language. However, assembly code can be useful in cases where highly optimized code is required. This is the case when when writing compilers, or where use of low level features not directly available in C is needed. It might be required for portions of boot code, device drivers, or when developing operating systems. Finally, it can be useful to be able to read assembly code when debugging C, and particularly, to understand the mapping between assembly instructions and C statements.

6.1 Instruction mnemonics

The A64 assembly language overloads instruction mnemonics, and distinguishes between the different forms of an instruction based on the operand register names. For example, the ADD instructions below all have different encodings, but you only have to remember one mnemonic, and the assembler automatically chooses the correct encoding based on the operands.

```
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
```

6.2 Data processing instructions

These are the fundamental arithmetic and logical operations of the processor and operate on values in the general-purpose registers, or a register and an immediate value. *Multiply and divide instructions* on page 6-4 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 to be the instruction, followed by the operands, as follows:

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

The second operand might be a register, a modified register, or an immediate value. The use of `R` indicates that it can be either an X or a W register.

The 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.

6.2.1 Arithmetic and logical operations

Table 6-1 shows some of the available arithmetic and logical operations.

Table 6-1 Arithmetic and logical operations

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

Some instructions also have an S suffix, indicating that the instruction sets flags. Of the instructions in Table 6-1, this includes `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
```

Example 6-1 Arithmetic instructions

```
ADD W0, W1, W2, LSL #3          // W0 = W1 + (W2 << 3)
SUBS X0, X4, X3, ASR #2         // X0 = X4 - (X3 >> 2), set flags
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 essentially the same as the corresponding boolean operators operating on individual bits of the register.

The `BIC` (Bitwise bit Clear) instruction performs an `AND` of the register that is the first after the destination register, with the inverted value of the second operand. For example, to clear bit [11] of register `X0`, use:

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

`ORN` and `EON` perform an `OR` or `EOR` respectively with a bitwise-NOT of the second operand.

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

6.2.2 Multiply and divide instructions

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

Table 6-2 Multiplication operations in assembly language

Opcode	Description
Multiply instructions	
`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
Divide instructions	
`SDIV`	Signed divide
`UDIV`	Unsigned divide

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.

```
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)
```

Additionally, 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 give a result size the same size as the operands:

- 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.

Widening multiply, that is signed and unsigned, with accumulation gives a single 64-bit result:

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

A 64 × 64 to 128-bit multiply requires a sequence of two instructions to generate a pair of 64-bit result registers:

- ± (64 × 64) gives the lower 64 bits of the result [63:0].
- (64 × 64) gives the higher 64 bits of the result [127:64].

——— **Note** ———

The list contains no 32 × 64 options. You cannot directly multiply a 32-bit W register by a 64-bit X register.

————————

The ARMv8-A architecture has support for signed and unsigned division of 32-bit and 64-bit sized values. 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 used for the operation. The result is always rounded towards zero, as in most C/C++ dialects.

6.2.3 Shift operations

The following instructions are specifically for shifting:

- *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.

Table 6-3 Shift and move operations

Instruction	Description
Shift	
`ASR`	Arithmetic shift right
`LSL`	Logical shift left
`LSR`	Logical shift right
`ROR`	Rotate right
Move	
`MOV`	Move
`MVN`	Bitwise NOT

