Faculty of Science, Engineering and Technology



Computer Systems

Week 7

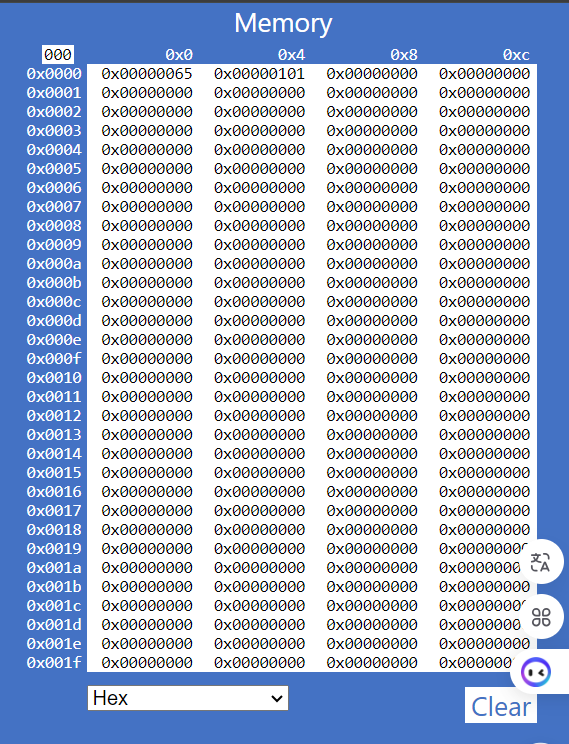
***A. Part 7.1: Memory in ARMLite***

A screenshot of a computer screen

Description automatically generated ***Question 7.1.1: What value is displayed? Why?***

The ARMLite Simulator uses 32-bit words, and when I input a decimal number, it converts that number into its hexadecimal representation. As a result, the input value ‘101’ will be stored as ‘0x00000065’ in the selected memory address.

**Question *7.1.2: What value is displayed, and why?***

 The displayed value is ‘0x00000101’ because the simulator stores the hexadecimal input directly as a 32-bit word, padding it with leading zeros to fit the format. Because of that, ‘0x101’ is displayed as ‘0x00000101’ in the memory address. Additionally, in decimal. ‘0x101' is displayed as 257 (160+161+162=257).

**Question *7.1.3: What value is displayed, and why?***

A screenshot of a computer code

Description automatically generated

- First, I would like to explain the input ‘0b101’ which is in the binary format:

+ The prefix ‘0b’ indicates that the number is in binary

+ The digits ‘101’ represent the binary number

- The simulator converts the binary input ‘0b101’ to its decimal equivalent, which is ‘5’, and then displays it in hexadecimal format as ‘0x00000005’, padding with leading zeros to fit the 32-word representation.

**Question *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?***

A screenshot of a computer

Description automatically generated

- From my perspective, it shouldn’t. In addition, changing the representation of the data in memory does not change the representation of the row and column headers. The row and column headers should remain consistent, displaying th memory addresses in a standard format (usually hexadecimal), regardless of the data representation used.

***B. Part 7.2: Memory Addressing***

A blue screen with white text

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

The reason memory address offsets increase in multiplies of ‘0x4’ (4 bytes) is primarily due to the following factors:

+ Word Sizes: Many computer architectures, inclduing ARM and x86, use a word size of 4 bytes (32 bts). This means that data bytes like integers are stored in 4-byte chunks.

+ Data Alignment: Data types often have alignment requirements. For instance, a 32-bit integer is typically aligned on a 4-byte boundary. This ensures that each integer in an array occupies addresses that are multiplies of 4.

+ Efficient Memory Access: CPUs access memory in blocks (cache lines) and accessing data in multiples of the world size optimizes these operations. This leads to better performance when reading or writing data.

***C. Part 7.3: Editting and Submitting Assembly Code***

A screenshot of a computer

Description automatically generated**Question 7.3.1: Take a screen shot of the simulator in full and add it to your submission document**

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

- Memory Window Changes:

+ The memory window displaying values indicates that the program has been executed or loaded into memory. This is typically in Von Neumann architectures, which both data and instructions are stored in the same memory space.

+ The presence of values in the first 13 rows suggests that the program has initialized certain variables or executed instructions that manipulate memory.

- Line numbers:

+ The addition of line numbers to the source code serves as a reference for easier navigation and debugging. While they do not form part of the actual source code, they help programmers identify specific lines when discussing or troubleshooting the code.

