Computer Science from the
Metal Up

Assembly Language Programming

Richard Pawson

with
Peter Higginson

Computer Science from the Metal Up:

Assembly Language Programming

Richard Pawson

with Peter Higginson

V1.0.0

Introduction to the Teacher Edition

This book provides a thorough introduction to the principles and practices of assembly language programming, appropriate to an A-level Computer Science curriculum. A hands-on approach is adopted throughout with many practical exercises. Specifically, the reader will explore and develop several games, starting very simple, but ending up satisfyingly complex.

The book makes heavy use of Peter Higginson's ARMlite simulator, a refined and significantly extended version of his earlier processor simulators, this one written specially to fit with the book. ARMlite simulates a cut down version of the ARM processor.

ARMlite has been designed specifically to implement the AQA assembly language instruction set. However, teachers should be aware that AQA has not specified the operation of its instruction set in sufficient detail to permit a rigorous implementation. Appendix I describes the small number of *potential* differences. Teachers should familiarise themselves with Appendix I, especially the point about byte-addressing vs. word-addressing. These small potential differences are likely to lead to serious misunderstandings, and are best not raised early on. However, it is recommended that students read and understand Appendix I before they move on to tackling AQA exam questions.

Relationship to AQA

This book is not endorsed, or approved, by AQA - nor has any such endorsement/approval been sought by the author.

The book is divided into two parts.

Book I – Fundamentals of assembly language Programming

In the author's opinion, Book I (Chapters 1 – 4) covers the full requirements of the AQA A-Level Computer Science specification (7517) in relation to Machine-code/assembly language operations (section 4.7.3.5). However, *teachers should verify this coverage for themselves*.

Book II – Delving a little deeper

Book II introduces several new assembly language programming techniques, as well as specific new instructions.

While these all fall outside the requirements of the AQA specification for machine code / assembly language programming, they are nonetheless well worth learning. Not only will they permit the student to write more interesting programs, but they will reinforce multiple aspects of the Computer Science curriculum, including:

- Algorithms and Data Structures
- Processor operation (especially the role and functioning of Interrupts)
- High level programming patterns

Book II finishes, in Chapter 8, with the guided development of a complete Snake game, making use of all the techniques introduced in previous chapters.

Potential for AQA NEA Projects in assembly language
Completing the exercises in Book II, would equip a student that enjoys the challenge of assembly language Programming, to consider undertaking their complete NEA A-level project in assembly language, running on ARMlite. The author would argue that this inherently hits the target for a 'Tier 1' technical skill, and has experience of supervising and marking at least one such project, but teachers might like to check with their own AQA advisors if they have any doubts.

Side bars
Throughout the book there are 'Side bars' (boxes with blue text) ,which provide interesting historical background, or introduce tangential technical points. However, these side bars may be safely skipped or omitted without fear of missing any aspect of the specification.

Exercises
Practical exercises are offered throughout the book, with model answers supplied in this Teacher's Edition.

EAD
An Electronic Answer Document (EAD) is available as a Word document, to allow students to paste in their answers and screenshots, to facilitate assessment by the teacher.

Subject pre-requisites
Readers are expected to be familiar with binary and hexadecimal representations, and with binary arithmetic.

It is not necessary to have covered the architecture of the processor before starting this book, but it might be helpful.

It is not expected that students have previous experience of two's complement representation, as it is explained when introduced. However, if the students have not previously covered this topic, they will need further practice on using it outside this context.

Technical pre-requisites
All the programming exercises may be run on the online ARMlite simulator (https://peterhigginson.co.uk/ARMlite/). The Chrome browser is recommended, but the simulator should work on most browsers. No pre-installation is necessary.

Book I – Fundamentals of assembly language

Teacher note

In the author's opinion, Book I (Chapters 1 – 4) covers the full requirements of the AQA A-Level Computer Science specification (7517) in relation to Machine-code/assembly language operations (section 4.7.3.5). However, *teachers should verify this coverage for themselves*.

Book I introduces only the instructions used in the AQA Assembly Language instruction set, and using only the direct address mode for memory references. This severely limits the kind of programs that can be written. Nevertheless, by Chapter 4 pupils will have written two simple games in assembly language.

Chapter 1. Introduction to assembly language and ARMlite

The program listed below is written in assembly language – it might look very unfamiliar to you! Assembly language is a 'low-level' programming language, each instruction (line of code in this case) performs a very simple operation, and it might take many such instructions to match the functionality of a single line of code in a 'high-level' language (such as Python, VB, or C#).

```
    MOV R1, #.PixelScreen
    MOV R2, #screen2
    MOV R6, #0
    MOV R9, #.black
    MOV R10, #.white
    MOV R3, #0
loopWhite: STR R10, [R2+R3]
    ADD R3, R3, #4
    CMP R3, #12288
    BLT loopWhite
    MOV R3, #260
randLoop: LDR R0, .Random
    AND R0, R0, #1
    CMP R0, #0
    BNE .+2
    STR R9, [R2+R3]
    BL nextCell
    CMP R3, #12032
    BLT randLoop
copyScreen2to1: MOV R3, #0
copyLoop: LDR R0, [R2+R3]
    STR R0, [R1+R3]
    ADD R3, R3, #4
    CMP R3, #12288
    BLT copyLoop
    ADD R6, R6, #1
    MOV R3, #260
nextGenLoop: MOV R5, #0
    SUB R4, R3, #256
    BL countIfLive
    SUB R4, R3, #252
    BL countIfLive
    ADD R4, R3, #4
    BL countIfLive
    ADD R4, R3, #260
    BL countIfLive
    ADD R4, R3, #256
    BL countIfLive
    ADD R4, R3, #252

    BL countIfLive
    SUB R4, R3, #4
    BL countIfLive
    SUB R4, R3, #260
    BL countIfLive
    CMP R5, #4
    BLT .+3
    STR R10, [R2+R3]
    B continue
    CMP R5, #3
    BLT .+3
    STR R9, [R2+R3]
    B continue
    CMP R5, #2
    BLT .+2
    B continue
    STR R10, [R2+R3]
continue: BL nextCell
    MOV R0, #12032
    CMP R3, R0
    BLT nextGenLoop
    B copyScreen2to1
countIfLive: LDR R0, [R1+R4]
    CMP R0, R10 //White
    BEQ .+2
    ADD R5, R5, #1
    RET
nextCell:
    ADD R3, R3, #4
    AND R0, R3, #255
    CMP R0, #0
    BEQ .-3
    CMP R0, #252
    BEQ .-5
    RET
    HALT
    .ALIGN 1024
screen2: 0
```

Each instruction in assembly language corresponds to an operation that can be executed directly by electronic circuits in the processor. Different processors therefore have different assembly languages, though there are many common features - the language shown here is for a 32-bit ARM processor.

A processor can't execute the assembly language directly: each line of code must be translated into a 'machine code' first – a process known as 'assembling' and the tool for performing the conversion is known as an assembler – but each assembly language instruction results in one 32-bit code.

Throughout this book we are going to be using an online program called ARMlite, which *simulates* a simple computer built around a cut-down version of a 32-bit ARM processor:
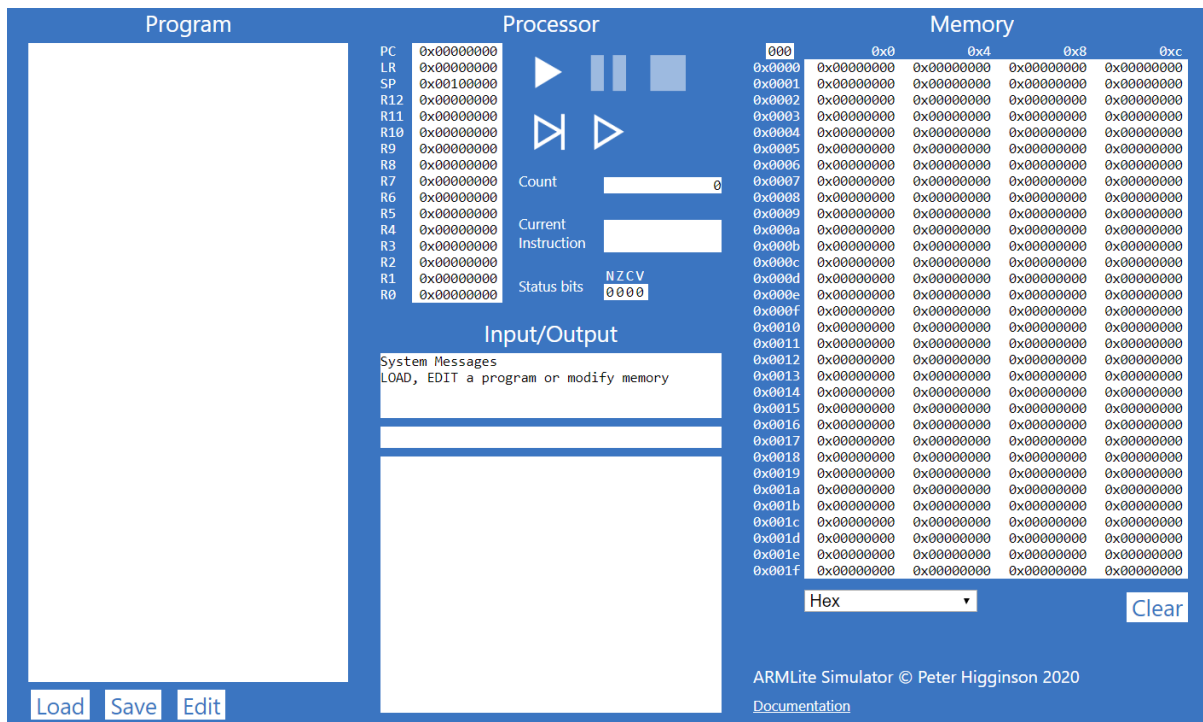
You can see that the screen is divided into four main areas: **Program**, **Processor**, **Memory**, and **Input/Output**.

The **Memory** is made up of 32-bit words. In the view above, each word contains zero, but this is shown as `0x00000000`. The `0x` is a standard prefix indicating that what follows is in hexadecimal (hex) format. Each hex digit corresponds to 4-bits – so there are eight hex digits.

Exercise 1

Access the simulation via: https://peterhigginson.co.uk/ARMlite/ preferably using Chrome. (Most modern browsers should work but IE11 does not).

Click on any visible memory word and type in **101** (followed by the Return key).

What value is displayed, and why?

On another memory word, enter **0x101**

What value is displayed, and why?

On another memory word, enter **0b101**

What value is displayed, and why?

If you now *hover* (don't click) the mouse over any of the memory words where you have entered a value you will get a pop-up 'tooltip'. What does the tooltip tell you?

The drop-down selector shown here: `Hex ▼` allows you to change the base in which data is *displayed*. Changing the base does not change the underlying data value.

Change to **Decimal (unsigned)**. Paste a partial screenshot showing all three of the memory words that you entered, in their new format.

When you the mouse over one of these words, what now appears in the tooltip?

Does changing the representation of the data in memory also change the representation of the row- and column-headers (the white digits on a blue background)?

Model answer 1

Entering **101** gives **0x00000065** (because 101 (decimal) is 65 hex).
Entering **0x101** gives **0x00000101** because the entry is specified as hex.
Entering **0b101** gives **0x00000005**. **0b** is the prefix for 'binary'. (101 in binary is 5 in decimal and hex.)

The pop-up tooltip shows you the value of the word in decimal and binary notations.

```
 000          0x0        0x4        0x8
0x0000        101        257          5
0x0001          0          0          0
```

The tooltip now shows the value in Hex, and Binary
The row- and column-headers do not change: they are always represented in Hex.

Addressing

The memory is laid out in four columns, for visual convenience only. Each word of memory has a unique 'address' - a *five*-digit hex number. The first four digits of the address are shown by the row-header, and the full address is specified by appending the single hex digit shown in the column header. Thus, the address of the *top-left* word on this screen is **0x00000**, and the bottom-right is **0x001fc**.

Exercise 2

What is the address of the word shown highlighted here:

If the address has five hex digits, and each digit is 4 bits, what is the largest possible address, as a hex number, and in decimal?

Model answer 2

The highlighted word has address `0x00038`.
The largest possible address is `0xfffff`, which is 2^{20}-1, or 1048575. This is commonly referred to throughout the computer industry as '1 megabyte', but, since 1998, the official IEC standard has been 'one **mebi**byte', seeking to avoid ambiguity (the SI meaning of 'mega' being 10^9 rather than 2^{20}).

Why do the address columns go up in jumps of four (`0x0`, `0x4`,`0x8`,`0xc`)? Each word of 32 bits is made up from four 8-bit 'bytes'. ARMlite, in common with most modern processors uses 'byte addressing' for memory. When storing or retrieving a word (which we'll learn how to do in Chapter 3) you specify only the address of the first of the four bytes making up that word.

What's in a word?

The size of a 'word' varies between machines. Modern computers usually have 32- or 64-bit words; older ones were 16-, 8-, or 4-bit. And before the emergence of the single-chip microprocessor, computers had custom-designed word sizes: 18,20,36,40,60…

However, since the advent of semiconductor memory chips, *memory* has been measured, and managed in 'bytes', where a byte is always 8-bits irrespective of the word-size of the machine that the memory will be used in.

ARMlite, like all modern computers is a 'stored program computer': memory is used both for storing the program instructions and data to be manipulated. To run an assembly language program, it is necessary both to translate the assembly language instructions into machine code, and to load those codes into memory. On old machines, these were two distinct steps; on ARMlite they are both performed in one.

Teacher note: the von Neumann vs. Harvard architectures
The AQA specification contrasts the so-called 'von Neumann architecture' with the so-called 'Harvard architecture'. The author of this book is not alone in believing that these two terms are poorly-defined, potentially very misleading, and historically inaccurate. For this reason, the terms are avoided in the pupil version of this book. However, since they do form part of the AQA specification, and must be therefore taught for the A-level, teachers might like to make the connection at this point.

Exercise 3

Access the simulator via https://peterhigginson.co.uk/ARMlite/

Click on the **Edit** button (below the **Program**) and then copy and paste the complete assembly language program listed at the beginning of the chapter, into this area. (You should be able to copy both columns on one go, but if this is not possible, ensure that the code from the second column is placed after the code from the first column.)

Then hit the **Submit** button. This should not give any errors (if it does you have pasted the code incorrectly). Before proceeding, use the **Save** button to save the program to a file on your local machine.

Submit did two things: first it 'assembled' (translated) the assembly language into machine code; then it loaded the machine code into memory.

You will also see that ARMlite has now added 'line numbers' to your program. These do not form part of the assembly language (which is also known as the 'source code'), but are there to help you navigate and discuss your code.

What is the highest line number?

If you hover the mouse over one of the lines of the source code (this is only after it has been submitted), you will see a pop-up tooltip that indicates the address in memory of the corresponding machine code instruction.

What hex address is given for line number 75 of code? Paste a screenshot highlighting the contents of this word in the memory area.

Model answer 3

The highest line number is 77.
The hex address given for line number **75** is **0x00124**.

000	0x0	0x4	0x8	0xc
0x0000	0xe3241000	0xe3a02b01	0xe3a06000	0xe3a09000
0x0001	0xacffffff	0xe3a03000	0xe782a003	0xe2833004
0x0002	0xe3530a03	0xbaffffb	0xe3a03f41	0xe51f0114
0x0003	0xe2000001	0xe3500000	0x1a000000	0xe7829003
0x0004	0xeb000030	0xe3530c2f	0xbaffff7	0xe3a03000
0x0005	0xe7920003	0xe7810003	0xe2833004	0xe3530a03
0x0006	0xbaffffa	0xe2866001	0xe3a03f41	0xe3a05000
0x0007	0xe2434c01	0xeb00001e	0xe24340fc	0xeb00001c
0x0008	0xe2834004	0xeb00001a	0xe2834f41	0xeb000018
0x0009	0xe2834c01	0xeb000016	0xe28340fc	0xeb000014
0x000a	0xe2434004	0xeb000012	0xe2434f41	0xeb000010
0x000b	0xe3550004	0xba000001	0xe782a003	0xea000007
0x000c	0xe3550003	0xba000001	0xe7829003	0xea000003
0x000d	0xe3550002	0xba000000	0xea000000	0xe782a003
0x000e	0xeb000008	0xe3a00c2f	0xe1530000	0xbaffffde
0x000f	0xeaffffd5	0xe7910004	0xe150000a	0x0a000000
0x0010	0xe2855001	0xe1a0f00e	0xe2833004	0xe20300ff
0x0011	0xe3500000	0x0affffb	0xe35000fc	0x0affff9
0x0012	0xe1a0f00e	<mark>0xe1000070</mark>	0x00000000	0x00000000
0x0013	0x00000000	0x00000000	0x00000000	0x00000000

(The reason why there is no machine code corresponding to lines 76 and 77 of the source, is that those lines are not strictly processor instructions - they are instructions *to the assembler*, known as 'assembler directives'. We'll see more of them later, but this is *not* a very important point to understand.)

Exercise 4

Hit **Edit** and try inserting:

- A couple of blank lines

- Additional spaces before an instruction, or just after a comma (but not between other characters)

- A comment on a line of its own, starting with `//` such as `//My first program`

- A comment *after an instruction* but on the same line

Submit the code again.

What has happened to:

- The blank lines
- Additional spaces
- The comments
- The line numbers
- The total number of instructions that end up as words in memory? (Why?)

Edit again and remove the comma from the first line of code. What happens when you **Submit** now?

Restore the program to its original condition, either by going back to **Edit**, or just **Load**ing it again, and **Submit.**

Model answer 4

Blank lines are removed
Spacing and alignment is standardised
Comments on their own line are given a line number
The line after the new lines (excepting blank lines) will have been renumbered
The total number of machine code instructions in memory has not changed, because comments are not translated into machine code.
Removing a necessary comma will result in a pink highlight and a syntax error message on the console (Input/Output area).

The program you have loaded is a simulation of a colony of simple organisms, being born, reproducing and, eventually dying. (Individual cells never move, but the patterns of cells being born and dying give the impression of movement, and many interesting dynamic patterns emerge). The code is a variant of a very famous program called Life (see panel).

Conway's Game of Life

The 'Game of Life', also known simple as 'Life' (it's not really a game, it's a simulation) was devised by British mathematician John Horton Conway in 1970. In the intervening 50 years there have been implementations of it written for almost every computer manufactured, real and virtual, now including ARMlite.

It simulates the birth, reproduction and death of single-celled, static, organisms living in a community. Each location on the grid has up to 8 immediate neighbours. (Strictly, Life should be played on an infinite board – the ARMlite screen has hard edges, and these do affect the behaviour.) If an organism has more than three live neighbours, it will die of 'overcrowding'. If it has less than one live neighbour, it will die of 'loneliness'. If an *empty* location has three live neighbours, a new organism will be born there.

Running on a fast machine, a simulation of Life produces patterns of extraordinary complexity. Starting with a random distribution of live organisms, the simulation will change dynamically for many 'generations' before settling to steady state consisting of static groups of live cells and some 'oscillators' - a group that cycles through a repeated pattern.

You may also observe 'gliders' – small groups of cells that appear to move (diagonally) across the screen – actually the cells don't move, but the pattern of births and deaths repeats itself moving one square diagonally each cycle. You can even observe (or specify as a starting pattern) one or more 'glider guns' that will regularly emit gliders, and other even more complex constructs. A glider gun is shown on the left.

Life shows a simple example of 'cellular automata', which is a branch of research into artificial life forms ('A-life'). The originator of this branch of mathematics was none other than John von Neumann, who also made significant contributions to many other branches of mathematics, computer science, weather forecasting, atomic weapons design, and economics! Von Neumann ('Johnny' to his friends) postulated the idea of an automaton that could both do useful work (as a computer) *and* reproduce itself from raw materials. It would be 50 years before anyone managed to implement an example of this; one shown on the right.

Picture credits and further reading:
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life and
https://en.wikipedia.org/wiki/Von_Neumann_universal_constructor

Exercise 5

Run the program, using the **Run** button:

You'll see a spinning gearwheel appear near the run controls to indicate that the processor is active.

You will also observe a lot of activity in the 'graphics screen' (the lowest of the three panes under Input/Output). After a short while (a few seconds to a couple of minutes) the colony will stabilise.

At any point you can hit the **Stop** button and then **Run** again. Since the starting pattern of cells is randomised, the behaviour will be different each time you run.

Then hit the **Pause** button. As well as freezing the graphics screen and stopping the spinning gear wheel. You will also see some orange highlights appear. What do you think they signify?

You can continue execution by pressing play again. Do this *and then pause again.*

What does clicking on this control do?

And this one?

What happens if you click on this button more than once in succession?

Finally, while paused, click line number **21** of the source code, which will paint a red background behind the line number. This is called 'setting a break point' and will cause processing to be paused when the breakpoint is breached.

Having set the breakpoint, continue running until the pause is observed (almost immediately!). Has the processor paused just *before*, or just *after* executing the line with the breakpoint?

From the breakpoint you will find that you can single-step, or continue running slowly or at full speed.

While paused you can remove a breakpoint by clicking on the line again.

Model answer 5

When you hit pause, an orange highlighter appears on one of the lines of the source code, and on the corresponding word in memory. These (both) indicate the last instruction completed.
(A system message also appears on the 'console')

          executes a single step of the program.
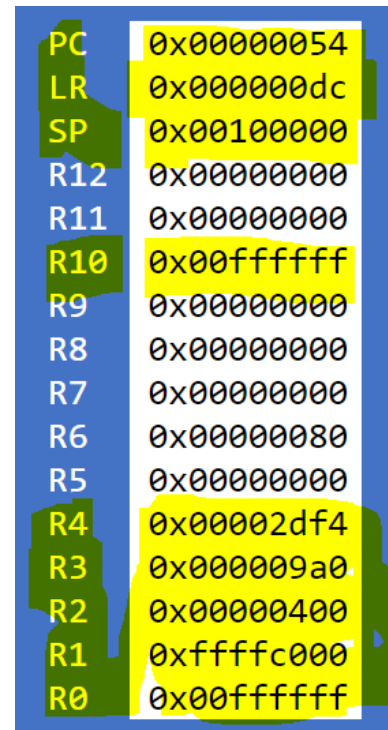
          executes continuously but slowly. Pressing it multiple times speeds it up slightly each time.

