CMPT 295

Junchen Li (Richard)

301385486

2020/2/6

Assignment 3

Objectives:

- IEEE floating point number addition and rounding
- Memory addressing modes
- Assembly instructions
- Reading object code (machine level instruction) expressed in hexadecimal and understanding how these instructions are stored in memory
- Writing a C program that corresponds to given assembly program

---

Submission:

- Submit your document called **Assignment_3.pdf**, which must include your answers to all of the questions in Assignment 3.
    - Add your full name and student number at the top of the first page of your document **Assignment_3.pdf**.
- When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
- **If you hand-write your answers (as opposed to using a computer application to write them):** When putting your assignment together, do not take photos (no .jpg) of your assignment sheets! Scan them instead! Better quality -> easier to read -> easier to mark!
- Submit your assignment electronically on CourSys

Due:

- Thursday, Feb. 6 at 3pm
- Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.

Marking scheme:

This assignment will be marked as follows:

- Questions 1, 2 and 5 will be marked for correctness.

- Questions 3 and 4 will be marked for completeness, i.e., you get marks for completing (answering) the question, but it is up to you to verify the correctness of your answer by looking up the solutions when they are posted.

The amount of marks for each question is indicated as part of the question.

A solution will be posted after the due date.

---

1. [3 marks] IEEE floating point number addition and rounding

   When adding real numbers expressed in scientific notation (base 10), we must first transform them such that they have the same exponent. For example, $3.1416 + 1.0 \times 10^3$ must be transformed to 3.1416 + 1000.0. Once the numbers are expressed with the same exponent, we need to align their decimal point, then we can add them ($1003.1416 = 1.0031416 \times 10^3$).

   The same is true when adding IEEE floating point numbers, except that the base we are working with is 2.

   Perform the following IEEE floating point number additions following the algorithm described above, i.e., first, transform the IEEE floating point numbers (expressed as hexadecimal numbers) such that they have the same exponents, align their binary points and add them. Express their sum as an IEEE floating point number, then express this IEEE floating point number as a hexadecimal number. Show your work and clearly show the result of rounding, if rounding occurs.

   a. 0x43E4FC80 + 0x41C52333
   b. 0x43E4FC80 + 0x41C52339
   c. 0x3E2AAAAB + 0x3F555555

      where 0.16666667 approximates 0x3E2AAAAB
      and 0.83333333 approximates 0x3F555555

1、a 0x43E4FC80 + 0x41C52333

0x43E4FC80 → 0 10000111 11001001111110010000000   76543210 10000111 $2^7+2^2+2^1+2^0 = 128+4+2+1 = 135$

bias $= 2^{8-1}-1 = 127$ E = exp − bias = 135 − 127 = 8 $1.11001001111110010000000_2$ ↓ $11100100.111110010000000_2$

0x41C52333 → 0 10000011 10001010010001100110011   76543210 10000011 $2^7+2^1+2^0 = 128+2+1 = 131$

bias $= 2^{8-1}-1 = 127$ E = 131 − 127 = 4 $1.10001010010001100110011_2$ → $11000.1010010001100110011_2$

```
  11100100.111110010000000
+    11000.1010010001100110011
  ---------------------------
  111100010.1001110101100110011 (2)
  8
```

E = 8
bias = 127
exp = 135_{10} → 10000111_2
And the result is positive

(1).11110001010011101011001 0011
27 round occur
0011 < ½ − round down

Final Answer: 0 10000111 11100010100111010110011

Hex number: 0x43F14EB3

b. 0x43E4FC80 + 0x41C52339

~~0x100~~ 0x43E4FC80 → 0 10000111 11001001111110010000000   76543210 10000111 $2^7+2^2+2^1+2^0 = 128+4+2+1 = 135$

bias $= 2^{8-1}-1 = 127$ E = 135 − 127 = 8 → $11100100.111110010000000_2$

0x41C52339   0 10000011 10001010010001100111001   E = 4   11000.1010010001100111001

```
  11100100.111110010000000
+    11000.1010010001100111001
  ---------------------------
  111100010.1001110101100111001 (2)
```

(1).111000101001110101100111001 (2)
27
1001 > ½ − up
...0011 → ...0100

Final Answer: 0 10000111 11100010100111010110100

Hex number: 0x43F14EB4

C. 0X3E2AAAAB + 0x3F555555

0x3E2AAAAB = 0 01111100 01010101010101010101011   76543210 01111100 $2^6+2^5+2^4+2^3+2^2 = 64+32+16+8+4 = 124$

bias $= 2^{8-1}-1 = 127$ E = 124 − 127 = −3 → $0.001010101010101010101011_2$

0x3F555555 = 0 01111110 10101010101010101010101   76543210 01111110 $2^6+2^5+2^4+2^3+2^2+2^1 = 126$

bias $= 2^{8-1}-1 = 127$ E = 126 − 127 = −1 $0.110101010101010101010101_2$

0 01111111 00000000000000000000000

```
  0.001010101010101010101011 (2)
  0.110101010101010101010101 (2)
  ---------------------------
  0.111111111111111111111111 (2)
```

E = ~~1~~ 0   110 > ½ − up
bias = ~~128~~ 127
exp = ~~128~~$_{10}$ → 0111111_2 $1.00000000000000000000000_2$

Final Answer: 0 01111111 00000000000000000000000

Hex number: 0x3F800000

CMPT 295 – Spring 2020

2. [7 marks] Memory addressing modes

Assume the following values are stored at the indicated memory addresses and registers:

Memory Address	Value
0x230	0x23
0x234	0x00
0x235	0x01
0x23A	0xed
0x240	0xff

Register	Value
%rdi	0x230
%rsi	0x234
%rcx	0x4
%rax	0x1

Imagine that the operands in the table below are the **Src** (source) operands for some unspecified assembly instructions, fill in the following table with the appropriate answers.