+ The tooltips showing a 5-digit hex value when hovering over the lines indicate the memory address associated with that line of code. This is useful for understanding where each instruction of data resides in memory.

- In conclusion:

+ The changes in the memory window reflect the prgram’s execution and memory allocation.

+ Line numbers and tooltips enhance code navigation and debuggging, providing clarity on the relationship between source code and memory addresses.

**Question 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?**

The 5-digit hex value represents the memory address of the instruction or data associated with the line of code. This is essential for the CPU to fetch and execute instructions correctly:

+ Memory Location: Indicates where in memory the code or variable is stored

+ Hexadecimal Format: Provides a compact representation of binary values

+ Contextual Significance: Crucial for program execution, debugging, and organization in Von Neuman architectures

***D. Part 7.4: Executing and Debugging Assembly Code***

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

The highlighting in both the Program and Memory windows signifies:

- Current Instruction Execution: It indicates which specific line of code is currently being executed, allowing users to track the program’s flow.

- Memory Access: It shows the specific memory address being accessed or modified by the current instruction, providing insight into how data is being manipulated during execution.

**Question 7.4.2: What do you think happens when you click the button circled in red?**

When clicking on the button circled in red which is typically the Pause button, the following occurs:

- Execution Freeze: The program execution is halted immediately. This means that the simulation of the colony of organisms stops, and no further changes occur in the graphics screen.

- Visual Feedback: The graphics screen displays the current state of the simulation.

- Highlighting Activation: The current instruction in the Program window and the accessed memory in the Memory window are highlighted for easier analysis.

=> This allows for observation and debugging of the program’s state at the moment.

**Question 7.4.3: Has the processor paused just before, or just after executing the line with the breakpoint?**

The processor has paused just before executing the line with the breakpoint:

- Breakpoint Functionality: When a breakpoint is set, the execution of the program will halt right before the line of code containing the breakpoint is executed. This allows the programmer to inspect the state of the program (variables, memory, etc) at that precise moment.

- Execution Flow: The typical flow of execution means that the program runs sequentially, and when it reaches a breakpoint, it pauses before executing that specific line.

***E. Part 7.5: Registers and Basic Operations***

A white card with black text

Description automatically generated**Question 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?**

In my opinion, this code is going to do following steps:

+ Line 1: ‘MOV R0, #1’ initializes register R0 with the value 1.

+ Line 2: ‘ADD R1,R0,#8’ computes 1 + 8, resulting in 9, and stores it in R1.

+ Line 3: ‘ADD R2,R1,#100’ computes 9 + 100, resulting in 109, and stores it in R2.

+ Line 4: ‘SUB R3,R2,#25’ computes 109 - 25, resulting in 84, and stores it in R3.

+ Line 5: ‘HALT’ means the end of the program.

A blue screen with white text

Description automatically generated **Question 7.5.2: When the program is complete, take a screen shot of the register table showing the values.**

