Faculty of Science, Engineering and Technology

Computer Systems

Week 7

Overview

Name: Le Quang Hai
Student ID: 104175779

7.1.1 What value is displayed ? Why?

Click on any visible memory word and type in 101

Value displayed: 65

Because: by default simulator display in hexadecimal so 101(decimal) → 65(hexadecimal)

7.1.2 What value is displayed, and why?

On another memory word, enter 0b101

Value displayed: 5

Because: ‘0b’ means in binary so 101(binary) → 5(hexadecimal)

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?

Ans: The value of the ‘word’ in hexadecimal and in binary

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.

Ans: The 65 in 7.1.1 turns 101. The 5 in 7.1.2 remains 5 since 5(hexadecimal) = 5(decimal)

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

Ans: It shows the value in hexadecimal and binary

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 ?

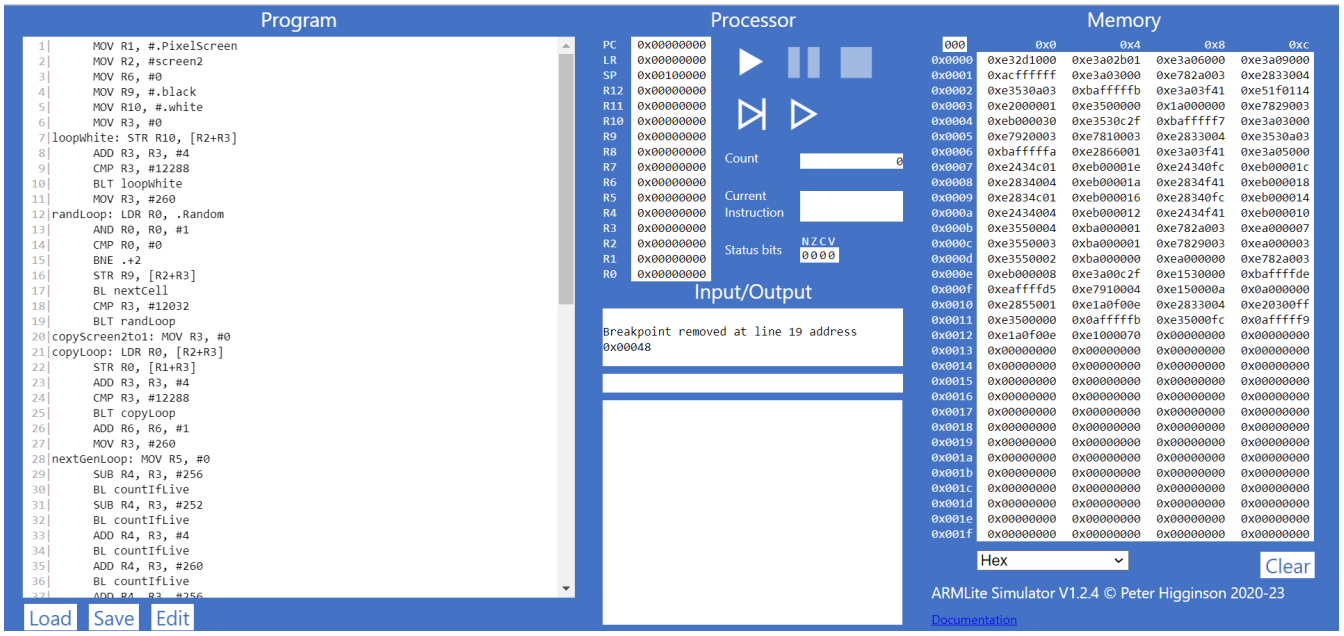
Ans: No, it does not

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 !

Ans: because each memory 'word' contains 4 bytes.

7.3.1 Take a screenshot of the simulator in full and add it to your submission document



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

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.

Ans: The source code is saved into the memory, each line of indented code is stored in a data 'word' in the memory

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

Ans: It represents the location address in the memory where the code semantic is stored in binary.

