Chenhan Dai, Junchao Zhou
ECE 469 Computer architecture I
May 19, 2023
Lab 4 Report

Procedure

This lab consisted of three tasks: writing a basic assembly program and tracing an Assembly program, design an algorithm for counting the number of 1's in a 32-bit number and write an ARM assembly language function that performs floating-point addition.

Task #1

The first task has two parts, the first part asks us to write an assembly program given figure 1. And then we need to search, discuss and answer a series of questions in figure 2.

```
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. Simple program that adds 2 numbers

Figure 1: A simple program that adds 2 numbers

1. Explain what these lines mean

1

```
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 2: A series of question need to answer

The second part asks us to open a new CPULator, assemble and debug the following program, we also discussed and answered a series of questions (Figure 3 and 4).

```
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 2. Assembly program

Figure 3: The given 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:

2

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

6. Save the current assembly code (Ctrl + S, or go to File -> Save). Change the name of the .s file to *Lab4_Task1_Part2.s*. You will need to submit this source file, along with your report. See the Deliverables section for more details.

Figure 4: A series of question need to answer

Task #2

In this task, we designed an algorithm for counting the number of 1's in a 32-bit number with the given order (Figure 5).

1. A pseudocode or a flow chart of your design steps.
2. Implement your algorithm using ARMv7 assembly in the CPUlator. Provide screenshots of the registers and/or memory locations used by the program.
3. Save the current assembly code (Ctrl + S, or go to File -> Save). Change the name of the .s file to *Lab4_Task2.s*. You will need to submit this source file, along with your report. See the Deliverables section for more details.

Figure 5: The order for task 2

Task #3

