CS230 Spring 2018
Programming Assignment #1: Bit Manipulation

Due date: Tuesday, April 24

1. Introduction

The purpose of this assignment is to become more familiar with the binary representation of signed integers & IEEE floating-point representation and to understand what happens during the multiplication and addition.

You should know how to operate multiplication of integer and addition/multiplication of floating-point at the bit level, and emulate it in C language.

2. Problem specification

2.1 Overview

2.1.1 emulation1: Multiplication of integers

Write a C function named mult_int_32_to_64() which receives two signed integers and computes the multiplication of those integers. The prototype of mult_int_32_to_64() is as follows:

```
mult_int_32_to_64(unsigned x, unsigned y, unsigned *xyh, unsigned *xyl);
```

The first two arguments, x and y, represents the multiplicand and multiplier, respectively. Even though x and y are unsigned parameters in the function prototype, they are bit representation of signed integers. The function **mult_int_32_to_64()** should store the result of (x * y) in the memory locations pointed to by xyh and xyl. Since the multiplication of two signed 32-bit integers produces a 64-bit result, the high-order 32 bits should be saved in the memory location pointed to by xyh and the low-order 32bits in the memory location designated by xyl.

2.1.2 emulation2: Addition of floating-point numbers

Write a C function named float_add() which receives two IEEE floating-point numbers and computes the addition of those numbers. The prototype of float_add() is as follows:

```
float_add(unsigned x, unsigned y, unsigned *result);
```

You need to add the first two arguments, x and y, represented as the bit representation of IEEE 754 single precision floating-point numbers. The function float_add() should store the result of (x + y) in the memory location pointed to by unsigned pointer result.

2.1.3 emulation3: Multiplication of floating-point numbers

Write a C function named float_mult() which receives two IEEE floating-point numbers and computes the multiplication of those numbers. The prototype of float_mult() is as follows:

```
float_mult(unsigned x, unsigned y, unsigned *result);
```

You need to multiply the first two arguments, x and y, representing the bit representation of IEEE 754 single precision floating-point number. The function float_mult() should store the result of (x * y) in the memory location pointed to unsigned pointer by result.

2.2 Backgrounds

2.2.1 Integer multiplication

When two integers x and y are represented as **w-bit** two's complement numbers, their product (x * y) requires as many as **2*w-bits** to be represented in two's complement form. This is because signed integers are represented as 32 bits and the multiplication of two signed integers produces a 64-bit result.
In C, however, signed multiplication (32bit *32bit) is performed by truncating the 64-bit product to 32 bits (For further details, please refer to Section: *Integer Arithmetic* of the textbook).

Instead of truncating the high-order 32 bits, mult_int_32_to_64() stores the high-order 32 bits into xyh, and the low-order 32 bits into xyl. In this way, you can get the true 64-bit product.

TIP:
It might be helpful to understand how to use bitwise operations effectively. (e.g. bit masking)
You should handle any exception:
Multiplication with zero, positive * positive, positive * negative, negative * negative

2.2.2 Floating-point addition

When you start this implementation, you should fully understand IEEE floating-point format. The bits of IEEE floating-point consist of three parts: sign-exponent-fraction.
Only normalized form is considered. Thus, addition procedure is not different with integer number but scaling up & down of exponent part is required.
(For further details, please refer to Section: *Floating Point* of the textbook).

TIP: You should handle any exception:
Add with zero, positive + positive, positive + negative, negative + negative

2.2.3 Floating-point multiplication

If you fully understand IEEE floating-point format in above section, you can start this implementation.
Multiplication procedure is similar to the addition procedure, but scaling up & down of exponent part is not required. Instead of this, "mult_int_32_to_64()" function, which you implemented in Integer multiplication, may be used to multiply fraction parts of x and y. Only normalized form is considered too.
(For further details, please refer to Section: *Floating Point* of the textbook).

TIP: You should handle any exception:
Multiplication with zero, positive * positive, positive * negative, negative * negative

2.3 Restrictions

When you implement the functions,

You should use only unsigned type variables. Do not use "long long type" and "float type".

