Kiana Peterson and Nattapon Oonlmaom
EE 469
May 19, 2023
Lab 4: Assembly Programming

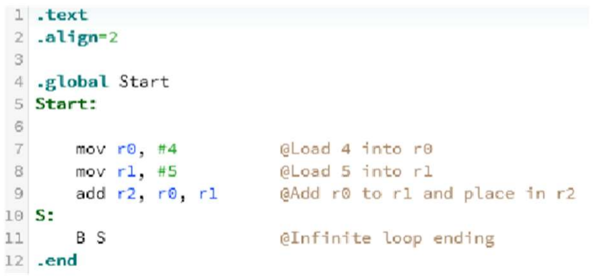
## Procedure

The purpose of this lab is to be practice using assembly programming using ARMv7 in CPUlator  to accomplish Tas#1, #2, and #3. The objective of this lab is to be familiarized with the layout of CPUlator, program in ARM assembly language, and understand Floating Point Addition.

## Task #1:

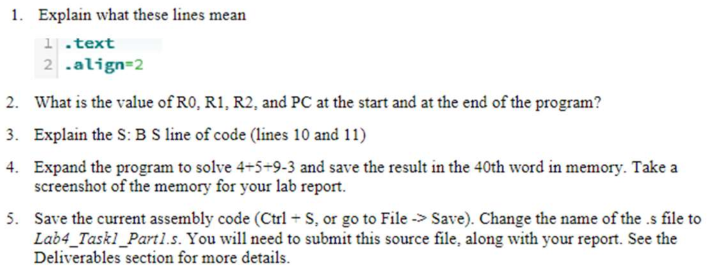
*Part 1*

The first part of Task #1 was to type the given program in Figure 1.1 into the CPUlator, assemble the program, make sure there is no errors, modify the code as instructed, and use the "Step Into" function to watch the values of register to answer the given questions in Figure 1.2. The code was then saved as Lab4_Task1_Part1.s. The answers can be found under the result section.



```
1  .text
2  .align=2
3
4  .global Start
5  Start:
6
7      mov r0, #4          @Load 4 into r0
8      mov r1, #5          @Load 5 into r1
9      add r2, r0, r1      @Add r0 to r1 and place in r2
10 S:
11     B S                 @Infinite loop ending
12 .end
```

Figure 1.1 Simple Program that Adds 2 Numbers



1. Explain what these lines mean

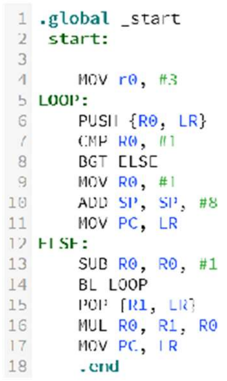
```
1  .text
2  .align=2
```

2. What is the value of R0, R1, R2, and PC at the start and at the end of the program?
3. Explain the S: B S line of code (lines 10 and 11)
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.
5. Save the current assembly code (Ctrl + S, or go to File -> Save). Change the name of the .s file to *Lab4_Task1_Part1.s*. You will need to submit this source file, along with your report. See the Deliverables section for more details.

Figure 1.2 Questions to be Answer

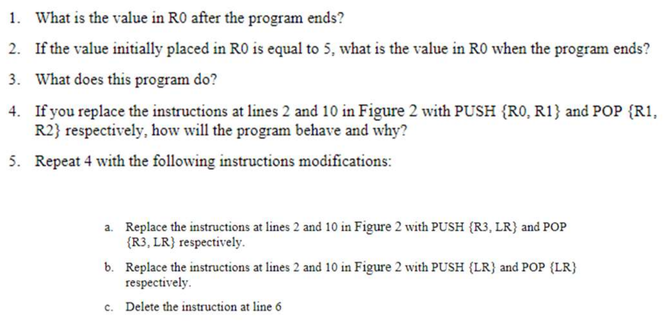
*Part 2:*

The second part of the task was to open a new CPUlator instance, assemble and debug the new given program in Figure 1.3. Then, follow the execution in the debugger and modify the code as instructed to answer the given question in Figure 1.4. The code was then saved as Lab4_Task1_Part2.s. The answers can be found under the results section.



```
1  .global _start
2   start:
3
4      MOV r0, #3
5  LOOP:
6      PUSH {R0, LR}
7      CMP R0, #1
8      BGT ELSE
9      MOV R0, #1
10     ADD SP, SP, #8
11     MOV PC, LR
12 ELSE:
13     SUB R0, R0, #1
14     BL LOOP
15     POP {R1, LR}
16     MUL R0, R1, R0
17     MOV PC, LR
18     .end
```