We wrote an ARM assembly language function that performs floating-point addition. And in this task we cannot use the ARM instructions specific to manipulating floating point numbers. The IEEE 754 Floating-Point Standard and several techniques are given in Figure 6 and 7.

Sign	Exponent (8 bits)								Fraction (23 bits)					
31	30	29	28	27	26	25	24	23	22	21	20	19	. . .	0

Figure 6: IEEE 754 Floating-Point Standard

AND 0000 0000 0110 1101 0001 0110 1110 0111
0000 0000 0011 1111 1111 0000 0000 0000

0000 0000 0010 1101 0001 0000 0000 0000

Figure7a: Bit-masking

OR 0000 0000 0011 0110 0010 1101 0001 1111
1100 1010 0000 0000 0000 0000 0000 0000

1100 1010 0011 0110 0010 1101 0001 1111

Figure7b: OR-ing

Figure 7: Some technique to implement floating-point addition

There are three techniques that are given to us:

Shift: Performing a left or right-shift will correspondingly remove bits that are shifted out.

Bit-masking: the output of the and operation is a new value containing only the desired bits. (Figure7a)

OR-ing: “OR”-ing registers together is a handy way to combine bit-fields. (Figure7b)

After understanding the background and some technique, we performed the hand analysis for several numbers(Figure 8) and implemented the algorithm that met the following requirements(Figure 9).

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

1. Write 2.0 as an IEEE single-precision floating point number.
2. Write 3.5 as an IEEE single-precision floating point number.
3. Write 0.50390625 as an IEEE single-precision floating point number.
4. Write 65535.6875 as an IEEE single-precision floating point number.
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.

Figure 8: the numbers to hand analysis

To write the algorithm, we followed the following steps as outline:

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 9: the outline for the algorithm

Results

Task 1: Introduction to Assembly Programming

Part 1: Writing your first Assembly Program

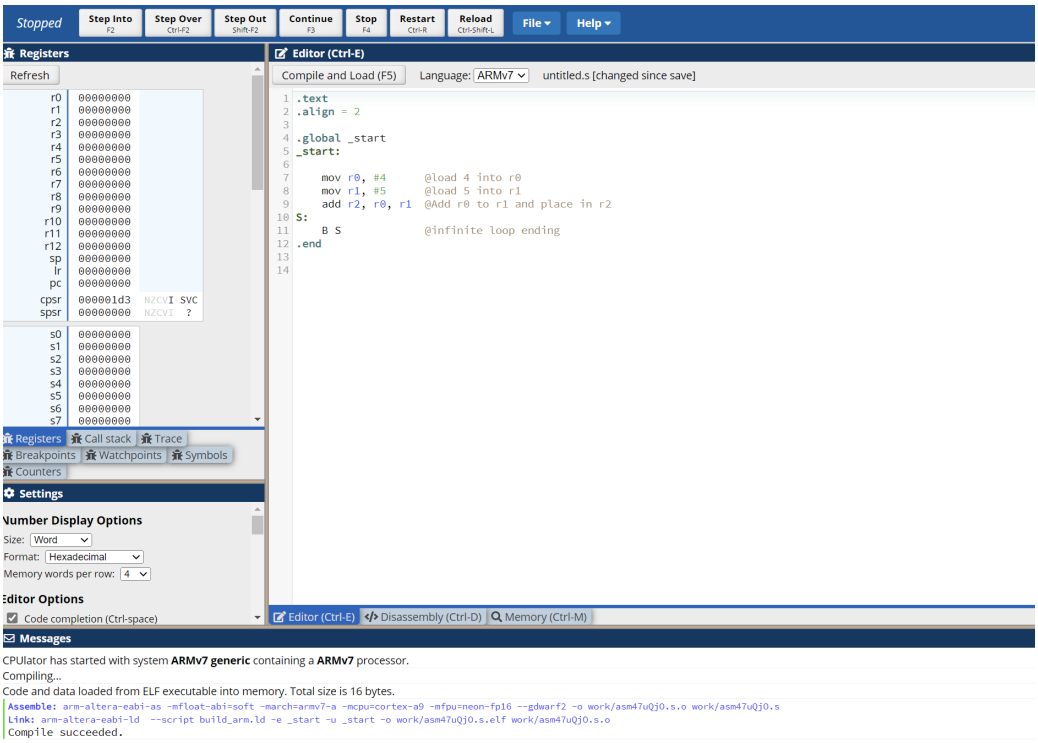

Figure 10: the codes for task 1 part 1

Answer the following questions and include screenshots of the registers:

1. Explain what these lines mean?

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