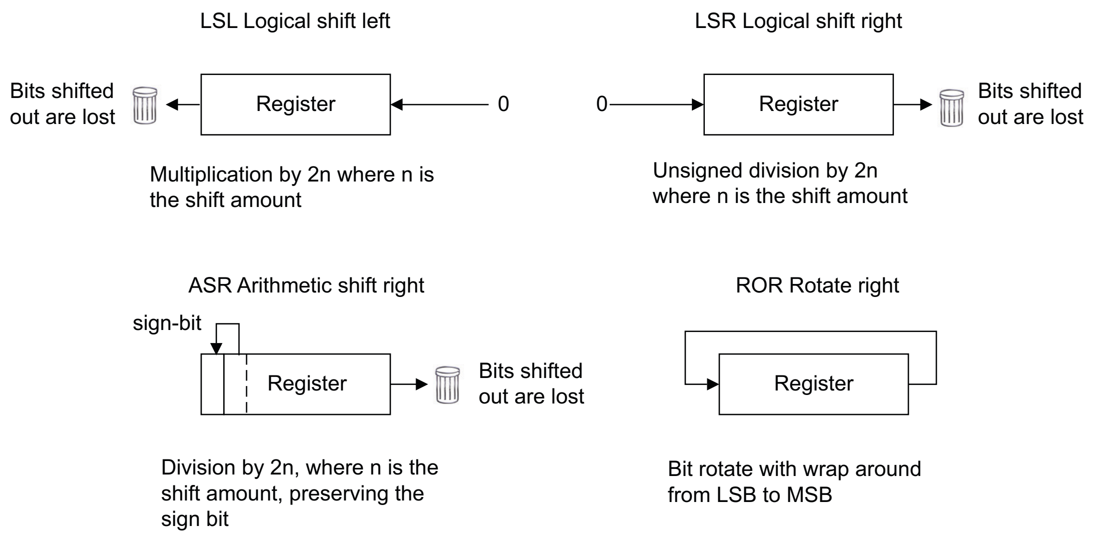
LSL Logical shift left
Bits shifted out are lost
Register
0
Multiplication by 2n where n is the shift amount
LSR Logical shift right
0
Register
Bits shifted out are lost
Unsigned division by 2n where n is the shift amount
ASR Arithmetic shift right
sign-bit
Register
Bits shifted out are lost
Division by 2n, where n is the shift amount, preserving the sign bit
ROR Rotate right
Register
Bit rotate with wrap around from LSB to MSB

Figure 6-1 Shift operations

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

6.2.4 Bitfield and byte manipulation instructions

There are instructions that extend a byte, halfword, or word to register size, which can be either X or W. These instructions exist in both signed (SXTB, SXTH, SXTW) and unsigned (UXTB, UXTH) variants and are aliases to the appropriate bitfield manipulation instruction.

Both the signed and unsigned variants of these instructions extend a byte, halfword, or word (although only SXTW operates on a word) to register size. The source is always a W register. The destination register is either an X or a W register, except for SXTW which must be an X 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 are similar to those that exist in ARMv7 and 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).

