Tài liệu không có tiêu đề

hahahaha (Trường Đại học Sư phạm Kỹ Thuật Thành phố Hồ Chí Minh)

Lab4

Full in name: Group:

Student ID:

Part 4.1 Memory in ARMLite

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).

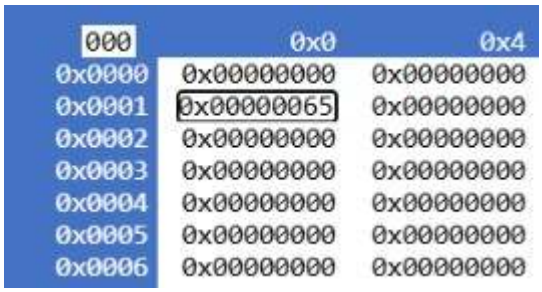
4.1.1 What value is displayed ? Why ?

Click on any visible memory word and type in 101 :

Answer:

- Khi nhập 101 ở hệ thập phân, giá trị hiển thị sẽ là 0x00000065 ở hệ hex. Điều này xảy ra vì 101 trong hệ thập phân tương ứng với 0x65 trong hệ hex.

- Hình minh họa:



Click on another memory word, enter 0x101:

Answer:

- Khi nhập 0x101 (hệ hex), giá trị hiển thị sẽ là 0x00000101. Điều này xảy ra vì 0x101 là giá trị hex trực tiếp được lưu trữ trong ô nhớ, tương đương với 257 trong hệ thập phân.

- Hình minh họa:

4.1.2 What value is displayed, and why?

Answer: Khi bạn nhập 0b101, giá trị hiển thị có thể 0x5. Tiền tố 0b biểu thị nhị phân và 0b101 trong nhị phân tương đương với 5 ở dạng thập phân, được hiển thị dưới dạng 0x5 trong hệ thập lục phân.

Hình ảnh minh họa:

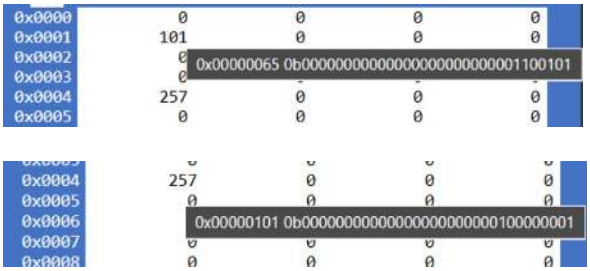
4.1.3 What value is displayed, and why?

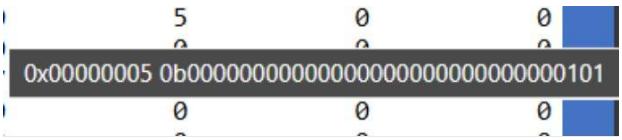
- ***Answer:***Khi nhập "0b101", ARMLite nhận dạng "0b" là định dạng số nhị phân và lưu trữ giá trị 0b101 trong bộ nhớ, tương đương với 5 trong hệ thập phân. Giá trị được lưu sẽ là 00000005 (hex).

  => Result:

- ***Decimal(unsigned):***

  => Result:





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?

Answer: The value in decimal and binary

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?

Answer: The value in hex and binary

4.1.4: 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 ?

explain:

1. ***Bộ nhớ và cách biểu diễn dữ liệu:***
    - Trong bộ giả lập ARMLite, dữ liệu có thể được hiển thị ở nhiều cơ sở số khác nhau (như thập phân, thập lục phân, nhị phân).
    - Việc thay đổi cơ sở biểu diễn chỉ thay đổi cách hiển thị dữ liệu trong các ô bộ nhớ mà không thay đổi giá trị thực sự được lưu trữ trong bộ nhớ.
2. ***Tiêu đề hàng và cột:***
    - Các tiêu đề hàng và cột (các chữ số trắng trên nền xanh dương) thể hiện địa chỉ bộ nhớ và không phải là dữ liệu được lưu trữ trong ô bộ nhớ. Chúng đại diện cho địa chỉ của các ô, được hiển thị dưới dạng thập lục phân.
    - Bởi vì tiêu đề hàng và cột dùng để chỉ địa chỉ, không phải dữ liệu, nên chúng không bị ảnh hưởng khi người dùng thay đổi cách hiển thị dữ liệu trong ô bộ nhớ.

