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
.align=2
```

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

2. *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.

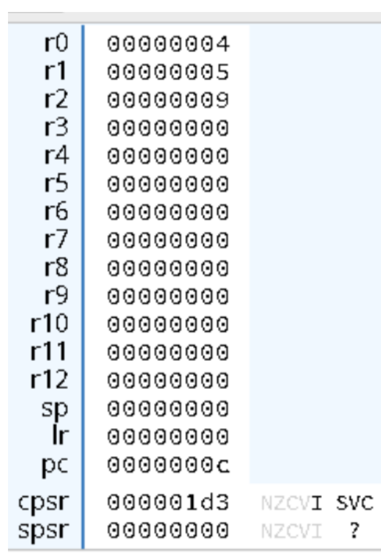
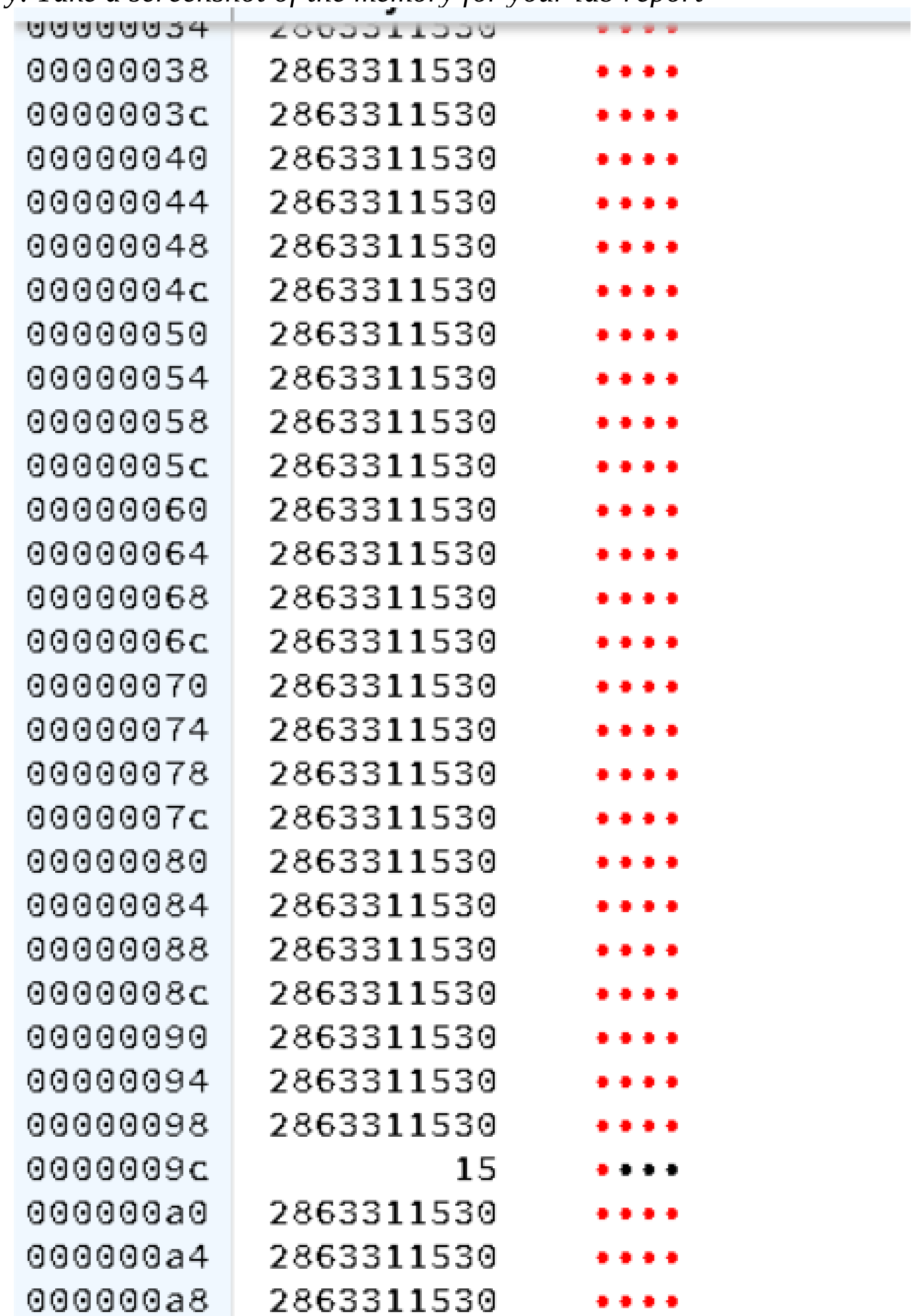


Figure 1: Registers at the end of Q2

3. *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
4. *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*



00000034	2863311530	••••
00000038	2863311530	••••
0000003c	2863311530	••••
00000040	2863311530	••••
00000044	2863311530	••••
00000048	2863311530	••••
0000004c	2863311530	••••
00000050	2863311530	••••
00000054	2863311530	••••
00000058	2863311530	••••
0000005c	2863311530	••••
00000060	2863311530	••••
00000064	2863311530	••••
00000068	2863311530	••••
0000006c	2863311530	••••
00000070	2863311530	••••
00000074	2863311530	••••
00000078	2863311530	••••
0000007c	2863311530	••••
00000080	2863311530	••••
00000084	2863311530	••••
00000088	2863311530	••••
0000008c	2863311530	••••
00000090	2863311530	••••
00000094	2863311530	••••
00000098	2863311530	••••
0000009c	15	••••
000000a0	2863311530	••••
000000a4	2863311530	••••
000000a8	2863311530	••••

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 $1*X^N$ 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 $1*0^N$ 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

