CS201 HOMEWORK 2

- *Attention:*
  - All Problems are presented in a single PDF file
  - Please explain and show the test case for each solution

Coding rule:

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. To this end, we define data type float_bits to be equivalent to unsigned:

```
/* 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. 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.

Problem 1:

Fill in the return value for the following procedure, which tests whether its first argument is less than or equal to its second. Assume the function f2u returns an unsigned 32-bit number having the same bit representation as its floating-point argument. You can assume that neither argument is *NaN*. The two flavors of zero, +0 and -0, are considered equal.

```
int float_le(float x, float y)
{
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);

    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;
    /* Give an expression using only ux, uy, sx, and sy */
    return ;
}
```

Problem 2:

Consider the following two 9-bit floating-point representations based on the IEEE floating-point format.

Format A
There is 1 sign bit.
There are k = 5 exponent bits. The exponent bias is 15.
There are n = 3 fraction bits.

Format B
There is 1 sign bit.
There are k = 4 exponent bits. The exponent bias is 7.
There are n = 4 fraction bits.

In the following table, you are given some bit patterns in format A, and your task is to convert them to the closest value in format B. If rounding is necessary you should round toward $+\infty$. In addition, give the values of numbers given by the format A and format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., 17/64 or 17/26).

Format A		**Format B**	
Bits	Value	Bits	Value
1 01111 001		1 0111 0010	
0 10110 011			
1 00111 010			
0 00000 111			
1 11100 000			
0 10111 100			

<u>Problem 3:</u>

The 2008 version of the IEEE floating-point standard, named IEEE 754-2008, includes a 16-bit "half-precision" floating-point format. It was originally devised by computer graphics companies for storing data in which a higher dynamic range is required than can be achieved with 16-bit integers. This format has 1 sign bit, 5 exponent bits (k = 5), and 10 fraction bits (n = 10). The exponent bias is $2^{5-1} - 1 = 15$.

Fill in the table that follows for each of the numbers given, with the following instructions for each column:

Hex: The four hexadecimal digits describing the encoded form.

M: The value of the significand. This should be a number of the form x or x/y, where x is an integer and y is an integral power of 2. Examples include 0, 67/64, and 1/256.

E: The integer value of the exponent.

V : The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

D: The (possibly approximate) numerical value, as is printed using the %f formatting specification of printf.

As an example, to represent the number 7/8, we would have s = 0, M =7/4, and E = -1. Our number would therefore have an exponent field of 01110_2 (decimal value 15 - 1 = 14) and a significand field of 1100000000_2, giving a hex representation 3B00. The numerical value is 0.875. You need not fill in black cell.

D*escription*	Hex	M	E	V	D
-0	----	----	----	-0	-0.0
Smallest value>2	----	----	----	----	----
512		----	----	512	512.0
Largest denormalize	----	----	----	----	----
$-\infty$	----			$-\infty$	$-\infty$
Number with hex representation	3BB0	----	----	----	----

Problem 4:

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

```
/* Compute 2*f. If f is NaN, then return f. */
float_bits float_twice(float_bits f);
```

For floating-point number f , this function computes 2.0 . f . If f is *NaN*, your function should simply return f . Test your function by evaluating it for all 2^{32} values of argument f and comparing the result to what would be obtained using your machine's floating-point operations.

Problem 5:

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

```
/*
* Compute (int) f.
* If conversion causes overflow or f is NaN, return 0x80000000
*/
int float_f2i(float_bits f);
```

For floating-point number f , this function computes *(int)* f . Your function should round toward zero. If f cannot be represented as an integer (e.g., it is out of range, or it is NaN), then the function should return *0x80000000*. Test your function by evaluating it for all 232 values of argument *f* and comparing the result to what would be obtained using your machine's floating-point operations.