Figure 1.3 Assembly Program

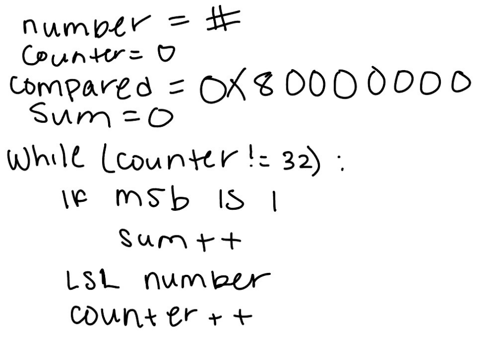


1. What is the value in R0 after the program ends?
2. If the value initially placed in R0 is equal to 5, what is the value in R0 when the program ends?
3. What does this program do?
4. If you replace the instructions at lines 2 and 10 in Figure 2 with PUSH {R0, R1} and POP {R1, R2} respectively, how will the program behave and why?
5. Repeat 4 with the following instructions modifications:
    a. Replace the instructions at lines 2 and 10 in Figure 2 with PUSH {R3, LR} and POP {R3, LR} respectively.
    b. Replace the instructions at lines 2 and 10 in Figure 2 with PUSH {LR} and POP {LR} respectively.
    c. Delete the instruction at line 6

Figure 1.4 Task1 P2 Questions to Answer

**Task #2:**

The second task was to design an algorithm for counting the number of 1's in a 32-bit number. The approach taken was first design a pseudocode found in Figure 1.5. Then, implement the pseudocode using ARMv7 before testing the program in CPUlator.



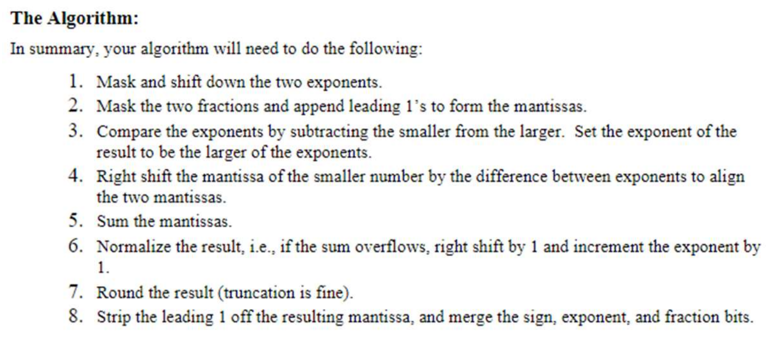
```
number = #
Counter= 0
compared = 0X80000000
 Sum = 0
While (counter != 32) :
    If msb is 1
        sum++
    LSL number
    counter ++
```

Figure 1.5 Pseudocode for Task#2

**Task #3:**

The third task was to write an ARM assembly language function that performs floating-point addition that follows the algorithm in Figure 1.6. Note that the code will add only strictly positive numbers and he sign bits in the numbers being summed can be ignored. The sign bit of the resulting sum should be set to zero.

The approach to this task was to first study and unterstant the floating-point arithmetic. Then, a pseudocode was written and implemented into the CPUlator. Then, the Step In founction was utalized to debug the code. Additionally, a hand analysis prompts were given and answered as a guiding questions. The hand analysis and its answers can be found in Figure 1.7. To test if the program is working properly, the register memory result in R0 was tested against the expected result after the program ended, following the mentioned algorithm. The registers were viewed and the register holding the sum of two floats was examined. The final is then saved as Lab4_Task3.s.

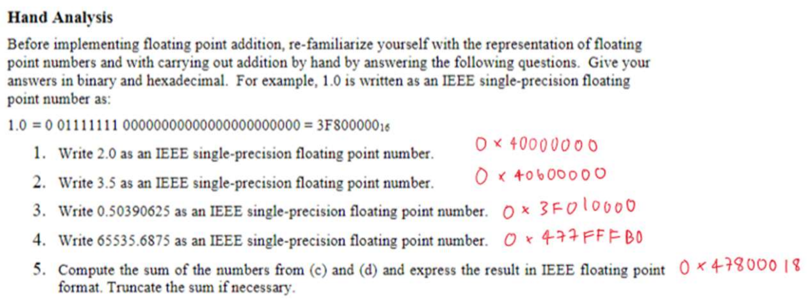


**The Algorithm:**

In summary, your algorithm will need to do the following:

1. Mask and shift down the two exponents.
2. Mask the two fractions and append leading 1's to form the mantissas.
3. Compare the exponents by subtracting the smaller from the larger. Set the exponent of the result to be the larger of the exponents.
4. Right shift the mantissa of the smaller number by the difference between exponents to align the two mantissas.
5. Sum the mantissas.
6. Normalize the result, i.e., if the sum overflows, right shift by 1 and increment the exponent by 1.
7. Round the result (truncation is fine).
8. Strip the leading 1 off the resulting mantissa, and merge the sign, exponent, and fraction bits.