Count 1s in a 32bit #

- Load 32 bit # into register
  Create a loop that runs 32 cycles
  ↳ like a java or c for loop! for (int i=0; i<32; i++){
  }
- Inside the for loop is checking each least significant digit
  ↳ then shift the number to the right by 1 which is allowing us to check every bit in the number
- For each time that the bit is checked using CMP bitMask(#), #1 we'll need to increment a sum variable which'll be stored in a register

e.g.
```
for(int i=0; i<32; i++){
    if(# & 1){
        sum++
    }
}
```

- Using the above concept in assembly instead of C or java
- Then test on CPUlator and record the results

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 = 100000010000000000000000002 , mantissa of D is = 111111111111111101100002. 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.

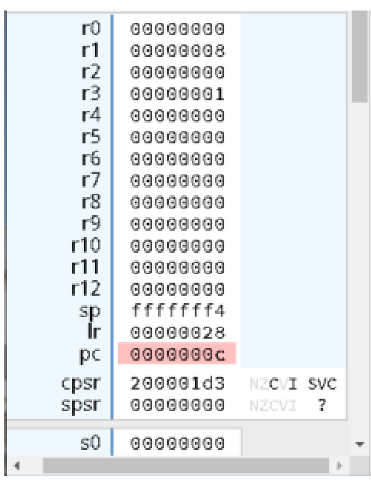


Figure 4: Output for 0xFF000000

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.

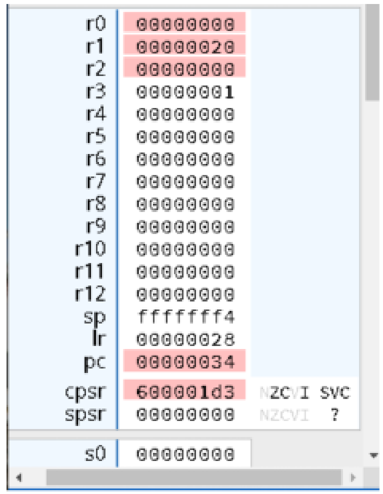


Figure 5: Output for 0xFFFFFFFF

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.

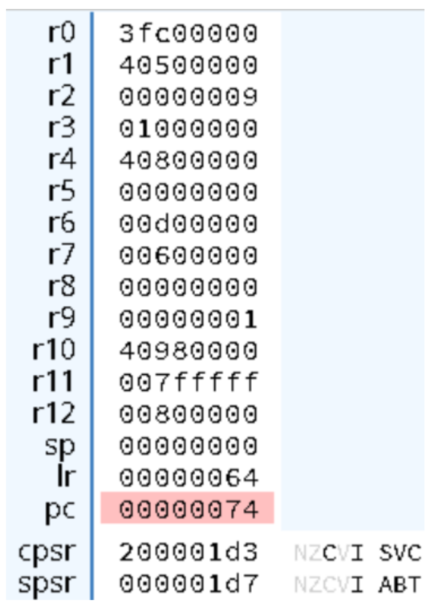


Figure 6: Output of adding 0x3FC00000 & 0x40500000

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

