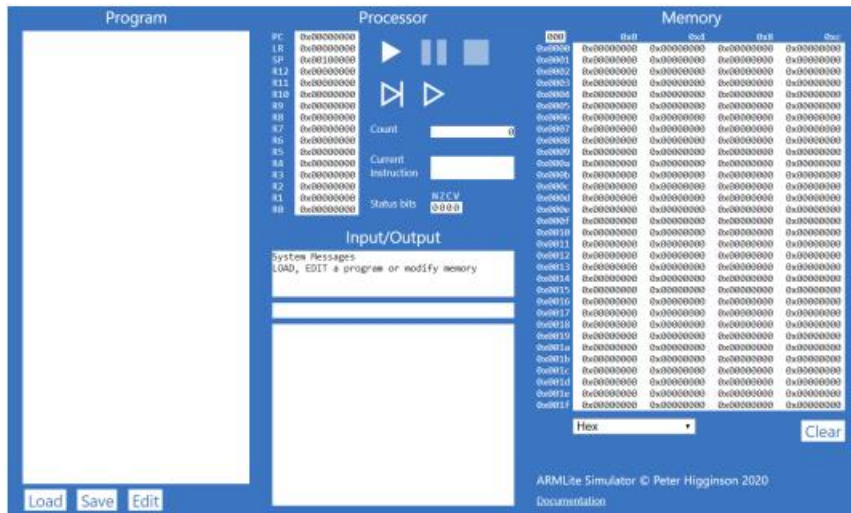


# Week 7 Lab - Part 7.1

## Memory in ARMLite

Access the ARMLite simulator via: <https://peterhigginson.co.uk/ARMLite/> [Links to an external site.](#) (Chrome is the recommended browser, however most modern browsers should work, though IE11 apparently does not).

You should see a GUI that looks like this:



On the right side of the ARMLite simulator is a grid of memory addresses, with each block of 8 hex digits (all initialised to 0) representing a 32 bit word.

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

### 7.1.1 What value is displayed? Why?

Click on another memory word, enter 0x101

### 7.1.2 What value is displayed, and why?

On another memory word, enter 0b101

### 7.1.3 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?

Below the grid of memory words is a drop down menu that looks like this:



This drop-down selector allows you to change the base in which data is displayed. Changing the base does not change the underlying data value, only the base number system in which the value is displayed.

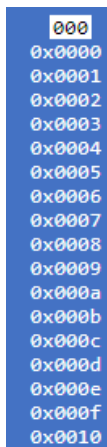
Change this to Decimal (unsigned) and note the change that has occurred to the three memory words you previously entered.

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

**7.1.4: 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)? Should it ?**

## Memory Addressing

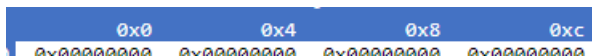
Each word of memory has a unique 'address', expressed as a five-digit hex number. On the ARMLite Simulator, memory words are laid out in four columns, however this is only for visual convenience. Notice that each row starts with a four digit hex value (in white) that looks like this:



000  
0x0000  
0x0001  
0x0002  
0x0003  
0x0004  
0x0005  
0x0006  
0x0007  
0x0008  
0x0009  
0x000a  
0x000b  
0x000c  
0x000d  
0x000e  
0x000f  
0x0010

These values represent the first four digits of the address for all memory words in that row.

Now look along the top of each column in the Memory grid and note these values:



0x0 0x4 0x8 0xc  
0x00000000 0x00000004 0x00000008 0x0000000c

These single digit hex values represent offsets from the row-header address. Therefore, the full address of any memory word is obtained by appending the column header digit to the 4 digit row-header. For example, the address of the top-left word on this screen is 0x00000, and the bottom-right is 0x001fc.

**7.2.1 Notice these column header memory address offsets go up in multiples of 0x4. Why is this ?**

*Hint: remember how many bits are in each memory word !*

ARMLite, in common with most modern processors uses 'byte addressing' for memory. When storing or retrieving a word you generally specify only the address of the first of the bytes making up each word - we'll come back to this when we start dealing with storing and loading values to and from memory.

## Editing and Submitting Assembly Code

On the left side of the ARMLite simulator is the Program window. This is where you can load and/or edit assembly code to be executed by the simulator.