Figure 1.6 The Algorithm for Floating-Point Addition



**Hand Analysis**

Before implementing floating point addition, re-familiarize yourself with the representation of floating point numbers and with carrying out addition by hand by answering the following questions. Give your answers in binary and hexadecimal. For example, 1.0 is written as an IEEE single-precision floating point number as:

1.0 = 0 01111111 00000000000000000000000 = $3F800000_{16}$

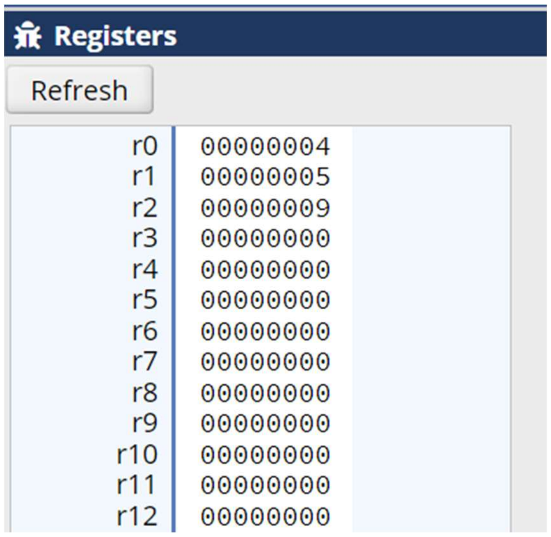
1. Write 2.0 as an IEEE single-precision floating point number. 0 × 40000000
2. Write 3.5 as an IEEE single-precision floating point number. 0 × 40600000
3. Write 0.50390625 as an IEEE single-precision floating point number. 0 × 3F010000
4. Write 65535.6875 as an IEEE single-precision floating point number. 0 × 477FFFB0
5. Compute the sum of the numbers from (c) and (d) and express the result in IEEE floating point format. Truncate the sum if necessary. 0 × 47800018

Figure 1.7 Hand Analysis (Answered)

**Results:**

**Task #1:**

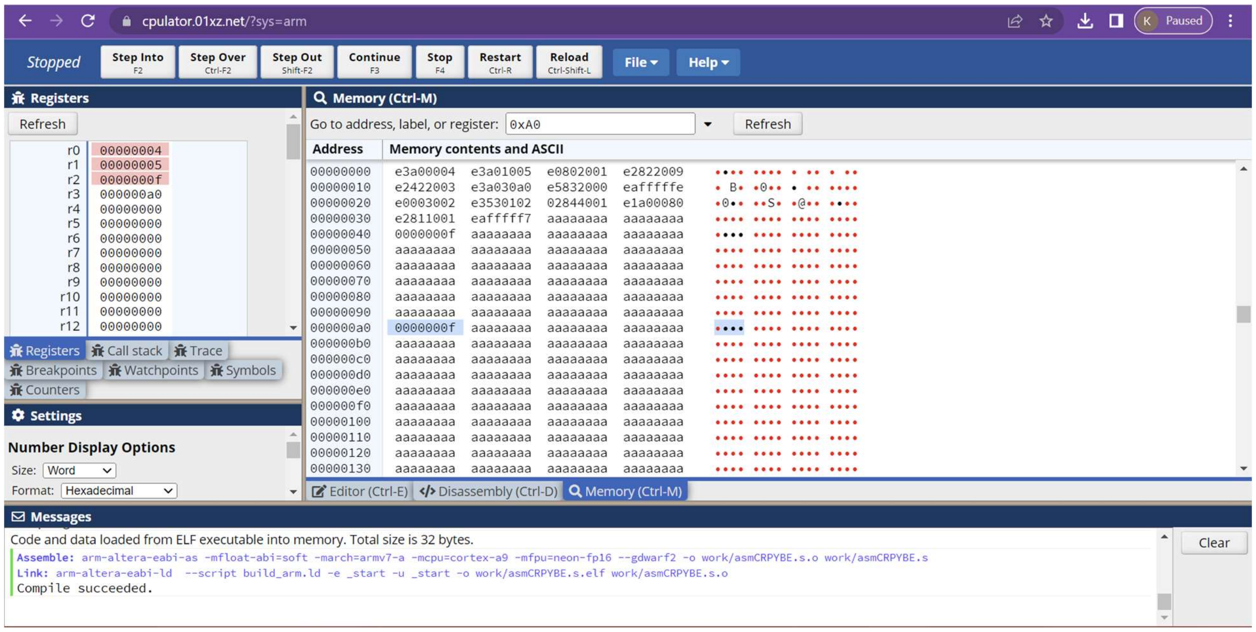
*Part 1*



```
Registers
r0   00000004
r1   00000005
r2   00000009
r3   00000000
r4   00000000
r5   00000000
r6   00000000
r7   00000000
r8   00000000
r9   00000000
r10  00000000
r11  00000000
r12  00000000
```

