RISC-V Assembly Language

November 1st 2024

Alan Johnson

Published by xelsys

ISBN 978-1-300-85087-8

This is the first release of a textbook covering ARM64 programming. Subsequent editions will include bare metal coding, interfacing to Raspberry Pi GPIO pins and will cover system registers and interrupt processing in more detail.

Feedback is greatly encouraged and will be acknowledged (if desired by the contributor) in future editions.

Send comments to arm64bookfeedback@duck.com

View on Github for updates https://github.com/alanxelsys/ARM64Book

Contents

Figures

Listings

Tables

Chapter 1. The fundamentals.

This chapter provides a foundation for the topics that will be discussed as the book progresses. It is reasonably general, staying away from any specific architecture.

Pre-requisites are not too demanding; however, knowledge of the following areas will ease the journey.

- Familiarity with basic computer hardware
- Microprocessor architecture
    - Memory and data buses, register, ALUs, ...
- Knowledge of Linux ®
    - Installation of the Operating System and applications
    - Bash
- Basic knowledge of the C programming language
- High school, level mathematics, although college level is helpful for some of the material in chapter 8.
- An inexpensive computing device such as the Raspberry Pi.

What is assembly language?

Many high-level languages place a strong emphasis on abstraction, treating functions as impenetrable black boxes and hiding the inner working. Assembly language takes a different approach and allows (indeed mandates) the coder to familiarize themself with the innards of the system.

The former method is similar to Rapid Application Development (RAD) methodology that works well with teams whereas the second approach often includes smaller groups with specialized knowledge. Both approaches have their place. Digital computers inherently process data in one of two states (binary) so it is essential that we understand the low level world of one's and zero's.

Processors have different *architectures* and they each understand their own *machine code* instructions – at their very heart these instructions are combinations of binary numbers that instruct the processor how to proceed. Binary numbers are cumbersome for human operators and instead a set of *mnemonic* instructions are used. A hypothetical example could be an instruction such as `add r1, r2,r3` which would add two numbers together that are contained in register2[1] and register3, placing the result in register1 or `add r1, r2, 45` which could add the value 45 to the value contained in register2, placing the result in register1. The corresponding native machine code (again hypothetical) could be the binary code 10101100 00010010 00101100. The *mnemonic* instructions make up the *assembly language*.

The role of the assembler (program) is to convert programmer-readable assembly instructions into the corresponding machine code instructions. The output code is termed an *object* file. Conversely a disassembler converts machine code instructions back into assembly language. The assembler has additional roles such as understanding a set of *directives* that can define and place data into the computer's memory locations. An example could be a set of error codes defined as textual informational messages. These messages are defined by the programmer rather than the specific processor itself. There are a number of these directives, and they will be discussed in more detail as the document progresses.

Higher-level languages use *compilers* to translate to machine code. After the assembly or compilation process the object files are *linked* to form an executable program. The *linker* may act on individual or multiple files. High level language instructions do not normally have a one-to-one correspondence with the underlying machine code instructions. They are designed to be more instinctive to the programmer by providing English like keywords such as if … then, while, and print. High level languages can be *interpretive* and translated into machine code instructions during runtime, or pre-compiled before runtime into native machine-code.

Why use assembly?

Assembly language has a direct relationship with the CPU that it is running on and as a result the programs will be more compact and efficient. It is also more suited to *system-level* programming. A disadvantage is that many lines of code may be required when compared to high level languages and as a result a hybrid approach may be deployed where the bulk of the code could be written using *C* or *Python* which can pass parameters *to* and accept return values *from* a smaller section of assembly code. *Portability* is also an issue since the assembly language is tightly coupled with the CPU that it is running on.

[1] Registers are low-capacity storage elements (typically anywhere from one to eight bytes in size) high-speed devices contained within the processor architecture.

Experienced system-level coders may wish to skip this chapter or treat it as a refresher. The material discussed in *this* chapter is general and does not necessarily apply to any specific system.

Hardware Vs Software Vs Firmware

Hardware

In computer terms *hardware* refers to the physical components that make up the system. Hardware is something that can be seen and touched.

Software

Software refers to the actual instructions that are loaded into the computer's memory. These instructions may direct the hardware to perform certain tasks. For example, the *system software* is responsible for displaying the result of an operation onto a hardware output device such as a display screen or printer and for taking input from a device such as a keyboard. Most users are more familiar with application software such as word processors, email, spreadsheets, etc.

Firmware

Firmware can be thought of as a set of instructions residing in hardware. This definition has become somewhat blurred as these instructions were originally loaded onto read only devices (ROMs). These devices would be physically replaced when new upgrade code was required. Over time Erasable Programmable integrated circuits (IC's) (EPROMs) were introduced, which as the name implies could be written over with new code. Today, non-volatile random-access memory (NVRAM) devices are used and can often be upgraded on-line without even requiring a reboot. This process is sometimes referred to as *flashing* since the underlying device is often Flash memory.

Number Systems

Anthropologists may make a claim that we count in base 10 as this is the number of digits on our hands. Other cultures have used base 60 and base 20 (possibly using both fingers and toes). These number systems are not as well suited to computer systems and today[2] base 2 and base 16 dominate when using low-level assembly programming.

Binary, Octal, Hexadecimal

Consider the base 10 number 4673_{10} – this breaks down into:

4×10^3

+

6×10^2

+

7×10^1

+

3×10^0

= 4000 + 600 + 70 + 3 = 4673

The use of ten (0-9) different characters along with their position represented a major advance in computation when compared to systems such as the Roman counting method.

Digital electronic systems naturally gravitate towards a two-state binary system where current either flows or it does not. These two states are represented by the symbols 0 or 1.

Each binary digit is termed a *bit(b)*. For convenience, binary digits are often grouped into 8 bits termed a *Byte(B)*. Since eight bits can represent numbers ranging from 00000000 through 11111111, the decimal values translate to 0 through 255. A disadvantage of binary numbers is that a three-digit decimal number may require an equivalent binary number up to ten binary digits. A more compact numbering system is base 16 (hexadecimal) which treats a group of four binary numbers as a single hexadecimal number. This means that two hexadecimal numbers will represent a single byte[3]. Hexadecimal numbers use the same symbols as decimal up to the value 9, and then use the characters A through F to represent the decimal numbers 10 through 15. The hex number 10_{16} corresponds to decimal number $16_{10.}$

[2] Base 8 - Octal was also used on many earlier computers such as Digital Equipment Corporation's PDP family of minicomputers.

[3] A single hexadecimal number is sometimes referred to as a nibble.

Table 1-1 Binary, Decimal and Hexadecimal equivalents

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Converting Binary to Decimal

Each binary digit can be converted to decimal by multiplying its value by two raised to an index where the index corresponds to the bit's position.

Table 1-2 Converting Binary to Decimal

The binary number 110101_2 then, can be converted to decimal using the following steps.

$1 \text{ x } 2^5 + 1 \text{ x } 2^4 + 0 \text{ x } 2^3 + 1 \text{ x } 2^2 + 0 \text{ x } 2^1 + 1 \text{ x } 2^0 =$

Value	**1**	**1**	**0**	**1**	**0**	**1**
Position	5	4	3	2	1	0
Multiply by	2^5	2^4	2^3	2^2	2^1	2^0

32 + 16 + 0 + 4 + 0 + 1

$= 53_{10}$

Converting Decimal to Binary

The following method breaks down a decimal number into powers of two, so to convert the number 843_{10} to its equivalent binary number –

1. First get the highest power of two contained in 843 which is 512 (2^9).

2. Subtract 512 from 843 = 331,
3. The highest power of two contained in 331 is 256 (2^8),
4. Subtract 256 from 331 to get 75,
5. The highest power of two contained in 75 is 64 (2^6),
6. Subtract 64 from 75 to get 11,
7. The highest power of two contained in 11 is 8(2^3),
8. Subtract 8 from11 to get 3,
9. The highest power of two contained in 3 is 2 (2^1),
10. Subtract from 3 to get 1,
11. The highest power of two contained in 1 is 1 (2^0),
12. Subtract 1 from 1 to get 0.

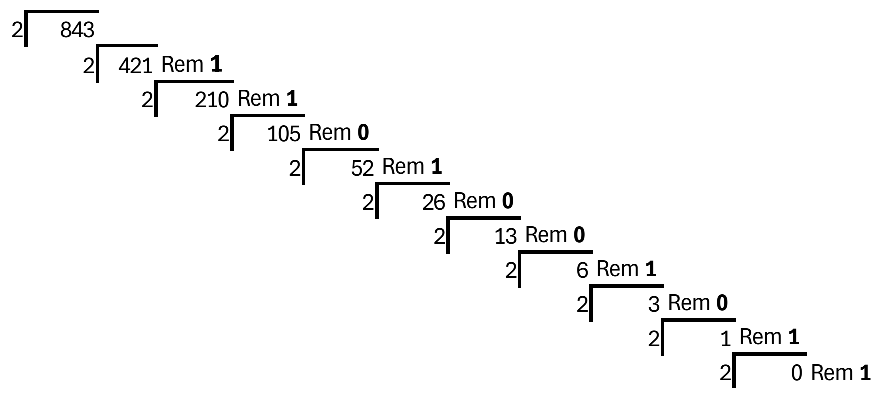
Everywhere that a power of two appears, write its index as the binary value one and where it did not appear write the binary value zero using the positional notation shown in Table 1-2.

Table 1-3 Converting decimal to binary

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0	1	1

Another way of converting is a repeated division method. Divide the number repeatedly until zero is reached. Take note of the remainders and put the first remainder in the left-most position, then the second remainder into the left-most second position, repeating until all reminders have been recorded.

Figure 1-1Converting Decimal to binary using repeated division by 2_{10}



Now write down the remainders starting from the *top* to get:

1101001011_2.

Converting Hexadecimal to Decimal

A hex number such as $5B7C_{16}$ can be converted to decimal using a power of sixteen method –

= 5 x 16^3, + B x 16^2, + 7 x 16^1, + C x 16^0

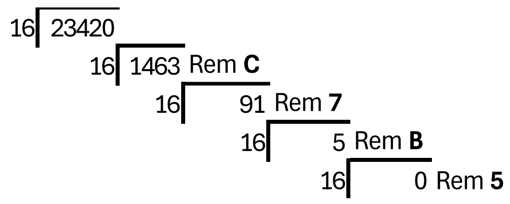
= *20,480* + *2816* + *112* + *12*

= *23420*

Converting Decimal to Hexadecimal

Take the number as shown, divide repeatedly by 16_{10} until zero is reached. Record the remainders in base 16 format (e.g. for a remainder of 10_{10}, record “A”). Note the remainders and put the last remainder in the left-most position, the second from last remainder into the left-most second position, repeating until all reminders have been recorded.

Figure 1-2 Converting Decimal to binary using repeated division by 16_{10}



Binary Fractions

The binary numbers that have been dealt with up to this point are *natural* number equivalents (positive whole numbers). Positional notation is used to show the corresponding power of two index. [4] Fractions can be represented in binary by moving to the left of the 2^0. These values then become 2^{-1}, 2^{-2}, . . .

Converting a binary fraction to decimal

1101.01 is equivalent to the base 10 number 13.25 since we have:

1 x 2^3 + 1 x 2^2 + 0 x 2^1 +1 x 2^0 + 0x 2^{-1} + 1x$2^{-2.}$

[4] Recall that negative indices can be resolved by changing the sign of the index and changing the operation from division to multiplication and vice versa so that 1 / 2^{-2} becomes 1 x 2^2 = 4 and 4 x 2^2 = 4/2^{-2} = 16

Converting a decimal fraction to binary.

Repeatedly multiply the fractional part by two until it becomes zero, taking note of the value to the left (integer portion) of the decimal point. Accumulate the values of the integer part from top to bottom to get the binary fractional part.

Example 0.625₁₀

0.625 x 2 = 1.25

0.25 x 2 = 0.5

0.5 x 2 = 1.0

Stop since the value to the right of the decimal point =0

Take the integer value from top to bottom = 0.101_2

The next example shows a recurring fraction

Example 0.3

0.3 x 2 = 0.6
0.6 x 2 = 1.2
0.2 x 2 = 0.4
0.4 x 2 = 0.8
0.8 x 2 = 1.6
0.6 x 2 = 1.2
0.2 x 2 = 0.4
0.4 x 2 = 0.8
0.8 x 2 = 1.6
0.6 x 2 = 1.2

This highlighted value has been met before, so this is a recurring fraction with the pattern 0011 repeating - .0100110011… This means that when evaluating, a halt counter should be added. The logic would be to end when the fractional part = 0 or when the required degree of precision has been reached.

One and Two's complement

An eight-bit byte can represent any one of 256 values ranging from $0 – 255_{10}$. This is known as *unsigned* notation. Another representation is to use half of the range as positive integers and the other half as negative, in this case the range is from +127[5] through -128. This method uses the *most significant bit* to represent the sign and is known as *signed* notation. The number line for an eight-bit signed number is:

-128, -127, …, 0, 1, 2, …, 127

Table 1-4 Signed number representation.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Sign bit	Magnitude Bits						

Interpreting the value of a signed number is straightforward –

The procedure is to add the corresponding powers of two of each bit's place value but leave out the sign bit. The next step is to add in the value of the sign bit. For positive numbers it makes no difference since the value of the sign bit is zero, but for negative numbers the value of the sign bit is -128.

Example

- Take the positive binary number 00101100
- Add the magnitude bits together

$0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

= 32 + 8 + 4 = 44

- Add in the value of the magnitude bit (2^7) to get:-

0 + 44 = 44

- For the negative number 10011001
- Add the magnitude bits together.

$0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

= 16 + 8 + 1 = 25

- Add in the value of the magnitude bit (2^7)to get

[5] Zero is treated as a positive number here

-128 + 25 = -103

Converting from a signed number to an unsigned number is a simple operation, the procedure is to invert the bits and then add the binary value 1.

So, to convert the positive number 63_{10} to negative 63_{10}.

- Convert the number to an eight-bit binary number -

00111111

- Invert the bits to get -

11000000 (one's complement)

- Add 1 to get –

11000001 (two's complement)

- Convert back to decimal to get:-

-128+64+1 = 63

The first stage of inverting the bits - obtains the one's complement, adding the binary digit 1 to the one's complement - obtains the two's complement.

The following table shows an extract of the first few signed numbers.

Table 1-5 Signed and unsigned numbers

Signed Binary Number	Decimal Equivalent
0111 1111	127
0111 1110	126
0111 1101	125
. .	.
0000 0000	0
1111 1111	-1
1111 1110	-2
.. . .	
1000 0010	-126
1000 0001	-127
1000 0000	-128

Addition and subtraction of binary numbers

Binary Addition

To add two binary numbers together is straightforward, there are only four outcomes.

- 0 + 0 = 0
- 0 + 1 = 1
- 1+ 0 = 1
- 1 + 1 =10 (0+ carry)

An example of an unsigned binary addition follows-

0	0	1	0	1	1	0	1
0	1	1	1	0	1	0	0
1	0	1	0	0	0	0	1

Checking by adding the decimal number equivalents together –

45 + 116 = 161

Consider if these numbers being added were in signed notation – here adding two positive numbers together would result in a negative number since the sign bit of the result = 1. This is an *overflow* condition since the result of 161 is clearly outside of the maximum positive number that can be represented in signed eight-bit binary arithmetic. This is something that needs to be checked and there are conditions built-in to the processor architecture to detect this kind of situation.

Larger numbers can be dealt with by using two bytes for storage, treating the second byte as having the values 2^8 through 2^{15}. Assemblers and compilers will refer to groups of bytes by designations such as long int, word etc. It is important to check the definitions.

One such definition is:

Table 1-6 Data type sizes

Unit	Width
Doubleword	64 bits
Word	32 bits
Halfword	16 bits
Byte	8 bits

Of course, it is important to specify signed or unsigned, again a definition for an unsigned integer in the programmer's documentation might be referred to as `uint`.

Binary subtraction

Binary subtraction can be dealt with using elementary rules for small numbers and then taking into account “borrows” rather than “carrys” but using the two’s complement method described on page 1-8 is by far the preferred method for larger numbers.

The steps for binary subtraction are:

1. Obtain the two’s complement of the *subtrahend* (the number that will be taken away)
2. Add this to the *minuend* (the number that will be subtracted from).
3. Add the two’s complement of the subtrahend to the minuend.
4. If there is a carry after the addition, then drop the carry (final result is positive)
5. If there is no carry, then compute the two’s complement of the result (final result is negative)

Taking a concrete example of subtracting 00100100 (36_{10}) from 00000010 (2_{10})

- Two’s complement of the subtrahend

1101 1011 +1 = 1101 1100

- Add to the minuend

0	0	0	0	0	0	1	0	Minuend
1	1	0	1	1	1	0	0	Two’s complement of subtrahend
1	1	0	1	1	1	1	0	

(Carry = 0)

Two’s complement of the result is

00100001+1 = 00100010

Result is negative since the carry was false = -34

Another example -

- Subtract 45_{10} from 120_{10}
- Convert numbers to eight-bit binary

45_{10}= 0010 1101_2

120_{10} = 0111 1000_2

- Two’s complement of 00101101

1101 0011

- Add to 0111 1000

0	1	1	1	1	0	0	0
1	1	0	1	0	0	1	1
0	1	0	0	1	0	1	1

(carry = 1)

The result is positive since carry was zero, 01001011 = 75_{10}

Binary multiplication

The rules for multiplication of two bits are

0 x 0 = 0

0 x 1 = 0

1 x 0 =0

1 x 1 =1

Note anything multiplied by zero is of course zero.

Example multiply binary 10 (2_{10}) by 11 (3_{10})

	1	0	
	1	1	x
	1	0	
1	0		
1	1	0	

= 6_{10}

Note this is the same as decimal multiplication where we multiply by each of the digits and then add these results together.

Binary Division

The rules for division of two bits are as follows (recall that division by zero is invalid)

- 0 / 0 invalid
- 0 / 1 = 0
- 1 / 0 invalid
- 1 / 1 =1

Division example

Divide 1 1011 (Dividend) by 00111 (Divisor)

Using long division -

```
Divide 11011        by      111
                0  0  0  1  1
       111 |    1  1  0  1  1  Bring down
Subtract           1  1  1  |  the 1
                   1  1  0  1
Subtract              1  1  1
                      1  1  0  <-- Remainder (since it is
                                   too small to be divided by 111)
Check      by converting to base 10   27/7    = 3 with remainder 6
Dividend      27
Divisor        7
Quotient       3
Remainder      6
```

Shift/ Rotate instructions to perform multiply and divide operations

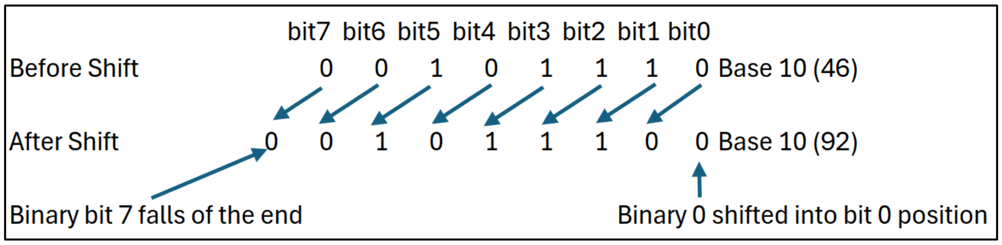
Consider an eight-bit byte 00101110 which has the decimal equivalent of 46. Next take each bit of the byte and shift them over one place to the left, filling in the now vacant bit 0 with the padded value 0 as shown below. Bit 7 has nowhere to go since it has no bit 8 position to occupy. The newly vacated bit 0 position is filled with a binary zero.

By shifting all the bits to the left the original number has been *multiplied by two* since the bit 0 value of 2^0 has been moved to the 2^1 position, bit 1's value of 2^1 has been moved to 2^2, etc.

Note that if the original bit 7 had a value of 1 then it would have been lost giving an incorrect result. This is a condition that *must* be checked for by the programmer and this will be covered in a later section.

Division by two is accomplished by shifting the bit values to the right.

Figure 1-3 Using shift operations to multiply and divide by two


bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0
Before Shift
0 0 1 0 1 1 1 0 Base 10 (46)
After Shift
0 0 1 0 1 1 1 0 0 Base 10 (92)
Binary bit 7 falls of the end
Binary 0 shifted into bit 0 position

bit 0 → bit 1→ bit 2 → bit 3 → bit 4 → bit 5 → bit 6 → bit 7 → bit 0, . . .

For simplicity the registers shown are byte-wide. In reality the width is more often 32 or 64 bits.

Other rotates are possible where the shifted-out bit feeds back to the input, giving a circular action.

Other types of shifting involves wrap around rotates where the pattern is:

Bit0 →Bit1 →Bit2 →Bit3 →Bit4 →Bit5 →Bit6 →Bit7 →Bit0 →Bit1…

Binary Coded Decimal (BCD)

Binary Coded Decimal represents decimal numbers in groups of bits, the encoding is normally done in four-bit nibbles. Each bit represents a power of two weight (2^3, 2^2, 2^1, 2^0, or 8,4,2,1). Since four bits can represent 16 distinct numbers, and there are only ten decimal digits, wastage occurs with this method. An alternative known as *packed BCD* may be used but is less common.

Converting Binary Coded Decimal to Decimal

BCD is similar to hexadecimal except that hex characters a through are illegal. A binary grouping of BCD characters could look like:

1001 0111 1000. Each group of 4 bits (nibbles) are read off as follows –

- 1001 = 9
- 0111 = 7
- 1000 =8

This corresponds to the decimal number 978.

BCD addition

Adding is straightforward, however if the addition of two nibbles results in a value greater than 9 (1010, 1011, 1100,1101, 1110, 1111) then it is an invalid decimal number. The resolution is to add 6 (0110) which will bring it back to a valid number. The carry will be added to the next nibble.

Addition examples –

1.
14 + 22 = 36 = 0011 0110
Verify by binary addition
0001 0100 (14)
0010 0010 (22) +
0011 0110 (36)

2.
20 +20 = 40 = 0100 0000
0010 0000 (20)
0010 0000 (20) +
0100 0000 (40)

3.
26+25 = 51 = 0101 0001
0010 0110 (26)
0010 0101 (25)+
0100 1011 Least significant nibble is greater than 9 so add 6
0000 0110 + (6)
01010001 (51)

4.
121 + 157 = 278 = 0010 0111 1000
0001 0010 0001 (121)
0001 0101 0111 (157)+
0010 0111 1000 (278)

5.
199 + 933 = 1132 = 0001 0001 0011 0010
0001 1001 1001(199)
1001 0011 0011 (933)+
1010 1100 1100 (Two nibbles invalid add 0110 0110
0000 0110 0110 +
1011 0011 0010 Now, the most significant nibble is invalid so add 6 to it
0110 0000 0000 +
0001 0001 0011 0010 (1132) Brings in a fourth nibble!

Conversion from Hex/Pure Binary to BCD

One way of converting a hex number to BCD is to convert the hex number to decimal and then to BCD. An alternative is to use the double-dabble method.

Double-Dabble

The double-dabble algorithm is fairly simple to implement, it consists of a series of shift[6] operations and additions. Note that an *n* digit hex number can translate into more than n decimal digits, (85_{16} = 133_{10}, FFF_{16} = 4095_{10}). The method sets up a register to hold n binary

[6] Shift/Rotate operations are discussed on page 1-13.

digits and partitions to hold the decimal powers of two – units, tens, hundreds, thousands, … The partitions are cleared to hold all zeros and then the binary digits are shifted in one bit at a time, adjustments (addition of decimal 3) are made to the partition values dependent on their magnitude (>4). Once all bits have been shifted[7] the algorithm has completed.

An example follows:

Consider the binary number 00011011 = hex 1B = decimal 27. The steps to convert from pure binary to BCD are shown in Table 1-7.

Table 1-7 Double-Dabble example

Hundreds Partition	Tens Partition	Units Partition	Binary Register	Action
0000	0000	0000	00011011	
0000	0000	0000	00110110	Shift left-most bit over to partitions (shift1)
0000	0000	0000	01101100	Shift left-most bit over to partitions (shift2)
0000	0000	0000	11011000	Shift left-most bit over to partitions (shift3)
0000	0000	0001	10110000	Shift left-most bit over to partitions (shift4)
0000	0000	0011	01100000	Shift left-most bit over to partitions (shift5)
0000	0000	0110	11000000	Shift left-most bit over to partitions (shift6)
0000	0000	1001	11000000	Add 3 to units, since unit is 5 or greater
0000	0001	0011	10000000	Shift left-most bit over to partitions (shift7)
0000	0010	0111	00000000	Shift left-most bit over to partitions (shift8)

Reading off the tens and unit columns gives the value 27_{10}.

Note 3 is added rather than 6 since the shift left operation multiplies by two!

A more complex 12-bit example is shown in Table 1-8.

[7] The number of shifts is equal to the number of binary digits

Table 1-8 Three digit double dabble example

Double Dabble Three digit Hex (200) number

12 binary digits so 12 shifts are required

Hundreds	Tens	Units	Binary			
0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0	Initial State
0 0 0 0	0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	Shift #1
0 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	Shift #2
0 0 0 0	0 0 0 0	0 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	Shift #3
0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #4
0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #5
0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #6
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 0	0 0 0 0	1 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 1	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #7
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 0	0 0 0 1	1 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 1 1	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #8
0 0 0 0	0 1 1 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #9
0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to tens
0 0 0 0	1 0 0 1	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 1	0 0 1 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #10
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 0 1	0 0 1 0	1 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 1 0	1 1 0 1	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #11
0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to Tens
0 0 1 0	1 0 0 0	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0	
0 0 0 0	0 0 0 0	0 0 1 1	0 0 0 0	0 0 0 0	0 0 0 0	Add 3 to units
0 0 1 0	1 0 0 0	1 0 0 1	0 0 0 0	0 0 0 0	0 0 0 0	
0 1 0 1	0 0 0 1	0 0 1 0	0 0 0 0	0 0 0 0	0 0 0 0	Shift #12

5	1	2	200 hex = 001000000000 binary = 512 decimal

Floating Point

An integer is a whole, complete and exact number such as 107 or 456. There is a limit to magnitude within a simple unit of storage such as a register. With floating -point representation a range of extremely large or extremely small numbers can be represented at the expense of precision. This means that a floating-point number may be an approximation that introduces *rounding* to nearest digits. There are two main parts to a floating-point number, the *significand* or *mantissa* and the *exponent*. There is also provision for a sign bit. The form is *significand* multiplied by the *base* raised to a *power*, an example being 3,450,000 = 345 X 10^4. Here 345 is the significand, ten is the base and four is the exponent.

There is a standard *IEEE 754* (https://standards.ieee.org/ieee/754/6210/) which is a specification for floating-point arithmetic. The standard defines Single and Double floating-point formats[8]as shown in Table 1-9. There is also provision to include Not-a-Number[9] (NaNs) and ±Infinity.

A 32-bit single precision floating-point binary number within IEEE 754 is defined as:

Sign Bit (1 bit)	Exponent (8 bits)	Significand (23 bits)

A 64-bit double precision floating-point binary number within IEEE 754 is defined as:

Sign Bit (1 bit)	Exponent (11 bits)	Significand (52 bits)

This is summarized in Table 1-9.

Table 1-9 Floating-Point formats

Format	Bits	Significand	Unbiased Exponent	Decimal Precision
Single	32	24 [10] (23+1)	8	6-9 digits
Double	64	53 (52+1)	11	15-17 digits

Biased exponents

The use of a *biased* exponent can represent negative exponents. For single precision the values range from decimal +127 to -126. The bias is normally given as 2^{n-1}-1 where n is the number of exponent bits, so here we have 2^7-1= 127. The value of the biased exponent is the unbiased exponent minus 127, so that an exponent of 10011011 gives a biased exponent of (128+16+8+2+1) – 127 = 155-127 = 28.

See Table 1-10 and Figure 1-4 for more on bias.

Infinity and Not-a-number representation

A biased exponent of all ones and a significand of all zeros (-127) represents infinity. The sign bit differentiates between negative and positive infinity.

[8] Other formats are defined but they will not be discussed here.

[9] This could arise from operations such as divide by zero or the square root of a negative number.

[10] There is an implied bit, since the normalized format is always 1.X then there is no need to specify the "1" value to the left of the decimal point.

Not-a-number is represented by the biased exponent being equal to all ones (+128)and the significand being non-zero. The sign bit is don’t care.

Table 1-10 BIAS within single precision IEEE 754

Exponent field		
Binary	Decimal	Exponent
00000001	1	2^{-126}
	…	…
01111011	123	2^{-4}
01111100	124	2^{-3}
01111101	125	2^{-2}
01111110	126	2^{-1}
01111111	127	2^{0}
10000000	128	2^{1}
10000001	129	2^{2}
100000010	130	2^{3}
100000011	131	2^{4}
	…	…

Bias set to mid way point

10000011

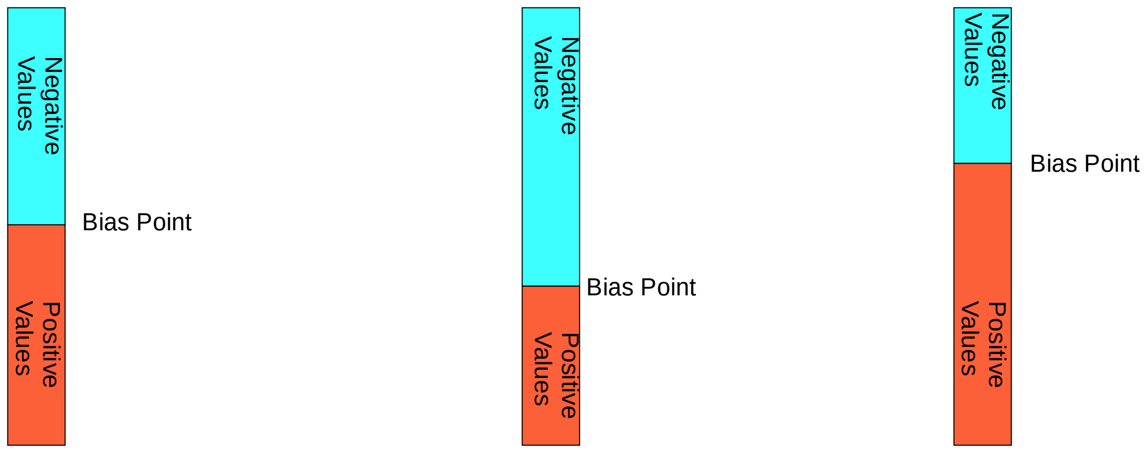
10000001

b= $2^{n-1}-1$ =127 where number of bits is 8

Understanding bias

The diagram shown in Figure 1-4 shows how varying the bias affects the ratio of negative to positive numbers. The bias is chosen in the standard to give similar *ranges* of positive and negative exponents.

Figure 1-4 Interpretation of Bias with floating point



With double precision numbers the bias is 1023 since the unbiased component shown in Table 1-9 is 11-bits wide.

Normalized

A normalized number has the form 1.XXXXX... The steps are to convert the number to binary and then perform shifts to give the desired result. Normalization shifts to the left or right depending on where the decimal point is

Example 410.625

Steps -

1. Convert to binary (See page 1-8, if needed. for a refresher on converting decimal fractions)

= 110011010.101

2. Perform repeated shift until desired pattern us reached.

110011010.101 x 2 (shift right operation)

= 11001101.0101 x2

= 1100110.10101 x2

= 110011.010101 x2

= 11001.1010101 x2

= 1100.11010101 x2

= 110.011010101 x2

= 11.0011010101 x2

=1.10011010101

This took a total of 8 shift operations. Add this number to 127 to get 135. Convert to binary to get:

10000111.

From our shifts earlier we had the value 10011010101, extend this to 23 bits to get 10011010101000000000000 giving the value:

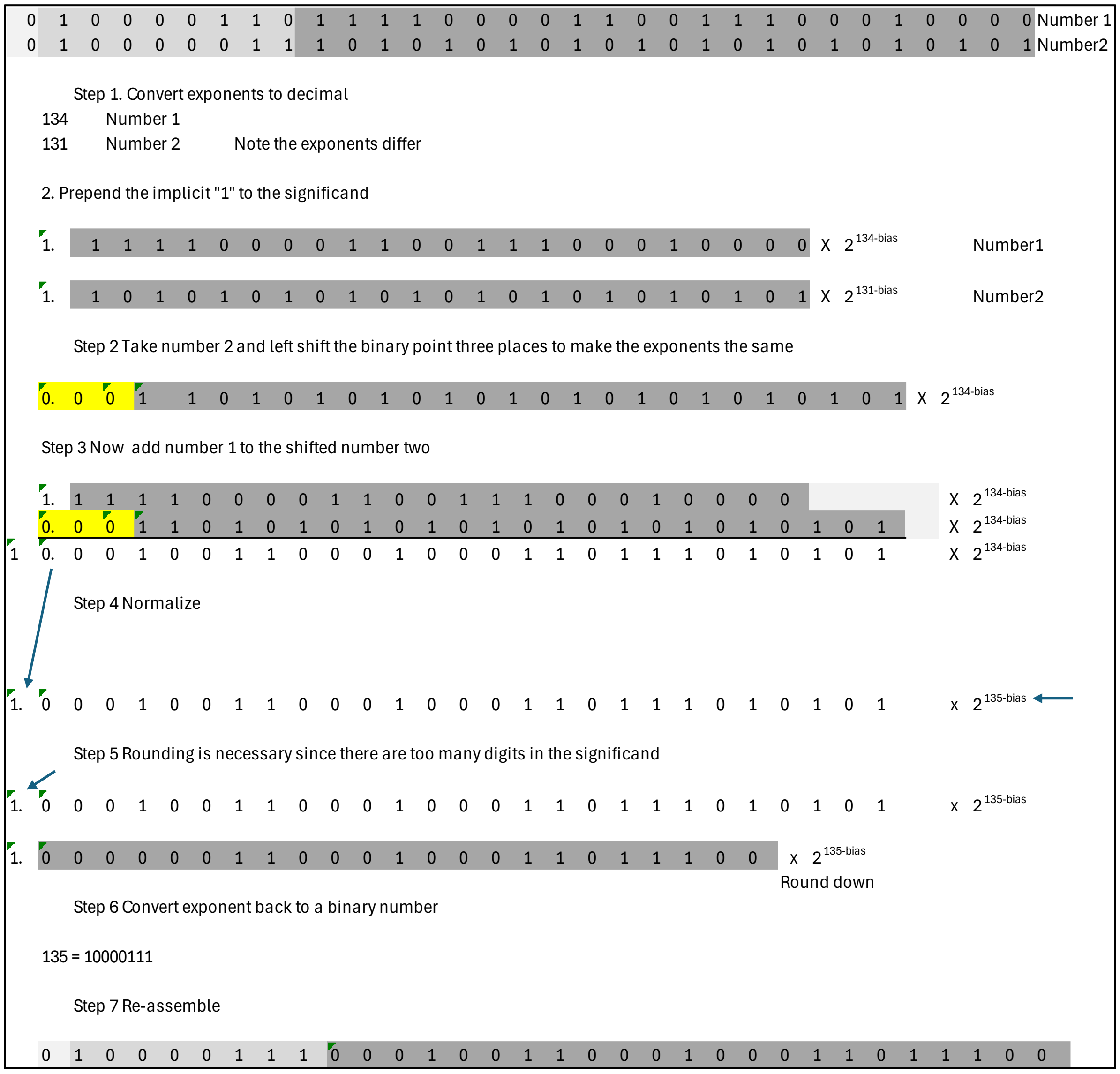
S	Exponent							Significand																							
0	1	0	0	0	0	1	1	1	1	0	0	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

=410.625

Addition of floating-point numbers

Addition is reasonably straightforward; the main concern is when the exponent differs. To equalize the exponents, take the lower number and shift over the binary point the required amount of positions. So, if one exponent is 136-*Bias* and the second is 134-*Bias*, the second number needs to be shifted two places to the left.

Figure 1-5 Addition of two floating point numbers



```
0 1 0 0 0 0 1 1 0 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 Number 1
0 1 0 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 Number2

    Step 1. Convert exponents to decimal
134     Number 1
131     Number 2        Note the exponents differ

2. Prepend the implicit "1" to the significand

1.  1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0  X 2^134-bias     Number1

1.  1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1  X 2^131-bias     Number2

    Step 2 Take number 2 and left shift the binary point three places to make the exponents the same

0. 0 0 1  1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1  X 2^134-bias

Step 3 Now  add number 1 to the shifted number two

  1. 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0          X 2^134-bias
  0. 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1    X 2^134-bias
1 0. 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 1    X 2^134-bias

    Step 4 Normalize

1. 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 1    x 2^135-bias

    Step 5 Rounding is necessary since there are too many digits in the significand

1. 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 1    x 2^135-bias

1. 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0  x 2^135-bias
                                                  Round down

    Step 6 Convert exponent back to a binary number

135 = 10000111

    Step 7 Re-assemble

0 1 0 0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0
```

Logic operations – and, OR, Exclusive OR, NOT

Logic operations are often used in decision making for example –

1. “If I feel hungry AND I have enough money, then I will order food in”.
2. “If it is cold OR it is raining, then I will wear a coat to go outside”.
3. “I can get a car discount if I pay the total amount in cash OR a I can get a lower interest rate if I take out a loan”.

Statement 1 is an AND condition and the decision to order food holds true if I am hungry AND I have enough money. Both conditions must be true.

Statement 2 is an OR condition and it states that I will wear a coat if either of these (or both) conditions are true.

Statement 3 is like statement 2 except that it is an either-or situation. Statement 2 applies equally well to both conditions in that it could be cold and also raining, and it would be similar to the AND condition. Statement 3 *exclusively* applies to the OR situation and is referred to as Exclusive OR *(XOR*).

These conditions are normally represented by *Truth Tables* such as if condition A is true AND condition B is true then result C is true. *True* and *false* values can be conveniently mapped to the binary values 1 and 0. These are known as *Boolean* variables.

Table 1-11 Truth table - AND

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	False (0)
False (0)	True (1)	False (0)
True (1)	True (1)	True (1)

Table 1-12 Truth table - OR

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
True (1)	True (1)	True (1)

Table 1-13 Truth table - XOR

A	B	C
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
True (1)	True (1)	False (0)

Other logic functions exist such as NOT which inverts the value, so a binary zero becomes a binary one. Repeating the operation, of course gets back to the original value. Boolean algebra is a complex topic by itself – which is dealt with in set theory.

For fun - a *simple* encoding can be done with XOR – take the word "Plaintext", converting this to seven-bit ASCII code becomes –

Table 1-14 Simple example of encoding text using XOR

Text string	ASCII code (decimal)	ASCII code (binary)	Apply XOR function with 10101010	Resultant ASCII code letter
P	80	1010000	1111010	z
l	108	1101100	1000110	.
a	97	1100001	1001011	K
i	105	1101001	1000011	C
n	110	1101110	1000100	D
t	116	1110100	1011110	^
e	101	1100101	1001111	O
x	120	1111000	1010010	4
t	116	1110100	1011110	^

So, the encoded string "Plaintext" becomes "z.KCD^O4^".

Of course, this is easily cracked and decoded!

The following rules show the resulting bitwise values:

- X AND 0 = 0
- X AND 1 = X
- X OR 0 = X
- X OR 1 = 1

Now that the foundation is in place it is time to move from generic concepts to programming on a specific architecture!

Summary of chapter 1

- Introduction to Assembly language
- Number Systems
- Shift Operations
- Logic and Truth tables

Exercises for chapter1

1. Divide 10111101 by 111 using manual long division
2. Convert 11.110 to base 10
3. Covert 0x1fd to BCD
4. Convert 35.65 to single precision floating-point according to IEEE 754
5. Write pseudo code to convert lower case ASCII characters a-z to upper case ASCII character A_Z.
6. Convert the signed binary byte to base10
7. Convert the octal number 341 to base 16
8. What are mnemonics?
9. Describe the advantages of a high-level language over assembly language
10. Describe the advantages of assembly language over higher level languages.

Chapter 2. Getting Started

This chapter is aimed at gaining familiarity with the ARM64 assembly language architecture. Subsequent chapters will concentrate on low level details and focus on topics in a more structured manner. The code snippets are short to allow for an easier grasp of the concepts presented.

Origin of ARM

In the early 1980's IBM introduced the IBM personal computer. Realizing that personal computing, would soon spread to the masses, the *British Broadcasting Corporation* (BBC) in the United Kingdom commissioned a company called *Acorn computers* to build a microcomputer for their TV series aimed at promoting computer literacy. This system was referred to as the BBC microcomputer.