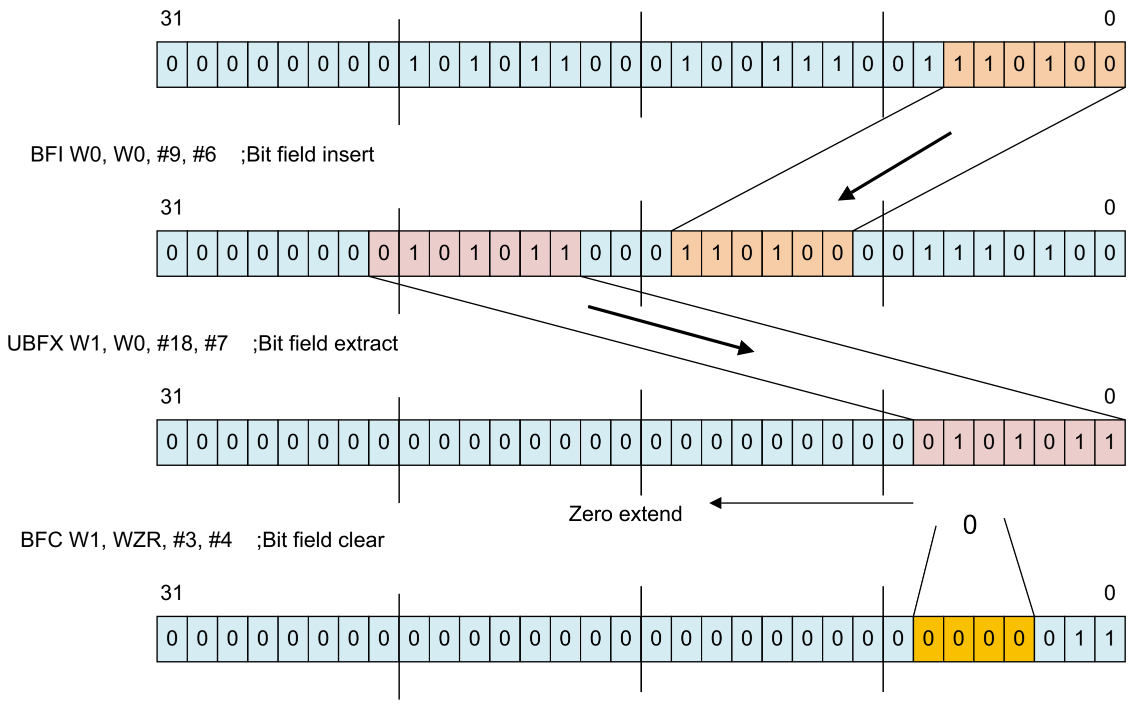

Figure 6-2 Bit manipulation instructions

Note

There are also BFM, UBFM, and SBFM instructions. These are *Bit Field Move* instructions, which are new for ARMv8. However, the instructions do not need to be used explicitly, as aliases are provided for all cases. These aliases are the bitfield operations already described: [SU]XT[BHWX], ASR/LSL/LSR immediate, BFI, BFXIL, SBFIZ, SBFX, UBFIZ, and UBFX.

If you are familiar with the ARMv7 architecture, you might recognize the other bit manipulation instruction:

- 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.

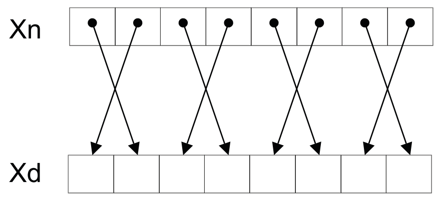

Figure 6-3 REV16 instruction

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

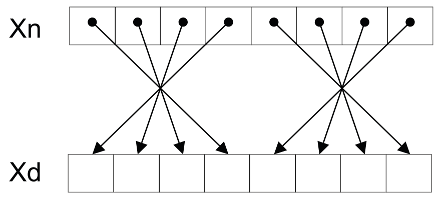

Figure 6-4 REV32 instruction

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.

6.2.5 Conditional instructions

The A64 instruction set does not support conditional execution for every instruction. Predicated execution of instructions does not offer sufficient benefit to justify its significant use of opcode space.

Processor state on page 4-6, describes the four status flags, Zero (Z), Negative (N), Carry (C) and Overflow (V). Table 6-4 indicates the value of these bits for flag setting operations.

Table 6-4 Condition flag

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.

Note

The condition flags (NZCV) and the condition codes are the same as in A32 and T32. However, A64 adds NV (`0b1111`), though it behaves the same as its complement, AL (`0b1110`). This differs from A32, which did not assign any meaning to `0b1111`.

Table 6-5 Condition codes

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

There are a small set of conditional data processing instructions. These instructions are unconditionally executed but use the condition flags as an extra input to the instruction. This set has been provided to replace common usage of conditional execution in ARM code.

The instructions types which 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.

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

The A64 instruction set enables conditional execution of only program flow control branch instructions. This is in contrast to A32 and T32 where most instructions can be predicated with a condition code. These can be summarized as follows:

Conditional select (move)

- `CSEL` Select between two registers based on a condition. Unconditional instructions, followed by a conditional select, can replace short conditional sequences.
- `CSINC` Select between two registers based on a condition. Return the first source register or the second source register incremented by one.
- `CSINV` Select between two registers based on a condition. Return the first source register or the inverted second source register.
- `CSNEG` Select between two registers based on a condition. Return the first source register or the negated second source register.