Figure 2.1 Register values for Task1 Part 1

1. R0 = 0, R1=0, R2=0, PC=0
   R0=4, R1=5, R2=9, PC=0xC
2. S branches to S which then branches back to S, creating an infinite loop
3.



```
Registers
r0   00000004
r1   00000005
r2   0000000f
r3   000000a0
r4   00000000
r5   00000000
r6   00000000
r7   00000000
r8   00000000
r9   00000000
r10  00000000
r11  00000000
r12  00000000
```

Number Display Options
Size: Word
Format: Hexadecimal

Memory (Ctrl-M)
Go to address, label, or register: 0xA0

```
Address   Memory contents and ASCII
00000000  e3a00004  e3a01005  e0802001  e2822009   •••• •••• • •• • ••
00000010  e2422003  e3a030a0  e5832000  eafffffe   • B• •0•• • •• ••••
00000020  e0003002  e3530102  02844001  e1a00080   •0•• ••S• •@•• ••••
00000030  e2811001  eafffff7  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000040  0000000f  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000050  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000060  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000070  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000080  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000090  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
000000a0  0000000f  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
000000b0  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
000000c0  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
000000d0  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
000000e0  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
000000f0  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000100  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000110  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000120  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
00000130  aaaaaaaa  aaaaaaaa  aaaaaaaa  aaaaaaaa   •••• •••• •••• ••••
```

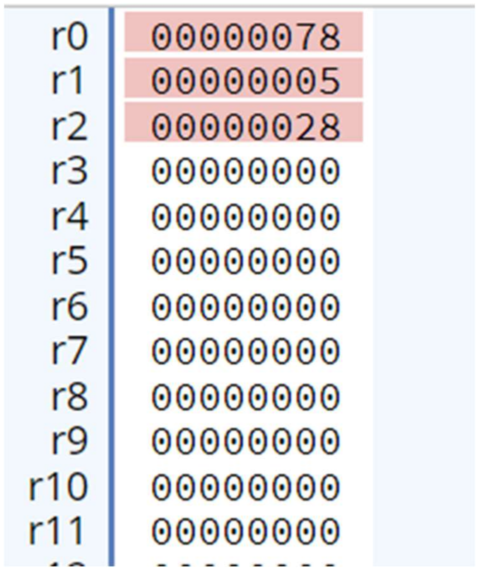
```
Code and data loaded from ELF executable into memory. Total size is 32 bytes.
Assemble: arm-altera-eabi-as -mfloat-abi=soft -march=armv7-a -mcpu=cortex-a9 -mfpu=neon-fp16 --gdwarf2 -o work/asmCRPYBE.s.o work/asmCRPYBE.s
Link: arm-altera-eabi-ld  --script build_arm.ld -e _start -u _start -o work/asmCRPYBE.s.elf work/asmCRPYBE.s.o
Compile succeeded.
```

