CSCI 463 - Assignment 2
Bitwise Operators & IEEE-754 Floating Point Number Decoding

Read 32-bit HEX representations of IEEE-754 Floating Point Numbers, decode them, and print their componenets.

Setup

Using the prior assignment as a reference, create a directory for this assignment and a suitable **`Makefile`** that will compile a program called **`prog2`** from a source file named **`prog2.cpp`**.

Files You Must Write

`prog2.cpp`

Create a single-file application that will read zero or more 32-bit hex values and then print a detailed description of the floating point value that they represent by extracting and showing the sign, exponent, and significand from the 32-bit value in the manner described below.

This file must contain (at least) two functions: **`main()`** and **`printBinFloat(uint32_t x)`**. Your **`main`** must have a read-loop that calls **`printBinFloat`** to do the decoding and printing.

- Do not define any global or static-class variables.
- Do not define or use any floating point variables or operations.
- **`printBinFloat(uint32_t x)`** must use the C bitwise operators to interpret, extract, and shift the fields of the IEEE number as needed to render the output as described below. (In other words, you must not use things like the **`bitset`** STL template class.)
- You must use **`std::cin`** and **`std::cout`** for reading and printing all of your data (in other words, you may *not* use the C printf/scanf/read/write functions, nor the std::fstream, std::ofstream, std::fstream, nor any other method of any kind.)

Sample Test Files

- An example of SOME valid input data can be found here: test.in.txt (*NOTE: It is your responsibility to create your own robust set of test data for this assignment.*
- Corresponding output created from the above input data example can be found here: test.out.txt

NOTE: In order to receive full credit, your program is expected to generate precisely indentical output in the exact same format (space for space) as the output file given above.

See below for a discussion on how to compare yours against the given reference.

Input

Read your input from **`stdin`** (aka **`std::cin`**).

To read a 32-bit hexadecimal value from **`stdin`** use the **`std::hex`** manipulator with the **`std::cin`** stream as shown below:

```
uint32_t  x;

std::cin >> std::hex >> x;
```

Output

Write your output to **`stdout`** (aka **`std::cout`**).

`printBinFloat` must format and print each value precisely in the following manner.

```
0x3f800000 = 0011 1111 1000 0000 0000 0000 0000 0000
sign: 0
```

```
 exp: 0x00000000 (0)
 sig: 0x00000000
+1.0000000000000000000000
```

In this particular example, the hex value `3f800000` was read and the values of its fields are shown as well as its actual binary value.

For each input value, the five output lines printed are:

- The first line displays the input value as a 32-bit hex value and again in binary (in groups of 4-bits.)
- The sign bit is zero. (Therefore the floating point value is positive.)
- The exponenet is zero (seen printed as a 32-bit signed integer and as a signed decimal value in parenthesis.)
- The significand is printed as a 32-bit unsigned hex value. In this case it is zero.
- The full value of the number is printed last in binary.

Positive infinity must be displayed like this: `+inf` and negative infinity: `-inf` as shown below:

```
0x7f800000 = 0111 1111 1000 0000 0000 0000 0000 0000
sign: 0
 exp: 0x00000080 (128)
 sig: 0x00000000
+inf
0xff800000 = 1111 1111 1000 0000 0000 0000 0000 0000
sign: 1
 exp: 0x00000080 (128)
 sig: 0x00000000
-inf
```

The positive and negative zero values must be printed like this:

```
0x00000000 = 0000 0000 0000 0000 0000 0000 0000 0000
sign: 0
 exp: 0xffffff81 (-127)
 sig: 0x00000000
+0
0x80000000 = 1000 0000 0000 0000 0000 0000 0000 0000
sign: 1
 exp: 0xffffff81 (-127)
 sig: 0x00000000
-0
```

Some other examples show how to display values with larger exponents:

```
0xbf400001 = 1011 1111 0100 0000 0000 0000 0000 0001
sign: 1
 exp: 0xffffffff (-1)
 sig: 0x00400001
-0.110000000000000000000001
0xf7ffffff = 1111 0111 1111 1111 1111 1111 1111 1111
sign: 1
 exp: 0x00000070 (112)
 sig: 0x007fffff
-11111111111111111111111100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000.0
0x00ffffff = 0000 0000 1111 1111 1111 1111 1111 1111
sign: 0
 exp: 0xffffff82 (-126)
 sig: 0x007fffff
+0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111111111111111111111111
```

Documentation

Document your code using the Doxygen rules as described in lecture, course syllabus, and links that can be found on the CSCI 463 faculty page where you found this assignment.

