CSE 6010

Final Project – Option 2

New Floating-Point Standards

Choose Option 1 or Option 2 for your final project

If you submit both Option 1 and Option 2, the higher grade will be counted toward as your final project and the lower grade will count as the "Substitution Assignment" explained previously

NO Initial Submission

NO 48-hour grace period

Final Submission Due Date: 11:59pm on Monday, December 11

(Serves as a final exam; Monday, December 11 is our final exam day)

Submit codes as described herein to Gradescope through Canvas

Updates from PREVIEW version highlighted in yellow

In this assignment, you will write a program in C that will implement a new standard for representing floating-point numbers. To do so, you will take user-provided values representing the total number of bits in the representation and the number of bits devoted to the mantissa. Using that information, you will perform various user-requested operations such as converting numbers represented as bit strings to their base-10 values.

You may test your code against the **autograder** for the final submission; you may submit as many times as you like.

Specifications:

- Input and command-line arguments: Your program should take 3-5 command-line arguments:
  - *First command-line argument **(required)**:* the **total number of bits** in the number. You must validate that the total number of bits specified is at least 8 bits and at most 64 bits.
  - *Second command-line argument **(required)**:* the number of bits devoted to the **mantissa**. You must perform two steps of validation regarding this entry. First, it must be at least 1. Second, because the representation includes a sign bit and at must also include at least one exponent bit, you must require that the mantissa is not too long—i.e., the number of bits devoted to the exponent must be positive (you get to figure out the exact validation condition).
  - *Third command-line argument **(required)**:* the operation to perform, as a string. Options (which will be described below) include `convert`, `minmax`, and `addhex`. If a different operation is specified, the program should print a suitable error message and exit.
  - Additional command-line arguments will depend on the third command-line argument (operation to perform). Specifically, `convert` will take a single additional command-line argument, the bitstring to convert; `minmax` will not take any

additional command-line arguments, and `addhex` will take two hexadecimal arguments. See "Operations" section below for details including validation.

- Floating-point standards:
  - The first bit of each bit-string will be a sign bit following the convention you have seen in this class: 0 indicates positive sign and 1 indicates negative sign. There will be two representations for 0 (+0 and -0), as in the widely used floating-point standards.
  - The mantissa will have the given number of bits and will be at the end of the bit-string. The rightmost bit will be the least significant. Both are similar to what was described in class.
  - Between the sign bit and the mantissa bits are the exponent bits, with the number of bits devoted to the exponent being whatever is leftover from the total number of bits after accounting for the mantissa bits and the sign bit. You should assume an exponent bias in the spirit of what we discussed in class. In that case, 8 exponent bits for a floating-point number were associated with a bias of $127 = 2^{8-1} - 1$. Since you are generalizing, here, you should consider that if the exponent has m bits, the exponent bias should be specified as $2^{m-1} - 1$.
  - As discussed in class for 32-bit floating-point numbers, you should assume that the number is normalized. That is, if the mantissa bits are 0101, you should interpret that as 1.0101 (then appropriately account for the exponent and sign).
  - You should also assume the following special representations:
    - If the exponent has 1's for all bits (largest possible value) and the mantissa has 0's for all bits, the number represented is Infinity (Inf), with the sign given by the sign bit.
    - If the exponent has 1's for all bits and the mantissa does not have 0's for all bits (any other value), the number represented is NaN (not a number—e.g., the result for division by 0).
- Operations:
  - If the third command-line argument is specified as `convert`, your code should read in an additional command-line argument that is a bitstring. You may assume that the string is composed of 0's and 1's only, but you should check that the number of bits in the bitstring matches the number of bits given as the first command-line argument. The code should then convert the number to base-10, following the specifications above, and output the result. E.g., if the result was 2.5, your code should output the following.
    ```
    2.5
    ```
  - If the third command-line argument is specified as `minmax`, your code should output the minimum positive base-10 number that can be represented and the maximum positive base-10 number that can be represented, without reaching infinity. Use the `%.40f` format specifier for the minimum value (which may be quite small) and `%f` for the maximum. For example, if the minimum and maximum values are 0.15625 and 14, respectively, your code should output the following.
    ```
    Min Positive: 0.1562500000000000000000000000000000000000
    Max: 14.000000
    ```
  - If the third command-line argument is specified as `addhex`, your code should read in two additional command-line arguments that are hexadecimal strings. Your

program should then convert the numbers to bit strings and interpret those bitstrings according to the new floating-point standard. Perform the addition by adding the base-10 representations of the numbers, then convert back to a bitstring and to a hexadecimal string. Your code should give output from all these steps. (Use sign extension to pad with leading 0's or 1's as needed.) For example, if we were working with unsigned integers (which we are not), your code might output the following if given 1A and 2B on the command line.

```
Binary 1: 00011010
Binary 2: 00101011
Value 1: 26
Value 2: 43
Sum: 69
Binary Sum: 01000101
Hex Sum: 45
```

  - Note that you should be able to perform basic addition with representations for +Inf, -Inf, and NaN. The specific rules to implement are the following.
    +Inf + (any value OR +Inf) = +Inf
    -Inf + (any value OR -Inf) = -Inf
    +Inf + -Inf = NaN
    NaN + (any value OR +Inf OR -Inf) = NaN

- Other notes:
  - Recall that a string is a character array. In addition, recall that a string should include one extra character known as the ***"end of string" character*** that can be specified as `\0`.
  - You likely will want to compare strings to know which operation should be performed. You can use the function `strncmp(str1,str2,nchars)` to do so. This function will return 0 if the first `nchars` of the two strings are equal, -1 if `str1` precedes `str2` lexicographically, and 1 otherwise (`str2` precedes `str1`). Don't forget to `#include <string.h>` to use this function.
  - Another function that may useful is `strcat(str1,str2)`, which concatenates the two input strings and stores the result in `str1`.
  - In addition, you may use `strlen()`, which returns the length of the input string.
  - Be sure to free any dynamically allocated memory.

To receive full credit, your code must be well structured and documented so that it is easy to understand. Be sure to include comments that explain your code statements and structure.

Submission: You should submit to the "New Float" Final Project option in Gradescope through Canvas the following files:

(1) your code (all .c and .h files); no specific filenames are required for your code but ***note the specification for your executable file below***.

(2) the **Makefile** you use to compile your program such that your executable file that will run your program is named '**newfloat**'. You may use the Makefile provided for previous assignments as a starting place. **Don't forget to include any needed compiler flags!**

(3) a series of 2-3 slides **saved as a PDF** and named **slides.pdf**, structured as follows:

- Slide 1: your name and a brief explanation of how you developed/structured your program. This should not be a recitation of material included in this assignment document but should focus on the main structural and functional elements of your program (e.g., the purpose of any loops you used, the purpose of any if statements you used to change the flow of the program, the purpose of any functions you created, etc.). The goal is to help us understand how you developed your code and your intentions. You are limited to one slide.
- Slides 2-3: a discussion of why you believe your program functions properly.

Some hints:

- Start early!
- Identify a logical sequence for implementing the desired functionality. Then implement one piece at a time and verify each piece works properly before proceeding to implement the next.