```
//Eugene Ngo
//EE 469
//Lab 4
//Task 1 Part 1

.global _start
_start:
    mov r0, #4
    mov r1, #5
    add r2, r0, r1
    add r3, r2, r0
    add r3, r3, r1
    sub r3, r3, #3
    mov r4, #39
    mul r4, r4, r0
    str r3, [r4]
S:
    B S
.end

```

Lab4_Task1_Part2.s

```
//Eugene Ngo
//EE 469
//Lab 4
//Task 1 Part 2

.global _start
_start:
    MOV R0, #5
LOOP:
    PUSH {LR}
    CMP R0, #1
    BGT ELSE
    MOV R0, #1
    MOV PC, LR
ELSE:
    SUB R0, R0, #1
    BL LOOP
    POP {LR}
    MUL R0, R1, R0
    MOV PC, LR
.end
```

Lab4_Task2.s

```
//Eugene Ngo
//EE 469
//Lab 4
//Task 2

//This part counts the number of bits that are set within a 32-bit
//operand stored in R0 and the output is stored in R2
.global _start
_start:
    MOV R0, #0xFF000000
    MOV R1, #0              //count of 1s
    MOV R2, #32             //loop index, will break the loop when R2 == 0
LOOP:
    CMP R2, #0
    BEQ END                 //end loops if R2 is 0, else decrement
    SUB R2, R2, #1
    BIC R3, R0, #0xFFFFFF   //clear all but last bit
    LSR R0, #1              //shifts all bits right by 1
    CMP R2, #0              //determines if last bit is 0 or 1
    BLNE INCREMENT          //then increment
    B LOOP;                 //continue loop
INCREMENT:
    ADD R1, R1, #1
    MOV PC, LR;             //go back

END:
    NOP
.end
```

Lab4_Task3.s

```
//Eugene Ngo
//EE 469
//Lab 4
//Task 3

//This part performs floating-point addition.
//The two operands are set in the parameters below and the
//result is stored in R10
leading1Mask: .word 0x800000
operand1: .word 0x3FC00000
operand2: .word 0x40500000
mantissaMask: .word 0x7FFFFF
.global _start
_start:
    LDR R0, operand1            //floating point operand 1
    LDR R1, operand2            //floating point operand 2
    LDR R12, leading1Mask       //for adding leading 1
    LDR R11, mantissaMask       //mantissa mask 23 lower bits
    LSR R4, R0, #23             //get exponent 1 into R4
    AND R4, R4, #0xFF           //last 8 bits are exponent
    SUB R4, R4, #127            //remove bias on exponent 1
    LSR R5, R1, #23             //get exponent 2 into R5
    AND R5, R5, #0xFF           //last 8 bits are exponent
    SUB R5, R5, #127            //remove bias on exponent 2
    AND R6, R0, R11             //put mant1 into R6
    ORR R6, R6, R12             //leading 1s
    AND R7, R1, R11             //put mant2 into R7
    ORR R7, R7, R12             //leading 1s
    SUBS R9, R4, R5             //store difference in exponent in R9
    BLMI NEG_EXP                //negative exp difference which swaps values in the exp and mant regs and twos complement of exp_diff
    LSR R7, R9                  //shift lesser mantissa by exp diff
    ADD R10, R6, R7             //sum mantissas
    ADD R3, R10, #0x1000000     //overflow bit of sum
    CMP R3, #0
    BLGT NORMALIZE              //if overflow then normalize
    ADD R4, R4, #127            //add bias onto exponent
    LSL R4, R4, #23             //shift exponent up 23 bits
    ORR R10, R10, R4            //put exponent into result word
    BIC R10, R10, #0x1000000    //sign bit is 0
    B _start

NORMALIZE:
    LSR R10, #1
    ADD R4, R4, #1              //inc exponent
    MOV PC, LR

//swap registers and do twos complement on difference in exponents
NEG_EXP:
    PUSH {R4}
    MOV R4, R5
    POP {R5}
    PUSH {R6}
    MOV R6, R7
    POP {R7}                    //swap registers for mant and expo
    MVN R9, R9                  //twos complement of r9 for expo
    ADD R9, R9, #1
    MOV PC, LR
.end
```