ARMlite will pause having completed the instruction *before* the breakpoint. Typically, this is the line above the breakpoint, but (it may be discovered) where there is a 'branch' involved (we'll cover this in Chapter 3) the previous instruction might be somewhere else in the code.

Registers

Another thing that you might have noticed when paused, single stepping, or running slowly, is frequent changes to the values in the 'registers' – highlighted in the screenshot below. Each register is like a single 32-bit word of memory, but with these differences:

- Registers are much faster to access than main memory (which is still very fast, though).
- The values in registers may be manipulated directly by instructions. To manipulate a value held in memory, it must first be loaded into a register, then manipulated, then (if the new value needs to be preserved) stored back to memory – to the same location or a different one.
- Registers have a name, rather than an address.
- ARMlite has thirteen 'general purpose' registers, named R0 to R12. These are typically used to hold the data items most frequently needed by the program. For a small program it is possible that *all* the data items needed can be held in these registers.
- There are also three 'special purpose' registers, named PC, LR and SP. These are typically accessed and manipulated by the processor. They may be accessed and even manipulated by program instructions directly, though there is a risk of interfering with the normal execution of the program if you are not careful.

PC	0x00000054
LR	0x000000dc
SP	0x00100000
R12	0x00000000
R11	0x00000000
R10	0x00ffffff
R9	0x00000000
R8	0x00000000
R7	0x00000000
R6	0x00000080
R5	0x00000000
R4	0x00002df4
R3	0x000009a0
R2	0x00000400
R1	0xffffc000
R0	0x00ffffff
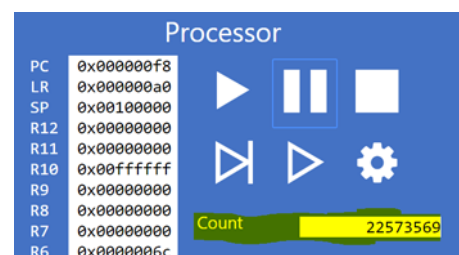
In Chapter 2, we will be learning how to use the general-purpose registers.

> **Teacher note**
>
> The AQA specification for assembly language refers only to registers R0 through R12 - it does not mention PC, LR, or SP. However, students are expected to be familiar with the operation of the PC in general terms.

Machine code is *fast*

When you ran the program in slow mode you doubtless observed that the program 'loops' many times over the same instructions. You may also have noticed that in slow mode, even if you speed it up as much as possible, *many* instructions are executed between each update to the graphics screen. You can see the total number of instructions executed since the program started in the **Count** field, shown highlighted on the right.

Processor	
PC	0x000000f8
LR	0x000000a0
SP	0x00100000
R12	0x00000000
R11	0x00000000
R10	0x00ffffff
R9	0x00000000
R8	0x00000000
R7	0x00000000
R6	0x0000006c
Count	22573569

> **Optional exercise**
>
> Using a stopwatch, run the Life program for exactly 10 seconds before pausing, and then take a note of the Count value. Divide this by 10 for an approximate measure of ARMlite's speed, in instructions per second, when *running on your browser and computer.*

Depending on the physical computer you are using, ARMlite can execute *several million instructions per second*. And this is actually very *slow* compared to real processing speeds, because ARMlite is a

simulation. Under the covers, ARMlite is a JavaScript program that interprets each machine code instruction from the program you are 'running' into JavaScript function calls. Your browser, in turn must translate the JavaScript into the machine code for the processor on your computer (which might be an ARM, or might be an Intel processor, with a different instruction set).

If you were to run the same machine code *directly* on an ARM processor, and the graphics screen was just an array of LEDs, say, then the program would run at *billions* of Instructions per second – and the pattern of organisms would stabilise almost immediately.

Why learn assembly language programming?

In the early days of computing, assembly language offered a considerable advance on writing machine code in hex, or binary (or, commonly, 'octal'). But why should you learn it *today*, when there are a multitude of high-level programming languages? (Apart from the fact that you need to learn some assembly language to pass your exams!)

If you were to pursue a career in computing, it is possible that you might end up having to write some assembly language, or a low-level language quite like assembly language, at some point. But it is also possible that you could get through an entire career in computing without ever seeing assembly language again! So that's not really a strong enough argument for learning it now.

The strongest argument for learning assembly language is that it will give you a better understanding of what is going on at the processor level when your high-level language programs are executing. For example, if you continue as far as Chapter 5 you should understand why, in a high-level language, any element of an array may be accessed in O(1) time instead of O(n). To use an analogy, it is not necessary to know how an internal combustion engine works to drive a car, but *most* racing drivers have a pretty good understanding of the mechanics of their car, in order to gain the best performance.

The final reason, however, is that learning assembly language can be very enjoyable. One thing that might have struck you already about the Life program, is that this *tiny* program (71 instructions and each one performing only a very simple operation), produces quite complex, and interesting behaviour. Many examples of assembly language that you see in textbooks cover only trivial, and, let's face it *boring*, examples, such as sorting three numbers into order. But it is perfectly feasible to write interesting programs in assembly language, and if you follow this book through to the end, you'll be writing a series of games, starting simple, but ending up satisfyingly complex. You'll also be able to read and understand exactly how the Life program is working.

> **Teacher note**
> ARMlite has been programmed with an 'Easter Egg'. To view this, **Clear** the memory (if not already clear) then hit **Edit**, type just the word **demo** into the program area and **Submit**.
>
> This will load and run an in-built, slightly more sophisticated implementation of Life, wherein the screen wraps around itself, joining top and bottom edges, left and right. (This might sound like it forms a sphere, but it is actually a 'toroid'). This tends to result in colonies staying dynamic for longer, but it takes more code to implement.
>
> Whether you share the information about the Easter Egg with your pupils, or keep this as a special trick known only to the teacher, is up to you!

Chapter 2: Countdown

In subsequent chapters of this book we will be writing assembly language code to implement a series of games. In this first chapter, however, we will instead be learning how to *play* a game - one that involves writing assembly language. The game is a variant of the 'Countdown Numbers Game', which you might have encountered previously, either on television, or in your Maths classroom (if not, see the panel).

Your task will be broadly similar: given a set of starting numbers, and a defined set of operations, you must write a short program in assembly language that results in the target number (or as close as you can get). Don't worry: we are going to get plenty of practice with how to write assembly language, and the specific operations, before you have to play the game. And you *won't* be asked to try to come up with a solution in 30 seconds!

We'll start by using just addition and subtraction. Our initial numbers are 100,25,8,7,3,1 and our target is 84. This is an easy challenge from a mathematical perspective: 1+8+100-25. Here's one way to code that expression in assembly language:

```
MOV R0,#1
ADD R1,R0,#8
ADD R2,R1,#100
SUB R3,R2,#25
HALT
```

This program consists of a sequence of five instructions, one per line. Each instruction consists of an 'operation', shown here in 'mnemonic' form (usually an abbreviation of the description of the instruction). You have probably guessed that ADD and SUB are the operations to add and subtract values, and that HALT brings the execution of the program to a halt; MOV is the operation to move a value (which really means 'copy and move').

Each operation, except HALT is followed by up to three 'operands', specifying what the operation is applied to in each case. Where there is more than one operand, they must be separated by commas.

The last operand in the first four of the instructions above, consists of one of our initial numbers: 1,8,100 and 25, in each case preceded by the # symbol (pronounced 'hash' - not 'hashtag', incidentally). In assembly language programming, these are known as 'immediate' values - meaning that they are written directly in the program code.

> **Teacher note**
> In high-level programming languages, the equivalent to an immediate value, is typically called a 'literal' value.

The other operands – **R0**,**R1**,**R2** and **R3** – specify registers, which are used to hold initial values, intermediate calculations, and the result. Registers are the fastest form of memory, and their contents can be manipulated *directly*. In this chapter, all the calculations can be handled using only the thirteen 'general purpose' registers (**R0** to **R12**). Later you will learn how to handle much larger amounts of data, held in main memory. However, since, on ARMlite, most operations cannot be applied directly to values held in main memory, you will find that much of assembly language programming consists of 'loading' values from memory into registers, manipulating them within registers, and, if appropriate, storing new or modified values back into memory. You will also find that input/output is handled much the same way.

> **The Countdown numbers game**
>
> Countdown is a long-running British television game show, involving word and number tasks. (See https://en.wikipedia.org/wiki/Countdown_(game_show) for more background).
>
> In the original version of the *numbers* part of the game, players are given a randomly selected set of six 'initial' numbers (in the range 1 to 100), and then a target number (in the range 1-999). Working against the clock, they must use the initial numbers and the four basic mathematical operations (add, subtract, multiply, divide) in order to produce the target number - *or get as close as possible*. (Since the target is chosen at random, it is not necessarily always possible to match it exactly). You may use brackets, or the calculation may be evaluated as a series of steps. Fractional numbers are not allowed - and nor are calculators!
>
> For example, given the initial numbers: **25,50,75,100,3,6** and the target number: **952** it is straightforward to get to **953** (off by just one) as follows:
>
> ```
> 6 + 3 = 9
> x 100 = 900
> + 50 = 950
> + 75/25 = 953
> ```
>
> All the original numbers were used in this case. (Players are not required to use *all* the initial numbers, but each may be used once only). Surprisingly, it *is* possible to get the target exactly, as follows:
>
> ```
> 100 + 6 = 106
> x 3 = 318
> x 75 = 23,850
> - 50 = 23,800
> /25 = 952
> ```
>
> Even more surprisingly, the second solution was devised by a contestant on the television show *within the time limit of 30 seconds*! You can watch him do it, and the astonished reaction of the presenters here: https://www.youtube.com/watch?v=pfa3MHLLSWI

Exercise 6

Set ARMlite to display data in ⌈Decimal (unsigned) ▾⌉ format. This will make things easier, *initially*, because our game will be working primarily with decimal numbers.

Select **Edit** so that you can click within the program area, then copy and paste the five-line program below into that **Program** area:

```
MOV R0,#1
ADD R1,R0,#8
ADD R2,R1,#100
SUB R3,R2,#25
HALT
```

Submit, then **Run** the program (with the 'play' button). When the program halts (almost immediately) paste a screen snippet showing just the value of **R3**.

Note: When the program has halted, if you want to run it again you must click the **Stop** button before **Play**. What happens if, when halted, you press play *without first pressing stop*.

Model answer 6

R3		84	

If, when halted, you click on **Play** *without* first clicking **Stop** you will get a 'bad instruction' error (shown on the console) because ARMlite will attempt to execute the next instruction in memory, which will be empty (zero) - not a valid instruction.

Teacher note
The reason that **HALT** does not do the same as hitting **Stop** is twofold:
- **HALT** leaves the values in the registers available for inspection, whereas hitting **Stop** resets them.
- You can put one or more **HALT** instructions in the middle of a program and allow the user to then continue execution by clicking on **Play** again.

We'll now look at the instructions in detail:

Instruction	Interpretation
`MOV R0,#1`	Move (copy) the immediate value **1** into register **R0**
`ADD R1,R0,#8`	Add **8** to the value currently in **R0** and put the result in **R1** Note that **R1**, here, is called the 'destination register' for this instruction.
`ADD R2,R1,#100`	Add **100** to the value currently in **R1** and put the result in **R2**
`SUB R3,R2,#25`	Subtract **25** from the value currently in **R2** and put the result in **R3**
`HALT`	(Temporarily) halt the execution of the program.

Exercise 7

If necessary, **Stop** the program and this time use the **Single step** button to execute it one instruction at a time. Notice that with each step the value in *one* register has changed.

Notice also that the (orange) highlighter moves across the assembly-language instructions and, simultaneously, across the corresponding 'machine code' instructions held in memory.

Looking carefully at the changing register values, and at the code highlighter, does the orange highlighter indicate the instruction *about to be executed*, or the one *that has just been executed*?

Model answer 7

The highlighter shows the instruction that has just been executed.

In our example, we used a different register to record each intermediate step of the calculation. However, this is not essential: we could do the whole of this simple calculation using a single register, but changing its contents with each step, as shown below:

```
MOV R0,#1
ADD R0,R0,#8
ADD R0,R0,#100
SUB R0,R0,#25
HALT
```

In the example code so far, the final operand for the `MOV`, `ADD` and `SUB` operations has always been an immediate value (prefixed by `#`). However, this operand may alternatively be specified as another register, as shown in the examples below:

Instruction	Interpretation
`ADD R3,R2,R1`	Add the values from `R2` and `R1`, and place the result in `R3`
`ADD R4,#1,#2`	This is invalid syntax - only the *last* operand may be an immediate value

Multiply and Divide?

The Countdown numbers game isn't going to be very challenging, or much fun, if we are restricted to addition and subtraction operations. ARMlite does not currently have a way perform multiplication or division using a single instruction, even on whole numbers. You would need to write your own routines for that (there is an example routine for integer multiplication in Chapter 6).

However, there are several other single-instruction operations that we can apply to the Countdown game, and these will make the game more specific to the field of Computer Science, and more challenging as well - because they will involve thinking simultaneously in decimal and binary (or hex).

Hardware multipliers

Many of the *earliest* digital electronic computers, such as the ENIAC (pictured), *did* have hardware multipliers, and some had hardware dividers. In part this was because most early computers were applied principally to complex mathematical calculations.

With the transition to single chip 'microprocessors', multiplication and division were moved from hardware to software, in the form of re-usable subroutines constructed from add, subtract and other bit-level operations.

However, the size and power of microprocessors has since grown dramatically, so most modern processors do have dedicated hardware circuits to perform multiplication and/or division, on integers or floating-point numbers. In these cases, there will usually be a single assembly-language instruction to specify the multiply/divide operation. The circuitry may be integrated into the processor chip, or provided on a separate 'maths co-processor'

Picture credit and further reading: https://en.wikipedia.org/wiki/ENIAC

Bit-wise instructions

The table below lists five new instructions that manipulate values in registers.

Instruction	Example	Description
AND	AND R2,R1,#4	Performs a bit-wise logical AND on the two input values, storing the result in the equivalent bit of the destination register.
ORR	ORR R1,R3,R5	As above but using a logical OR
EOR	EOR R1,R1,#15	As above but using a logical 'Exclusive OR'
LSL	LSL R1,R1,#3	'Logical Shift Left'. Shifts each bit of the input value to the left, by the number of places specified in the third operand, losing the left-most bits, and adding zeros on the right.
LSR	LSR R1,R1,R2	'Logical Shift Right'. Shifts each bit of the input value to the *right*, by the number of places specified in the third operand, losing the right-most bits, and adding zeros on the left.

Teacher note

ARMlite will accept **XOR** as a synonym for **EOR**, and **OR** as a synonym for **ORR** – because **XOR** and **OR** are commonly used in assembly language notations. However, **EOR** and **ORR** are the form used in the AQA instruction set and hence are adopted in this book.

These are all described as 'bit-wise' operations, because they manipulate individual bits in the operands. They are best understood by viewing the values in binary format (or in hex if you are experienced in mentally transforming hex to binary).

Exercise 8

Write your own simple program, that starts with a `MOV` (as in the previous example) followed by five instructions, using each of the five new instructions listed above, once only, but in any order you like – plus a `HALT` at the end, and with whatever immediate values you like.

Note: Keep all your immediate values less than `100` (decimal). Also, when using `LSL`, don't shift more than, say `#8` places. Using very large numbers, or shifting too many places to the left, runs the risk that you will start seeing negative results, which will be confusing at this stage. (We'll be covering negative numbers in the final part of this chapter.)

You may use a different destination register for each instruction, or you may choose to use only `R0`, for both source and destination registers in each case - both options will work.

Paste in your complete program, and then step through the program, completing the table below. You can do this either by using the tooltip, or by switching the display format between **Decimal (unsigned)** and **Binary**. *It is not necessary to include the leading zeros, or the* `0b` *prefix, in your answers, although you may.*

Describe in words, what is the effect *on a decimal number* of doing a logical shift left (`LSL`) by one bit ? And by two bits? Similarly, for a logical shift right (`LSR`)?

Your complete instruction	Decimal value of the destination register after executing this instruction	Binary value of the destination register after executing this instruction
`MOV R0,`		
`HALT`	_	_

Model answer 8

This is an *example* of what is expected:

Your complete instruction	Decimal value of the destination register after executing this instruction	Binary value of the destination register after executing this instruction
MOV R0,#63	63	111111
AND R1,R0,#51	51	110011
LSL R2,R1,#3	408	110011000
ORR R3,R2,#11	411	110011011
LSR R4,R3,#2	102	1100110
EOR R5,R4,#25	127	1111111
HALT	_	_

Logically shifting left a decimal value by one bit has the effect of doubling it, and by two bits of multiplying it by four. Logically shifting right one by divides the number by two, *rounded down if the original number was odd.* Shifting right by two places divides it by four, similarly.

Play the game

You are ready to start playing the game. Remember these rules:

- You do not have to use *all* the initial numbers, but each may be used once only as an immediate value.
- You may use as many registers as you wish, but you may use each register *only once as the destination register*, and then *only once as a source register* (i.e. as an argument).
- You may use only the operations introduced so far in this chapter.
- The result must be visible in a register, and must be the correct answer *as a decimal number.* For example, the binary answer 0b....101 (5 in decimal) would *not* count as decimal 101!

Hint: You may find it helpful to write the decimal result going into the destination register for each instruction in a comment

Exercise 9

Your six initial numbers are: **12,11,7,5,3,2** and your target number is: **79**

Paste a screenshot showing your program, and with the result in a register.

Model answer 9

Some possible answers (there may be many others)

```
MOV R0, #2
LSL R1, R0, #5 //64
ADD R2, R1, #12 //76
ADD R3, R2, #3 //79
```

```
MOV R0, #12
ADD R1, R0, #7 //19
LSL R2, R1, #2 //76
ADD R3, R2, #3 //79
```

Exercise 10

Your six initial numbers are: **99,77,33,31,14,12** and your target number is: **32**

Paste a screenshot showing your program, and with the result in a register.

Model answer 10

Possible answer (there may be many others)

```
MOV R0,#99
SUB R1,R0,#77 //22
ADD R2,R1,#12 //34
AND R3,R2,#33 //32
```

Exercise 11

Your six initial numbers are: **30,13,7,5,2,1** and your target number is: **390**

Paste a screenshot showing your program, and with the result in a register.

Model answer 11

Possible answer (there may be many others)

```
MOV R0,#13
LSL R1,R0,#5 //416
SUB R2,R1,#30 //386
MOV R3, #1
LSL R4, R3, #2 //4
ADD R5, R2, R4 //390
```

Negative numbers

Exercise 12

Set ARMlite to display data in Decimal (signed) ▾ format.

Run the following simple program and capture the result shown in **R1**.

```
MOV R0, #9999
LSL R1, R0, #18
HALT
```

Model answer 12

```
R1    -1673789440
R0           9999
```

Why is the result shown as a negative decimal number, and with no *obvious* relationship (in decimal) to **9999**?

If you use the tooltip, you will see that the binary representations of **R0** and **R1** are:

R0 - 0b0000000000000000000010011100001111

R1 - 0b1001110000111110000000000000000000

As indicated by the added highlights above, the processor has shifted the binary value in **R0** left by **18** bits to produce the value in **R1**, *as expected*.

In setting the display format to **Decimal (signed)** we are asking ARMlite to display all word values as a *signed* decimal number (i.e. positive or negative), by interpreting their *binary* representation as 32-bit two's complement.

Teacher note

Two very important points need to be made, or reinforced, regarding two's complement format, probably in plenary session:

1. When doing anything with two's complement, you <u>must</u> specify the number of bits of the two's complement representation you are working in. For example, in an 8-bit two's complement representation:

```
+5 = 0b00000101
-5 = 0b11111011
```

However, in 16-bit two's complement format:

```
+5 = 0b0000000000000101
-5 = 0b1111111111111011
```

Failing to understand/remember this is a common source of mistakes in binary arithmetic.

In ARMlite, all use of two's complement representation is 32-bit.

You might find this online tool helpful: https://www.exploringbinary.com/twos-complement-converter/ as it requires you to specify the number of bits when converting.

2. Even if you know the numbers of bits, you *cannot* look at a binary number and say whether it *is* in two's complement format or not. You can only look at a binary number and say what its value *would be if interpreted as (n-bit) two's complement*.

In the same way, you cannot look at the 8-bit value `0b01100011` and say whether it *is:*

- The letter C (in ASCII)
- The number 99 (or 0x63
- The colour Turquoise
- A person's age
- An E$^{\#}$ in a music program

You can only say what those 8 bits mean *if interpreted* using a specific representation, and/or context.

Computers do not *understand* colours, sounds, letters, decimal numbers, hex, *or even binary*. They manipulate patterns of bits.

Any word where the left-most or 'most significant' bit is **1**, will be a negative number, *if interpreted as 32-bit two's complement format*. But this bit is not simply representing a plus or minus bit (as would be the case for a 'sign and magnitude' representation). In 32-bit two's complement format the left-most bit is given the *negative* value: -2^{31} or -2147483648 . All the other bits to the right of it are given positive values: $+2^{30}$, $+2^{29}$ … $+2^{1}$.

Exercise 13

Switch ARMlite to display in **Binary** format.