“.text” refers to a section of a computer program’s executable binary code that contains the instructions or machine code for the program’s main executable logic. “.align” is a directive that is used in assembly language to align data or code at a specific memory boundary. In this case, “=2” is the argument to the “.align” directive, which means that the following code or data should be aligned on a 2-byte boundary.

2. What is the value of R0, R1, R2, and PC at the start and end of the program?
   At the start, R0, R1, R2 and PC all equals 0. At the end, R0 is 4, R1 is 5, R2 is 9 and PC is c.

3. Explain the S:B S line of code (line 10 and 11)
   The “S:” label marks a location in the code, and the “B S” instruction is a branch instruction that jumps to the “S” label, creating an infinite loop that continuously executes the same set of instructions.

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.

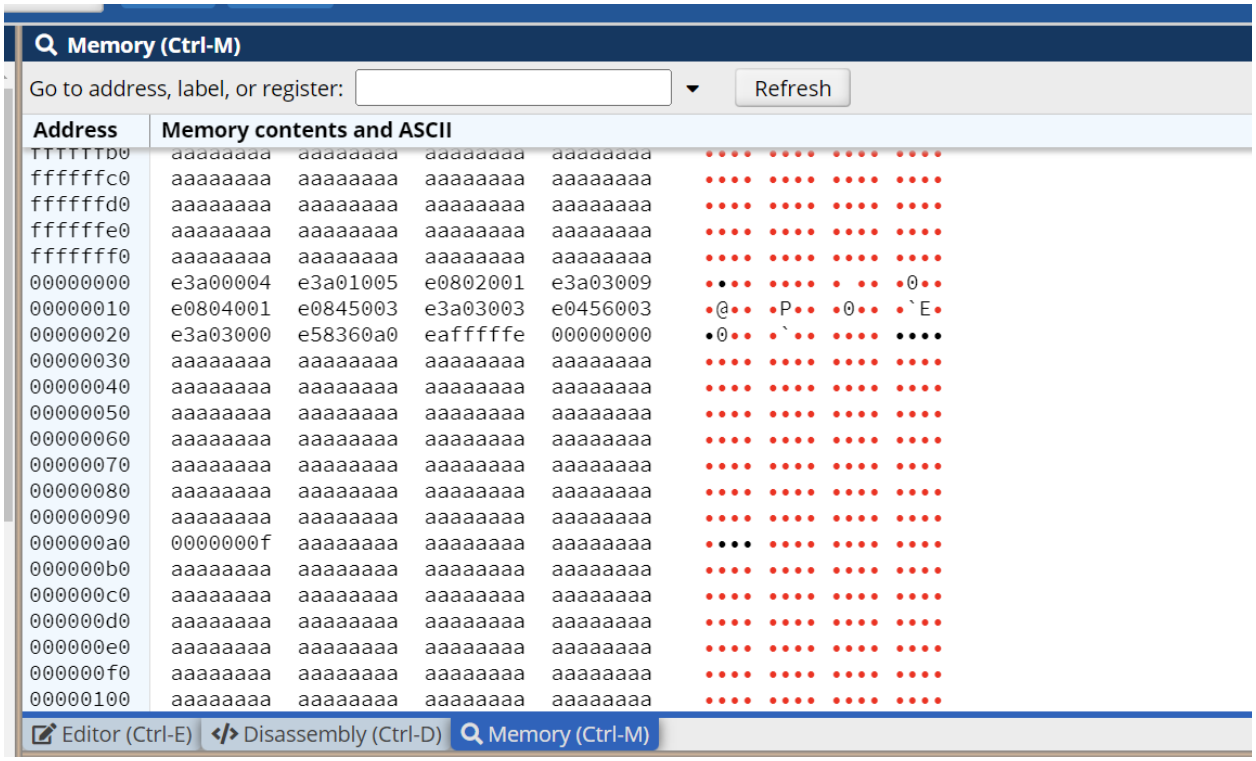

Figure 11: the screenshot of the memory

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.

Part 2: Tracing an Assembly Program

Follow the execution of the program in the debugger and notice the change of values of the registers. Answer the following questions:

1. What is the value in R0 after the program ends?
   6.

2. If the value initially placed in R0 is equal to 5, what is the value in R0 when the program ends?
   78.

3. What does this program do?
   This program would compute the factorial of the number in R0.

4. If you replace the instructions at line 2 and 10 in Figure 2 with PUSH {R0, R1} and POP{R1, R2} respectively, how will the program behave and why?
   This program basically does the same thing like the original code except it doesn't adjust the stack pointer by 8 byte before returning from the subroutine.

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.
   This program will still calculate the factorial of 3, but it will use register R3 instead of R0 for pushing and popping onto the stack.

b. Replace the instructions at lines 2 and 10 in Figure 2 with PUSH{LR} and POP{LR} respectively.
   This program calculates the factorial of 3 recursively by multiplying the current value R0 with the factorial of (R0 – 1).