Kết luận:

- ***Việc thay đổi cách biểu diễn dữ liệu trong ô bộ nhớ không làm thay đổi cách biểu diễn của các tiêu đề hàng và cột.***
- ***Lý do: Các tiêu đề hàng và cột là phần cố định thể hiện địa chỉ và luôn được hiển thị ở dạng thập lục phân để dễ nhận biết, bất kể dữ liệu được hiển thị ở cơ sở nào.***

Part 4.2 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:

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:

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.

4.2.1 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.

Trả lời:

- **Các độ lệch địa chỉ bộ nhớ ở tiêu đề cột lại tăng lên theo bội số của 0x4 vì:**

- Mỗi từ bộ nhớ trong ARMLite có kích thước 32 bit (hay 4 byte). Khi sử dụng địa chỉ, mỗi từ bộ nhớ chiếm 4 byte, suy ra địa chỉ của từng từ sẽ cách nhau 4 byte trong bộ nhớ. Trong hệ hex, 4 được biểu diễn là 0x4.

- Vì vậy các offset tăng theo bội số của 0x4 (0x0, 0x4, 0x8, 0xC)

-> Từ đó giúp đảm bảo rằng mỗi word 32-bit được lưu trữ và truy cập đúng theo alignment của nó trong bộ nhớ.

Ví dụ: Địa chỉ đầu tiên (tiêu đề hàng) là **0x00000**, là từ trong bộ nhớ đầu tiên.

Từ bộ nhớ tiếp theo sẽ có địa chỉ là **0x00004** (tăng thêm 4 byte).

Từ bộ nhớ thứ ba sẽ có địa chỉ là **0x00008**.

Part 4.3: Editting 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:

```
 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 MOV R1, #.PixelScreen
 MOV R2, #screen2
```

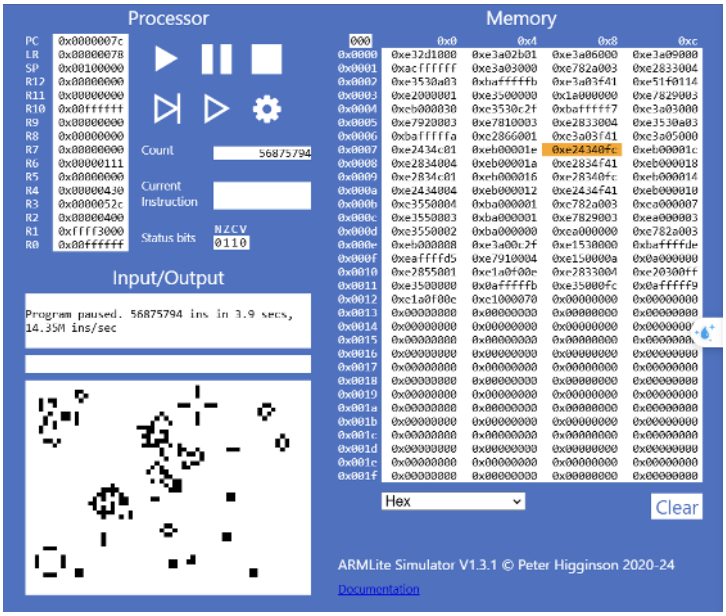
```
 MOV R6, #0
 MOV R9, #.black
 MOV R10, #.white
 MOV R3, #0
loopWhite: STR R10, [R2+R3]
 ADD R3, R3, #4
 CMP R3, #12288
 BLT loopWhite
 MOV R3, #260
randLoop: LDR R0, .Random
 AND R0, R0, #1
 CMP R0, #0
 BNE .+2
 STR R9, [R2+R3]
 BL nextCell
 CMP R3, #12032
 BLT randLoop
copyScreen2to1: MOV R3, #0
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
```

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).

4.3.1 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.