If you violate this restriction in one problem, ***you will get 0 points for that particular problem***.
But if plagiarism in your code is found, ***you must get 0 points for all problems.***
Do not copy source codes submitted in past CS230 courses. You are not allowed to copy other students' source codes at all. If you violate this rule, you will get 0 points without exception.

3. Checking Your Work

- The name of your work directory is initially named as “00000000”, but you need to change the name to your student ID. To do this, in terminal command line, type as follows
  ```
  xxxx/cs230_project1# mv ./00000000 ./(your student ID)
  ```
- To compile your source code, just type as follows
  ```
  xxxx/cs230_project1# ./build.sh
  ```
- Then, executable file named “program_00000000” is created in your work directory. To execute your program, just type as follows
  ```
  xxxx/cs230_project1# ./(your student ID)/program_(your student ID)
  ```
- For evaluation, we will use “testset” file as input to your program. To check your program makes correct answers, just type as follows
  ```
  xxxx/cs230_project1# ./score.sh
  ```

4. Evaluation

90 - Correctness: 18 problems, each problem with 5 points.
10 - Style points: We expect you to have good comments in a source code (skeleton.c) as well as a document.

5. Submission

Basic skeleton codes are provided. So you simply need to fill in the skeleton.
(Complete 3 blank functions: **mult_int_32_to_64()**, **float_add()**, and **float_mult()**)

```
1.   cs230_project1 (uppermost directory of project#1)
2.   ├ build.sh
3.   ├ score.sh
4.   ├ main.c
5.   ├ testset
6.   └ 00000000 (your work directory)
7.       └ skeleton.c
```

When you submit, rename <u>your working directory</u> to your student ID and compress it to “tar.gz” using the following command:

```
xxxx/cs230_project1# tar -cvfz your_student_id.tar.gz your_student_id
```

The file you submit should include the following items:

```
1.   20xxxxxxx (your student ID)
2.   ├ skeleton.c (The source code filled with your implementation.)
3.   └ A document describing your implementation, Briefly.
```

Please upload the compressed file on KLMS PA#1 submission board. **Do not submit it through e-mail.**

If you have any questions, please ask on the KLMS Q&A borad.

Some sample runs:

```
$ ./program_xxxxxxxx
Problem#1  x  y  ?  1  1
result#1  0x0000000000000001  0x0000000000000001
Problem#2  f1  f2  ?  0.1  0.1
result#2   0.20000   0.20000
Problem#3  f1  f2  ?  0.3  0.3
result#3   0.09000   0.09000

$ ./program_xxxxxxxx
Problem#1  x  y  ?  1  -1
result#1  0xffffffffffffffff  0xffffffffffffffff
Problem#2  f1  f2  ?  0.1  -0.1
result#2  0.00000   0.00000
Problem#3  f1  f2  ?  0.3  -0.3
result#3  -0.09000  -0.09000

$ ./program_xxxxxxxx
Problem#1  x  y  ?  0 123456789
result#1  0x0000000000000000  0x0000000000000000
Problem#2  f1  f2  ?  20000000.00000  0.00000000001
result#2  20000000.00000  20000000.00000
Problem#3  f1  f2  ?  20000000.00000  0.00000000001
result#3   0.00020   0.00020

$ ./score.sh
----- test start id for xxxxxxxx  -----

<test-case-1>
0x0000000000002b70 0x0000000000002b70 pass
          123.53546              123.53546  pass
           15.19699               15.19699  pass

<test-case-2>
0x0000000000005b2c 0x0000000000005b2c pass
         1123.31226             1123.31226  pass
        40048.33203            40048.33203  pass

<test-case-3>
0x0000001a160ea0c9 0x0000001a160ea0c9 pass
          268.39999              268.39999  pass
        10360.08105            10360.08105  pass

<test-case-4>
0x000000104d3da844 0x000000104d3da844 pass
            1.34685                1.34685  pass
       203430.71875           203430.71875  pass

<test-case-5>
0xffffc2323c67fe9c  0xffffc2323c67fe9c  pass
            0.62447                0.62447 pass
        15230.60156            15230.60156  pass

<test-case-6>
0xffffc2323c67fe9c  0xffffc2323c67fe9c  pass
            0.62447                0.62447 pass
        15230.60156            15230.60156  pass
```