c. Delete the instruction at line 6
If we delete the “PUSH {R0, LR}”, then the values of R0 and LR are not being pushed onto the stack before the loop starts.

6. Save the current assembly code (Ctrl + S, or go to File -> Save). Change the name of the .s file to LAB4_Task1_Part2.s. You will need to submit this source file, along with your report. See the Deliverables section for more details.

Task 2: Counting the number of 1’s

Design an algorithm for counting the number of 1’s in a 32-bit number. Please provide the following:

1. A pseudocode or a flow chart of your design steps.
Firstly, we need to initialize a counter variable to 0, and we use a loop to iterate through each bit in the 32-bit number, check if the current bit is 1 and if it is 1, increase the counter by 1. And we return the counter as a result.

Initialize counter variable
Initialize comparing variable
LOOP: if the target number has non 0 digits
comparing variable equals to target numbers AND 1
Counter add one when comparing variable is 1
Target number shift right by one bit

2. Implement your algorithm using ARMv7 assembly in the CPUlator. Provide screenshots of the registers and/or memory locations used by the program.

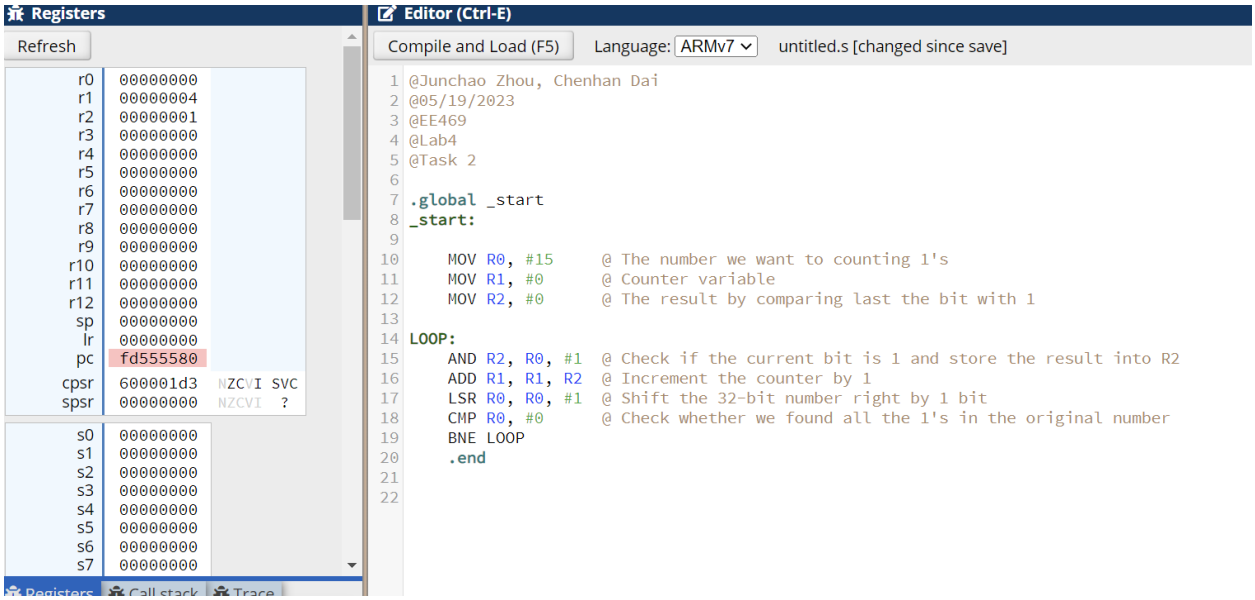

Figure 12: the code of counting the number of 1's

Save the current assembly code (Ctrl + S, or go to File -> Save). Change the name of the .s file to Lab4_Task2.s. You will need to submit this source file, along with your report. See the Deliverables section for more details.

Task 3: Floating Point Addition

Hand Analysis:

1. (2.0) = 0 10000000 00000000000000000000000 = 40000000
2. (3.5) = 0 10000000 11000000000000000000000 = 40600000
3. (0.50390625) = 0 01111110 00000010000000000000000 = 3F010000
4. (65535.6875) = 0 10001110 11111111111111110110000 = 477FFFB0
5. (3.) + (4.) = 0 10001111 00000000000000000011000 = 47800018

And we wrote an ARM assembly language function that performs floating-point arithmetic.

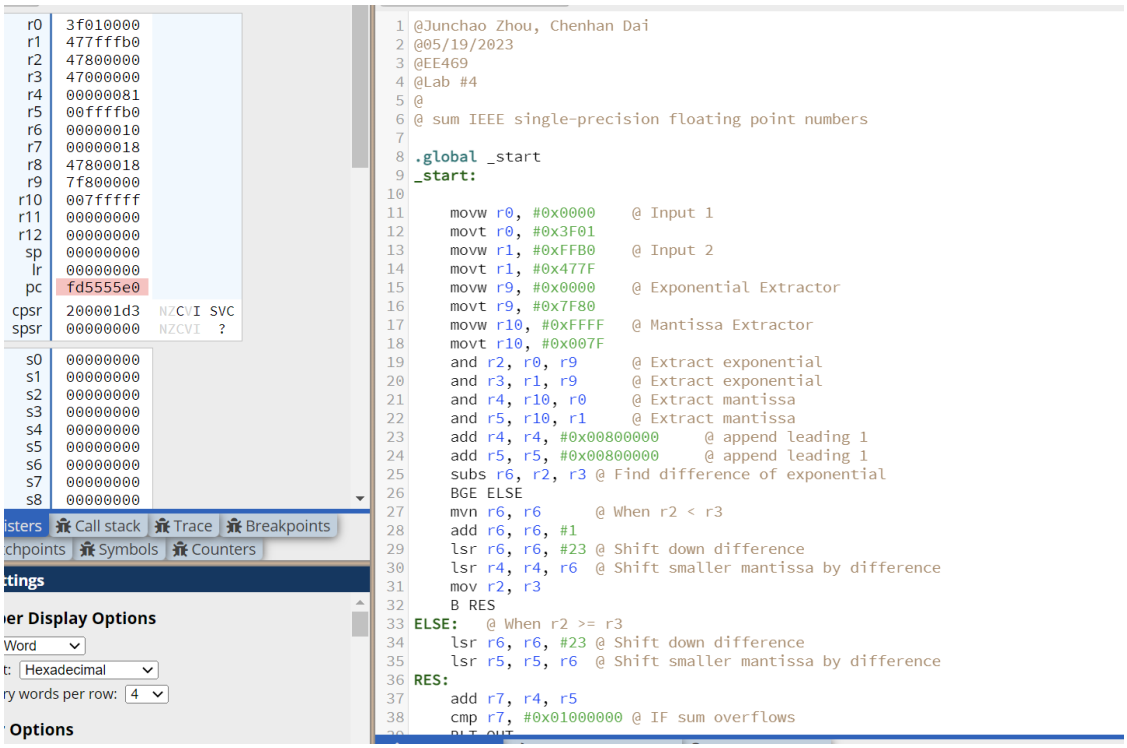

Figure 13: the code of doing floating point addition

Appendix: Assembly Code

Lab4_Task1_Part1.s