4.3.2 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.

answer:

1. **Hiện tượng sau khi nộp mã nguồn:**
   - Khi bạn nộp mã nguồn trong cửa sổ chương trình của ARMLite và không có lỗi xảy ra, mã sẽ được biên dịch thành mã máy và nạp vào bộ nhớ.
   - **Dấu hiệu nhận biết:** Cửa sổ bộ nhớ sẽ hiển thị các giá trị khác nhau, thay đổi từ trạng thái ban đầu là toàn số `0`. Các giá trị mới xuất hiện là mã máy tương ứng với các lệnh đã được biên dịch.
   - Ngoài ra, bạn cũng sẽ thấy các dòng mã lệnh có thêm số dòng ở phía bên trái để giúp theo dõi dễ dàng hơn. Các số dòng này không phải là một phần của mã nguồn mà chỉ là chỉ dẫn để điều hướng và thảo luận mã.
2. **Giải thích chi tiết điều gì đã xảy ra:**
   - Khi nộp mã, trình biên dịch đã chuyển đổi mã Assembly thành các giá trị nhị phân 32 bit (mã máy) tương ứng với từng lệnh. Những giá trị này được nạp vào bộ nhớ để có thể được bộ xử lý đọc và thực thi.
   - **Kết quả trong cửa sổ bộ nhớ:** Các dòng đầu tiên của bộ nhớ sẽ chứa mã máy cho từng lệnh từ chương trình, mỗi lệnh chiếm một từ bộ nhớ (4 byte).

- **Sự xuất hiện của các số dòng:** Các số dòng xuất hiện sau khi nộp mã giúp bạn dễ dàng định vị các lệnh trong mã nguồn khi thực hiện các bước như chỉnh sửa, debug, hoặc thảo luận về mã..

4.3.3 Based on what we have learnt about memory addressing in ARMlite, and your response to 4.3.2, what do you think this value represents ?

1. **Địa chỉ bộ nhớ trong ARMLite:**
   - Trong ARMLite, mỗi lệnh trong chương trình được lưu trữ tại một vị trí cụ thể trong bộ nhớ.
   - Địa chỉ bộ nhớ thường được biểu diễn dưới dạng số thập lục phân (hexadecimal) để dễ dàng đọc và quản lý.
2. **Hiển thị địa chỉ khi di chuột:**
   - Khi di chuột qua một dòng mã nguồn, ARMLite hiển thị địa chỉ bộ nhớ tương ứng với lệnh đó.
   - Điều này giúp lập trình viên biết chính xác vị trí của lệnh trong bộ nhớ, hỗ trợ quá trình debug và tối ưu hóa chương trình.
3. **Tại sao địa chỉ có 5 chữ số thập lục phân:**
   - Bộ vi xử lý ARM sử dụng không gian địa chỉ 32-bit, cho phép đánh địa chỉ từ `0x00000000` đến `0xFFFFFFFF`.
   - Tuy nhiên, trong nhiều trường hợp, các địa chỉ thực tế chỉ sử dụng một phần của không gian này.
   - Việc hiển thị 5 chữ số thập lục phân (từ `0x00000` đến `0xFFFFF`) đủ để bao quát phạm vi địa chỉ của nhiều chương trình nhỏ, giúp giao diện gọn gàng và dễ theo dõi hơn.

Ví dụ minh họa:

Giả sử bạn có một chương trình với các lệnh được lưu trữ bắt đầu từ địa chỉ `0x00000`. Khi di chuột qua dòng lệnh đầu tiên, bạn sẽ thấy địa chỉ `0x00000` xuất hiện. Di chuột qua dòng lệnh thứ hai, địa chỉ `0x00004` sẽ hiện ra, và cứ tiếp tục như vậy.

Kết luận:

Giá trị 5 chữ số thập lục phân xuất hiện khi di chuột qua các dòng mã nguồn trong ARMLite là địa chỉ bộ nhớ nơi lệnh tương ứng được lưu trữ. Việc hiển thị địa chỉ này giúp lập trình viên theo dõi và quản lý chương trình một cách hiệu quả.

Part 4.4 Executing and Debugging Assembly Code

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

Run.PNG

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 LifeLinks 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.