Conditional set

Conditionally select 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`) Sets 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 very useful for expressing nested or compound comparisons.

Note

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

For example:

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

Some aliases to the example instructions are provided, where either the zero register is used, or the same register is used as both destination and both source registers for the instruction.

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 simple C code:

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

This might produce 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
```

6.3 Memory access instructions

As with all prior ARM processors, the ARMv8 architecture is a Load/Store architecture. This means that no data processing instruction operates directly on data in memory. The data must first be loaded into registers, modified, and then stored to memory. The program must specify an address, the size of data to be transferred, and a source or destination register. There are additional Load and Store instructions which provide further options, such as non-temporal Load/Store, Load/Store exclusives, and Acquire/Release.

Memory instructions can access Normal memory in an unaligned fashion (see Chapter 13 *Memory Ordering*). This is not supported by exclusive accesses, load acquire or store release variants. If unaligned accesses are not desired, they can be configured to be faulted.

6.3.1 Load instruction format

The general form of a Load instruction is as follows:

```
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>` (see *Specifying the address for a Load or Store instruction* on page 6-14). Programmers will not normally need 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 a W register effectively zero extends to the entire register width.

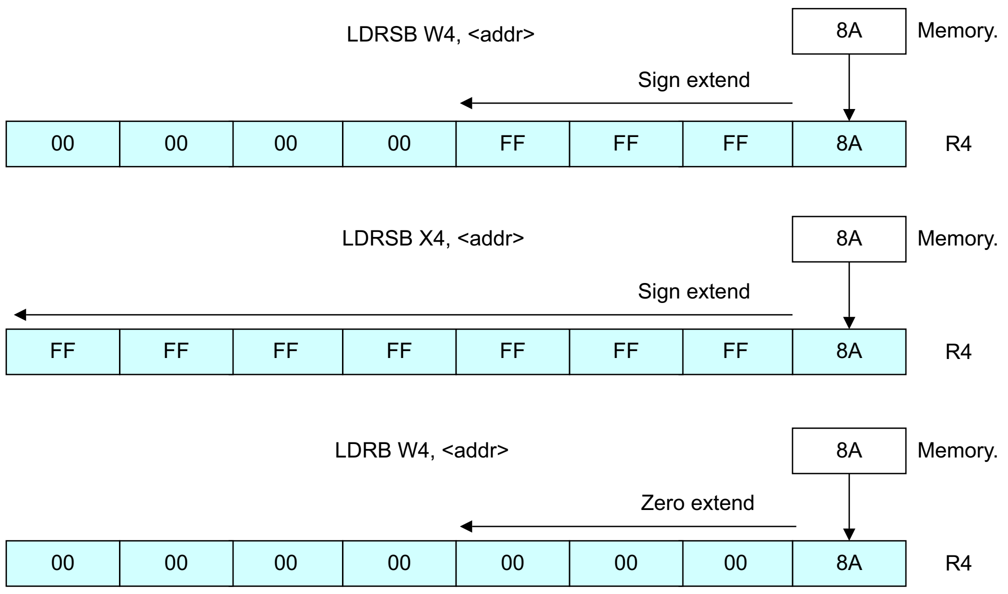

Figure 6-5 Load instructions

6.3.2 Store instruction format

Similarly, the general form of a Store instruction is as follows:

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

There are also unscaled-offset forms such as `STUR<type>` (see *Specifying the address for a Load or Store instruction* on page 6-14). Programmers will not normally need 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 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.

6.3.3 Floating-point and NEON scalar loads and stores

Load and Store instructions can also access floating-point/NEON registers. Here, 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. This information is summarized in Table 6-6, and Table 6-7.

For Load instructions:

Table 6-6 Memory bits written by Load instructions

Load	Xt	Wt	Qt	Dt	St	Ht	Bt
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	-	-	-	-	-	-
LDPSW	-	-	-	-	-	-	-

For Store instructions:

Table 6-7 Memory bits read by Store instructions

Store	Xt	Wt	Qt	Dt	St	Ht	Bt
STR	64	32	126	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`.