Many UK schools adopted the computer part of this computer literacy thrust. The BBC Micro used a 6502 microprocessor and featured BBC Basic as its default programming language. Acorn then decided to embark on their own design, initially known as the Acorn RISC Machine. ARM (Advanced RISC Machines) was formed in late 1990.

The design used a Reduced Instruction Set Computer (RISC) design which differed from the Complex Instruction Set (CISC) design of other leading microprocessors such as the Z-80 from Zilog, the 6800 from Motorola and the 8080 from Intel®[11]. RISC has the advantage of a simpler design with lower power consumption making it ideal for use in embedded systems. Success came with the 32-bit design used in Apple and Android phones. The 64-bit ARM (ARM64) was announced in late 2011 and is the focus of this book.

Figure 2-1 BBC Micro

[11] *Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries*

Currently there are three ARM architecture profiles –

- A-Profile for applications,
- R-Profile for Real-Time applications
- M-Profile for small deployments where power consumption is a primary concern, examples are microcontrollers.

The ARM business model is to license their intellectual property to other manufacturers.

Choosing a candidate platform

The examples shown here will run quite happily on Raspberry Pi systems. Raspberry PI models 4 and 5 are recommended although the 64-bit Raspberry Pi 3 system can be used if needed. The recommended Operating System[12] is Raspberry Pi OS (64-bit) which includes the GNU tools that will be used.

Once the Pi[13] has been set up verify -

The command below shows that the architecture is indeed ARM64 (aarch64).

```
uname -a

Linux pi5a 6.1.0-rpi8-rpi-2712 #1 SMP PREEMPT Debian 1:6.1.73-1+rpt1 (2024-01-25)
aarch64 GNU/Linux

$ lscpu

Architecture:             aarch64

  CPU op-mode(s):         32-bit, 64-bit

  Byte Order:             Little Endian

CPU(s):                   4

  On-line CPU(s) list:    0-3

Vendor ID:                ARM

  Model name:             Cortex-A76

    Model:                1
```

[12] Refer to https://www.raspberrypi.com/software/operating-systems/ for a compatibility list of Raspberry Pi's that can run a 64-bit O/S.

[13] See https://www.raspberrypi.com/documentation/computers/getting-started.html

```
Thread(s) per core:  1
Core(s) per cluster: 4
Socket(s):           -
Cluster(s):          1
Stepping:            r4p1
CPU(s) scaling MHz:  100%
CPU max MHz:         2400.0000
CPU min MHz:         1500.0000
BogoMIPS:            108.00
Flags:               fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp
                      asimdhp cpuid asimdrdm lrcpc dcpop asimddp
```

Architecture

From an assembly language programmer's perspective, the architecture refers to the make-up of the system. It includes higher level areas such as memory addressing, CPU behavior, register layout, and the instruction set. A lower level is the *micro-architecture* which discusses how the instructions are executed and the interconnections (the *datapath*) through which the data traverses.

ARM64 Registers

Registers are locations that store values that are similar to variables in high-level languages. The primary way of interfacing with the ARM64 system is via the register set. Generically they may be referred to as *Rd* (destination register), *Rn* (first source register), *Rm* (second source register).

ARM64 provides 31 general purpose registers 0 through 30. The registers can be used as 32- bit or 64-bit. If a register is addressed with an "x" prefix then it functions as 64-bit using bits 63 through bit 0, if it is addressed with a "w" prefix then it is designated as a 32-bit register using bits 31 through bit 0. The registers can be designated as *wn* or *xn* for any of the w and x registers or more specifically as x4 for the fifth 64-bit general purpose register. Again, a more generic reference is *rn* which does not specify whether the 32-bit or 64-bit register is used.

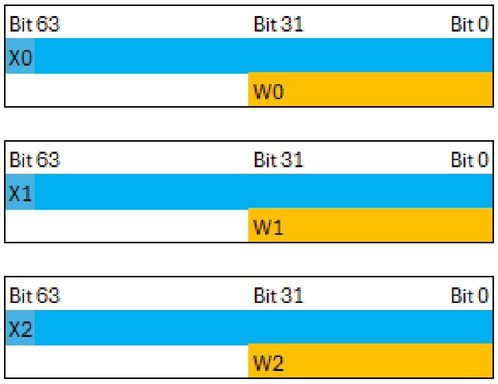
The 32-bit w register forms the lower half of the corresponding 64-bit x register. That is, w0 maps onto the lower word of x0, and w1 maps onto the lower word of x1.

When reading from a 32-bit w register the higher 32 bits of the x register are ignored. A write operation however, to a 32-bit w register will set the higher 32 bits of the x register to zero.

Register x30 is known as the *link register* (LR) and holds the return address of a function so it should be used with care. The 64-bit XZR and 32-bit WZR registers will return zero when read. Write operations will not change the value.

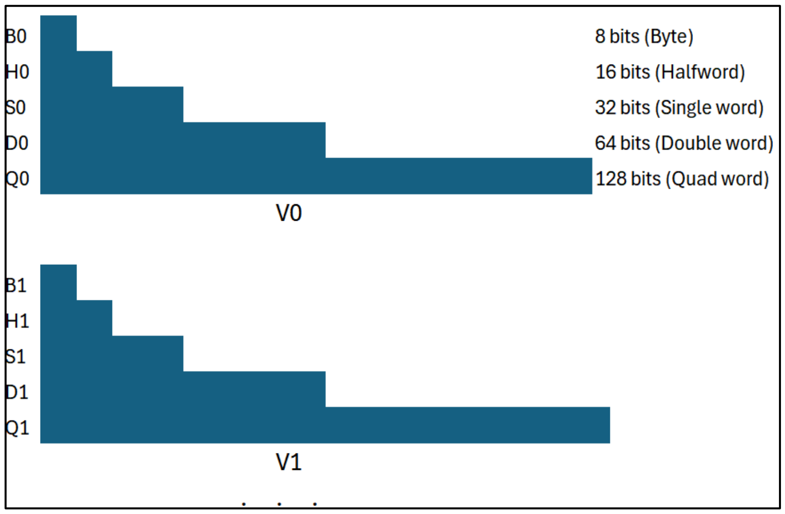
The *program counter* (PC) keeps track of program execution and is not used as a general-purpose register. Not all registers are programmer accessible.

Table 2-1 Register width.



. . .

Figure 2-2 Floating Point and Vector Registers



There are 32 additional registers used for *floating point* and *vector* operations. These registers have a width of 128 bits, but can be addressed with 8, 16, 32, 64 or 128 bits. Like the w and x general purpose registers a prefix is also used to determine the width. The smallest value of 8 bits is *Bx* up to *Qx* which has a width of 128 bits. These vector registers can operate on multiple data streams in parallel and are discussed in chapter 8.

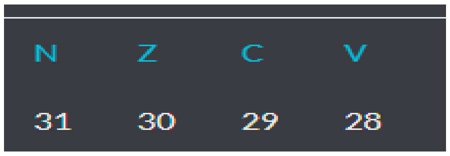
PSTATE and Exception levels

ARM64 defines four *exception levels* – EL0 through EL3. Not all of these levels may be implemented; so a system might only implement ELO and EL1. These exception levels are privilege levels with the highest EL number corresponding to the highest privilege level. User code typically runs at EL0 and kernel code runs at EL1. If EL2 and EL3 are implemented, they *typically* are used for Hypervisor and lower-level firmware functions.

The Processor state (PSTATE) shows the current state of the processor. The PSTATE includes *flags* that convey event information. These flags are single bit Boolean variables conveying True or False conditions.

These flags are:

• Negative (N)	True when signed number is negative, false if positive.
• Zero (Z)	True if result such as comparison of values are equal, false if not equal.
• Carry (C)	True If carry or no borrow condition occurs, shifted out bit
• Overflow (V)	True if and overflow condition occurs.



The flags are held in a special purpose register *Saved Program Status Register* (SPSR) .

These are known as *condition* flags and occupy bit positions 31 through 28.

Other fields are used for *exception masking* (***DAIF***) and are:-

- Debug (**D**) Enable/Disable debug exceptions.
- Asynchronous (**A**) Enable/Disable external asynchronous events (interrupts).
- IRQ (*I*) Enable/Disable interrupt requests.
- FIQ *(F*) Enable/Disable fast interrupt requests. FIQ takes priority over IRQ!

To summarize:-

Table 2-2 ARM64 Flags

Name	Description
N	Negative condition flag.
Z.	Zero condition flag
C	Carry condition flag.
V	oVerflow condition flag.
D	Debug mask bit.

A	SError mask bit.
I	IRQ mask bit.
F	FIQ mask bit.

In AArch64, the `ERET` instruction is used to return from an exception. The PSTATE. Flags N, Z, C, V are accessible at Exception Level 0. Accessing the other PSTATE fields requires exception levels higher than EL0.

For more information and bit field definitions, a good starting point is Arm Armv8-A Architecture Registers[14], specifically looking at Saved Program Status Register sections.

A Slight change of notation!

A programming note – From now on in this document the number's base will no longer have a subscript to differentiate them. Programmers use the more convenient shorthand 0b for binary and 0x for hexadecimal so the byte 00110100 is written as 0b0110100, hexadecimal numbers are written with the prefix 0x such as 0xF3AD and decimal numbers are devoid of a prefix. In addition, the abbreviation "hex" will be used for base 16 rather than the more cumbersome term "hexadecimal".

This is shown below:

Table 2-3 preferred number base notation

Binary	Decimal	Hexadecimal
0b00101111	47	0x2F

Assembling and Linking

Prior to looking at the instruction set in depth, it is beneficial to create some program snippets and then analyze the results. The code following does very little except for some register manipulation, nevertheless it will provide a good introduction for technical discussion and understanding. The ARM64 architecture uses 64 bits for the *memory address* and *instructions* are 32 bits in length. Data is processed within the registers rather than memory directly. This means that data must be *loaded* from memory into the registers and *stored* in memory from the registers forming a *Load and Store* architecture.

[14] https://developer.arm.com/documentation/ddi0595/2020-12/AArch64-Registers?lang=en

mov Instruction

Using the editor of your choice create and edit the file `moveregisters.s` with the following lines –

Listing 2-1 Using the mov instruction

```
.global       _start
_start:
.text
       mov x3, 0XFFFF
       mov x4, x3
       mov w8, 93    //ARM64 Syscall to exit
       svc #0
```

The first line includes an assembler *directive* (*.global*) using a label *_start* which defines the program's entry point and is declared as `.global` allowing external access to other files. Only one global `_start label` should appear when multiple files are involved. Instructions starting with "." are *directives* that communicate with the assembler program. The next directive `.text` introduces the actual code.

The first instruction (`mov`)places the value FFFF (hex) into the 64-bit register x3. This is a 16-bit value and is the largest number that can be placed into the register at any one time.

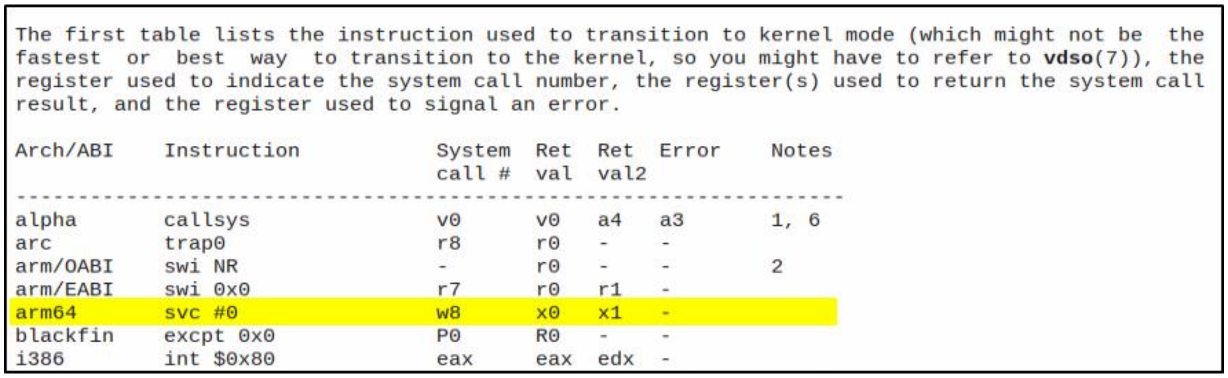
The second instruction takes the contents of the x3 register and copies it to the 64-bit register x4. After this has been executed, register x3 and x4 will have identical contents.

The third instruction invokes the *exit* system call. System Calls (*syscalls*) are dependent on the underlying architecture/operating system, and they specify how and where the *call/return* values are to be configured. Table 2-4 below shows an extract from Linux[15]. These are *privileged* instructions. User-mode programs interact with system resources via an Application Programming Interface (API). User-mode applications typically run in *Exception Level zero (EL0*) and this is the lowest level of privilege. The application calls the Operating System to perform the task on its behalf. These applications interact with the operating system's kernel resources by running under a higher level of privilege - *Exception Level one (EL1)*.

[15] Invoke with `man syscall.`

The ARM64 architecture passes the system call via register w8.

Table 2-4 Registers for system calls and return values



```
The first table lists the instruction used to transition to kernel mode (which might not be  the
fastest  or  best  way  to transition to the kernel, so you might have to refer to vdso(7)), the
register used to indicate the system call number, the register(s) used to return the system call
result, and the register used to signal an error.

Arch/ABI    Instruction           System  Ret  Ret  Error    Notes
                                  call #  val  val2
-------------------------------------------------------------------
alpha       callsys               v0      v0   a4   a3       1, 6
arc         trap0                 r8      r0   -    -
arm/OABI    swi NR                -       r0   -    -        2
arm/EABI    swi 0x0               r7      r0   r1   -
arm64       svc #0                w8      x0   x1   -
blackfin    excpt 0x0             P0      R0   -    -
i386        int $0x80             eax     eax  edx  -
```

The fourth instruction is the supervisor call to trigger the system call.

The program is converted to object code by the command –

```
as -o moveregisters.o moveregisters.s
```

The meaning of the instruction is to *assemble* the source file (.s) to an object file(.o) which is the binary code.

The next step is to link and create the executable file –

```
ld -o moveregisters moveregisters.o
```

Here the object code `moveregister.s.o` is *linked* to create the executable file `moveregisters`.

Finally make the code executable with the command:

```
chmod 777 moveregisters
```

Run the code with –

```
./moveregisters
```

The program has completed, but did it really do what we asked it? To find out there is a debugging `(GDB)` tool which allows us to interactively display the registers and execute the code one step at a time.

Recreate the object code but this time add the -g switch (debug) as shown –

```
as -g -o  moveregisters.o moveregisters.s

ld -o  moveregisters moveregisters.o
```

Next invoke the debugger

```
$ gdb moveregisters

GNU gdb (Debian 13.1-3) 13.1

Copyright (C) 2023 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from moveregisters...
```

List the code (l)

```
(gdb) l
1       .global       _start
2       _start:
3       .text
4               mov x3, 0XFFFF
5               mov x4, x3
6               mov w8, 93   //ARM64 Syscall to exit
7               svc #0
```

Set a breakpoint to stop the program (b)

```
(gdb) b 1
Breakpoint 1 at 0x400078: file moveregisters.s, line 4.
```

Note line 4 is the first line of actual code.

Start the program (run)

```
(gdb) run
Starting program: /home/alan/asm/moveregisters
Breakpoint 1, _start () at moveregisters.s:4
4               mov x3, 0XFFFF
(gdb)
```

The program has stopped at our first line of code, show the register contents by `(i)nfo (r)egisters`.

```
(gdb) i r
x0             0x0                 0
x1             0x0                 0
x2             0x0                 0
x3             0x0                 0
x4             0x0                 0
x5             0x0                 0
x6             0x0                 0
. . .
x29            0x0                 0
x30            0x0                 0
sp             0x7fffffff140       0x7fffffff140
pc             0x400078            0x400078 <_start>
cpsr           0x1000              [ EL=0 BTYPE=0 SSBS ]
fpsr           0x0                 [ ]
fpcr           0x0                 [ Len=0 Stride=0 RMode=0 ]
tpidr          0x0                 0x0
tpidr2         0x0                 0x0
```

Hit (`s`)tep to step into the next line of code

```
(gdb) s
5            mov x4, x3
```

Show register 3 and 4 only

```
(gdb) i r x3
x3             0xffff              65535
(gdb) i r x4
x4             0x0                 0
```

We can see that x3 has the content `0xffff`, hit `s` to execute the next line of code and show registers 3 and 4 again.

```
(gdb) i r x3
x3             0xffff              65535
(gdb) i r x4
```

```
x4              0xffff              65535
```

Register x4 now has the content `ffff` and register x3 has been left untouched.

Show all the registers again –

```
(gdb) i r
x0              0x0                 0
x1              0x0                 0
x2              0x0                 0
x3              0xffff              65535
x4              0xffff              65535
x5              0x0                 0
x6              0x0                 0
x7              0x0                 0
x8              0x5d                93
x9              0x0                 0
x10             0x0                 0
x11             0x0                 0
.. .
x29             0x0                 0
x30             0x0                 0
sp              0x7ffffffff140      0x7ffffffff140
pc              0x400084            0x400084 <_start+12>
cpsr            0x201000            [ EL=0 BTYPE=0 SSBS SS ]
fpsr            0x0                 [ ]
fpcr            0x0                 [ Len=0 Stride=0 RMode=0 ]
tpidr           0x0                 0x0
tpidr2          0x0                 0x0
```

Aliases

With assembly code there are often multiple ways of accomplishing the same task, for example the CMP (Compare instruction) is an alias of the Sub (Subtract instruction).The CMP Compare (immediate) subtracts an immediate value from a register value. The SUB Subtract (immediate), subtracts an immediate value from a register value, and writes the result to the destination register. Rather than have the programmer work out the equivalency, the assembler will perform the substitution allowing the coder to continue using (perhaps) mnemonics that they are more used to. Again, with RISC architectures there is limited space for instructions.

Re-assemble the program again, without the -g option (to remove debug information).

```
as -o  moveregisters.o moveregisters.s
ld -o moveregisters moveregisters.o
```

Now run the `objdump` program with the -D(issasemble) option –

```
$ objdump -D moveregisters


moveregisters:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000400078 <_start>:
  400078:     d29fffe3     mov    x3, #0xffff                  // #65535
  40007c:     aa0303e4     mov    x4, x3
  400080:     52800ba8     mov    w8, #0x5d                    // #93
  400084:     d4000001     svc    #0x0
```

Re-run `objdump` again but this time use `-M no-aliases.`

```
$ objdump -D -M no-aliases moveregisters
moveregisters:     file format elf64-littleaarch64
Disassembly of section .text:
0000000000400078 <_start>:
  400078:     d29fffe3     movz   x3, #0xffff
  40007c:     aa0303e4     orr    x4, xzr, x3
  400080:     52800ba8     movz   w8, #0x5d
  400084:     d4000001     svc    #0x0
```

Typically,[16], instructions in ARM64 are of the form `– Instruction <Rd> <Rn>, 2nd operand`. Rd is the destination register followed by a source register and a possible second operand that can be a register or an immediate (literal value). The use of R indicates that the registers can be

[16] Not always, see `str` instructions!

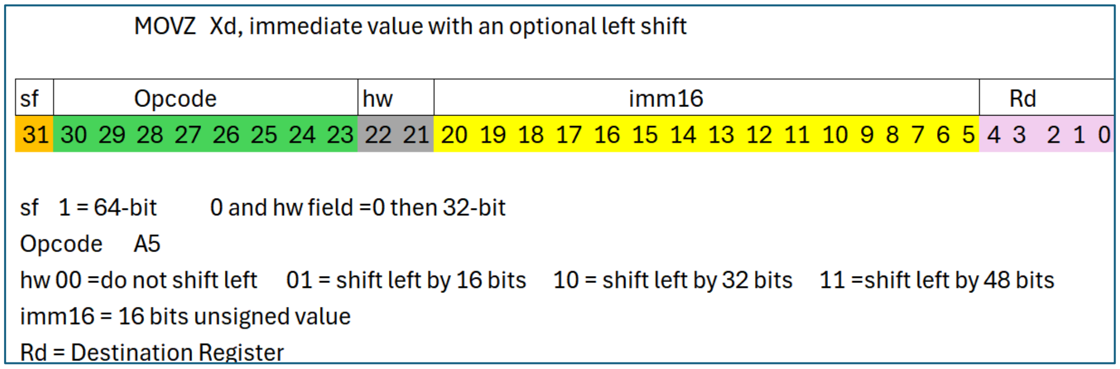
either X or W registers. Modification can be made to a source register such as performing a *shift* operation.

The output of the utility `objdump` as shown above has the following format –

Address in memory   Instruction (Hex)   ARM64 instruction

Notice that the memory location increments by four bytes (corresponding to the 32-bit wide ARM64 instruction) after each instruction is executed.

Figure 2-3 Format of MOVZ instruction



MOVZ Xd, immediate value with an optional left shift

sf	Opcode	hw	imm16	Rd
31	30 29 28 27 26 25 24 23	22 21	20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4 3 2 1 0

sf 1 = 64-bit 0 and hw field =0 then 32-bit
Opcode A5
hw 00 =do not shift left 01 = shift left by 16 bits 10 = shift left by 32 bits 11 =shift left by 48 bits
imm16 = 16 bits unsigned value
Rd = Destination Register

Referring to the ARM64 instruction set architecture documentation[17] the `MOVZ` instruction states that “This instruction is used by the alias mov (wide immediate).” So, there is no actual mov instruction as such, however it transparently accomplishes the action that is to be executed. The format of the MOVZ (Move wide with zero) instruction is :-

Breaking down the bits d29fffe3 (first line of `objdump` non-aliased code shown on page 2-12) gives a binary value of –

sf	opcode								hw		Immediate																Xd				
1	1	0	1	0	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1

- 64-bit
- Opcode is A5
- Immediate value is FFFF
- Register is x3

[17] See https://developer.arm.com/documentation/ddi0602/2023-12/Base-Instructions/

Instruction Aliases

Note that when the no-aliases option is used the disassembly process listed the code as an actual ARM64 instruction rather than pseudo-code. *Aliases* are mnemonics that are familiar to the programmer and the assembler will replace them with an actual ARM.instruction.

Moving 32-bit and 64-bit immediate values

Question - Since there are only 16 bits available for the immediate value, how would a register be loaded with the 32-bit value 0X12345678?

Response - The approach is to move the values in stages with the `movk` instruction. This instruction moves data 16 bits at a time and optionally puts the values into the register with a shifted offset value; this offset can be 0, 16, 32 or 48 bits as defined by the 2-bit hw field and leaves the other bits alone.

Our plan is to move in the first 16 bits with a default shift of zero, followed by another move of 16 bits but in the second quarter of the register.

Example –

Listing 2-2 Using the movk instruction

```
.global       _start

.text

_start:       movk x3, #1234, lsl #0

              movk x3, #5678, lsl #16

              mov  x8, #93

              svc  0
```

After execution of the code `movk x3, #5678, lsl #16` the content of x3 is:

```
x3             0x162e04d2          372114642

(gdb) i r

x0             0x0                 0

x1             0x0                 0

x2             0x0                 0

x3             0x162e04d2          372114642
```

Checking → 0x162e = 5678 and 0x04d2 = 1234.

`objdump` shows –

```
Disassembly of section .text:
```

```

0000000000400078 <_start>:
  400078:      f2809a43      movk   x3, #0x4d2
  40007c:      f2a2c5c3      movk   x3, #0x162e, lsl #16
  400080:      d2800ba8      mov    x8, #0x5d                         // #93
  400084:      d4000001      svc    #0x0
```

Looking at the second line `movk x3, #0x162e, lsl #16`

- 64-bit
- The format of `movk` (Move wide with keep) instruction is
- Opcode is E5
- Immediate value is 162e
- Register is x3

Figure 2-4 Format of movk instruction

MOVK with keep moves immediate value to xd with an optional left shift

sf	Opcode								hw		imm16																Rd				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

sf 1 = 64-bit 0 and hw field =0 then 32-bit
Opcode E5
hw 00 =do not shift left 01 = shift left by 16 bits 10 = shift left by 32 bits 11 =shift left by 48 bits
imm16 = 16 bits unsigned value
Rd = Destination Register

The next example shows the `MOVN` instruction – The listing is included in GDB's output. Comments can be placed on the same line by appending "//" after the instruction (as shown below) or block style starting with "/*" and ending with "*/".

MOVE Negated instruction (MOVN)

```
(gdb) list
```

Listing 2-3 Using the MOVN instruction

```
1       .global _start
2       _start:
3               MOVN   x2, #0xfedc   //This is the move negated instruction
4               NOP                  //after execution it will change the value above
to
```

```
5             NOP                 //0xffffffffffff0123
6             NOP                 //can be useful with bitmask operations
7             mov    w8, #0x5d    //Time to go.
8             svc #0
(gdb) b _start
Breakpoint 1 at 0x400078: file moven.s, line 3.
(gdb) b 7
Breakpoint 2 at 0x400088: file moven.s, line 7.
(gdb) run
Starting program: /home/alan/asm/moven


Breakpoint 1, _start () at moven.s:3
3             MOVN   x2, #0xfedc  //This is the move negated instruction
(gdb) continue
Continuing.
Breakpoint 2, _start () at moven.s:7
7             mov    w8, #0x5d    //Time to go.
(gdb) i r x2
x2              0xffffffffffff0123  -65245
```

Effectively, this has produced the one's complement of our number.

Displaying output

Listing 2-4 Displaying output with the Write syscall

```
// listing2-4
/* This example shows how to write a string to the screen. It uses the write
system call for this. The call expects three arguments -
- x0 holds the file descriptor (1=stdout),
- x1 holds the starting address in memory of the string to be written
- x2 holds the length of the string */
.text
.global _start
_start:
       mov    x0, #1 //stdout
       ldr    x1, =result
```

```
      mov    x2, #6        //Print 6 characters
      mov    w8, #64       //This is the write system call
      svc    #0            //Put it out to screen
      mov    x0, #0        //Return code of 0
      mov    w8, #0x5d     //Time to go.
      svc    #0


.data
result: .ascii "Hello\n"
.align 4
```

This program uses the *write syscall (0x40)* to output a string of text to *stdout*. This works by loading register x0 with the value 1 corresponding to `stdout`. Register x1 points to the starting address in memory to where the string of is located and register x2 is loaded with the length of the string. After the string has been written, register x0 is loaded with a return code of 0 (success) and the exit service call is triggered.

The assembler directive `.data` defines the start of memory.

```
objdump -s -d -M no-aliases printhello
printhello:     file format elf64-littleaarch64
Contents of section .text:
 4000b0 200080d2 e1000058 c20080d2 08088052   ......X.......R
 4000c0 010000d4 000080d2 a80b8052 010000d4  ...........R....
 4000d0 e0004100 00000000                    ..A.....
Contents of section .data:
 4100e0 48656c6c 6f0a0000 00000000 00000000  Hello...........
Disassembly of section .text:
00000000004000b0 <_start>:
  4000b0:     d2800020      movz   x0, #0x1
  4000b4:     580000e1      ldr    x1, 4000d0 <_start+0x20>
  4000b8:     d28000c2      movz   x2, #0x6
  4000bc:     52800808      movz   w8, #0x40
  4000c0:     d4000001      svc    #0x0
  4000c4:     d2800000      movz   x0, #0x0
  4000c8:     52800ba8      movz   w8, #0x5d
  4000cc:     d4000001      svc    #0x0
  4000d0:     004100e0      .word  0x004100e0

  4000d4:     00000000      .word  0x00000000
```

The data section is located at memory address *0x4100e0* and the hex codes for the ASCII string *(Hello*) is highlighted above.

Make

The commands that have been used so far for assembling and linking (`as,ld`) have worked well enough for our situation, however when multiple files are involved it is normal to use a build tool to accomplish this. The *make* utility keeps track of what has been done and will only apply actions to the changed portions. The instructions are conveyed to the utility using a *makefile*. The `makefile` below can be used to assemble link the program `moveregister.s.`

Simple `makefile`

```
moveregisters: moveregisters.o

	ld -o moveregisters moveregisters.o

moveregisters.o: moveregisters.s

	as -o moveregisters.o moveregisters.s
```

The line at the top denotes the *target* file which *depends* on the *object* file which in turn is dependent on the *source* file. The rules on how to create the target file are shown above, so the flow is :-

Create the target file (`moveregister.s`) from the object file (`moveregister.o`) which is created from the source file (`moveregister.s`). The first target (here `moveregister`) is termed the *default goal*.

Note use Tab characters for indentation in the `makefile`.

The next example assembles and links two programs into a single executable file,

```
OBJECTS = program1.o program2.o

all: myprogram

%.o : %.s

	as $< -g -o $@

myprogram: $(OBJECTS)

	ld -o myprogram $(OBJECTS)
```

This example will allow the target to be passed to the `makefile`:-

```
TARGETFILE = $(targetfile)

print: $(TARGETFILE).o
```

```
	ld -o $(TARGETFILE) $(TARGETFILE).o
$(TARGETFILE).o: $(TARGETFILE).s
	as -o $(TARGETFILE).o $(TARGETFILE).s
```

```
$ make targetfile=print
make: 'print' is up to date.
$ ls
makefile  print  print.o  print.s
```

Make will be revisited in more depth later on!

Using strace

The `strace` utility can be used to monitor which syscalls have been invoked by a particular program or process:-

```
$ strace -c ./print
Hello again!
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           write
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000           0         2           total
```

With this particular program `strace` shows that the syscalls `write` and `exit` were invoked once.

Summary of chapter 2

- Register Set
- Assembling and linking
- Makefiles
- Aliases and pseudo code
- Debugging with GDB

Exercises for chapter 2

1. Install Raspberry PI OS on a 64-bit Raspberry PI system
2. What qualifier would you add to the as command to embed debug information?
3. What is the purpose of a linker?
4. How many w registers are available for general purpose use?
5. What are assembly directives?
6. Describe two ways of loading the value 0x1256 into the top 16 bits of register x3
7. What are syscalls?
8. What is the function of a makefile?
9. What are ARM assembly aliases?
10. What tool is used to disassemble an ARM executable program?
11. Describe two flags that the ARM instruction set uses to convey conditions.

Chapter 3. Dealing with memory

Chapter 3 delves further into the architecture and discusses memory topics , addressing modes with LDR and STR instructions. Graphical debuggers are introduced.

Load and Store instructions

ARM64 deals with register operations, to work with memory, addresses are *loaded* into registers, and *stored* back from registers to memory. Operations are with respect to memory so loading from memory to registers is a *read* operation and storing from registers is a *write* operation. The method by which memory addresses are derived is known as addressing modes and there are several. The code fragments in this chapter will show how to communicate with memory and will also introduce various addressing modes.

Load and store instructions can access memory. Data is loaded (ldr) from memory, acted on and then stored (str) back to memory. This is termed *load-store architecture*.

LOAD Instructions (Memory → Registers)

Examining memory with GDB

GDB can be used to examine memory. The format of the command is `x/nfu addr`. Here the parameters have the following meaning:

Table 3-1 Using GDB to display memory contents

`n`	**How much memory to display in units, with a default value of one.**
`f`	This is the display format; default is to display in hex. The main options are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string)
`u`	Unit size b = byte h = halfword (2 bytes) w = word (4 bytes) g = giant (8 bytes)

Example

```
(gdb) x/16w  0x4100e0
0x4100e0:     0x6c6c6548    0x00000a6f    0x00000000    0x00000000
0x4100f0:     0x0000002c    0x00000002    0x00080000    0x00000000
0x410100:     0x004000b0    0x00000000    0x00000028    0x00000000
0x410110:     0x00000000    0x00000000    0x00000000    0x00000000
```

This is the contents of memory after running the `printhello` program shown on page 2-12.

This shows the ASCII data highlighted in default hex values, strings can be shown more clearly by using the `x/s` command –

```
gdb) x/s  0x4100e0

0x4100e0:     "Hello\n"
```

Consider a similar program `(printhelloARM64.s)` that writes out a slightly longer string.

Listing 3-1 String printing

```
/* This example shows how to write a string to the screen. It uses the write system call

       for this. The call expects three arguments -


       - x0 holds the file descriptor (1=stdout),

       - x1 holds the starting address in memory of the string to be written

       - x2 holds the length of the string  */


       .text


       .global _start


       _start:

              mov    x0, #1 //stdout

              ldr    x1, =string1

              mov    x2, #13      //Print 13 characters

              mov    w8, #64      //This is the write system call

              svc    #0           //Put it out to screen

              mov    x0, #0 //Return code of 0

              mov    w8, #0x5d    //Time to go.

              svc    #0


       .data

       string1: .ascii "Hello ARM64!\n"

       .align 4
```

Using gdb shows :-

```
(gdb) list 1,30
1      /* This example shows how to write a string to the screen. It uses the
write system call
2      for this. The call expects three arguments -
3
4      - x0 holds the file descriptor (1=stdout),
5      - x1 holds the starting address in memory of the string to be written
6      - x2 holds the length of the string  */
7
8      .text
9
10     .global _start
11
12     _start:
13             mov    x0, #1 //stdout
14             ldr    x1, =string1
15             mov    x2, #13      //Print 13 characters
16             mov    w8, #64      //This is the write system call
17             svc    #0           //Put it out to screen
18             mov    x0, #0 //Return code of 0
19             mov    w8, #0x5d    //Time to go.
20             svc    #0
21
22     .data
23     string1: .ascii "Hello ARM64!\n"
24     .align 4
gdb) b 20
Breakpoint 1 at 0x4000cc: file helloARM64.s, line 20.
(gdb) run
Starting program: /home/alan/asm/helloARM
Hello ARM64!


Breakpoint 1, _start () at helloARM64.s:20
```

```
20            svc    #0
gdb) x/s 0x4100e0
0x4100e0:     "Hello ARM64!\n"
(gdb) x/16xb 0x4100e0
0x4100e0:     0x48   0x65   0x6c   0x6c   0x6f   0x20   0x41   0x52
0x4100e8:     0x4d   0x36   0x34   0x21   0x0a   0x00   0x00   0x00
```

Individual characters of the string can be shown by issuing `x/1qc <address>`. The memory layout is actually:-

0x4100e0:	72 'H'
0x4100e1:	101 'e'
0x4100e2:	108 'l'
0x4100e3:	108 'l'
0x4100e4:	111 'o'
0x4100e5:	32 ' '
0x4100e6:	65 'A'
0x4100e7:	82 'R'
0x4100e8:	77 'M'
0x4100e9:	54 '6'
0x4100ea:	52 '4'
0x4100eb:	33 '!'
0x4100ec	10 '\n'

The directive `.data` placed the starting character of the string is placed at the lowest memory location. This is termed *little-endian*[18] where the least significant byte is stored at the lowest address.

[18] This term originally comes from Jonathan Swift's novel Gulliver's travels and refers to which end a boiled egg is broken from.

The debugger shows us the bytes in increasing address order, starting from the left, (the same order as when reading a book published in English).

```
(gdb) x/16xc 0x4100e0
0x4100e0:     72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' ' 65 'A' 82 'R'
0x4100e8:     77 'M' 54 '6' 52 '4' 33 '!' 10 '\n' 0 '\000' 0 '\000' 0 '\000'
```

```
Disassembly produces :-
objdump -d -M no-aliases printhelloARM64


printhelloARM64:     file format elf64-littleaarch64
Disassembly of section .text:
00000000004000b0 <_start>:
  4000b0:     d2800020      movz   x0, #0x1
  4000b4:     580000e1      ldr    x1, 4000d0 <_start+0x20>
  4000b8:     d28001a2      movz   x2, #0xd
  4000bc:     52800808      movz   w8, #0x40
  4000c0:     d4000001      svc    #0x0
  4000c4:     d2800000      movz   x0, #0x0
  4000c8:     52800ba8      movz   w8, #0x5d
  4000cc:     d4000001      svc    #0x0
  4000d0:     004100d8      .word  0x004100d8  < Start of Data section
  4000d4:     00000000      .word  0x00000000
```

The first line puts the value of one into register x0

sf	Opcode							hw		imm16																Rd				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The second line loads register x1 with contents of the memory pointed to by the current instruction's location (as pointed to by the Program Counter (PC with an offset of 0x20 which is where the first part of our data resides. Note this is 8 instructions from the start label or 7 instructions away from the current instruction. Recall that each instruction takes 4 bytes, so the offset is 28 bytes or 0x1c bytes.

0x4000b0+0x20 = 0x4000d0

LDR Xt, label   *580000e1*   *ldr*   *x1, 4000d0 <_start+0x20>*

op		imm19	Rt
31 30	29 28 27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4 3 2 1 0
0 1	0 1 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1	0 0 0 0 1

op   01 = 64-bit

imm19   Multiple of 4 and is the offset (in words) from the current instruction so offset is 0x7 instructions further on

Rt = Register to be loaded = R1

The way that memory is addressed by the `ldr` instruction is termed *PC Relative* addressing, if no offset is given, then it defaults to an immediate value of 0x0.

The ldr instruction as we have used it puts the *address* of the string into register x1. The next program uses ldr to put the *contents* of the string into register x4. The instruction is:

ldr x4, [x1] as highlighted below:

```
Breakpoint 1, _start () at printhelloARM2.s:
                           mov    x0, #1 //stdout
(gdb) s                    ldr    x1, =string1 //This loads the address string1
into x1
(gdb) s                    ldr    x4, [x1] //This loads the actual data into x4
(gdb) s                    mov    x2, #13      //Print 13 characters
(gdb) i r
x0             0x1                 1
x1             0x4100e0            4260064
x2             0x0                 0
x3             0x0                 0
x4             0x5241206f6c6c6548  5927054247528785224
x5             0x0                 0
. . .
x30            0x0                 0
sp             0x7ffffffff0a0      0x7ffffffff0a0
pc             0x4000bc            0x4000bc <_start+12>
cpsr           0x201000            [ EL=0 BTYPE=0 SSBS SS ]
fpsr           0x0                 [ ]
fpcr           0x0                 [ Len=0 Stride=0 RMode=0 ]
tpidr          0x0                 0x0
tpidr2         0x0                 0x0
```

Decoding the contents of register x4 shows:-

Byte	ASCII
0x48	H
0x65	e
0x6c	l
0x6c	l
0x6f	0
0x20	<Space>
0x41	A
0x52	R

Since the register is 64-bits only eight characters of the string can be accommodated. Altering the line to add an offset of two (=string1 + 2) will cause the string to skip the first two characters (He) as shown below.

```
Breakpoint 1, _start () at printhelloARM3.s:12
              mov    x0, #1 //stdout
              ldr    x1, =string1 + 2 //This loads the address string1 into x1
              ldr    x4, [x1, #4] //This loads the actual data into x4
              mov    x2, #26      //Print 26 characters
              mov    w8, #64      //This is the write system call
              svc    #0     //Put it out to screen
llo ARM64!
Hello again!
18                   mov    x0, #0 //Return code of 0
```

```
.text
.global _start
       _start:
              mov    x0, #1 //stdout
              ldr    x1, =string1 //This loads the address string1 into x1
              ldr    x4, [x1, #4] //This loads the actual data into x4
              mov    x2, #26      //Print 26 characters
              mov    w8, #64      //This is the write system call
```

```
             svc    #0     //Put it out to screen
             mov    x0, #0 //Return code of 0
             mov    w8, #0x5d    //Time to go.
             svc    #0
.data
      string1: .ascii "Hello ARM64!\n"
      string2: .ascii "Hello again!\n"
$ objdump -d -M no-aliases printhelloARM2

printhelloARM2:     file format elf64-littleaarch64
```

```
Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:     d2800020      movz   x0, #0x1
  4000b4:     58000121      ldr    x1, 4000d8 <_start+0x28>
  4000b8:     f8404024      ldur   x4, [x1, #4]
  4000bc:     d2800342      movz   x2, #0x1a
  4000c0:     52800808      movz   w8, #0x40
  4000c4:     d4000001      svc    #0x0
  4000c8:     d2800000      movz   x0, #0x0
  4000cc:     52800ba8      movz   w8, #0x5d
  4000d0:     d4000001      svc    #0x0
  4000d4:     00000000      udf    #0
  4000d8:     004100e0      .word  0x004100e0
  4000dc:     00000000      .word  0x00000000
```

Note Rt is the *transfer* register and Rn is the *base* register.

To summarize :-

- Register x1 holds the address of the text by using *program counter relative addressing*.

