

Lab 9: Fun with Floating Point

CPSC 2310 – Spring 2020

# Clare DuVal (ckduval C17822756)

# Due Date: Friday, April 10, 2020 @ Midnight

# Lab Objective

* Practice with bit manipulation
* Floating point

# Introduction

This lab will have 2 parts;

1. Practice converting fractional binary numbers.

2. Implement two functions both requiring you to use, bit manipulations such as: >>, <<, &, ~, ^, |

These functions are related what we covered with respect to floating point number and how they are represented in the computer. You should review this information and take another look at the Float Toy website:

<https://evanw.github.io/float-toy/>

# Assignment

**Read the entire document. There are several lines of code that require you to do specific task. These task are not necessarily labeled as task.**

Part 1:

It has been determined that the single-precision floating-point **approximation** of (3.1415927) has the hexadecimal representation of 0x40490FBD. Fill in the following chart for . You may use a calculator but you must **show your work.**

|  |  |  |
| --- | --- | --- |
| Fractional value | Binary representation | Decimal representation |
| 43/16 | 10.1011 | 2.6875 |
| 22/7 | 0011.0010010000111111011110 | 3.1415927 |

½ = .5, 1/3 = .33, ¼ = .25, 1/5 = .20, 1/6 = .1667, 1/7 = .14285 (close enough) 1/7 + ((3\*7)/7) = 22/7 = pi

.1415927 \* 2 = .2831854 🡪 0

. 2831854 \* 2 = .5663708 🡪 0

. 5663708 \* 2 = 1.1327416 🡪 1

. 1327416 \* 2 = .2654832 🡪 0

. 2654832 \* 2 = .5309664 🡪 0

. 5309664 \* 2 = 1.0619328 🡪 1

. 0619328 \* 2 = .1238656 🡪 0

.1238656 \* 2 = .2477312 🡪 0

. 2477312 \* 2 = .4954624 🡪 0

. 4954624 \* 2 = .9909248 🡪 0

. 9909248 \* 2 = 1.9818496 🡪 1

. 9818496 \* 2 = 1.9636992 🡪 1

. 9636992 \* 2 = 1.9273984 🡪 1

. 9273984 \* 2 = 1.8547968 🡪 1

. 8547968 \* 2 = 1.7095936 🡪 1

. 7095936 \* 2 = 1.4191872 🡪 1

. 4191872 \* 2 = .8383744 🡪 0

. 8383744 \* 2 = 1.6767488 🡪 1

. 6767488 \* 2 = 1.3534976 🡪 1

. 3534976 \* 2 = .7069952 🡪 1

. 7069952 \* 2 = 1.4139904 🡪 1

. 4139904 \* 2 = .8279808 🡪 0

3 = 0011

Not required but an interesting observation: As I was calculating the binary value of .1415927 (by multiplying by 2), I realized, I was following a particular pattern even when I was using my calculator. Which made me thing, these steps (pattern) could be replicated in a “program” fairly easy. Hint: to get as close .1415927 as possible this should go out ~22 binary spaces.

Part 2:

You will write two functions for this part. For this code, there are Bit-Level Floating Point Rules you must follow:

You will write functions that will implement floating-point functions, operating directly on bit-level representations of floating-point numbers. Your code should exactly replicate the conventions for IEEE floating- point operations.

You will use a typedef to define an unsigned with the name of **fbits**:

typedef unsigned **fbits**;

I can now use fbits to represent ‘unsigned’.

Instead of using data type float in your code, you will use **fbits**. You may use both int and unsigned data types, including unsigned and integer constants and operations. You may not use any unions, stucts, or arrays. Most significantly, you may not use any floating point data types, operations, or constants.

The following function illustrates the use of these coding rules. For argument ‘f’, it returns 0 if ‘f’ is denormalized (preserving the sign of ‘f’), and returns ‘f’ otherwise.

Example:

In order to write the two required functions you need to understand the following code. Therefore, you are to explain each line of code in the following function.

/\*If ‘f’ is denorm, return 0. Otherwise, return ‘f’\*/

fbits f\_denorm\_zero(fbits f)

{

/\*EXPLAIN THIS LINE OF CODE.

What and why is it doing what it is doing\*/

// How: Shifting the variable f by 31 bits and saving it to the unsigned variable sign

// What: Getting the signed bit of the fbits variable

unsigned sign = f >> 31;

/\*EXPLAIN THIS LINE OF CODE

What and why is it doing what it is doing \*/

// How: Shifting the variable f by 23 bits then finding the common 1s it has with 1111 1111 and set it to the variable exp

//What: Getting the exponent of the fbits variable

unsigned exp = f >> 23 & 0xFF;

/\*EXPLAIN THIS LINE OF CODE

What and why is it doing what it is doing \*/

// How: Find the common 1 bits the variable f has with the limit of ints and set it equal to the variable frac

// What: Get the mantissa of the fbits variable

unsigned frac = f & 0x7FFFFF;

/\*EXPLAIN THIS BLOCK OF CODE

What and why is it doing what it is doing \*/

// How: If the variable exponent equals zero, then also set the variable frac equal to zero

// What: If the exponent is denormalized, then turn the fractional bit (mantissa) into 0

if (exp == 0)

{

frac = 0;

}

/\*EXPLAIN THIS LINE OF CODE

What and why is it doing what it is doing \*/

// How: First left shift the sign variable by 31 bits. Then left shift the exp variable by 23 bits.

// Then return a series of bits composed if any of the three variables (sign, exp, or frac) have a

// 1 in that place.

// What: Placing the signed bit(bit 0), exponent (bits 1-9), and mantissa (bits 10-32) back in their respective bits

return (sign << 31) | (exp << 23) | frac;

}

**floatFunctions.h has been written for you**

**Create a file called floatFunctions.c and implement the following:**

Function 1:

The prototype for the first function is:

**fbits fNeg(fbits fnum);**

You are to write a function that computes the negative of the fbits variable **fnum** passed to the function. If fnum is NaN, then simply return fnum.

You are required to comment your code.

You should test your function by evaluating it for all 232 values of argument fnum and comparing the result to what would be obtained using your machine’s floating point operations.

Function 2:

The prototype of the second function is:

**fbits fAbsVal(fbits fnum);**

You are to write a function that computes the absolute value of **fnum**. If fnum is NaN, your function should simply return fnum.

You are required to comment your code.

You should test your function by evaluating it for all 232 values of argument fnum and comparing the result to what would be obtained using your machine’s floating point operations.

**Additional instructions:**

Be sure to include a header similar to the following in your file.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\*Your name \*

\*CPSC2310 Spring 20 \*

\*Lab Section: <Your section> \*

\*UserName: \*

\*Instructor: Dr. Yvon Feaster \*

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# Submission Instructions

Tar the **completed lab document**, **floatFunctions.h**, and **floatFunctions.c** and use handin to submit the tarred file to the Lab9 folder.