**Question 7.5.3: 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)**

A screenshot of a computer

Description automatically generated

**Question 7.5.4:  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.**

A blue screen with white text

Description automatically generatedA blue screen with white text

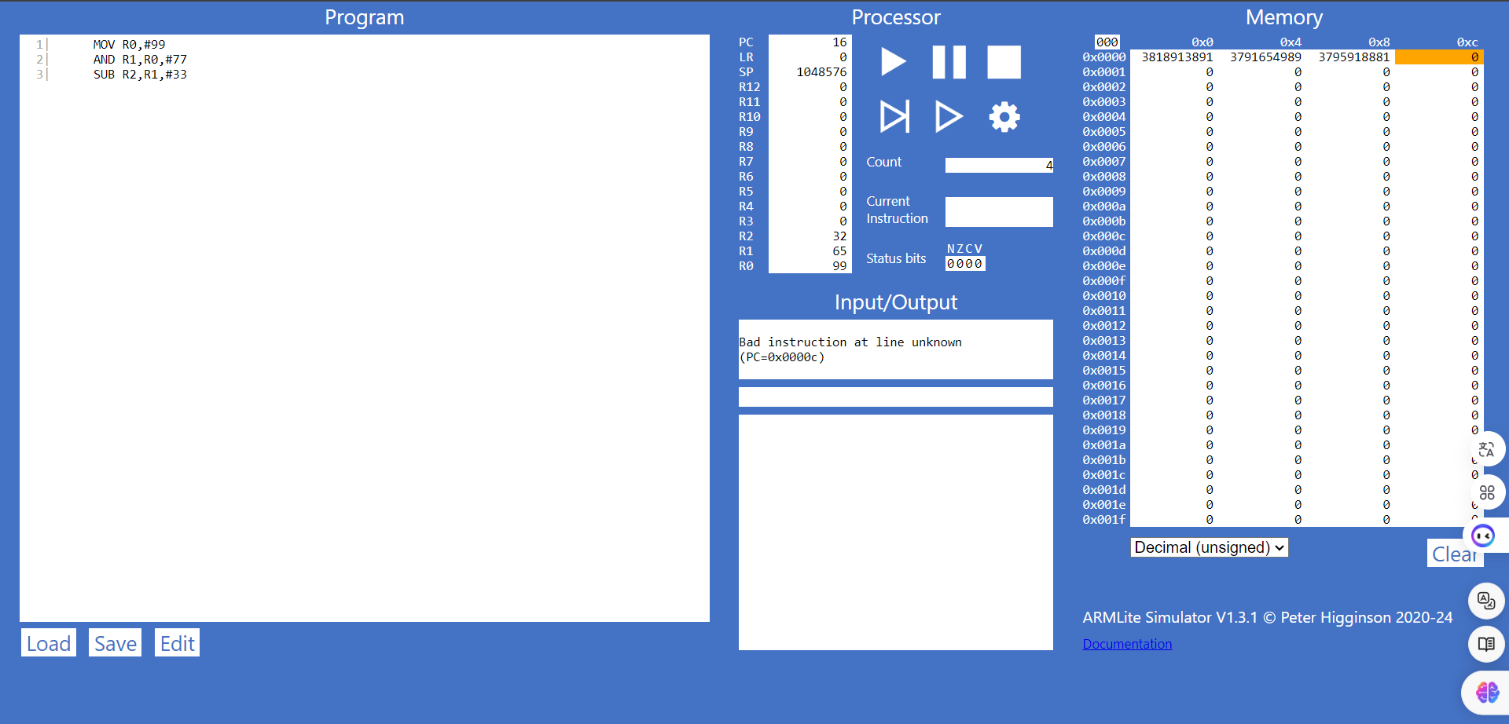
Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Decimal value of the destination register after executing this instruction** | **Binary value of the destination registers after executing this instruction** |
| MOV R0, #15 | 15 | 0000 1111 |
| AND R1, R0, #3 | 3 | 0000 0011 |
| ORR R2, R1, #12 | 15 | 0000 1111 |
| EOR R3, R2, #5 | 10 | 0000 1010 |
| LSL R4, R3, #1 | 20 | 0001 0100 |
| LSR R5, R4, #2 | 5 | 0000 0101 |

A screenshot of a computer

Description automatically generatedA blue screen with white text

Description automatically generated **Task 7.5.5: Let’s 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).**

 **Task 7.5.6:  Let's play again! Six initial numbers are: 99, 77, 33, 31, 14, 12 and your target number is: 32**

***F. Part 7.6: Signed Integers***

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

**-** Code explaination:

+ “MOV R0, #9999” means that load value 9999 into register R0

+ “LSL R1, R0, #18” indidcates that this intsruction performs a logical shift left: [R1 = R0\*218= 9999\*262144 = 2621415424]

+ “HALT” means end:

- Why R1 is negative:

+ Exceeding Maximum Value: The calculated value (2621415424) exceeds the maximum limit for a signed 32-bit integer, which is 2,147,438,647.

+ Two’s Complement Representation: In a 32-bit signed integer format, the highest bit (bit 31) is the sign bit. If this bit is set to 1, the number is interpreted as negative.

**Question 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 binary representation of decimal numbers:

\* 1: “00000000 00000000 00000000 00000001”

\* -1: “11111111 11111111 11111111 11111111”

\* 2: “00000000 00000000 00000000 00000010”

\* -2: “11111111 11111111 11111111 11111110”

- Pattern notice:

+ For positive numbers (1 and 2), the most significant bit (MSB) is 0, indicating a positive sign.

+ For negative numbers (-1 and -2), the most significant bit (MSB) is 1, indicating a negative sign

+ The remaining bits represent the magnitude of the number

**Question 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.**

A computer screen with a blue and white screen

Description automatically generatedA computer screen shot of a blue screen

Description automatically generated