You can't edit register values directly, but you can edit memory words. Click on the top-left memory word (address `0x00000`) and type in the following values, which will be interpreted as decimal and translated into the 32-bit two's complement format, which you can then copy back into your answers.

What is the binary representation of each of these *signed* decimal numbers:

```
 1
-1
 2
-2
```

Try to spot the pattern, if you can, before reading on.

Model answer 13

```
0b00000000000000000000000000000001
0b11111111111111111111111111111111
0b00000000000000000000000000000010
0b11111111111111111111111111111110
```

The pattern is as follows. To get the negative version of a number:

- invert (or 'flip') each of the bits
- then add 1 to the end.

We can simulate this, by introducing a new operation `MVN`, which stands for 'Move NOT'. It works like `MOV`, but each of the bits in the second operand (whether that's an immediate value or the value in a specified register) has a logical NOT operation applied to it before going into the destination register.

In following code, then, the second and third instructions implement the transformation specified above:

```
MOV R0, #27
MVN R1,R0
ADD R2,R1,#1
HALT
```

Switch back to [Decimal (signed) ▾] display format.

Verify for yourself that `R2` ends up containing `-27`, if presented as a signed decimal.

You *could* reverse this transformation by subtracting one and then inverting all the bits. Surprisingly, however, you could instead simply apply the *original* transformation *again*:

```
MOV R0, #27
MVN R1,R0
ADD R2,R1,#1
MVN R3,R2
ADD R4,R3,#1
HALT
```

> Verify for yourself that by applying the same transformation twice, you get back (in **R4**) to the original value of **27**.

At first sight, this two-step transformation might seem rather arbitrary. But the extraordinary thing about it is that it works. In means that the processor can add and subtract numbers without having to care whether the values are positive or negative to begin with.

Exercise 14

Run this program:

```
MOV R0, #27
MOV R1, #-5
ADD R2, R0, R1
HALT
```

And verify for yourself that adding **-5** to **27** produces the same result as subtracting 5 from 27.

What happens if you add **-49** to **27**?

Model answer 14

Will correctly produce the answer **-22** (if presented as signed decimal).

Teacher note

It would be possible to use the new knowledge of handling negative numbers, together with the `MVN` operation to play an extended version of the Countdown game. However, pupils might be ready for something new by now, and might find Chapter 3 more engaging.

Chapter 3: Matchsticks

In this chapter we will be writing a very simple game called Matchsticks. The game starts with a pile of 15 matchsticks (although it could be any number). Players take it in turns to remove either 1,2, or 3 matchsticks from the remaining pile. A player wins the game by forcing the opponent to take the last matchstick. Our implementation will pit a single human player against the computer.

To implement the game, we will need to learn how to implement iteration (looping), and selection (branching), in assembly language. We will also need to learn some patterns for writing text to the console, and reading inputs from the user during the game. In order to do the latter, we first need to understand the loading and saving of values from/to memory locations using the **LDR** – '**LoaD R**egister (from memory)', and **STR** – '**ST**ore **R**egister (to memory)', instructions.

Working with memory addresses

Exercise 15

Run ARMlite in the default [Hex ▼] display format, and ensure that the **Program** and **Memory** are clear. (The **Clear** button, bottom right of the simulator, will do that, if necessary.)

Enter and **Submit** the following code:

```
MOV R0, #255
STR R0, 68
HALT
LDR R1, 72
HALT
```

You will see that the program code has been translated into machine code and loaded into the first five words of main memory.

Run the program and notice that when it reaches the first **HALT**, the value of one of the other words in main memory will have changed. Paste a screenshot of the memory only highlighting that changed memory location.

Explain why the value is shown is *what* it is, and *where* it is.

Now click on the memory location immediately to the right of the one that was changed, and type in another decimal value). *Continue* running (by hitting the **Play** icon) and show, with a partial screenshot, that the value you entered has been copied ('loaded') into **R1**.

Why is the second operand for the **LDR** instruction **72**, and not **69**? What happens if you change it to **69** and attempt to run again?

Model answer 15

The value, `0x000000ff`, is **255** decimal. The memory location is `0x00044`, which is **68** decimal.

The memory address must be a multiple of 4 because each word of memory is 4 bytes (and ARMlite, like most modern processors uses 'byte addressing').

If you change the address to **69** then, upon hitting **Submit**, you get an `Unaligned access …` error message, indicating that the address is not aligned to a word boundary (i.e. is not divisible by 4).

It is very important to understand that with the ARMlite instruction set:

- The `MOV` instruction cannot work with memory addresses – the second operand must be a register or an immediate value. (Similarly, for `MVN`).
- The `LDR` instruction cannot be used to load an immediate value into a register – the second operand must specify a memory address.

By contrast, in some real processors, `MOV` (or its equivalent) can deal with registers, immediate values, or memory addresses. One advantage of the enforced separation of roles, as on ARMlite, is that it helps to remind the programmer that operations involving memory access are slower than those that work only with registers and/or immediate values.

Teacher note `LDR` vs. `MOV`

ARMlite's enforced distinction in role deliberately mimics the AQA specification.

In past years, mark schemes have explicitly deducted marks for writing e.g. `LDR R0, #32`
when `MOV R0, #32` is required - even though the pupil's intention might be clear.

RISC vs. CISC

As the early computers evolved, processors typically acquired larger instruction sets, to make it easier for programmers to express algorithms and hence to improve their productivity.

However, by the 1980s it was recognised that since almost all programming was now being done in high level languages, and automatically translated into machine code, the case for making machine assembly language easy for human programmers to read or write was much weaker. By reducing the set of instructions, processors could be made more performant. The resulting change in processor design became known as the shift from CISC (Complex Instruction Set Computers) to RISC (Reduced Instruction Set Computers). Most modern processors are now considered to be RISC, although there is no precise definition of the distinction.

Back in the days of CISC, many instructions could deal directly with memory locations. With RISC the more common pattern is for most instructions to deal only with data in a small set of registers, with just a few specialised instructions for loading and storing values in main memory.

Labels

When writing a program in assembly language it can be hard enough keeping track of what the values in the general-purpose registers currently represent - let alone with memory addresses, potentially many thousands of them. But this is what you had to do with the earliest assemblers - all they did was translate the 'mnemonic' form of an instruction, such as `MOV R0, #32`, into the corresponding, binary, machine code: `0b11100011101000000000000000100000`. The next step forward was the introduction of the 'symbolic assembler', which allowed the programmer to define 'symbols' (today, more commonly called 'labels') to stand for specific memory addresses. Today, all modern assemblers have this capability.

Teacher note

1) Using labels for data values in assembly language can be thought of as analogous to named variables in a high-level programming language - and, indeed, the former evolved into the latter. However, we should be cautious about over-using analogies with high-level languages to explain low-level languages work, because the low-level languages came first.

2) At the time of writing, AQA has not used labels for data addresses in an exam question, only for identifying the position of instructions (used, for example, when branching). However, the principle is the same and there is nothing in the specification to suggest that labels may not be used for data addresses, as they commonly are in most forms of assembly language.

The following short program defines two labels for memory addresses, **xCoordinate** and **yCoordinate**, and initialises those memory addresses with the values **3** and **4** respectively. These label *definitions* are located *after* all the program instructions - this is the recommended practice. A label definition must have a colon immediately after it – as shown highlighted, below. The program instructions use, or 'reference', these labels, but a label *reference* does not have a colon.

```
        LDR R0, xCoordinate
        ADD R0, R0, #6
        STR R0, xCoordinate
        LDR R0, yCoordinate
        ADD R0, R0, #2
        STR R0, yCoordinate
        HALT
xCoordinate: 3
yCoordinate: 4
```

The programmer does not know, or in many cases even care, where exactly the values of ,
xCoordinate and **yCoordinate**, are located - because they can always be referenced by the label.

Exercise 16

With ARMlite in default (hex) mode, enter and **Submit** the code above.

Before running it, hover the mouse over the label *definitions* (in the last two lines) of the code. The
pop-up tooltip will show you the memory address (in hex) that the label refers to in memory. What
are the addresses for **xCoordinate** and **yCoordinate**?

Paste two partial screenshots of the Memory area of ARMlite, one taken before the program is run,
and one after, in both cases highlighting the two memory words for **xCoordinate** and **yCoordinate**.

Model answer 16

xCoordinate is located in `0x0001c`

yCoordinate is located in `0x00020`

Before running the program:

After:

This example also reveals why we need the **HALT** instruction. If you were to remove the **HALT** then
ARMlite would attempt to execute the next word (which holds the data value for **xCoordinate**) as
an instruction. For the values used in our example, this will fail - giving a 'bad instruction' error. But
on a real ARM processor, the data values might well correspond to real instructions and this would
result in some unwanted, or unpredictable, behaviour.

Self-modifying programs

When the idea of the 'stored program' computer (as we now call it) was proposed, towards the end of WWII, *one* of the motivations was that it would be possible for programs to deliberately create, or modify, data values (to memory locations) that could then be executed as program instructions – in other words what we now call 'self-modifying code'. After the war, Alan Turing foresaw this as a possible way to achieve what we would today call 'machine learning' or 'artificial intelligence'. However, most of the early uses of self-modifying code were more mundane – including the ability to modify, dynamically, the memory address used by a specific instruction. The latter requirement was later made redundant by the introduction of 'Indirect' addressing, which we will cover in Chapter 5.

It is also worth noting that an assembler, or even a simple 'loader' program, that can read instructions from external storage into memory, both require ability to write program instructions into memory. Later, the same would apply to compilers.

Other early computer pioneers, such as Howard Aiken, who designed the machine that became known as the Harvard Mark I, were strongly opposed to the idea of programs creating or modifying code. Today, most modern processors deliberately prevent self-modifying code because of the risks of accidentally, or, in the case of 'malware', deliberately corrupting the system.

Simple input/output

Teacher note - Input/Output on ARMlite and AQA

Input/Output is not the responsibility of the processor, so many processors do not have specific instructions for this. In these cases the approach to input/output is specific to the computer that is built around the processor. The 'memory-mapped I/O' approach used in ARMlite is a common pattern. The AQA assembly language specification does not define specific mechanisms for input output, though the general principle is clearly recognised. For example, one past exam question indicated that writing a value to memory location 17 (for the purposes of *that* question) would control the speed of a connected motor.

Part of the ARMlite screen is labelled Input/Output. The topmost field within this area is the 'console' - which may be used for sending text to the user; the second field is to allow the user to input data when the program requests it.

ARMlite makes use of **STR** and **LDR**, together with labels, to manage interaction with these fields on the screen. This whole concept is known as 'memory-mapped I/O'.

We'll introduce these ideas by making a start on writing the Matchsticks game. We'll be taking an 'iterative' approach to development: writing just a little bit more functionality each iteration. Here's Iteration 1:

```
//R0 - remaining matchsticks
//R1 - used for writing messages
//R2 - number to remove
    MOV R0, #15
    STR R0, .WriteUnsignedNum
    MOV R1, #msg1
    STR R1, .WriteString
    MOV R1, #msg2
    STR R1, .WriteString
    LDR R2, .InputNum
    SUB R0, R0, R2
    HALT
msg1: .ASCIZ "remaining\n"
msg2: .ASCIZ "How many do you want to remove (1-3)?\n"
```

Note the following:

- The program starts with comments (rendered in green, above) which define, where possible, the usages of the registers within the code. This is a recommended practice.

- `msg1`, and `msg2` (short for 'message') are user defined labels for memory locations, as we used before, but instead of defining one or more words, each defines an ASCII string. `.ASCIZ` is another 'assembler directive' meaning 'ASCII, terminated by a Zero'. The zero byte is added onto the string, so that ARMlite knows where the string ends. Each character will be stored as a single byte, so four to a word.

- The instruction `MOV R1, #msg1` does <u>not</u> load the contents of `msg1` into `R1`. Loading data from memory would require an `LDR` instruction, but it would not be possible in this case because the contents of `msg1` would not fit into a single register. Instead, `MOV R1, #msg1` moves the *immediate value* of the *label* `msg1` into `R1`, in other words the address in memory where the contents of `msg1` starts.

- `.WriteSignedNum` is a like label, but the dot in front of it indicates that it is a label *recognised by the ARMlite assembler* - rather than a *user-defined* label such as `msg1`. The assembler translates this label into real memory addresses to be used at run time, though the actual memory locations used for input/output are deliberately outside the range that you can view in the Memory area of the simulation. At run-time, when a value is written to the memory location corresponding to `.WriteSignedNum`, ARMlite knows that this needs to be written to the console, translated into a signed decimal representation

- `.WriteString` is another ARMlite system label, that writes a whole string instead of a single character. `R0` cannot hold the string, because no more than four ASCII characters could fit in a register, so instead, `R0` holds the address in memory where ARMlite may find start of the string may be found (the end being defined by the zero byte).

- Each use of `.WriteString` is therefore preceded by an instruction specifying the starting address of the required string in a register, for example: `MOV R0, #msg2`. This may be articulated as 'Move into `R0`, an *immediate* value – being the *address* that `msg2` will be translated into by the assembler.'

- `LDR R2, .InputNum` is another example of ARMlite's memory-mapped I/O. When executed this instruction will request the user to enter a number into the input field, and this will then be loaded into `R2`, *as if* it were being loaded directly from a memory address.

- `\n` is called an 'escape character'. When output to the console, this will result in a new line. (This same syntax is recognised in many high-level languages when used within strings).

Exercise 17

Run the program above and run it. When requested for input, enter **1**, **2** or **3**. When the program halts, capture a partial screenshot showing the console, and showing the value in **R0** which should be the number of matches remaining (shown in hex).

Model answer 17

```
 1 //R0 - remaining pencils
 2 //R1 - used for writing messages
 3 //R2 - number to remove
 4       MOV R0, #15
 5       STR R0, .WriteUnsignedNum
 6       MOV R1, #msg1
 7       STR R1, .WriteString
 8       MOV R1, #msg2
 9       STR R1, .WriteString
10       LDR R2, .InputNum
11       SUB R0, R0, R2
12       HALT
13 msg1: .ASCIZ "remaining\n"
14 msg2: .ASCIZ "How many do you want to
```

PC	0x00000024
LR	0x00000000
SP	0x00100000
R12	0x00000000
R11	0x00000000
R10	0x00000000
R9	0x00000000
R8	0x00000000
R7	0x00000000
R6	0x00000000
R5	0x00000000
R4	0x00000000
R3	0x00000000
R2	0x00000002
R1	0x0000002f
R0	0x0000000d

Count

Current
Instruction

Status bits   NZCV
               0000

Input/Output

```
15 remaining
How many do you want to remove (1-3)?

Program HALTED. STOP, LOAD or EDIT
```

Branching

For the moment we will just imagine that there is only one player (not a very interesting game!). We want the program to loop around, displaying the number of matchsticks remaining. In ARMlite assembly language programming, the simplest way to implement a loop is with the **B** instruction which stands for 'Branch' followed by details of where we want to branch back (or forward) to. The clearest way to specify the branch destination is with a user-defined label, for example, **loop:** as shown below:

```
//R0 - remaining matchsticks
//R1 - used for writing messages
//R2 - number to remove
    MOV R0, #15
loop: STR R0, .WriteUnsignedNum
    MOV R1, #msg1
    STR R1, .WriteString
    MOV R1, #msg2
    STR R1, .WriteString
    LDR R2, .InputNum
    SUB R0, R0, R2
    B loop
    HALT
msg1: .ASCIZ "remaining\n"
msg2: .ASCIZ "How many do you want to remove (1-3)?\n"
```

> **Teacher note - B and GOTO**
>
> **B** (for Branch) or an equivalent (such as JUMP or JMP) exists in most processor instruction sets. When the first high-level programming languages – FORTRAN and then ALGOL – were developed, this same idea became the GOTO instruction.
>
> Later, with the introduction of 'structured programming' languages (helped along by Edsger Dijkstra's famous letter in 1968 to the *Communications of the ACM*, 'GOTO Statements Considered Harmful'), the GOTO statement gradually disappeared from high-level languages, in favour of improved high-level programming structures such as WHILE loops. However, these high level structures are still typically translated into direct branches in the machine code.

Note also that specifying the location to branch to as a label means that we don't have to worry about changing the address as we insert or delete instructions.

Exercise 18

Make the changes shown above and run the program to check for yourself what it now does.

Why has the `loop:` definition been placed on the *second* instruction and not on the first? (If you are not sure, try changing it and running the program again).

Even as a single-player version of the game, can you identify two serious shortcomings of the functionality?

Model answer 18

If `loop:` was defined on the first instruction, the number of remaining matchsticks would be reset to 15 each time.

Shortcomings:
- The game never ends. The number of matchsticks eventually goes negative.
- There is nothing to stop you taking more than 3 matchsticks, or even taking 0 or a negative number!

Several of the current shortcomings require some sort of 'selection' functionality – also known, in the context of assembly language programming, as 'conditional branching'. These operate like the B instruction, but the branch is made only when certain conditions are met. There are four versions of the conditional branch available to us at this stage:

BEQ – '**B**ranch if **EQ**ual'

BGT – '**B**ranch if **G**reater **T**han'

BLT – '**B**ranch if **L**ess **T**han'

BNE – '**B**ranch if **N**ot **E**qual'

Teacher note
ARMlite has more conditional branching options, all documented in the ARMlite Programming Manual (available via the **Documentation** link on ARMlite's screen) The four listed above match those in the AQA instruction set.

'Branch if *what* is equal?' you might be saying. These conditional branch instructions are designed to follow a CMP instruction that compares two values, for example:

`CMP R0,R1` compares the values in two registers
`CMP R3,#16` compares the value in a register to an immediate value

CMP works somewhat like SUB – it subtracts the second operand from the first – but it does not assign the result to a destination register, the result is immediately thrown away. The only memory it keeps of the *result* is held in the status flags, which are displayed on ARMlite (highlighted, right).

The N bit indicates that the result of the compare was **N**egative, and Z that it was **Z**ero.

(The **C** and **V** bits stand for **C**arry and o**V**erflow. Broadly speaking, they are used to signal when the result of an operation is not correct, because the correct result would not fit in 32-bits. We will not need them for now.)

Teacher note - Carry and oVerflow

Unfortunately, the *exact* operation of the **C** and **V** bits on an ARM processor, on which ARMlite is based, is very complex to explain, and is best left alone. There are more details in the ARMlite Programming Manual for those who really want to get to grips with them. AQA appears to require pupils to understand the general principle of carry/overflow status bits, but does not have a clear specification of how it expects these bits to work.

The highlighted change below introduces a new label, `input:` and a compare, followed immediately by a conditional branch back to input. The effect is that if the player enters a value of greater than 3 it will be ignored, and the player will simply be asked to enter a number again:

```
//R0 - remaining matchsticks
//R1 - used for writing messages
//R2 - number to remove
    MOV R0, #15
loop: STR R0, .WriteUnsignedNum
    MOV R1, #msg1
    STR R1, .WriteString
    MOV R1, #msg2
    STR R1, .WriteString
input: LDR R2, .InputNum
    CMP R2, #3
    BGT input
    SUB R0, R0, R2
    B loop
    HALT
msg1: .ASCIZ "remaining\n"
msg2: .ASCIZ "How many do you want to remove (1-3)?\n"
```

Teacher note - Input validation

The code just added is a form of 'input validation'. This is an important concept for all forms of programming and may be related to other parts of the course.

Exercise 19

Make the changes shown above and test the program.

Now, with reference to the four possible conditional branch instructions listed above, add further instructions to enforce the rule that the number cannot be less than 1.

Test your program.

Try entering a negative number, does the code prevent this?

Finally, play the game until there are just 1 or 2 or fewer matches remaining. What happens if the player then attempts to remove more matches than remain? Can you figure out a way to prevent this?

Paste a screenshot showing the final version of the code, highlighting the new instructions that you added.

Model answer 19

The second test should prevent a choice of zero, or a negative number.
However, in the final stages, the user can attempt to remove more matches than remain. This can be prevented with the third test shown below.

```
 1|//R0 - remaining pencils
 2|//R1 - used for writing messages
 3|//R2 - number to remove
 4|       MOV R0, #15
 5|loop: STR R0, .WriteUnsignedNum
 6|       MOV R1, #msg1
 7|       STR R1, .WriteString
 8|       MOV R1, #msg2
 9|       STR R1, .WriteString
10|input: LDR R2, .InputNum
11|       CMP R2, #3
12|       BGT input
13|       CMP R2, #1
14|       BLT input
15|       CMP R2, R0
16|       BGT input
17|       SUB R0, R0, R2
18|       B loop
19|       HALT
20|msg1: .ASCIZ "remaining\n"
21|msg2: .ASCIZ "How many do you want to
```

We now need to introduce the automated (computer) player. To begin with, we'll get the computer to take 1, 2, or 3 matchsticks, selected at random, but not more than the remaining number. We could write our own pseudo-random number generator, but ARMlite offers a ready-made way to load a random number from a random number generator. In the following snippet of code:

`select: LDR R2,  .Random`	instructs ARMlite to load a random 32-bit pattern into R2
`AND R2, R2, #3`	removes all except the least significant 2 bits (i.e. reduces range to 0-3)
`CMP R2, #0`	if the choice is zero …
`BEQ select`	… choose again
`CMP R2, R0`	if the choice is greater than remaining matchsticks …
`BGT select`	… choose again
`BEQ select`	or if the choice would mean removing *all* the matchsticks. choose again

Note that at the end of the code we have a **CMP** instruction followed by two, different, conditional branch instructions. This works because these conditional-branch instructions always refer to the result of the most recent comparison - the latter does not have to be the instruction immediately before the branch.