Note

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

6.3.4 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 as well as 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.

Table 6-8 Offset addressing modes

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)

Typically, when specifying a shift or extension option, the shift amount can be either 0 (the default) or log2 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, and post-index modes which apply the offset *after* accessing the memory.

Table 6-9 Index addressing modes

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:

```
// A 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');
}
```

PC-relative modes (load-literal)

A64 adds another addressing mode specifically for accessing literal pools. Literal pools are blocks of data encoded in an instruction stream. 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.

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.

In A64, PC is not generally accessible, but instead there is a special addressing mode (for load instructions only) that accesses a PC-relative address. This special-purpose addressing mode also has a much greater range than the PC-relative loads in A32 and T32 could achieve, so literal pools can be positioned more sparsely.

Table 6-10

Example instruction	Description
`LDR W0, <label>`	Load 4 bytes from <label> into W0
`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.

6.3.5 Accessing multiple memory locations

A64 does not include the Load Multiple (`LDM`) or Store Multiple (`STM`) instructions that are available to A32 and T32 code.

In A64 code, there are the *Load Pair* (`LDP`) and *Store Pair* (`STP`) instructions. Unlike the A32 `LDRD` and `STRD` instructions, any two integer registers can be read or written. Data is read or written to or from adjacent memory locations. The addressing mode options 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`.

Table 6-11 Register Load/Store pair

Load and Store pair	Description
`LDP W3, W7, [X0]`	Loads word at address X0 into W3 and word at address X0 + 4 into W7. See Figure 6-6.
`LDP X8, X2, [X0, #0x10]!`	Loads doubleword at address X0 + 0x10 into X8 and the doubleword at address X0 + 0x10 + 8 into X2 and add 0x10 to X0. See Figure 6-7.
`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.
`LDP D8, D2, [X11], #0x10`	Loads doubleword at address X11 into D8 and the doubleword at address X11 + 8 into D2 and adds `0x10` to X11.
`STP X9, X8, [X4]`	Stores the doubleword in X9 to address X4 and stores the doubleword in X8 to address X4 + 8.

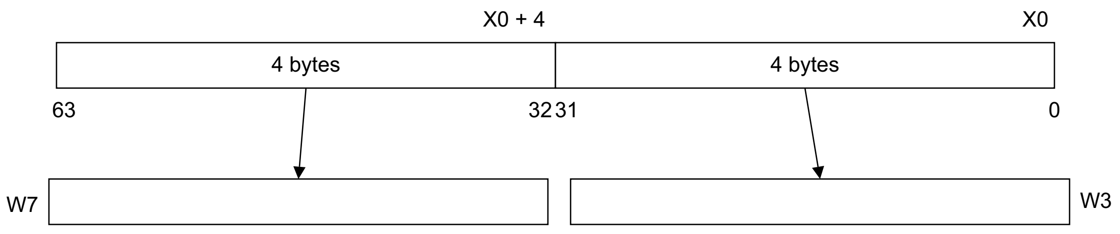

Figure 6-6 LDP W3, W7 [X0]

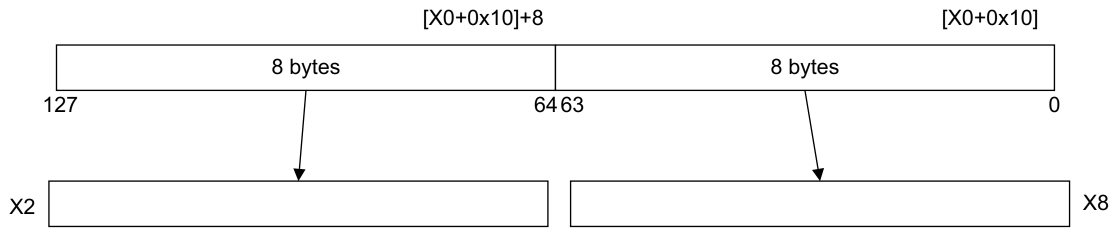

Figure 6-7 LDP X8, X2, [X0 + #0x10]!