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

Ans:

- The blank lines: **is eliminated**
- Additional spaces: **is eliminated**
- The comments: **is colored green**
- The line numbers: **increments by 1**
- The total number of instructions that end up as words in memory: **it does not change since no additional command was added.**

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

Ans: The first line is highlighted and output says ‘syntax error at line 1 MOV’

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

Ans: It signifies the code line being executed

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

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

Ans: The code is executed one line on click and the line on run is highlighted

7.4.2b Now click the button circled in black and notice what happens.

Ans: The code is executed one by one continuously and the line on run is highlighted

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 ?

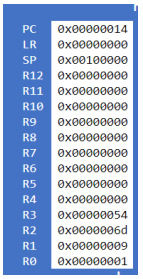
Ans: The processor paused before the breakpoint

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.

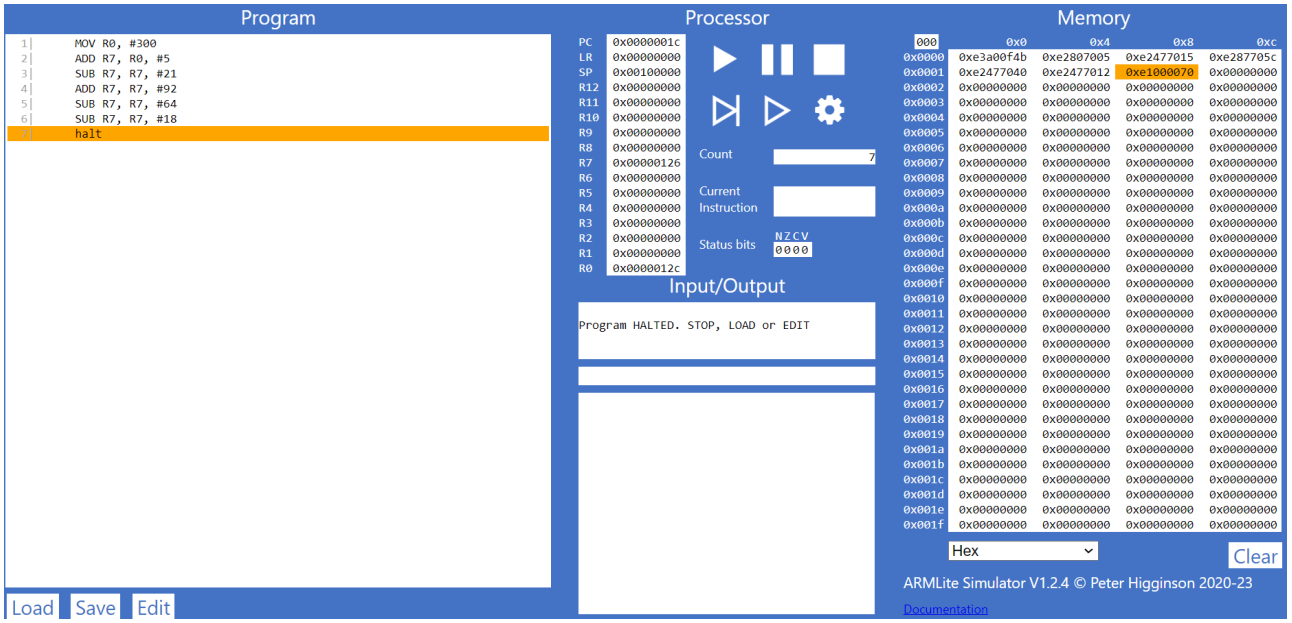
7.5.1 Before executing this instruction, describe in words what you think this instruction is going to do, and what values you expect to see in R0 and R1 when it is complete ?

Ans: expected outcome: R0 = 1 & R1 = 8 + R0 = 9 (hexadecimal)



7.5.2 When the program is complete, take a screenshot of the register table showing the values → →

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).***



When the program is complete, take a screenshot of the code and the register table.