We're now ready to have a go at the whole program:

```
//R0 - remaining matchsticks
//R1 - used for writing messages
//R2 - number to remove
MOV R0, #15
loop:
STR R0, .WriteUnsignedNum //Print remaining matchsticks
MOV R1, #msg1
STR R1, .WriteString
//Computer's turn
select: LDR R2, .Random
AND R2, R2, #3
CMP R2, #0
BEQ select
CMP R2, R0
BGT select
BEQ select
cont: STR R2, .WriteSignedNum
MOV R1, #msg4
STR R1, .WriteString
SUB R0, R0, R2
//Print remaining matchsticks
STR R0, .WriteUnsignedNum
MOV R1, #msg1
STR R1, .WriteString
//Check for computer win
CMP R0, #1
BEQ computerWins
//Player's turn
MOV R1, #msg2
STR R1, .WriteString
input: LDR R2, .InputNum
CMP R2, #3
BGT input
CMP R2, #1
BLT input
CMP R2, R0
BGT input
SUB R0, R0, R2
CMP R0, #1
BEQ playerWins
b loop
playerWins: MOV R1,#msg3
STR R1, .WriteString
HALT
computerWins: MOV R1,#msg5
STR R1, .WriteString
HALT
msg1: .ASCIZ "remaining\n"
msg2: .ASCIZ "How many do you want to remove (1-3)?\n"
msg3: .ASCIZ "You win!\n"
msg4: .ASCIZ "taken by computer. "
msg5: .ASCIZ "Computer wins! \n"
```
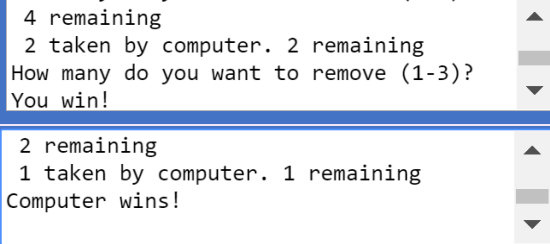
> **Exercise 20**
>
> Enter and run the complete program, more than once.
>
> Capture a partial screenshot showing the console at the end of the game where you have won, and one where the computer has won.

There is actually a very simple strategy which is guaranteed to win if you make the first move, and has a very high prospect of winning even if you are the second player *provided that your opponent is not playing the same strategy* (as at present, where the computer is selecting 1-3 matchsticks at random).

Can you work out the winning strategy?

Model answer 20

```
 4 remaining
 2 taken by computer. 2 remaining
How many do you want to remove (1-3)?
You win!
```

```
 2 remaining
 1 taken by computer. 1 remaining
Computer wins!
```

The winning strategy (which, be warned, is fairly easy to find on the web) is to get the remaining number of matchsticks, as quickly as possible to a multiple of four *plus 1* i.e. to 13, 9, or 5. This means that however many matches your opponent *then* takes, you take 4 minus the number they took - guaranteeing to leave one remaining match for your opponent at the end.

Optional exercises to improve/extend the game

If you have time available, try modifying and/or extending the program to achieve the following:

- When the game is completed, loop back to the beginning to play again automatically
- Either take it in turns to go first, or select who goes first at random, each round
- Change the starting number of matchsticks from 15 to a random number
- Keep scores of the number of times the computer, and the player, has won
- Figure out, and implement a smarter algorithm for the computer to play the game. (Note that by following the optimum algorithm it is always possible for the first player to guarantee a win).

Chapter 4: Hangman

In this chapter we will use the instructions and techniques learned in the previous two chapters, to write a slightly more complex game: Hangman. This will require the implementation of simple graphics, and the ability to manipulate data that represent ASCII characters, rather than just numbers.

> **Teacher note**
> (The text assumes that pupils are familiar with the paper-and-pencil game known as Hangman. If not, then a brief practice game on paper might be warranted).
>
> This chapter adheres strictly to the AQA instruction set, plus Input/Output techniques (which are necessarily specific to ARMlite as AQA does not specify any Input/Output). The approach to pixel graphics used in this chapter - setting each pixel with a separate instruction - is recognised as being rather clunky, but this is forced on us by the constraint of AQA not including indirect/indexed addressing.
>
> For pupils that have the time and motivation to progress past this chapter, however, Hangman serves as an excellent stepping-stone, because it helps to identify the *need* for more flexible addressing modes, introduced in Chapter 4.

Low-res pixel graphics

ARMlite supports pixel graphics in three forms: lo-res, mid-res, ,and hi-res. In all three cases the graphics appear in their own pane within the Input/Output area. In this chapter we will be using the lo-res graphics, which offers a resolution of 32 x 24 pixels. Like other forms of I/O in ARMlite, pixel graphics are memory mapped. Here's an example:

```
MOV R0, #.red
STR R0, .Pixel0
MOV R0, #0xffa503
STR R0, .Pixel32
HALT
```

Notes:

- ARMlite recognises many common colour names, such as `.red` above, *written in lower-case*. The assembler simply translates these into a number representing the RGB colour format, as used in HTML. Because it is translated into a number value, the colour must be preceded by `#` to specify that it is to be used as an immediate value.
- You may also specify any RGB colour value directly as a number. The most convenient way to do this is using six digits of hex, such as `#0xffa503` above, because the three pairs of hex digits specify the red, green, and blue colour components respectively.

Exercise 21

Enter and run the code above - you will see two pixels drawn on the graphics screen.

Given that the lo-res version is a grid of 32x24 pixels, work out the pixel numbers for the four corners of the screen, and modify the code so that it draws just one pixel of a different colour (any) in each corner.

Paste partial screenshots showing your code, and the resulting graphics screen.

Model answer 21

Example:

```
1|        MOV R0, #.red
2|        STR R0, .Pixel0
3|        MOV R0, #.green
4|        STR R0, .Pixel31
5|        MOV R0, #.orange
6|        STR R0, .Pixel736
7|        MOV R0, #.black
8|        STR R0, .Pixel767
```

We'll use this capability to draw the finished hangman picture, something like this:
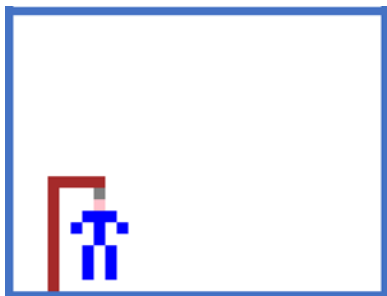
In the game, we will need to draw the picture *progressively*, according to how many wrong letters the player has guessed. The code below shows a programming pattern for this, showing just the first two parts of the drawing, the 'Upright' and 'Cross beam':

```
//Register uses:
//R0 multiple, temporary, purposes
//R8 number of wrong guesses
    MOV R8, #10
drawPic:
    CMP R8, #1
    BLT endDraw
    MOV R0, #.brown
    STR R0, .Pixel739 //Upright
    STR R0, .Pixel707
    STR R0, .Pixel675
    STR R0, .Pixel643
    STR R0, .Pixel611
    STR R0, .Pixel579
    STR R0, .Pixel547
    STR R0, .Pixel515
    STR R0, .Pixel483
    STR R0, .Pixel451
    CMP R8, #2
    BLT endDraw
    STR R0, .Pixel452 //Cross beam
    STR R0, .Pixel452
    STR R0, .Pixel453
    STR R0, .Pixel454
    STR R0, .Pixel455
    CMP R8, #3
    BLT endDraw
//TODO:
//Rope, Head, Body, Left leg, Right leg, Left arm, Right arm
endDraw:
    HALT
```

Notes:

- As before, we will place comments at the start recording the uses of the registers. R0 will be re-used for different purposes during the game (this is common).
- R8 will hold the number of wrong letters guessed so far. In this example code it has been set straight to 10 at the top of the program, but later this will be determined by game logic.
- As highlighted above, before each element of the drawing is started, the value of R8 is compared, first to 1, then to 2, and so on. If at any point R8 is less than the specified value, execution is branched to endDraw, which at this stage is just a HALT.

Exercise 22

Run the program above and confirm for yourself that it draws the first two components of the drawing. Then change the value put into **R8** at the start from **#10** down to **#1**, and confirm that this time just the first component is drawn.

Now, *following the same pattern,* create your own version of the complete hangman drawing. It does not have to look exactly like the one shown above, and it may be made larger if you wish. However:

- It should have exactly **nine** components in total.

- Then for the tenth incorrect guess (which results in the player losing) make a small change to the drawing, to indicate that the person has been hung. (For example, replacing the face with a black pixel, signifying the black hood placed over it).

- Do not spend a large amount of time on this, you can always improve the artistic quality of the drawing later.

When done, test the program, by running it with different initial values for **R8**, to check that, in each case the program draws the correct number of components.

When tested, paste a screenshot showing the *complete* drawing, and then paste in your code separately (it might not be possible to show all the code on the ARMlite screen, so it is better to copy the code as text).

Then make sure you **SAVE YOUR CODE** (e.g. as 'drawPic routine'), because you'll need to insert your routine into later code that we develop.

Model answer 22

[A screenshot showing a suitable finished hangman drawing]

Here's the code for the version of the drawing shown above.

```
//Register uses:
//R0 multiple, temporary, purposes
//R8 number of wrong guesses
   MOV R8, #10
drawPic:
   CMP R8, #1
   BLT endDraw
   MOV R0, #.brown
   STR R0, .Pixel739 //Upright
   STR R0, .Pixel707
   STR R0, .Pixel675
   STR R0, .Pixel643
   STR R0, .Pixel611
   STR R0, .Pixel579
   STR R0, .Pixel547
   STR R0, .Pixel515
   STR R0, .Pixel483
   STR R0, .Pixel451
   CMP R8, #2
   BLT endDraw
   STR R0, .Pixel452 //Cross beam
   STR R0, .Pixel452
   STR R0, .Pixel453
   STR R0, .Pixel454
   STR R0, .Pixel455
```

```
    CMP R8, #3
    BLT endDraw
    MOV R0, #.grey
    STR R0, .Pixel487 //Rope
    CMP R8, #4
    BLT endDraw
    MOV R0, #.pink
    STR R0, .Pixel519 //Head
    CMP R8, #5
    BLT endDraw
    MOV R0, #.blue
    STR R0, .Pixel551 //Body
    STR R0, .Pixel583
    STR R0, .Pixel615
    CMP R8, #6
    BLT endDraw
    STR R0, .Pixel646 //Left leg
    STR R0, .Pixel678
    STR R0, .Pixel710
    CMP R8, #7
    BLT endDraw
    STR R0, .Pixel648 //Right leg
    STR R0, .Pixel680
    STR R0, .Pixel712
    CMP R8, #8
    BLT endDraw
    STR R0, .Pixel550 //Left arm
    STR R0, .Pixel581
    CMP R8, #9
    BLT endDraw
    STR R0, .Pixel552 //Right arm
    STR R0, .Pixel585
endDraw:
    HALT
```

Now we can start to turn this into a game. The game is for two players - one sets the word, and one guesses it, and they should swap roles between runs.

Because we are going to have to hold and test the word in a register, this limits us to four characters, and since a three-character word is less interesting (and harder to guess, surprisingly) *the game will require that words have exactly four characters.*

Teacher note

The limit of four characters is undesirable, but to get a round that we would need to split the word over multiple registers (resulting in quite complex code) or introduce indirect/indexed addressing, which is outside the scope of AQA - though it is introduced in the second half of this book.

While we have studiously avoided using the phrase 'four-letter word', some pupils are bound to make the connection. It is left to the teacher's discretion as to whether a reminder about appropriate language in the classroom is warranted!

The following code asks the setter to enter a word and stores it in a memory address labelled `secretWord`.

```
//Register uses:
//R0 multiple, temporary, purposes

captureWord:
    MOV R0, #setter
    STR R0, .WriteString
    MOV R0, #secretWord
    STR R0, .ReadSecret
    HALT

setter: .ASCIZ "Setter: Enter word\n"
secretWord:
```

Notes:

- **.ReadString** would read in a string from the input field. **.Read**Secret does the same thing, but obscures the input on the display - often used for password entry, for example.

Exercise 23

Run the code shown above and test that you can enter the word. The requirement for four characters is not enforced here, but you must play by the rules.

You should be able to see the secret word, encoded as four ASCII values, has been copied into the memory location (at the end of the program) corresponding to the `secretWord` label. (The way that ARMlite presents this on screen gives the impression that the order has been reversed, but don't worry about this).

Paste a partial screenshot highlighting the memory location where your word has been stored, indicating separately how this corresponds to the ASCII values of the four characters of your word.

Having the word shown in the Memory location would allow a player who could remember ASCII values to guess every letter! So … in the code, *just above* the `secretWord:` label definition, insert a new line:

 `.ALIGN 512`

This will move the location of the secret word to the next available byte address that is divisible by `512`, which is just beyond the default view of memory.

Make the change, run the program, and confirm that you can no longer see the encoded word in the first page of memory.

Then change the Memory page number to `002` (as shown below):
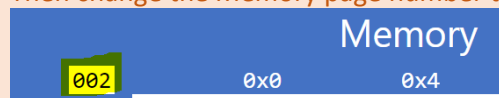
and paste a partial screenshot that shows the encoded version of your entered word appears in location `0x00020`.

Then make sure you change the Memory page number back to `000` for the rest of the game!

Model answer 23

Here, `0x64657266` represents the secret word `fred`. The hex values for these characters are `66,72,65,64` so they are shown in reverse.

Teacher note – 'Little-endianism'

The reason the apparent reversal of characters is that the ARMlite simulator, in common with most *modern* processors (but unlike much older processors) uses 'little endian' format. It means that when a 32-bit value is stored into a word address in memory (i.e. using `STR`), the *least significant* eight bits are stored in the first byte, and the *most significant* as the fourth byte. Similarly, when a register is loaded with a whole word from memory (using `LDR`), the first byte is taken to be the least significant 8-bits.

To make things more understandable to the user, ARMlite switches the order back when *presenting* a number to the user. That way the 32-bit value representing the year **2019**, say, will be shown as `0x000007e3` as expected, whereas if presented strictly in byte order it would be shown as `0xe3070000`, which would be confusing. However, when presenting a word that represents a string of four ASCII characters (ARMlite does not *know* what any item of data represents), the presentation appears to have reversed the characters.

Fortunately, for many coding purposes you don't even need to know about the distinction.

Origin of the terms 'Little-endian' and 'Big-endian'

Although 'little endian' and 'big endian' are reasonably descriptive of the choice of representing the least-significant or the most-significant digits first, the two terms were originally borrowed from literature: 'Gulliver's Travels' by Jonathan Swift, published in 1726. In the book's fictional kingdom of Lilliput, the king requires that his subjects break their boiled eggs at the little (more pointed) end of the egg, but a radical political faction tries to establish the opposite practice. The book is a satire on many aspects of eighteenth-century British society, religion, and politics - in this case the tendency to split society over matters of seemingly trivial dogma. The appropriation of the terms to Computer Science is therefore doubly relevant, where academics and practitioners alike are known for taking polarised decisions about many aspects of the science. You'll encounter many more examples yet.

Further reading: https://en.wikipedia.org/wiki/Endianness Picture credit: https://en.wikipedia.org/wiki/File:Soft-boiled-egg.jpg

We're ready to go for the whole program. The following code is complete, *except for your own* `drawPic` routine (*the code continues over the page*):

```
//Register uses:
//R0 multiple, temporary, purposes
//R1 secret word, loaded from memory (loaded & cleared each guess so it does not
show on the UI)
//R2 current letter guess
//R3 successful guesses in right place(s), otherwise zeros
//R4 holds the built up result (of underscores and correct letters)
//R5 to R7 temporary use in processing
//R8 number of wrong guesses
//R9 number of matches made

captureWord:
MOV R0, #setter
STR R0, .WriteString
MOV R0, #secretWord
STR R0, .ReadSecret

guessLetter:
MOV R0, #player
STR R0, .WriteString
waitForKey: LDR R2, .LastKeyAndReset
CMP R2, #0
BEQ waitForKey
ORR R2,R2,#32 //Forces the character to be lower case
MOV R6, #0 //Set to 1 if a NEW match is found
MOV R7, #0 //This will increment 0 > 8 > 16> 24 as a shift amount to select
successive characters

checkForMatch:
MOV R5, #0xff //The mask for character 1
LSL R5, R5, R7 //Shift mask to character of interest (first shift will be zero!)
AND R0,R3,R5 //Apply mask to past guesses
CMP R0, #0
BEQ cont //If char position is NOT empty this char has been correctly guessed
LSR R4,R0,R7
B writeChar
cont:
LDR R1, secretWord
AND R0, R1, R5
MOV R1, #0 //Reset so it does not reveal word on UI when paused
LSR R4, R0, R7 //Get the char back to position 1
ORR R4,R4, #32 //Force the character to lower case
CMP R4, R2
BNE notAMatch
ORR R3,R3,R0
ADD R9,R9,#1
MOV R6, #1 //Set R6 to flag that the character has been matched
B writeChar
notAMatch:
MOV R4, #95 //Underscore
writeChar:
STR R4, .WriteChar
ADD R7,R7, #8 //Shift to next character
CMP R7, #32
BLT checkForMatch
MOV R0, #10 //Newline
```

```
STR R0, .WriteChar

//If there has not been a match, increment the number of misses
CMP R6, #0
BNE checkForWin
ADD R8, R8, #1
STR R8, .WriteUnsignedNum
MOV R0, #wrong
STR R0, .WriteString
B drawPic

checkForWin:
CMP R9, #4
BNE drawPic
MOV R0, #win
STR R0, .WriteString
HALT

//Insert your drawPic code here - not including the endDraw, which has been
redefined below

endDraw: CMP R8, #10 //This many guesses means you're hung!
BLT guessLetter
MOV R0, #.black
STR R0, .Pixel519 //Head again
MOV R0, #lose
STR R0, .WriteString
MOV R0, #secretWord
STR R0, .WriteString
HALT

setter: .ASCIZ "Setter: Enter word\n"
player: .ASCIZ "Player: Try a letter\n"
win: .ASCIZ "You win!\n"
lose: .ASCIZ "You lose! Word was:"
wrong: .ASCIZ "wrong. "

.ALIGN 512
secretWord:
```

Exercise 24

Copy the code above into a new file using a text editor and *save it* (e.g. as `Hangman.txt`). Then open your `drawPic` routine, copy and paste the code into the **Hangman** file, to replace the comment line highlighted above. Note that instead of halting at `endDraw`, the code now continues, so make sure you have the new code and that you aren't duplicating the `endDraw` label definition. Save the file again.

Load the complete file and ensure that it can be submitted and run successfully.

Now edit the URL used to access ARMlite by adding the highlighted text (known technically as a 'query string') on the end:

https://peterhigginson.co.uk/ARMlite/?profile=player

Adding the 'player profile' to the URL configures ARMlite to display only the controls that are relevant to someone playing the game, rather than a programmer. Also, the **Input/Output** area has been made twice the size.

Switching the configuration like this will have cleared the program, but you will find that you can still **Load** and then **Run** the program.

Run the program, and with a partner play the Hangman game. Remember that the secret words must be exactly four characters in length.

Paste a partial screenshot (from ARMlite in Player mode) showing the full screen.

Then run the program again, this time using **Single Step** (when it requests a user input you will need to enter the value, hit the **Return** key and then continue pressing the **Single Step** icon). Try to get as much feel as you can for how the code is working. You will find it helpful to refer to the register definitions at the top of the code.

Model answer 24
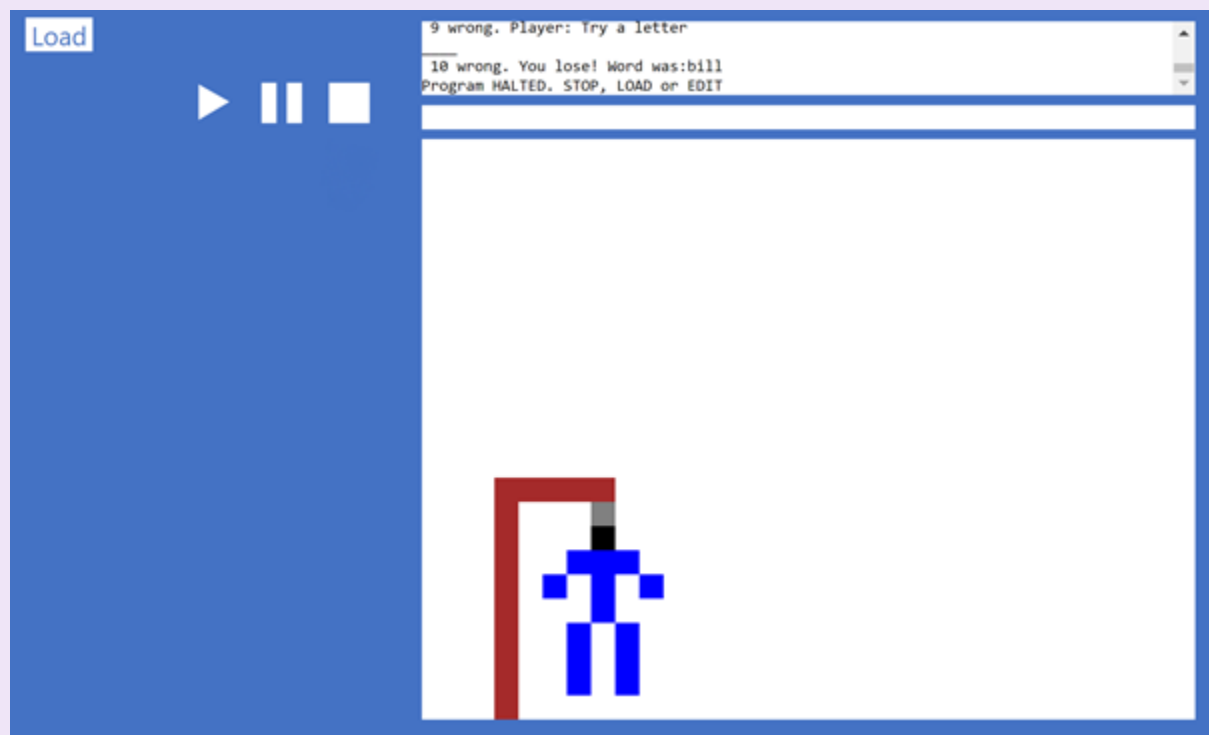
Exercise 25

Think about ways in which the program might be improved, from a user's perspective, and *describe* your suggested improvements on your answer sheet (no need to implement them here).