Figure 2.2 Register and Memory Results for Extended Task #1 Part 1

*Part 2*

1. 6
2. 120
3. The program takes the factorial of r0

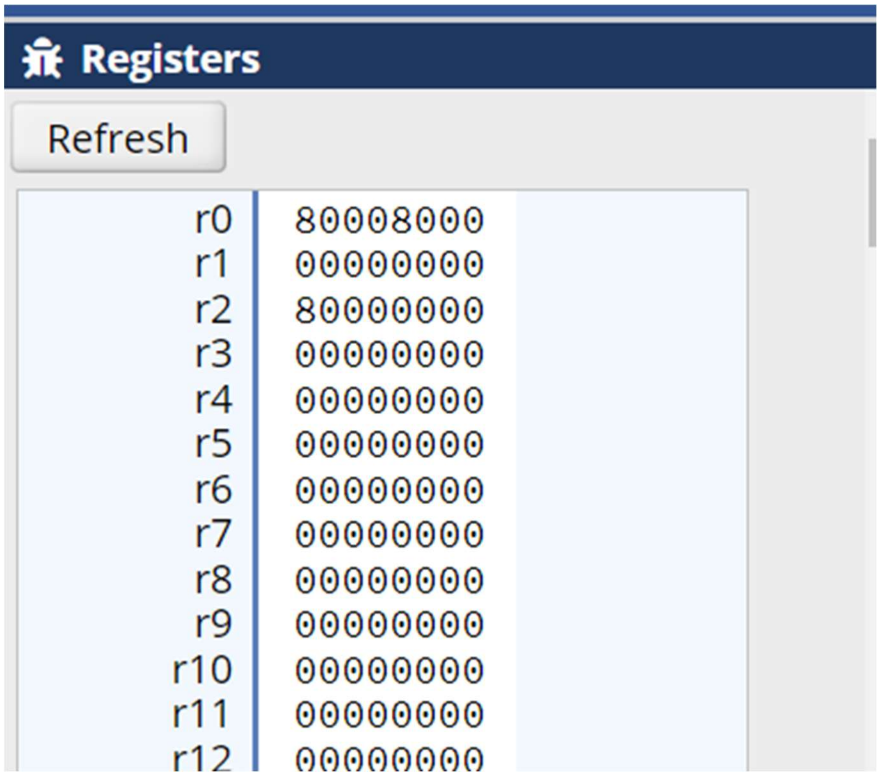
4. R0 will become 120 because it will have more iterations multiplying for each decrement of r0



```
r0   00000078
r1   00000005
r2   00000028
r3   00000000
r4   00000000
r5   00000000
r6   00000000
r7   00000000
r8   00000000
r9   00000000
r10  00000000
r11  00000000
```

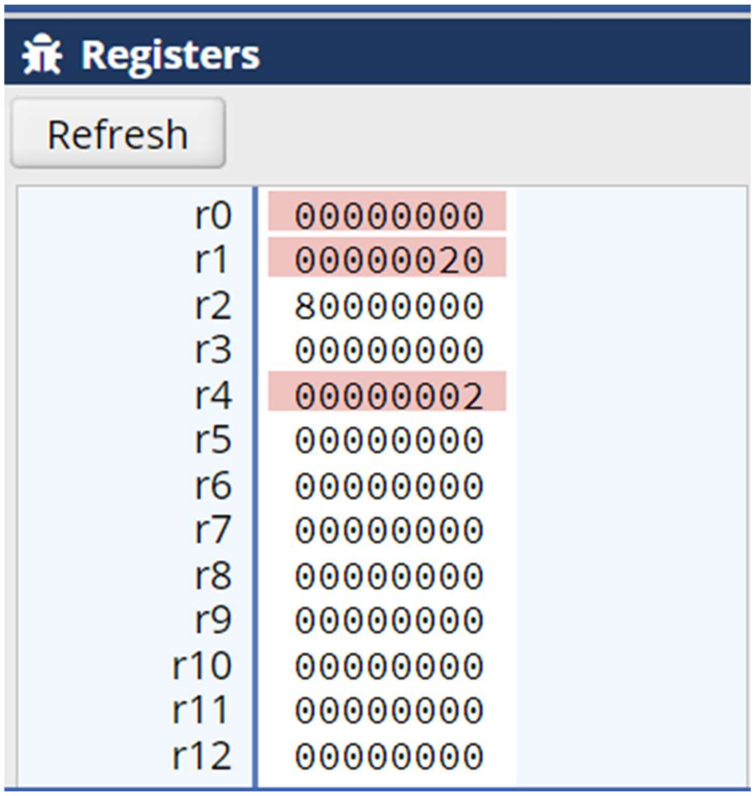
Figure 3.1 Register Results for Task #1 Part 2

**Task #2:**



```
r0   80008000
r1   00000000
r2   80000000
r3   00000000
r4   00000000
r5   00000000
r6   00000000
r7   00000000
r8   00000000
r9   00000000
r10  00000000
r11  00000000
r12  00000000
```