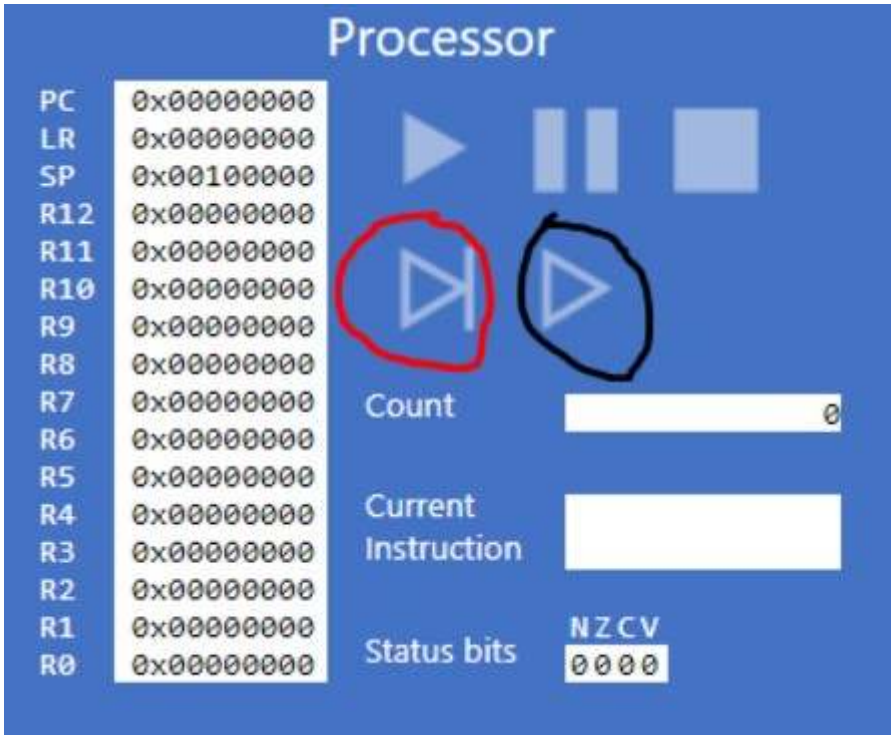
While the program is executing, click the Pause button (between the Run and Stop buttons).

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

4.4.1 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.



Nút được khoanh tròn màu đỏ biểu thị chức năng cho phép thực thi mã theo từng lệnh một. Khi nhấp vào, nút này sẽ đẩy nhanh quá trình thực thi theo từng lệnh và cập nhật trạng thái bộ xử lý theo đó.

Nút được khoanh tròn màu đen cho phép chương trình thực thi ở tốc độ chậm hơn, tự động, giúp dễ dàng quan sát các thay đổi trạng thái của bộ xử lý theo thời gian thực.

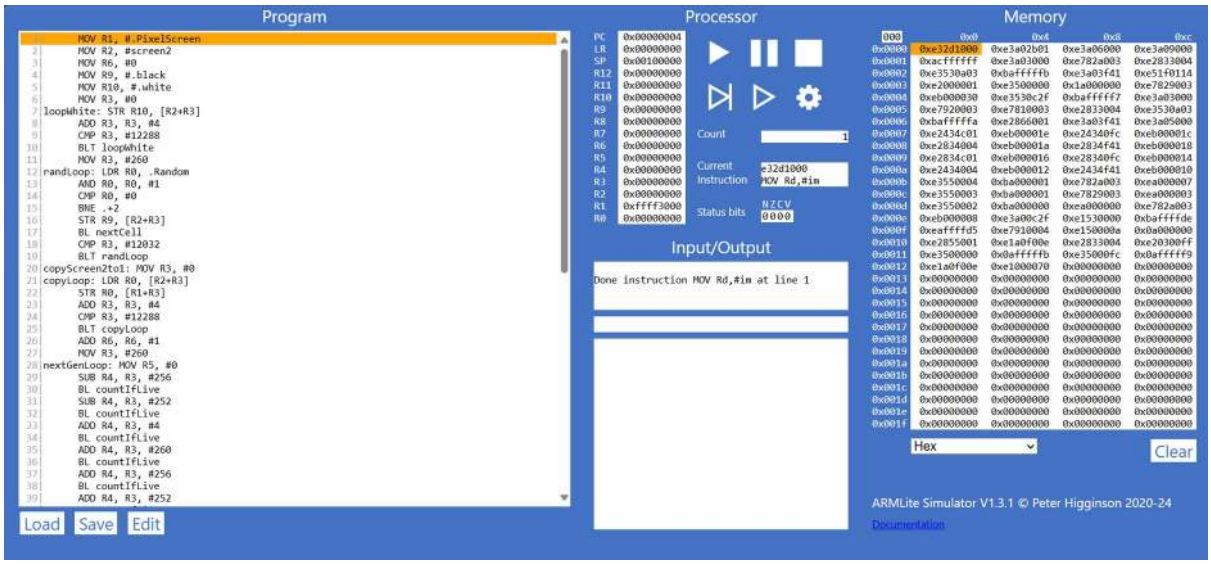
4.4.2 What do you think happens when you click the button circled in red ?

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.