Model answer 25

Here are a few suggested improvements, but pupils may think of many more:

- Add input validation to enforce that the secret word is exactly 4 characters
- Loop back from the end to start a new game
- Get the two players to enter names (stored as new ASCII strings in the data). The programmer should then use the names to prompt the players to alternate between setting and guessing.
- Keep the players' scores over multiple rounds.

If you have time, you might like to attempt some of your simpler suggested changes.

However, the most obvious improvement - letting the game work with longer secret words, and of variable length - is going to be very difficult using only the instructions that you have learned so far. It would be *possible* to split the word over two or more registers, but the program will end up with a lot of duplicated code.

Also, you might have found it frustrating, when drawing a line of pixels, that you had to specify each pixel individually. And if you wanted to move your whole drawing slightly to the right, say, you would have to change every single instruction that writes a pixel!

'Surely,' you might be saying, 'good programming is about generalisation?' - and you would be right. What we really need is a mechanism to allow us to 'iterate' over a range of memory locations, which might represent the characters in a string, a line of pixels on the screen, or an array of numeric values to be searched or sorted.

If you are thinking like this then you are re-enacting one of the crucial chapters in the early development of modern digital computers in the mid-1940s, which led to more flexible forms of specifying memory addresses that we today refer to as 'indirect' and/or 'indexed' addressing. If you are able to progress to the second part of the book you will learn how to learn these, and other powerful assembly language programming constructs, to write programs that are far more powerful – and interesting – than those that we have been able to write thus far.

Book II – Delving deeper

> **Teacher note**
>
> Book II goes beyond the minimal requirements of the AQA A-level specification and introduces several new constructs that are key to writing more interesting, and more advanced, programs in assembly language.
>
> The individual concepts are introduced using interesting example programs, wherever possible.
>
> Chapter 8 concludes the book with the guided development of a complete Snake game, making use of all the techniques introduced within the book.
>
> Working through Book II will deepen a student's understanding of assembly language, as well as their experience and understanding of programming in general. Book II will give students enough technical understand to consider undertaking their NEA A-level project in Assembly Language, running on ARMlite.

Chapter 5: Indirect & Indexed addressing

So far, when we have used `LDR` and `STR` to access memory locations we have used 'direct addressing' - we specify the address of the memory location *directly*, either as a number (in decimal or hex), or as a label that is translated into a number by the assembler, before running:

`LDR R0, 100` Load into `R0` the contents of memory location (decimal) 100
`STR R1, 0x000f4` Store the contents of `R1` into hex location `0x000f4`
`LDR R2, myExamGrade` Load into `R2` the contents of the location labelled `myExamGrade` in the code

---

Teacher note

Many books, and the AQA specification, also refer to 'immediate addressing', for example: `MOV R0, #34`. This is potentially rather confusing to the newcomer, because such instructions do not involve memory *addresses* at all, and, anyway, 'immediate' operands cannot be used with `LDR` or `STR`.

This terminology was clearer in earlier 'CISC' microprocessors such as the 6502, which did not have multiple general purpose registers, but where many instructions could take as the argument an immediate value, or a direct address, or an indirect address, or an indexed address, for example (6502):

`LDA #$09` Load Accumulator with immediate value 9 (`$` indicates hex) – 'immediate addressing'
`LDA $09` Load from the memory location `$09` – 'direct addressing'

The 6502 had several additional address modes, including 'Zero-page Indexed', 'Indexed Indirect', and 'Indirect Indexed'. The indirect and indexed modes for ARMlite that we will introduce below are comparatively straightforward.

The limitation of direct addressing should have become apparent to you in the Hangman exercise (Chapter 4). With direct addressing there is no way to *generalise* many routines: even to draw a straight line on the pixel screen, you must write a separate instruction for each pixel.

Now carefully read the code below, including comments. Note that we are now switching from the 'lo-res' pixel graphics used in the previous chapter, to 'mid-res': a grid of 64 x 48 pixels, each of which is still represented by 1 word (4 bytes) holding a 24-bit HTML colour value:

```
    MOV R1, #.PixelScreen //Start of the mid-res pixel screen memory
    MOV R2, #.red
    MOV R3, #0 //This is our pixel counter
loop: ADD R4, R1, R3 //R4 now holds the address of the pixel of interest
    STR R2, [R4] //Paint the pixel at the address specified in R4.
    ADD R3,R3,#4 //Increment the pixel number (by 4 bytes = 1 word = 1 pixel)
    CMP R3, #256 //256 will be one pixel past the end of the line (64 pixels x 4
bytes per pixel)
    BLT loop
    HALT
```

Teacher note – hi-res graphics

If a student wishes to use the highest resolution graphics mode, refer them to the ARMlite Programming Reference Manual (see Appendix II).

Notes:

The key new idea comes in this instruction: `STR R2, [R4]`. The use of the square brackets indicates that the second operand is using 'indirect addressing' meaning that we are specifying the address *indirectly* (using, in this case, a register value).

`STR R2, [R4]` does <u>not</u> mean: 'store the contents of `R2` into `R4`' (which would anyway not be possible with a `STR` instruction). Instead it means: 'store the contents of `R2` into the *memory address* that is held in `R4`'.

Another name for an indirect address is 'pointer'. Pointers are used extensively in computers, for example in the low-level implementation of all the main types of data structure.

Exercise 26

Run the code above, which should print a red line right across the pixel screen. Paste a partial screenshot showing the result.

Model answer 26

In the beginning …

The earliest computers did not have indirect or indexed addressing. However, the benefit of being able to vary the address in an instruction, especially within a loop, was recognised very early on. John Von Neuman (pictured) proposed a solution whereby instructions could modify other instructions in memory, an idea that became known as 'self-modifying code', and this idea was widely used on several early machines.

Von Neumann was aware of the risks posed by self-modifying code. For this reason, he proposed the safety mechanism that instructions could only overwrite the *operand* part(s) of another instruction, not the opcode (i.e. the type of instruction). On machines that followed his design, the most significant bit of each word determined whether that word was an instruction or data, so the rule could be enforced.

Within a couple of years, however, designers had figured out a better way to implement 'variable addresses' in hardware, specifically though the introduction of one or more new registers to hold a variable address. The first machine to implement this was the British-designed, Manchester Mark I. The new register was called the 'B' register, to distinguish it from the 'A' (accumulator) register. Later microprocessors introduced an 'X' (and sometimes a 'Y') register specifically to hold an index.

With the introduction of RISC ('Reduced Instruction Set Computer') processors, of which ARM is an example, dedicated registers were replaced with a set of general-purpose registers that could be used to hold data values or address values.

Picture credit: https://en.wikipedia.org/wiki/John_von_Neumann

---

In the example above, the indirect address, held in R4, is made up of from a constant value held in R1, (the starting address for the grid of pixels) plus a variable number (in R3). We could say that the value in R1 is the 'base' address and the value in R3 is a variable 'index', added to the base. For this circumstance, which is very common, there is an even simpler syntax, known as 'indexed addressing', which (on ARMlite) is a specialised form of indirect addressing, as shown below:

```
    MOV R1, #.PixelScreen //Start of the mid-res pixel screen memory - CONSTANT
    MOV R2, #.red
    MOV R3, #0 //This is our counter or 'index'
loop: STR R2,[R1+R3] //Paint the pixel at address defined by R1+R3
    ADD R3,R3,#4 //Increment the index (by 4 bytes = 1 word = 1 pixel)
    CMP R3,#256 //i.e. a total of 64 pixels
    BLT loop
    HALT
```

Notes:

- Using indexed addressing we have eliminated an instruction (`ADD R4,R1,R3`) from the previous version, and the need to use another register (`R4`).
- We still need the square brackets around the argument – `[R1+R3]` – because the sum of the values of those two registers forms an indirect address.

Teacher note

The conventional *ARM* syntax for indexed addressing is `STR R2, [R1,R3]`. ARMlite will recognise this syntax as an alternative. We introduced the `STR R2, [R1+R3]` syntax for ARMlite because we believe it is easier for pupils to read and understand.
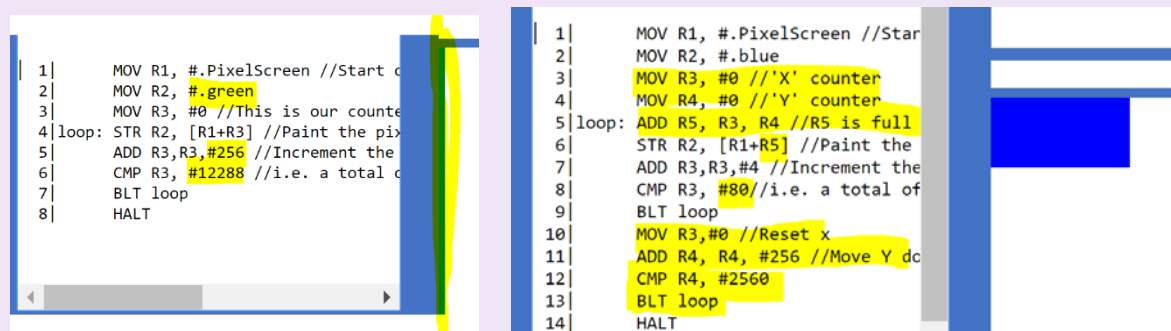

Exercise 27

Run the revised version of the code above (i.e. using indexed addressing) and validate for yourself that it produces the same result.

Now modify the code so that it draws, instead, a *vertical, green* line, starting at the top left position. Remember that there are now 64 pixels per line, not 32, and that there are still 4 bytes (= 1 word) per pixel. Paste a partial screenshot showing your new code and the result.

Finally, write some code to draw a solid blue rectangle 20 pixels wide, by 10 pixels deep, starting at the top left pixel as before. Now you will need to have two registers, keeping track of the 'x' and 'y' coordinates, combining them into a third register that forms the index to be added to `R1` to draw the pixel. *Comment your code.* Paste a partial screenshot showing your new code and the result.

Model answer 27


```
 1|      MOV R1, #.PixelScreen //Start
 2|      MOV R2, #.green
 3|      MOV R3, #0 //This is our counte
 4|loop: STR R2, [R1+R3] //Paint the pix
 5|      ADD R3,R3,#256 //Increment the
 6|      CMP R3, #12288 //i.e. a total
 7|      BLT loop
 8|      HALT
```

```
 1|      MOV R1, #.PixelScreen //Star
 2|      MOV R2, #.blue
 3|      MOV R3, #0 //'X' counter
 4|      MOV R4, #0 //'Y' counter
 5|loop: ADD R5, R3, R4 //R5 is full
 6|      STR R2, [R1+R5] //Paint the
 7|      ADD R3,R3,#4 //Increment the
 8|      CMP R3, #80//i.e. a total of
 9|      BLT loop
10|      MOV R3,#0 //Reset x
11|      ADD R4, R4, #256 //Move Y do
12|      CMP R4, #2560
13|      BLT loop
14|      HALT
```

Indexed addressing is the means by which a high-level language implements arrays, allowing the retrieval of any element of an array in O(1) time rather than O(n).

Implementing Bubble Sort using indexed addressing

In the following example we will use indexed addressing to sort an 'array' of numbers held as data in memory. We'll use the 'Bubble sort' algorithm because it is simple to write and to observe when running slowly. It is, though, one of the least efficient sorting algorithms.

```
// Define & initialise registers
    MOV R0, #arrayData //Address of array data
    LDR R1, arrayLength
    LSL R1, R1, #2 //Multiply this by 4 (for byte count)
    MOV R2, #0  //outerLoop counter initialized
    // R3 innerLoop counter
    // R4 length of innerLoop
    // R5 spare index into array
    // R6 first in pair
    // R7 second in pair
    // R8 ?
startOfOuterLoop:
    MOV R3, #0 //reset inner loop counter to zero
    SUB R4, R1, R2 //set innerloop max to data length - outerLoopCounter...
    SUB R4, R4, #4 //...minus 4 more
    MOV R8, #0
innerLoop:
    LDR R6, [R0+R3] //Load first value from address (base + index)
    ADD R5, R3, #4  //Generate index for second value in pair
    LDR R7, [R0+R5] //Load second value
    CMP R6, R7       //Compare and swap if appropriate
    BGT swap
    B continueInnerLoop
swap:
    STR R7, [R0+R3]
    STR R6, [R0+R5]
    MOV R8, #1
continueInnerLoop:
    ADD R3, R3, #4
    CMP R3, R4 //Check if reached the end
    BLT innerLoop
continueOuterLoop:
    CMP R8, #0
    BEQ done
    ADD R2, R2, #4
    CMP R2, R1
    BGT done
    B startOfOuterLoop
done: HALT
.ALIGN 256 //Just to make data distinct from code in memory view
arrayLength: 10
arrayData: 9 //1st element
    8
    7
    6
    5
    4
    3
    2
    1
    0 //last element
```

Exercise 28

Enter the program, and submit, but don't run. Inspect the data in memory, starting at address `0x00100`. Then run the program and inspect the data again. Confirm for yourself that it has been sorted into ascending order.

Now modify the starting data, using randomly chosen values of multiple decimal digits. Add more data elements to the end, but adjust the value in `arrayLength` to reflect your new array size.

Submit your code and this time capture a partial screenshot showing just the array data in the *Memory* view, before and after sorting.

In the `// Define & initialise registers` section you will see: `// R8 ?`. Your final task is to identify the meaning and role of `R8`.

Start by identifying all the lines of code where `R8` is referenced. Then trace through the algorithm, either on paper, or by running the program in Slow and/or Single Step mode, to figure out what role `R8` is performing. Describe it in your own words. What advantage does the coding relating to `R8` confer on the routine?

Model answer 28

Before (example)

0x0010	0x0000000c	0x00000013	0x000001d7	0x0000030c
0x0011	0x0000003d	0x00000005	0x0000029b	0x00000022
0x0012	0x0000012c	0x0000000c	0x0000000b	0x0000000b
0x0013	0x00000055	0x00000000	0x00000000	0x00000000

After

0x0010	0x0000000c	0x00000005	0x0000000b	0x0000000b
0x0011	0x0000000c	0x00000013	0x00000022	0x0000003d
0x0012	0x00000055	0x0000012c	0x000001d7	0x0000029b
0x0013	0x0000030c	0x00000000	0x00000000	0x00000000

R8 is tracking whether, or not, a swap has taken place within the last pass of the inner loop. If no swap has taken place on the last pass, then the data is deemed sorted and the outer loop is terminated. This, on average, will improve the performance, except for the worst-case scenario where the data starts in reverse order.

Implementing a binary search using indirect addressing

In the following code we use indirect addressing to implement a binary search algorithm.

```
// Define registers
   //R0 Target value
   //R1 Pointer to first data item
   //R2 Pointer to mid-point
   //R3 Pointer to last data item
   //R4 Temp data value
   //R5 Temp use to display messages
start:
   MOV R1, #first
   MOV R3, #last
   MOV R5, #msg1
   STR R5, .WriteString
   LDR R0, .InputNum
   STR R0, .WriteUnsignedNum
loop:
   ADD R2, R1, R3
   LSR R2, R2, #3 //Divide by 8, then...
   LSL R2, R2, #2 //...multiply by 4. Net effect is divide by 2, but modulo 4.
   LDR R4, [R2]  //Get mid-point value
   CMP R0,R4     //Compare target to mid value
   BEQ found
   BLT belowMid
   //Must be above mid if here
   MOV R1, R2
   ADD R1, R1, #4 //start = mid + 4 (bytes)
   B checkForOverlap
belowMid:
   MOV R3, R2
   SUB R3, R3, #4 //start = mid - 4 (bytes)
   B checkForOverlap
checkForOverlap:
   CMP R1, R3
   BGT notFound
   B loop
notFound:
   MOV R5, #msg3
   STR R5, .WriteString
   B start
found:
   MOV R5, #msg2
   STR R5, .WriteString
   STR R2, .WriteHex
   B start

msg1: .ASCIZ "\nSearch for ?"
msg2: .ASCIZ "\nIs at memory location: "
msg3: .ASCIZ "\nNot found!"

.ALIGN 256 //Just to separate data from code in the memory view
first: 3
   6
   7
   15
   22
   24
   31
   50
   79
last: 94
```
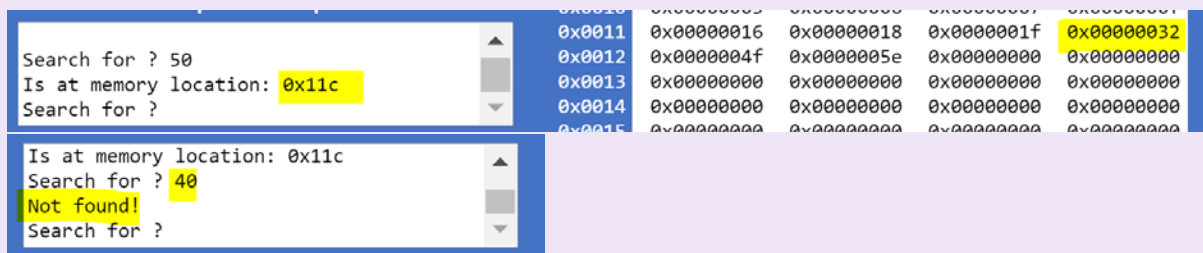
Exercise 29

Run the program above. Paste a screenshot that shows a successful search for a number that is in the array, highlighting in the memory area the hex address that has been returned.

Then show the result of searching for a number that isn't in the array.

What assumption does the binary search algorithm make about the data being searched?

Model answer 29

The binary search algorithm assumes that the data is sorted. In this implementation it assumes that the data is sorted in increasing order.

Chapter 6: The System Stack, and Subroutines

You have probably already encountered the idea of the 'stack' data structure (if not, see panel) and you might have implemented a stack in a high-level programming language, using an array and a pointer. You might also have learned that on your computer there is something called 'the system stack', if only by accident – when you encountered a 'Stack Overflow Error'.

The system stack is typically implemented right down at the processor level. ARMlite supports a system stack, and there are dedicated assembly language instructions for using it: **PUSH** and **POP**.

Stacks

In Computer Science a 'stack' is a data structure designed specifically for holding values temporarily. Think of one of those spring-loaded plate 'dispensers' (pictured) that you see in some eateries. These hold a stack of plates, but only the topmost plate is visible. If you remove a plate the next one 'pops' up in its place; and if you add a plate it 'pushes' the others down.

We can describe a stack as a 'LIFO' data structure, which stands for 'Last In, First Out'. Items are removed from the stack in the reverse of the order in which they were added.

When we make use of a stack in programming, the function for adding an item to the stack is typically called 'push', and the function for removing an item is typically called 'pop'. (Sometimes there is also a function called 'peek', which allows you to inspect the item at the top of the stack without removing it.)

Teacher note - PEEK
To do a 'peek' on ARMlite, the stack pointer, **SP**, can be used as an indirect address e.g. `LDR R0, [SP]`

It is sometimes convenient to make use of this system stack within your own assembly language code. The following program asks the user to type in a string, which is stored to a memory location. Then the string is reversed, by pushing the characters successively onto the stack (making use of indexed addressing to work through them), and then popping them and storing them back from the beginning, before writing the modified string to the console. There are certainly other ways to implement this functionality, but the use of a stack is a simple and common pattern:

```
// Define registers
// R0 Used for writing messages to screen
// R1 Index into string in memory
// R2 holding individual string characters
   MOV R0, #myString
   STR R0, .ReadString
   MOV R1, #0
   MOV R2, #0
   PUSH {R2} //So that when we pop the string we can detect the end
loop1: LDRB R2, [R0+R1]
   CMP R2, #0
   BEQ popAll
   PUSH {R2}
   ADD R1, R1, #1 //Because we're now working one byte at a time
   B loop1
popAll: MOV R1, #0 //Reset index
loop2: POP {R2}
   CMP R2, #0 //Look for end of string marker
   BEQ write
   STRB R2, [R0+R1]
   ADD R1, R1, #1
   B loop2
write: STR R0, .WriteString
   HALT
myString:
```

Notes:

- For both the PUSH and POP instructions, the operand is surrounded by braces { … }. This is because it is possible to push or pop multiple registers in one instruction, so the braces effectively define a list of registers. We'll see how to do that shortly.
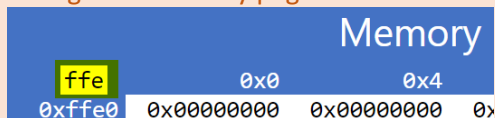

Exercise 30

Run the program and paste a partial screenshot showing the input string and the result.

Now run it again in **Slow** mode and, after inputting the string, observe the SP ('**S**tack **P**ointer') register. Describe what you see happening there.

Now click **Edit** then **Submit** (this is just to reset the memory).

