**Faculty of Science, Engineering and Technology**



**Computer Systems**

***Week 7***

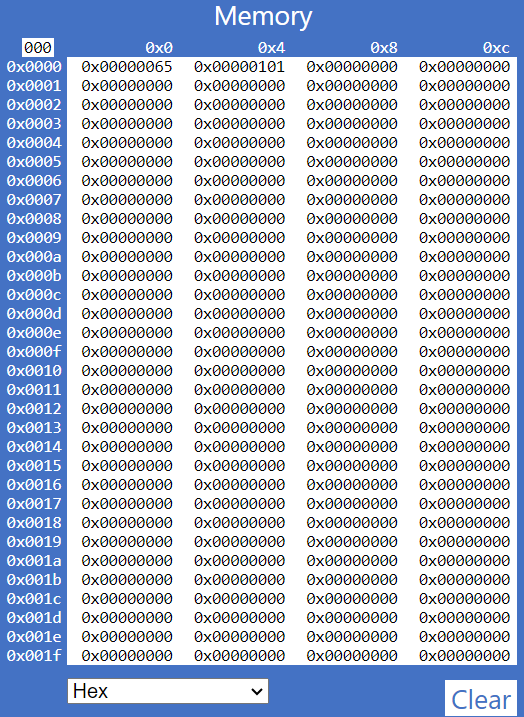
***Part 7.1***

A screenshot of a computer screen

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

==> The simulator interprets the input as a decimal integer and converts it to its hexadecimal representation. The decimal number 101 is equal to 65 in hexadecimal.

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



==> When entering a value prefixed with 0x, the simulator recognizes it as a hexadecimal input. The hexadecimal 0x101 is equal to 257 in decimal, but it will display as 0x00000101 to fit into the 32-bit word format.

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

A screenshot of a computer code

Description automatically generatedA screenshot of a computer

Description automatically generated

==> The prefix 0b indicates that the input is a binary number. The binary 101 is equal to 5 in decimal, which is represented as 0x00000005 in hexadecimal.

- Tooltip: When you hover over any memory word where you have entered a value, the tooltip will typically show the value in both decimal and hexadecimal formats. This provides a quick reference for understanding how the value is represented in different numeral systems.

\* When you hover over any of the memory words after changing the display to Decimal (unsigned), the tooltip will now show the value in both decimal and hexadecimal formats.

***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?***

==> No, changing the representation of the data in memory does not change the representation of the row and column headers. Ideally, it would be beneficial for the row and column headers to reflect the same base as the displayed data for consistency and clarity. However, in many simulators, the headers remain constant (often in hexadecimal) to maintain a standard reference point, especially since hexadecimal is commonly used in low-level programming and debugging.

***Part 7.2***

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

==> The column header memory address offsets go up in multiples of 0x4 because each memory word in the ARMLite simulator is 32 bits (or 4 bytes), and the simulator uses byte addressing, where each address points to a single byte of memory. This means that each word occupies 4 consecutive bytes in memory, and the address of subsequent words will be spaced by 4 bytes.

***Part 7.3***

A screenshot of a computer

Description automatically generated ***7.3.1 Take a screen shot 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.***

Line Numbers:

- Added for easier navigation and reference in your source code.

- They do not form part of the actual code but help in discussions and debugging.

Tooltip Information:

- Hovering over a line shows a 5-digit hex value representing the memory address of that line in the simulator.

- This address indicates where the instruction is stored in memory.

Relation to Assemblers and Von Neumann Architecture:

- The assembler translates code into machine language, and the line numbers with memory addresses help track the mapping of code to executable instructions.

- In Von Neumann architecture, instructions and data share memory, making these addresses crucial for understanding code execution and debugging.

***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?***

==> 5-Digit Hex Value:

- Memory Address: Represents where the instruction or data is stored in memory, crucial for CPU operation.

- Instruction Mapping: Each assembly line corresponds to a machine instruction, aiding in debugging and optimization.

- Debugging Aid: Helps trace execution flow and identify issues during runtime.