Recall the following:

Instruction	Example	Description
AND	AND R2, R1, #4	Performs a bitwise 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 leftmost 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 rightmost bits, and adding zeros on the left.

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 ARM lite, 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)

Instruction	Decimal value of the destination register after executing this instruction	Binary value of the destination register after executing this instruction
MOV R0, #3	3	11
MOV R1, #4	4	100
STR R0, _var1	3	11
STR R1, _var2	4	100
HALT		
_var1: .word 1		
_var2: .word 2		

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 bitwise 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 screenshot 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 screenshot of the code and the register table and paste into your submission document.

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

Ans: A = 0d9999 = 0b10011100001111

→ Shift 18 we get A' = 0b10011100001111000000000000000000

→ In 2 complement - A' = 0b01100011110001000000000000000000 = 0d1673789440

→ In decimal A = 0d (-1673789440)

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

1 = 0b1

-1 = 0b11111111111111111111111111111111

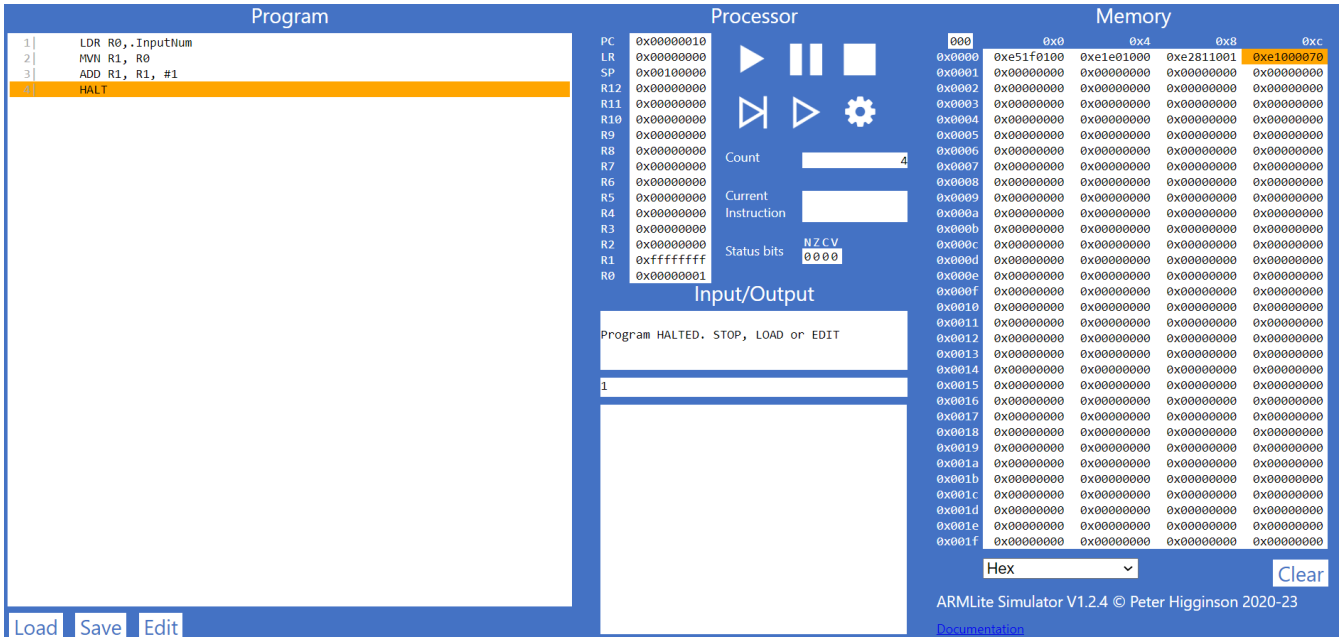
2 = 0b10

-2 = 0b11111111111111111111111111111110

What pattern do you notice

Ans: Negative number has the first bit being 1 and to decrement a negative number we decrement the last 31 bits.

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.



What do you notice ?

Ans: It works with negative input, too!