Change the memory page selector from 000 to ffe as shown:

Now run again in slow mode, switching between **Single** step and **Slow** if you wish, this time using **ABCDE** as your input word (because you should be able to recognise the hex values for the ASCII codes – 0x41 to 0x45). Once you've entered the word describe what you observe happening in the visible page of memory. Describe also how this appears to relate to the value in SP.
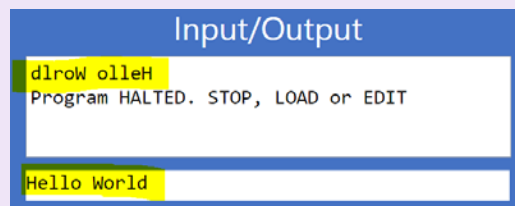
> **Model answer 30**
>
>
>
> The **SP** register starts at **0x00100000** and decreases with each iteration of **loop1**, by **4** units. Then, in **loop2** it increases again, in units of **4**, until it gets back to the starting value.
>
> The characters are written to successive words (one character per word) at the end of main memory, working backwards. The stack pointer points to the current 'top' of the stack (which is one past the end of memory when the stack is empty). When the stack is popped, however, the stack pointer is altered but the words of memory are not cleared.

Why are the values (one character per word in this case) not cleared when the stack is emptied? Because there is no need to do so - it would just be wasting processing time to clear the word. The stack pointer tells the processor where to push (or pop) the next value.

Now consider the following short program:

```
loop: MOV R0, #0xffff
    PUSH {R0}
    B loop
```

> **Exercise 31**
>
> Run the above program until it stops.
>
> Describe what the program has done and why has it stopped.
>
> *Where* has it stopped i.e. what was the last memory location that was written. (Hint: look at the contents of memory on page **000**).

> **Model answer 31**
>
> The program writes the same value (**0x0000ffff**) to every location in memory until it stops, when a **Stack overflow error** message is written on the console.
>
> The last memory location written to is **0x0000c** i.e. the word *after* the three words taken up by the program itself.

What you have hopefully discovered is that ARMlite, like most modern processors, knows where your program (and any data areas that you have declared in the assembly code) stops. It will allow the stack to expand to use up all the available memory (in this case almost a megabyte) but not to overwrite your instructions, or declared data areas.

Subroutines

The following code will allow you to draw any one specific pixel in mid-res mode. The pixel is specified as X, Y coordinates in **R0** & **R1**respectively, and with the colour specified in **R3**:

```
    MOV R0, #32 //X coordinate (0-63)
    MOV R1, #24 //Y coordinate (0-47)
    MOV R2, #.red //or any colour

 drawPixel: MOV R3, #.PixelScreen
    LSL R4, R0, #2 //Multiply X coordinate by 4
    LSL R5, R1, #8 //Multiply Y coordinate by 256
    ADD R5, R5, R4 //Get the pixel index
    STR R2, [R3+R5]
    HALT
```

We could build this code into a loop in order to draw a line, say, but if we wanted to draw just two arbitrary pixels, we would need to duplicate the five instructions from **drawPixel** onwards. This violates the DRY principle ('Don't Repeat Yourself') of good coding. We could easily branch to this code from multiple places, but how would we specify that, at the end of the routine, the code must branch back *to a different place each time*?

The answer lies in the idea of a subroutine. The following code shows the same **drawPixel** code, now set up as subroutine, and invoked twice:

```
    MOV R0, #32 //X coordinate (0-63)
    MOV R1, #24 //Y coordinate (0-47)
    MOV R2, #.red //or any colour
    BL drawPixel
    MOV R0, #37
    MOV R1, #19
    MOV R2, #.green
    BL drawPixel
    HALT

//Subroutine
drawPixel: MOV R3, #.PixelScreen
    LSL R4, R0, #2 //Multiply X coordinate by 4
    LSL R5, R1, #8 //Multiply Y coordinate by 256
    ADD R5, R5, R4 //Get the pixel index
    STR R2, [R3+R5]
    RET
```

Notes:

- After setting up the required values in **R0**, **R1**, and **R2**, we branch to the **drawPixel** code, but this time using a **BL** ('**B**ranch with **L**ink back') instruction rather than a regular **B** instruction.
- At the end of the **drawPixel** routine, there is a new instruction **RET** ('Return') which returns execution *to the instruction following the last **BL** to be invoked*.
- Here, the values held in in **R0**, **R1**, and **R2** may be considered as 'parameters' passed into the subroutine.

The Wheeler Jump

The invention of the subroutine is generally credited to David Wheeler, who worked on the EDSAC at Cambridge University in the late 1940s. The idea of having re-usable general-purpose routines was not new, but until his invention, re-using them meant inserting the routine into your own code wherever you needed it. The idea of a routine that could be called from several places required a different kind of 'branch' or 'jump' (the terms were synonymous) that would preserve a link back. As a result, for many years, this new instruction was referred to informally as the 'Wheeler jump'. On ARMlite, the equivalent is **BL**.

Picture credit: https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

Exercise 32

Run the program above and check its operation. Extend it with another call to draw a third pixel, *changing only one of the three parameters*. Paste a partial screenshot showing your code and the resulting pixel screen.

Now run the program again, using **Single Step** this time. For each step, observe closely the values in the **PC** and **LR** registers, and then describe, *in some detail,* how these two registers are being used both during normal execution, and when a subroutine is invoked.

Model answer 32

```
 1|      MOV R0, #32 //X coordinate (
 2|      MOV R1, #24 //Y coordinate (
 3|      MOV R2, #.red //or any colou
 4|      BL drawPixel
 5|      MOV R0, #37 //X coordinate (
 6|      MOV R1, #19 //Y coordinate (
 7|      MOV R2, #.green//or any colo
 8|      BL drawPixel
 9|      MOV R0, #39
10|      BL drawPixel
11|      HALT
12|drawPixel: MOV R3, #.PixelScreen
13|      LSL R4, R0, #2 //Multiply X
14|      LSL R5, R1, #8 //Multiply Y
15|      ADD R5, R5, R4 //Get the pix
16|      STR R2, [R3+R5]
17|      RET
```

In normal execution, at the end of each instruction the **PC** points to the address of the following instruction. The value in **LR** is unchanged (from zero).

Immediately after a **BL** the **PC** is changed to the address indicated by the operand of the **BL** (in this case the address of the **drawPixel** label). **LR** is changed to point to the instruction immediately after the **BL**.

Execution then continues in the subroutine code, with **PC** changing as usual, and **LR** unmodified, until the **RET** instruction, when the address in **LR** is copied back to the **PC**. Note that **LR** is not cleared at this point, but will be over-written by the next **BL**.

Teacher note
RET in ARMlite is in fact just a shorthand way of writing **MOV PC,LR**

LR as you have probably guessed by now, stands for '**L**ink **R**egister' - it holds the link referred to 'Branch with Link back'.

One of the principles of good programming is 'separating the interface from the implementation': we should be able to invoke a subroutine just by knowing its label, and by setting up required parameters (if any) in the appropriate registers. (A subroutine may also act as a 'function' by returning a result, by putting the result into one or more defined registers). We shouldn't have to care, or even know, about how the subroutine works internally.

But there's a potential problem. *Before reading on*, see if you can figure out what that problem might be? Think about invoking the **drawPixel** routine from lots of places in a much larger, more complex, program …

The problem is that **drawPixel** makes use of registers **R3**, **R4**, and **R5**, and in a larger program we might be using any or all those registers for other purposes. Calling the subroutine without inspecting its inner code, might cause that data to be lost or modified without us realising, thus causing errors that might not be noticed initially. We could re-write our **drawPixel** routine to use *fewer* additional registers, certainly, but the problem will still exist.

The solution makes use of the system stack:

```
//Draws a single pixel (medium resolution)
//Specify X coordinate (0-63) in R0
//Specify Y coordinate (0-47) in R1
//Specify colour in R2
drawPixel:
   PUSH {R3-R5, LR}
   MOV R3, #.PixelScreen
   LSL R4, R0, #2
   LSL R5, R1, #8
   ADD R5, R5, R4
   STR R2, [R3+R5]
   POP {R3-R5,LR}
   RET
```

Notes:

- We have moved all the comments out of the code to above the 'entry point' of the subroutine. The comments explain just what you need to know to use the subroutine, without having to look at the internal implementation.
- The first instruction in the subroutine pushes multiple register values onto the stack – including the value of any registers that we are going to change, or *might* change, within the routine
- The last instruction restores the original values of any changed registers, by popping them from the stack.
- The only changed registers *not restored* to their original values would be any results being passed back (which should be documented in the comments). Our **drawPixel** routine does not pass back any result - its result is shown on the pixel screen.
- **PUSH** and **POP** may take any **R0** to **R12**, and/or **LR**, either as a comma-separated list of individual registers, or contiguous ranges of numbered registers (e.g. **R3-R5**), or a combination.

- The list of registers for the **PUSH** and **POP** instructions should be identical: then it is the responsibility of the processor to ensure that the values go back to the correct registers, even though they will have been through the LIFO mechanism of the stack.

But why have we included **LR** in the list of registers to be saved and restored, when we are not explicitly changing it within our code, and the example code already worked correctly?

The answer is that in a larger program we will have subroutines that call other subroutines. (We can even have subroutines that call themselves 'recursively', which is the most obvious way to implement a Merge Sort routine, for example. Each nested, or recursive subroutine call, will push another set of values (known as a 'stack frame') onto the stack, popping them again until all the calls are completed. But the **LR** can only remember one 'link back' address at a time. By saving and restoring the **LR** within the subroutine we ensure that the 'return address' isn't being lost by any nested within our code. OK, we haven't got any nested **BL** call within our **drawPixel** routine. But, *trust us*, if you fail to save and restore the **LR** when you *do* need to, the resulting bugs can sometimes be very hard to diagnose - so *it is simply a good, safe practice to always save and restore* **LR** *in every subroutine, even though it is very slightly wasteful*.

A Multiply subroutine

The following subroutine will multiply two integers:

```
//Multiplies two integers in R0 & R1 returing the product in R2
//The product must fit within 32 bits to be correct.
multiply:
    PUSH {R0,R1,R3,LR}
    MOV R2, #0 // result
processRightmostDigit:
    AND R3, R1,#1 //To test rightmost bit
    CMP R3, #0
    BEQ skip    //Rightmost bit is a 0
    ADD R2,R2, R0
skip:
    LSR R1,R1, #1
    CMP R1, #0  //If there are no more digits
    BEQ end
    LSL R0,R0,#1
    B processRightmostDigit
end:
    POP {R0,R1,R3,LR}
    RET
```

Exercise 33

Write a program that calls this subroutine within a loop, to print out the 'times table' for any selected single digit number, up to the 9th multiple. For example, if the user enters 5, the console should show:

```
1 x 5 = 5
2 x 5 =10

...

9 x 5 = 45
```

Capture your complete program, and a screenshot showing the multiples of 7 (or as much as fits on the console).

Model answer 33

```
 1|        LDR R0, .InputNum
 2|        MOV R1, #1
 3|loop: bl multiply
 4|        STR R0, .WriteUnsignedNum
 5|        MOV R3, #120 //ASCII for 'x'
 6|        STR R3, .WriteChar
 7|        STR R1, .WriteUnsignedNum
 8|        MOV R3, #61 //ASCII for '='
 9|        STR R3, .WriteChar
10|        STR R2, .WriteUnsignedNum
11|        MOV R3, #10 //ASCII for new lin
12|        STR R3, .WriteChar
13|        ADD R1, R1, #1
14|        CMP R1, #10
15|        BLT loop
16|        HALT
17|//Multiplies two integers in R0 & R1
18|//The product must fit within 32 bits
19|multiply:
20|        PUSH {R0,R1,R3,LR}
21|        MOV R2, #0 // result
22|processRightmostDigit:
23|        AND R3, R1,#1 //To test rightmo
24|        CMP R3, #0
25|        BEQ skip //Rightmost bit is a 0
26|        ADD R2,R2, R0
27|skip:
28|        LSR R1,R1, #1
29|        CMP R1, #0 //If there are no mo
30|        BEQ end
31|        LSL R0,R0,#1
32|        B processRightmostDigit
33|end:
34|        POP {R0,R1,R3,LR}
35|        RET
```

Register	Value
PC	0x00000040
LR	0x0000000c
SP	0x00100000
R12	0x00000000
R11	0x00000000
R10	0x00000000
R9	0x00000000
R8	0x00000000
R7	0x00000000
R6	0x00000000
R5	0x00000000
R4	0x00000000
R3	0x0000000a
R2	0x0000003f
R1	0x0000000a
R0	0x00000007

Inp

```
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
```

7

Teacher note – What about division?

The algorithm for the division of two binary integers is essentially the reverse of multiplying by shifting and adding. It returns a quotient and a remainder:

- Set the quotient to 0
- Align leftmost digits in dividend and divisor
- Repeat
  - If that portion of the dividend above the divisor is greater than or equal to the divisor
    - Then subtract divisor from that portion of the dividend and
    - Concatenate 1 to the right-hand end of the quotient
    - Else concatenate 0 to the right-hand end of the quotient
  - Shift the divisor one place right
- Until dividend is less than the divisor
- quotient is correct, dividend is now the remainder

There is a useful diagram of this algorithm here:
https://courses.cs.vt.edu/~cs1104/BuildingBlocks/divide.030.html

The *implementation* of this algorithm in assembly language is a little bit more complex than multiplication, however. For those interested, here is a divide subroutine for ARMlite:

```
//Divides integer  R0 by integer R1, returning the quotient in R2 & remainder in R3
//Input numbers must be positive numbers of 30 bits or less
divide:
    PUSH {R0,R1,R4,LR}
    MOV R4,#1
div1:  CMP R0,R1
    BLT divNext // loop until divisor >= dividend
    LSL R1,R1,#1
    LSL R4,R4,#1
    B div1
divNext:    MOV R2,#0
div2:  CMP R1,R0
    BGT divNoSub
    SUB R0,R0,R1
    ADD R2,R2,R4
divNoSub: LSR R4,R4,#1
    LSR R1,R1,#1
    CMP R4,#0
    BNE div2
    MOV R3, R0
    POP {R0,R1,R4,LR}
    RET
```

Chapter 7: Interrupts

Interrupts are a mechanism for responding to events. In an industrial control system, or an engine management system, an event might arise from a hardware sensor such as a pressure switch, or thermostat. Events can also be generated inside the computer: from a hardware timer, a key press, or a mouse movement.

Without interrupts the only way to handle such events is to 'poll' each possible event source, regularly, to see if it has been activated. This is inefficient, and requires a lot of specialised code. It is also difficult to ensure that the poll is executed at regular intervals, since responding to different events may take different amounts of time. And in many cases, there is a 'main' program to be executed as well.

An interrupt is a hardware mechanism that effectively pauses execution of the main program, branches to a special type of subroutine that determines the source and nature of the event, and any immediate action that needs to be taken, and then resumes execution of the main program.

Interrupts and the Fetch-Execute cycle

Interrupts are processed only at the end of each fetch-execute cycle. Why? To allow interrupts *during* fetch, decode, or execute would not permit the state of the processor to be saved in a consistent manner, and therefore it would not possible to resume processing, safely, once the interrupt had been processed.
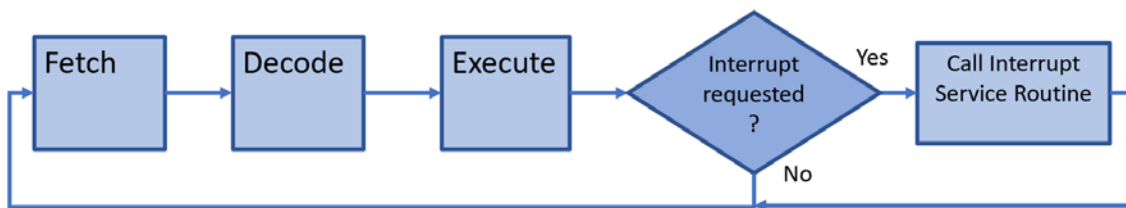
Interrupt storm

On systems that have frequently interrupts there is a risk of an 'interrupt storm', where new interrupts are being generated faster than they can be processed. Systems designers must avoid this possibility, for example by:

- ensuring there is sufficient processing power to render the possibility unlikely
- providing an interrupt prioritisation mechanism in hardware, so that high priority interrupts can interrupt the handling of lower priority interrupts
- deliberately 'throttling' interrupts: using hardware to limit the minimal interval between interrupt requests.

Pin interrupts

Early microprocessors had a single pin on the package labelled something like 'IRQ' (for '**I**nterrupt **R**e**Q**uest'). ARMlite simulates such a pin, but it is made visible on screen only when interrupts have been explicitly enabled. We are now going to use it. First, we need to write a main program that keeps the processor busy:

```
//Main program
   MOV R1, #.PixelScreen
   MOV R2, #0 //Pixel index
loop: LDR R0, .Random //Colour
   STR R0, [R1+R2]
   ADD R2, R2, #4
   CMP R2, #12288
   BLT loop
   MOV R2, #0
   B loop
```

> **Exercise 34**
>
> Run the program above and verify for yourself that it runs continuously - and fast. Pause at some point and paste a partial screenshot showing the output.

> **Model answer 34**
>
>

We want to interrupt this main routine, and write something to the console each time. To keep it simple we will write a subroutine that just writes the character **A** to the console each time it is called:

```
//Interrupt Routine
writeA: PUSH {R0}
   MOV R0, #65
   STR R0, .WriteChar
   POP {R0}
   RFE
```

Notes:

- As with any subroutine, if we are going to make any use of registers, we must take care to preserve and restore the values in those registers on the stack, using **PUSH** and **POP**.
- However, we do not need to save and restore the value of **LR** (unless we are explicitly modifying it in our routine, which would be unusual), because when an interrupt routine is called, the processor does not make use of **LR** for the return link. (We'll see how it manages the return link shortly).

- Because of this different approach to managing the return to the main program, we end our routine not with **RET** instruction (as we do for normal subroutines) but with an **RFE** ('**R**eturn **F**rom **E**xception') instruction, because an interrupt is an exceptional circumstance.

Having added the interrupt routine, we need to specify how and when this is to be called. This is done with the following code, *at the start of the program*:

```
// Set Up Interrupts
   MOV R0, #writeA
   STR R0, .PinISR //Specify the routine to call when the interrupt Pin is set
   MOV R0, #1
   STR R0, .PinMask //Enable the interrupt Pin, by setting bit 0 to 1
   STR R0, .InterruptRegister //Enable interrupts generally
```

Notes:

- As explained in the comments, this code does three *necessary* things.
- It might seem odd that having specified that you want the 'Pin interrupt' to call the routine **writeA**, that you still need to 'enable the interrupt Pin', and then to 'enable interrupts generally'. This is needed because (as we shall see) ARMlite supports different kinds of interrupts, and, for some applications, it is important to enable/disable each type of interrupt, or disable all interrupts, during certain critical parts of the main program.
- ARMlite's coding patterns for interrupts closely matches those of a real ARM processor, although the specific forms of interrupt are particular to this simulation.

Exercise 35

Add the code for the `Interrupt Routine` after the end of the `Main Program`, and then add the `Set Up Interrupts` code *at the start*.

Run the program and verify that the main program continues to execute as before.

However, you will now see that the 'Interrupt Pin' has appeared within the processor area:
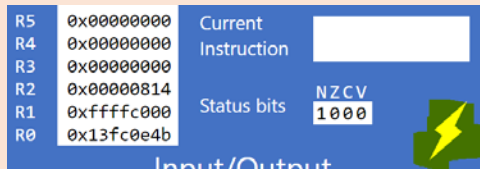
Verify that each time you click this icon (while the program is running) an `A` is written to the console.

Does this interrupt *appear to* slow down the main program?

Pause the program, put a breakpoint on the `MOV R0, #65` line , then resume running.

Click the interrupt Pin (remember that the processor will pause just *before* this breakpoint is reached).

What is the value in `LR`?

Without resuming execution, go to memory page `ffe`, to view the stack at the end of the page. The stack pointer (`SP`) will tell you the memory address of the top of the stack at this point. It will indicate that there are three values currently in the stack. Paste a partial screenshot showing those three values.

What is the significance of the value at the top of the stack (i.e. in the memory address held in `SP`). (hint: look for this value elsewhere on the screen)?

Press **Single Step** four times. From which instruction does the execution of the main program resume? How does the processor know where to resume from?

Model answer 35

Clicking the interrupt pin does not *appear* to slow down the main program, but that is because the interrupt routine is very short. In fact, the interrupts do pause the main program while the interrupt routine is being executed.

The value of `LR` is `0x0` - it remains unused in this program.

`SP` indicates that the top of the stack is `0x000fff4` i.e. there are three values in the stack.

| 0xffff | 0x00000000 | 0x6cc2bd8f | 0x00000024 | 0x80000001 |
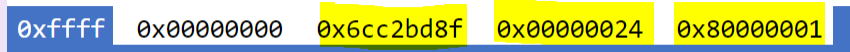
(Note: the first value will probably be different for each pupil's answer. The second one is likely to be `0x24`, but *could* be different).

The value at the top of the stack is the current value of `R0` (the random colour), which has been pushed onto the stack by the first instruction in the interrupt routine.

After four single steps, the highlighter will return to the instruction in the main program, at the memory address that is held in the second word on the stack e.g. `0x24`. When the `RFE` instruction is called this value is popped from the stack and then copied into the `PC`.

The last value in the stack (`0x80000001`) is the means by which the processor saves its current status, including the four status bits that you can see on the ARMlite's user interface. The reason for this is that any of those items could be changed within an interrupt service routine, and need to be restored.

The Interrupt Pin icon simulates the way that a hardware pin works on a real microprocessor. On a real computer this pin will be connected to one or more hardware devices such as the keyboard, mouse, or a hardware timer (regular pulse generator). ARMlite simulates such capabilities.