```
@Junchao Zhou, Chenhan Dai
@05/19/2023
@EE469
@Lab #4
@Task 1 Part 1
.text
.align = 2

.global _start
_start:

    mov r0, #4      @load 4 into r0
    mov r1, #5      @load 5 into r1
    add r2, r0, r1  @Add r0 to r1 and place in r2
    mov r3, #9      @load 9 into r3
    add r4, r0, r1  @Add r0 to r1 and place in r4
    add r5, r4, r3  @Add r4 to r3 and place in r5
    mov r3, #3      @load 3 into r3
    sub r6, r5, r3  @Sub r3 from r5 and place in r6
    mov r3, #0      @Load 0 into r3
    str r6, [r3, #160]  @Store the value in r6 to the 40th word in memory
S:
    B S             @infinite loop ending
.end

```

Lab4_Task1_Part2.s

```
@Junchao Zhou, Chenhan Dai
@05/19/2023
@EE469
@Lab #4
@Task 1 Part 2

.global _start
    PUSH {LR}

    MOV r0, #3
LOOP:

    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
```

Lab4_Task2.s

```
@Junchao Zhou, Chenhan Dai
@05/19/2023
@EE469
@Lab4
@Task 2

.global _start
_start:

    MOV R0, #15     @ The number we want to counting 1's
    MOV R1, #0      @ Counter variable
    MOV R2, #0      @ The result by comparing last the bit with 1

LOOP:
    AND R2, R0, #1  @ Check if the current bit is 1 and store the result into R2
    ADD R1, R1, R2  @ Increment the counter by 1
    LSR R0, R0, #1  @ Shift the 32-bit number right by 1 bit
    CMP R0, #0      @ Check whether we found all the 1's in the original number
    BNE LOOP
    .end

```

Lab4_Task3.s

```
@Junchao Zhou, Chenhan Dai
@05/19/2023
@EE469
@Lab #4
@
@ sum IEEE single-precision floating point numbers

.global _start
_start:

    movw r0, #0x0000    @ Input 1
    movt r0, #0x3F01
    movw r1, #0xFFB0    @ Input 2
    movt r1, #0x477F
    movw r9, #0x0000    @ Exponential Extractor
    movt r9, #0x7F80
    movw r10, #0xFFFF   @ Mantissa Extractor
    movt r10, #0x007F
    and r2, r0, r9      @ Extract exponential
    and r3, r1, r9      @ Extract exponential
    and r4, r10, r0     @ Extract mantissa
    and r5, r10, r1     @ Extract mantissa
    add r4, r4, #0x00800000     @ append leading 1
    add r5, r5, #0x00800000     @ append leading 1
    subs r6, r2, r3 @ Find difference of exponential
    BGE ELSE
    mvn r6, r6      @ When r2 < r3
    add r6, r6, #1
    lsr r6, r6, #23 @ Shift down difference
    lsr r4, r4, r6  @ Shift smaller mantissa by difference
    mov r2, r3
    B RES
ELSE:   @ When r2 >= r3
    lsr r6, r6, #23 @ Shift down difference
    lsr r5, r5, r6  @ Shift smaller mantissa by difference
RES:
    add r7, r4, r5
    cmp r7, #0x01000000 @ IF sum overflows
    BLT OUT
    lsr r7, r7, #1  @ Right shift mantissa by 1 bit
    add r2, r2, #0x00800000 @ Increase exponential by 1
OUT:
    sub r7, r7, #0x00800000 @ Strip leading 1
    orr r8, r7, r2 @ Combine
    .end
```