\* Effects of Code Modifications

- Blank Lines: Ignored by the assembler; do not affect instruction count or line numbers.

- Additional Spaces: Disregarded by the assembler; do not change execution or instruction count.

- Comments:

+ On their own line: Ignored; do not count as instructions.

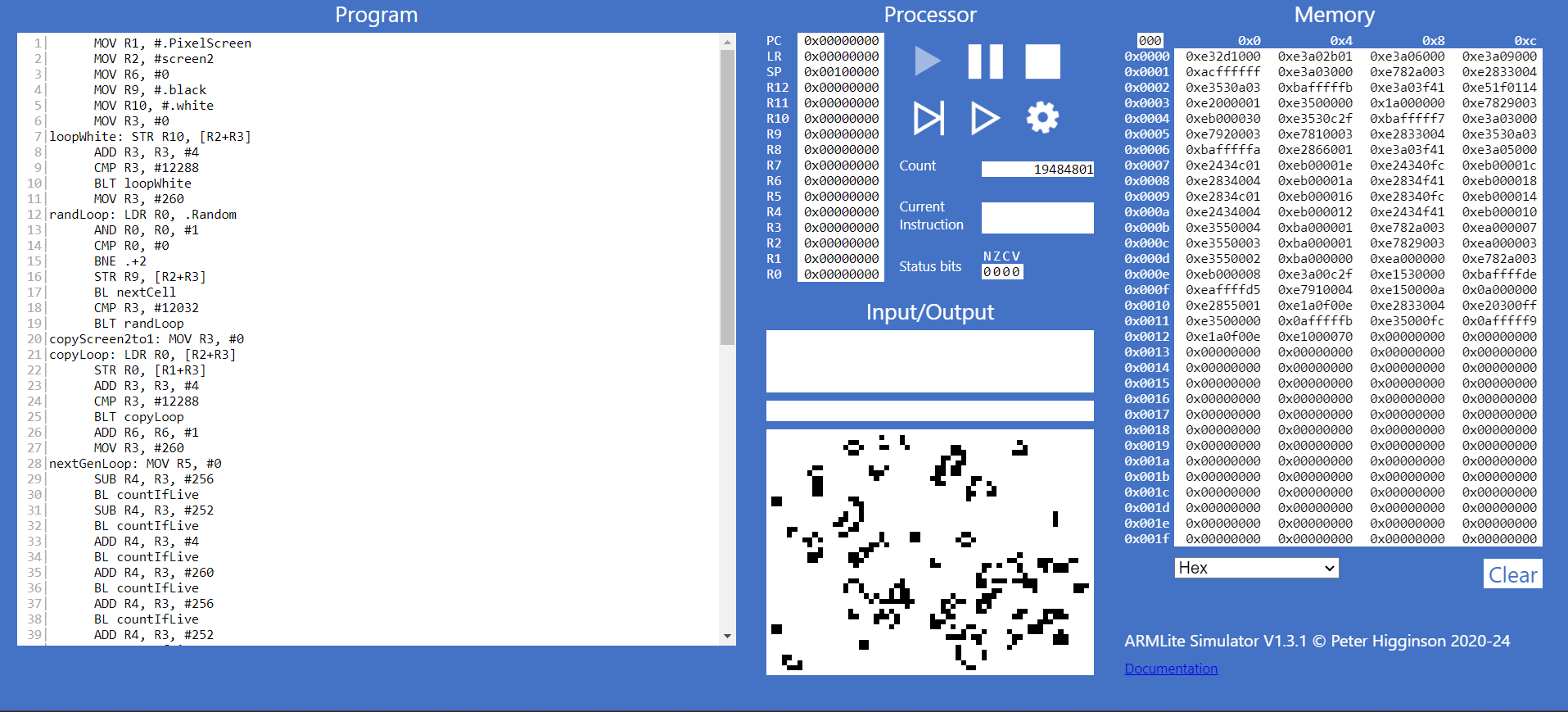
+ After instructions: Also ignored; maintain instruction count.