Figure 3.1 Register Results before execution

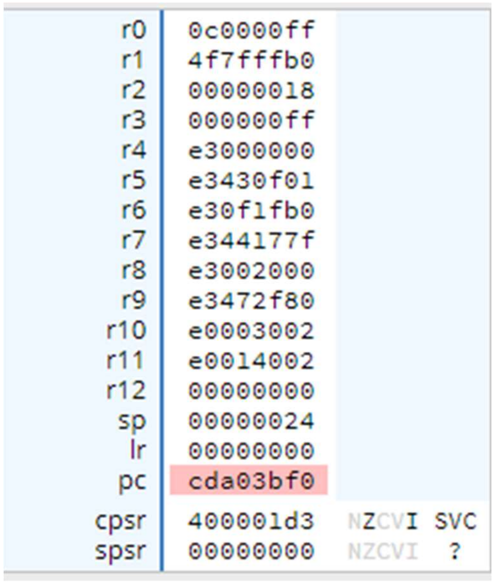


```
  r0   00000000
  r1   00000020
  r2   80000000
  r3   00000000
  r4   00000002
  r5   00000000
  r6   00000000
  r7   00000000
  r8   00000000
  r9   00000000
 r10   00000000
 r11   00000000
 r12   00000000
```

Figure 3.2 Register Results after execution

**Task #3:**

Task #3 did not work as expected. According to the Hand Analysis, R0 should display the result of 47800018 stored in the resistor memory as a result of the sum of the numbers found. One of the major problem encounter was in the comparing step. As a solution, more time needs to be put to study the float-point arithmiaitc and more understand of assembly code must be put to study and more debugging required.



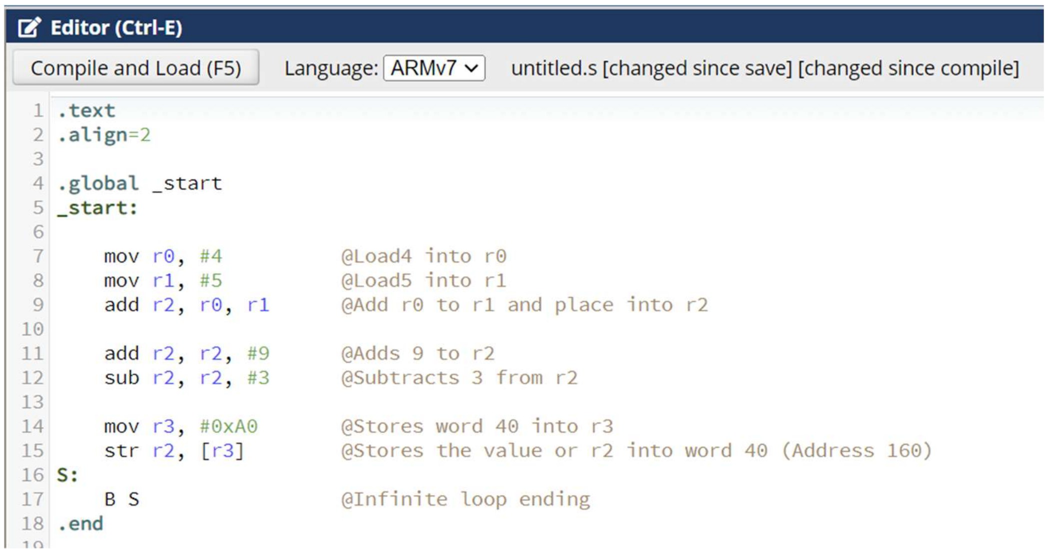
```
  r0   0c0000ff
  r1   4f7fffb0
  r2   00000018
  r3   000000ff
  r4   e3000000
  r5   e3430f01
  r6   e30f1fb0
  r7   e344177f
  r8   e3002000
  r9   e3472f80
 r10   e0003002
 r11   e0014002
 r12   00000000
  sp   00000024
  lr   00000000
  pc   cda03bf0

cpsr   400001d3   NZCVI SVC
spsr   00000000   NZCVI  ?
```

