Extended Floating Point Numbers

Goals:

- Learn about IEEE floating point numbers
- Learn about manipulating bits ("bit twiddling") in C
- Understand number type conversions

Background: We will discuss IEEE floating point numbers (float and double) in class. We will learn how to use the IEEE standard to represent floating point numbers in 32 or 64 bits worth of data. Some applications need the flexibility of floating point numbers, but may not need the full precision offered by the standard float (32 bit) or double (64 bit) implementations of floating point numbers. This project asks you implement a generalized floating point number capability – one which uses an arbitrary number of bits, expBits to represent the exponent of a floating point number, and an arbitrary number of bits to represent the fractional part of the number. We will call this generalized floating point number a "floatx" number. Other than changing the number of bits for the sub-fields, all the rest of the IEEE conventions for floating point numbers must be followed:

- The left-most (most significant) bit is a sign bit – 0 for positive, 1 for negative
- The next exBits bits represent an exponent field, biased by [$2^{(\text{expBits-1})}$ – 1]
- Exponent bits of all ones represents special values like infinity (if fraction bits are all zero), or "Not a Number" (Nan) (if fraction bits are not all zero.)
- The remaining bits represent a fraction field, "F". For most numbers, there is an implied "1." in front of the fraction so we interpret it as "1.F".
- If the exponent field and the fraction field are all zeroes, that represents the number 0.0.
- If the exponent field is all zeroes, but the fraction field is not zero, then we treat the number as a "denormalized" number, and assume it has the value of 0.F x $2^{-\text{bias}}$

Specifications: You will be given the C code for a main function, and skeleton functions in an auxiliary file called "floatx.c" with definitions in "floatx.h" to convert a double value into a floatx format value, and to convert a floatx format value into a double value.

The "main" function takes two positive integer arguments:

- The total number of bits to be used for the floatx number – This must be less than 65.
- The number of bits to be used for the exponent of the floatx number.

The main function provided to you will check and format these parameters into the following structure (defined for you in "floatx.h"):

```
typedef struct {
    unsigned int totBits;
    unsigned int expBits;
} floatxDef;
```

After parsing the arguments, the main function will read double values from standard input. For each double value, the main function will then call:

```
floatx doubleToFloatx(const floatxDef * def,double value);
```

The main function will then pass the resulting floatx number to your second function:

```
double floatxToDouble(const floatxDef * def,floatx result);
```

The program will then print a message that shows:

- The input double value
- Your floatx value in hexadecimal
- The result of the floatxToDouble calculation invoked on your floatx value

The type "floatx" is defined in "floatx.h" as follows:

```
typedef unsigned long floatx;
```

Note that this gives 64 bits to represent a floatx number, even though the total number of bits is often less than 64. You must right justify your floatx value within this 64 bit area, and pad to the left with zero bits. So, for example, if you are given totBits=12, and expBits=5, then the resulting floatx number would have the format:

63	…	12	11	10	9	8	7	6	5	4	3	2	1	0
P	…	P	S	E	E	E	E	E	F	F	F	F	F	F
Pad				Exp					Frac					

The following is an example log from a correct implementation of this program:

```
~/CS220/proj2> make test
./floatx 32 8 <test1.txt
Input : 1 = 3f800000 (floatx hex), round trip=1
Input : 100 = 42c80000 (floatx hex), round trip=100
Input : 1.3333333 = 3faaaaab (floatx hex), round trip=1.3333334
Input : 1.27563e+45 = 7f800000 (floatx hex), round trip=inf
Input : 6.23e+22 = 655314a3 (floatx hex), round trip=6.2300001e+22
~~Input : 7.9e-39 = 005605fd (floatx hex), round trip=7.9000009e-39~~
Input : 7.9e-39 = 005605fc (floatx hex), round trip=7.8999995e-39
~/CS220/proj2>
```

Implementation: Download the proj2.tar.gz file from the class web site, and expand it by using the command "tar –xvzf proj2.tar.gz". This will create a subdirectory of your current directory called "proj2", which should contain:

- main.c – The implementation of the "main" function provided to you. Please do not change anything in this file.
- floatx.h – Definitions for floatx provided to you. Please do not change anything in this file.
- floatx.c –This is the file you need to edit to define the functions doubleToFloatx and floatxToDouble.
- Makefile – A makefile so that after editing, you can run "make" to build and test the floatx command, or make submit to build the tar file to submit.
- test1.txt – A list of numbers used as input in the example above.

Edit the floatx.c file, build the floatx command using make, and test your implementation.

Submission: Once you have completed implementing and testing your project, run the command "make submit". This will collect the files in your directory into a file called "proj2_<userid>.tar.gz", where <userid> is your LDAP user ID. Turn this tar file in on myCourses.

Grading: Your project will be graded on LDAP machines using the following criteria…

Criteria	Score Impact
No submission	-100 points
Plagiarism	-100 points
Compiler Errors	-75 points
Submission error (files missing, tar file incorrect, etc.)	-20 points
Compiler Warnings	-10 points per warning
Mismatch on published test case	-30 points
Mismatch on unpublished tests	-15 points / test (3 tests)
Incomprehensible/Inelegant Code	Max -10 points

Hints:

- Remember that in C, the right shift operator, >>, behaves differently on signed and unsigned data types.
- Constants in C are ints (32 bits long) by default. To get long constants, either use an "L" prefix, or cast the constant to a long type, e.g. "1L<<numShift" or (floatx)1<<numShift.
- The implementation of scanf on most machines, including the LDAP machines, does not support "NaN" correctly. Therefore, test cases will not include NaN (although it's not that hard to support.)
- Write your code wherever you want to (e.g. LDAP or Cygwin or on your dual boot laptop), but test it on an LDAP machine before you submit. Not all UNIX implementations are exactly the same!
- Test your code with test cases different than the published log. You will be graded on unpublished test cases, so try to think about how to handle all situations.
- Use the online floating point calculators reference from the home page to check your results. Specifically, when you use the specification of "32 8", you can check your results against the calculator results for floats.