- Line Numbers: Adjust based on total lines, including blank lines and comments, but executable instruction count remains unchanged.

- Total Instructions: Reflects only executable code; blank lines and comments do not affect this count.

\* Removing the Comma: May cause a syntax error or alter instruction interpretation, depending on assembler rules.

Run the program:



***Part 7.4***

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

==> Significance of Highlighting

Program Window Highlighting:

- The highlighted section indicates the current instruction that the processor is executing or has just executed.

- This helps you identify which part of the code is being processed at the moment of pausing.

Memory Window Highlighting:

- The highlighted memory addresses represent the data being accessed or modified by the currently executing instruction.

- This allows you to observe how the program interacts with memory, which is crucial for debugging and understanding program behavior.

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

Immediate Outcomes:

- Execution of Current Instruction:

- The program executes the currently highlighted instruction in the Program window.

Program State Update:

- The state of the program, including variable values and memory contents, is updated to reflect the effects of the executed instruction.

Highlighting Shift:

- The highlighting moves to the next instruction in the sequence, preparing for the next execution step.

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

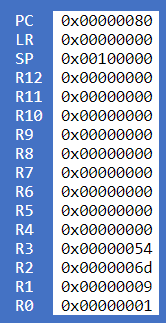
The processor has paused just after executing the line with the breakpoint. This allows you to inspect the program state immediately after the effects of that line have been applied.

***Part 7.5***

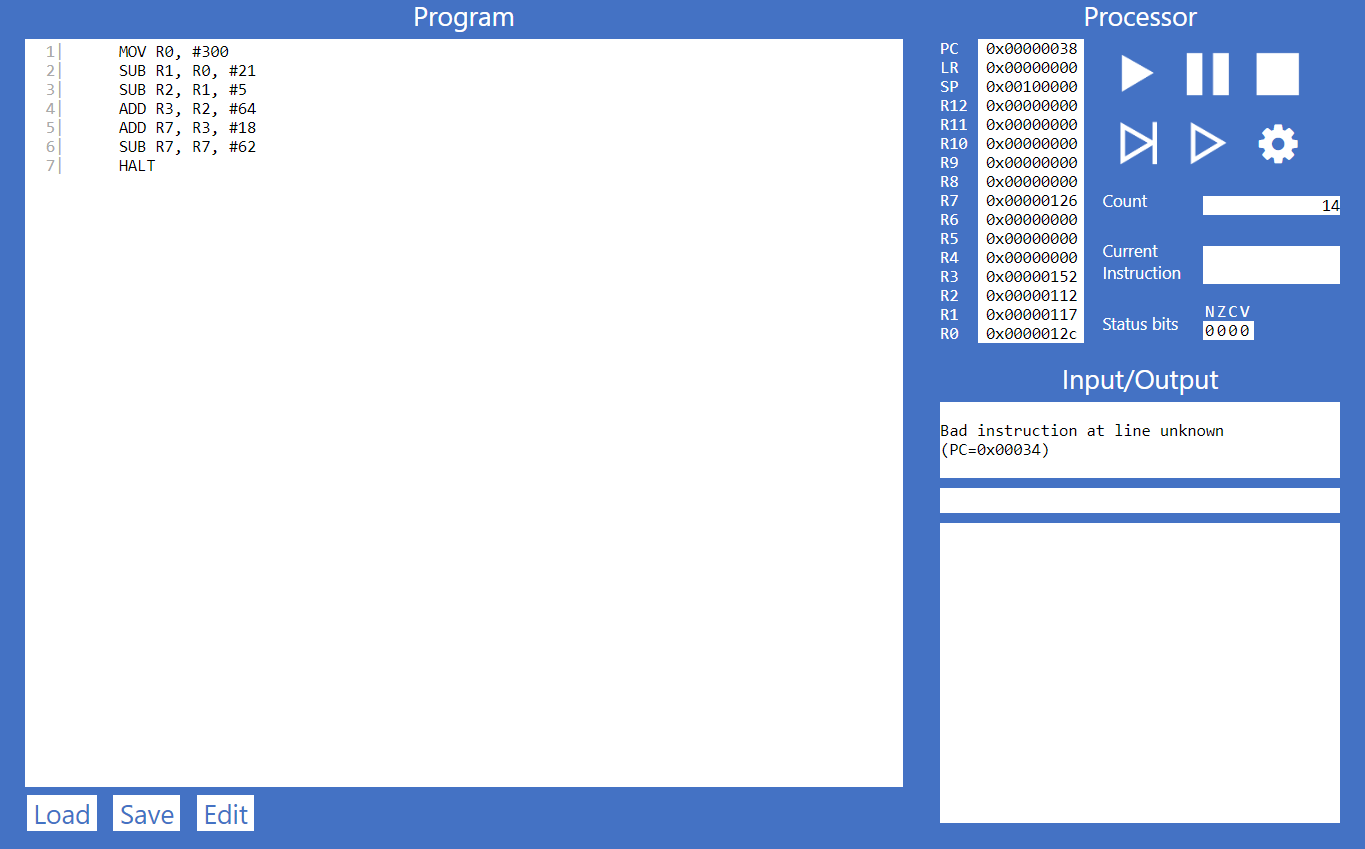
*****7.5.1 Before executing this instruction, describe in words what you think this instruction is going to do, and what values you expect to see in R0 and R1 when it is complete ?*****