Figure 4 Register Memory Result for Task #3

**Final Product**

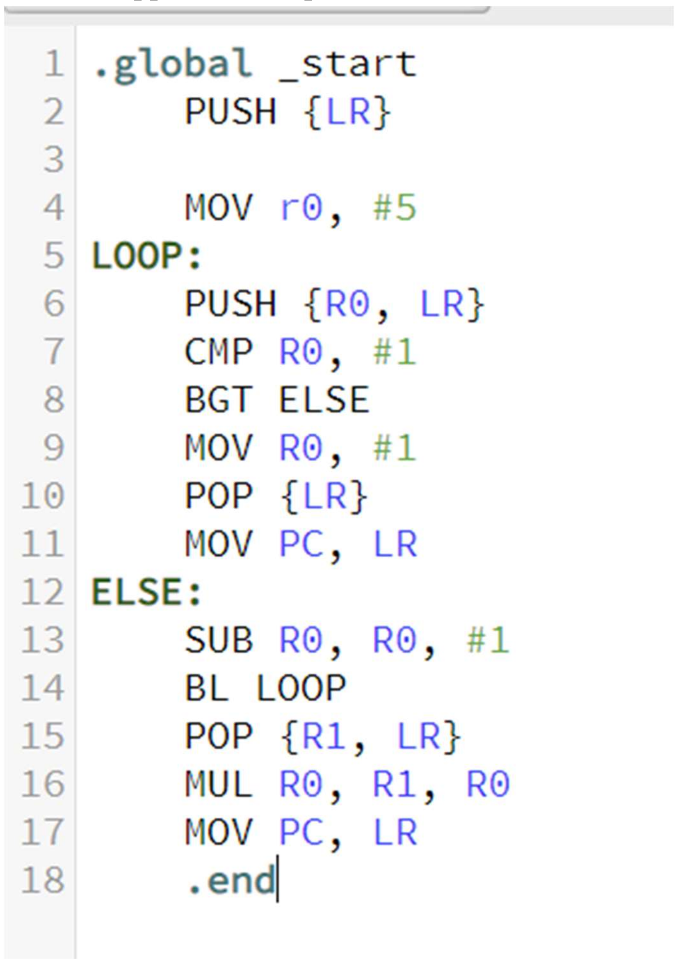
As a final product, we assembled the decrementing program from Task #1 and successfully  familarized ourselves with using CPUlator. For task #2, we successfully made an algorithm for counting the number of 1's in a 32-bit number using ARM assembly language. For task #3, we attempted to wite an ARM assembly language function that performs floating-point addition that strictly adds positive numbers. Although unsuccessful fully throughout the lab, we learned to use CPUlator and assembly language well.