- Register x4 holds the value of the memory address pointed to by register x1 using *register indirect with offset addressing*
- The square brackets are used to show *indirect* memory addressing
- Indirect addressing refers to loading the data stored at the address pointed to by the register.
    - This is similar to a book index which points the reader to the page where the content is stored.
- Note where there is an offset no-aliases gives the instruction `ldur x4, [x1, #4]`
- A summary of addressing modes is given in Table 3-3

The instruction LDUR is *load unscaled register*. In this case the 64-bit value from register x1 plus an offset of 4 is loaded into register x4.

```
                    mov    x0, #1 //stdout
(gdb) s
                    ldr    x1, =string1 //This loads the address string1 into x1
(gdb) s
                    ldr    x4, [x1, #4] //This loads the actual data into x4
(gdb) s
                    mov    x2, #26      //Print 26 characters
(gdb) i r
x0             0x1                 1
x1             0x4100e0            4260064
x2             0x0                 0
x3             0x0                 0
x4             0x2134364d5241206f  2392597007760957551
```

!46MRA o

Skipping “Hell”

The ldr instruction is actually `ldr {type}` where type is actually an unsigned byte (B), a signed byte (SB), unsigned halfword (H), signed halfword (SH).

Store Instructions (Registers → Memory)

We have already shown how to define memory contents using the .data directive on page 2-17. The format of the data can be specified in multiple ways, some examples include: -

```
.data
       msg:          .ascii “Hello ARM”
       randombytes:  .byte 52, 35, 46, 95, 0x42
       characters:   .byte ‘H’, ‘e’, ‘l’, ‘l’, ‘o’
       somewords:    .word 0x0123456789abcdef
       negnumbers:   .byte -0xaa,0xff
       blanks:       .space 8
```

The next program generates a string and loads it into memory, previously a string was defined using the .ascii directive. In this example 8 bytes of memory will be reserved using the .space directive. The default will be to zero out these bytes but they can be set to other values by using `.space <number_of_bytes> {,<fill_byte>}` for example `message1: .space 8, 0x55`

Listing 3-2 str example

```
// Listing3-2
/* This example shows how to write a string to the screen. It uses the write
system call
for this. The call expects three arguments -


- x0 holds the file descriptor (1=stdout),
- x1 holds the starting address in memory of the string to be written
- x2 holds the length of the string
A block of memory is reserved using the .data directive with the label message1.
It is initialized with 8 bytes of zero value
The string is loaded into register x4 2 bytes at a time via movk
and then stored into the memory location pointed to by x1
*/
.text


.global _start
       _start:
              mov    x0, #1 //stdout
              ldr    x1, =message1 //This loads the address of the label message1
into x1
              mov    w4, #0x6548         //Load first two bytes "He" just use w4
for this rather than x4
```

```
             movk   x4, #0x6C6c, lsl #16       //Load next two bytes "ll"

             movk   x4, #0x206f, lsl #32       //Next two bytes "o "

             movk   x4, #0x654d, lsl #48       //Next two "Me "

             str    x4, [x1]             // Put the eight byte string into memory
pointed to by register x1

             mov    x2, #8 //Print 8 characters

             mov    w8, #64       //This is the write system call

             svc    #0     //Put it out to screen

             mov    x0, #0 //Return code of 0

             mov    w8, #0x5d     //Time to go.

             svc    #0

.data

      message1: .space 8
```

Output -

```
$ ./store1


Hello Me
```

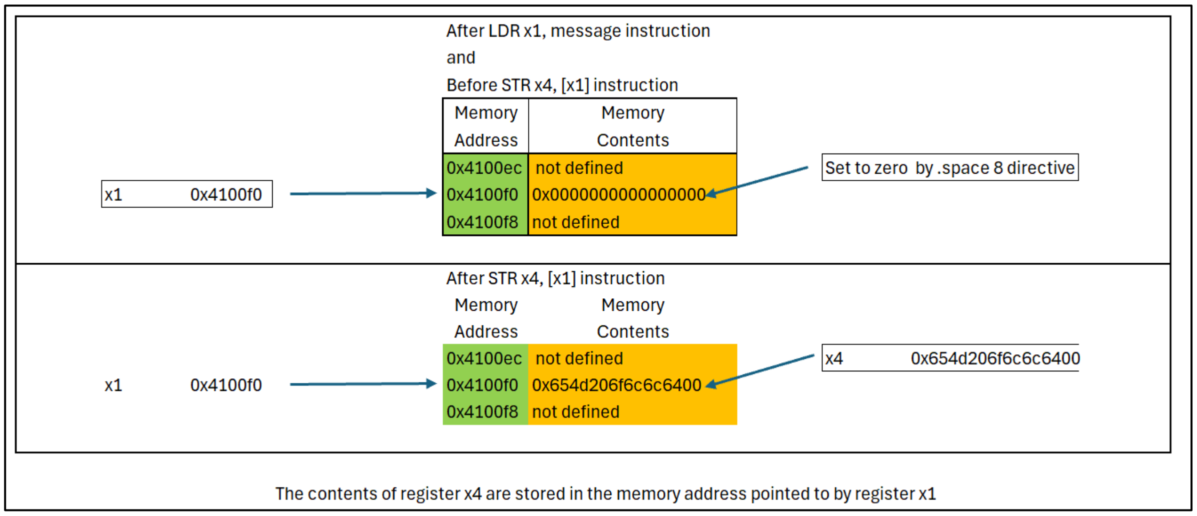
The output of gdb up until the `str` command has been executed shows:-

```
$ gdb store1

. . .

Type "apropos word" to search for commands related to "word"...

Reading symbols from store1...

(gdb) b 1

Breakpoint 1 at 0x4000b0: file store1.s, line 16.

(gdb) run

Starting program: /home/alan/asm/stores/store1


Breakpoint 1, _start () at store1.s:16

16                  mov    x0, #1 //stdout

17                  ldr    x1, =message1 //This loads the address of the label
message1 into x1

18                  mov    w4, #0x6548   //Load first two bytes "He" just use w4
for this rather than x4

19                  movk   x4, #0x6C6c, lsl #16 //Load next two bytes "ll"
```

```
20                  movk   x4, #0x206f, lsl #32 //Next two bytes "o "
21                  movk   x4, #0x654d, lsl #48 //Next two "Me "
22                  str    x4, [x1] // Put the eight byte string into memory
pointed to by register x1
23                  mov    x2, #8 //Print 8 characters
(gdb) i r
x0             0x1                 1
x1             0x4100f0            4260080
x2             0x0                 0
x3             0x0                 0
x4             0x654d206f6c6c6548  7299526233969943880
x5             0x0                 0
. . .
x30            0x0                 0
sp             0x7fffffff060       0x7fffffff060
pc             0x4000cc            0x4000cc <_start+28>
cpsr           0x201000            [ EL=0 BTYPE=0 SSBS SS ]
fpsr           0x0                 [ ]
fpcr           0x0                 [ Len=0 Stride=0 RMode=0 ]
tpidr          0x0                 0x0
tpidr2         0x0                 0x0
(gdb) x/s 0x4100f0
0x4100f0:     "Hello Me"
```

The figure below shows how the memory contents change after the str instruction has been executed.

Table 3-2 Action of str instruction to memory


After LDR x1, message instruction
and
Before STR x4, [x1] instruction

Memory Address	Memory Contents
0x4100ec	not defined
0x4100f0	0x0000000000000000
0x4100f8	not defined

After STR x4, [x1] instruction

Memory Address	Memory Contents
0x4100ec	not defined
0x4100f0	0x654d206f6c6c6400
0x4100f8	not defined

The contents of register x4 are stored in the memory address pointed to by register x1

Referring to the ARM instruction document, Rt corresponds to register x4 and register Rn corresponds to register x1.

Disassembly shows:-

```
4000b0:      d2800020     movz   x0, #0x1
  4000b4:    580001a1     ldr    x1, 4000e8 <_start+0x38>
  4000b8:    528ca904     movz   w4, #0x6548
  4000bc:    f2ad8d84     movk   x4, #0x6c6c, lsl #16
  4000c0:    f2c40de4     movk   x4, #0x206f, lsl #32
  4000c4:    f2eca9a4     movk   x4, #0x654d, lsl #48
  4000c8:    f9000024     str    x4, [x1]
```

The instruction f9400024 ldr x4, [x1] breaks down as follows:-

ldr x4, [x1]   with optional positive immediate unsigned byte offset which is a multiple of 4
f9400024

size		VR		opc	imm12	Rn	Rt
31 30	29 28 27	26	25	24 23 22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5	4 3 2 1 0
1 1	1 1 1	0	0	1 0 1	0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1	0 0 1 0 0
size 11 = 64-bit 10=32-bit						x1	x4

immdiate value is 0
Rn is x1
Rt is x4

Addressing modes

The table below summarizes various addressing modes used with ARM64 architecture –

Table 3-3 Summary of addressing modes

Addressing Mode	Parameters	Meaning	Format
Simple (Pc relative addressing)	Register	Register x1 is loaded with the contents of the address pointed to by base register x0. The base register is always 64-bit, since the addresses are 64-bit wide.	ldr x1, [x0] ldr x1, =mylabel
Offset	Register plus an offset	Register w2 is loaded with the contents of the address pointed to by base register x0 plus an offset. The offset may be a constant (immediate value) or another register	ldrh w2, [x0, #8] ldrb w2, [x0, x10]
Pre-indexed	Offset	Similar to the offset address mode, except that the base register(x0) is updated with the new calculated address and data is loaded from the new location. The update happens *before* fetching the data	ldrh w2, [x0, #8]!
Post-index	Offset	Similar to the pre-indexed mode, except that the data is loaded from the current base register and the base register is updated only then with the new calculated address. Update happens after fetching the data.	ldrh w2, [x0], #7

- [ ] signifies *indirection*
- ! signifies *pre-indexing*, offset inside brackets
- Post indexing, offset outside of brackets

The next listing shows examples of these addressing modes and resulting register contents using the data in the layout shown below.

Memory Address	Memory Contents
0x4100d8	0x1817161514131211
0x4100e0	0x191a1b1c1d1e1f20

Listing 3-3 Addressing modes

```
/* This example shows ARM64 addressing modes-
.text
.global _start
_start:
       ldr    x0, =baselocation //This loads the address baselocation (0x4100d8)
into x0
```

```
// Simple Addressing Mode

       ldr    x1, [x0] //This loads the actual data (0x1817161514131211) from
baselocation into x1

//Offset addressing with a constant as an offset

       ldrh     w2, [x0, #8]      // Loads contents (0x1a19) of location 4100d8+8)
into w2

//Offset addressing with a register as an offset

       mov x10, #4  // Move offset value into register x10

       ldrb w2, [x0, x10]  // Loads contents (0x15) of location 4100d8+4) into w2

//Pre-Index Addressing Mode

       ldrh   w2, [x0, #8]! // Similar to offset except that x0 is updated with the
new calculated address. x0 now contains the address 0x4100e0 and w2 with the data
0x1a19


//Post-index Addressing mode

       ldrh   w2, [x0], #7 // Picks up the data at location 0x4100e0 and only then
updates x0 to 0x4100ef

              mov    w8, #0x5d    //Time to go.

              svc    #0

.data

       baselocation:             .byte 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18

       baselocationwithoffset:    .byte 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
0x20
```

Table 3-4 Effect of addressing modes on pointer registers

Addressing Mode	Register x0 contents after instruction	Register x1 contents	Register w2 contents	Comments
Simple	0x4100d8	0x1817161514131211	-	Loads full 64 bits
Offset with constant	0x4100d8	-	0x1a19	Loads halfword
Offset with register	0x4100d8	-	0x15 (byte)	Loads byte
Pre-index	0x4100e0	-	0x1a19 halfword	Loads halfword from new address
Post-index	0x4100ef	-	0x1a19 halfword	Loads halfword from address prior to update

Simple addressing is really *Program Counter relative with an* offset?

The instruction `ldr x0, = baselocation`, actually disassembles to `ldr x0, 4000d8 <_start+0x28>` where the program counter corresponds to the location 0x4000d8 and 0x28 being the offset where the data is located. The offset must be a multiple of four!

The difference between pre and post indexing is the order in which the data comes from.

- Pre-index - the pointer register location is *first* updated, and the data is *then* fetched from the updated location.
- Post-index – the data is fetched from the *current* location and *only then* is the update applied to the pointer register.

Other modifications are possible with the ldr and str instructions which are fully documented in the ARM architecture guides.

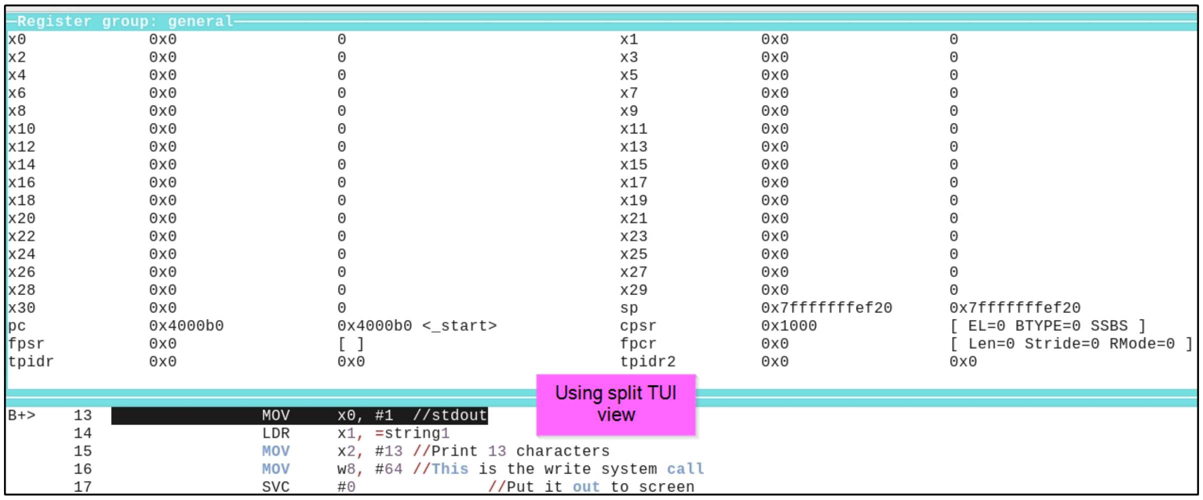
Enhancements to GDB

So far GDB has been used as the default tool for analyzing code. The following commands entered into the file `~/.gdbinit` will give a better (TUI) layout experience.

```
layout split
layout regs
set history save on
set history filename ~/gdbhistory
set logging enabled on
```

Note that if using the GDB TUI then the up and down arrows are no longer available for command history; use Ctrl-P(revious) and Ctrl-N(ext) instead.

Figure 3-1 GDB using TUI



```
-Register group: general-
x0             0x0                 0                                  x1             0x0                 0
x2             0x0                 0                                  x3             0x0                 0
x4             0x0                 0                                  x5             0x0                 0
x6             0x0                 0                                  x7             0x0                 0
x8             0x0                 0                                  x9             0x0                 0
x10            0x0                 0                                  x11            0x0                 0
x12            0x0                 0                                  x13            0x0                 0
x14            0x0                 0                                  x15            0x0                 0
x16            0x0                 0                                  x17            0x0                 0
x18            0x0                 0                                  x19            0x0                 0
x20            0x0                 0                                  x21            0x0                 0
x22            0x0                 0                                  x23            0x0                 0
x24            0x0                 0                                  x25            0x0                 0
x26            0x0                 0                                  x27            0x0                 0
x28            0x0                 0                                  x29            0x0                 0
x30            0x0                 0                                  sp             0x7ffffffffef20     0x7fffffffef20
pc             0x4000b0            0x4000b0 <_start>                  cpsr           0x1000              [ EL=0 BTYPE=0 SSBS ]
fpsr           0x0                 [ ]                                fpcr           0x0                 [ Len=0 Stride=0 RMode=0 ]
tpidr          0x0                 0x0                                tpidr2         0x0                 0x0

B+>                     MOV     x0, #1  //stdout
                        LDR     x1, =string1
                        MOV     x2, #13 //Print 13 characters
                        MOV     w8, #64 //This is the write system call
                        SVC     #0              //Put it out to screen
```

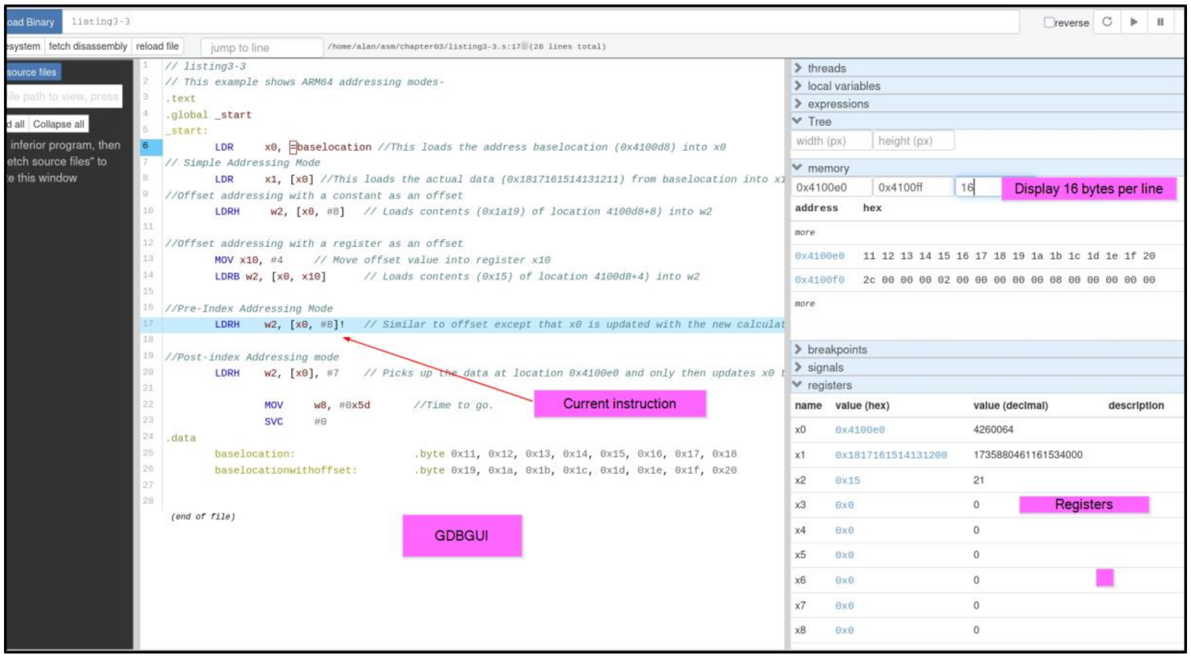
There are several enhancements/alternatives to GDB. One such tool that enhances the debugging experience is `gdbgui`. Installation instructions for installation can be found at `www.gdbgui`.

Start `gdbgui` from the command line by entering the following command:-

```
gdbgui --args ./asm/printhelloARM3/printhelloARM3
```

The screenshot is a snapshot of the program midway through. The memory location shows the values in hex and in character format. The GDB command window (not shown) is at the bottom left. Registers x0 through x8 are shown along with the source code: -

Figure 3-2 GDBGUI



```
/home/alan/asm/chapter03/listing3-3.s:17 (28 lines total)
```

```
// listing3-3
// This example shows ARM64 addressing modes-
.text
.global _start
_start:
        LDR     x0, =baselocation //This loads the address baselocation (0x4100d8) into x0
// Simple Addressing Mode
        LDR     x1, [x0] //This loads the actual data (0x1817161514131211) from baselocation into x1
//Offset addressing with a constant as an offset
        LDRH     w2, [x0, #8]   // Loads contents (0x1a19) of location 4100d8+8) into w2

//Offset addressing with a register as an offset
        MOV x10, #4     // Move offset value into register x10
        LDRB w2, [x0, x10]      // Loads contents (0x15) of location 4100d8+4) into w2

//Pre-Index Addressing Mode
        LDRH    w2, [x0, #8]!   // Similar to offset except that x0 is updated with the new calculat

//Post-index Addressing mode
        LDRH    w2, [x0], #7    // Picks up the data at location 0x4100e0 and only then updates x0 t

                MOV     w8, #0x5d       //Time to go.
                SVC     #0
.data
        baselocation:                   .byte 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18
        baselocationwithoffset:         .byte 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20


 (end of file)
```

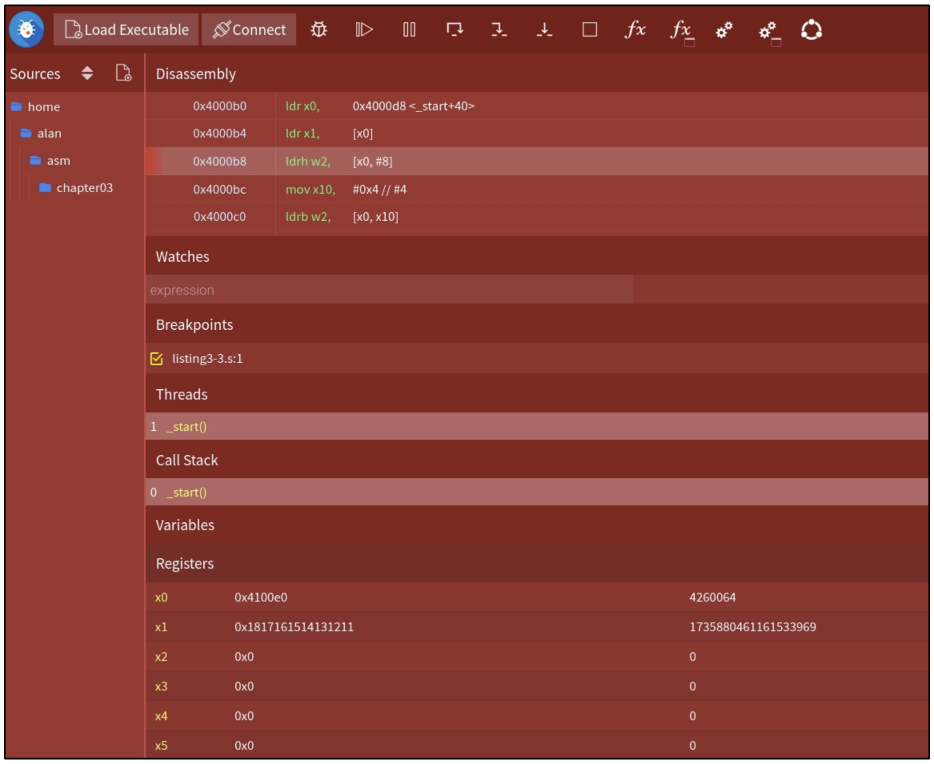
```
0x4100e0    0x4100ff    16
address     hex
more
0x4100e0    11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20
0x4100f0    2c 00 00 00 02 00 00 00 00 00 08 00 00 00 00 00
more
```

```
name  value (hex)           value (decimal)        description
x0    0x4100e0              4260064
x1    0x1817161514131200    1735880461161534000
x2    0x15                  21
x3    0x0                   0
x4    0x0                   0
x5    0x0                   0
x6    0x0                   0
x7    0x0                   0
x8    0x0                   0
```

Another alternative is `gdbfrontend`. This can be installed from

https://github.com/rohanrhu/gdb-frontend

Figure 3-3 GDB Frontend



```
0x4000b0    ldr x0,     0x4000d8 <_start+40>
0x4000b4    ldr x1,     [x0]
0x4000b8    ldrh w2,    [x0, #8]
0x4000bc    mov x10,    #0x4 // #4
0x4000c0    ldrb w2,    [x0, x10]
```

```
listing3-3.s:1
```

```
1  _start()
```

```
0  _start()
```

```
x0    0x4100e0              4260064
x1    0x1817161514131211    1735880461161533969
x2    0x0                   0
x3    0x0                   0
x4    0x0                   0
x5    0x0                   0
```

Summary of chapter 3

- Memory layout
- Load and Store Instruction
- Outputting text
- Addressing Modes
- Graphical Debuggers

Exercises for chapter3

1. How many bits are contained in an ARM64 instruction?
2. What does the command `x/32w  0x4100f0` do when executed in GDB?
3. What is the significance of the square [ ] brackets when used with `ldr` or `str` instructions?
4. What assembly directive is used to define a string of characters within an assembly language program?
5. Describe the purpose of the instruction `ldrh w2, [x0, #8]!`
6. What is the role of the x2 register when using the write syscall to print a string of text to the screen?

Chapter 4. Arithmetic operations (First Pass)

This section will introduce the arithmetic instruction capabilities of ARM64. A subsequent chapter discusses more advanced operation utilizing vector registers. Logic instructions such as AND, OR and EOR are also covered.

Floating Point operations are not covered in this section.

Recall the bit sizes as defined in Byte, . . . Quadword

Table 4-1 ARM64 Data Types

# of bits	Definition
8	Byte
16	Halfword
32	Word
64	Doubleword
128	Quadword

Add Instruction

– First start with `add`. Two numbers are placed in registers x4 and x5 with the result being stored in register x6.

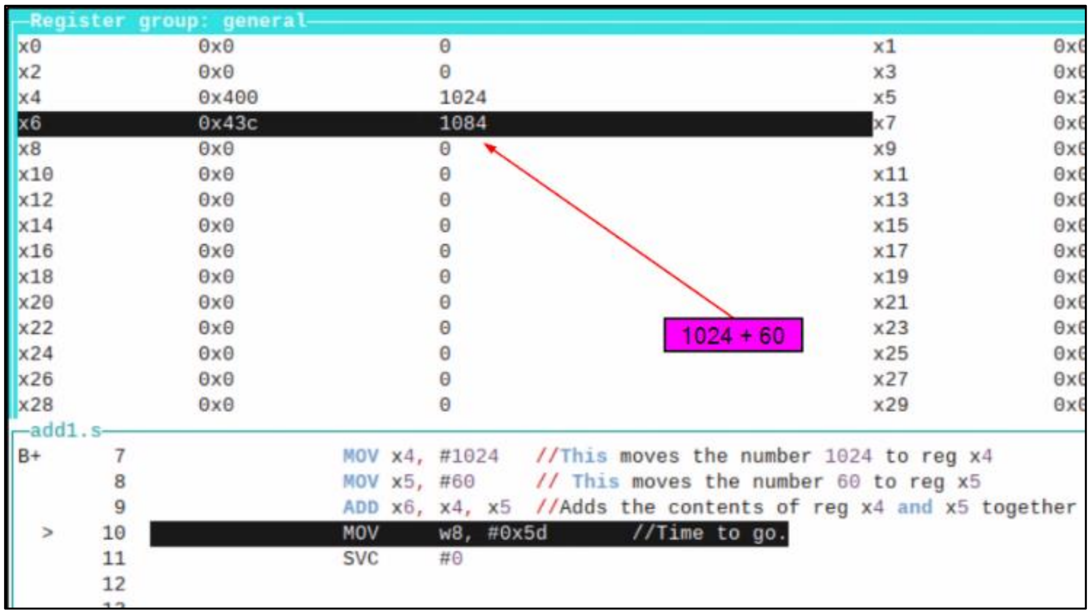
Listing 4-1Add (Extended Register)

```
* This example shows various addition instructions */
.text
.global _start
_start:
       mov x4, #1024
//This moves the number 1024 to reg x4
       mov x5, #60
// This moves the number 60 to reg x5
       add x6, x4, x5
//Adds the contents of x4 and x5 placing the result in x6
       mov    w8, #0x5d
svc #0
//Time to go.
```

Disassembly produces `objdump -d -M no-aliases add1`

```
add1:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000400078 <_start>:

  400078:     d2808004      movz   x4, #0x400

  40007c:     d2800785      movz   x5, #0x3c

  400080:     8b050086      add    x6, x4, x5

  400084:     52800ba8      movz   w8, #0x5d

  400088:     d4000001      svc    #0x0
```

Note that even though three distinct registers were used, using an instruction such as add *x5*, x4, *x5* is perfectly valid.



```
Register group: general
x0             0x0                0                                     x1             0x0
x2             0x0                0                                     x3             0x0
x4             0x400              1024                                  x5             0x3
x6             0x43c              1084                                  x7             0x0
x8             0x0                0                                     x9             0x0
x10            0x0                0                                     x11            0x0
x12            0x0                0                                     x13            0x0
x14            0x0                0                                     x15            0x0
x16            0x0                0                                     x17            0x0
x18            0x0                0                                     x19            0x0
x20            0x0                0                                     x21            0x0
x22            0x0                0                                     x23            0x0
x24            0x0                0                                     x25            0x0
x26            0x0                0                                     x27            0x0
x28            0x0                0                                     x29            0x0
add1.s
B+                      MOV x4, #1024    //This moves the number 1024 to reg x4
                        MOV x5, #60      // This moves the number 60 to reg x5
                        ADD x6, x4, x5   //Adds the contents of reg x4 and x5 together
  >                     MOV      w8, #0x5d        //Time to go.
                        SVC      #0
```

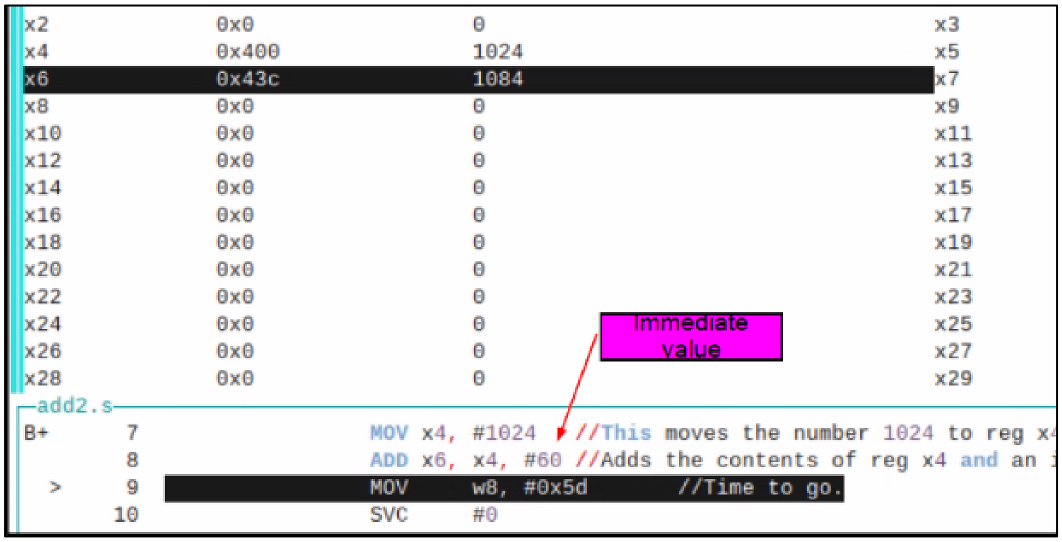
The next listing gives a similar result, the difference being that instead of a third register an immediate value is added.

Listing 4-2 Add (immediate)

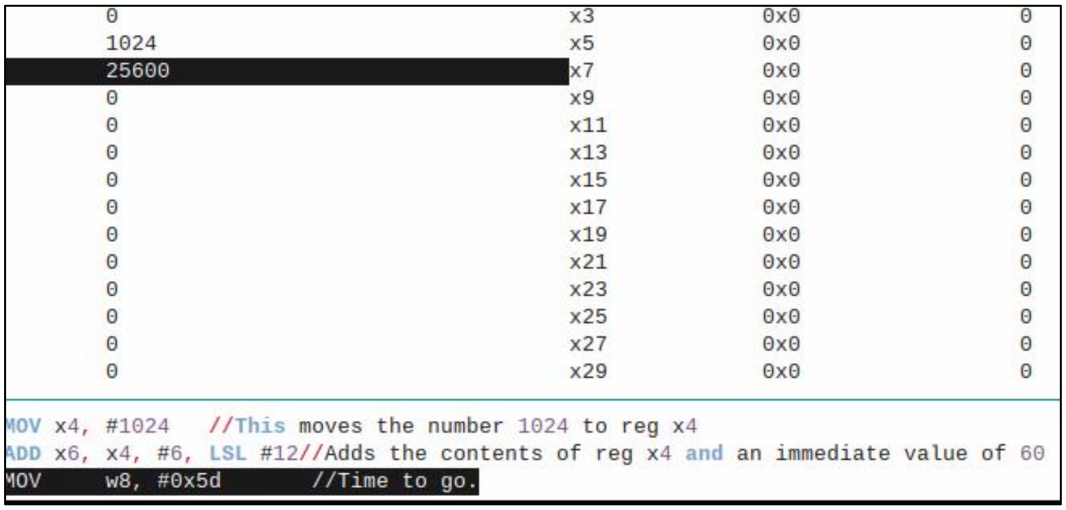
```
// listing4-2

.text

.global _start

       _start:

              mov x4, #1024        //This moves the number 1024 to reg x4
```

```
             add x6, x4, #60     //Adds the contents of reg x4 and an immediate
value of 60 placing the result in reg x6

             mov    w8, #0x5d    //Time to go.

             svc    #0
```

In Listing 4-3 a 12-bit offset is used – here the immediate value of 6 is left shifted by 12 places giving the value:- 0110 0000 0000 0000 = 0x6000 and then this is added to the content of reg x4 (0x400) = 0x6400.



```
x2             0x0                 0                                   x3
x4             0x400               1024                                x5
x6             0x43c               1084                                x7
x8             0x0                 0                                   x9
x10            0x0                 0                                   x11
x12            0x0                 0                                   x13
x14            0x0                 0                                   x15
x16            0x0                 0                                   x17
x18            0x0                 0                                   x19
x20            0x0                 0                                   x21
x22            0x0                 0                                   x23
x24            0x0                 0                                   x25
x26            0x0                 0                                   x27
x28            0x0                 0                                   x29
add2.s
B+                     MOV x4, #1024  //This moves the number 1024 to reg x4
                       ADD x6, x4, #60 //Adds the contents of reg x4 and an i
  >                    MOV     w8, #0x5d       //Time to go.
                       SVC     #0
```



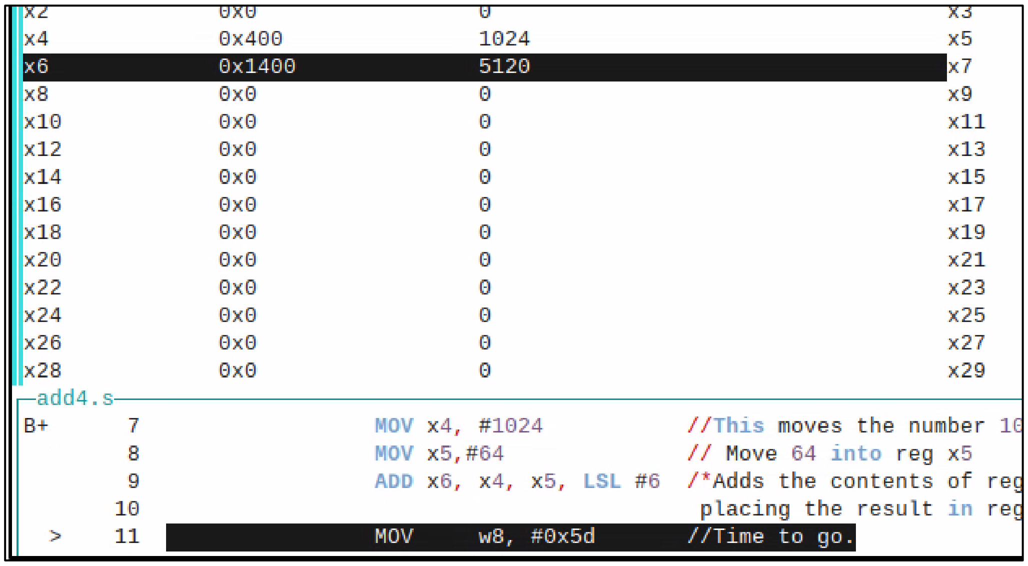
```
      0                                x3        0x0                 0
      1024                             x5        0x0                 0
      25600                            x7        0x0                 0
      0                                x9        0x0                 0
      0                                x11       0x0                 0
      0                                x13       0x0                 0
      0                                x15       0x0                 0
      0                                x17       0x0                 0
      0                                x19       0x0                 0
      0                                x21       0x0                 0
      0                                x23       0x0                 0
      0                                x25       0x0                 0
      0                                x27       0x0                 0
      0                                x29       0x0                 0

MOV x4, #1024   //This moves the number 1024 to reg x4
ADD x6, x4, #6, LSL #12//Adds the contents of reg x4 and an immediate value of 60
MOV     w8, #0x5d       //Time to go.
```

Listing 4-3 Add immediate with a left shift

```
// listing4-3

.text

.global _start

_start:
        mov x4, #1024

//This moves the number 1024 to reg x4

        add x6, x4, #6, LSL #12

//Adds the contents of reg x4 and an immediate value of 60

left shifted by 12, placing the result in reg x6*/

        mov    w8, #0x5d     //Time to go.

        svc    #0
```

Listing 4-4 Add with a left shifted register

```
// listing4-4

.text

.global _start

_start:

        mov x4, #1024 //This moves the number 1024 to reg x4

        mov x5,#64    // Move 64 into reg x5

        add x6, x4, x5, LSL #6


/*Adds the contents of reg x4 and reg x5 left shifted by 6 places placing the
result in reg x6*/

        mov    w8, #0x5d     //Time to go.

        svc    #0
```



```
x2             0x0                0                           x3
x4             0x400              1024                        x5
x6             0x1400             5120                        x7
x8             0x0                0                           x9
x10            0x0                0                           x11
x12            0x0                0                           x13
x14            0x0                0                           x15
x16            0x0                0                           x17
x18            0x0                0                           x19
x20            0x0                0                           x21
x22            0x0                0                           x23
x24            0x0                0                           x25
x26            0x0                0                           x27
x28            0x0                0                           x29
add4.s
B+                    MOV x4, #1024          //This moves the number 10
                      MOV x5,#64             // Move 64 into reg x5
                      ADD x6, x4, x5, LSL #6 /*Adds the contents of reg
                                              placing the result in reg
  >                   MOV     w8, #0x5d      //Time to go.
```

The next example introduces the extend operators. Values that can be extended are bytes, halfwords and words. In addition, they can be signed or unsigned. One further operation is to shift the values by one through four bits. The operations are shown in Table 4-2.

Table 4-2 Extend Operators

Operator	Meaning	Optional Shift
UXTB	Unsigned byte 8 bits to 64 bits	N = 1|2|3|4[19]
SXTB	Signed byte 8 bits to 64 bits	N = 1|2|3|4
UXTH	Unsigned halfword 16 bits to 64 bits	N = 1|2|3|4
SXTH	Signed halfword 16bits to 64 bits	N = 1|2|3|4
UXTW	Unsigned word 32 bits to 64 bits	N = 1|2|3|4
SXTW	Signed word 32bits to 64 bits	N = 1|2|3|4

The next listing shows the effect of a UXTB byte operation shifted by four places.

Listing 4-5 UXTB byte operation

```
// listing4-5
.text
.global _start
       _start:
             mov x4, #0x400      //This moves the number 1024 to reg x4
             mov x5, #0x55       // Move into reg x5
             add x6, x4, x5, UXTB #4 /* Unsign extends the byte in reg x5 (0x55),
shifting it four places, adding it to reg 4 placing the result in reg x6*/
             mov    w8, #0x5d    //Time to go.
             svc    #0
```

The value ending up in register x6 is 0x950. A breakdown follows:-

- x5 = 0x55 = 0b01010101
- Shift the *value* of x5 by four places to the left = 0b010101010000 = 0x550

[19] For "|" read or.

- Add 0x550 to the contents of register x4 (0x400) to get 0x950.

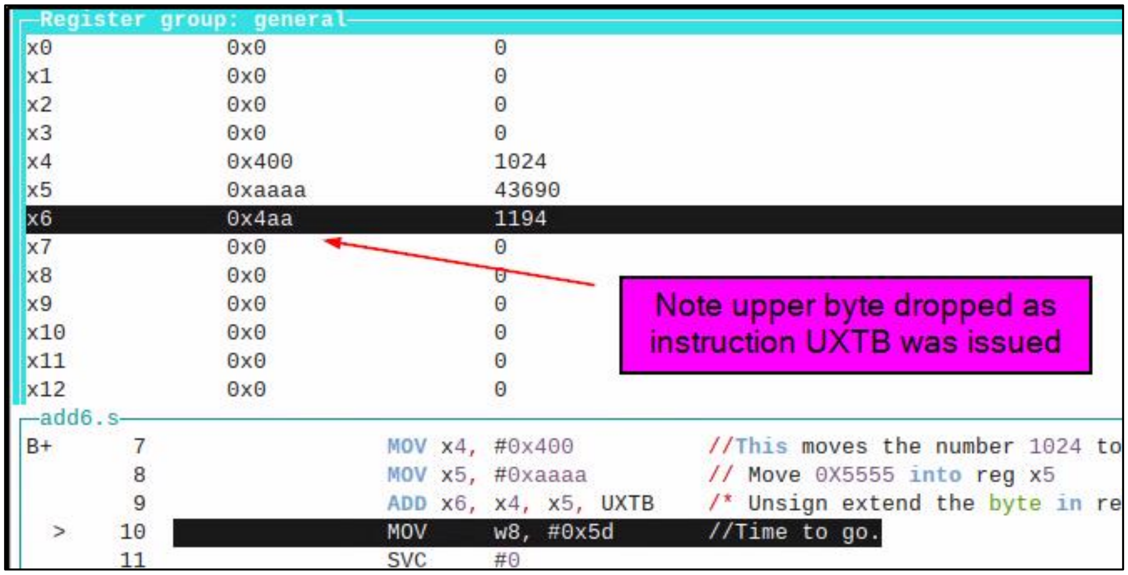
NOTE register x5 is unchanged, only its value is acted on.

