Eugene Ngo

EE 469

May 19, 2022

Lab 4 Report

**Procedure:**

**Task #1: Introduction to Assembly Programming**

**Part 1: Writing your first Assembly Program**

1. *Explain what these lines mean:*  
   

‘.text’ tells the assembler and linker that this section is a chunk of code and should be labeled as such in the memory / building process. This makes it both readable and executable memory and, in some cases, / systems, writable.

‘.align=2’ means that the next variable or instruction is going to fall on a memory address that is divisible by 2X with .align=X. So for our system, the 1st instruction is going to fall on a memory address divisible by 22, aka 4 because we are on a 32-bit or 4-byte system.

1. *What is the value of R0, R1, R2, and PC at the start and at the end of the program?*

As shown in the figure below, R0 = 4, R1 = 5, R2 = 9, and PC = 0xc at the end of the program.

A screenshot of a computer

Description automatically generated with medium confidence

*Figure 1: Registers at the end of Q2*

1. *Explain the S: B S line of code (lines 10 and 11)*

The S: B S is essentially just a never ending loop, the same as while(1); in C

1. *Expand the program to solve 4+5+9-3 and save the result in the 40th word in memory. Take a screenshot of the memory for your lab report*

A screenshot of a computer

Description automatically generated with low confidence

*Figure 2: The memory after storing 15 in word 40*

The 40th word in memory is 0x9c because each word is 4 bytes and the 0x9c  
counter is how many bytes we are at. 0x9c / 4 = decimal 39 aka the 40th word in  
memory as the 1st word is stored at address 0. The 40th word is stored in 0x9c  
through 0x9F for 4 bytes.

**Part 2: Tracing an Assembly Program**

1. The value in R0 after the program ends is N! With N = the initial value of R0. So for R0=3 initially, the program ends with R0=6.
2. If R0 =5 initially, R0 = 120 when the program ends.
3. This program computes the factorial of the initial value of R0 and stores it back into R0 at the end.
4. After replacing these lines, the program essentially never ends and the stack pointer begins to run out of bounds because the link register never changes value. So the program continues popping from the stack and eventually starts to pull garbage data from other memory sections of the hardware.
5. a. In this scenario, R3 starts at 0 and the LR works similar to the original program, but this time we are performing 1\*XN where N is equal to the initial value of R0 and X is equal to the value of R1. For the case where R1 = 0, we get 1\*0N which will result in 0 being stored into R0.

b. This scenario is slightly different because the line ADD SP, SP #8 wasn’t changed. So by the time we get to POP {LR}, the stack pointer is 1 word off from where it should be and so we get errors and warnings about the function messing up the stack pointer. The program eventually works fine if starting with clean working registers but it could be bad if the stack pointer pulls data from outside the stack frame.

c. Now by deleting the stack pointer modification line, we are still going to get issues because PUSH is still being used without the stack pointer being correctly modified. We will again get many errors about the stack pointer being incorrect due to our function, which is essentially a misuse of the stack and could cause issues in a larger program.

**Task #2:**

Figure 3 shown below shows the pseudocode of my design steps

A picture containing text, handwriting, letter, calligraphy

Description automatically generated

*Figure 3: Pseudocode Procedure for Task 2*

**Task #3:**

The first step of this task is the following hand analysis:

1. 2.0 → 0 10000000 00000000000000000000000 → 0x40000000
2. 3.5 → 0 10000000 11000000000000000000000 → 0x40600000
3. 0.50390625 → 0x3f010000
4. 65535.6875 → 0x477fffb0
5. The sum of c and d is found by doing the same process that we did in lecture just with  
   larger numbers. Exponent of c = -1 without bias, exponent of d is 15, mantissa of C is = 1000000100000000000000002 , mantissa of D is = 1111111111111111101100002. This means mantissa of C needs to be shifted over by 16 to the right → mantissa of C\_shifted = 000000000000000100000012 and the exponent for the result is set to 15. The sum of the mantissas is now 0001 0000 0000 0000 0000 0011 00012. This result has to be normalized so the mantissa → 0001 0000 0000 0000 0000 0011 0002 and the exponent rises to 16. The result can now be rewritten as exponent = 143 with bias = 100011112 and mantissa is the sum above. This gives a number of 0 10001111 000 0000 0000 0000 0001 10002 = 0x47800018 or 65536.1875 converted into human view-able decimal. If we do this manually, we get 65535.6875+0.50390625 = 65536.1914063 so our answer is a rounding point from a more precise version.