You will lose ALL your documentation points if you do not provide proper Doxygen format including, when appropriate, the **`@param`**, **`@return`**, and other things as discussed in lecture.

Handing In Your Assignment

Hand in your assignment by using the **`mailprog.463`** script in the manner described in lecture.

Grading

We will compile your program on **`hopper`** EXACTLY like this:

```
g++ -Wall -Werror -std=c++11 prog2.cpp -o prog2
```

If the above compilation fails to create an executable file then you will receive zero points for the assignment.

If the compilation succeeds then your program will be run and its output compared against the reference output EXACTLY like this:

```
./prog2 < grade-data.in > student.out
diff grade-data.out student.out
```

If there are ANY lines of output that are printed from the diff command then your output is wrong and a large portion (possibly all) of your output grade points will be deducted.

You will lose many coding points if you read your input data or write your output data in manner that is messier than the clear and simple methods that are discussed in detail in the "*Reading, Printing, and Formatting Numbers in C++*" review lectures on the course web page.

Your program will be graded using different test data than that given in the assignment handouts. Your program MUST be able to handle reading an empty file as well as files with varying numbers of data values.

Hints

- See the RVALP appendix on floating point numbers (as seen in lecture) for details on the format and use of the bits in a 32-bit floating point number.
- To print the 32-bit binary value on the first output line, use an AND-mask with a single 1-bit in the position you want to display. For example, to print a 1 if the Most Significant Bit of a 32-bit variable **`x`** is a 1 and a 0 when it is 0, use logic like this:

  ```
  uint32_t x;
  ...
  std::cout << (x & 0x80000000 ? '1':'0');
  ```

  - You'll want to do this in a suitable loop to print the 32-digits. To do that, put the **`0x80000000`** in another variable and shift it to the right in your loop.
  - You will lose ALL your coding points if you hard-code 32 lines of print logic to print the 32-bit display of the input data.
- Don't even ***think*** about formatting the final binary value until you know you have extracted and properly interpreted each of the **`sign`**, **`exponent`**, and **`significand`** values! ...the easiest test values to start with are the special cases (for them, just hard-code an **`if-then`**, **`else-if`**, **`else-if`**... to detect them and print the various zero and infinity values.)
- Consider using a text editor that shows the cursor's character column number to view your output so you don't have to count dozens of zeros to make sure your exponents are applied properly!
- Learn how to use **`hexdump -C`** on **`hopper`** and **`turing`** to inspect your program's output to make SURE there are no extra spaces, null characters or other garbage that will result in a failure to match the reference output.
- Don't forget to account for the implied **`1`** on the left end of the significand!!!!
- Use the **`man`** command on **`hopper`** to read the doc on how to use command-line programs like **`hexdump`**, **`diff`** and others like this:

  ```
  winans@hopper:~$ man hexdump
  HEXDUMP(1)                          BSD General Commands Manual                          HEXDUMP(1)

  NAME
       hexdump, hd -- ASCII, decimal, hexadecimal, octal dump

  SYNOPSIS
  ```

```
     hexdump [-bcCdovx] [-e format_string] [-f format_file] [-n length] [-s offset] file ...
     hd [-bcdovx] [-e format_string] [-f format_file] [-n length] [-s offset] file ...

DESCRIPTION
     The hexdump utility is a filter which displays the specified files, or the standard input, if no files are
     ...
```

Last modified: 2022-08-18 13:52:50 CDT