The following example shows the operation of Unsigned to byte when a 16-bit value is contained in reg x5.

Listing 4-6 Add extended using UXTB on a halfword value

```
// listing4-6

.text

.global _start

_start:

       mov x4, #0x400

//This moves the number 1024 to reg x4

       mov x5, #0xaaaa

// Move 0Xaaaa into reg x5

       add x6, x4, x5, UXTB

/* Unsign extend the byte in reg x5 (0xaaaa), placing the result in reg x6*/

              mov    w8, #0x5d     //Time to go.

              svc    #0
```

Here the instruction sign extended a *byte* value so only 8 bits were extended not 16 bits!



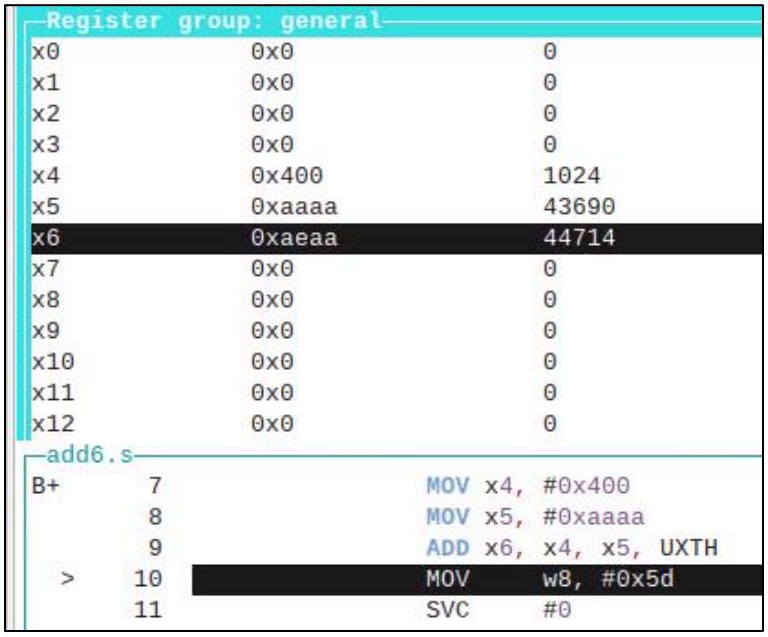
```
Register group: general
x0             0x0                 0
x1             0x0                 0
x2             0x0                 0
x3             0x0                 0
x4             0x400               1024
x5             0xaaaa              43690
x6             0x4aa               1194
x7             0x0                 0
x8             0x0                 0
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
add6.s
B+                          MOV x4, #0x400          //This moves the number 1024 to
                            MOV x5, #0xaaaa         // Move 0X5555 into reg x5
                            ADD x6, x4, x5, UXTB    /* Unsign extend the byte in re
   >                        MOV     w8, #0x5d       //Time to go.
                            SVC     #0
```

The next example uses UXTH to extend the full halfword value.

Listing 4-7 Add extended using UXTH on a halfword value

```
// listing4-7
```

```
.text

.global _start

       _start:

              mov x4, #0x400             //This moves the number 1024 to reg x4

              mov x5, #0xaaaa            // Move 0X5555 into reg x5

              add x6, x4, x5, UXTW    /* Unsign extend the byte in reg x5 (0xaaaa),
placing the result in reg x6*/

              mov    w8, #0x5d    //Time to go.

svc    #0
```



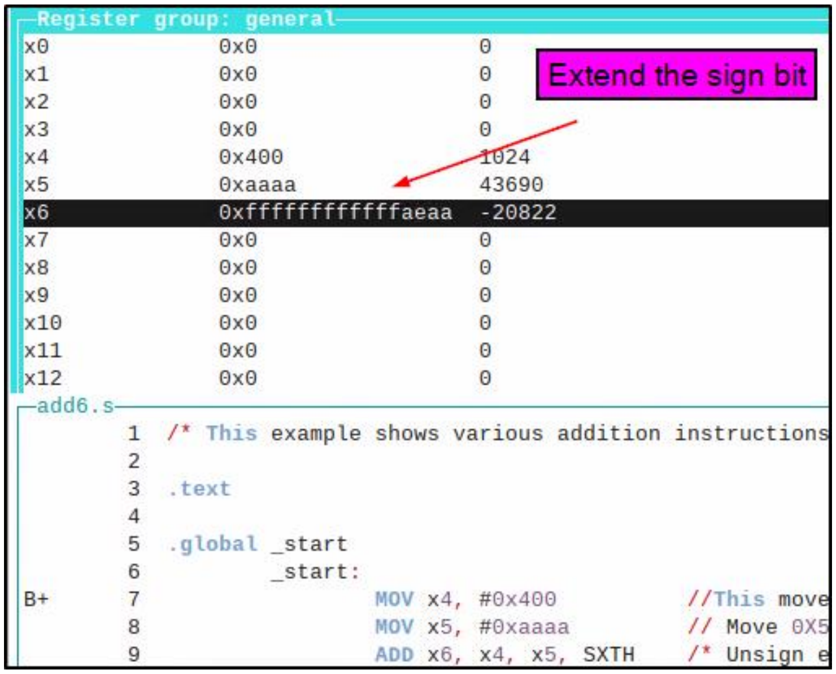
```
┌Register group: general────────────────────────
x0             0x0                 0
x1             0x0                 0
x2             0x0                 0
x3             0x0                 0
x4             0x400               1024
x5             0xaaaa              43690
x6             0xaeaa              44714
x7             0x0                 0
x8             0x0                 0
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
┌add6.s─────────────────────────────────────────
B+                      MOV x4, #0x400
                        MOV x5, #0xaaaa
                        ADD x6, x4, x5, UXTH
  >                     MOV     w8, #0x5d
                        SVC     #0
```

Changing the listing to SXTH extends the sign bit. Previously the extend operation was unsigned so the extended leading zeroes were simply dropped. The value 0xaaaa is 1010 1010 1010 1010 in binary so the leading bit is a one denoting that it is a negative number using signed binary.

Listing 4-8 Add extended using SXTH on a negative number

```
//listing4-8

.text

.global _start

       _start:

              mov x4, #0x400             //This moves the number 1024 to reg x4

              mov x5, #0xaaaa            // Move 0Xaaaa into reg x5

              add x6, x4, x5, SXTH    /* Unsign extend the byte in reg x5 (0xaaaa),
placing the result in reg x6*/
```

```
            mov    w8, #0x5d    //Time to go.
            svc #0
```

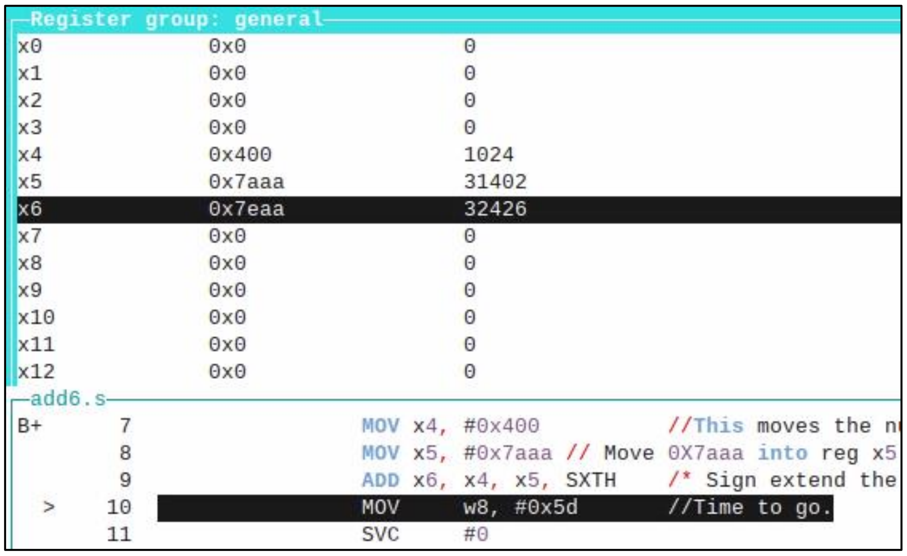


```
─Register group: general─
x0             0x0                 0
x1             0x0                 0
x2             0x0                 0
x3             0x0                 0
x4             0x400               1024
x5             0xaaaa              43690
x6             0xffffffffffffaeaa  -20822
x7             0x0                 0
x8             0x0                 0
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
─add6.s─
        /* This example shows various addition instructions

        .text

        .global _start
                _start:
B+                      MOV x4, #0x400          //This move
                        MOV x5, #0xaaaa         // Move 0X5
                        ADD x6, x4, x5, SXTH    /* Unsign e
```

Changing the sign bit to zero gives :-

Listing 4-9 Add extended using SXTH on a positive number

```
//listing4-9
.text
.global _start
      _start:
            mov x4, #0x400            //This moves the number 1024 to reg x4
            mov x5, #0x7aaa     // Move 0X7aaa into reg x5 giving a signed
positive number
            add x6, x4, x5, SXTH    /* Sign extend the halfword in reg x5
(0x7aaa), placing the result in reg x6*/
            mov    w8, #0x5d    //Time to go.
            svc #0
```

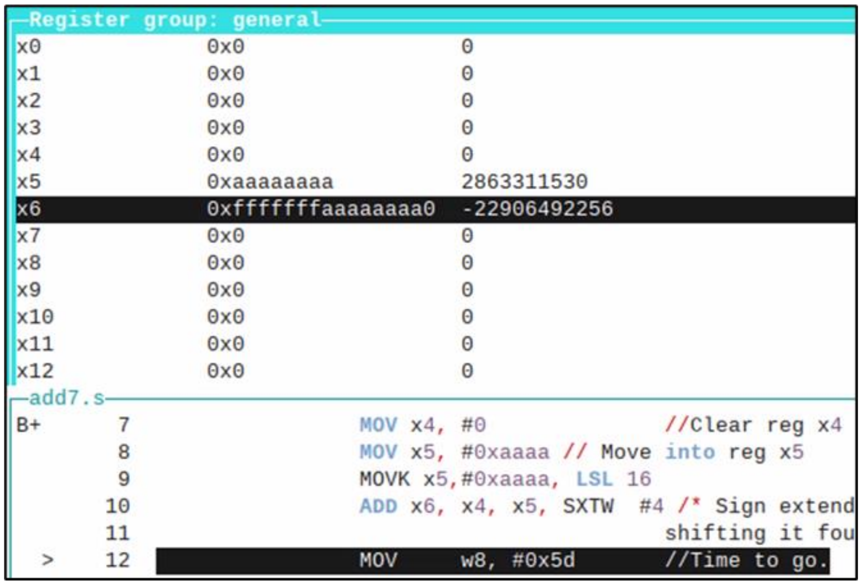


```
Register group: general
x0             0x0                 0
x1             0x0                 0
x2             0x0                 0
x3             0x0                 0
x4             0x400               1024
x5             0x7aaa              31402
x6             0x7eaa              32426
x7             0x0                 0
x8             0x0                 0
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
add6.s
B+                      MOV x4, #0x400          //This moves the n
                        MOV x5, #0x7aaa // Move 0X7aaa into reg x5
                        ADD x6, x4, x5, SXTH    /* Sign extend the
  >                     MOV     w8, #0x5d       //Time to go.
                        SVC     #0
```

Listing 4-10 Add extended SXTW with a 4-place shift

```
// listing4-10
.text
.global _start
_start:
       mov x, #0             //Clear reg x4
       mov x5, #0xaaaa       // Move into reg x5
       movk x5,#0xaaaa, LSL 16
       add x6, x4, x5, SXTW #4 /* Sign extend the word in reg x5 (0xaaaaaaaa),
shifting it four places, adding it to reg 4 placing the result in reg x6*/
       mov    w8, #0x5d      //Time to go.
       svc    #0
```

Here 0xaaaaaaaa is shifted four places to give 0xaaaaaaaa0 and then sign extended (since the leading bit is a one) to get 0xfffffffaaaaaaaa0.



```
Register group: general
x0             0x0                 0
x1             0x0                 0
x2             0x0                 0
x3             0x0                 0
x4             0x0                 0
x5             0xaaaaaaaa          2863311530
x6             0xffffffaaaaaaaa0   -22906492256
x7             0x0                 0
x8             0x0                 0
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
add7.s
B+                        MOV x4, #0               //Clear reg x4
                          MOV x5, #0xaaaa // Move into reg x5
                          MOVK x5,#0xaaaa, LSL 16
                          ADD x6, x4, x5, SXTW  #4 /* Sign extend
                                                   shifting it fou
  >                       MOV     w8, #0x5d        //Time to go.
```

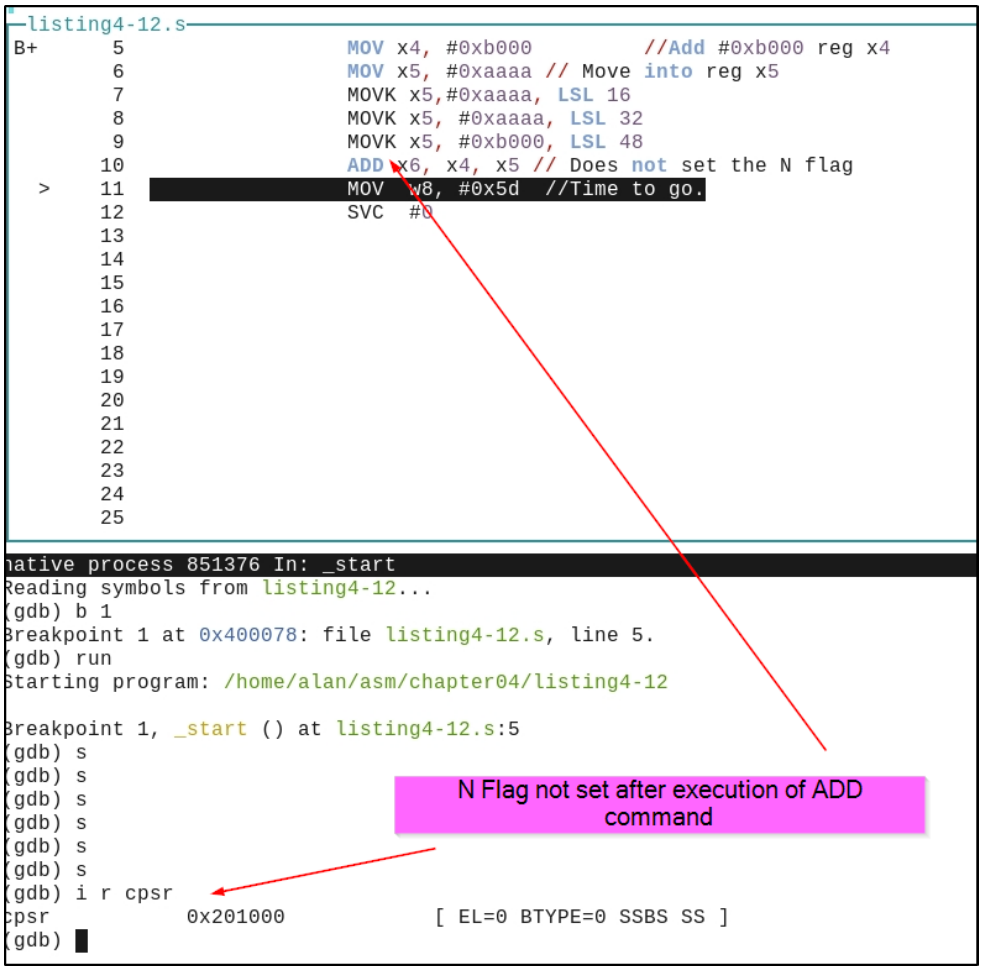
ADDS instruction.

So far, the instructions that have been used do not set the condition flags. The `adds` instruction will do this. Consider the first listing where the add instruction is used, after execution of the add instruction the CPSR bits are unchanged.

Due to the large data sizes involved many operations do not have to take the condition flags into account. An example could be the number of employees in a company – using a 32-bit data size is never going to reach an overflow condition! This will also speed up operations without having to carry out checks.

Listing 4-11 Leaving condition flags unchanged with the add instruction.

```
//listing4-11
.text
.global _start
       _start:
              mov x4, #0xb000            //Add #0xb000 reg x4
              mov x5, #0xaaaa     // Move into reg x5
              movk x5,#0xaaaa, LSL 16
              movk x5, #0xaaaa, LSL 32
              movk x5, #0xb000, LSL 48
              add x6, x4, x5 // Does not set the N flag
              mov  w8, #0x5d      //Time to go.
              svc  #0
```



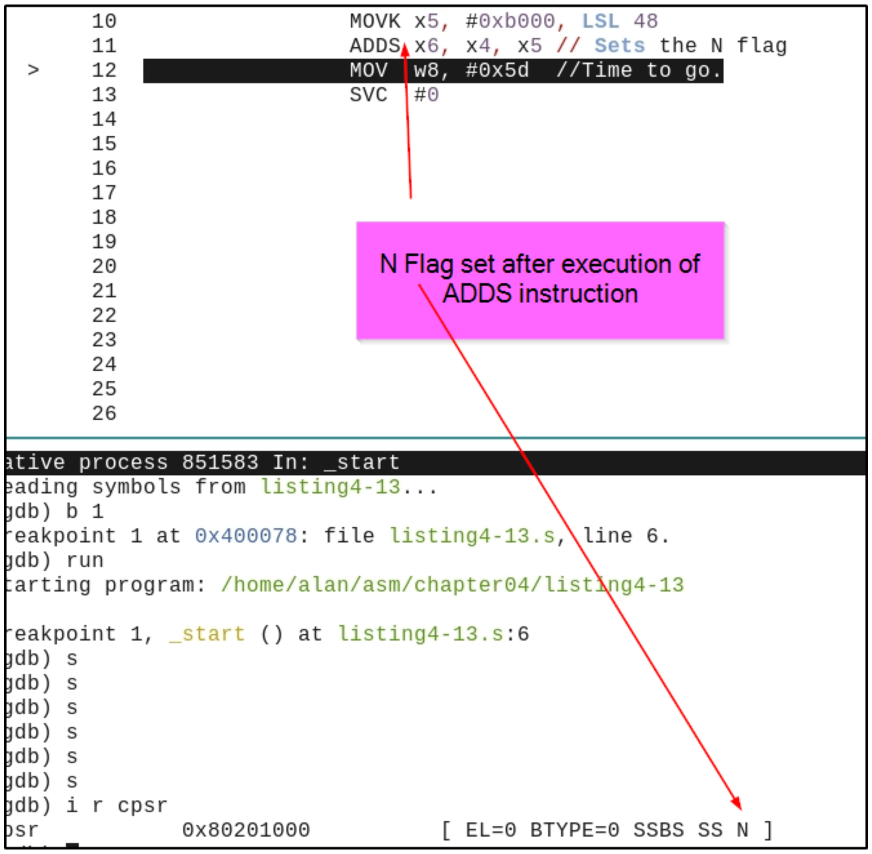
```
—listing4-12.s—
B+                 MOV x4, #0xb000         //Add #0xb000 reg x4
                   MOV x5, #0xaaaa // Move into reg x5
                   MOVK x5,#0xaaaa, LSL 16
                   MOVK x5, #0xaaaa, LSL 32
                   MOVK x5, #0xb000, LSL 48
                   ADD x6, x4, x5 // Does not set the N flag
   >               MOV  w8, #0x5d  //Time to go.
                   SVC  #0

native process 851376 In: _start
Reading symbols from listing4-12...
(gdb) b 1
Breakpoint 1 at 0x400078: file listing4-12.s, line 5.
(gdb) run
Starting program: /home/alan/asm/chapter04/listing4-12

Breakpoint 1, _start () at listing4-12.s:5
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) i r cpsr
cpsr           0x201000            [ EL=0 BTYPE=0 SSBS SS ]
(gdb)
```

Note that the `adds` **instruction *does* change the CPSR status.**

Listing 4-12 Setting the negative flag using the adds instruction

```
//listing4-12

.text

.global _start

       _start:

              mov x4, #0xb000      //Add #0xb000 reg x4

              mov x5, #0xaaaa      // Move into reg x5

              movk x5,#0xaaaa, LSL 16

              movk x5, #0xaaaa, LSL 32

              movk x5, #0xb000, LSL 48

              adds x6, x4, x5 // Sets the N flag
```

```
            mov  w8, #0x5d      //Time to go.

            svc  #0
```



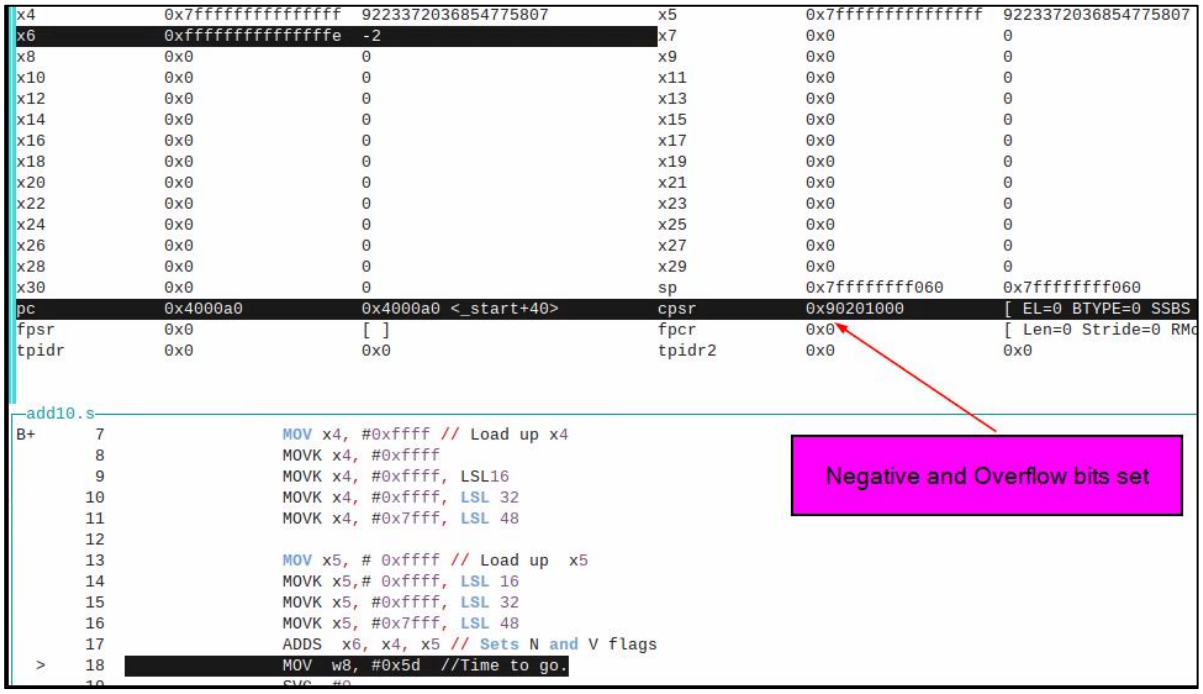
```
                    MOVK x5, #0xb000, LSL 48
                    ADDS x6, x4, x5 // Sets the N flag
>                   MOV  w8, #0x5d  //Time to go.
                    SVC  #0

ative process 851583 In: _start
eading symbols from listing4-13...
gdb) b 1
reakpoint 1 at 0x400078: file listing4-13.s, line 6.
gdb) run
tarting program: /home/alan/asm/chapter04/listing4-13

reakpoint 1, _start () at listing4-13.s:6
gdb) s
gdb) s
gdb) s
gdb) s
gdb) s
gdb) s
gdb) i r cpsr
psr           0x80201000          [ EL=0 BTYPE=0 SSBS SS N ]
```

The next snippet causes an overflow condition as well as setting the negative flag:-

Listing 4-13 Setting the overflow flag using the adds instruction

```
//listing4-13

text

.global _start

       _start:

            mov x4, #0xffff     // Load up x4

            movk x4, #0xffff

            movk x4, #0xffff, LSL16

            movk x4, #0xffff, LSL 32

            movk x4, #0x7fff, LSL 48

            mov x5, # 0xffff // Load up  x5

            movk x5,# 0xffff, LSL 16

            movk x5, #0xffff, LSL 32
```

```
             movk x5, #0x7fff, LSL 48

             adds  x6, x4, x5 // Sets N and V flags

             mov  w8, #0x5d      //Time to go.

             svc  #0
```



```
x4             0x7fffffffffffffff  9223372036854775807    x5             0x7fffffffffffffff  9223372036854775807
x6             0xfffffffffffffffe  -2                     x7             0x0                 0
x8             0x0                 0                      x9             0x0                 0
x10            0x0                 0                      x11            0x0                 0
x12            0x0                 0                      x13            0x0                 0
x14            0x0                 0                      x15            0x0                 0
x16            0x0                 0                      x17            0x0                 0
x18            0x0                 0                      x19            0x0                 0
x20            0x0                 0                      x21            0x0                 0
x22            0x0                 0                      x23            0x0                 0
x24            0x0                 0                      x25            0x0                 0
x26            0x0                 0                      x27            0x0                 0
x28            0x0                 0                      x29            0x0                 0
x30            0x0                 0                      sp             0x7fffffff060       0x7fffffff060
pc             0x4000a0            0x4000a0 <_start+40>   cpsr           0x90201000          [ EL=0 BTYPE=0 SSBS
fpsr           0x0                 [ ]                    fpcr           0x0                 [ Len=0 Stride=0 RMo
tpidr          0x0                 0x0                    tpidr2         0x0                 0x0

-add10.s-
B+                  MOV x4, #0xffff // Load up x4
                    MOVK x4, #0xffff
                    MOVK x4, #0xffff, LSL16
                    MOVK x4, #0xffff, LSL 32
                    MOVK x4, #0x7fff, LSL 48

                    MOV x5, # 0xffff // Load up  x5
                    MOVK x5,# 0xffff, LSL 16
                    MOVK x5, #0xffff, LSL 32
                    MOVK x5, #0x7fff, LSL 48
                    ADDS  x6, x4, x5 // Sets N and V flags
 >                  MOV  w8, #0x5d  //Time to go.
```

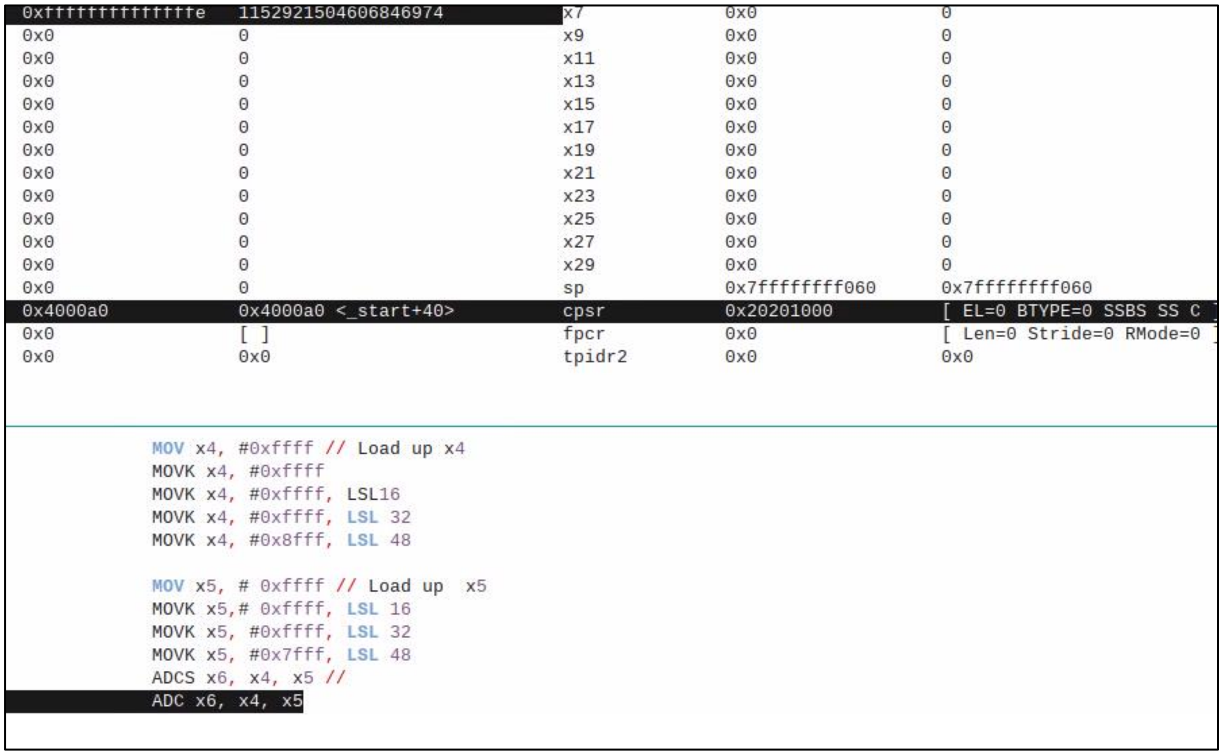
The next listing will differentiate between `ADCS, ADC` and `add` instructions.

Listing 4-14 Effect of ADCS, ADC and add instructions

```
// listing4-14

.text

_start:

             mov x4, #0xffff     // Load up x4

             movk x4, #0xffff

             movk x4, #0xffff, LSL16

             movk x4, #0xffff, LSL 32

             movk x4, #0x8fff, LSL 48

             mov x5, # 0xffff // Load up  x5

             movk x5,# 0xffff, LSL 16

             movk x5, #0xffff, LSL 32

             movk x5, #0x7fff, LSL 48

             ADCS x6, x4, x5 //
```

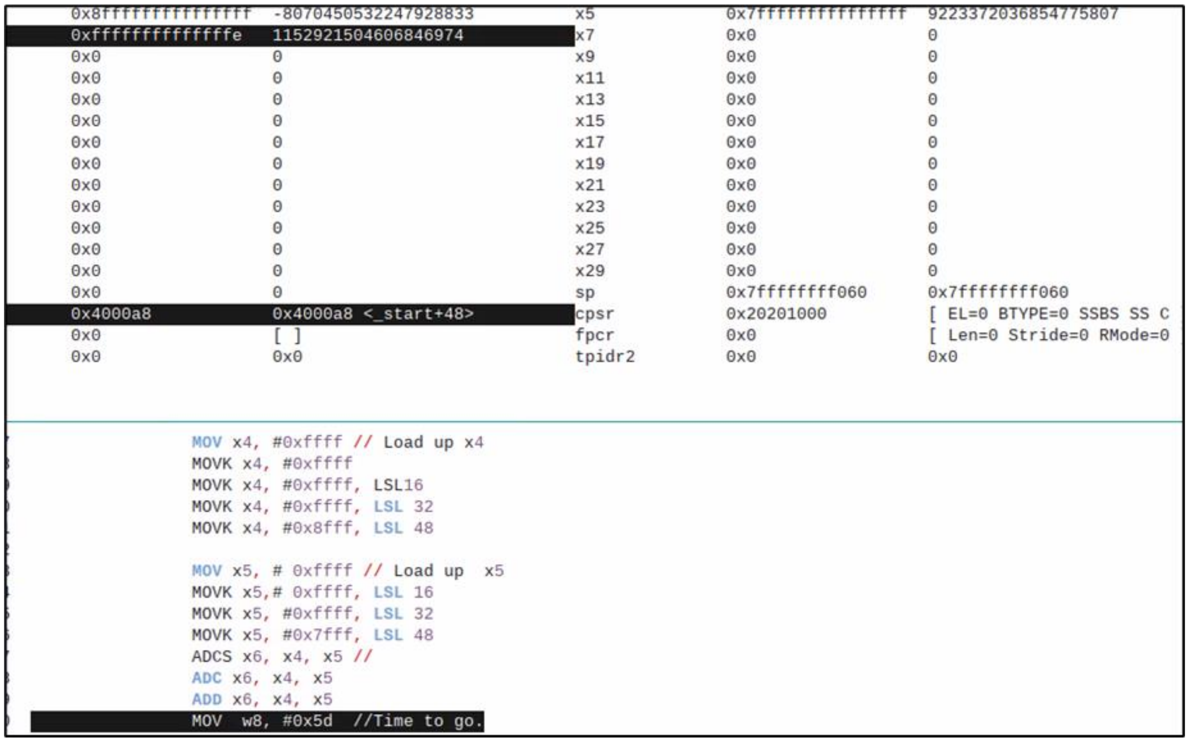
```
            ADC x6, x4, x5

            add x6, x4, x5

            mov  w8, #0x5d      //Time to go.

            svc  #0
```

After execution of `ADCS`, register x6 contains the value 0xfffffffffffffffe and the Carry bit has been set (CPSR = 0x20201000.



```
0xfffffffffffffffe   1152921504606846974        x7          0x0                0
0x0                  0                          x9          0x0                0
0x0                  0                          x11         0x0                0
0x0                  0                          x13         0x0                0
0x0                  0                          x15         0x0                0
0x0                  0                          x17         0x0                0
0x0                  0                          x19         0x0                0
0x0                  0                          x21         0x0                0
0x0                  0                          x23         0x0                0
0x0                  0                          x25         0x0                0
0x0                  0                          x27         0x0                0
0x0                  0                          x29         0x0                0
0x0                  0                          sp          0x7ffffffff060     0x7ffffffff060
0x4000a0             0x4000a0 <_start+40>       cpsr        0x20201000         [ EL=0 BTYPE=0 SSBS SS C
0x0                  [ ]                        fpcr        0x0                [ Len=0 Stride=0 RMode=0
0x0                  0x0                        tpidr2      0x0                0x0

              MOV x4, #0xffff // Load up x4
              MOVK x4, #0xffff
              MOVK x4, #0xffff, LSL16
              MOVK x4, #0xffff, LSL 32
              MOVK x4, #0x8fff, LSL 48

              MOV x5, # 0xffff // Load up  x5
              MOVK x5,# 0xffff, LSL 16
              MOVK x5, #0xffff, LSL 32
              MOVK x5, #0x7fff, LSL 48
              ADCS x6, x4, x5 //
              ADC x6, x4, x5
```

The instruction `ADC` also adds x4 and x5 but this time it includes the C bit giving the result in x6 of 0xffffffffffffffff. The last instruction add does not include the C bit; giving the result in x6 of 0xfffffffffffffffe.



```
        0x8fffffffffffffff   -8070450532247928833    x5        0x7fffffffffffffff  9223372036854775807
        0xfffffffffffffffe   1152921504606846974     x7        0x0                 0
        0x0                  0                       x9        0x0                 0
        0x0                  0                       x11       0x0                 0
        0x0                  0                       x13       0x0                 0
        0x0                  0                       x15       0x0                 0
        0x0                  0                       x17       0x0                 0
        0x0                  0                       x19       0x0                 0
        0x0                  0                       x21       0x0                 0
        0x0                  0                       x23       0x0                 0
        0x0                  0                       x25       0x0                 0
        0x0                  0                       x27       0x0                 0
        0x0                  0                       x29       0x0                 0
        0x0                  0                       sp        0x7ffffffff060      0x7ffffffff060
        0x4000a8             0x4000a8 <_start+48>    cpsr      0x20201000          [ EL=0 BTYPE=0 SSBS SS C
        0x0                  [ ]                     fpcr      0x0                 [ Len=0 Stride=0 RMode=0
        0x0                  0x0                     tpidr2    0x0                 0x0
```

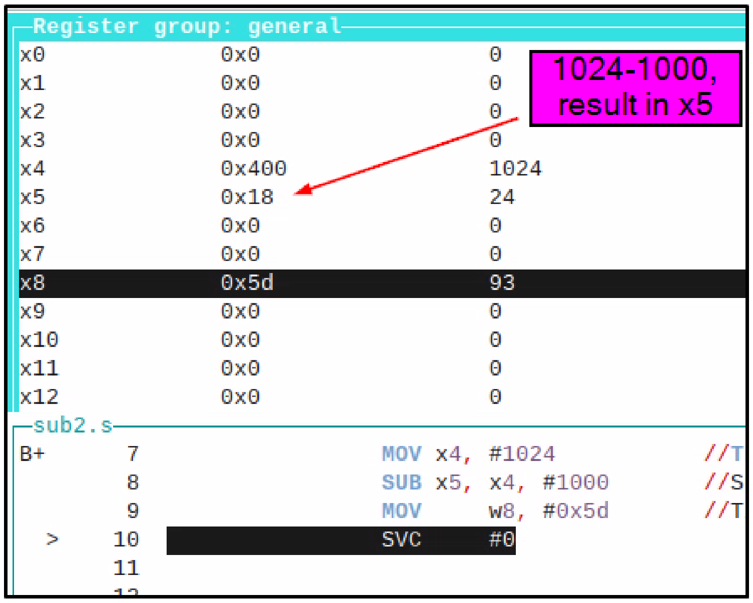
```
                 MOV x4, #0xffff // Load up x4
                 MOVK x4, #0xffff
                 MOVK x4, #0xffff, LSL16
                 MOVK x4, #0xffff, LSL 32
                 MOVK x4, #0x8fff, LSL 48

                 MOV x5, # 0xffff // Load up  x5
                 MOVK x5,# 0xffff, LSL 16
                 MOVK x5, #0xffff, LSL 32
                 MOVK x5, #0x7fff, LSL 48
                 ADCS x6, x4, x5 //
                 ADC x6, x4, x5
                 ADD x6, x4, x5
                 MOV  w8, #0x5d  //Time to go.
```

SUB Instruction

Subtraction instructions are similar to addition, consequently not too much time will be spent here.

Listing 4-15 SUB (extended register)

```
//listing4-15

.text

.global _start

      _start:

             mov x4, #1024       //This moves the number 1024 to reg x4

             mov x5, #60         // This moves the number 60 to reg x5

      SUB x5, x4, x5      //Subtracts the contents of reg x4 from x5 placing the
result in reg x5

             mov    w8, #0x5d    //Time to go.

svc #0
```



```
Register group: general
x0             0x0                 0
x1             0x0                 0
x2             0x0                 0
x3             0x0                 0
x4             0x400               1024
x5             0x18                24
x6             0x0                 0
x7             0x0                 0
x8             0x5d                93
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
sub2.s
B+                    MOV x4, #1024           //T
                      SUB x5, x4, #1000       //S
                      MOV     w8, #0x5d       //T
  >                   SVC     #0
```

Listing 4-16 SUB (immediate instruction)

```
//listing4-16

.text

.global _start

       _start:

              mov x4, #1024       //This moves the number 1024 to reg x4

       SUB x5, x4, #1000    //Subtracts the contents of reg x4 from 80 placing the
result in reg x5

              mov    w8, #0x5d     //Time to go.

              svc    #0
```

MUL Instruction and variants

Note Multiply and divide instructions do not set flags!

madd

The `MUL` instruction is an alias of `madd`. `madd` takes two registers, multiplies their contents together, then adds a third value placing the result in the destination. If no addition is required, then this operand will have a value of zero (see note on page 4-19).

The format of the instruction is `madd Xd, Xn, Xm, Xa.`

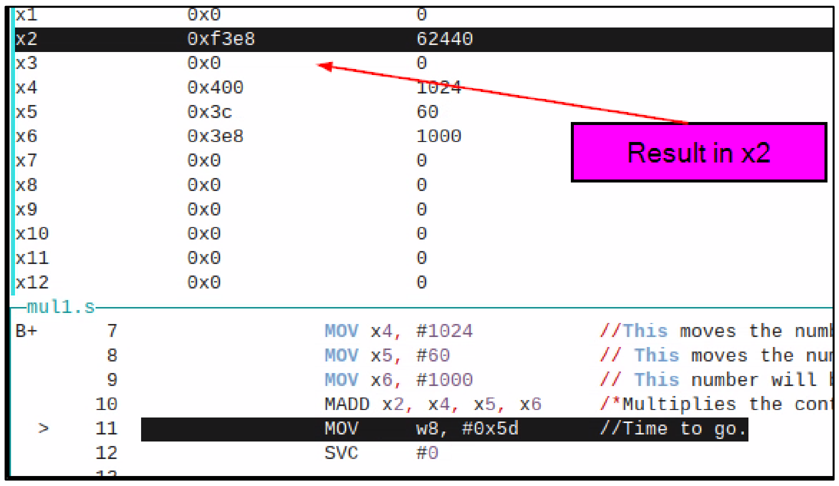
Note that the first two operands are 64 registers and so is the destination. Since a 128-bit destination would be required, the action is to discard the upper 64 bits. Now this is often acceptable for smaller numbers that do not cross a 64-bit threshold, but it is an issue that the programmer needs to be aware of.