Answer: Khi tôi ấn vào nút đỏ nó sẽ ra:



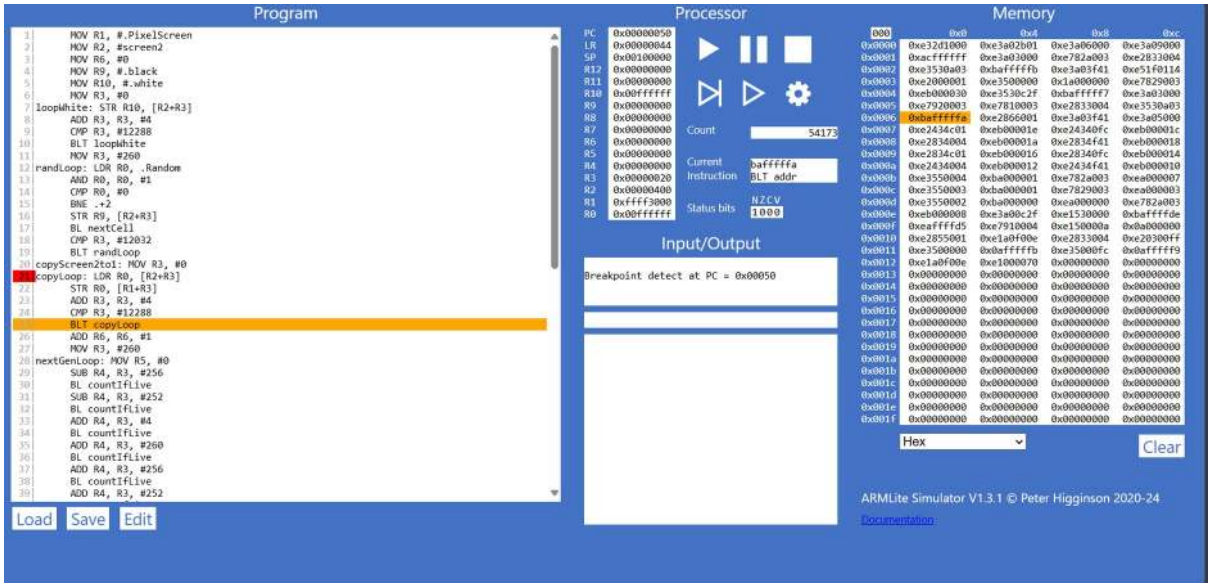
Có nghĩa là chương trình sẽ thực hiện từng lệnh một cách tuần tự vì khi mỗi lần ấn nút đỏ có nghĩa là Step thì sẽ tiến tới lệnh kế tiếp trong mã nguồn cho phép bạn quan sát và phân tích từng bước hoạt động của mã

Sau khi tôi ấn nút đen có nghĩa là slow, chương trình sẽ chạy vòng lặp ở loopwhite vô tận ko dừng

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 break point’ 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!).

Answer: Chương trình sẽ tiếp tục thực thi cho đến khi đạt đến điểm dừng này và sẽ tự động tạm dừng ngay lập tức tại đó, thấy qua thông báo “Breakpoint detect at PC = 0x00050” trong ô Input/Output.

4.4.3 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.

Điểm ngắt rất cần thiết để gỡ lỗi, vì chúng cho phép bạn tạm dừng chương trình tại các điểm chiến lược và kiểm tra trạng thái hiện tại, giúp xác định các vấn đề dễ dàng hơn. Khi bị tạm dừng, bạn có thể bước qua mã, tiếp tục chạy với tốc độ chậm hơn hoặc tiếp tục thực thi tốc độ tối đa. Để xóa điểm ngắt, chỉ cần nhấp lại vào dòng, thao tác này sẽ xóa nền đỏ.

Part 4.5 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 hexidecimal 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, 25,8,4,3,1 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.

4.5.1 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.

Trước khi thực hiện lệnh
MOV R0,#1
ADD R1,R0,#8
ADD R2,R1,#100
SUB R3,R2,#25
HALT

Ta giải thích hàm MOV là đưa giá trị 1 vào thanh ghi R0(gán R0 là 1)
hàm ADD là cộng giá trị xong lưu vào thanh ghi trước đó
VD hàm ADD R1,R0, #8 với R0 gán là 1 thì ta sẽ cộng 8+1 sau đó gán kết quả vào thanh ghi R1 là R1=9

Hàm SUB là lệnh trừ thì sau khi thực hiện hàm ADD thứ 2 ta có thanh ghi R2 là 109 thì hàm SUB sẽ là 109 - 25 =84 sau đó sẽ gán R3= 84 và lệnh HALT là sẽ dừng chương trình lại không cho thực hiện câu lệnh tiếp theo.