This instruction takes the current value in register R0, adds the immediate value 8 to it, and stores the result in register R1. Essentially, it performs the operation ( R1 = R0 + 8 ).

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



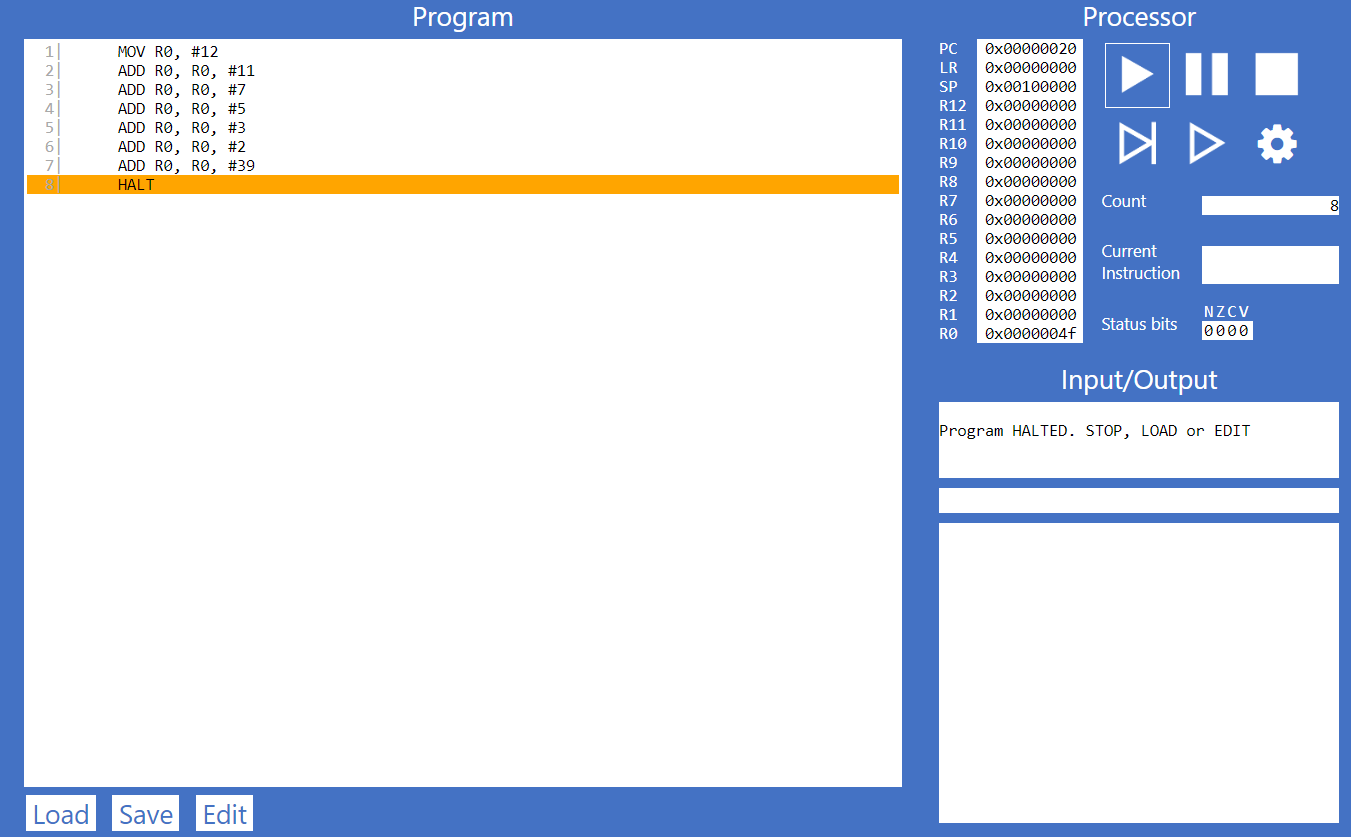
****7.5.3  Task:***Your 6 initial numbers are now 300, 21, 5, 64, 92, 18.   Write an Assembly Program that uses these values to compute a final value of 294  (you need only use MOV, ADD and SUB).  Place your final result in register R7  (don't forget the HALT instruction)***