6.3.6 Unprivileged access

The A64 `LDTR` and `STTR` instructions perform an unprivileged Load or Store (see `LDTR` and `STTR` in *ARMv8-A Architecture Reference Manual*):

- At EL0, EL2 or EL3, they behave as normal Loads or Stores.
- When executed at EL1, they behave as if they had been executed at privilege level EL0.

  These instructions are equivalent to the A32 `LDRT` and `STRT` instructions.

6.3.7 Prefetching memory

Prefetch from Memory (`PRFM`) enables code to provide a hint to the memory system that data from a particular address will be used by the program soon. The effect of this hint is IMPLEMENTATION DEFINED, but typically, it 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`.

These instructions are similar to the A32 `PLD` and `PLI` instructions.

6.3.8 Non-temporal load and store pair

A new concept in ARMv8 is the *non-temporal* load and store. These are the `LDNP` and `STNP` instructions that perform a read or write of a pair of register values. They also 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. However, it indicates that caching is unlikely to increase performance. A typical use case might be streaming data, but take note that effective use of these instructions requires an approach specific to the microarchitecture.

Non-temporal loads and stores relax the memory ordering requirements. In the above case, the `LDNP` instruction might be observed before the preceding `LDR` instruction, which can result in reading from an uncertain address in X0.

For example:

```
LDR X0, [X3]
LDNP X2, X1, [X0]      // Xo may not be loaded when the instruction executes!
```

To correct the above, you need an explicit load barrier:

```
LDR X0, [X3]
DMB nshld
LDNP X2, X1, [X0]
```

6.3.9 Memory access atomicity

An aligned memory access, using a single general-purpose register, is guaranteed to be atomic. 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. Additionally, floating-point and SIMD memory accesses are not guaranteed to be atomic.

6.3.10 Memory barrier and fence instructions

Both ARMv7 and ARMv8 provide support for different barrier operations. These are described in more detail in Chapter 13 *Memory Ordering*:

- *Data Memory Barrier* (`DMB`). This forces all earlier-in-program-order memory accesses to become globally visible before any subsequent accesses.
- *Data Synchronization Barrier* (`DSB`). All pending loads and stores, cache maintenance instructions, and all TLB maintenance instructions, are completed before program execution continues. A `DSB` behaves like a `DMB`, but with additional properties.
- *Instruction Synchronization Barrier* (`ISB`). This instruction flushes the CPU pipeline and prefetch buffers, causing instructions after the ISB to be fetched (or re-fetched) from cache or memory.

ARMv8 introduces one-sided *fences*, which are associated with the Release Consistency model. These are called *Load-Acquire* (`LDAR`) and *Store-Release* (`STLR`) and are address-based synchronization primitives. (See *One-way barriers* on page 13-8.) The two operations can be paired to form a full fence. Only base register addressing is supported for these instructions, no offsets or other kinds of indexed addressing are provided.

6.3.11 Synchronization primitives

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

The `LDXR` instruction loads a value from a memory address and attempts to silently claim an exclusive lock on the address. The Store-Exclusive instruction then writes a new value to that location 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.

The `CLREX` instruction clears the monitors, but unlike in ARMv7, exception entry or return also clears the monitor. The monitor might also be cleared spuriously, for example by cache evictions or other reasons 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.

There is also an exclusive pair of Load Acquire/Store Release instructions called `LDAXR` and `STLXR`. See *Synchronization* on page 14-6.

6.4 Flow control

The A64 instruction set provides a number of different kinds of branch instructions (see Table 6-12). 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 it is necessary for the return address to 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.