Click the Edit button below the Program window, and then copy and Paste the following ARMLite assembly program into the window:

```

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

```

Once copied, click the **Submit** button. This invokes the assembler and all going well, should not give any errors. If it does then you may need to check you correctly copied all the code above (nothing more, nothing less).

### 7.3.1 Take a screen shot of the simulator in full and add it to your submission document

Notice that the memory window has changed ? You should see lots of values in at least the first 13 rows of memory words.

### 7.3.2 Based on what we've learnt about assemblers and Von Neuman architectures, explain what you think just happened.

You will also see that ARMLite has now added 'line numbers' to your program. These do not form part of the source code, but are there to help you navigate and discuss your code.

Hover the mouse over one of the lines of the source code (after the code has been submitted).

You will see a pop-up tooltip showing a 5 digit hex value.

### 7.3.3 Based on what we have learnt about memory addressing in ARMLite, and your response to 7.3.2, what do you think this value represents ?

*If you're not sure what's going on here, ask your tutor for assistance.*

It's time to try a few other things.

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:

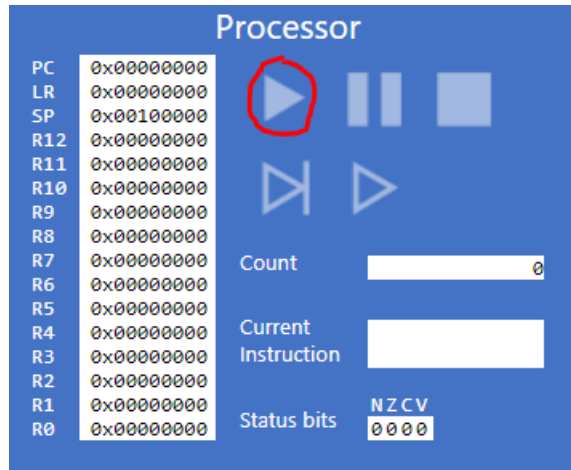
- The blank lines
- Additional spaces
- The comments
- The line numbers
- The total number of instructions that end up as words in memory? (Why?)

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

OK - enough mucking about. Restore the program to its original condition, either by going back to Edit, or just copying it again, and click **Submit**. Time to run this thing!

## Executing and Debugging Assembly Code

To execute the assembled source code, we need to click the Run button, circled below:



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 display will stabilise.

*The program you have loaded and just run 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 [Conway's Game of Life](#) [Links to an external site.](#) for more information).*

To stop the program, you can click on the square **Stop** button in the above image. To run the program again, simply click **Run**.

You will notice the behaviour of the program is different each time you run it - this is because the starting pattern of cells is randomised.

Run the program once more.

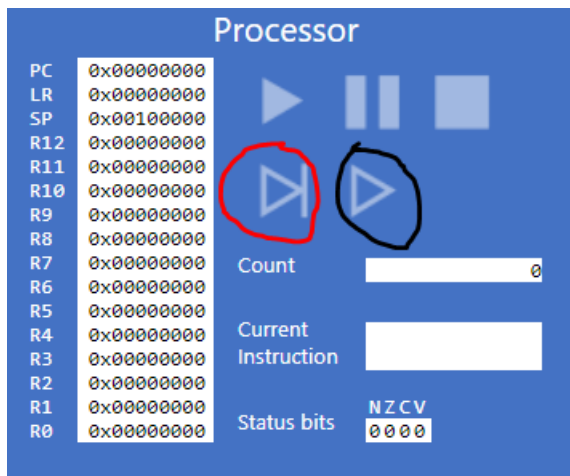
While the program is executing, click the **Pause** button (between the **Run** and **Stop** buttons).

As well as freezing the graphics screen, you will also see orange highlighting appear in both the Program and Memory windows.

### 7.4.1 What do you think the highlighting in both windows signifies ?

You can continue execution by pressing **Play** again. Do this and then click **Pause** again.

Now click the button circled in red below.



#### 7.4.2 What do you think happens when you click the button circled in red ?

Now click the button circled in black and notice what happens.

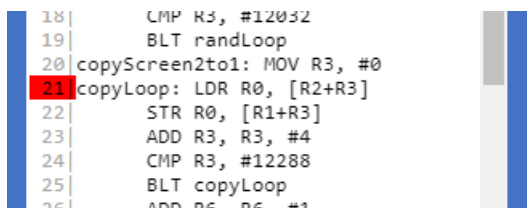
You will hopefully notice that the program resumes execution, but at a substantially slower pace than before. You will also notice the orange highlighting in the Program window stepping through lines of code.