Listing 4-17 madd Instruction

```
// listing4-17

.text

.global _start

_start:

      mov x4, #1024       //This moves the number 1024 to reg x4

      mov x5, #60         // This moves the number 60 to reg x5

      mov x6, #1000       // This number will be added

      madd x2, x4, x5, x6 /*Multiplies the contents of x4 and x5 together, adding
the contents of x6 and placing the result in x2*/

      mov    w8, #0x5d    //Time to go.

      svc    #0
```

Disassembly shows –

```
objdump -d mul1

mul1:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000400078 <_start>:

  400078:    d2808004     mov    x4, #0x400                         // #1024

  40007c:    d2800785     mov    x5, #0x3c                          // #60

  400080:    d2807d06     mov    x6, #0x3e8                         // #1000

  400084:    9b051882     madd   x2, x4, x5, x6

  400088:    52800ba8     mov    w8, #0x5d                          // #93

  40008c:    d4000001     svc    #0x0
```

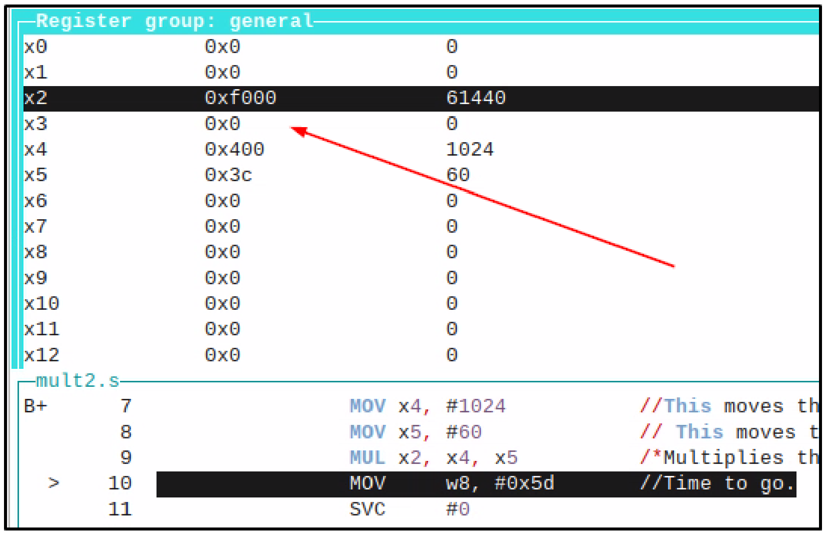


```
x1             0x0                 0
x2             0xf3e8              62440
x3             0x0                 0
x4             0x400               1024
x5             0x3c                60
x6             0x3e8               1000
x7             0x0                 0
x8             0x0                 0
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
-mul1.s-
B+              MOV x4, #1024           //This moves the numb
                MOV x5, #60             // This moves the nu
                MOV x6, #1000           // This number will b
                MADD x2, x4, x5, x6     /*Multiplies the cont
  >             MOV     w8, #0x5d       //Time to go.
                SVC     #0
```

Without the add (third operand) component:-

Listing 4-18 MUL instruction

```
//listing4-18

.text

.global _start

       _start:

              mov x4, #1024       //This moves the number 1024 to reg x4

              mov x5, #60         // This moves the number 60 to reg x5

              MUL x2, x4, x5      /*Multiplies the contents of x4 and x5 together,
placing the result in reg x2*/

              mov    w8, #0x5d    //Time to go.

              svc    #0
```



```
Register group: general
x0             0x0                 0
x1             0x0                 0
x2             0xf000              61440
x3             0x0                 0
x4             0x400               1024
x5             0x3c                60
x6             0x0                 0
x7             0x0                 0
x8             0x0                 0
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x0                 0
mult2.s
B+                    MOV x4, #1024       //This moves th
                      MOV x5, #60         // This moves t
                      MUL x2, x4, x5      /*Multiplies th
  >                   MOV     w8, #0x5d   //Time to go.
                      SVC     #0
```

Note how the unaliased disassembly for MUL produces the instruction `madd x2, x4, x5, xzr`. Recall from page 2-4 that the XZR register returns zero when read.

```
$ objdump -d -M no-aliases mult2

mult2:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000400078 <_start>:

  400078:      d2808004      movz    x4, #0x400

  40007c:      d2800785      movz    x5, #0x3c

  400080:      9b057c82      madd    x2, x4, x5, xzr

  400084:      52800ba8      movz    w8, #0x5d

  400088:      d4000001      svc     #0x0
```

Aliased disassembly produces :-

```
~/asm/addition $ objdump -d mult2

mult2:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000400078 <_start>:

  400078:      d2808004      mov     x4, #0x400                      // #1024

  40007c:      d2800785      mov     x5, #0x3c                       // #60

  400080:      9b057c82      mul     x2, x4, x5
  400084:     52800ba8      mov    w8, #0x5d                            // #93
  400088:     d4000001      svc    #0x0
```

Multiply two 32-bit numbers using madd –

Listing 4-19 Using madd to multiply two 32-bit numbers.

```
//listing4-19
.text
.global _start
_start:
      mov x4, #0xffff     // Load up first 16 bits to reg x4
      movk x4,#0xffff,lsl #16    //Load up next set of 16 bits
      mov x5, #0xffff            // Now load reg x5
      movk x5,#0xffff,lsl #16
      MUL x2, x4, x5      /*Multiplies the contents of x4 and x5 together,
placing the result in reg x2*/
      mov    w8, #0x5d    //Time to go.
             svc    #0
```

Disassembly produces –

```
objdump -d -M no-aliases mult4
mult4:     file format elf64-littleaarch64
Disassembly of section .text:
0000000000400078 <_start>:
  400078:     d29fffe4      movz   x4, #0xffff
  40007c:     72bfffe4      movk   x4, #0xffff, lsl #16
  400080:     d29fffe5      movz   x5, #0xffff
  400084:     72bfffe5      movk   x5, #0xffff, lsl #16
  400088:     9b057c82      madd   x2, x4, x5, xzr
  40008c:     52800ba8      movz   w8, #0x5d
  400090:     d4000001      svc    #0x0
```

The result of the multiplication is:

0XFFFFFFFE00000001.

UMULL and SMULL

The instructions `UMULL`(Unsigned Multiply Long) and `SMULL` (Signed Multiply Long) are used to multiply two 32-bit w registers, giving a 64-bit (placing the result in an x register) unsigned or signed result.

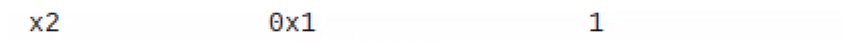
Listing 4-20 Unsigned Multiply Long

```
//listing 4-20
.text
.global _start
_start:
       mov w4, #0xffff     // Load up first 16 bits to reg w4
       movk w4,#0xffff,lsl #16    //Load up next set of 16 bits
       mov w5, #0xffff            // Now load reg w5
       movk 5,#0xffff,lsl #16
       UMULL x2, w4, w5           /*Multiplies the contents of w4 and w5 together,
placing the unsigned result in reg x2*/
              mov    w8, #0x5d    //Time to go.
              svc    #0
$ objdump -d -M no-aliases mult3
mult3:     file format elf64-littleaarch64
Disassembly of section .text:
0000000000400078 <_start>:
  400078:     529fffe4      movz   w4, #0xffff
  40007c:     72bfffe4      movk   w4, #0xffff, lsl #16
  400080:     529fffe5      movz   w5, #0xffff
  400084:     72bfffe5      movk   w5, #0xffff, lsl #16
  400088:     9ba57c82      umaddl x2, w4, w5, xzr
  40008c:     52800ba8      movz   w8, #0x5d
  400090:     d4000001      svc    #0x0
```

Note the unaliased UMULL instruction is `umaddl` which is the mnemonic for *Unsigned Multiply-Add Long*. The format is `UMAADDL Xd, Wn, Wm, Xa`. In this example Xd = x2, wn = w4, wm = w5, xa =0.

After execution the contents of x2 is 0xFFFFFFFE00000001.

Using SMULL gives the signed number 1.



```
x2               0x1                    1
```

Listing 4-21 Signed Multiply Long

```
//listing4-21
.text
.global _start
       _start:
              mov w4, #0xffff     // Load up first 16 bits to reg x4
              movk w4,#0xffff,lsl #16    //Load up next set of 16 bits
              mov w5, #0xffff            // Now load reg x5
              movk w5,#0xffff,lsl #16
              SMULL x2, w4, w5           /*Multiplies the contents of x4 and x5
together, placing the result in reg x2*/
              mov    w8, #0x5d    //Time to go.
              svc    #0
$ objdump -d -M no-aliases mult3
mult3:     file format elf64-littleaarch64
Disassembly of section .text:
0000000000400078 <_start>:
  400078:     529fffe4      movz   w4, #0xffff
  40007c:     72bfffe4      movk   w4, #0xffff, lsl #16
  400080:     529fffe5      movz   w5, #0xffff
  400084:     72bfffe5      movk   w5, #0xffff, lsl #16
  400088:     9b257c82      smaddl x2, w4, w5, xzr
  40008c:     52800ba8      movz   w8, #0x5d
  400090:     d4000001      svc    #0x0
```

Multiplication of two 64-bit numbers to give a 128-bit result.

The instructions `UMULH` (Unsigned Multiply High) and `SMULH` (Signed Multiply High) calculate the upper 64 bits of a 64-bit multiplication. UMULL and SMULL are used to multiply two 32-bit (w registers) together to get a 64-bit result. The U prefix signifies unsigned while the S prefix signifies signed. In this example UMULH is used to calculate the high order bits and MUL is used to calculate the low order bits.

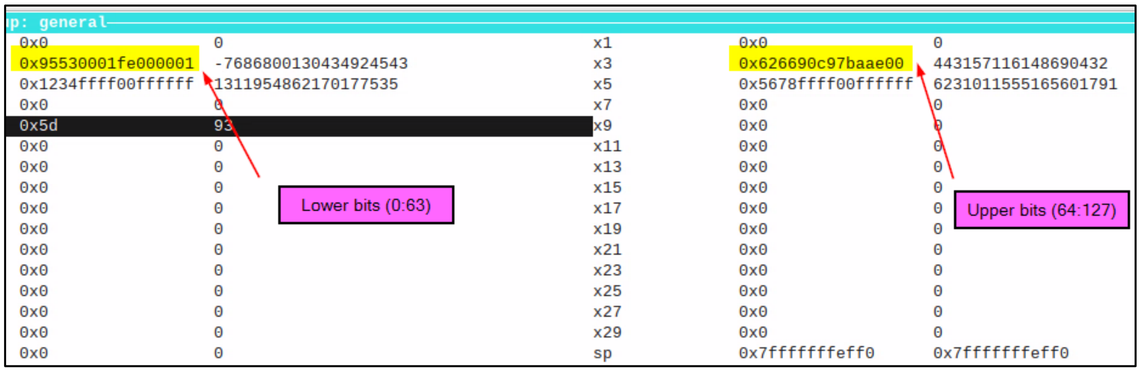
Note that UMUL**H** and SMUL**H** are not complementary to UMUL**L** and SMUL**L** .

Listing 4-22 Multiplying two 64-bit numbers to give a 128-bit result (Unsigned)

```
/* listing4-22

This example shows how to multiply two 64-bit numbers, placing the 128-bit result
in two 64-bit registers.

MUL is used for the lower 64 bits and UMULH is used for the higher 64 bits */

.text

.global _start

_start:

       mov x4, #0xffff     // Load up first 16 bits to reg x4

       movk x4,#0x00ff,lsl #16    // Load up next set of 16 bits

       movk x4,#0xffff,lsl #32 // Next 16 bits

       movk x4,#0x1234,lsl #48 // Last 16 bits


       mov x5, #0xffff            // Now load reg x5

       movk x5, #0x00ff,lsl #16

       movk x5, #0xffff,lsl #32

       movk x5, #0x5678,lsl #48

MUL x2, x4, x5      /*Multiplies the contents of x4 and x5 together, placing the
lower 64-bit result in reg x2*/

       UMULH x3, x4, x5 /*Multiplies the contents of x4 and x5 together, placing
the higher unsigned 64-bit result (64:127) result in reg x3, discarding lower 64
bits (0:63)*/

       mov    w8, #0x5d     //Time to go.

       svc    #0
```

The complete 128-bit result is: 0x0626 690c 97ba ae00 9553 0001 fe00 0001.

Note that *UMULL* is an alias of *UMADDL*

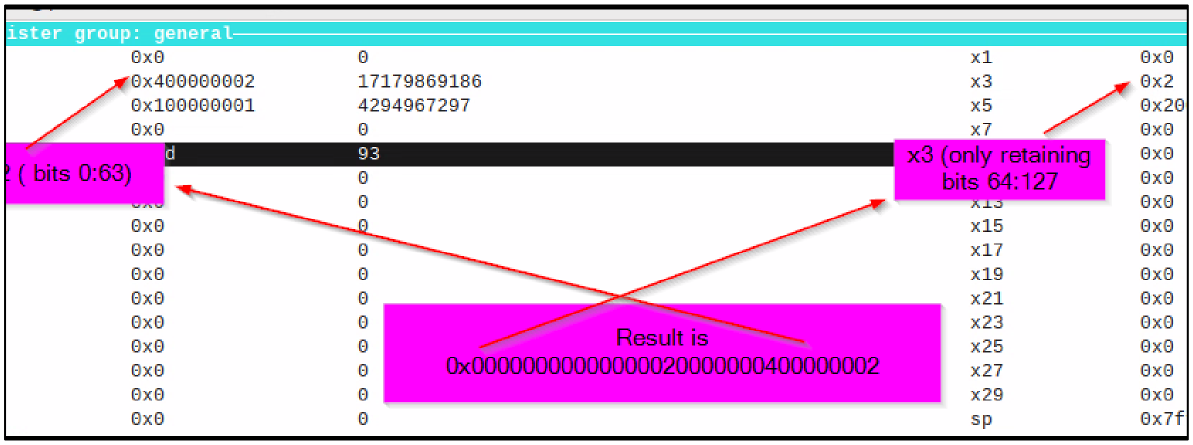


```
p: general
 0x0                0                                     x1         0x0                 0
 0x95530001fe000001 -7686800130434924543                  x3         0x626690c97baae00   443157116148690432
 0x1234ffff00ffffff 1311954862170177535                   x5         0x5678ffff00ffffff  6231011555165601791
 0x0                0                                     x7         0x0                 0
 0x5d               93                                    x9         0x0                 0
 0x0                0                                     x11        0x0                 0
 0x0                0                                     x13        0x0                 0
 0x0                0                                     x15        0x0                 0
 0x0                0                                     x17        0x0                 0
 0x0                0                                     x19        0x0                 0
 0x0                0                                     x21        0x0                 0
 0x0                0                                     x23        0x0                 0
 0x0                0                                     x25        0x0                 0
 0x0                0                                     x27        0x0                 0
 0x0                0                                     x29        0x0                 0
 0x0                0                                     sp         0x7ffffffeff0       0x7ffffffeff0
```

Another Example -

Listing 4-23 Second example - Multiplying two 64-bit numbers to give a 128-bit result (Unsigned)

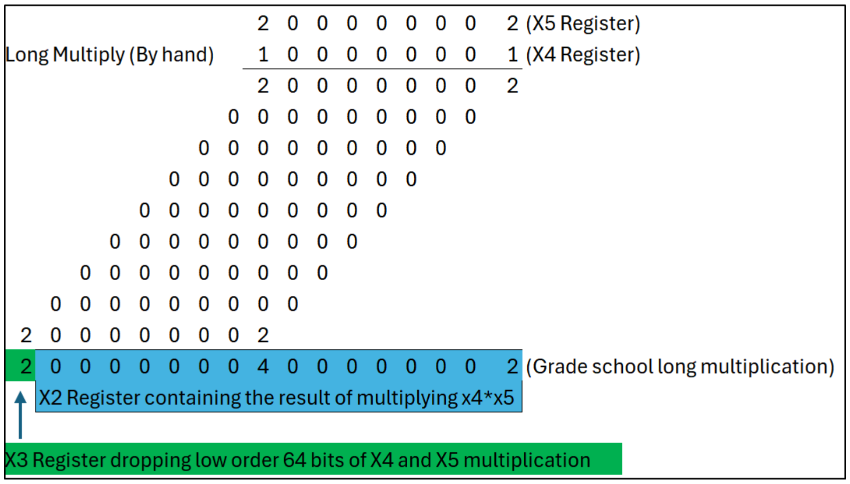
```
/* listing4-23

This example shows how to multiply two 64-bit numbers, placing the 128-bit result
in two 64-bit registers.

MUL is used for the lower 64 bits and UMULH is used for the higher 64 bits */

.text

.global _start

_start:

       mov x4, #0x1  // Load up first 16 bits to reg x4

       movk x4,#0x0000,lsl #16    // Load up next set of 16 bits

       movk x4,#0x0001,lsl #32 // Next 16 bits

       movk x4,#0x0000,lsl #48 // Last 16 bits

       mov x5, #0x2         // Now load reg x5

       movk x5, #0x0000,lsl #16

       movk x5, #0x0002,lsl #32

       movk x5, #0x0000, lsl #48

       MUL  x2, x4, x5     /*Multiplies the contents of x4 and x5 together,
placing the 64-bit result in reg x2*/

UMULH x3, x4, x5 /*Multiplies the contents of x4 and x5 together, placing the
higher unsigned 64-bit result (64:127) result in reg x3, discarding lower 64 bits
(0:63)*/

       mov    w8, #0x5d     //Time to go.

       svc    #0
```



```
ister group: general
               0x0             0                              x1      0x0
               0x400000002     17179869186                    x3      0x2
               0x100000001     4294967297                     x5      0x20
               0x0             0                              x7      0x0
               d               93                                     0x0
                               0                                      0x0
                               0                              x13     0x0
               0x0             0                              x15     0x0
               0x0             0                              x17     0x0
               0x0             0                              x19     0x0
               0x0             0                              x21     0x0
               0x0             0                              x23     0x0
               0x0             0                              x25     0x0
               0x0             0                              x27     0x0
               0x0             0                              x29     0x0
               0x0             0                              sp      0x7f
```

2 (bits 0:63)
x3 (only retaining bits 64:127
Result is
0x00000000000000020000000400000002

Figure 4-1 64-bit multiplication (verified by hand)



```
                                  2  0  0  0  0  0  0  0   2 (X5 Register)
Long Multiply (By hand)           1  0  0  0  0  0  0  0   1 (X4 Register)
                                  2  0  0  0  0  0  0  0   2
                               0  0  0  0  0  0  0  0  0
                            0  0  0  0  0  0  0  0  0
                         0  0  0  0  0  0  0  0  0
                      0  0  0  0  0  0  0  0  0
                   0  0  0  0  0  0  0  0  0
                0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0
          2  0  0  0  0  0  0  0  2
          2  0  0  0  0  0  0  0  4  0  0  0  0  0  0  0   2 (Grade school long multiplication)
             X2 Register containing the result of multiplying x4*x5

X3 Register dropping low order 64 bits of X4 and X5 multiplication
```

Disassembly shows:

```
$ objdump -d -M no-aliases mult8

mult8:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000400078 <_start>:

  400078:     d2800024     movz   x4, #0x1

  40007c:     f2a00004     movk   x4, #0x0, lsl #16

  400080:     f2c00024     movk   x4, #0x1, lsl #32

  400084:     f2e00004     movk   x4, #0x0, lsl #48

  400088:     d2800045     movz   x5, #0x2

  40008c:     f2a00005     movk   x5, #0x0, lsl #16

  400090:     f2c00045     movk   x5, #0x2, lsl #32

  400094:     f2e00005     movk   x5, #0x0, lsl #48
  400098:     9b057c82      madd   x2, x4, x5, xzr
  40009c:     9bc57c83      umulh  x3, x4, x5
  4000a0:     52800ba8      movz   w8, #0x5d
  4000a4:     d4000001      svc    #0x0
```

MSUB and MNEG

`MNEG` (Multiply-Negate) is an alias of `MSUB`. The format is `MSUB Xd, Xn, Xm, Xa`, where Xd is the 64bit destination register, Xm is the first operand (multiplicand) , Xn is the second operand (multiplier) , Xa is the third operand holding the minuend. The operation multiplies Xm and Xn, then subtracts the product from the third operand register.

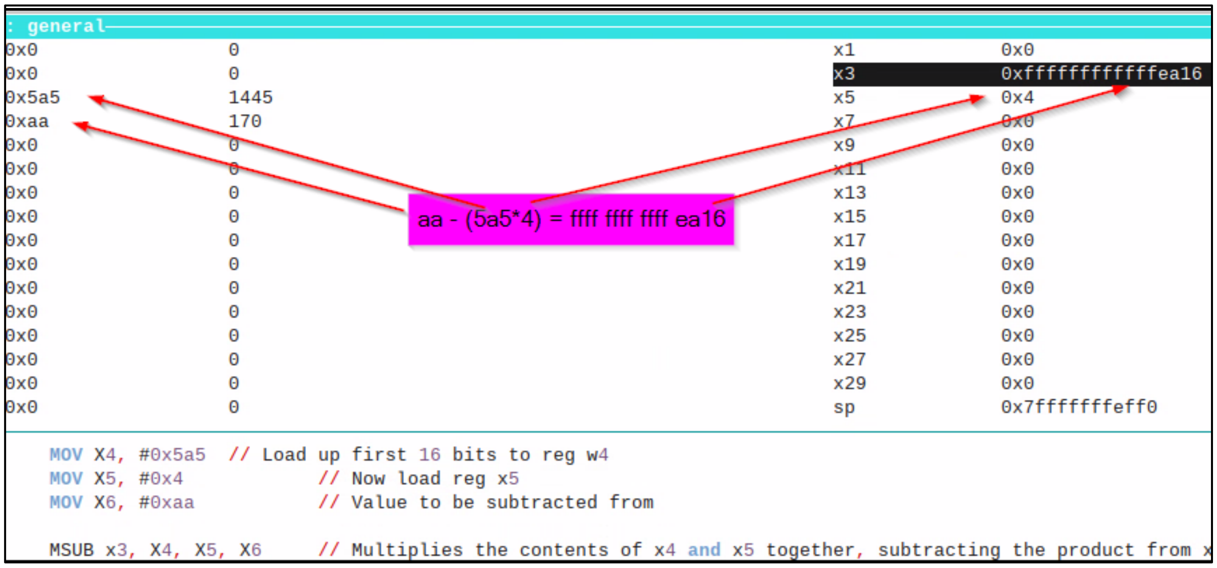
Listing 4-24 Use of MSUB

```
//listing4-25
// This example illustrates the MSUB instruction which multiplies two operands and
then subtracts the product from a third operand.
.text
.global _start
_start:
      mov x4, #0x5a5      // Load up first 16 bits to reg w4
      mov x5, #0x4        // Now load reg x5
      mov x6, #0xaa       // Value to be subtracted from

      MSUB x3, x4, x5, x6     // Multiplies the contents of x4 and x5 together,
subtracting the product from x6
      mov   w8, #0x5d     //Time to go.
      svc    #0
```

Dissassembly

```
$ objdump -d -M no-aliases mult11
mult11:     file format elf64-littleaarch64
Disassembly of section .text:

0000000000400078 <_start>:
  400078:     d280b4a4      movz   x4, #0x5a5
  40007c:     d2800085      movz   x5, #0x4
  400080:     d2801546      movz   x6, #0xaa
  400084:     9b059883      msub   x3, x4, x5, x6

  400088:     52800ba8      movz   w8, #0x5d

  40008c:     d4000001      svc    #0x0
```

When MNEG is used it is equivalent to using XZR as the third operand, so it negates the product.



```
: general
0x0               0                                        x1          0x0
0x0               0                                        x3          0xffffffffffffea16
0x5a5             1445                                     x5          0x4
0xaa              170                                      x7          0x0
0x0               0                                        x9          0x0
0x0               0                                        x11         0x0
0x0               0                                        x13         0x0
0x0               0                                        x15         0x0
0x0               0                                        x17         0x0
0x0               0                                        x19         0x0
0x0               0                                        x21         0x0
0x0               0                                        x23         0x0
0x0               0                                        x25         0x0
0x0               0                                        x27         0x0
0x0               0                                        x29         0x0
0x0               0                                        sp          0x7ffffffeff0

    MOV X4, #0x5a5  // Load up first 16 bits to reg w4
    MOV X5, #0x4            // Now load reg x5
    MOV X6, #0xaa           // Value to be subtracted from

    MSUB x3, X4, X5, X6     // Multiplies the contents of x4 and x5 together, subtracting the product from x
```

Listing 4-25 Use of MNEG

```
//listing4-25

// This example illustrates the MNEG instruction which multiplies two operands and then subtracts the product from XZR.

.text

.global _start

_start:

        mov x4, #0x5a5              // Load up reg w4

        mov x5, #0x4        // Now load reg x5

        mov x6, #0xaa       // Value to be subtracted from

        MNEG x3, x4, x5             // Multiplies the contents of x4 and x5 together, subtracting the product from 0 (XZR)

        mov   w8, #0x5d             //Time to go.

        svc   #0

$ objdump -d -M no-aliases mult11


mult11:     file format elf64-littleaarch64

Disassembly of section .text:
```

```
0000000000400078 <_start>:
  400078:     d280b4a4      movz   x4, #0x5a5
  40007c:     d2800085      movz   x5, #0x4
  400080:     d2801546      movz   x6, #0xaa
  400084:     9b05fc83      msub   x3, x4, x5, xzr
  400088:     52800ba8      movz   w8, #0x5d
  40008c:     d4000001      svc    #0x0
```

UMNEGL and SMNEGL

These instructions multiply two 32-bit (w) registers, negate the product placing the result in a 64-bit (X) register. UMNEGL and SMNEGL are aliases for UNSUBL and SMSUBL, respectively.

Division

Division operations can be *signed* or *unsigned* using the instructions `UDIV` and `SDIV`. The format is SDIV|UDIV Rd, Rn, Rm where Rd is the destination, Rn contains the numerator and Rm contains the denominator. Registers can be 32-bit or 64-bit.

Note dividing by zero does not give an error, it returns the value 0, so it needs to be tested separately.

Listing 4-26 Using UDIV

```
// listing4-26
// This example illustrates the UDIV instruction which uses two operands as the
numerator and denominator.
.text
.global _start
_start:
      mov x4, #2000              // Load up reg x4 (number to be divided)
      mov x5, #0x4        // Now load reg x5 (number that will divide)

      UDIV x3, x4, x5            // Divides x4 by x5 together, result goes into x3

      mov x4, #1999       // x4 no longer evenly divisble by contents of x5
      UDIV x3, x4, x5            // No remainder recorded
      mov x5, #0          // Dividing by zero does not error, but returns zero
      UDIV x3, x4, x5
      mov    w8, #0x5d              //Time to go.
      svc     #0
```

Note there is no provision made for recording the remainder, this needs to be calculated separately.

In the second part of Listing 4-26 where x4 contains 1999, the remainder is calculated by subtracting the product of x3 and x5 from x4:

- x3 = 499
- x4 = 1999
- x5 = 4
    - Remainder = 1999 – (499*4) = 1999 – 1996 = **3**.

Shift and Rotate

Some of the listings have used shift/rotate instruction already but this section will formally introduce them. The instructions and their descriptions are shown in Table 4-3

Table 4-3 Rotate and shift instructions

Operation	Example	Description
Logical Shift Left	lsl rd, rn, #shift	Shift bits left by specified amount, zeros move in from the right, can be immediate or register
Logical Shift Right	lsr Rd, Rn, #shift	Shift bits right by specified amount, zeros move in from the left, can be immediate or register
Arithmetic Shift Right	asr rd, rn, #shift	Shift bits right by specified amount, maintaining the sign bit. Use for signed integers, can be immediate or register.
Rotate Right	ror rd, rn, rm	Rotate right in that the bit shifted from bit0 moves into the most significant bit, can be immediate or register

Examples are shown in Listing 4-27.

Listing 4-27 Examples of Shift and Rotate instructions

```
// listing4-27
.text
.global _start
_start:
```

```
mov w0, #0xaaaa

mov w1, #0x33333333

mov w2, #0x44444444

mov w3, #0x55555555

mov w4, #0x66666666


lsl w5, w1, #3      // w5 = 0x99999998

lsl w6, w1, w0      // w6 = 0xcccccc00

asr w7, w2, #3      // w7 = 0x88888888

ror w3, w3, #5      // w3 = 0xaaaaaaaa

mov w8, #93 //Time to go

svc 0
```

Disassembly shows the non-aliased form of the instructions.

```
rotate:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000400078 <_start>:

  400078:    52955540     movz   w0, #0xaaaa

  40007c:    3200e7e1     orr    w1, wzr, #0x33333333

  400080:    3202e3e2     orr    w2, wzr, #0x44444444

  400084:    3200f3e3     orr    w3, wzr, #0x55555555

  400088:    3203e7e4     orr    w4, wzr, #0x66666666

  40008c:    531d7025     ubfm   w5, w1, #29, #28

  400090:    1ac02026     lslv   w6, w1, w0

  400094:    13037c47     sbfm   w7, w2, #3, #31

  400098:    13831463     extr   w3, w3, w3, #5

  40009c:    52800ba8     movz   w8, #0x5d

  4000a0:    d4000001     svc    #0x0
```

The instruction results are straightforward except for `lsl w6, w1, w0`. Here register w0, which holds the value 0xaaaa, is divided by the data size which in this case is 32. The remainder of the division is used to specify the rotation. The remainder is 0xa, so the rotation will be applied 10 times.

												1	0	1	0	1	0	1	0	1	0	1
1	0	0	0	0	0	1	0	1	0	1		0	1	0	1	0	1	0	1	0	1	0
						1	0	0	0	0		0										
						0	0	1	0	1		0	1	0								
								1	0	0		0	0	0								
								0	0	1		0	1	0	1	0						
										1		0	0	0	0	0						
										0		0	1	0	1	0	1	0				
													1	0	0	0	0	0				
													0	0	1	0	1	0	1	0		
															1	0	0	0	0	0		
															0	0	1	0	1	0	1	0
																	1	0	0	0	0	0
							Shift by					Remainder =							1	0	1	0

Rotate by 10 positions to get 0xcccccc00

0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
c				c				c				c				c				c				0				0			

Logic Operations – AND/OR/EOR

Truth tables for AND and OR operations are shown in Table 1-11 and Table 1-12.

- To test whether a bit is a one or a zero, the bit can be AND'ed with a binary one. If the result of the AND is a one then the tested bit is also one, since this is the only AND operation that will generate a binary one, otherwise it has the value zero.
- Similarly, if a bit is OR'ed with a zero and the result is a zero then the tested bit is also zero since the OR operation only produces a zero when both bits are zero.

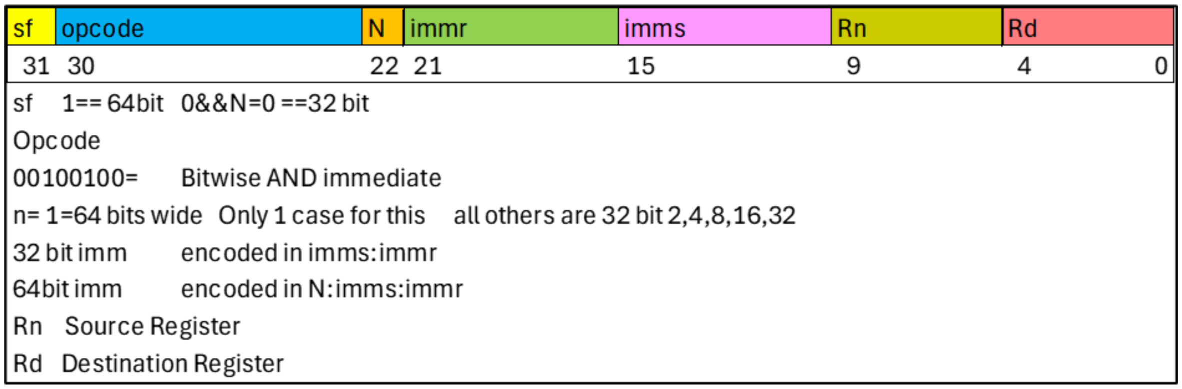
To summarize:

- 1 AND X = 1 iff[20] X=1
- 0 OR X = 0 iff X=0

Multiple bits can be cleared or set by the use of a *bitmask*.

The format of the Bitwise AND (immediate) instruction is shown in Figure 4-2. The immediate data is 12 bits in size, limiting the size of the bitmask for OR/AND instructions. There is though, a form of immediate termed *logical immediate* that provides for larger bitmask sizes. The approach is to provide a pattern with some compromises on the data that can be represented.

Figure 4-2 Format of AND (immediate) instruction



sf	opcode	N	immr	imms	Rn	Rd
31	30	22	21	15	9	4 0

sf 1== 64bit 0&&N=0 ==32 bit

Opcode

00100100= Bitwise AND immediate

n= 1=64 bits wide Only 1 case for this all others are 32 bit 2,4,8,16,32

32 bit imm encoded in imms:immr

64bit imm encoded in N:imms:immr

Rn Source Register

Rd Destination Register

The ARM architecture reference manual[21] states:

[20] Iff – if and only if!

[21] Text may be version dependent.

"Logical (immediate)

The Logical (immediate) instructions accept a bitmask immediate value that is a 32-bit pattern or a 64-bit pattern viewed as a vector of identical elements of size e = 2, 4, 8, 16, 32 or, 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to (e - 1) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to (e - 1) bits. This mechanism can generate 5334 unique 64-bit patterns as 2667 pairs of pattern and their bitwise inverse."

This means that there are repeated patterns of bits with varying sizes. The elements can be made up of 2 bits, 4 bits, 8 bits, 16 bits, 32 bits or 64 bits.

The two-bit element contains 32 x 1 bit sub-patterns which looks like:

01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01

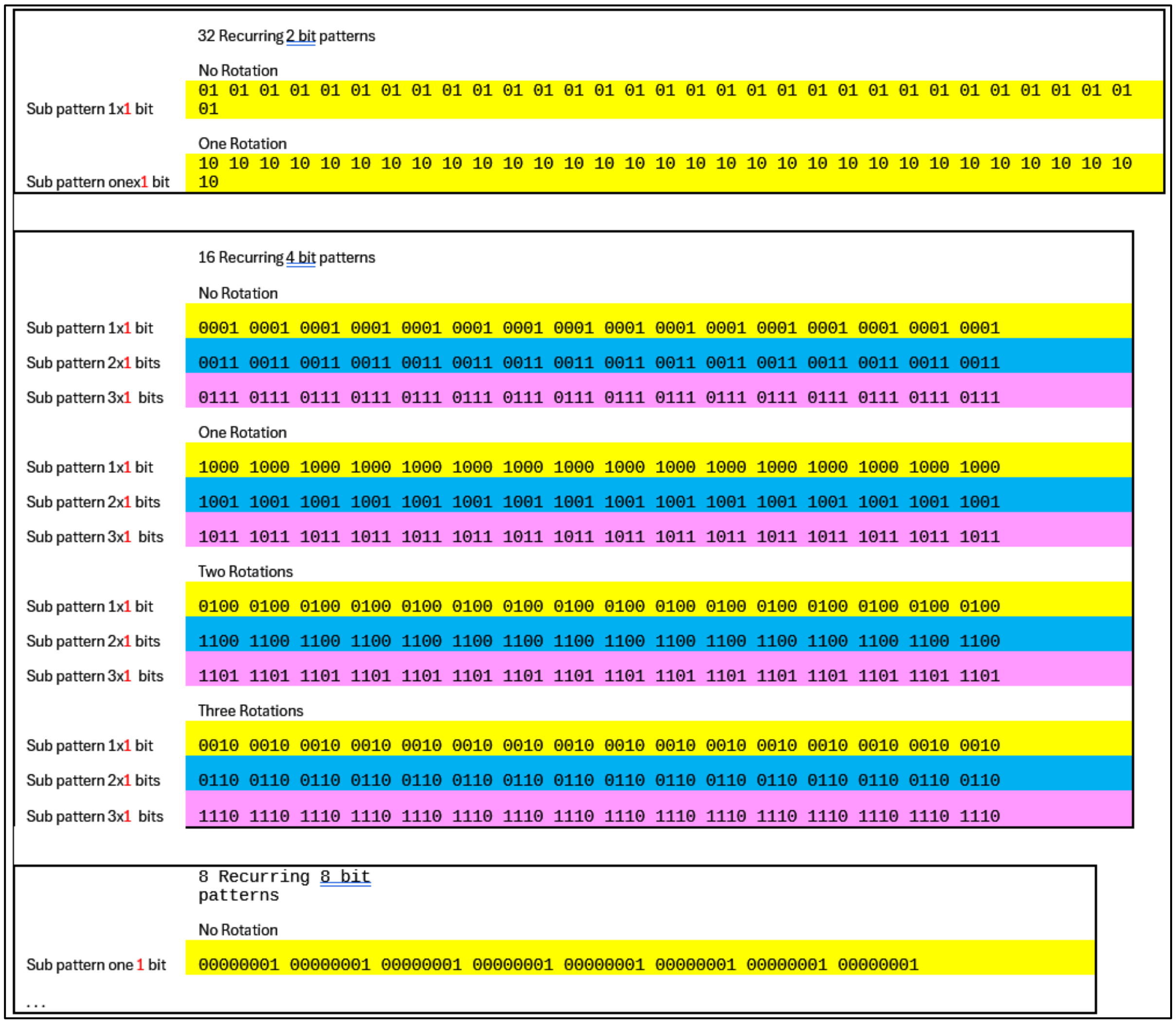
There is only one possible (base) pattern since the rule states that the sub pattern is a single run of 1 to e-1 nonzero bits, and since size element e is 2, then the run is 1 to (2-1) = 1. The pattern starts with a 1 at the bit zero position followed by zero bits. The pattern can be right rotated, however giving a second sub-pattern as shown:

1 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 010

Other patterns give a wider range, for example when e = size 4, there can be 1 to 3 ones (1 to e-1) in the sub pattern. A subset of patterns is shown in Figure 4-3.

Another way of using large bitmasks is to use a series of `movk` instructions into a register and then use this register (with an optional shift) to perform the logic operation. This is often preferred than the use of logical immediate instructions. Examples of logical operations using registers are covered later in this chapter.

Figure 4-3 Examples of Logical immediate values



	32 Recurring 2 bit patterns
	No Rotation
Sub pattern 1x1 bit	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
	One Rotation
Sub pattern onex1 bit	10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10

	16 Recurring 4 bit patterns
	No Rotation
Sub pattern 1x1 bit	0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001 0001
Sub pattern 2x1 bits	0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011 0011
Sub pattern 3x1 bits	0111 0111 0111 0111 0111 0111 0111 0111 0111 0111 0111 0111 0111 0111 0111 0111
	One Rotation
Sub pattern 1x1 bit	1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
Sub pattern 2x1 bits	1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001
Sub pattern 3x1 bits	1011 1011 1011 1011 1011 1011 1011 1011 1011 1011 1011 1011 1011 1011 1011 1011
	Two Rotations
Sub pattern 1x1 bit	0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100
Sub pattern 2x1 bits	1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100
Sub pattern 3x1 bits	1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101 1101
	Three Rotations
Sub pattern 1x1 bit	0010 0010 0010 0010 0010 0010 0010 0010 0010 0010 0010 0010 0010 0010 0010 0010
Sub pattern 2x1 bits	0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110
Sub pattern 3x1 bits	1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1110

	8 Recurring 8 bit patterns
	No Rotation
Sub pattern one 1 bit	00000001 00000001 00000001 00000001 00000001 00000001 00000001 00000001
...	

Note that the patterns are identical as stated in the ARM documents, therefore a pattern such as 0x0fff0fff0fff0fff0 would be valid (recurring consecutive ones) but 0xfff0fff0fff00fff would not.

The actual encoding for the bitwise AND instruction is shown in Table 4-4 below.

The logical immediate is made up of the *N*, *immr* and *imms* fields.

- The single bit N field is set to 1 if the element size is 64 bits.
- The 6-bit immr field specifies the rotation amount and since there are 6 bits then 0-63 rotations are possible.