4.5.2 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.*

Lệnh đầu tiên

4.5.3 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 R4 (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:

Hàm

MOV R0, #300

SUB R1, R0, #5

SUB R2, R1, #1

MOV R4, R2

HALT

Instruction	Decimal value of the destination register after executing this instruction
MOV	300
SUB	295
SUB	294
MOV	294
HALT	exit

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 bit-wise logical OR
EOR	EOR R1, R1, #15	As above but using a bit-wise 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 right-most bits, and adding zeros on the left.

4.5.4 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)

Hàm:

```
MOV R0, #20
AND R1, R0, #15
ORR R2, R0, #30
EOR R3, R0, #25
LSL R4, R0, #8
LSR R5, R0, #2
HALT
```

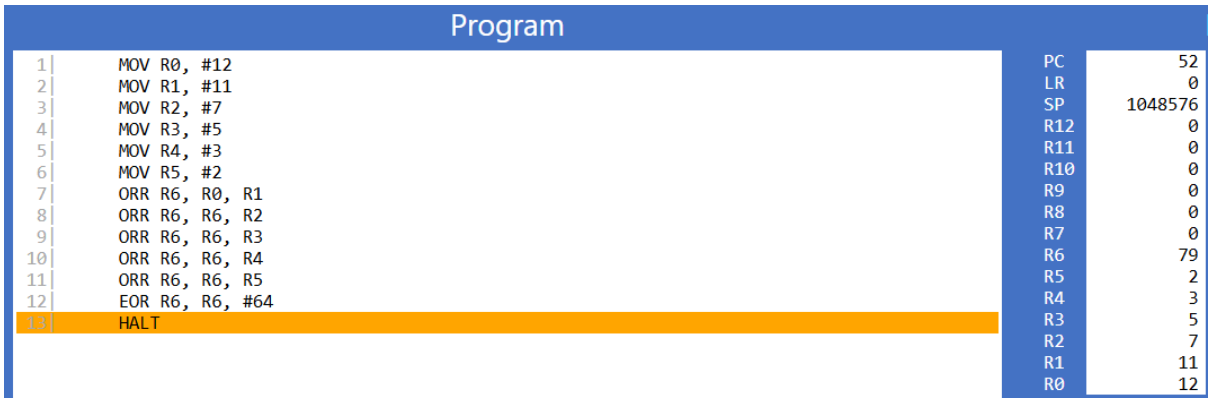
Instruction	Decimal value of the destination register after executing this instruction	Binary value of the destination register after executing this instruction
MOV	20	0b10100
AND	4	0b01000
ORR	30	0b11110
EOR	13	0b01101
LSL	5120	0b10100
LSR	5	0b00101

Task 4.5.5 Lets play the game we played in 4.5.3, 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.

Answer:

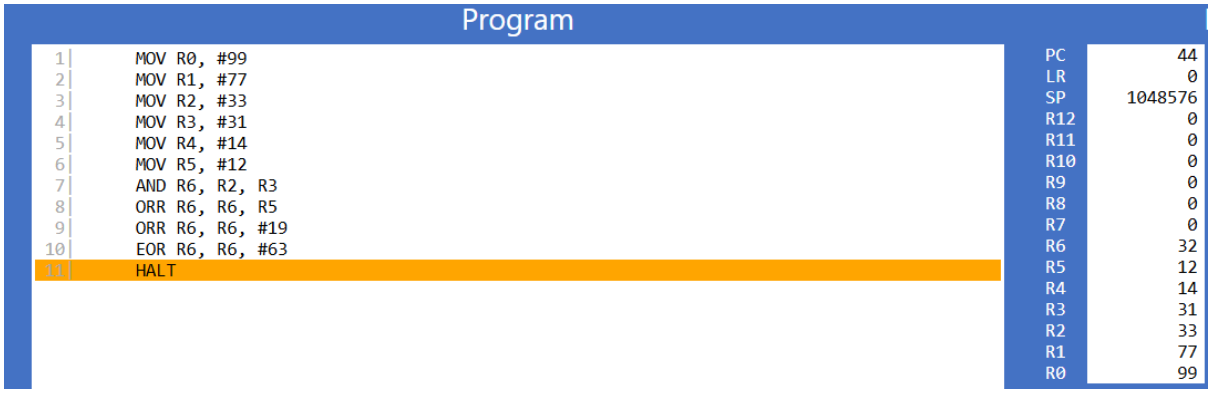


Task 4.5.6: Let's play again !

