Note 1: In all assignments that asks for writing a program, it should contain at least one test scenario in addition to the main functionality of your program.

Note 2: In several of the following problems, we will artificially restrict what programming constructs you can use to help you gain a better understanding of the bit-level, logic, and arithmetic operations of C. In answering these problems, your code must follow these rules:
Note 3: Start with Q3 to Q7 and then, solve Q1 and Q2.

Bit-level floating-point coding rules

In the following problems, you will write code to 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, including using round-to-even mode when rounding is required.

Toward this end, we define data type float_bits to be equivalent to unsigned. This way we would be able to do bitwise operations. Basically we simulate the floating point numbers.

```
/* Access bit-level representation floating-point number */
typedef unsigned float_bits;
```

Rather than using data type float in your code, you will use float_bits.

You may use both int and unsigned data types, including unsigned and integer constants and operations. You may not use any unions, structs, or arrays. You can use unions **Only** for data presentation as shown in Q2. Most significantly, you may not use any floating-point data types, operations, or constants. Instead, your code should perform the bit manipulations that implement the specified floating-point operations.

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.

```
/* If f is denorm, return 0. Otherwise, return f */
float_bits float_denorm_zero(float_bits f) {
        /* Decompose bit representation into parts */
        unsigned sign = f>>31;
        unsigned exp = f>>23 & 0xFF;
        unsigned frac = f & 0x7FFFFF;
        if (exp == 0) {
                /* Denormalized. Set fraction to 0 */
                frac = 0;
        }
        /* Reassemble bits */
        return (sign << 31) | (exp << 23) | frac;
```

}

Q1. (15 points)

Following the **bit-level floating-point coding rules**, implement the function with the following prototype:

```
/* Compute -f. If f is NaN, then return f. */
float_bits float_negate(float_bits f);
```

For floating-point number f, this function computes $-f$. Basically change the sign of f. If f is NaN, your function should simply return f, i.e., no change in sign bit.

Test your function for some values, which you can define them as hex for simplicity since float_bits is of type unsigned.

Q2. (35 points)

Following the **bit-level floating-point coding rules**, implement the function with the following prototype:

```
/* Compute (float) i */
float_bits float_i2f(int i);
```

Hint1: Since the conversion is from integer to float, all digits after the decimal point would be zeros. The main visible change in conversion would occur in rounding large numbers. For example for the integer 2099901000, the casting will result in float value of 2099901056.000000.

Hint 2: You can define a union (Refer to your Chapter 6 of your C book to know more about union) to be able to print a binary presentation as an int or a float. An example is provided also in

/home/TU/tug32055/CIS2107public/formatingi2f.c

on the cis-linux2.temple.edu

Clearly this program is not complete because the implementation of myi2f() function is not complete.

For the following questions, submit a .pdf file.

Q3. (15 points)

How would 121 $\frac{3}{128}$ be stored in a C float variable? Write the result in binary.

Helps: 8 bits for exponent. 23 bit for fraction. Bias: 127.

0 10000101 11100100000110000000000

Q4. (10 points)

Considering a machine with 8-bits word size. To store a floating point number in IEEE format, we allocate 3 bits for exponent and 4 bits for fraction. What is the largest normalized value that can be stored? Compute the value in **decimal**.

Help: Bias is 3.

Normalized numbers: exponent is neither all ones nor all zeroes.

0 (sign bit) 110 (exponent) 1111 (fractional)

6 – 3 = 3

1.1111 * 2^3 = 1111.1

15.5

Q5. (15 points)

Consider the following binary representation for an 32-bit IEEE floating point number. What decimal is this?

1	10101010	10111001000000000000000

11010101010111001000000000000000

Helps: Bias: 127.

(128+32+8+2)-127 = 170-127 = 43

1.10111001 * 2^43 *-1

11011100100000000000000000000000000000000000. * -1
-441/256 * 2^43
-1.51526446E13

Q6. (15 points) Using the nearest even rounding, what is the rounded value for each of these number? Consider rounding to 2-bits after the binary point.

10.00011_2 -> **10.00**

10.11100_2 -> **11.00**

10.10100_2 -> **10.10**

Q7. (20 points) Assume a machine with 8 bit for floating point numbers (1 bit for sign, 4 bits for exponent and 3 bits for fraction).
Consider these two encoded representation for two floating point number in such a system :
0 1101 101 and 0 1010 101.

a. What is the result of adding these two numbers. Show the result in encoded format.
b. What is the result of multiplying these two numbers. Show the result in encoded format.

For both parts, include the details of your computation. Note that you should first decode the numbers, then do add or multiplication operation. And then decode again.

A)
1.101 * 2^(13-7) + 1.101* 2^(10-7)
1.101 * 2^(6) + 1.101* 2^(3)
1101000
+ 0001101
= 1110101
1.111 * 2 ^(6)
0 1101 111

B)
(1.101 * 2^(13-7)) * (1.101* 2^(10-7))
(1.101 * 2^(6)) * (1.101* 2^(3))

E = 6 + 3 = 9 (1001)
BIAS = 7 (0111)
9 (1001) + 7 (0111) = 16 (10000)
16 is not a representable exponent with 4 bits and a bias of 7, positive overflow.

0 1111 000