- The 6-bit imms field (in conjunction with the N bit is used to specify the element size and the sub patterns. Table 4-4shows examples of the patterns generated by the imms field bits.

Table 4-4 imms field examples

N	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Sub pattern
0	1	1	1	1	0	0	01 01 01 . . .
32-bit	2-bit element size One “1”						
0	1	1	1	0	0	0	0001 0001 . . .
32-bit	4-bit element size One “1”						
0	1	1	1	0	0	1	0011 0011 . . .
32-bit	4-bit element size Two “1s”						
0	1	1	1	0	1	0	0111 0111 . . .
	4-bit element size Three “1s”						
0	1	1	0	0	0	0	00000001 00000001 . . .
32-bit	8-bit element size One “1”						
0	1	1	0	0	0	1	00000011 00000011 . . .
32-bit	8-bit element size Two “1s”						
0	1	1	0	0	1	0	00000111 00000111 . . .
32-bit	8-bit element size Three “1s”						
0	1	1	0	0	1	1	00001111 00001111 . . .
32-bit	8-bit element size Four “1s”						
0	1	0	1	0	0	1	0000001111111111 . . .
32-bit	16-bit element size Ten “1s”						
0	0	0	1	1	1	1	
	32-bit element size sixteen “1’s”						00000000000000001111111111111111 . . .
1	0	0	0	0	0	1	
	64-bit element size two “1s”						0000000000000000000000000000000000000000000000000000000000000011

The relevant bits for the element size are shown in Table 4-5. The bits that do not correspond to the element sizes are used for positioning the 1's values. Listing 4-29 shows the results of using the values shown in Figure 4-3.

Table 4-5 Interpreting the imms field bits

Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Element size
1	1	1	1	0	-	2
1	1	1	0	-	-	4
1	1	0	-	-	-	8
1	0	-	-	-	-	16
0	-	-	-	-	-	32
-	-	-	-	-	-	64[22]

The listing below shows the `orr` instruction in operation

Listing 4-28 Use of the orr and ORN instructions

```
//listing 4-28
.text
.global _start
_start:
// orr (Bitwise OR immediate)
mov w0, #0xaaaaaaaa
orr w1, w0, #0x55555555    // w1=0xffffffff
orr w1, w0, #0xaaaaaaaa    // w1=0xaaaaaaaa
orr w1, w0, #0x0000ffff    // w1=0xaaaaffff
orr w1, w0, #0xffff0000    // w1=0xffffaaaa

//orr (Bitwise OR shifted register)
mov w0,#3
orr w1, w0, w0, lsl #6     // w1 = 0xc3 = 0b11000011
```

[22] In this case N = 1

```
mov w0, #9
orr w1, w0, w0, lsl #8     // w1 = 0x909 = 0b100100001001


//ORN (Bitwise OR NOT shifted register)
mov x0, #0x1122
ORN x1, x0, x0, lsl #8     //x1 = 0xfffffffffffeeddff, note 0x1122 gets inverted
mov w8, #93
svc 0
```

Note the disassembly aliases –

```
Orr wo, wzr, #aaaaaaaa = mov w0, #0xaaaaaaaa
```

...

```
$ objdump -d -M no-aliases orr
orr:     file format elf64-littleaarch64
Disassembly of section .text:
0000000000400078 <_start>:
  400078:    3201f3e0     orr    w0, wzr, #0xaaaaaaaa
  40007c:    3200f001     orr    w1, w0, #0x55555555
  400080:    3201f001     orr    w1, w0, #0xaaaaaaaa
  400084:    32003c01     orr    w1, w0, #0xffff
  400088:    32103c01     orr    w1, w0, #0xffff0000
  40008c:    52800060     movz   w0, #0x3
  400090:    2a001801     orr    w1, w0, w0, lsl #6
  400094:    52800120     movz   w0, #0x9
  400098:    2a002001     orr    w1, w0, w0, lsl #8
  40009c:    d2822440     movz   x0, #0x1122
  4000a0:    aa202001     orn    x1, x0, x0, lsl #8
  4000a4:    52800ba8     movz   w8, #0x5d
  4000a8:    d4000001     svc    #0x0
```

Listing 4-29 Using logical immediates with and/orr instructions

```
$ //listing4-29
.text
.global _start
_start:
```

```
//  and (Bitwise and immediate)

mov x4, #0xffff

movk x4, #0x0000, lsl 16

movk x4, #0x0000, lsl 32

movk x4, #0xfff, lsl 48

// Use objdump to see encoding for logical immediates

// Format is N, immr, imms. If N = 1 then pattern is 64 bit

// If N = 0 then pattern is 32 bit repeating n times, where n is specified in the
imms field

// imms = 11110X then 2 bit pattern recurring 32 times, one "1",

// imms = 1110xx then 4 bit pattern recurring 16 times, one thru three "1s"

// imms = 110xxx then 8 bit pattern recurring 8 times, one thru seven "1s"

// imms = 10xxxx then 16 bit pattern recurring 4 times, one thru 15 "1s"

// imms - 0xxxxx then 32 bit pattern recurring 2 times, one thru 31 "1s"

// x field = # of ones, where the # of ones is one less than the x value,00 = 1
"1", 01 = 2 "1s", 10 = 3 ", . . .

// so for imms = 111001 pattern is 4 bits and the # of ones is 2 = 0011 0011 0011
. . .

// for imms = 110110 pattern is 8 bits and # of ones is 7 = 01111111 01111111
01111111 . . .

// immr is the rotate field where 000000 = no rotation and 111111 =sixty-three
rotations

// Note for imms = 11110x then 1 rotation is possible

// Note for imms = 110xxx then 1 thru 7 rotations are possible

// Examples follow

and x3, x4, #0x5555555555555555 // r3 = 555000000005555

// Pattern is 2 bits wide imms, = 111100, one sequential one, immr = 000000, no
rotate

and x3, x4, #0xaaaaaaaaaaaaaaaa, // r3 = 0xaaa00000000aaaa

// Pattern is 2 bits wide imms, = 111100, one sequential one, immr = 000001, one
rotate


and x3, x4, #0x1111111111111111 // r3 = 0x1110000000001111

// Pattern is 4 bits wide, imms = 111000, one sequential one, immr = 000000, no
rotates

and x3, x4, #0x3333333333333333 // r3 = 0x3330000000003333
```

```
// Pattern is 4 bits wide, imms = 111001, two sequential ones, immr = 00000, no
rotate

and x3, x4, #0x7777777777777777  // r3 = 0x7770000000007777

// Pattern is 4 bits wide, imms = 111010, three sequential ones, immr = 000000, no
rotate

and x3, x4, #0x8888888888888888  // r3 = 0x8880000000008888

// Pattern is 4 bits wide imms = 111000, one sequential one, immr = 000001, one
rotate

and x3, x4, #0x9999999999999999  // r3 = 0x9990000000009999

// Pattern is 4 bits wide imms = 111001, two sequential ones, immr = 000001, one
rotate

and x3, x4, #0xbbbbbbbbbbbbbbbb // r3 = 0xbbb000000000bbbb

// Pattern is 4 bits wide,imms = 111010, three sequential ones, immr = 000001, one
rotate

orr x3, x4, #0x4444444444444444 // r3 = 0x4fff44444444ffff

// Pattern is 4 bits wide, imms = 111000, one sequential one, immr = 000010, two
rotates

orr x3, x4, #0xcccccccccccccccc // r3 = 0xcfffccccccccffff

// Pattern is 4 bits wide, imms = 111001, two sequential ones, immr = 000010, two
rotates

orr x3, x4, #0xdddddddddddddddd // r3 = 0xdfffddddddddffff

// Pattern is 4 bits wide, imms = 111010, three sequential ones, immr = 000010,
two rotates

orr x3, x4, #0x2222222222222222 // r3 = 0x2fff22222222ffff

// Pattern is 4 bits wide, imms = 111000, one sequential one, immr = 000011, three
rotates

orr x3, x4, #0x6666666666666666 // r3 = 0x6fff66666666ffff

// Pattern is 4 bits wide, imms = 111001, two sequential ones, immr = 000111,
three rotates

orr x3, x4, #0xeeeeeeeeeeeeeeee // r3 = 0xefffeeeeeeeeffff

// Pattern is 4 bits wide, imms = 111010, three sequential ones, immr = 000111,
three rotates

orr x3, x4, #0x0101010101010101 // r3 = 0xfff01010101ffff

// Pattern is 8 bits wide, imms = 110000, one sequential one, immr = 000000, no
rotate

orr x3, x4, #0b0110011001100110011001100110011001100110011001100110011001100
//r3 = 0xcfffccccccccffff
```

```
//Same as orr x3,x4, #0x4444444444444 but expressed in binary (often easier when
working with bitmasks)

and x3, x4, #0x0000000000000001 // r3 = 0x1

// N=1, One 64 -bit pattern of one one, imms = 000000, immr = 000000, no rotation

and x3, x4, #0x1000000000000000 // r3 = 0x0

// N=1, One 64-bit pattern of one one, imms = 000000, immr = 000100, four
rotations (0001 - 0001 . . .)

mov w8, #93 //Time to go

svc 0
```

The imms and immr fields can be shown from the disassembly :

```
$ objdump -d -M no-aliases examples

examples:     file format elf64-littleaarch64

Disassembly of section .text:

0000000000400078 <_start>:

  400078:     d29fffe4      movz   x4, #0xffff

  40007c:     f2a00004      movk   x4, #0x0, lsl #16

  400080:     f2c00004      movk   x4, #0x0, lsl #32

  400084:     f2e1ffe4      movk   x4, #0xfff, lsl #48

  400088:     9200f083      and    x3, x4, #0x5555555555555555

  40008c:     9201f083      and    x3, x4, #0xaaaaaaaaaaaaaaaa

  400090:     9200e083      and    x3, x4, #0x1111111111111111

  400094:     9200e483      and    x3, x4, #0x3333333333333333

  400098:     9200e883      and    x3, x4, #0x7777777777777777

  40009c:     9201e083      and    x3, x4, #0x8888888888888888

  4000a0:     9201e483      and    x3, x4, #0x9999999999999999

  4000a4:     9201e883      and    x3, x4, #0xbbbbbbbbbbbbbbbb

  4000a8:     b202e083      orr    x3, x4, #0x4444444444444444

  4000ac:     b202e483      orr    x3, x4, #0xcccccccccccccccc

  4000b0:     b202e883      orr    x3, x4, #0xdddddddddddddddd

  4000b4:     b203e083      orr    x3, x4, #0x2222222222222222

  4000b8:     b203e483      orr    x3, x4, #0x6666666666666666

  4000bc:     b203e883      orr    x3, x4, #0xeeeeeeeeeeeeeeee

  4000c0:     b200c083      orr    x3, x4, #0x101010101010101
  4000c4:     b202e483      orr    x3, x4, #0xcccccccccccccccc
  4000c8:     92400083      and    x3, x4, #0x1
  4000cc:     92440083      and    x3, x4, #0x1000000000000000
  4000d0:     52800ba8      movz   w8, #0x5d
  4000d4:     d4000001      svc    #0x0
```

Exercise – Try using a logical immediate with a mov instruction.

and shifted register instruction

The `and` shifted instruction and's two registers together. Placing the result in the destination register. The second register can also have an optional shift applied to it prior to the AND operation. The format is:

```
and <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

ANDS instructions

The ands instruction is used to set flags.

orr shifted register instruction

The orr shifted instruction OR's two registers together. Placing the result in the destination register. The second register can also have an optional shift applied to it prior to the orr operation. The format is:

```
orr <Xd>, <Xn>, <Xm>, <shift> #<amount>
```

ORN bitwise shifted register

The `orn` bitwise is similar to the and and orr bitwise shifted register instructions, except that it inverts the bits in the second register prior to applying the ORN instruction

EOR instructions

Exclusive or instructions can be used as bitwise immediate or in shifted register forms.

Listing 4-30 shows more logical instruction examples.

Listing 4-30 Example of logical instruction with shifted register operands

```
//listing4-30
.text
.global _start
_start:
//  and (Bitwise and immediate)
mov w4, #0xcccccccc
mov w5, #0x55555555
// Examples follow
and w3, w4, w5 // w3 = 0x44444444
orr w3, w4, w5 // w3 = 0xdddddddd
and w3, w4, w5, lsl 4 // w3 = 0x44444440
orr w3, w4, w5, lsr 2 // w3 = 0xdddddddd
orn w3, w4, w5, asr 1 // w3 = dddddddddd
eor w3, w4, w5, ror 6 // w3 = 0x99999999
mov w8, #93 //Time to go
svc 0
```

BIC and BFI instructions

Bitwise bit clear `BIC` can clear bits by executing the AND instruction with the inverse of the contents of an optionally shifted register. The format is:

```
BIC Xd, Xn, Xm, shift type  amount
```

The bitfield insert (`BFI`) instruction copies a set of bits (from the least significant positions) specified in the width field in the source register to a bit position (specified in the lsb field) in the target register. `BFI` is an alias for `BFM` (bitfield move).

Examples are shown in Listing 4-31.

Listing 4-31 BIC and BFI instructions

```
//listing4-31
//  BIC instruction
mov x3, #0x5555
mov x4, #0x6666666666666666
bic x5, x3, x4, lsl 3  // x5 = 0x4445
bfi x5, x4, 6,5     // x5 = 0x4185
mov w8, #93 //Time to go
svc 0
```

The instruction format and steps are shown in Figure 4-3.

The disassembly shows –

```
objdump -d -M no-aliases bic
bic:     file format elf64-littleaarch64
```

```
Disassembly of section .text:

0000000000400078 <_start>:

  400078:    d28aaaa3     movz   x3, #0x5555

  40007c:    b203e7e4     orr    x4, xzr, #0x6666666666666666

  400080:    8a240c65     bic    x5, x3, x4, lsl #3

  400084:    b37a1086     bfm    x6, x4, #58, #4

  400088:    52800ba8     movz   w8, #0x5d

  40008c:    d4000001     svc    #0x0
```

Summary of chapter 4

- Arithmetic operations
    - Add, Subtract, Multiply and Divide
- Logical Operations
    - Bitwise operators
- Shift and Rotate instructions
- Logical Immediate instructions
- Condition flags

Exercises for chapter4

1. Describe the difference between the add and ADDs instruction?
2. After executing the following code what value ends up in register x6

```
mov x4, #1024 //This moves the number 1024 to reg x4

mov x5,#64    // Move 64 into reg x5

add x6, x4, x5, LSL #6
```

3. What is the value in w3 after the code below has been executed.

```
//  and (Bitwise and immediate)

mov w4, #0xcccccccc

mov w5, #0x55555555

and w3, w4, w5
```

4. Describe two ways to place the immediate value 0xbbbbbbbb into register X6

Chapter 5. Loops, Branches and Conditions

This chapter will show how to use iteration and decision making with ARM64 assembly code. The next listing shows how to compare two numbers and will print out an appropriate message.

Listing 5-1 Simple comparison and branch example

```
//listing5-1
// This example shows how basic comparison and branch instructions work
.text
.global _start
_start:
        mov x4, #0X8000 // Load up reg x4
        mov X5, #0x4000 // Now load reg x5
        cmp x4, x5
        bgt printlower
        ldr x1, =lower  // Point x1 to lower string location
        mov x2, #22     // Length of lower string
        B printit
printlower:
        ldr x1, =upper  // Point x1 to upper string location
        mov x2, #23     // Length of upper string
printit:
        mov x0, #1
        mov w8, #64     // Invoke the Write system call
        svc #0
        mov w8, #0x5d   // Time to go.
        svc #0
.data
        lower: .ascii "First number is lower\n"
.align
        upper: .ascii "First number is higher\n".
Output: - First number is higher
```

This snippet compares two numbers held in register x4 and x5. It does a comparison and if the second number is lower than the first number then it branches to the code located at the label **printlower**. The instruction `bgt` is a *conditiona*l branch. If the number is not greater, then the write parameters are set up (string location and its length) and the code performs an unconditional branch to the label **printit**, skipping over the **printlower** code. Regardless of the comparison the code at **printit** is common and its function is to invoke the Write System call and then exit.

Disassembling the code is instructive:

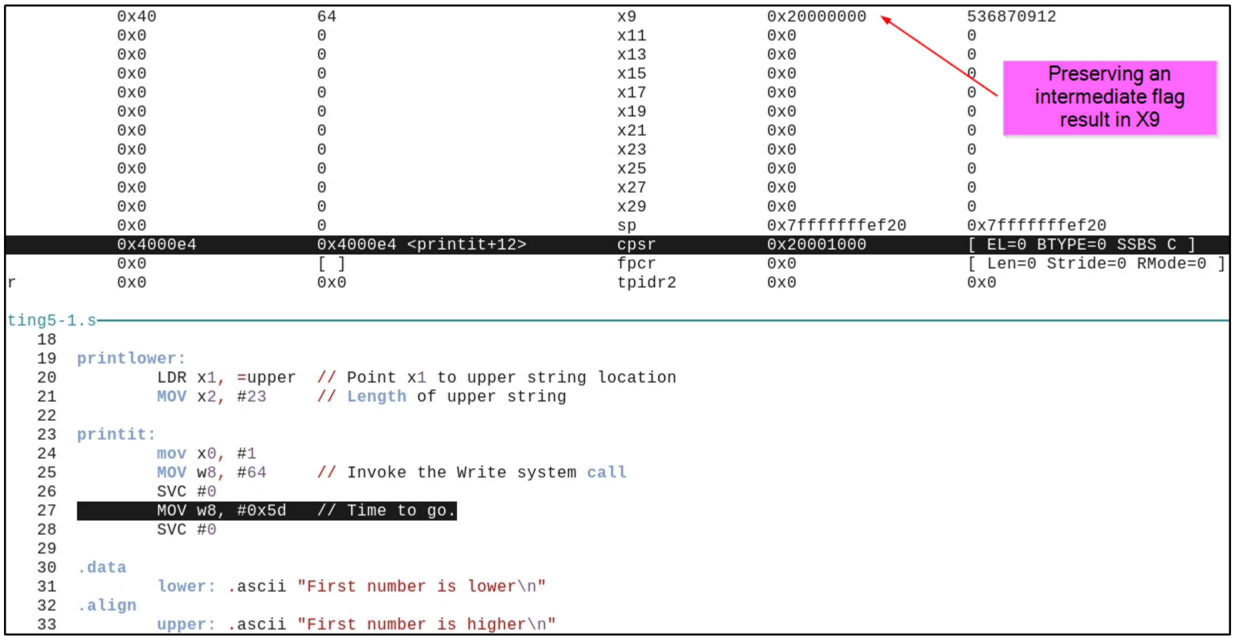
```
objdump -d -M no-aliases cmp1

cmp1:     file format elf64-littleaarch64

Disassembly of section .text:

00000000004000b0 &lt;_start&gt;:

  4000b0:     d2900004      movz   x4, #0x8000

  4000b4:     d2880005      movz   x5, #0x4000

  4000b8:     eb05009f      subs   xzr, x4, x5

  4000bc:     5400008c      b.gt   4000cc <printlower>

  4000c0:     58000141      ldr    x1, 4000e8 <printit+0x14>

  4000c4:     d28002c2      movz   x2, #0x16

  4000c8:     14000003      b      4000d4 <printit>

00000000004000cc <printlower>:

  4000cc:     58000121      ldr    x1, 4000f0 <printit+0x1c>

  4000d0:     d28002e2      movz   x2, #0x17

00000000004000d4 <printit>:

  4000d4:     d2800020      movz   x0, #0x1

  4000d8:     52800808      movz   w8, #0x40

  4000dc:     d4000001      svc    #0x0

  4000e0:     52800ba8      movz   w8, #0x5d

  4000e4:     d4000001      svc    #0x0

  4000e8:     004100f8      .word  0x004100f8

  4000ec:     00000000      .word  0x00000000

  4000f0:     0041010e      .word  0x0041010e

  4000f4:     00000000      .word  0x00000000./cmp1
```

Note that the `cmp` instruction is an alias of `SUBS` which performs a subtraction; using the xzr register to discard the result.

Adding in the command `MRS` X9, NZCV shows which flags are set, in the case where the value of X4 is higher than the value in x5 then the **C**arry bit is set (X9=0x20000000), in the second case where the value of X4 is lower than the value of x5 then the **N**egative bit is set (x9=0x80201000).

Case1 (x4 > x5)

Example x4 = 0X8000, x5 = 0x4000 Negative bit is clear, Zero bit is clear, Carry bit is set, Overflow bit is clear



```
           0x40                64                             x9              0x20000000         536870912
           0x0                 0                              x11             0x0                0
           0x0                 0                              x13             0x0                0
           0x0                 0                              x15             0x0                0
           0x0                 0                              x17             0x0                0
           0x0                 0                              x19             0x0                0
           0x0                 0                              x21             0x0                0
           0x0                 0                              x23             0x0                0
           0x0                 0                              x25             0x0                0
           0x0                 0                              x27             0x0                0
           0x0                 0                              x29             0x0                0
           0x0                 0                              sp              0x7ffffffef20      0x7ffffffef20
           0x4000e4            0x4000e4 <printit+12>          cpsr            0x20001000         [ EL=0 BTYPE=0 SSBS C ]
           0x0                 [ ]                            fpcr            0x0                [ Len=0 Stride=0 RMode=0 ]
r          0x0                 0x0                            tpidr2          0x0                0x0

ting5-1.s

printlower:
        LDR x1, =upper  // Point x1 to upper string location
        MOV x2, #23     // Length of upper string

printit:
        mov x0, #1
        MOV w8, #64     // Invoke the Write system call
        SVC #0
        MOV w8, #0x5d   // Time to go.
        SVC #0

.data
        lower: .ascii "First number is lower\n"
.align
        upper: .ascii "First number is higher\n"
```

Case2 (x4 < x5)

Example x4=0X8000, x5=0X9000,Negative bit is set, Zero bit is clear, Carry bit is clear, Overflow bit is clear.

For conditional branches the format is **Branch** on ***condition*** to ***label (B.condition label)*** so in Listing 5-1 the command `bgt printlower` was used with `GT` being the condition <Greater Than> and <printlower> being the label to branch to.

Table 5-1 shows the applicable conditional branches as determined by flag settings.

Table 5-1 Conditional branches

Command	Condition	Flags
B.CS/B.HS	Unsigned greater than or equal to	Carry Set
B.CC/ B.LO	Unsigned less than (lower)	Carry Clear
B.MI	Negative (Minus)	Negative Set
B.PL	Plus (Positive, note zero is positive)	Negative Clear
B.EQ	Equal	Zero set
B.NE	Not equal	Zero clear
B.VS	Overflow set	Overflow set
B.VC	No Overflow	Overflow clear
B.HI	Higher	Carry Set, Zero clear
B.LS	Lower or the same	Carry clear, Zero set
B.GE	Signed greater than or equal to	Negative and Overflow the same
B.LT	Signed less than	Negative and Overflow different
B.GT	Signed greater than	Zero clear, Negative and Overflow the same
B.LE	Signed less than or equal to	Zero set, Negative and Overflow different

From Table 5-1 the listing has been adapted to include Branch if equal (B.EQ)

Listing 5-2 Using B.EQ condition

```
//listing5-2
// This example shows how basic comparison and branch instructions work
.text
.global _start
_start:
      mov x4, #0X8000 // Load up reg x4
      mov x5, #0x8000     // Now load reg x5
      cmp x4, x5
      mrs X9,NZCV   // Get Flags
      b.mi printlower
      b.eq printthesame // Position B.EQ before B.PL since zero is considered
positive
```

```
      b.pl printhigher
printlower:
      ldr x1, =lower      // Point x1 to lower string location
      mov x2, #22   // Length of lower string
      B printit
printhigher:
      ldr x1, =higher // Point x1 to higher string location
      mov x2, #23   // Length of higher string
      B.AL printit
printthesame:
      ldr x1, =same // Point to the same string location
      mov x2, #22
printit:
      mov x0, #1
      mov w8, #64   // Invoke the Write system call
      svc #0
      mov w8, #0x5d // Time to go.
      svc #0
.data
      lower: .ascii "First number is lower\n"
      higher: .ascii "First number is higher\n"
      same:  .ascii "The numbers are equal\n"
```

Nested Loops

Programming often involves iterative algorithms where multiple loops are employed. The next listing shows two loops (inner and outer) and outputs the loop value to the screen as they are being calculated.

Listing 5-3 Nested For loop

```
// listing 5-3
/* This example shows a Nested For Loop in action
Both loops start with an index of 1 and count up to 3
- w3 holds the index value for the inner loop
- w4 holds the index value for the outer loop
- w9 holds the termination loop value

- w5 holds the ASCII equivalent of the current index value

For writing -

w0 = 1 <stdout>

w1 = Character location in memory

w2 = Character count for output to stdout

*/

.text

.global _start

_start:

       mov w9, #4           //ending value for loop

       mov w7, #1           // For loop iteration value

       mov w0, #1 //stdout

       ldr w1, =printheader

       mov w2, #14          // Character count of printheader string

       mov w8, #64          // Write out header text

       svc #0

       ldr w1, =printvalues       // Now that the header has been printed get ready
to print values

       mov w3, #0x1         // Load up reg w3 with starting inner loop value

       mov w4, #0x1         // Load up reg w4 with starting outer loop value

incrementinner:

       add w5, w3, #48            // Convert inner index to ascii

       add w6, w4, #48            // Convert outer index to ascii

       mov w0, #1           // stdout, don't assume x0 is preserved after svc call

       strb w5, [x1]        // Put character into string space

       mov w2, #2           // One character at a time

       mov w8, #64          // Write out inner index

       svc #0

       mov w0, #1

       mov w2, #1

       mov w5, #9           // Tab for neatness

       strb w5, [x1]        //Load up a tab character
       mov w8, #64          // Write out the Tab
       svc #0
       add w5, w4, #48             // Convert outer index to ASCII
       mov w0, #1
       mov w2, #1
       strb w5, [x1]        // Put outer index ascii value into string space
       mov w8, #64
       svc #0
       mov w5, #10
       strb w5, [x1]        // Newline character
       mov w8, #64
       svc #0
       add w3, w7, w3              // incrementinner loop
       cmp w3, w9           // End of inner loop index reached?
       B.EQ incrementouter // Time to increment adjacent for loop
       B incrementinner
incrementouter:
       mov w3, #1           // Set innerloop index back to starting value
       add w4, w7, w4              // Increment adjacent loop index
       cmp w4, w9           // End of outer loop finished?
       b.ne incrementinner
exit:
       mov w0, #1
       mov w10, #10
       mov w11, #13
       strb w10, [x1]
       mov w2, #1
       mov w8, #64
       svc 0
       mov w0, #1
       strb w11, [x1]
       mov w2, #1
```

```
      mov w8, #64          // Invoke the Write system call
      svc #0
      mov w8, #0x5d        // Time to go.
      svc #0
.data
      printheader: .ascii "\nInner Outer\n"
      printvalues: .space 8
```

Output:

```
Inner Outer
1      1
2      1
3      1
1      2
2      2
3      2
1      3
2      3
3      3
```

Two data regions have been defined – ***printheader*** which is used to print out a heading and then ***printvalues*** which is an area of memory that reserves empty space to hold the calculated loop values. Prior to printing out the values the numbers are converted to ASCII text by adding the value 48[23], placing the result in w5.

Each character is stored into the empty *printvalues* space by the instruction `strb` w5, [x1,#0[24]]. This instruction stores the value held in w5 to the memory location pointed to by x1 with an

[23] This because the ASCII numeric characters are consecutive in value and the ASCII character for zero is 48 (0x30).

[24] Note When the offset is zero, then GDB will assume this if no immediate value is given, hence the listing omits #0 in the strb commands.

offset of 0. The addressing mode calculates *a register plus an offset*. Square brackets signify indirection. This program stored each character at the same memory location, destroying the previous contents. The Pre-index addressing mode preserves the data that was generated by incrementing the memory location, thereby storing data in consecutive locations. The format of the instruction is highlighted in Listing 5-4. The instruction stores the byte held in register w5 at the memory location pointed to by register x1.

This is different from other instructions, in that the first register is the source.

Listing 5-4 Nested loops with pre-index addressing mode

```
//listing 5-4

/* This example shows a Nested For Loop in action

Both loops start with an index of 1 and count up to 3

- w3 holds the index value for the inner loop

- w4 holds the index value for the outer loop

- w9 holds the termination loop value

- w5 holds the ASCII equivalent of the current index value

For writing -

w0 = 1 <stdout>

w1 = Character location in memory

w2 = Character count for output to stdout

*/

.text

.global _start

_start:

      mov W9, #4           //ending value for loop

      mov w7, #1           // For loop iteration value

      mov w0, #1           //stdout

      ldr w1, =printheader

      mov w2, #14          // Character count of printheader string

      mov w8, #64          // Write out header text

      svc #0

      ldr w1, =printvalues // Now that the header has been printed get ready to
print values

      mov w3, #0x1  // Load up reg w3 with starting inner loop value
```

```
        mov w4, #0x1  // Load up reg w4 with starting outer loop value
incrementinner:
        add w5, w3, #48             // Convert inner index to ascii
        add w6, w4, #48             // Convert outer index to ascii
        mov w0, #1          // stdout, don't assume x0 is preserved after svc call
        strb w5, [x1,#1]!           // Put character into string space
        mov w2, #2          // One character at a time
        mov w8, #64         // Write out inner index
        svc #0
        mov w0, #1
        mov w2, #1
        mov w5, #9          // Tab for neatness
        strb w5, [x1,#1]!           //Load up a tab character
        mov w8, #64         // Write out the Tab
        svc #0
        add w5, w4, #48             // Convert outer index to ASCII
        mov w0, #1
        mov w2, #1
        strb w5, [x1,#1]!           // Put outer index ascii value into string space
        mov w8, #64
        svc #0
        mov w5, #10
        strb w5, [x1,#1]!           // Newline character
        mov w8, #64
        svc #0
        add w3, w7, w3              // incrementinner loop
        cmp w3, w9          // End of inner loop index reached?
        B.EQ incrementouter // Time to increment adjacent for loop
        B incrementinner
incrementouter:
        mov w3, #1          // Set innerloop index back to starting value
        add w4, w7, w4              // Increment adjacent loop index
        cmp w4, w9          // End of outer loop finished?
        b.ne incrementinner
exit:
        mov w0, #1
        mov w10, #10
        mov w11, #13
        strb w10, [x1,#1]!
        mov w2, #1
        mov w8, #64
        svc 0
        mov w0, #1
        strb w11, [x1,#1]!
        mov w2, #1
        mov w8, #64         // Invoke the Write system call
        svc #0
        mov w8, #0x5d       // Time to go.
        svc #0
.data
        printheader: .ascii "\nInner Outer\n"
        printvalues: .space 48
```

At Program completion -

```
memory address      hex                                                        char
0x4101a1            00 31 09 31 0a 32 09 31 0a 33 09 31 0a 31 09 32
        .1.1.2.1.3.1.1.2
0x4101b1            0a 32 09 32 0a 33 09 32 0a 31 09 33 0a 32 09 33
        .2.2.3.2.1.3.2.3
R
Registers
name    value (hex)
x0      0x1
x1      0x4101c4
x2      0x1
x3      0x3
```

```
x4    0x3

x5    0xa

x6    0x33

x7    0x1

x8    0x40

x9    0x4

x10   0x0

. . .

x30   0x0

sp    0x7fffffff080

pc    0x400128

cpsr  0x80201000
```

Output

```
Inner Outer

1       1

2       1

3       1

1       2

2       2

3       2

1       3

2       3

3       3
```

Summary of chapter 5

- Compare instructions
- Conditional branching
- Nested loops

Exercises for chapter5

1. Describe the difference between a conditional and unconditional branch
2. Which instruction is `cmp` an alias for?
3. How does the flag condition signify that the signed less than condition is true?
4. Modify listing 5-2 to accept user input (hint think syscalls)

Chapter 6. The Stack, Macros and Functions

Macros and Functions

This chapter introduces areas that are used by real-world (and other coders) to better manage and clarify their programs. Now that the listings are getting longer, it makes sense to introduce the concept of *macros* and *functions*. The concepts are similar but the way that the programs are assembled leads to tradeoffs behind performance and code size.

Listing 6-1 shows a print macro which requires two parameters – the location of the string to be printed and its location. Output goes to stdout. The macro is called twice, each time with different parameters. This basic example does not save much in typing, but the benefit is significant when larger macros are used. The macro code is enclosed between the assembler directives `.macro` and `.endm`. Macros are used to repeat frequently used instructions using different parameters

Listing 6-1 A simple macro

```
* This shows an example of a macro

The macro prints to stdout, input parameters are the location of the string and
its character count

It is called twice, to print both strings*/

.text

.global _start

      _start:

             mov w2, #39

             .macro print location, length     // Macro expects string location
and its length

                    mov w0, #1          //stdout

                    ldr w1, =\location  //Pass location

                    mov w2, \length     //Pass length

                    mov w8, #64

                    svc #0

             .endm

      print string1, w2   // Call macro with parameters string1 and 39!

      mov w2, 16

      print string2, w2   // Call macro with parameters string2 and 18
      mov w8, #93         // Exit the program

      svc 0

.data

      string1: .ascii "\nThis string was printed using a macro\n"

             string2: .ascii "and so was this\n"
```

Output –

```
This string was printed using a macro

and so was this
```

Disassembly shows:

```
$ objdump -d -M no-aliases macro2

macro2:     file format elf64-littleaarch64

Disassembly of section .text:

00000000004000b0 <_start>:

  4000b0:     528004e2      movz   w2, #0x27

  4000b4:     52800020      movz   w0, #0x1

  4000b8:     18000181      ldr    w1, 4000e8 <_start+0x38>

  4000bc:     2a0203e2      orr    w2, wzr, w2

  4000c0:     52800808      movz   w8, #0x40

  4000c4:     d4000001      svc    #0x0

  4000c8:     52800202      movz   w2, #0x10

ca  4000cc:   52800020      movz   w0, #0x1

  4000d0:     180000e1      ldr    w1, 4000ec <_start+0x3c>

  4000d4:     2a0203e2      orr    w2, wzr, w2

  4000d8:     52800808      movz   w8, #0x40

  4000dc:     d4000001      svc    #0x0

  4000e0:     52800ba8      movz   w8, #0x5d

  4000e4:     d4000001      svc    #0x0

  4000e8:     004100f0      .word  0x004100f0

  4000ec:     00410117      .word  0x00410117
```

The highlighted sections show the macro, which has been written inline twice. In line code can be fast but will generate larger code when called extensively.

It is also possible to break out the macro into a separate file which can be called using the `.include` directive.

Listing 6-2 Separate macro file

```
$ cat printmacro.s
             .macro print location, length     // Macro expects string location
and its length
             mov w0, #1          //stdout
             ldr w1, =\location  //Pass location
             mov w2, \length     //Pass length
             mov w8, #64
             svc #0
             .endm
```

Listing 6-3 Calling a macro using the include directive.