****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.***

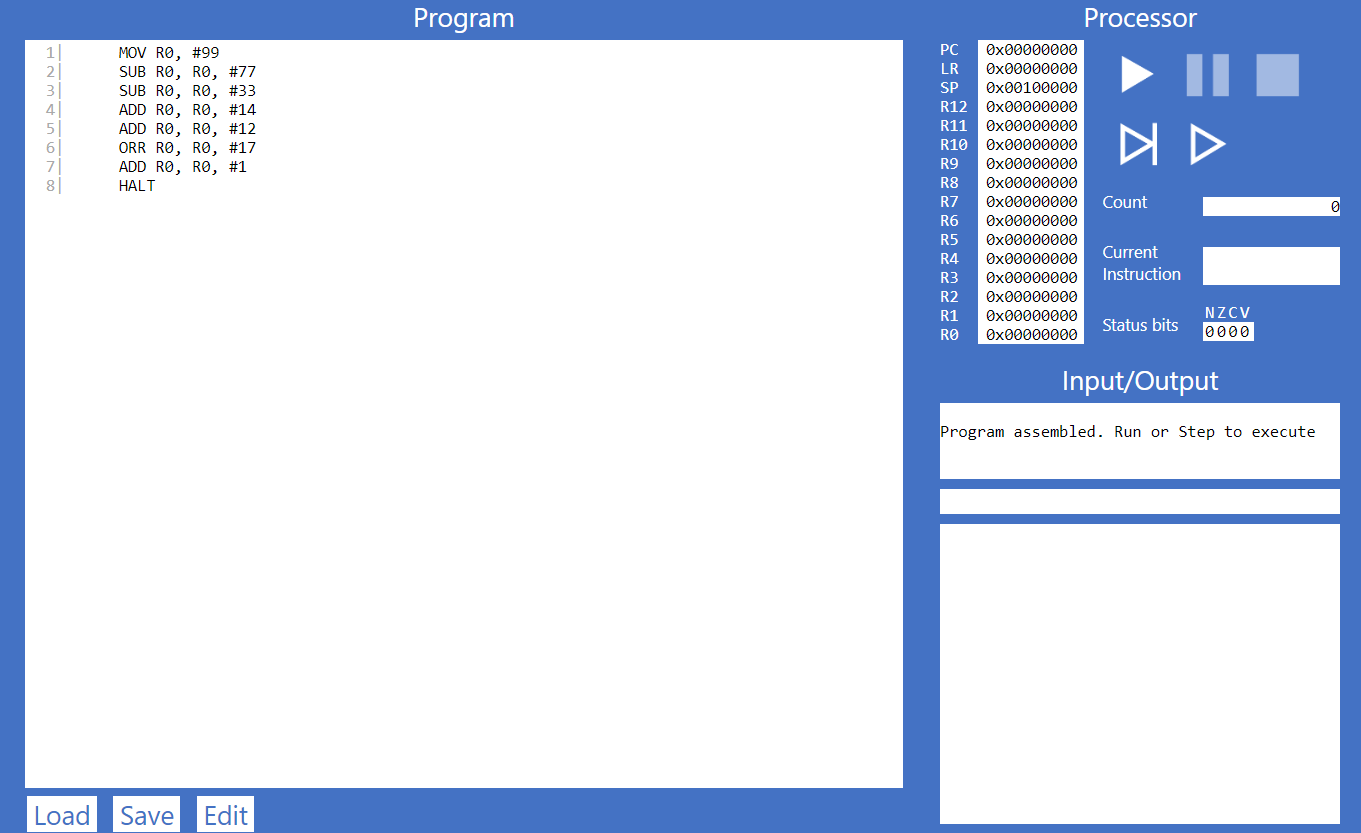
|  |  |  |
| --- | --- | --- |
| **Instruction** | **Decimal Value of the Destination Register** | **Binary Value of the Destination Register** |
| MOV R0, #50 | 50 | 00110010 |
| AND R1, R0, #25 | 16 | 00010000 |
| ORR R2, R0, #10 | 52 | 00110100 |
| EOR R3, R0, #30 | 44 | 00101100 |
| LSL R4, R0, #3 | 400 | 110010000 |
| LSR R5, R0, #1 | 25 | 00011001 |
| HALT | - | - |