## Appendix

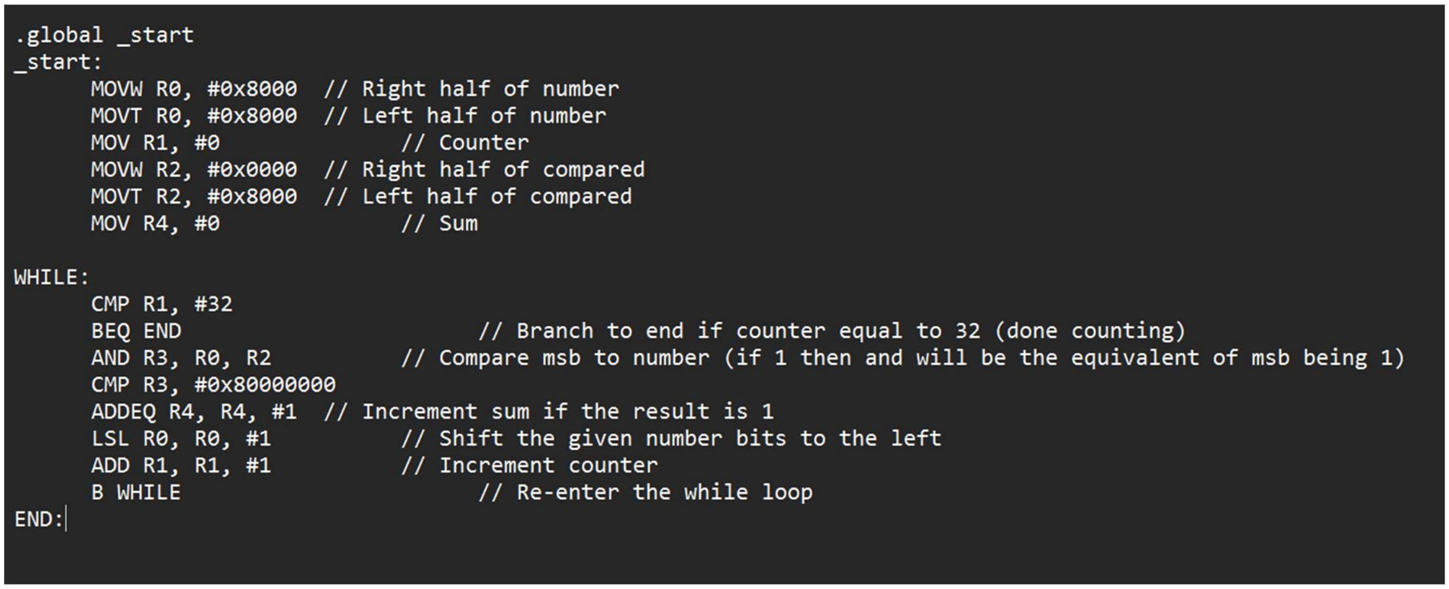

Editor (Ctrl-E)
Compile and Load (F5)
Language: ARMv7
untitled.s [changed since save] [changed since compile]

```
.text
.align=2

.global _start
_start:

    mov r0, #4          @Load4 into r0
    mov r1, #5          @Load5 into r1
    add r2, r0, r1      @Add r0 to r1 and place into r2

    add r2, r2, #9      @Adds 9 to r2
    sub r2, r2, #3      @Subtracts 3 from r2

    mov r3, #0xA0       @Stores word 40 into r3
    str r2, [r3]        @Stores the value or r2 into word 40 (Address 160)
S:
    B S                 @Infinite loop ending
.end
```

Appendix 1 Expanded Task1 Part 1


```
.global _start
    PUSH {LR}

    MOV r0, #5
LOOP:
    PUSH {R0, LR}
    CMP R0, #1
    BGT ELSE
    MOV R0, #1
    POP {LR}
    MOV PC, LR
ELSE:
    SUB R0, R0, #1
    BL LOOP
    POP {R1, LR}
    MUL R0, R1, R0
    MOV PC, LR
    .end
```

Appendix 2 Expanded Task1 Part 2



```
.global _start
_start:
      MOVW R0, #0x8000  // Right half of number
      MOVT R0, #0x8000  // Left half of number
      MOV R1, #0              // Counter
      MOVW R2, #0x0000  // Right half of compared
      MOVT R2, #0x8000  // Left half of compared
      MOV R4, #0              // Sum

WHILE:
      CMP R1, #32
      BEQ END                       // Branch to end if counter equal to 32 (done counting)
      AND R3, R0, R2          // Compare msb to number (if 1 then and will be the equivalent of msb being 1)
      CMP R3, #0x80000000
      ADDEQ R4, R4, #1  // Increment sum if the result is 1
      LSL R0, R0, #1          // Shift the given number bits to the left
      ADD R1, R1, #1          // Increment counter
      B WHILE                       // Re-enter the while loop
END:
```

Appendix 3 Expanded Task2

```
// Kiana Peterson and Nattapon Oonlamom
// 05/19/2023
// EE 469
// Lab 4: Assembly Programming

/* Overview:
 *              This code perform floating-point addition.
 *              The implementation assumes that the inputs are
 *              strictly positive single-precision floating-
point.
 * Inputs:
 *              R0 = The first floating-point number
 *              R1 = The second floating-point number
 * Output:
 *              R0 = Result of the addition
 */
.global _start
_start:

      // Initialize the floating-point number
      MOVW R0, #0x0000    // Initialize R0 with the first
floating-point number
      MOVT R0, #0x3F01
      MOVW R1, #0xFFB0    // Initialize R1 with the second
floating-point number
      MOVT R1, #0x477F

      // Step 1: Mask and shift down the two exponents
      MOVW R2, #0x0000       // Initialize exponent mask
      MOVT R2, #0x7F80
      AND R3, R0, R2         // Extract the exponent
      AND R4, R1, R2
      LSR R3, R2, #23         // Shift the exponent to the right
      LSR R4, R3, #23

      // Step 2: Mask the two fractions and append leading 1â€™s
to form the mantissas
      MOVW R2, #0x007F       // Initialize fraction mask
      MOVT R2, #0x0000
      AND R5, R0, R2         // Extract the fraction
      AND R6, R1, R2
      ORR R0, R0, #0x8000000 // Append leading 1 to memory
      ORR R1, R1, #0x8000000

      // Step 3: Compare the exponents by subtraction and
      //           Set the result exponent to be the larger
      CMP R3, R4
      MOVGE R7, R3           // If R3 > R4
      MOVLT R7, R4           // If R3 < R4

      // Step 4: Right shift the smaller mantissa to align the two
mantissas
```

Appendix 4 Task 3