```
$ cat callmacro.s
/* This shows an example of a macro call
The macro prints to stdout, input parameters are the location of the string and
its character count
It is called twice, to print both strings*/
.text
.include "printmacro.s"
.global _start
_start:
      mov w2, #39
      print string1, w2   // Call macro with parameters string1 and 39!
      mov w2, 16
      print string2, w2   // Call macro with parameters string2 and 18
      mov w8, #93         // Exit the program
             svc 0
      .data
             string1: .ascii "\nThis string was printed using a macro\n"
             string2: .ascii "and so was this\n"
```

The Stack

Functions will make use of the *stack*. The stack is a data structure which stores data in a structured manner. As an example, a register's contents can be *Pushed* on to the stack and can be restored by *Popping* the data from the stack back to the register again. Push and Pop operations are performed in a Last in First out (LIFO) manner, in that if multiple registers were pushed on to the stack the last register pushed would be the first one restored. The stack is a location in memory.

A stack pointer will show where in memory the top of the stack is situated. When data is pushed the stack pointer will be decremented to a lower memory location and when data is popped, the stack pointer will be incremented. A push to the stack is accomplished using the `str` instruction and a pop is accomplished using the `ldr` instruction. Both these instructions are familiar, the only difference being that the stack pointer is used as the operand rather than a normal register. With ARM64, the stack grows downwards in memory and must be 16-byte aligned.

- The ARM64 documentation states that
    - Formally, sp must lie in the range stack_limit < sp <= stack_base, though the values of stack_limit and stack_base are often inaccessible.
- The memory below sp (but above stack_limit) must not be accessed by your code.

Listing 6-4 shows examples of push and pop operations.

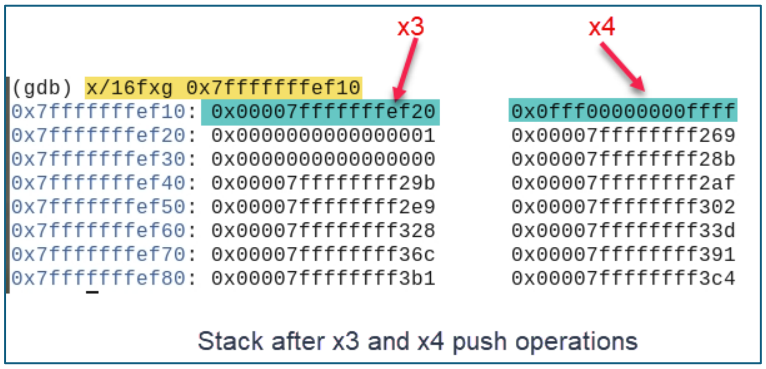
Listing 6-4 Push and Pop operations using str and ldr

```
.text
  .global _start
  _start:
  //This program shows how to interact with the stack
  mov x4, #0xffff
  movk x4, #0x0000, lsl 16
  movk x4, #0x0000, lsl 32
 movk x4, #0xfff, lsl 48
 mov x3, sp              // Move stack to register x3. SP at 0x7ffffffef20
 str x4, [SP, #-16]!     // SP now at 0x7fffffffef10 (lower memory location)
 mov x4, #0      //Clobber X4
 ldr x4, [sp], #16       // Restore x4, SP now back to 0x7fffffe20
```

```
  stp x3,x4, [sp, #-16]!  // Store register x3 and x4 on to the stack SP =
0x7ffffffef10

  mov x3, xzr             // Clobber x3

  mov x4, xzr             // Clobber x4

  ldp x3, x4, [sp], 16    // Restore both, SP = 0x7ffffffe20

  mov w8, #93 //Time to go

  svc 0
```

The stack could also push to higher memory addresses as shown in **Error! Reference source n ot found.**. The actual implementation is architecture dependent!

Figure 6-1 Stack memory contents after stp x3, x4, [sp, #-16]! instruction



```
(gdb) x/16fxg 0x7ffffffffef10
0x7ffffffffef10: 0x00007ffffffffef20      0x0fff00000000ffff
0x7ffffffffef20: 0x0000000000000001      0x00007fffffffff269
0x7ffffffffef30: 0x0000000000000000      0x00007fffffffff28b
0x7ffffffffef40: 0x00007fffffffff29b      0x00007fffffffff2af
0x7ffffffffef50: 0x00007fffffffff2e9      0x00007fffffffff302
0x7ffffffffef60: 0x00007fffffffff328      0x00007fffffffff33d
0x7ffffffffef70: 0x00007fffffffff36c      0x00007fffffffff391
0x7ffffffffef80: 0x00007fffffffff3b1      0x00007fffffffff3c4
```

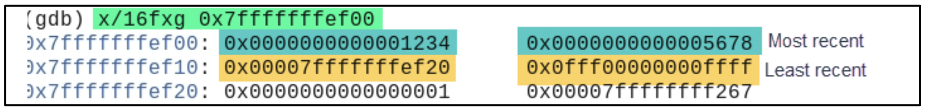
The stack supports nested operations, as shown in Listing 6-5

Listing 6-5 Nested stack operations

```
// listing6-5  1

.text


  .global _start

  _start:

  //  This program shows nested stack operations

  mov x4, #0xffff

  movk x4, #0x0000, lsl 16

  movk x4, #0x0000, lsl 32

  movk x4, #0xfff, lsl 48

  mov x3, sp              // Move stack to register x3. SP at 0x7ffffffef20

  stp x3,x4, [sp, #-16]!  // Store register x3 and x4 on to the stack SP =
0x7ffffffef10
```

```
mov x3, #0x1234          // Fresh write to x3
mov x4, #0x5678          // Fresh write to x4
stp x3, x4, [sp, #-16]!
mov x3, xzr              // Clobber x3
mov x4, xzr              // Clobber x4
ldp x3, x4, [sp],16      // Resore most recent value of x3 and x4
ldp x3, x4, [sp], 16     // Restore next most recent values of x3 and x4
mov w8, #93 //Time to go
svc 0
```

Figure 6-2 Stack contents with nested operations



```
(gdb) x/16fxg 0x7fffffffef00
0x7fffffffef00: 0x0000000000001234      0x0000000000005678
0x7fffffffef10: 0x00007fffffffef20      0x0fff00000000ffff
0x7fffffffef20: 0x0000000000000001      0x00007fffffffff267
```

Functions are used to promote coding efficiency and clarity. They are sections of code that can be included in a program and shared with others as *libraries*. Over time a coder will usually generate their own functions for use in their code. When using external functions, registers can be saved on the stack prior to calling the function, thus ensuring that on return from the function code everything has been restored and coding will continue from where it left off. The *Program Counter (PC)* keeps track of the location in memory where the code is next to be executed. When a portion of code calls a function, it is termed the *caller*. The code that was called (the function itself) is termed the *callee*. When calling a function there are several tasks that the caller must perform and similarly the callee has its own responsibilities.

The registers follow certain conventions:

- Parameters are passed via registers x0 through x7[25].
- Values are returned through register x0.
    - Other parameters can be stored in memory using the return register to point to the address.

[25] Additional parameters can be passed using the stack. The parameters are pushed and then popped

- The x8 register (in Linux) is used for svc calls.
- Registers x19 through x28 are to be preserved for the caller.
    - The callee will save these values.
- Register X29 is the frame pointer register and will be discussed later.
- Register X30 is the link register and is discussed below.

The rules are documented in the *ARM Procedure Call Standard (PCS)*. The standard also defines which registers are corruptible and which are not. A called function can overwrite corruptible registers. If the function uses non-corruptible registers, then it will perform a stack push and then a stack pop prior to returning.

Link Register

The *link register* (LR) is register x30 and is used to hold the address of the next instruction to be executed after the function has been returned from. The Branch with link `(BL`) instruction is used to call the function and put the returning address into the link register.

The next program consists of a main program (`main.s`) which call two functions[26] `(cubit.s`) and `dubdab.s`). A set of integers ranging from 1 to 10 are passed to the *cubeit* function which calculates the cube of the numbers. There are several locations in memory used for specific purposes –

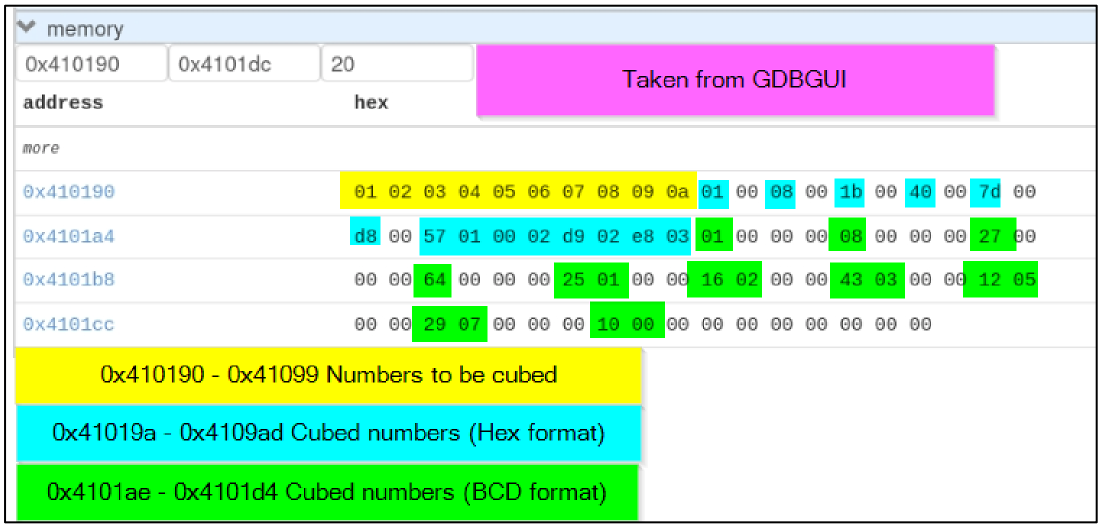
Table 6-1 Memory locations used by the listcubes program

Location Name	Purpose
numberlist	Holds the bye values 1 through 10
cubeslist	Holds the calculated cubes held in numberlist
bcdlist	Holds the list of cubes converted to BCD

Figure 6-3 shows the memory regions and associated values prior to formatting.

[26] In most case functions accept inputs and return value. The listings *cubeit* and *dubdab* are more like routines and could be implemented as macros.

Figure 6-3 Memory locations for the cube program and their values



```
memory
0x410190    0x4101dc    20
address                 hex
more
0x410190                01 02 03 04 05 06 07 08 09 0a 01 00 08 00 1b 00 40 00 7d 00
0x4101a4                d8 00 57 01 00 02 d9 02 e8 03 01 00 00 00 08 00 00 00 27 00
0x4101b8                00 00 64 00 00 00 25 01 00 00 16 02 00 00 43 03 00 00 12 05
0x4101cc                00 00 29 07 00 00 00 10 00 00 00 00 00 00 00 00 00
```

Taken from GDBGUI
0x410190 - 0x41099 Numbers to be cubed
0x41019a - 0x4109ad Cubed numbers (Hex format)
0x4101ae - 0x4101d4 Cubed numbers (BCD format)

The *cubeit* routine is simple, it takes a value from the memory location *numberlist* pointed to by register x1. It multiplies the number by itself twice, storing the value in register w0. Main will store the returned value into *cubeslist*, incrementing it to the next location and the calls cubit again until the loop count has reached zero[27].

The next routine to be called is *dubdab* which performs the double dabble routine. Each number has room for a units weight, a tens weight and a hundreds weight. These partitions take up 4 bits so a total of 12 shifts are used[28]. The double-dabble algorithm is covered on page 1-16. The routine is responsible for storing the bcd number in the memory location *bcdlist* pointed to by register x19. This is done at the label *putbcd*.

The final task is to separate the nibbles into bytes and then apply offsets corresponding to ASCII values.

[27] Make sure that instruction SUBS is used and not SUB to set the flags appropriately.

[28] Note it is not necessary to perform 12 shifts if only single or double digits will result, however rather than parse out the number of digits and then calculate the required shift count, it was considered "cleaner to have a fixed worst case shift number. Of course, if performance were a consideration, then significant savings would be realized by reducing the shift count.

This is done by the function `convert` (in `main.s`) which extracts the values from *bcdlist* starting at the label *getbcd*. The first task is to separate the nibbles and put them into byte form, that is to say where the digits previously occupied four bits they now require an eight bit space. This is because they are to be converted to ASCII format which requires byte space. A lot of bit twiddling is performed here to move the bytes into the correct position. After this the `rev` instruction is used to reverse the byte order putting them in the correct locations. In between each store a line feed/carriage return is inserted to improve formatting.

Stripping away the leading zeros is not performed!

The output looks like –

```
./listcubes

00000001

00000008

00000027

00000064

00000125

00000216

00000343

00000512

00000729

00001000
```

Listing 6-6 Main program to print out cubed numbers

```
main.s

text

numbercount=10

.global _start

_start:

//  This program shows how to call functions

// The program will print out the first 10 cubes of 1-10.

// Two functions are called, the first to calculate the cubes

// and the second to convert the cubes to BCD

// main.s

      ldr x1, =numberlist
```

```
       ldr x2, =numbercount
       ldr x3, = cubeslist
       ldr x19, = bcdlist
loop:
       bl cubeit //Call function cubeit
       strh w0, [x3], #2 //Put the returned cube from w0 into cubeslist
       subs x2, x2, #1
       bne loop
//Reset registers
       ldr x2, =numbercount
       ldr x3, = cubeslist //all cube results are now stored in word space
       bl dubdab
convert:
       mov w7, #10 // Get all ten BCD cube numbers
       ldr w19, = bcdlist
       ldr w20, = cubes
       mov x4, #0x3030 // ASCII adjustment
       movk x4, #0x3030, lsl #16
       movk x4, #0x3030, lsl #32
       movk x4, #0x3030, lsl #48
       mov x14, #0xff      // Use for masking out leading zeros
       mov w6, #0x0a0d     // Line feed and carriage return
getbcd:
       ldr x10, [x19], #4  // Get entry from BCD list
       and x15, x10, #0xf  // mask out all but first nibble
       mov x16, x15        // w16 holds first byte
       and x15, x10, #0xf0        // Mask out all but second nibble
       lsl x15, x15, #4    // Nibble2 now in second byte position
       orr x16, x15, x16   // Don't destroy existing data in w16
       and x15, x10, 0xf00 // Mask out all but third nibble
       lsl x15, x15, #8    // Nibble3 now in third byte position
       orr x16, x15, x16   // Don't destroy existing data in w16
       and x15, x10, #0xf000      // Mask out all but fourth nibble
       lsl x15, x15, #12   // Nibble 4 now in fourth byte position
       orr x16, x15, x16   // Don't destroy existing data in x16
       and x15, x10, #0xf0000  // mask out all but fifth nibble
       lsl x15, x15, #20   // Nibble5 now in fifth byte position
       orr x16, x15, x16       // Don't destroy existing data in x16
       and x15, x10, #0xf00000 // Mask out all but sixth nibble
       lsl x15, x15, #24       // Nibble6 now in sixth byte position
       orr x16, x15, x16       // Don't destroy existing data in w16
       and x15, x10, 0xf000000 // Mask out all but seventh nibble
       lsl x15, x15, #28        // Nibble7 now in seventh byte position
       orr x16, x15, x16       // Don't destroy existing data in w16
       and x15, x10, #0xf0000000   // Mask out all but eighth nibble
       lsl x15, x15, #32        // Nibble8 now in eighth byte position
       orr x16, x15, x16       // Don't destroy existing data in x16
       add x16, x16,x4     // Convert bytes to ASCII format
       rev x17, x16        // The low shall become high and the high become low!
       str x17, [x20], #8
       strh w6, [x20], #2 // Put in line feed
       subs w7, w7, #1
       b.ne getbcd
printbuffer:
       mov x0, #1
       ldr x1, =cubes
       mov x2, #150
       mov x8, #64
       svc #0
exit:
       mov w8, #93 //Time to go
       svc 0
.data
numberlist: .byte 1,2,3,4,5,6,7,8,9,10
```

```
.align

cubeslist: .space 200

.align

bcdlist: .space 40

cubes: .space 110

linefeed: .ASCII "\n"
```

Listing 6-7 Routine to calculate cube numbers

```
cubeit.s

// Simple function to cube a number

// cubeit.s

.global cubeit

cubeit:

       ldrb w5, [x1], #1

       mul w0, w5, w5

       mul w0, w5, w0

ret
```

Listing 6-8 Double-Dabble routine to convert hex/binary to binary coded decimal

```
dubbdabb.s

.text

//  This function implements the double dabble algorithm

// It takes a list of 10 numbers and converts them to BCD

// w9 holds the hex number to be shifted

// w10 holds the BCD number

// w11 for the unitsmask

// w12 for the tensmask

// w13 for the hundredsmask

// W14 holds the number of cubes to be converted

// w15 holds the number of binary digits that the cube has (here n =12)

// w16 is a scratchpad to hold the result of anding in routine getnumberofdigits

// X17 is a counter for the number of shifts performed in double dabble

// w19 points to the location where the BCD cubes are stored

// dubbdabb.s
```

```
.global gethexnumbers

.global getnumberofdigits

.global dodoubledabble

.global dubdab

dubdab:

unitsmask=0xf000

tensmask=0xf0000

hundredsmask=0xf00000

numberofshifts=12

numberofcubes=10

       mov w11, #unitsmask

       mov x12, #tensmask

       mov x13, #hundredsmask

       mov w14 ,#numberofcubes

       mov x15, #numberofshifts

gethexnumbers:

       ldrh w9, [x1], #2 //load cube

dodoubledabble:

// Start pushing

       mov w17, w15 // Use w17 as the shift counter

       shiftnbits: // n is held in w17

       lsl w9, w9,  #1 //shift into units, tens and hundreds area

       subs w17, w17, #1

       beq putbcd

checkhundreds:

       and w16, w9, w13 // Only look at hundreds column

       cmp w16, 0x500000   // Dabble needed ?

       b.lt checktens  // If not try the tens column

       add w9, w9, #0x300000 //Dabble the hundreds column

checktens:

       and w16, w9, w12 // Only look at the tens column

       cmp w16, 0x50000
```

```
      b.lt checkunits
      add w9, w9, #0x30000 // Dabble the tens column
checkunits:
      and w16, w9, w11  // Only look at units column
      cmp w16, 0x5000
      b.lt skipunits
      add w9, w9, #0x3000
skipunits:
      b shiftnbits
putbcd:
      mov w10, w9 //Put the double dabble number into x10 (units)
      lsr w10, w10, w15 // Discard bits 0-11 and move bcd number into its place
      str w10, [x19], #4
      subs w14, w14, #1
      b.eq exitdd
      b gethexnumbers
exitdd:
      ret
```

The *makefile* combines the three listings into the program *listcubes* and is as follows:

```
OBJECTS = main.o cubeit.o dubbdabb.o
all: listcubes
%.o : %.s    # Any .o .s
      as $< -g -o $@      # $< source file $@ output file
listcubes: $(OBJECTS)
      ld -o listcubes $(OBJECTS)
```

Summary of chapter 6

- Use of the stack
- Macros
- Functions
- Calling conventions
    - Caller and callee

Exercises for chapter6

1. What is the purpose of the Link register?
2. What is the ARM Procedure call standard used for?
3. Modify the `listcubes` program to strip out leading zeros
4. Explain the difference between a function and a macro
5. Which directives signify the start and end of a macro?
6. When is the .include directive used?
7. What instruction can be used to push values on the stack?

Chapter 7. Calling assembly functions from a high-level language

However instructive the previous `listcubes` code was, outputting the text was complex, often it would not be practical to code in all parts of a program assembly language. Some of the many disadvantages include:

- Complexity
- Difficult to debug
- Hard to test
- Time to develop, optimize and document

In the real world, a more pragmatic approach is used. Code is more often (than not) written in a higher-level language such as C, C++ or Python, which has many built-in functions and libraries that the programmer can call upon. A hybrid approach is often taken where assembly code might be used for time critical parts or for direct access to the target machine's hardware. The GNU Compiler Collection (GCC) allows compilation of a mixture of code. The following example shows how to call ARM64 assembly from C code.

First develop a simple assembly language function which cubes a number and then adds an offset.

Listing 7-1 Cube and add assembly code

```
.global cubeandadd
cubeandadd:
mov w2, w0
mul w0, w0, w0  // Arguments are in r0 and r1
mul w0, w2, w0
add w0, w0, w1
ret
```

The function *cubeandadd* has been declared as a global function to allow for external access. It receives its parameters from the c code shown in Listing 7-2.

Listing 7-2 Cube and add C code

```
#include <stdio.h>
```

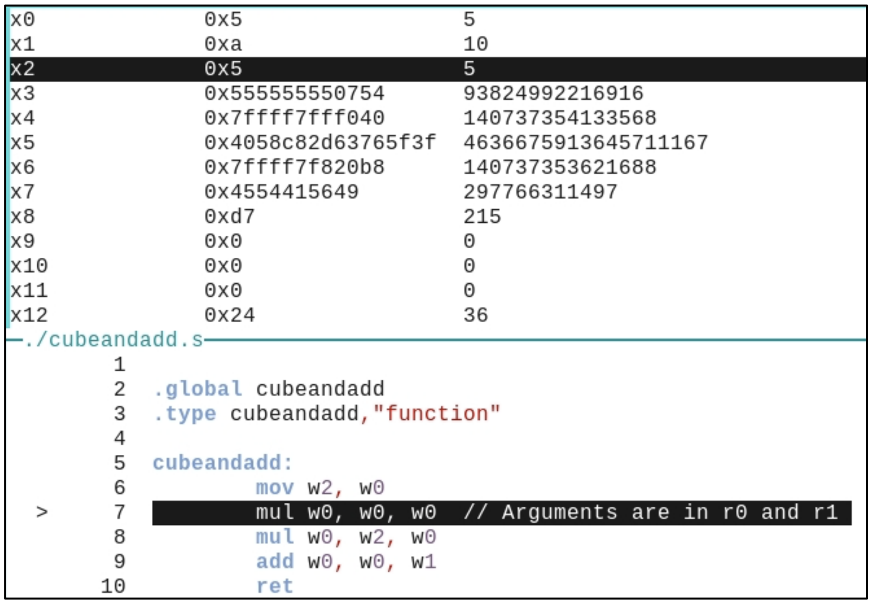
```
extern int cubeandadd(int a, int b);

int main()

{

       int a = 5;

       int b = 10;

       printf ("\n The cube and add function, calls assembly code to cube the
first number %d and then add the second number %d, the result is %d\n", a, b,
cubeandadd(a,b));

    return (0);

}
```

The assembly function (`cubeandadd`) has been declared as *external* and it passes its parameters (a and b to the assembly code.

The output code is generated by gcc using the command:

```
gcc -g -o cubeandadd ./listing7-2.c ./listing7-1.s
```

The debugger shows the code midway through execution.



```
x0             0x5                 5
x1             0xa                 10
x2             0x5                 5
x3             0x555555550754      93824992216916
x4             0x7ffff7fff040      140737354133568
x5             0x4058c82d63765f3f  4636675913645711167
x6             0x7ffff7f820b8      140737353621688
x7             0x4554415649        297766311497
x8             0xd7                215
x9             0x0                 0
x10            0x0                 0
x11            0x0                 0
x12            0x24                36
-./cubeandadd.s-

           .global cubeandadd
           .type cubeandadd,"function"

           cubeandadd:
                   mov w2, w0
   >               mul w0, w0, w0  // Arguments are in r0 and r1
                   mul w0, w2, w0
                   add w0, w0, w1
                   ret
```

The C library function printf is defined within `<stdio.h`> as `int printf(const char *format,…)` It is a *variadic* function which means that it can take a variable number of arguments. This is conveyed by the ellipsis… in the prototype. The function takes a minimum of one argument which is a pointer to the location of the starting character of the text. The text itself can embed formatting tags which specify how the arguments that are passed are to be printed – for example a variable using "%d" will be formatted as a signed base 10 integer. To print a string, register x0 will have been loaded with the address of the text (see Listing 7-5), variables are passed into the other registers (see Listing 7-6).

A non-exhaustive list of format specifiers are shown in Table 7-1.

Table 7-1 printf format specifiers

Format specifier	interpretation
%d	Signed decimal number
%u	Unsigned decimal number
%s	Pointer to an array of characters
%c	Outputs a single character
%x	Represents an unsigned integer in lower case hexadecimal form
%X	Represents an unsigned integer in upper case hexadecimal form
%%	Outputs a literal "%" character
%e	Represents floating point as decimal exponent notation
%f	Represents floating point as decimal

Using in-line code

Basic and Extended ASM

Listing 7-3 makes use of assembly instructions with operands. This is known as *Extended ASM* as opposed to *Basic ASM*

Basic ASM

Basic ASM is a set of assembly instructions. With inline code the `asm` keyword is not an actual C keyword[29] but it is understood by the assembler. Note that non-GNU assemblers may use an alternative keyword. An example is shown below:

```
asm(
       "mov w4, #5\n\t"
       "mov w5, #15\n\t"
       "add w6, w4, w5\n\t"
       );
```

[29] This is not the case with C++.

Note that the instructions are separated by the combination of /n and /t.

Extended ASM

Extended ASM can use variables from the C source code. Extended ASM cannot be used outside of C functions The assembler template consists of:

```
asm(code template :  output operand(s) : input operand(s) : clobber list);
```

Table 7-2 gives an explanation.

Table 7-2 Inline assembly template

Template		
Phase	Example	Description
Code - Assembler Instruction	`mov w0, w1`	Regular assembly instruction
Code Template	`mov %[inputa], %[inputb]`	Using parameters passed as inputs to the code template
Output Operand(S) List	`[answer] "r" (result)` Can be left empty using :	List of output operand(s) [answer] is a symbolic name, r is a constraint string meaning register and (result) is returned to the Calling code.
Input Operands List	`inputa] "r" (a), [inputb] "r" (b)`	Similar syntax to operand list
Clobber List	`"x5", "x6"`	Optional list of registers, that *may* not be preserved

A significant advantage of using inline assembly code like this is that the task of procedure call handling (see page 6-7) is left to the compiler.

Listing 7-3 shows an example of assembly code being executed in-line with the C code. This code cubes a number and then adds a constant ($x^3 + y$). Here the number 5 is cubed and then the constant 10 is added.

Listing 7-3 Using inline assembly code with C

```
int cubeandadd(int a, int b, int c)
{
        int res = 0;
        /*Assembly Template is as follows:
        Code (Assembly language instruction such as add x0, X0, X3)
```

```
       Code template ( add %result, %[input1], %[input2], . . .)

       Output Operands ([result] "=r" (res); r is a constraint string which is a
general purpose 64-bit X register

       = is a constraint modifier for writing, + is for read and write

       Input Operands ([inputa] "r" (a) [inputb] "r" (b); two input operands a and
b*/

       asm(

       "mov %[inputc], %[inputa]\n"

       "mul %[inputa], %[inputa], %[inputa]\n\t"

       "mul %[inputa], %[inputa], %[inputc]\n\t"

       "add %[result], %[inputa], %[inputb]\n\t"

       : [result] "=r" (res)   // Output Operand(s) list r = general registers

       : [inputa] "r" (a), [inputb] "r" (b), [inputc] "r" (c) // Input Operand(s)
list

       );

       return res;

}

int main (void)

{

       int a = 5;

       int b = 10;

       int c = 0;

       int result = cubeandadd(a,b,c);

       printf ("Cubing %d and adding %d = %d\n", a,b,result);

}
```

Using the `gcc` option `gcc -save-temps listing7-3.c` will allow the preservation of the intermediate files that were generated during the compilation process. An extract of the assembly file is shown below:

```
cubeandadd:

.LFB0:

       .cfi_startproc

       sub    sp, sp, #32

       .cfi_def_cfa_offset 32
```

```
        str    w0, [sp, 12]
        str    w1, [sp, 8]
        str    w2, [sp, 4]
        str    wzr, [sp, 28]
        ldr    w0, [sp, 12]
        ldr    w1, [sp, 8]
        ldr    w2, [sp, 4]
#APP
// 15 "listing6-12.c" 1
        mov x2, x0
mul x0, x0, x0
        mul x0, x0, x2
        add x0, x0, x1
```

Looking at the *cubeandadd* routine , it can be seen that registers x0, x1 and x2 are used. Register X0 holds the first parameter (5), a copy of X0 is placed in register X2. X0 is then multiplied by itself with the result 25 being stored in register X0. The updated X0) value (25) is then multiplied by the original value of X0 (which is stored in X2) and X0 now holds the value 125. The second operand passed in X1 is added to X0 giving the final result of 135.
The is shown in Table 7-3.

Table 7-3 In line assembly converted

Source	Assembled code
“mov %[inputc], %[inputa]”	mov x2, x0
"mul %[inputa], %[inputa], %[inputa]”	mul x0, x0, x0
"mul %[inputa], %[inputa], %[inputc]”	mul x0, x0, x2
“add %[result], %[inputa], %[inputb]”	add x0, x0, x1

The next listing revisits Listing 6-6 that cubed the first ten numbers –

Listing 7-4 Cube numbers revisited

```
include <stdio.h>
int cubenumbers(int counter, int index)
{
        int res;
        asm(
        "mul %[outputresult], %[inputcounter], %[inputcounter]\n\t"
        "mul %[outputresult], %[inputcounter], %[inputindex]\n\t"
        : [outputresult] "=r" (res)   // Output Operand(s) list r = general
registers
        : [inputcounter] "r" (counter), [inputindex] "r" (index)  // Input
Operand(s) list
        );
        return res;
}
int main (void)
{
        int counter = 1;
        int index = 1;
        int result = cubenumbers(counter,index);
        while (counter <11)
        {
                result=cubenumbers(counter,index);
                printf ("Cubing %d = %d\n", counter,result);
                counter++;
                index++;
        }
}
```

Compilation String

```
$ gcc -g -o cubenumbers ./listing7-4.c
```

Output

```
asm/chapter07 $ ./cubenumbers
Cubing 1 = 1
Cubing 2 = 8
Cubing 3 = 27
Cubing 4 = 64
Cubing 5 = 125
Cubing 6 = 216
Cubing 7 = 343
Cubing 8 = 512
Cubing 9 = 729
Cubing 10 = 1000
```

Much more concise!

The next two listings use `printf` to print out values. Arguments are passed to `printf` via the X registers or the vector registers in the case of floating-point numbers.

Listing 7-5 Using printf to print a string from assembly

```
// listing 7-5
.text
.global _start
_start:
ldr x0, =string1
bl printf    // Use -nostartfiles when linking with gcc
mov w8, #93
svc #0
string1: .asciz "This string was printed from assembly using printf\n"
```

Use the following commands to build the program

```
as -g -o listing7-5.o listing7-5.s
gcc  -nostartfiles -o listing7-5 listing7-5.o
```

The `-nostartfiles` option means do not use the standard system startup files when linking.

The listing passes the location of the string to `printf` and outputs:

```
"This string was printed from assembly using printf"
```

The next listing uses registers x0, x1, x2 and x3.

Listing 7-6 Using printf to print numbers

```
// listing7-6
.text
.global _start
_start:
ldr x0, =string1
mov x1,#5
mov x2, #15
```

```
add x3, x1, x2

ldr x0, =string1

bl printf    // Use -nostartfiles when linking with gcc

mov w8, #93

svc #0

string1: .asciz "The first number is %d, the second number is %d, the addition of the two numbers is: %d\n"
```

Use the following commands to build the program

```
as -g -o listing7-6.o listing7-6.s

gcc  -nostartfiles -o listing7-6 listing7-6.o
```

The output is:

“The first number is 5, the second number is 15, the addition of the two numbers is: 20”.

Another example shows output using some of the format specifiers shown in Table 7-1

Listing 7-7 Use of format specifiers

```
//listing7-7

.text

.global _start

_start:

ldr x0, =string1

mov x1,#-140

mov x2, #15

add x3, x1, x2

ldr x0, =string1

str x1, [SP, #-16]!

str x2, [SP, #-16]!

str x3, [SP, #-16]!

bl printf    // Use -nostartfiles when linking with gcc

ldr x0, =string2

ldr x3, [sp], #16

ldr x2, [sp], #16

ldr x1, [sp], #16

bl printf
```

```
mov w8, #93

svc #0

.data

      string1: .asciz "The first number represented as signed decimal is %d, the
second number represented as lower case hexadecimal is %x, the addition of the two
numbers represented as upper case hexadecimal is: %X\n"

      string2: .asciz "\nThe first number represented as unsigned decimal is %u,
the second number represented as signed decimal is %d, the addition of the two
numbers represented as upper case hexadecimal is: %X\n"
```

```
Output:

The first number represented as signed decimal is -140, the second number
represented as lower case hexadecimal is f, the addition of the two numbers
represented as upper case hexadecimal is: FFFFFF83

The first number represented as unsigned decimal is 4294967156, the second number
represented as signed decimal is 15, the addition of the two numbers represented
as upper case hexadecimal is: FFFFFF83.
```

Summary of chapter 7

- Using in line assembly code
- Compiling C and assembly code together
- Printf and variants

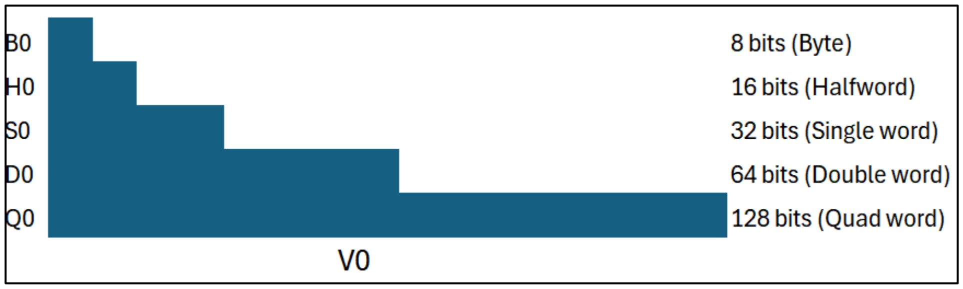
- Exercises for chapter7

1. How would you print the literal character “%” with `printf`
2. Which register is used to convey the location of the string to be printed when using printf?
3. How would you preserve intermediate files that were generated during the compilation process?
4. What is the purpose of `-nostartfiles`, try compilation without using it

Chapter 8. Floating Point and Neon Coprocessor

This section discusses the vector registers and the concept of Single Instruction Multiple Data (SIMD) with emphasis on arithmetic operations. ARM64 adheres to the floating-point IEEE 754 standard as discussed earlier (see page 1-18). There are 32 x 128-bit vector registers (see page 2-4). These registers have a width of 128 bits, and can be addressed with 8, 16, 32, 64 or 128 bits as shown in Figure 8-1. The smallest value of 8 bits is *Bx* up to *Qx* which has a width of 128 bits. Even though there are 128 bits, floating point operations are limited to 64-bits.

Figure 8-1 V Register layout



The following listing will show the layout of data in the vector registers and confirm that the single precision layout of IEEE 754 is followed

Listing 8-1 Loading floating point values into vector registers (single precision)

```
//listing8-1
// Single precision floating-point
.text
.global _start
_start:
ldr x0, = floating01
ldr x1, = floating02
ldr s0, [x0] // Load into single precision s0 fp register
ldr s1, [x1] // Load into single precision s1 fp register
fadd s2, s0,s1      // Perform fp addition putting the result into s2
fmul s3, s0,s1      // Perform fp multiplication putting the result into s3
mov x8, #93
svc #0
.data
```

```
        floating01:   .single 1.414
        floating02:   .single 3.14
```

Listing 8-1 shows:

- Two single precision floating point numbers have been defined – 1.414 and 3.14.
- The addresses of these values are loaded into registers x0 and x1.
- The contents of the locations pointed to by the x registers are stored in the single word registers s0 and s1
- An addition of s0 and s1 is performed with the result shown in register s2
- A multiplication of registers 0 and s1 is performed with the result showing in s3

Use the GDB command `info vector` to show the contents of the vector registers[30].

```
v0             {d = {f = {0x3fb4fdf4, 0x0}, u = {0x3fb4fdf4, 0x0}, s =
{0x3fb4fdf4, 0x0}}, s = {f = {0x3fb4fdf4, 0x0, 0x0, 0x0}, u = {0x3fb4fdf4, 0x0,
0x0, 0x0}, s = {0x3fb4fdf4, 0x0, 0x0, 0x0}}, h = {bf = {0xfdf4, 0x3fb4, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0}, f = {0xfdf4, 0x3fb4, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, u =
{0xfdf4, 0x3fb4, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, s = {0xfdf4, 0x3fb4, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0}}, b = {u = {0xf4, 0xfd, 0xb4, 0x3f, 0x0 <repeats 12 times>}, s
= {0xf4, 0xfd, 0xb4, 0x3f, 0x0 <repeats 12 times>}}, q = {u = {0x3fb4fdf4}, s =
{0x3fb4fdf4}}}
```

```
..
s0    . . . s = 0x3fb4fdf4} {f = 1.41400003, . . .
s1    . . . s = 0x4048f5c3} {f = 3.1400001,  . . .
s2    . . . s = 0x4091ba5e} {f = 4.5539999,  . . .
s3    . . . s = 0x408e1428} {f = 4.43996048, . . .
```

If using the TUI with GDB then the command *tui reg float* will show the relevant floating-point registers[31]

There is a lot of information shown in the vector registers, normally we are only interested in a subset. The values shown in the full vector V0 list correspond to the unsigned and signed

[30] Use `p $vn` to show a specific vector register such as `p $v1`. Use `$v3.d` or just `p $d3` to just show the d part of the vector register. P /d $v1.b will show bytes in decimal format/

[31] The command `tui reg next` will cycle through the various register groups.

entries of the D,S,H,B and Q registers along with their signed and unsigned values. There are floating point representations given in the single precision (s) and double precision (d) registers.

The next listing uses `printf` to display three floating-point operations –

- Addition
- Multiplication
- Square root

Listing 8-2 Using printf to display floating-point values.

```
//listing 8-2
// Double precision floating-point
.text
  .global _start
  _start:
       ldr x0, = floating01
       ldr x1, = floating02
       ldr d0, [x0]    // Load into double precision d0 fp register
       ldr d1, [x1]    // Load into double precision d1 fp register
       ldr x0, =string1 // Free to use x0 again
       // Add and Multiply
       fadd d2, d0,d1  // Perform fp addition putting the result into d2
       fmul d3, d0,d1  // Perform fp multiplication putting the result into d3
       stp d0,d1,[sp, #-16]!      // Save d0 and d1
       stp d2,d3,[sp, #-16]!      // save d2 and d3
       bl printf
       ldp d2,d3, [sp],16  // Bring back the registers, observing LIFO
       ldp d0,d1, [sp],16

       // Square root
       ldr x0, =string2
       fsqrt d1, d0
       stp d0,d1,[sp, #-16]!   // Save d0 and d1
      stp d2,d3,[sp, #-16]!   // save d2 and d3
       bl printf
       ldp d2,d3, [sp],16      // Bring back the registers, observing LIFO

       ldp d0,d1, [sp],16

       mov x8, #0x5d

       svc #0

  .data

        floating01:     .double 1.414

        floating02:     .double 3.14

       string1: .asciz "The floating point number %f, added to the floating point
number %f, is %f,when multiplied the result is %f\n"

       string2: .asciz "The square root of register d0 containing %f, is %f\n"
```

Output

```
./listing8-2

The floating point number 1.414000, added to the floating point number 3.140000,
is 4.554000,when multiplied the result is 4.439960

The square root of register d0 containing 1.414000, is 1.189117
```

There are also precision specifiers that can be used for floats with `printf`. The default value is 6 (base 10) digits of precision which can be overridden by placing a point after % followed by a number to the left of the specifier as shown in the code snippet below.

```
       string1: .asciz "The floating point number %.3f, added to the floating
point number %.3f, is %.2f,when multiplied the result is %.2f\n"

       string2: .asciz "The square root of register d0 containing %.8f, is %.8f\n"
```

Output

```
The floating point number  1.414, added to the floating point number  3.140, is
4.55,when multiplied the result is 4.44

The square root of register d0 containing 1.41400000, is 1.18911732
```

Neon Coprocessor

The Neon coprocessor allows for parallel processing of operations. This is *termed Single Instruction Multiple Data* *(SIMD) since a single instruction operates on multiple pieces of data. The register set is shown in Figure 8-1 and allows for 128-bit processing across multiple *lanes* of data. There are 32 x 128-bit registers available referenced as *vn.t* where n stands for the vector register in question, t stands for the number of lanes and the data width. To take a specific example, v2.4s refers to vector register 2 broken up into 4 X 32-bit (S) paths. The data types available are:

8 bits (B) uint8 or sint8

- 16 bits (H) uint16 or sint16
- 32 bits (S)
- 64 bits (D)
- Single and double precision floats

A single128-bit vector register (bits 127:0) supports 2 X 64-bit, 4 X 32-bit, 8 X 16-bit, or 16 X 8-bit integer simultaneous operations. A single 64-bit bit vector register (bits 0:63) supports 2 X 32-bit, 4 X 16-bit, or 8 X 8-bit integer simultaneous operations.

Table 8-1 shows possible lane configurations.

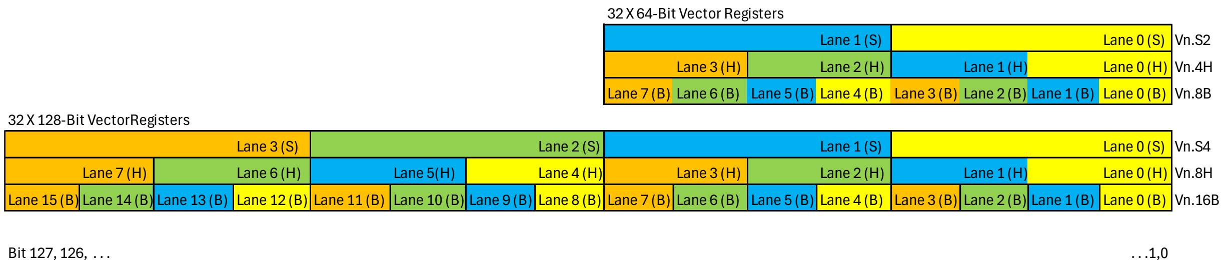
Table 8-1 Lane division in 128-bit / 64 bit vector registers

Register Size	Lane Width (B)	Lane Width (H)	Lane Width (S)	Lane Width (D)
128-bits (Q)	16 lanes x 8 (16B)	8 lanes x 16 (8H)	4 lanes x 32 (4S)	2 lanes x 64 (2D)
64-bits (D)	8 lanes x 8 (8B)	4 lanes x 16 (4H)	2 lanes x 32 (2S)	

A single lane represents a *scalar* value. Using only the low order 64-bits maintains 32-bit backward compatibility. Operations are performed in *parallel* on the individual lanes separately, not as a complete 64-bit or 128-bit register operation. The data size and lane layout is shown in Figure 8-2. Vector values are composed of multiple groups of numbers, for example a three-dimensional x,y,z co-ordinate could look like: 23, 42, -9 and be held in 3 different lanes.

A scalar instructions include the single lane designator such as V1.h[2].

Figure 8-2 Vector registers lane distribution



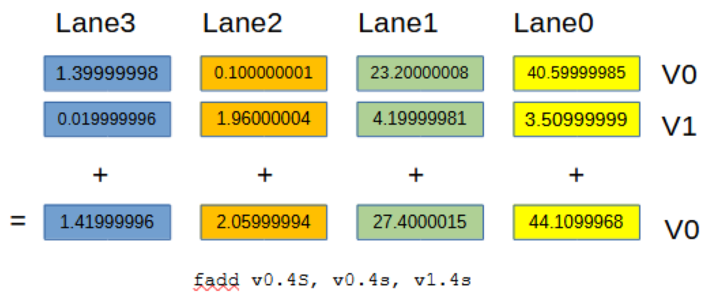
For the 128-bit vector registers there are:

- 16 Byte-wide lanes
- 8 Halfword-wide lanes

- 4 Singleword-wide lanes
- 2 Doubleword-wide lanes

Adding data from lane 1 in register V0 to the data in lane1 in vector register V1 is a completely independent operation. This is illustrated in Figure 8-3.

Figure 8-3 Four lane 128-bit floating-point addition



The code to generate the above data is shown in Listing 8-5.

Examples:

`mov1, v0.16b, #0x55` will load the vector register V0 with 16 bytes each byte having the value 0x55.