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.

Answer:



Part 4.6 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.

4.6.1 - 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 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.

Answer:

Giải thích các lệnh:

MOV R0, #9999: Gán giá trị 9999 vào thanh ghi R0R0R0.

LSL R1, R0, #18: Dịch trái R0 18 bit và lưu kết quả vào R1.

Giải thích kết quả

- **Dịch trái (LSL)** 18 bit tương đương với việc nhân giá trị ban đầu của R0 với 2^18
- Giá trị ban đầu R0=9999 sẽ trở thành 9999×2^18= 9999 x 262144 sau khi dịch trái.

Hiện tượng số âm

- Trong hệ thống 32-bit, nếu kết quả tính toán vượt quá 2,147,483,6472,147,483,6472,147,483,647 (giới hạn của số dương 32-bit có dấu), bit cao nhất sẽ trở thành 1, khiến ARM diễn giải giá trị này là số âm theo bù 2.
- Trong trường hợp này, giá trị nhị phân của 2621184256 là 10011100001110010100000000000000, với bit cao nhất (bit thứ 31) là 1. Do đó, ARM hiểu đây là một số âm.

4.6.3 - What is the binary representation of each of these signed decimal numbers: 1, -1, 2, -2

- number 1: 0000 0000 0000 0000 0000 0000 0000 0001

- number -1: 1111 1111 1111 1111 1111 1111 1111 1111

- number 2: 0000 0000 0000 0000 0000 000 0000 0010

- number -2: 1111 1111 1111 1111 1111 1111 1111 1110

What pattern do you notice ? Make a note of these in your submission document before reading on.

- khi đưa 1 số về số âm của nó thì tất cả ta sẽ flip tất cả các bit (từ 1 thành 0, từ 0 thành 1) và cộng thêm 1 bit

- ví dụ số 2 sang -2 là 0000 0010 thành (1111 1101 + 1) = 1111 1110

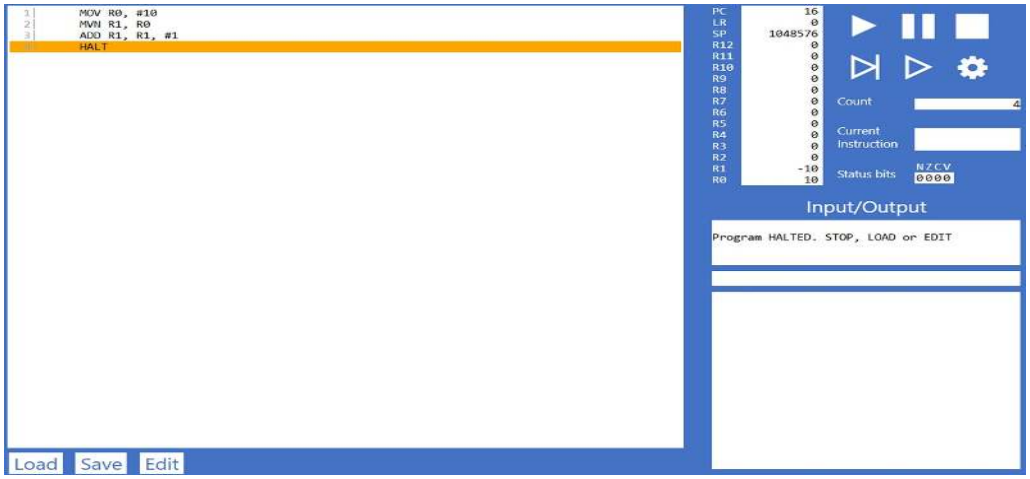
The ARMlite simulator is using 2's Compliment to represent signed integer values. Recall from lectures how 2's Compliment works to get the negative version of a number:

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

4.6.4 - 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?

- Khi lấy kết quả số âm của input số dương vừa nhập vào làm input ở R0 mới thì kết quả ở R1 là số dương của số âm đó

- Với cách thức flip tất cả bit và cộng thêm 1 thì chương trình sẽ chuyển số dương thành số âm và ngược lại

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