Keyboard Interrupts

Through small modifications to the previous code, shown highlighted below we can use the simulated keyboard interrupts:

```
// Set up interrupts
    MOV R0, #writeA
    STR R0, .KeyboardISR
    MOV R0, #1
    STR R0, .KeyboardMask
    STR R0, .InterruptRegister
//Main task - random dots
    MOV R1, #.PixelScreen
    MOV R2, #0 //Pixel index
loop: LDR R0, .Random //Colour
    STR R0, [R1+R2]
    ADD R2, R2, #4
    CMP R2, #12288
    BLT loop
    MOV R2, #0
    B loop
//Interrupt routine
writeA: PUSH {R0}
    LDR R0, .LastKey
    STR R0, .WriteChar
    POP {R0}
    RFE
```

Exercise 36

Make the modifications shown above and run the program. Is the Interrupt Pin now shown?

While the program is running, type characters on the keyboard and verify that they are immediately written to the console.

Does it distinguish between upper and lower case letters?

Model answer 36

The Interrupt Pin is no longer shown because *pin interrupts* are not now being set up or enabled at the start of the program (we changed the code to use keyboard interrupts instead).

No, for keyboard interrupts, ARMlite is restricted to using upper case letters and numbers only.

Clock interrupts

Many applications need accurate knowledge of the passage of time, to pace things. This is achieved by means of a hardware timing device that interrupts the processor on a regular basis. ARMlite can simulate this capability, as shown in the following code which causes a single large pixel to flash black and white:

```
// Set up Interrupt handling
    MOV R0,#pixelClock
    STR R0,.ClockISR
    MOV R0,#1000
    STR R0,.ClockInterruptFrequency
    MOV R0,#1
    STR R0,.InterruptRegister //Enable all interrupts

mainProgram: B mainProgram //Here, just an empty loop!

pixelClock:
    PUSH {R0,R1}
    MOV R1, #.white
    LDR R0, .Pixel0
    EOR R0,R0,R1 //ExOr with all 1s switches black to white or vice versa
    STR R0, .Pixel0
    POP {R0,R1}
    RFE
```

Exercise 37

Run the program. Time the flashing pixel with your watch - what is its periodicity?

The frequency of interrupts is set by `ClockInterruptFrequency`.

How would you get the pixel clock to flash black and white once a second?

Model answer 37
The pixel switches between black and white once a second – i.e. two seconds for the full cycle, or 0.5 Hz.

To get a 1 Hz frequency set `MOV R0, #0x500`

Click-pixel interrupts

The other form of interrupt that ARMlite can handle is the user clicking with the mouse on a pixel within the (mid-res, or hi-res) pixel screen. This can be useful for writing simple drawing applications, or for interactive games.

The following example code will paint a pixel red wherever the user clicks within the pixel screen:

```
//Set up interrupts
   MOV R0, #paint
   STR R0,.PixelISR
   MOV R0,#1
   STR R0,.PixelMask //Set pixel click interrupts on
   STR R0,.InterruptRegister //Enable all interrupts
//Set up colour
   MOV R12, #.red

mainLoop: B mainLoop //Does nothing at present

//Interrupt driven routine to paint a pixel that user clicks on
//Colour is specified in R12 (global variable)
paint:PUSH {R1,R2}
   MOV R1, #.PixelScreen
   LDR R2, .LastPixelClicked
   LSL R2,R2,#2 //Multiply pixel number by 4 to get byte address
   STR R12, [R1+R2]
   POP {R1,R2}
   RFE
```

> **Exercise 38**
>
> Run the program above and validate that you can click on any pixel to paint it red.
>
> Create an additional, *separate,* interrupt routine that is driven by *keyboard interrupts* (refer to previous section) and, depending on which of two (or more) keys is hit, switches the value of the colour in the global variable R12.
>
> Run the modified program and show, with a partial screenshot that you can now create pixels of different colours (i.e. the screen should show multiple pixels of at least two colours). Also capture your modified code.

Model answer 38

```
 1 //Set up interrupts
 2      MOV R0, #paint
 3      STR R0,.PixelISR
 4      MOV R0,#1
 5      STR R0,.PixelMask //Set pixel c
 6      MOV R0,#switchColour
 7      STR R0, .KeyboardISR
 8      MOV R0,#1
 9      STR R0, .KeyboardMask
10      STR R0,.InterruptRegister //Ena
11 //Set up colour
12      MOV R12, #.red
13 mainLoop: B mainLoop //Does nothing a
14 //Interrupt driven routine to paint a
15 //Colour is specified in R12 (global
16 paint:PUSH {R1,R2}
17      MOV R1, #.PixelScreen
18      LDR R2, .LastPixelClicked
19      LSL R2,R2,#2 //Multiply pixel n
20      STR R12, [R1+R2]
21      POP {R1,R2}
22      RFE
23 //Keyboard interrupt driven
24 switchColour:
25      PUSH {R0}
26      LDR R0, .LastKey
27      CMP R0, #82 //'R'
28      BNE .+2
29      MOV R12,#.red
30      CMP R0, #71 //'G'
31      BNE .+2
32      MOV R12,#.green
33      POP {R0}
34      RFE
```

PC	0x00000000
LR	0x00000000
SP	0x00100000
R12	0x00000000
R11	0x00000000
R10	0x00000000
R9	0x00000000
R8	0x00000000
R7	0x00000000
R6	0x00000000
R5	0x00000000
R4	0x00000000
R3	0x00000000
R2	0x00000000
R1	0x00000000
R0	0x00000000

Count

Current
Instruction

Status bits

Input/Out

Interrupts and the operating system

As you might have realised, the code you have just written for reading the keyboard and writing the character to the console, for maintaining a real-time clock, and responding to mouse clicks, while other processes are running, emulates – in a simple fashion – two of the core functions of the operating system on your computer.

Chapter 8: Snake

In this chapter we are going to use many of the techniques we have learned in this book to build a realistic implementation of the well-known Snake game.

Snake

Most people who play video games will have played 'Snake' or 'Serpent' at some point: steering a snake around the screen to gobble apples, the snake growing one segment longer for each apple eaten. More sophisticated versions of the game exist for modern PCs and mobile phones, but the earliest versions were created for hand-held dedicated game devices, with low-resolution monochrome liquid-crystal displays, like the one shown on the right.

The earliest versions of these devices were built using 8-bit microprocessors, with very limited memory. Both for performance and memory efficiency, the software for this and similar games was written in assembly language and then assembled into machine code.

In this chapter we are going to re-create a simple version of Snake from scratch, written in a modern assembly language to run on the ARMlite simulation. With very little modification the same code could run on, say, a Raspberry Pi with a memory-addressable display device.
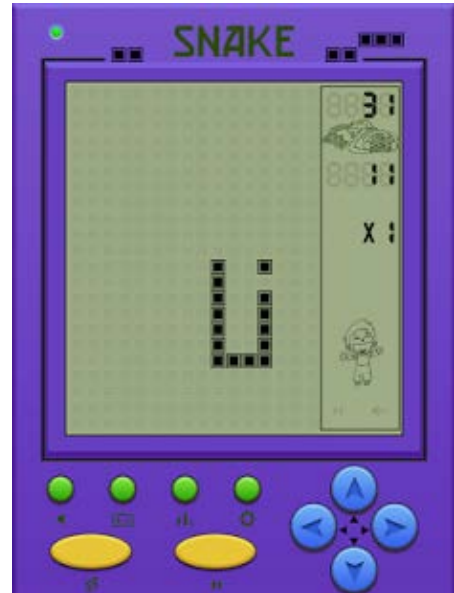
Tip

As we are going to be a developing a more substantial application this time, incrementally, you might find it easier to use a freestanding code editor - which can be as simple as Notepad, or as sophisticated as an IDE. With each iteration, make the edits in the code editor and save the new version of the file (as simple text) - then use the **Load** button on ARMlite to load the latest version of the code.

Create a moving snake

Let's get started by writing a simple loop that will move the head of the snake one pixel to the right each iteration. In this version, the snake will grow continually, which is not what we want for the final game, but we'll fix that in due course:

```
//Define registers
    // R0-2 reserved for temporary uses
    // Global variables:
    // R3 Tail position
    // R4 Head position
    // Constants:
    MOV R10,#.PixelScreen
    MOV R11, #.green //Colour of snake

//Initialise game
    MOV R3, #1084  //Initialise tail and ...
    MOV R4, #1088  //Head next to it (4 bytes = 1 word = 1 pixel)
    STR R11,[R10+R3] //Draw 2-segment snake
    STR R11,[R10+R4]

update:
    ADD R4,R4,#4   //Increment head position
    STR R11,[R10+R4] //Draw new head
    B update
```

Exercise 39

Load the code, and *before running it*, **Single Step** through a few iterations of the update loop, so that you understand what all the code is doing.

Then run the program (using the **Play** button).

What problems do you immediately notice?

Save your program

Model answer 39

Problems:
- The snake wraps around the edge of the screen
- Eventually, the snake moves off the bottom edge of the screen and this causes a 'Bad I/O address…' error.
- The snake moves too fast for a playable game!

Control the frequency of updates

Of the problems with this first iteration, the most immediate one is the speed. The best way to control the speed is to use the clock interrupts. Make the following modifications, to the code:

```
//Define registers
    // R0-2 reserved for temporary uses
    // Global variables:
    // R3 Tail position
    // R4 Head position
    // Constants:
    MOV R10,#.PixelScreen
    MOV R11, #.green //Colour of snake

//Set up interrupts - but don't enable yet
    MOV R0, #update
    STR R0, .ClockISR
    MOV R0, #0x50
    STR R0,.ClockInterruptFrequency

//Initialise game
    MOV R3, #1084   //Initialise tail and ...
    MOV R4, #1088   //Head next to it (4 bytes = 1 word = 1 pixel)
    STR R11,[R10+R3] //Draw 2-segment snake
    STR R11,[R10+R4]
    MOV R0, #1
    STR R0, .InterruptRegister //Now we are ready to handle interrupts

mainLoop: b mainLoop //Just keeps the processor running, pending interrupts

//Interrupt driven
update:
    ADD R4,R4,#4    //Increment head position
    STR R11,[R10+R4] //Draw new head
    B update
    RFE
```

Notes:

- We set up the clock interrupts at the start (by convention), but interrupts *overall* are not enabled until after the game has been initialised - otherwise the `update` routine might be called before there is any snake to move.
- The `ClockInterruptFrequency` is defined in milliseconds. Here we have set it up to fire every `50` milliseconds.
- We need a `mainLoop`, even though it is empty, so that the processor continues running.
- Because `update` is now interrupt driven, it now ends with `RFE` *instead of* `B update`.
- `update` does not save any registers on the stack because the only registers it uses are global variables.

Exercise 40

Make the changes shown and run the program.

Try altering the `ClockInterruptFrequency`, up or down from 50 millisecond intervals.

How might it be possible to allow the user to select the speed, corresponding to different levels of difficult (description only required - no need to code it now)?

Save your program

Model answer 40

At the start of the game write a message to the console asking to select the speed. Input a number (e.g. 1 – 9) and transform (scale) this to a suitable millisecond value, then use this value to replace the constant currently specified for the clock frequency.

Change direction with the W,A,S,D keys

Next, we will permit the user to alter the direction of the snake's movement, using the standard **W**,**A**,**S**, and **D** keys for Up, Left, Down, and Right. We could check for keypresses within our update loop, but a more elegant solution is to use keyboard interrupts. Here's the interrupt routine:

```
//Called by keyboard interrupt
//If valid key (W,A,S,D) has been pressed, transfer this to R9
keyPress: PUSH {R0}
   LDR R0,.LastKey //Read the last key pressed (but don't wait for one)
   CMP R0,#87 //W key
   BEQ updateLastKey
   CMP R0,#65 //A key
   BEQ updateLastKey
   CMP R0,#83 //S key
   BEQ updateLastKey
   CMP R0,#68 //D key
   BEQ updateLastKey
   B .+2 //If not a valid new key don't change last key
updateLastKey:
   MOV R7, R0
   POP {R0}
   RFE
```

Notes:

- This new routine can be placed after the existing **update** routine.
- **B .+2** means 'Branch 2 instructions forward' (i.e. skip over the next instruction). When making *small, local* jumps, this syntax can be less cluttered than defining an additional label.
- In this routine, we are saving the value of **R0** on the stack because it might have been in use for another purpose when the interrupt was called. **R7**, however, is globally defined.

And here are the changes needed to the start of the program - one new global variable definition, and an additional interrupt set up:

```
//Define registers
     ...
// R7 ASCII value of last key pressed
     ...

//Set up interrupts - but don't enable yet
   MOV R0, #update
   STR R0, .ClockISR
   MOV R0,#0x19
   STR R0,.ClockInterruptFrequency
   MOV R0, #keyPress
   STR R0, .KeyboardISR
   MOV R0, #1
   STR R0, .KeyboardMask
```

Now we must make use of the last key press (in R9) to control the direction of movement in the **update** routine:

```
update:
        //Switch on direction of last key
    CMP R7,#87 //W key
    BEQ up
    CMP R7,#65 //A key
    BEQ left
    CMP R7,#83 //S key
    BEQ down
    // By default the snake will move right
right:ADD R4,R4,#4   //+4 (bytes) moves right one pixel
    B reDraw
down: ADD R4,R4,#256 //+64*4 moves down one row
    B reDraw
up:  SUB R4,R4,#256 //-64*4 moves up one row
    B reDraw
left: SUB R4,R4,#4  //-4 moves left one pixel
reDraw:
    ADD R4,R4, #4 //Increment head position DELETED
    STR R11,[R10+R4] //Draw new head
    RFE
```

Exercise 41

Add the new code and make all the changes shown above, in the correct places then run the program. Make the snake change in all four directions and then pause the program, capturing a screenshot showing the pixel screen.

Save your program

Model answer 41
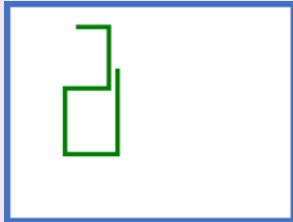
Hitting an edge is 'Game Over'

In the real game, if the snake hits any of the four edges, the game is over. The top and bottom edges are straight forward: we can test whether the value of R4 is less than 0, or greater than the maximum pixel address: 12284 (= 64 x 48 x 4, the 4 because a pixel is 4 bytes). For the left and right edges, we need to look at the least-significant 8 bits of the pixel number only, by ANDing the number with #255. If the snake is moving right, we need to look out for the least-significant 8 bits evaluating to 0. If the snake is moving left, we need to look out for the least-significant 8 bits evaluating to #252 (because we are deducting 4 each time, not 1). Here are the modifications required to the core of the update routine:

```
right:ADD R4,R4,#4 //+4 (bytes) moves right one pixel
    AND R0,R4,#255
    CMP R0,#0
    BEQ gameOver
    B reDraw
down: ADD R4,R4,#256 //+64*4 moves down one row
    MOV R0, #12284 // One past the last valid pixel
    CMP R4,R0
    BGT gameOver
    B reDraw
up:   SUB R4,R4,#256 //-64*4 moves up one row
    CMP R4,#0
    BLT gameOver
    B reDraw
left: SUB R4,R4,#4 //-4 moves left one pixel
    AND R0,r4,#255
    CMP R0,#252
    BEQ gameOver
```

> **Teacher note**
> Some students might wonder why, for the 'down' test we did not just write CMP R4, #12284, and thereby save one instruction. This is because the immediate value of #12284 will not fit within the available bits for the second operand of a CMP instruction (try it and you will get an assembly error), whereas it will fit for a MOV.
>
> This might seem an odd feature of ARMlite, but it does accurately reflect the specification of the ARM 32-bit processor. The rules are quite complex – because the immediate value is held as a value plus a scale (*somewhat* like a floating point-representation) – but they are specified in the ARMlite Programming Manual.

We will also need to define the gameOver label, which may be placed after the current end of the code:

```
gameOver: MOV R0, #over
    STR R0,.WriteString
    HALT //To stop program execution running into data area
over: .ASCIZ " Game Over!\n"
```

Exercise 42

Make the changes shown above and run the program four times, validating that the game is over if the snake's head touches any of the four edges, but that you can turn *just before the edge.*

(In programming we talk about the need to test 'edge' or 'boundary' conditions - in this case the condition corresponds to a physical edge or boundary!)

Paste a partial screenshot showing a turn having been made just before the edge.

Save your program

Model answer 42

The snake may not cross itself

We can also add the rule that the game is over if the snake's head passes over its body. The simplest way to do this is to test whether the pixel the head is *about to move onto* is already green:

```
reDraw:
    //First check if the snake would be crossing itself
    LDR R0,[R10+R4] // read, from screen, contents of next pixel
    CMP R0,R11        //If it is snake colour...
    BEQ gameOver
    STR R1,[R10+R4] //Draw new head
    RFE
```

Exercise 43

Make the changes shown above and run the program. Capture a partial screenshot showing that the game is over if the snake crosses itself (include both the pixel screen and the console message).

Run the game and try 'reversing direction' (e.g. hit the **A** key while the snake is moving right). What happens?

Save your program

Model answer 43

```
Game Over!
Program HALTED. STOP, LOAD or EDIT
```
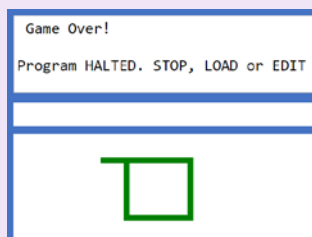
If you accidentally reverse direction, this is an illegal move and it is Game Over!

Create an apple in a random position

Now let's add an apple, in a random location on the screen. Then, whenever the snake 'eats' (passes over) the apple, add one to the player's score and generate an apple in a new random position. We need to generate a random number in the range 0 – 12280, and it must be divisible by 4. By default, ARMlite's `.Random` capability generates a 32-bit random number. We can `AND` the result with a 'bit mask' of `0b00000000000000000011111111111100` (`0x3ffc`) to cut it down to the range 0 – 16380, and then we will just have to test to see if the result is within the required range and if necessary 'throw [the dice] again'.

First, we'll define a new constant for the apple colour, and a new global variable to hold the count of apples eaten:

```
//Define registers
    ...
    MOV R8, #0 //Score of apples eaten
    // Constants:
    ...
    MOV R12, #.red //Colour of apple
```

The following subroutine will generate a random pixel number in the required range and paint the apple on that pixel. It also checks to the see that we are not placing the apple anywhere on the snake's body i.e. on a pixel that is already green:

```
//Generates apple in random valid location
createApple: push {R0,R1, LR}
newRandom: LDR R1,.Random // gets a random 32 bit pattern
    MOV R0, #0x3ffc // Limit random to 14 bits
    AND R1,R1,R0
    MOV R0, #12284 //Max pixel number
    CMP R1,R0
    BGT newRandom //'Throw again'
    LDR R0, [R10+R1] //Get intended pixel
    CMP R0,R11 //Compare pixel to snake colour
    BEQ newRandom
    STR R12, [R10+R1] //Draw apple
    POP {R0,R1,LR}
    RET
```

Notes:

- This new routine can be placed anywhere that doesn't overlap with an existing routine. The author chose to place it just before `gameOver`, and it is suggested that you do the same for consistency with the book.
- Because this is a subroutine that will be called using `BL` (not an interrupt routine) it ends with `RET`
- The `PUSH` and `POP` are not strictly necessary for this code (because `R0` has only local uses, and we don't have any nested subroutine calls that could result in losing the value of `LR`), but we have included these because it is a good, safe practice.

And we can call `createApple` within the game initialisation sequence (but still before enabling interrupts):

```
//Initialise game
    ...
    BL createApple
    STR R0, .InterruptRegister //Now we are ready to handle interrupts
```

And then at the end of the **update** routine, within the **reDraw** section we can test whether the apple has been eaten, and if so, update the score and create an apple in a new position:

```
reDraw:
    //First check if the snake would be crossing itself
    LDR R0,[R10+R4] // read, from screen, contents of next pixel
    CMP R0,R11          //If it is snake colour...
    BEQ gameOver
    CMP R0, R12    //Check if pixel is apple colour
    BNE .+3      //Skip to RFE
    ADD R8,R8,#1 //Increment score
    BL createApple
    STR R11,[R10+R4] //Draw new head
    RFE
```

We can also write out the score when the game is over by modifying the **gameOver** sequence:

```
gameOver: MOV R0, #over
    STR R0,.WriteString
    MOV R0, #score
    STR R0,.WriteString
    STR R8, .WriteSignedNum
    HALT //To stop program execution running into data area
over: .ASCIZ " Game Over!\n"
score: .ASCIZ "Your score: "
```

Exercise 44

Make all the changes shown since the last exercise. Pause the game and capture a partial screenshot showing the snake and an apple.

Play again, eating at least two apples then capture a screenshot showing the final pixel screen and your score.