Now click the same button again and see what happens. You will hopefully see that when you click the button a few times in succession, the execution speed increases.

*These two buttons allow you to slow things down to literally, in the case of the red circled button, single steps of code execution. This is invaluable for debugging code, particularly when you want to check whether the outcome of a given instruction has produced what you expect (be that a value in memory, in a register, or a graphical element on the display etc).*

Finally, while paused, click line number 21 of the source code in the Program Window.

This will paint a red background behind the line number like this:



This is called 'setting a breakpoint' and will cause processing to be paused when the breakpoint is reached.

Having set the breakpoint, continue running until the pause is observed (almost immediately!).

#### 7.4.3 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.

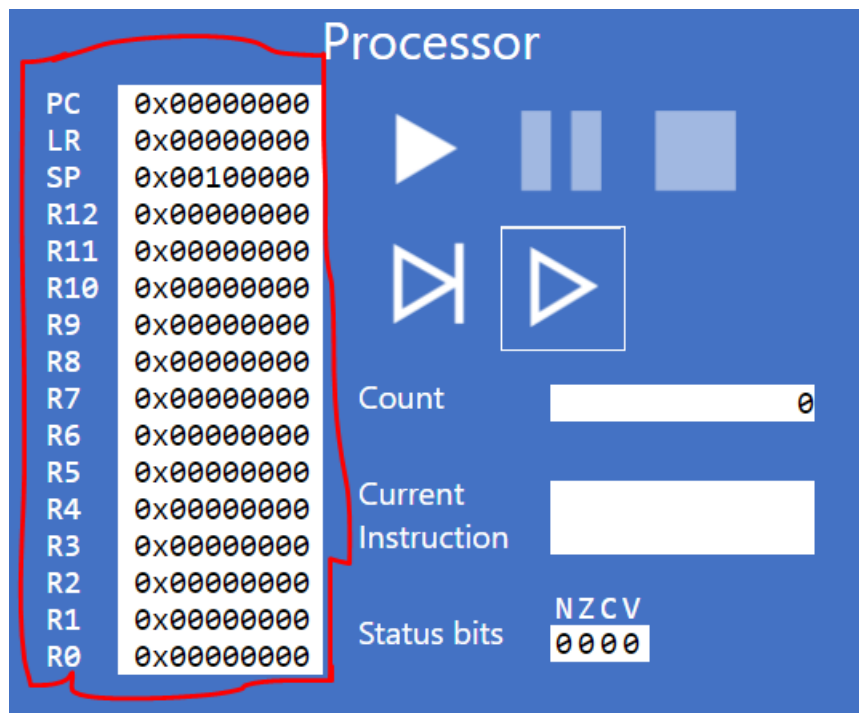
While paused you can remove a breakpoint by clicking on the line again.

## Registers and Basic Operations

Registers are fundamental to how data is stored and manipulated within a CPU. In the early part of this unit we learnt how registers are generally built from banks Flip Flops, each storing a single bit. Here we see how registers serve the needs of the programs we seek to write and execute.

The ARMLite simulator provides 16 registers, including 13 so called *general purpose registers* which can be used within the programs we write.

The general purpose registers are labelled, R0-R12 as shown in the image below.



As with main memory storage in ARMLite, registers each hold 32 bits, with the value in each register represented by an 8 digit hexadecimal value.

In lectures we introduced the MOV instruction for storing values in registers. For example:

```
MOV R1, #15
```

takes the decimal value 15 and stores it in register R1.

```
MOV R2, R1
```

takes the value stored in R1 and copies it to register R2.

We also introduced some basic arithmetic operations like ADD (for adding) and SUB (for subtracting).

We are not going to write a game in assembly language (yet!), but rather, we are going to play a game that involves assembly programming.

**Task:** You are given 6 input values and a target value. Your task is to write a simple assembly program that implement a mathematical equation involving only these 6 input values, and the three instructions MOV, ADD and SUB, such that the result is as close as possible to the target value.

Here is an example to get you started. Your initial input numbers are 100, 25,8,7,3,1 and your target is 84. This is pretty straight forward from a mathematical perspective:  $1+8+100-25 = 84$

A possible implementation of this equation in ARM assembly code would be:

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



Clear the Program window in ARMLite and type in the lines of code above. When done, Submit the code ready for execution (and if any errors occur, check your syntax matches the above).