```
(gdb) p  /x $v0.q
$2 = {u = {0x55555555555555555555555555555555},
s = {0x55555555555555555555555555555555}}
```

Some examples of the move immediate (`movi`) instruction are shown in Listing 8-3.

Listing 8-3 Vector move instruction examples

```
// listing8-3
// Vector register examples
.text
  .global _start
_start:
  mov x0,#0xaa
  movi v0.16b, #0x55       // Q0 will contain 0x55555555555555555555555555555555
  movi v1.8b, #0x55        // D1 will contain 0x5555555555555555
  movi v2.8h, #0x55        // Q2 will contain 0x00550055005500550055005500550055
  movi v3.4h, #0x55        // D3 will contain 0x0055005500550055
  movi v4.4s, #0x55        // Q4 will contain 0x00000055000000550000005500000055
  movi v5.2s, #0x55        // D5 will contain 0x0000005500000055
  ins v6.b[10], v0.b[1]    // Insert vector element into v6 at index10, from v0,
index0
  cnt v7.16b, v0.16b       // Counts the # of ones in the specified elements of a
vector register, placing the result in another register
  // V7 now contains 0x0404040404040404
  dup v8.8b, w0     // V8 contains 0xaaaaaaaaaaaaaaaa aa duplicated across eight
bytes
  dup v0.2d, v7.d[0]       // v0 contains 0x404040404040404
 mov w8, #93
 svc #0
```

Note that ins is an alias for mov

400090: 6e150c06 `mov v6.b[10], v0.b[1]`

Again, single values are *scalar* values.

The eight-bit immediate values in the instruction, are actually held in non-contiguous locations. Looking at the disassembly for the instruction (= 0x4f02e6a0) `movi v0.16b, 0x55` , the immediate data is held in bits: (18:16) and (9:5). These bits are designated `(a,b,c)` and `(d,e,f,g,h)`. This is shown in Figure 8-4 where the eight bits correspond to 010 10101 = 0x55.

Refer to the ARM documents for a more complete breakdown of the remaining fields.

Figure 8-4 Layout of immediate data bits in the movi instruction

						movi v0.16b					4f02e6a0																				
	Q	op											a	b	c	cmode						d	e	f	g	h		Rd			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	1	0	0	0	0	0	0	1	0	1	1	1	0	0	1	1	0	1	0	1	0	0	0	0	0

Rd = V0

Imm8 = bits 18,17,16,9,8,7,6,5 specifies 8-bit operation

cmode 1110, op 0 = 8-bit;

Q 1 = 16Bytes (T=16b)

The next listing performs 16 addition operations. Two vector registers (V0 and V1) contain 16 bytes each. The additive result of all 16 bytes is placed in V0 overwriting the previous contents.

Listing 8-4 Adding sixteen bytes in parallel

```
// listing 8-4

// Vector register examples

.text

  .global _start

  _start:

       ldr x0,=values

       ldp q0, q1, [x0]

       add v0.16b, v0.16b, v1.16b

mov w8, #93

svc #0

.data

       values: .byte  1, 6, 3, 4, 9, -3, 7, 12, 9, 3, -4, 19, 5, 14, 3, 20, After
execution:23, 5, 7, 8, 10, 2, 4, 17, 3, 8, 45, 2, -4, 30, 4, 0
```

Initial contents of V0

1, 6, 3, 4, 9, -3, 7, 12, 9, 3, -4, 19, 5, 14, 3, 20, 1, 6, 3, 4, 9, -3, 7,12, 9, 3, -4, 19, 5, 14, 3, 20

Initial contents of V1

23, 5, 7, 8, 10, 2, 4, 17, 3, 8, 45, 2, -4, 30, 4, 0, 23, 5, 7, 8, 10, 2, 4, 17, 3, 8, 45, 2, -4, 30, 4, 0

Result in V0

24, 11, 10, 12, 19, -1, 11, 29, 12, 11, 41, 22, 1, 45, 7, 20, 24, 11, 10, 12, 19, -1, 11, 29, 12, 11, 41, 22, 1, 45, 7, 20

The next listing shows two operations:

1. How to load the 128-bit Q registers Q0 and Q1 with single floating-point word values and then to add these values in parallel, placing the result in Q0.
2. How to multiply each lane by a scalar quantity

Listing 8-5 Vector register addition and multiplication examples

```
   // listing 8-5

  // Vector register examples

  // 1. Floating point additions carried out in parallel

  // 2. Multiply by a scalar

  .text

    .global _start

    _start:

          ldr x0,=v0values
```

```
        ldr x1,=v2values
        ldp q0, q1, [x0]
        fadd v0.4s, v0.4s, v1.4s // Vector addition, lanes added in parallel
        ldp q0,q1, [x1]
        movi v2.4s, #5
        mul v0.8h, v1.8h, v2.h[0] // Multiplying by a scalar, each lane of V1 is
multiplied by 5 (lane0 of v2) with the result placed in V0
  // V0 now holds: 7700, 2600, 12845, 6455, 3940, 2585, 3900, 7835

  mov w8,#93
  svc #0
  .data
        v0values: .single 1.4, 0.1, 23.2, 40.6, 0.02, 1.96, 4.2, 3.51
        v2values: .byte 20, 34, 5, 9, -4, 10, 2, 7, 100, 40, 3, 8, 3, 4, 64, 56,
4, 6, 8, 2, 9, 10, 11, 5, 20, 3, 5    , 2, 12, 3, 31, 6
```

Note: Floating-point values can be shown within a vector register with the command p $v(register number>.<size>.f such as `p $v0.s.f`.

Lanes and data placement

Lanes can be referenced by an *index*. The `ld` instruction takes different forms. A non-exhaustive summary of instructions `ld1, ld2, ld3 and ld4` is presented in the following tables:

Table 8-2 Sample ldx (no offset) instructions

Instruction	Description	Example
ld1{vt.b}[index],Xn]	Loads a single element (8 - bits) to a single lane of a vector register	`Ld1{v0.b}[0],[x0] // Loads v0 with the single byte pointed to by x0, placing the data in lane0`
ld1{vt.h}[index],Xn]	Loads a single element (16 - bits) to a single lane of a vector register	`Ld1{v2.h}[3],[x0] // Loads v1 with the halfword pointed to by x0, placing the data in lane3`
ld1{vt.s}[index],Xn]	Loads a single element (32 - bits) to a single lane of a vector register	`Ld1{2.s}[0],[x0] // Loads v2 with the singleword pointed to by x0, placing the data in lane0`

ld1{vt.d}[index],Xn]	Loads a single element (64 -bits) to a single lane of a vector register	`Ld1{3.d}[0],[x0] // Loads v3 with the doubleword pointed to by x0`
ld2{vt.b,vt.2.b}[index],Xn]	Loads a two-element structure (8 -bits) to a single lane of two vector registers	`Ld2{v3.b,v4.b}[6],[x0] // Loads v3 and v4, lane6 with the byte pointed to by x0 and x0+1`
ld2{vt.h,vt2.h}[4],[xn]	Loads multiple byte structures into two vector registers	`ld2 {v5.h,v6.h,[x0] // Loads eight, 8-bit structures into registers v5 and v6, alternating the values pointed at by x0`
Ld2{vt.b,vt2.b}[index],Xn]	Loads a single two element (8 -bits) structure to a single lane of two vector registers	`ld2{v3.b,v4.b}[6],[x0] // Loads lane6 of v3 and v4 with the bytes pointed to at x0`
Ld2{vt.h,vt2.h}[index],Xn]	Loads a single two element (16 -bits) structure to a single lane of two vector registers	`ld2{v5.h,v6.h}[4],[x0] // Loads lane4 of v3 and v4 with the halfwords pointed to by x0`
Ld2{vt.s,vt2.s}[index],Xn]	Loads a single two element (32 -bits) structure to a single lane of two vector registers	`ld2{v7.s,v8.s}[0],[x0] // Loads lane0 of v3 and v4 with the word pointed to by x0`
Ld2{vt.d,vt2.d}[index],Xn]	Loads a single element two element (64 -bits) to a single lane of a vector register	`ld2{v3.d,v4.d}[2],[x0] // Loads lane2 of v3 and v4 with the doubleword word pointed to by x0`
Ld3{vt.b,vt2.b,vt3.b}[index],Xn]	Loads a single three element structure (8 -bits) to a single lane of three vector registers	`Ld3{v0.b,v1.b,v2.b}[0],[x0] // Loads lane0 of v0, lane0 of v1 and lane0 of v2 with the bytes pointed to by x0`

Ld3{vt.h,vt2.h,vt3.h}[index],Xn]	Loads a single three element (16 -bits) structure to a single lane of two vector registers	`Ld3{v5.h,v6.h,v7.h}[4],[x0] // Loads lane4 of v3, lane4 of 6 and lane4 of v7 with the halfword structures pointed to by x0`
ld3{vt.s,vt2.s,vt3.s}[index],Xn]	Loads a single three element (32 -bits) structure to a single lane of two vector registers	`ld3{vt.h,vt2.h,vt3.h}[index],Xn]`
ld3{vt.h,vt2.h,vt3.h}[index],Xn]	Loads a single two element (16 -bits) structure to a single lane of two vector registers	`ld3{vt.h,vt2.h,vt3.h}[index],Xn]`
ld4 {vt.b,vt2.b,vt3b,vt4.b},[x0]	Multiple 4-element structure, move to four registers with de-interleaving	`ld4 {v10.8b,v11.8b,v12,8b,v13,8b},[x0]`

Example of `ldx` instructions are shown in Listing 8-6.

Listing 8-6 ld1, ld2, ld3 and ld4 non-offset examples

```
// listing 8-6
// Vector register ldx examples
 .text
   .global _start
   _start:
   ldr x0,=values  // Set x0 to point at the 48 bytes in memory location (values)


   //Single Structure format of the instruction ld1, loading one element to one
lane.
   ld1 {v0.b}[0],[x0] //lane0 of v0 will contain the value 1
   ld1 {v1.b}[1],[x0] //lane1 of v1 will contain the value 1
   ld1 {v2.h}[3],[x0] //lane3 of v2 will contain the value 0x0601

   ld1 {v2.h}[2],[x0] // V2 now contains the value 0x601060100000000 (lanes 2 and
3 each    hold 0x601)


   //This is the multiple structure format of the instruction ld1, writing
multiple single elements to three registers

  ld1 {v0.8b, v1.8b, v2.8b},[x0]  // Loads multiple (8) single element byte
structures into v0, v1 and v2

  // V0 now holds unsigned bytes = {0x1, 0x6, 0x3, 0x4, 0x9, 0xfd, 0x7, 0xc, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}

  / v1 now holds unsigned bytes = {0x9, 0x3, 0fc, 0x13, 0x5, 0xe, 0x3, 0x14, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}

  // v2 now holds unsigned bytes = {0x17, 0x5, 0x7, 0x8, 0xa, 0x2, 0x4, 0x11,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}

  // Note values shown above are listed as ascending in memory contents of [x0]=
0x01, [x0]+1 = 0x06, . . .

  ld1 {v0.16b, v1.16b, v2.16b},[x0]  // Loads multiple (8) single element byte
structures into v0, v1 and v2


  // Single two-element structure format using ld2

  ld2 {v3.b,v4.b}[6],[x0] // lane6 of v3 holds 0x01 and lane6 of v4 holds 0x06

   ld2 {v5.h,v6.h}[4],[x0] // Lane4 of V5 contains 0x0106 and lane4 of v6 contains
0x0304

   ld2 {v7.s,v8.s}[0],[x0] // Lane7 of v7 contains 0x010603094 and lane 8 contains
0x09fd070c

   ld2 {v3.d,v4.d}[0],[x0] // Lane0 of v3 contains 0x0106030409fd070c and lane0 of
v4 contains 0x0903fc13050e0314


/* Multiple two-element structure format with de-interleaving. takes the data, and
puts the first element in register1, the second element in register2, third in
register 1, . . .*/

  ld2 {v5.8b,v6.8b},[x0] // Moves eight, byte structures into registers v5 and v6

  // v5 holds 0x01 0x03 0x09 0x07 0x09 0xfc 0x05 0x03

  // v6 holds 0x06 0x04 0xfd 0x0c 0x03 0x13 0x0e 0x14

  ld2 {v5.8h,v6.8h},[x0]

  // v5 holds 0x0601 0xfd09 0x0309 0x050e

  // v6 holds 0x0403 0x0c07.
// Multiple three-element structure move to three registers with de-interleaving
  ld3 {v7.4H, v8.4H,v9.4H},[x0]
  // v7 holds 0x0601, 0x070c, 0x050e, 0x0807
  // v8 holds 0x0403, 0x0309, 0x1403, 0x200a
  // v9 holds 0xfd09, 0x13fc, 0x0517, 0x1104
  ld3 {v7.2d-v9.2d}, [x0] // Note the syntax Vm-Vn is also valid
  // v7 holds 0x0106030409fd070c0
  // v8 holds 0x0903fc13050e0314
  // v9 holds 0x170507080a020411

  // Single four-element structure to one single lane of four registers
  ld4 {v15.s,v16.s,v17.s,v18.s}[2],[x0]
  // v15 holds 0x04030601
  // v16 holds 0x0c07fd09
  // V17 holds 0x13fc0309
  // V18 holds 0x14030e05

  // Multiple four-element structure move to four registers with de-interleaving
  ld4 {v10.8b,v11.8b,v12.8b,v13.8b},[x0]
  // v10 holds 0x01, 0x09, 0x09, 0x05, 0x17, 0x0a, 0x03, 0xfc
  // v11 holds 0x06, 0xfd, ox03, 0x0e, 0x05, 0x02, 0x08, 0x1e
  // v12 holds 0x03, 0x07, 0xfc, ox03, 0x07, 0x04, 0x2d, 0x04
  // v13 holds 0x04, 0x0c, 0x13, 0x14, 0x08, 0x11, 0x02, 0x00

  mov w8, #93
  svc #0
.data
 values: .byte  1, 6, 3, 4, 9, -3, 7, 12, 9, 3, -4, 19, 5, 14, 3, 20, 23, 5, 7, 8,
10, 2, 4, 17, 3, 8, 45, 2, -4, 30, 4, 0, 2, 5, 9, 2, 11, 5, 14, 0, 23, 44, 21, 5,
13, 14, 15, 16
```

Note when using multiple registers, they must be consecutive in number. The reason for this is that the last 5 bits of the Rt field (see Table 8-4) is used to encode the Vt registers. This is shown in Table 8-3:

Table 8-3 ld4 instruction Rt field

Vt encoding	Vt2 encoding	Vt3 encoding	Vt4 encoding
Bits 4:0	((Bits 4:0) +1), modulo 32	((Bits 4:0) +2), modulo 32	((Bits 4:0) +3), modulo 32

The disassembly for the instruction ld4 `{v15.s-v18.s}[2], [x0] is 4d60a00f.`

Table 8-4 gives a breakdown of the bit fields.

Table 8-4 Bit fields of the ld4 instruction

Breakdown of `{v15.s-v18.s}[2], [`Instruction 4d60a00f

	Q								L	R	Rm				o2	Opcode			S	Size		Rn					Rt				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1

Interpretation

1	Rt	vt0
	Rt+1, modulo32	Vt1
	Rt+2, modulo32	Vt2
	Rt+3, modulo32	Vt3

2	Rn	Xn|SP	(x0)
3	Rm	Post-index register	
4	Opcode	101&&S=00 = 32-bit	
5	Element Index encoded in Q:S = 10	Q:S for 32-bit (S)	

As seen the functionality of the fields is well thought out and gives a lot of capability for an instruction that is only 32-bits wide.

The `ldx` instructions also have *Post-Index* capability. The offset can be *register* or *immediate*. The format of `ld4` with an eight-bit register offset `is LD4 { vt.b,vt2.b, vt3.b,vt4.b }[index], [Xn|SP], <Xm>.`

The last `ldx` instruction to consider is `ld` with replicate. Here the `ld4r` instruction has the no-offset format of `LD4r {vt.T,vt2.T,vt3.T,vt4.T},[Xn|SP}.` Its function is to load a single four-element structure and replicate it to all four lanes of the four registers.

This instruction has similar variants to the `ldx` instructions shown in Table 8-2. Listing 8-7 shows a brief example.

Listing 8-7 ld4r instruction

```
// listing 8-7

// Vector register example ld4R

.text

  .global _start

  _start:

       ldr x0,=values      // Set x0 to point at the 48 bytes in memory location
(values)

       ld4r {v0.16b,v1.16b,v2.16b,v3.16b,[x0]

       // v0 = 0x0101010101010101

       // v1 = 0x0606060606060606

       // V2 = 0x0303030303030303

       // V3 = 0x0404040404040404

       ld4r {v0.4h,v1.4h,v2.4h,v3.4h,[x0]

       // v0 = 0x0601060106010601

       // v1 = 0x0403040304030403

       // V2 = 0xfd09fd09fd09fd09

       // v3 = 0x0c070c070c070c07

       mov w8, #93

       svc #0

.data

             values: .byte  1, 6, 3, 4, 9, -3, 7, 12, 9, 3, -4, 19, 5, 14, 3, 20,
23, 5, 7, 8, 10, 2, 4, 17, 3, 8, 45, 2, -4, 30, 4, 0, 2, 5, 9, 2, 11, 5, 14, 0,
23, 44, 21, 5, 13, 14, 15, 16
```

Permutations and Interleaving

Zip and uzp

There are several options for permuting data. The zip instruction alternatively fetches elements from a pair of registers, placing the result in a third register. The instruction uses two source registers and one destination which can only accommodate half of the data. In the case of 128-bit Q registers, two destination registers are required to interleave all of the elements.This is achieved by performing two zip instructions. The zip instruction uses two forms – `zip1` and `zip2`. The first form `zip1` stores the low order bytes (bytes0:byte7) into a destination register and the second form `zip2` stores the high order bytes (byte15:byte8) into a second destination register.

This is shown in Figure 8-5.

The counterpart of zip is uzp to perform the opposite task with the instruction `uzp1`working on the low order and `uzp2` working on the high order. Listing 8-8 gives an example. The bytes have consecutive values making for easy interpretation during the interleaving process.

Reversing elements

The *reverse* (rev) instruction preserves the order of the elements but reverses the byte order. Examples of word and half word reversals are shown in Figure 8-6.

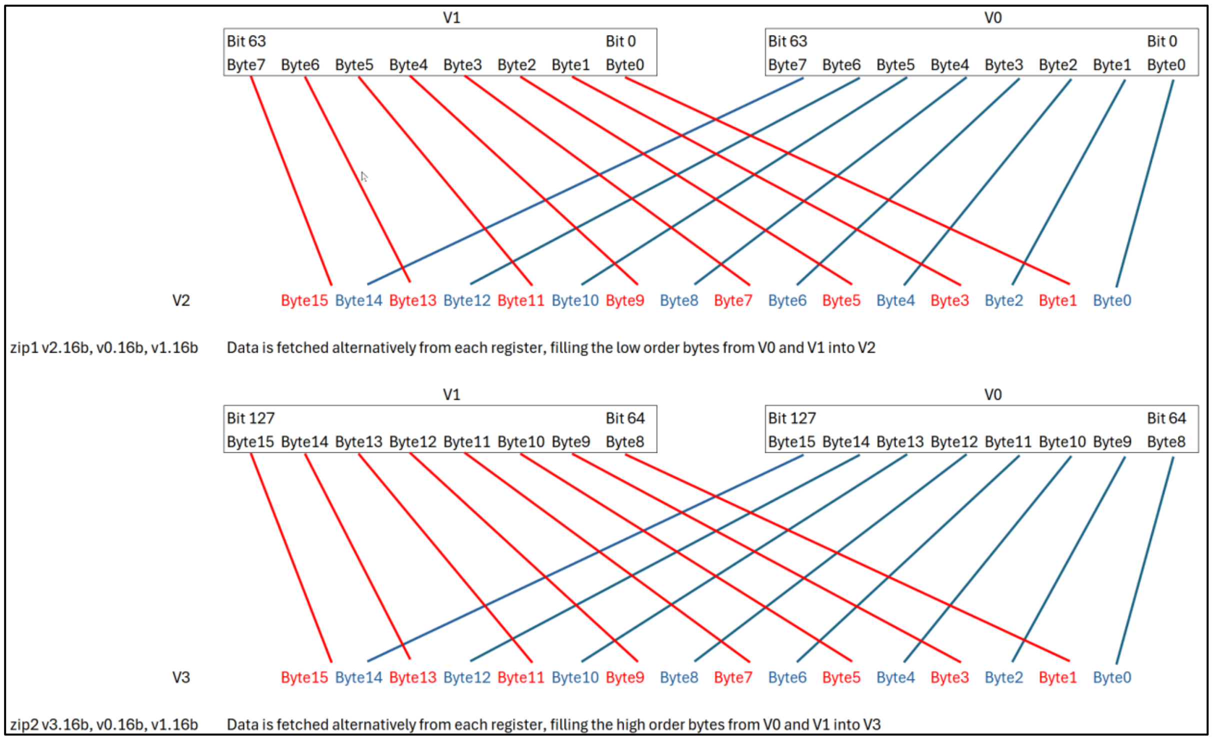
Extraction of elements extracts a number of elements from one register with the balance coming from another register. The combination is then placed in a destination register

Extraction of elements is accomplished with the `ext` command.Listing 8-8 gives an example where the instruction extracts the top ten bytes from v0, writing them to the bottom ten bytes of v9 and then writes the lower six bytes from v1 to the remaining high order six bytes of v9.

The xtn instruction (extend and narrow) takes the lower 32 bits of each half of V0 and stores them in a destination register.

The xtn2 instruction takes the upper 32 bits of each half of a register and stores them in a destination register.

Figure 8-5 Use of zip instruction


V1
Bit 63
Bit 0
Byte7 Byte6 Byte5 Byte4 Byte3 Byte2 Byte1 Byte0
V0
Bit 63
Bit 0
Byte7 Byte6 Byte5 Byte4 Byte3 Byte2 Byte1 Byte0
V2
Byte15 Byte14 Byte13 Byte12 Byte11 Byte10 Byte9 Byte8 Byte7 Byte6 Byte5 Byte4 Byte3 Byte2 Byte1 Byte0
zip1 v2.16b, v0.16b, v1.16b
Data is fetched alternatively from each register, filling the low order bytes from V0 and V1 into V2
V1
Bit 127
Bit 64
Byte15 Byte14 Byte13 Byte12 Byte11 Byte10 Byte9 Byte8
V0
Bit 127
Bit 64
Byte15 Byte14 Byte13 Byte12 Byte11 Byte10 Byte9 Byte8
V3
Byte15 Byte14 Byte13 Byte12 Byte11 Byte10 Byte9 Byte8 Byte7 Byte6 Byte5 Byte4 Byte3 Byte2 Byte1 Byte0
zip2 v3.16b, v0.16b, v1.16b
Data is fetched alternatively from each register, filling the high order bytes from V0 and V1 into V3

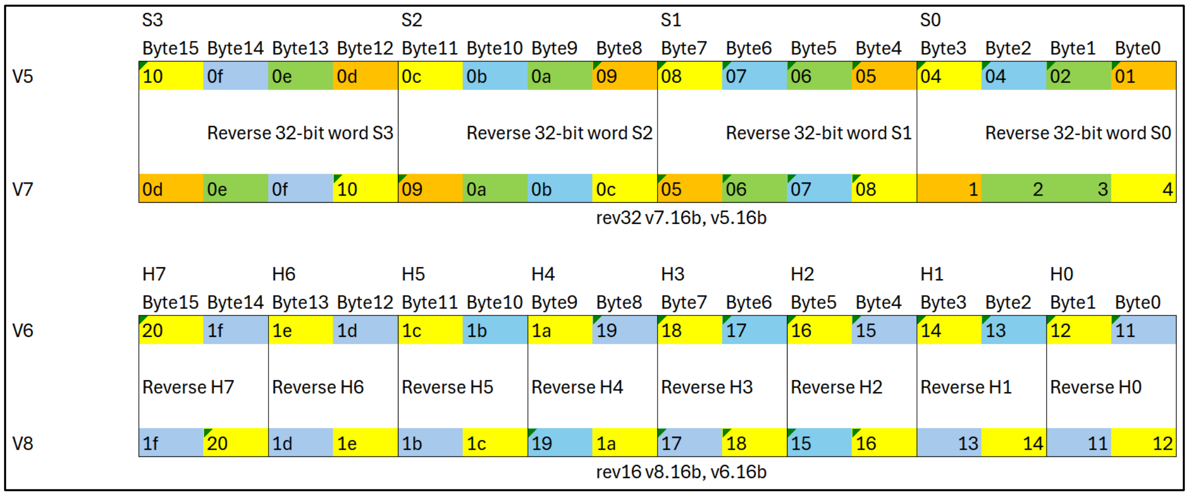
Transposition

Transposing elements takes an odd numbered elements (bytes) from two registers placing them in sequence to a third destination register. An example of the syntax is:

```
trn1 v12.16b, V0.16b, v1.16b
```

The counterpart to `trn1` is `trn2` which takes the even numbered elements (words) from two registers placing them in sequence to a third destination register.

Figure 8-6 Rev instruction


S3
S2
S1
S0
Byte15 Byte14 Byte13 Byte12 Byte11 Byte10 Byte9 Byte8 Byte7 Byte6 Byte5 Byte4 Byte3 Byte2 Byte1 Byte0
V5
10 0f 0e 0d 0c 0b 0a 09 08 07 06 05 04 04 02 01
Reverse 32-bit word S3
Reverse 32-bit word S2
Reverse 32-bit word S1
Reverse 32-bit word S0
V7
0d 0e 0f 10 09 0a 0b 0c 05 06 07 08 1 2 3 4
rev32 v7.16b, v5.16b
H7
H6
H5
H4
H3
H2
H1
H0
Byte15 Byte14 Byte13 Byte12 Byte11 Byte10 Byte9 Byte8 Byte7 Byte6 Byte5 Byte4 Byte3 Byte2 Byte1 Byte0
V6
20 1f 1e 1d 1c 1b 1a 19 18 17 16 15 14 13 12 11
Reverse H7
Reverse H6
Reverse H5
Reverse H4
Reverse H3
Reverse H2
Reverse H1
Reverse H0
V8
1f 20 1d 1e 1b 1c 19 1a 17 18 15 16 13 14 11 12
rev16 v8.16b, v6.16b

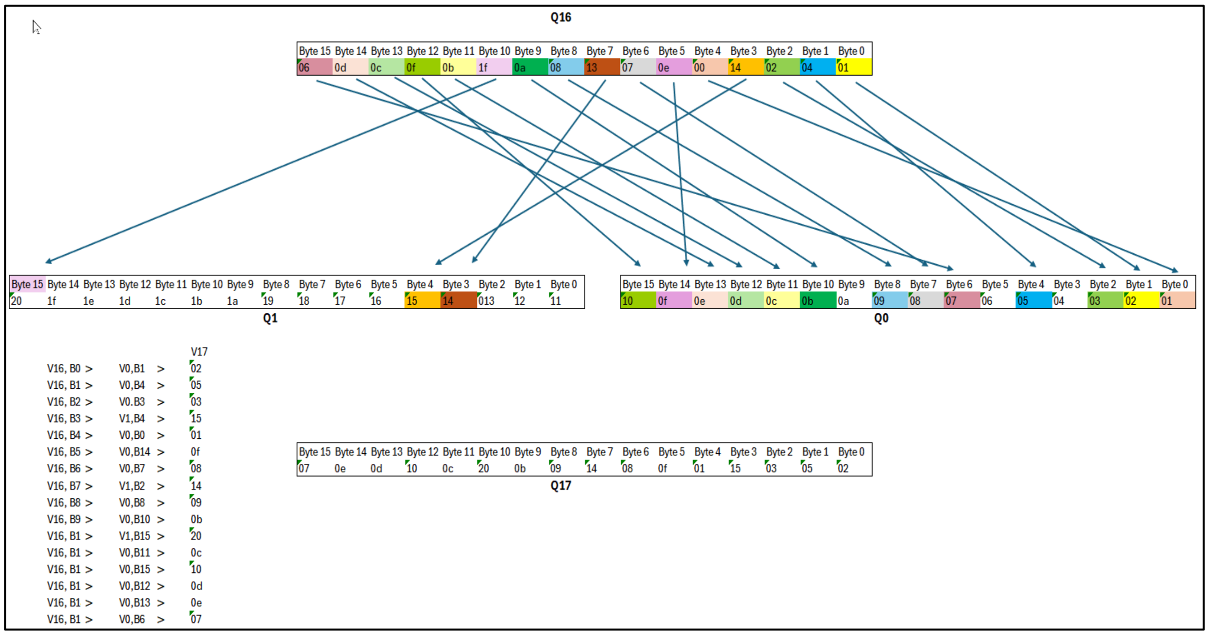
Lookup

The final permutation instruction looked at is `tbl` which uses a vector register to hold *lookup* values which index into a group of registers that hold the data which will be sent to a destination register. An example of the syntax is:

```
tbl v17.16b, {v0.16b,v1.16b},v16.16b
```

Figure 8-7 shows an example of a lookup,

Figure 8-7 Use of a lookup table to change less structured element lists


Q16
Byte 15 Byte 14 Byte 13 Byte 12 Byte 11 Byte 10 Byte 9 Byte 8 Byte 7 Byte 6 Byte 5 Byte 4 Byte 3 Byte 2 Byte 1 Byte 0
06 0d 0c 0f 0b 1f 0a 08 13 07 0e 00 14 02 04 01
Byte 15 Byte 14 Byte 13 Byte 12 Byte 11 Byte 10 Byte 9 Byte 8 Byte 7 Byte 6 Byte 5 Byte 4 Byte 3 Byte 2 Byte 1 Byte 0
20 1f 1e 1d 1c 1b 1a 19 18 17 16 15 14 013 12 11
Q1
Byte 15 Byte 14 Byte 13 Byte 12 Byte 11 Byte 10 Byte 9 Byte 8 Byte 7 Byte 6 Byte 5 Byte 4 Byte 3 Byte 2 Byte 1 Byte 0
10 0f 0e 0d 0c 0b 0a 09 08 07 06 05 04 03 02 01
Q0
V17
V16, B0 > V0,B1 > 02
V16, B1 > V0,B4 > 05
V16, B2 > V0.B3 > 03
V16, B3 > V1,B4 > 15
V16, B4 > V0,B0 > 01
V16, B5 > V0,B14 > 0f
V16, B6 > V0,B7 > 08
V16, B7 > V1,B2 > 14
V16, B8 > V0,B8 > 09
V16, B9 > V0,B10 > 0b
V16, B1 > V1,B15 > 20
V16, B1 > V0,B11 > 0c
V16, B1 > V0,B15 > 10
V16, B1 > V0,B12 > 0d
V16, B1 > V0,B13 > 0e
V16, B1 > V0,B6 > 07
Byte 15 Byte 14 Byte 13 Byte 12 Byte 11 Byte 10 Byte 9 Byte 8 Byte 7 Byte 6 Byte 5 Byte 4 Byte 3 Byte 2 Byte 1 Byte 0
07 0e 0d 10 0c 20 0b 09 14 08 0f 01 15 03 05 02
Q17

Listing 8-8 Interleaving data from the vector registers

```
// listing 8-8
// Vector register permutations
 .text
   .global _start
   _start:
         ldr x0,=avalues // Set x0 to point at the 16 bytes in memory location
(avalues)
         ldr x1,=bvalues // Set x1 to point at the 16 bytes in memory location
(bvalues)
         ldr x2,=lookupvalues


         ld1 {v0.2d},[x0]
        ld1 {v1.2d},[x1]         ld1 {v16.2d},[x2]
         // q0 = 0x100f0e0d0c0b0a09 0807060504030201
         // q1 = 0x201f1e1d1c1b1a19 1817161514131211
         // q2 = 0x060d0c0f0b1f0a08 13070e0014020401
          zip1 v2.16b, v0.16b, v1.16b // q2 now has interleaved low order bytes
from q0 and q1
         // q2 = 0x1808170716061505 1404130312021101


         zip2 v3.16b, v0.16b, v1.16b // q3 now has interleaved high order bytes
from q0 and q1
         // q3 = 0x20101fof1e0e1d0d 1c0c1b0b1a0a1009


         zip1 v4.16b, v1.16b, v0.16b // Change order of source registers
         // q4 = 0x0818071706160515 0414031302120111


         uzp1 v5.16b, v2.16b, v3.16b //Unscramble low order bytes, result in q5
         uzp2 v6.16b, v2.16b, v3.16b //Unscramble high order bytes, result in q6
         // Long-winded way of copying q0 to q5 and q1 to q6


         rev32 v7.16b, v5.16b // reverses bytes within each word element
         rev16 v8.16b, v6.16b // reverses bytes within each halfword element
         // Extraction

         ext v9.16b, v0.16b, v1.16b, #6

         // Extracts top 10 bytes from v0, writing them to bottom 10 bytes of v9
and writes the lower 6 bytes from v1 to the remaining high order bytes of v9

         // v9 now contains 0x161514131211100f0e0d0c0b0a090807

         xtn v10.2s, v0.2d // Extend and narrow takes the lower 32 bits of each
half of V0 and stores them in v10 giving 0x0c0b0a0904030201

        // v10 contains 0x0c0b0a09 04030201

        xtn2 v10.8h, v0.4s

         /* Takes the upper 32 bits of each half of V0 and stores them in v10
giving 0x100f0e0d0b0a090807060504030201;

         since the previous instruction  wrote to the lower half already and the
instruction does not affect the other bits*/


          // Transposition

          trn1 v12.16b, V0.16b, v1.16b // Takes the odd numbered elements (bytes)
from v0 and v1 placing them in sequence to V12

          // V12 contains 0x1f0f1d0d1b0b19091707150513031101

         trn2 v13.4s, v0.4s, v1.4s // Takes the even numbered elements (words)
from v0 and v1 placing them in sequence to V13

         // V13 now contains 0x201f1e1d100f0e0d1817161508070605

         // Lookup Tables

          // tbl uses a vector register to hold lookup values which index into a
group of registers that hold the data which will be sent to a destination register

         tbl v17.16b, {v0.16b,v1.16b},v16.16b

          V17 now contains 070e0d100c200b0914080f0115030502

  mov w8, #93

  svc #0
```

Summary of chapter 8

- SIMD
- Layout of the vector registers
- Floating-point operations
- Scalar and vector operations
- Permutations and interleaving

Exercises for chapter8

1. Generate a program to multiply 4 floating point numbers together using SIMD instructions
2. Explain the difference between scalar and vector values
3. Is the instruction add v0.8s, v0.8s, v1.8s valid? Explain your answer
4. (Advanced) Generate the inverse of a three by three matrix, using single precision floats, then multiply the result by the original matrix, comment on the answer
5. Explain the action of the `rev` instruction.

Chapter 9. Cross Compilation

Cross compiling[32] allows development of programs on machines with a different architecture. In this section cross compilation will be performed on a Linux machine running Debian –

```
uname -a

Linux debian1 6.1.0-26-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.112-1 (2024-09-30)
x86_64 GNU/Linux
```

Start by following the steps listed below

Step 9-1. Install the necessary tools

```
sudo apt install gcc make gcc-aarch64-linux-gnu binutils-aarch64-linux-gnu

[sudo] password for alan:

Reading package lists... Done

Building dependency tree... Done

Reading state information... Done

gcc is already the newest version (4:12.2.0-3).

gcc set to manually installed.

make is already the newest version (4.3-4.1).

make set to manually installed.

The following additional packages will be installed:

  cpp-12-aarch64-linux-gnu cpp-aarch64-linux-gnu gcc-12-aarch64-linux-gnu

  gcc-12-aarch64-linux-gnu-base gcc-12-cross-base libasan8-arm64-cross

  libatomic1-arm64-cross libc6-arm64-cross libc6-dev-arm64-cross

  libgcc-12-dev-arm64-cross libgcc-s1-arm64-cross libgomp1-arm64-cross

  libhwasan0-arm64-cross libitm1-arm64-cross liblsan0-arm64-cross

. . .
```

[32] The following link may be helpful `Cross-compiler | Arm Learning Paths` `https://learn.arm.com/install-guides/gcc/cross/)`

Step 9-2. Create `helloworld.c` file

```
cat helloworld.c

#include <stdio.h>

int main()

{

	printf("Hello World");

	return 0;

}
```

Step 9-3. Compile the program using the ARM64 gcc compiler

```
aarch64-linux-gnu-gcc helloworld.c -o helloworld-arm64
```
[33]

Step 9-4. Execute the program

```
$ ./helloworld-arm64

bash: ./helloworld-arm64: cannot execute binary file: Exec format error
```

This is to be expected as the ARM64 program is running on X86 architecture!

Step 9-5. Check the file format

```
file helloworld-arm64

helloworld-arm64: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
BuildID[sha1]=7d70ff2387ca56fe82e50f708c75aa3f47209127, for GNU/Linux 3.7.0, not
stripped
```

The output of the file command indicates that the executable is ARM aarch64.

Step 9-6. Verify that the program runs correctly by transferring it (if available) to an ARM64 based system

```
 scp helloworld-arm64 pi5b:/home/alan/asm

alan@pi5b's password:
```

[33] Appending `-static` to the compilation string will invoke static linking and may help since it includes the necessary dependencies.

```
helloworld-arm64                                              100%   69KB
4.0MB/s   00:00

ssh pi5b

alan@pi5b's password:

Linux pi5b 6.6.31+rpt-rpi-2712 #1 SMP PREEMPT Debian 1:6.6.31-1+rpt1 (2024-05-29)
aarch64


The programs included with the Debian GNU/Linux system are free software;

the exact distribution terms for each program are described in the

individual files in /usr/share/doc/*/copyright.


Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent

permitted by applicable law.

Last login: Mon Oct 14 15:01:08 2024

alan@pi5b:~ $ cd asm

alan@pi5b:~/asm $ chmod +x helloworld-arm64

alan@pi5b:~/asm $ ./helloworld-arm64

Hello Worldalan@pi5b:~/asm $
```

Cross compiling assembly code

Create the assembly file below:

```
.global main

main: mov x3, #0xf0f0f0f0f0f0f0f0

       mov w4, w3    // Read from w3

       mov w3, w4    // Write to w3

       svc 0
```

Assemble it with -

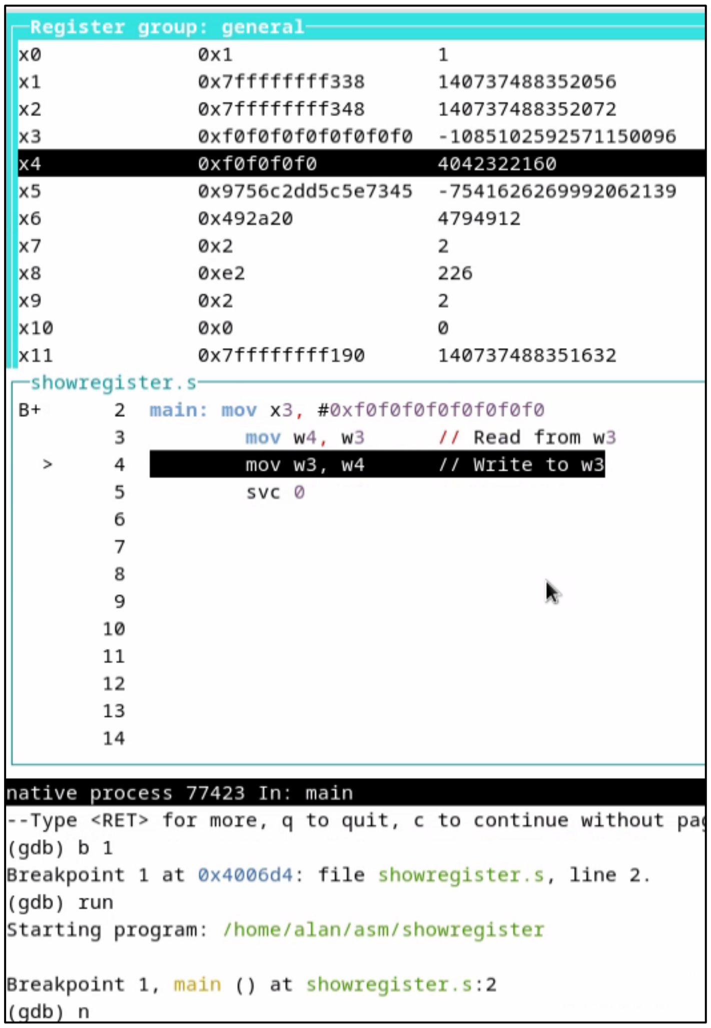
```
aarch64-linux-gnu-as -g -o showregister.o showregister.s

aarch64-linux-gnu-gcc -static -o showregister showregister.o
```

Copy the file to an ARM64 system.

```
scp showregister pi5b:/home/alan/asm
```

The program should now run with GDB as shown in Figure 9-1

Figure 9-1 Running a cross-compiled program with GDB



```
┌─Register group: general──────────────────────────────
x0             0x1                 1
x1             0x7fffffff338       140737488352056
x2             0x7fffffff348       140737488352072
x3             0xf0f0f0f0f0f0f0f0  -1085102592571150096
x4             0xf0f0f0f0          4042322160
x5             0x9756c2dd5c5e7345  -7541626269992062139
x6             0x492a20            4794912
x7             0x2                 2
x8             0xe2                226
x9             0x2                 2
x10            0x0                 0
x11            0x7fffffff190       140737488351632
┌─showregister.s───────────────────────────────────────
B+      2  main: mov x3, #0xf0f0f0f0f0f0f0f0
        3          mov w4, w3        // Read from w3
  >     4          mov w3, w4        // Write to w3
        5          svc 0
        6
        7
        8
        9
       10
       11
       12
       13
       14

native process 77423 In: main
--Type <RET> for more, q to quit, c to continue without pag
(gdb) b 1
Breakpoint 1 at 0x4006d4: file showregister.s, line 2.
(gdb) run
Starting program: /home/alan/asm/showregister

Breakpoint 1, main () at showregister.s:2
(gdb) n
```

The QEMU emulator supports ARM64 based virtual machines on X86 architectures. For further information consult qemu.com or the many resources found on the Internet. The Ubuntu documentation describes this and can be found at `https://documentation.ubuntu.com/server/how-to/virtualisation/arm64-vms-on-qemu/`

Summary of chapter 9

- Cross compilation tools
- Testing and executing
- QEMU Virtualization

Exercises for chapter 9

1. Using an X86 based platform, install the necessary tools to cross compile an ARM64 based program and then verify that ARM64 code runs successfully on an ARM64 platform

2. Generate an ARM based VM running on X86 under QEMU.