****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).****



|  |  |  |
| --- | --- | --- |
| **Instruction** | **Decimal Value of the Destination Register** | **Binary Value of the Destination Register** |
| MOV R0, #12 | 12 | 00001100 |
| ADD R0, R0, #11 | 23 | 00010111 |
| ADD R0, R0, #7 | 30 | 00011110 |
| ADD R0, R0, #5 | 35 | 00100011 |
| ADD R0, R0, #3 | 38 | 00100110 |
| ADD R0, R0, #2 | 40 | 00101000 |
| ADD R0, R0, #39 | 79 | 01001111 |
| HALT | - | - |

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



| **Instruction** | **Decimal Value of the Destination Register** | **Binary Value of the Destination Register** |
| --- | --- | --- |
| MOV R0, #99 | 99 | 01100011 |
| SUB R0, R0, #77 | 22 | 00010110 |
| SUB R0, R0, #33 | -11 | 11110101 (two's complement) |
| ADD R0, R0, #14 | 3 | 00000011 |
| ADD R0, R0, #12 | 15 | 00001111 |
| ORR R0, R0, #17 | 31 | 00011111 |
| ADD R0, R0, #1 | 32 | 00100000 |
| HALT | - | - |

***Part 7.6***

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

==> The result in register R1 is a negative decimal number because of how the ARM architecture represents signed integers using two's complement.

When the code executes the instruction LSL R1, R0, #18, it shifts the value in R0 (9999) left by 18 bits, effectively multiplying it by (2^{18}). This results in a large positive number, specifically 2621440000.

However, in a 32-bit signed integer format, the binary representation of this number starts with a 1, indicating it is treated as a negative number due to the two's complement system. To interpret it as a negative value, you can invert the bits and add 1, which reveals that the value in R1 corresponds to -167772160 in decimal.

*****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.*****

|  |  |
| --- | --- |
| **Decimal** | **Binary Representation** |
| 1 | 0000 0001 |
| -1 | 1111 1111 |
| 2 | 0000 0010 |
| -2 | 1111 1110 |

==> These patterns can be summarized as follows:

- Positive numbers: MSB is 0, last bit is 0

- Negative numbers: MSB is 1, last bit is 1

These patterns are essential in understanding how signed decimal numbers are represented in binary using two's complement, and they will be useful in further analysis and calculations.

*****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.*****

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