Now, using the Step button (rather than the Run button), execute the first instruction, `MOV R0, #1`

Verify R0 contains the value 0x00000001

The second instruction is `ADD R1,R0,#8`.

**7.5.1 Before executing this instruction, describe in words what you think this instruction is going to do, and what values you expect to see in R0 and R1 when it is complete ?**

Now execute the instruction and verify whether the output matches your expectation.

Do the same for the remaining instructions.

**7.5.2 When the program is complete, take a screen shot of the register table showing the values.**

*You will have noticed the **HALT** instruction, which does the obvious task of halting the program. This instruction is important for telling the program counter (which automatically steps through the program instructions one-by-one to cease doing so. Without this instruction, program execution would simply continue to sequentially read and execute 32 bit words of memory beyond the last address of the loaded program code. In practise, this will most likely result in the program simply not working, however on a real microprocessor, can be potentially dangerous due to the possibility of executable code residing in memory from previously loaded code - this can result in unpredictable program behaviour.*

**7.5.3 Task: Your 6 initial numbers are now 300, 21, 5, 64, 92, 18. Write an Assembly Program that uses these values to compute a final value of 294 (you need only use MOV, ADD and SUB). Place your final result in register R7 (don't forget the HALT instruction)**

***When the program is complete, take a screen shot of the code and the register table.***

In this week's lectures we also introduced a small set of so-called Bit-wise instructions, designed to manipulate bits within a register in specific (and highly useful ways).

Recall the following:

Instruction	Example	Description
<b>AND</b>	<code>AND R2, R1, #4</code>	Performs a bit-wise logical AND on the two input values, storing the result in the equivalent bit of the destination register.
<b>ORR</b>	<code>ORR R1, R3, R5</code>	As above but using a logical OR
<b>EOR</b>	<code>EOR R1, R1, #15</code>	As above but using a logical 'Exclusive OR'
<b>LSL</b>	<code>LSL R1, R1, #3</code>	'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 leftmost bits, and adding zeros on the right.
<b>LSR</b>	<code>LSR R1, R1, R2</code>	'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.



Instruction	Decimal value of the destination register after executing this instruction	Binary value of the destination register after executing this instruction
MOV ...		

**7.5.4 Task:** *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.

**Enter your program into ARMLite, submit the code and when its ready to run, step through the program, completing the table below (make a copy of it in your submission document)**

**Task 7.5.5** Lets play the game we played in 7.5.3, but this time you can use any of the instructions listed in this lab so far (ie,. MOV, AND, OR, and any of the bit-wise operators).

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

**When the program is complete, take a screen shot of the code and the register table and paste into your submission document.**

**Task 7.5.6: Let's play again !**

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

**When the program is complete, take a screen shot of the code and the register table and paste into your submission document.**

## Signed Integers

Copy and Paste the following code into the ARMLite code editor and submit the code.

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

Before executing, switch ARMLite to display data in memory in *Decimal (signed)* using the drop down box below the memory grid.

Now run the program and note the result in register R1.

**7.6.1 - Why is the result shown in R1 a negative decimal number, and with no obvious relationship to 9999 ?**

*Hint: Mouse over the values in R0 and R1 and take a look at the binary strings.*

Switch ARMLite to display in Binary format using the dropdown box under the memory grid.

You can't edit register values directly, but you can edit memory words. Click on the top-left memory word (address 0x000000) 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.

**7.6.3 - What is the binary representation of each of these signed decimal numbers: 1, -1, 2, -2 What pattern do you notice ? Make a note of these in your submission document before reading on.**

The ARMLite simulator is using 2's Complement to represent signed integer values. Recall from lectures how 2's Complement works to get the negative version of a number:

1. invert (or 'flip') each of the bits
2. Add 1

**7.6.4 - Write an ARM Assembly program that converts a positive decimal integer into its negative version. Start by moving the input value into R0, and leaving the result in R1.**

*Hint: the ARM instruction MVN, which works like MOV but flips each bit in the destination register, may be useful for this !*

**Take a screen shot showing your program and the registers after succesful execution, and paste into your submission document.**

Using the program you wrote above, enter the negative version of the number you previously tested as the input (ie use the output of the previous test as the input).

What do you notice ?

All things working as they should be, you should see that the exact same 2's Complement conversion works in both directions (ie positive to negative, and negative to positive).