After completing the hand analysis section, I began writing code. I essentially followed the steps that we normally do to add positive floating point IEEE numbers but in assembly. These steps are the same as outlined in the “The Algorithm” section of the spec. Throughout the code, as you will see in the appendix, the procedure of the code is walked through step-by-step in my comments. After a lot of time editing, I managed to reduce my code down to less than 50 lines of code (not including comments, headers, parameters, etc). Then I tested the code before submitting.

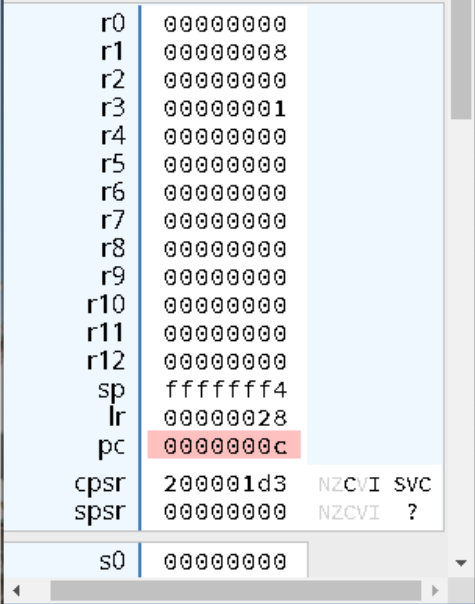
**Results**

**Task #1:**

This task was following instructions so the results section is pointless as the results of it was shown in the procedure above.

**Task #2:**

Testing my program on a few numbers, it successfully counted the number of bits that were set.



*Figure 4: Output for 0xFF000000*

Figure 4 above shows that the output counted number of bits when using 0xFF000000 as an input which has 8 bits set. R1 holds our answer at the end of the program, showing that 8 bits are counted. Similarly, Figure 5 below shows the output when the input is 0xFFFFFFFF as 32 bits which is stored in R1.

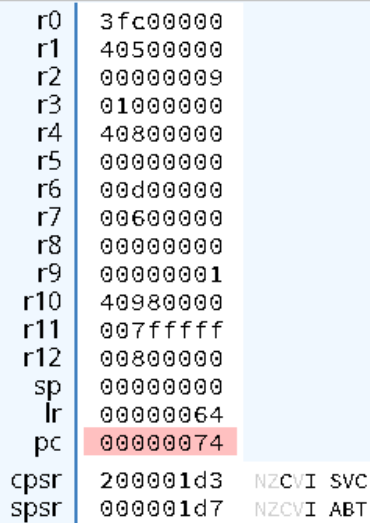
A screenshot of a computer

Description automatically generated

*Figure 5: Output for 0xFFFFFFFF*

**Task #3:**

For task 3, my results can be seen below by using the test case done in lecture of adding together 0x3FC00000 and 0x40500000. I expect to see 0x40980000 as the final result.



*Figure 6: Output of adding 0x3FC00000 & 0x40500000*

The output is stored in R10 and as we can see it is 0x40980000 which was as expected. The mantissas, with leading 1s, are stored in R6 and R7. One of the exponents is stored in R4 (shifted and with bias of +127, without those it is equal to exponent = 2 which is correct). The operands are stored in R0 and R1. That 1-bit being set in R3 indicates that the result was normalized.  
To double check, I also tested this on the operation done during the hand analysis and got the correct result.

**Appendix**

Lab4\_Task1\_Part1.s  
A screenshot of a computer

Description automatically generated

Lab4\_Task1\_Part2.s  
A screenshot of a computer program

Description automatically generated with medium confidence

Lab4\_Task2.s

A screenshot of a computer program

Description automatically generated with medium confidence

Lab4\_Task3.s

A screenshot of a computer program

Description automatically generated with medium confidence