Note: We do not need to know what these assembly instructions are in order to fill the table.

$560_{10} \rightarrow 1000110000$
0x2 3 0

$550_{10} \rightarrow 1000100110$
2 2 6

Operand	Operand Value (expressed in hexadecimal)	Operand Form (Choices are: Immediate, Register or one of the 9 Memory Addressing Modes)
%rsi	0X234	Register
(%rdi)	0X23	Indirect memory addressing mode
$0x23A	0X23A	Immediate
0x240	0xff	Absolute memory addressing mode
10(%rdi)	0X230 + 0XA = 0X23A → 0Xed	"Base + displacement" memory addressing mode
560(%rcx,%rax)	0X1 + 0X4 + 0X230 0X235 → 0X01	Indexed memory addressing mode
-550(, %rdi, 2)	2 · 0X230 = 0X460 0X234 → 0X00	Scaled indexed memory addressing mode
0x6(%rdi, %rax, 4)	0Xff	Scaled indexed memory addressing mode

0X4 + 0X230 + 0X6

0X240 → 0Xff

CMPT 295 – Spring 2020

Still using the first table listed above displaying the values stored at various memory addresses and registers, fill in the following table with three different **Src** (source) operands for some unspecified assembly instructions. For each row, this operand must result in the operand **Value** listed and must satisfy the **Operand Form** listed.

Operand	Value	Operand Form (Choices are: Immediate, Register or one of the 9 Memory Addressing Modes)
0X234	0x00	Absolute memory addressing mode
Jb0(,%rax,4)	0x00	Scaled indexed memory addressing mode
(%rdi,%rcx)	0x00	Indexed memory addressing mode

3. [2 marks] Assembly instructions

Requirement 1:

We would like to write assembly code (instruction(s)) that multiplies the value stored in the register `%esi` by `c`, where `c` is a positive integer constant (fits in 32 bits), and stored their product in the register `%eax`, i.e., `%eax <- c * %esi`.

In the table below, write the assembly code (instruction(s)) that satisfies **Requirement 1** above and the other requirements found in the **Other Requirements** column:

Other Requirements	Assembly Code (instruction(s))
• Using **two** assembly instruction • c is any positive integer constant (you can use $c in your instruction)	IMUL $c,%esi movq %esi,%eax
• Using **one** assembly instruction • c = 8	Leaq (,%esi,8),%eax
• Using **one** assembly instruction • c = 5	leaq %esi(,%esi,4),%eax
• Using **two** assembly instructions • c = 21	IMUL $21,%esi movq %esi,%eax

CMPT 295 – Spring 2020

4. 2 marks] Machine level instructions and their memory location

Consider a function called `arith`, defined in a file called `arith.c` and called from the main function found in the file called `main.c`.

This function `arith` performs some arithmetic manipulation on its **three parameters**.

Compiling `main.c` and `arith.c` files, we created an executable called `ar`, then we executed the command:

```
objdump -d ar > arith.objdump
```

We display the partial content of `arith.objdump` below. The file `arith.objdump` is the disassembled version of the executable file `ar`.

Your task is to fill in its missing parts, which have been underlined:

```
0000000000400527 <arith>:
  400527:   48 8d 04 37          lea    (%rdi,%rsi,1),%rax
  40052b:   48 01 d0             add    %rdx,%rax
  40052e:   48 8d 0c 76          lea    (%rsi,%rsi,2),%rcx
  400532:   48 c1 e1 4           shl    $0x4,%rcx
  400536:   48 8d 54 0f 04       lea    0x4(%rdi,%rcx,1),%rdx
  40053b:   48 0f af c2          imul   %rdx,%rax
  40053f:   C3                   retq
                                  C3
```

5. [6 marks] C program versus assembly program

Do the Homework Problem 3.58 at the end of Chapter 3 and include your program called **decode2.c** below. Make sure you satisfy the following requirements:

- Variables and constants must be descriptively named.
- Your code must be commented and well spaced such that others (i.e., TA's) can read your code and understand it.
- You cannot use the `goto` statement.
- You must write your program using C (not C++) and your program must compile on a CSIL computer using the Linux operating system.

Once you have created you program **decode2.c**, generate its assembly code version using the optimization level "g" (`-Og`) and call it **decode2.s**. Include it below as well without making any modifications to it.

```
#include <stdio.h>
// x->%rdi   y->%rsi   z->%rdx
long decode2 (long x,long y,long z)
{
     y=y-z;          // subq y<-y-z
     x=x*y;          // imulq x<-x*y
     long temp1=y;       //movq %rax<-y
     long temp2=temp1<<63; //salq y<-$63 left shift same as SHL
     long temp3=temp2>>63;         // sarq y<-$63 right shift
     long ans=temp3^x;        // xorq %rax<-^x Exclusive-or
     return ans;         //return
}
```