Save your program

Model answer 44

Here is the complete code at this point:

```
//Define registers
// R0-2 reserved for temporary uses
// Global variables:
// R3 Tail position
// R4 Head position
// R7 ASCII value of last key pressed
      MOV R8, #0 //Score of apples eaten
// Constants:
      MOV R10,#.PixelScreen
      MOV R11, #.green //Colour of snake
      MOV R12, #.red //Colour of apple
//Set up interrupts - but don't enable yet
      MOV R0, #update
      STR R0, .ClockISR
      MOV R0, #0x50
      STR R0,.ClockInterruptFrequency
      MOV R0, #keyPress
      STR R0, .KeyboardISR
      MOV R0, #1
      STR R0, .KeyboardMask
//Initialise game
      MOV R3, #1084 //Initialise tail and ...
      MOV R4, #1088 //Head next to it (4 bytes = 1 word = 1 pixel)
      STR R11,[R10+R3] //Draw 2-segment snake
      STR R11,[R10+R4]
      MOV R0, #1
      BL createApple
      STR R0, .InterruptRegister //Now we are ready to handle interrupts
mainLoop: b mainLoop //Just keeps the processor running, pending interrupts
update:
//Switch on direction of last key
      CMP R7,#87 //W key
      BEQ up
      CMP R7,#65 //A key
      BEQ left
      CMP R7,#83 //S key
      BEQ down
// By default the snake will move right
right:ADD R4,R4,#4 //+4 (bytes) moves right one pixel
      AND R0,R4,#255
      CMP R0,#0
      BEQ gameOver
      B reDraw
down: ADD R4,R4,#256 //+64*4 moves down one row
      MOV R0, #12284 // One past the last valid pixel
      CMP R4,R0
      BGT gameOver
      B reDraw
up:   SUB R4,R4,#256 //-64*4 moves up one row
      CMP R4,#0
      BLT gameOver
      B reDraw
left: SUB R4,R4,#4 //-4 moves left one pixel
      AND R0,r4,#255
      CMP R0,#252
      BEQ gameOver
reDraw:
//First check if the snake would be crossing itself
      LDR R0,[R10+R4] // read, from screen, contents of next pixel
      CMP R0,R11  //If it is snake colour...
      BEQ gameOver
```

```
        CMP R0, R12 //Check if pixel is apple colour
        BNE .+3 //Skip to RFE
        ADD R8,R8,#1 //Increment score
        BL createApple
        STR R11,[R10+R4] //Draw new head
        RFE
//Called by keyboard interrupt
//If valid key (W,A,S,D) has been pressed, transfer this to R9
keyPress: PUSH {R0}
        LDR R0,.LastKey //Read the last key pressed (but don't wait for one)
        CMP R0,#87 //W key
        BEQ updateLastKey
        CMP R0,#65 //A key
        BEQ updateLastKey
        CMP R0,#83 //S key
        BEQ updateLastKey
        CMP R0,#68 //D key
        BEQ updateLastKey
        B .+2 //If not a valid new key don't change last key
updateLastKey:
        MOV R7, R0
        POP {R0}
        RFE
//Generates apple in random valid location
createApple: push {R0,R1, LR}
newRandom: LDR R1,.Random // gets a random 32 bit pattern
        MOV R0, #0x3ffc // Limit random to 14 bits
        AND R1,R1,R0
        MOV R0, #12284 //Max pixel number
        CMP R1,R0
        BGT newRandom //'Throw again'
        LDR R0, [R10+R1] //Get intended pixel
        CMP R0,R11 //Compare pixel to snake colour
        BEQ newRandom
        STR R12, [R10+R1] //Draw apple
        POP {R0,R1,LR}
        RET
gameOver: MOV R0, #over
        STR R0,.WriteString
        MOV R0, #score
        STR R0,.WriteString
        STR R8, .WriteSignedNum
        HALT //To stop program execution running into data area
over: .ASCIZ " Game Over!\n"
score: .ASCIZ "Your score: "
```

Teacher note

At this point we have the beginnings of an interesting game, and you might prefer to stop here. The following section makes the game more like the original snake. Continuing to the next section carries the further advantage that it makes use of a 'queue' data structure - first a linear queue, and then a circular queue.

Making the snake grow only when an apple is eaten

Our current game it is not like the *real* snake game: where the snake starts at two segments, but grows in length only with each apple eaten. Implementing this poses some significant challenges - because we need to keep track not just of the head and the tail of the snake, but also of each body segment - so that the tail follows the same path as the head.

If we store each memory address that corresponds to a snake segment in a *queue* data structure, then as we move the head we can *enqueue* its new address (i.e. add it to the *end* of the queue), and as we move the tail forwards we can dequeue its position. (Perhaps counter-intuitively, this means that the tail is technically at the *start* of the queue, and the head is at the *end*).

A queue consists of a collection of successive words in memory. We can define this with a label at the very end of our code:

```
over: .ASCIZ " Game Over!\n"
score: .ASCIZ "Your score: "
.ALIGN 256
body: //The 'queue' of body segments from here onwards
```

Notes:
- `.ALIGN` is needed to ensure that the queue starts on a word boundary (it might not otherwise, because of the previous data being a list individual bytes). `.ALIGN 4` would achieve what we need, but `.ALIGN 256` starts the queue on the next page of memory, which is convenient for viewing.

We also need to define two more registers, to act as pointers to the front and back of the queue:

```
//Define registers
   ...
   // R5 Front of queue (address of snake's tail)
   // R6 Back of queue (address of snake's head)
```

And then initialise those pointers:

```
//Initialise game
   MOV R3, #1084 //Initialise tail and ...
   MOV R4, #1088 //Head next to it (4 bytes = 1 word = 1 pixel)
   STR R11,[R10+R3] //Draw 2-segment snake
   STR R11,[R10+R4]
   MOV R5, #body //Pointer front of queue, initialised to first data loc
   ADD R6,R5,#4  //Pointer to head address in body data (1 after tail)
   STR R3, [R5]  //R3 points to the tail address
   STR R4, [R6]  //R4 points to the head address
   MOV R0, #1
   BL createApple
   STR R0, .InterruptRegister //Now we are ready to handle interrupts
```

Notes:

- Here we are making effective use of indirect addressing. In **R5**, for example, we are <u>not</u> holding the value (`#.green`) of the tail of the snake; rather, we are holding the address in memory of the pixel representing the tail of the snake.

Now within the **reDraw** code, as well as drawing the green head in its new location we need to add that new location to the back of the queue. Then, unless the snake has eaten an apple, we need to paint the current tail pixel white again, and *dequeue* that pixel reference. (If the snake has eaten an apple, we skip over the code that moves the tail, with the result that the snake will grow in length by one pixel).

```
reDraw:
    //First check if the snake would be crossing itself
    LDR R0,[R10+R4] // read, from screen, contents of next pixel
    CMP R0,R11          //If it is snake colour...
    BEQ gameOver
    ADD R6,R6,#4  //Increment the back of queue pointer (by 1 word = 4 bytes)
    STR R4, [R6]  //Store the new head pixel number at the rear of the queue
    CMP R0, R12  //Check if pixel is apple colour
    BEQ eat
    MOV R0, #.white
    STR R0, [R10+R3] //Paint the current tail pixel white
    ADD R5,R5,#4  //Increment front of queue pointer
    LDR R3,[R5]  //Retrieve pixel number for the new tail
    BNE .+3      //Skip to RFE
    B .+3        //Skip to RFE
eat: ADD R8,R8, #1 //Increment score
    BL createApple
    STR R11,[R10+R4] //Draw new head
    RFE
```

Exercise 45

Make all the highlighted changes since the last exercise. You should now find that the snake remains at two segments until you eat an apple.

Grow the snake by *at least two segments* then paste a partial screenshot showing the ending screen and your score.

When stopped, go to memory page **002** and take a partial screenshot of that page of memory.

Describe in your own words what is being held in those memory words.

Save your program

Model answer 45

```
Game Over!
Your score: 3
Program HALTED. STOP, LOAD or EDIT
```

Memory				
002	**0x0**	**0x4**	**0x8**	**0xc**
0x0020	0x0000043c	0x00000440	0x00000444	0x00000448
0x0021	0x0000044c	0x00000450	0x00000454	0x00000458
0x0022	0x0000045c	0x00000460	0x00000464	0x00000468
0x0023	0x0000046c	0x00000470	0x00000474	0x00000478
0x0024	0x0000047c	0x00000480	0x00000484	0x00000488
0x0025	0x00000588	0x00000688	0x00000788	0x00000888
0x0026	0x00000988	0x00000a88	0x00000b88	0x00000c88
0x0027	0x00000d88	0x00000e88	0x00000f88	0x00001088
0x0028	0x00001188	0x00001288	0x00001388	0x00001488
0x0029	0x00001588	0x00001688	0x00001788	0x00001888
0x002a	0x00001988	0x00001a88	0x00001b88	0x00001c88
0x002b	0x00001c84	0x00001c80	0x00001c7c	0x00001c78
0x002c	0x00001c74	0x00001c70	0x00001c6c	0x00001c68
0x002d	0x00001c64	0x00001c60	0x00001c5c	0x00001c58
0x002e	0x00001c54	0x00001c50	0x00001c4c	0x00001c48
0x002f	0x00001c44	0x00001c40	0x00001c3c	0x00001c38
0x0030	0x00001c34	0x00001c30	0x00001c2c	0x00001c28
0x0031	0x00001c24	0x00001c20	0x00001c1c	0x00001c18
0x0032	0x00001c14	0x00001c10	0x00001c0c	0x00001b0c
0x0033	0x00001a0c	0x0000190c	0x0000180c	0x0000170c
0x0034	0x0000160c	0x0000150c	0x0000140c	0x0000130c
0x0035	0x0000120c	0x0000110c	0x0000100c	0x00000f0c
0x0036	0x00000e0c	0x00000d0c	0x00000c0c	0x00000c08
0x0037	0x00000c84	0x00000c00	0x00000d00	0x00000e00
0x0038	0x00000f00	0x00001000	0x00001100	0x00001200
0x0039	0x00001300	0x00001400	0x00001500	0x00001600
0x003a	0x00001700	0x00001800	0x00001900	0x00001a00
0x003b	0x00001b00	0x00001c00	0x00001d00	0x00001e00
0x003c	0x00001f00	0x00002000	0x00002100	0x00002200
0x003d	0x00002300	0x00002400	0x00002404	0x00002408
0x003e	0x0000240c	0x00002410	0x00002414	0x00002418
0x003f	0x0000241c	0x00002420	0x00002424	0x00002428

This memory is a record of the history of the pixel indexes of the tail. The active queue in this case has only 5 pixel addresses in it (having eaten 3 apples), but the queue keeps advancing through memory, leaving a trail of 'dead' data behind it.

Although our program is working correctly from the point of view of the user, it is not a good implementation.

The problem is that with each movement of the snake, the active part of the queue (the memory locations between the front and back of the queue) is continually advancing through memory, leaving 'dead' data in its trail. Eventually the queue is going to hit the end of memory even if the actual length off the snake remains quite short.

ARMlite has a 1Mbyte or memory, so even if the snake advances by 10 pixels a second, so you would need to keep playing the game, deliberately missing the apple, for several hours before you'd reach the end of memory.

But a program that behaves in this way is sometimes described as having a 'memory leak' and no self-respecting programmer would leave their code doing this.

Implementing a circular queue

The answer, as you might know if you have already studied data structures, is to fix the maximum length of queue and implement it as a circular queue - wrapping the pointers around the end.

How long should we make the queue? Well, given that there are only 3072-pixel locations, even a player with *perfect reactions and planning* cannot make a snake longer than that, so let's reserve 3072 words of memory for the queue, with this change:

```
.ALIGN 256
body: .BLOCK 3072 //For the 'queue' of body segments
limit: //1 past end of queue data
```

Then within reDraw:

```
reDraw:
    //First check if the snake would be crossing itself
    LDR R0,[R10+R4] // read, from screen, contents of next pixel
    CMP R0,R11 //If it is snake colour...
    BEQ gameOver
    ADD R6,R6,#4 //Increment the back of queue pointer (by 1 word = 4 bytes)
    CMP R6,#limit //Check pointer is still within end of queue data area
    BLT .+2
    MOV R6, #body //If not loop pointer back to start of body data
    STR R4, [R6] //Store the new head pixel number at the rear of the queue
    CMP R0, R12 //Check if pixel is apple colour
    BEQ eat
    MOV R0, #.white
    STR R0, [R10+R3] //Paint the current tail pixel white
    ADD R5,R5,#4 //Increment front of queue pointer
    CMP R5,#limit //Check pointer is still within end of queue data area
    BLT .+2
    MOV R5, #body //If not loop pointer back to start of body data
    LDR R3,[R5] //Retrieve pixel number for the new tail
    B .+3 //Skip to RFE
eat: ADD R8,R8, #1 //Increment score
    BL createApple
    STR R11,[R10+R4] //Draw new head
    RFE
```

Exercise 46

Make the changes, and , when you've checked that it submits OK, save your program.

Then switch to the 'Player mode':

https://peterhigginson.co.uk/ARMlite/?profile=player

and enjoy playing the game.

Model answer 46   Complete code

```
// Define registers
// R0-2 reserved for temporary uses
// Global variables:
// R3 Tail position
// R4 Head position
// R5 Front of queue (address of snake's tail)
// R6 Back of queue (address of snake's head)
// R7 ASCII value of last key pressed
```

```
        MOV R8, #0 //Score of apples eaten
// Constants:
        MOV R10,#.PixelScreen
        MOV R11, #.green //Colour of snake
        MOV R12, #.red //Colour of apple
//Set up interrupts - but don't enable yet
        MOV R0, #update
        STR R0, .ClockISR
        MOV R0, #0x50
        STR R0,.ClockInterruptFrequency
        MOV R0, #keyPress
        STR R0, .KeyboardISR
        MOV R0, #1
        STR R0, .KeyboardMask
//Initialise game
        MOV R3, #1084 //Initialise tail and ...
        MOV R4, #1088 //Head next to it (4 bytes = 1 word = 1 pixel)
        STR R11,[R10+R3] //Draw 2-segment snake
        STR R11,[R10+R4]
        MOV R5, #body //Pointer front of queue, initialised to first data loc
        ADD R6,R5,#4 //Pointer to head address in body data (1 after tail)
        STR R3, [R5] //R3 points to the tail address
        STR R4, [R6] //R4 points to the head address
        MOV R0, #1
        BL createApple
        STR R0, .InterruptRegister //Now we are ready to handle interrupts
mainLoop: b mainLoop //Just keeps the processor running, pending interrupts
update:
//Switch on direction of last key
        CMP R7,#87 //W key
        BEQ up
        CMP R7,#65 //A key
        BEQ left
        CMP R7,#83 //S key
        BEQ down
// By default the snake will move right
right:ADD R4,R4,#4 //+4 (bytes) moves right one pixel
        AND R0,R4,#255
        CMP R0,#0
        BEQ gameOver
        B reDraw
down: ADD R4,R4,#256 //+64*4 moves down one row
        MOV R0, #12284 // One past the last valid pixel
        CMP R4,R0
        BGT gameOver
        B reDraw
up:   SUB R4,R4,#256 //-64*4 moves up one row
        CMP R4,#0
        BLT gameOver
        B reDraw
left: SUB R4,R4,#4 //-4 moves left one pixel
        AND R0,r4,#255
        CMP R0,#252
        BEQ gameOver
reDraw:
//First check if the snake would be crossing itself
        LDR R0,[R10+R4] // read, from screen, contents of next pixel
        CMP R0,R11 //If it is snake colour...
        BEQ gameOver
        ADD R6,R6,#4 //Increment the back of queue pointer (by 1 word = 4 bytes)
        CMP R6,#limit //Check pointer is still within end of queue data area
```

```
        BLT .+2
        MOV R6, #body //If not loop pointer back to start of body data
        STR R4, [R6] //Store the new head pixel number at the rear of the queue
        CMP R0, R12 //Check if pixel is apple colour
        BEQ eat
        MOV R0, #.white
        STR R0, [R10+R3] //Paint the current tail pixel white
        ADD R5,R5,#4 //Increment front of queue pointer
        CMP R5,#limit //Check pointer is still within end of queue data area
        BLT .+2
        MOV R5, #body //If not loop pointer back to start of body data
        LDR R3,[R5] //Retrieve pixel number for the new tail
        B .+3 //Skip to RFE
eat:    ADD R8,R8, #1 //Increment score
        BL createApple
        STR R11,[R10+R4] //Draw new head
        RFE
//Called by keyboard interrupt
//If valid key (W,A,S,D) has been pressed, transfer this to R9
keyPress: PUSH {R0}
        LDR R0,.LastKey //Read the last key pressed (but don't wait for one)
        CMP R0,#87 //W key
        BEQ updateLastKey
        CMP R0,#65 //A key
        BEQ updateLastKey
        CMP R0,#83 //S key
        BEQ updateLastKey
        CMP R0,#68 //D key
        BEQ updateLastKey
        B .+2 //If not a valid new key don't change last key
updateLastKey:
        MOV R7, R0
        POP {R0}
        RFE
//Generates apple in random valid location
createApple: push {R0,R1, LR}
newRandom: LDR R1,.Random // gets a random 32 bit pattern
        MOV R0, #0x3ffc // Limit random to 14 bits
        AND R1,R1,R0
        MOV R0, #12284 //Max pixel number
        CMP R1,R0
        BGT newRandom //'Throw again'
        LDR R0, [R10+R1] //Get intended pixel
        CMP R0,R11 //Compare pixel to snake colour
        BEQ newRandom
        STR R12, [R10+R1] //Draw apple
        POP {R0,R1,LR}
        RET
gameOver: MOV R0, #over
        STR R0,.WriteString
        MOV R0, #score
        STR R0,.WriteString
        STR R8, .WriteSignedNum
        HALT //To stop program execution running into data area
over: .ASCIZ " Game Over!\n"
score: .ASCIZ "Your score: "
        .ALIGN 256
body: .BLOCK 3072 //For the 'queue' of body segments
limit: //1 past end of queue data
```

Possible game enhancements

If you have time you might consider improving the game. Here are some suggestions, though you might think of your own, too:

- At the end of the game, give the user the option to play another game without having to stop and re-start the program.
- Make the start position of the snake, and its initial direction of movement, random.
- Give the player the option, at the start of the game, to change the speed, by inputting a number and translating this to the appropriate range for the `ClockInterruptFrequency`.
- Speed up the clock frequency as more apples are eaten.
- Create multiple apples.
- Create one or more 'hazards', at random, that the snake must avoid touching.
- Ignore accidental reverses rather than dying instantly.

Appendices

Appendix I: AQA vs. ARMlite

ARMlite simulates a cut-down version of a 32-bit ARM processor. The AQA instruction set (as issued with each of the A-level exam questions to date) is also *based on* the ARM instruction set, though cut down even further, and not followed as rigorously. AQA does not specify the word size, and specific exam questions have sometimes suggested that a register is 16, or even 8 bits wide. In practice, this lack of definition is not a big issue, since exam questions tend to deal with small data values - in which case it would make no difference if a register was 8,16 or 32 bits wide.

The AQA instruction set is limited to these instructions:

`LDR`, `STR`, `ADD`, `SUB`, `MOV`, `CMP`, `B`, `BEQ`, `BNE`, `BGT`, `BLT`, `AND`, `ORR`, `EOR`, `MVN`, `LSL`, `LDR`, and `HALT`

Chapters 1 – 4 of this book use only those instructions, and in a manner consistent with the AQA instruction set, which, to date, is included with each exam question on assembly language. Students should be aware that in an exam, they may use only those instructions and only in the ways spelled out on the instruction sheet included in the exam.

AQA specifies that `LDR` and `STR` take, as their second operand, a `<memory ref>`. Surprisingly, AQA does not specify exactly what form(s) this `<memory ref>` may take. However, as of the time of writing this book, the following may be *inferred* from past questions:

- `<memory ref>` is a *direct* address. AQA has never, to date, made any use of, or reference to, indexed or indirect addressing modes. The AQA exam questions to date have involved simple programming exercises where there is no need for indexed/indirect addressing, and where the latter would confer no advantage. Nonetheless, students that have learned these approaches should be clear that they should not use indexed/indirect modes in an exam.
- `<memory ref>` is a *decimal* number. AQA has not, to date, specified any direct *address* in hexadecimal or binary. Nor has AQA, to date, made use of a *label* as a memory address (labels have been used, to date, only to label instructions – not memory addresses).
- Although AQA has not, to date, specified it, we may infer from past questions, that `<memory ref>` is a 'word address'. ARMlite, in common with the ARM and most other modern processors, uses 'byte addressing'.

The last point is the most important one to understand. Assuming AQA does mean *word addressing* (and assuming *decimal* addresses), then the following AQA code:

```
LDR R0, 100
LDR R1, 101
```

would result in registers `R0` and `R1` being loaded with two different values from two successive words in memory. With ARMlite, and on a real ARM processor, the second instruction would give an assembly error ('`Unaligned address …`') because the second address would be only 1 *byte* after the first, and legitimate word addresses must be divisible by 4.

Appendix II: Useful links

ARMlite Programming Reference Manual
The full ARMlite Programming Reference Manual, and further technical documentation, may be found downloaded from here:

https://peterhigginson.co.uk/ARMlite/doc.php

ASCII table
http://www.asciitable.com/

Online convertor between decimal, hex, binary
https://www.rapidtables.com/convert/number

Online convertor for two's complement
https://www.exploringbinary.com/twos-complement-converter/

Appendix III: Versioning

This book adopts 'semantic versioning' with the following meanings:

- A third-level version change (e.g. 1.0.**n**) indicates *minor* edits or corrections to text, layout, or formatting.
- A second level version change (e.g. 1.**n**.0) indicates correction to consequential error, in text or code.
- A first level version change (e.g. **n**.0.0) indicates new material and/or or re-structuring.

V1.0.0
Released 4th January 2020. Works with ARMlite version 1

Assembly language means programming 'close to the metal': constructing a program from very simple instructions that can be executed directly by the processor.

Today, most software is written in high level programming languages. However, learning assembly language will give you a deeper understanding not only of how a processor works, but of programming in general.

The book guides the reader through a series of small projects, applying new techniques to write programs that are useful and interesting. The final chapter demonstrates the use of the all the techniques learned to create a complete implementation of the well-known 'Snake' game.

All examples in the book are designed to run on ARMlite, an online simulation of a simple computer based on a cut-down version of an ARM 32 processor.

Peter Higginson, who implemented ARMlite for this book, spent 40 years writing systems software in the computing and communications industry, and working in computer science education. In 1973 he connected the London Host to ARPANET - the first host outside the USA on what is now the Internet.

Richard Pawson worked in the computing industry for 40 years before teaching A-level Computer Science at Stowe School. He has a BSc in Engineering, a PhD in Computer Science, and a PGC in Intellectual Property Law. He now splits his time between managing a large open source product, writing books, and guest teaching in various schools.