Table 6-12 Branch instructions

	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
`CBZ Rt, label`	Compare and branch if zero. If Rt is zero, branch forward or back up to 1MB.
`CBNZ Rt, label`	Compare and branch if non-zero. If Rt is not zero, branch forward or back up to 1MB.
`TBNZ Rt, bit, label`	Test and branch if zero. Branch forward or back up to 32kB.
`TBNZ Rt, bit, label`	Test and branch if non-zero. Branch forward or back up to 32kB.

In addition to these PC-relative instructions, the A64 instruction set 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..

The A64 instruction set includes some special conditional branches. These allow improved code density in some cases because an explicit comparison is not necessary.

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

These instructions compare the source register, either 32-bit or 64-bit, with zero and then conditionally perform a 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 specified by the immediate and conditionally branch depending on whether the bit is set or clear. The branch offset has a range of ±32kB. As with `CBZ`/`CBNZ`, these instructions do not read or write the condition code flags (NZCV).

6.5 System control and other instructions

The A64 instruction set contains instructions that relate to:

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

6.5.1 Exception handling instructions

There are three exception handling instructions whose purpose it is to 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 hypervisor (EL2).
  ```

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

The immediate value is made available to the handler in the *Exception Syndrome Register*. This is a change from ARMv7, where the immediate value had to be determined by reading the opcode of the calling instruction. See Chapter 10 *AArch64 Exception Handling* for further information.

To return from an exception, use the `ERET` instruction. This instruction restores processor state by copying SPSR_EL*n* to `PSTATE` and branches to the saved return address in ELR_EL*n*.

6.5.2 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
  ```

-
  ```
  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 associated with EL0 or the current Exception level:

-
  ```
  MSR SPSel, #imm        // A value of 0 or 1 in this register is used to select
                         // between using EL0 stack pointer or the current exception
                         // level stack pointer
  ```

There are special forms of these instructions that can be used to clear or set individual exception mask bits (see *Saved Process Status Register* on page 4-5):

- `MSR DAIFClr, #imm4`
- `MSR DAIFSet, #imm4`

See *System registers* on page 4-7.

6.5.3 Debug instructions

There are two debug-related instructions:

- `BRK #imm16   // Enters monitor mode debug, where there is on-chip debug monitor`
  `             // code`
- `HLT #imm16   // Enters halt mode debug, where external debug hardware is connected`

For information on debugging, see Chapter 18 *Debug*.

6.5.4 Hint instructions

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

- `NOP        // No operation - not guaranteed to take time to execute`
- `YIELD      // Hint that the current thread is performing a task that`
  `           // can be swapped out`
- `WFE        // Wait for Event`
- `WFI        // Wait for interrupt`
- `SEV        // Send Event`
- `SEVL       // Send Event Local`

These concepts are also covered in Chapter 14 *Multi-core processors* and Chapter 15 *Power Management*.

6.5.5 NEON instructions

The NEON instruction set also has several enhancements, some of which are quite significant. Chapter 7 *AArch64 Floating-point and NEON* describes these in more detail.

Changes to NEON in A64 include

- Support for double precision floating-point, enabling C code using double precision floating-point to be vectorized.
- New instructions to operate on scalar data stored in NEON registers.
- New instructions to insert and extract vector elements.
- New instructions for type conversion and saturating integer arithmetic.
- New instructions for normalization of floating-point values.
- New cross-lane instructions for vector reduction, summation, and taking the minimum or maximum value.
- Instructions to perform actions such as compare, add, find absolute value, and negate have been extended to be able to operate on 64-bit integer elements.

6.5.6 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 a number of 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 have been added relating to the IEEE754-2008 standard, for example to calculate the minimum and maximum of a pair of numbers.
- A rounding mode can now 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 A32 and T32.
- Instructions have been added 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 manually transfer integer values between floating-point and integer registers for conversion operations.

6.5.7 Cryptographic instructions

An optional extension for ARMv8 adds cryptographic instructions that significantly improve performance on tasks such as AES encryption and SHA1 and SHA256 hashing.