CS 367 Project #1 - Fall 2020:

Floating Point Representation

Due: Friday, October 9th 11:59pm

This is to be an individual effort. No partners.
No late work allowed after 48 hours; each day late automatically uses one token.

1. Project Overview:

In class, we talked about the IEEE standard for floating point representation and did examples using different sizes for exponent and fraction fields so that you could learn how to do the conversions.

For this assignment, you are going to write several functions in C to add a custom floating-point system to a scripting language called `MLKY`. Your functions will allow `MLKY` to convert standard C `float` floating point values to a custom bit-level floating point representation for use internally in its language. You will also write code to convert this `MLKY` floating point representation back into standard C float values. Finally, you will write code to perform addition and multiplication of these `MLKY` bit-level floating point representations.

2. Running the MLKY Scripting Language

The main input will be reading in scripts for the `MLKY` scripting language, which we have implemented and provided to you as the starting code. The language will call your functions to implement floating point support for these scripts. The `MLKY` language is very simple with only 5 different kinds of statements: `assignment`, `print`, `display`, `add`, and `multiply`. `MLKY` scripts also only support single-letter variables, as shown below.

Here is an example `MLKY` script (one statement per line):

```
zeus-1:P1$ cat sampleprogram.mlky
x = 0.26
print x
y = -15.25
print y
a = x + y
print a
z = x * y
print z
o = 1.0
print o
display o
```

This script file represents entering 11 statements into the `MLKY` interpreter. The output, displayed on the next page, shows that some of these values are not representable directly due to the precision of the custom floating point system, but will be rounded.

3. Output from the MLKY Scripting Language

The output given to you when running `MLKY` will be the current values of the given variables at the `print` statements or the bit representation of the `MLKY` floating point encoding for `display` statements.

For the script in Section 2, the output would be:

```
> > x =                   0.25976562500000000000000000000000000000
> > y =                 -15.25000000000000000000000000000000000000
> > a =                 -14.96875000000000000000000000000000000000
> > z =                  -3.96093750000000000000000000000000000000
> > o =                   1.00000000000000000000000000000000000000
> o = 001111100000000
> Exiting
```

Note you will only get an output to the screen for `display` or `print` commands only.

You also have the option of entering the commands directly from the command line by running the fp program without a script file (like with a Python interpreter). This will prompt you to enter commands using the > prompt.

```
zeus-1:P1$ ./fp
> a = 1.5
> b = 3.2
> c = a + b
> print c
c =                  4.68750000000000000000000000000000000000
> display c
c = 010000100101100
>
```

In the above display, we are interactively assigning values to variables `a` and `b`, then we are doing arithmetic and assigning the sum to `c`, and finally we are printing out `c`.

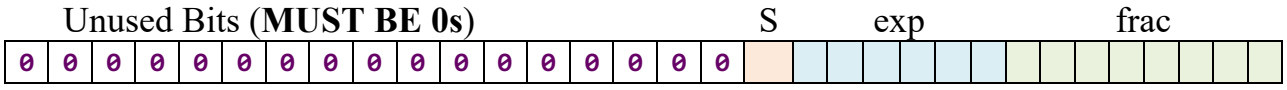
4. Project Code Specification

We provide a program that reads in the given `MLKY` scripts, saves the values (as integers that encode the corresponding bit-level representation in our floating-point format) and calls your functions (described next) that you will be implementing. You will only need to modify **fp_functs.c**.

Your functions will convert the user-input `float` values into a custom 15-bit floating point encoding, which will be of type `fp_gmu`. This custom `fp_gmu` type is stored as a 32-bit integer in memory, so you can work with it just like you would work with a signed int. (e.g. Bitwise operations and Shifting). You will be getting the S, M, and E values from the input and convert them to the S, exp, and frac fields. You will then encode these fields into the `fp_gmu` type. (Using bitwise operations and shifting).

MLKY Floating Point Bit-Level Representation

We encode the MLKY Floating Point values within a signed 32-bit **int** in this format:



The int is typecast as a `fp_gmu` type for this project.

You are going to implement this 15-bit floating point representation:
1 bit for sign (s), 6 bits for exponent (exp) and 8 bits for fraction (frac)

Functions to Implement

`fp_functs.c` starting file has stubs for four functions, which you will be implementing. You are free to add any extra functions you like, but you can only modify `fp_functs.c`

Using bit-level operators, you will write the code for the functions (shown below):

Assignment statement (`variable = value`) – this operation calls your function:

```
fp_gmu compute_fp(float val){}
```

`compute_fp` converts input from a standard C `float` to our custom 15-bit mini-float representation `fp_gmu` (which only uses the **15** lowest of the given 32-bits of an int).

`fp_gmu` is a custom type for our representation. It is actually a 32-bit signed integer.

The return value of the `compute_fp` function will be the `fp_gmu`, which encodes the corresponding bit representation of our `MLKY` floating-point value.

For example, if a negative floating-point number is represented by the "exp" field expressed in bits as `011111`, and the "frac" field expressed in bits as `00000000`, then the integer that must be returned is the one that corresponds to the 32-bit pattern `0000 0000 0000 0000 0101 1111 0000 0000` – specifically `0x00005F00`.

Observe how the "s", "exp" and "frac" bits are preceded by a sequence of leading 0s to make the representation 32 bits that fit within a `fp_gmu` type (int).

Given the number of bits in frac, the rounding you will have to do for this representation may be substantial. In this assignment, we will **truncate** for the fraction. This is the same as rounding to zero.

For example, the closest representable value for `56.24` can be determined by looking at the all_values program output (detailed later in this document).

The relevant output is below:

```
M = 1.753906 (b1.11000001), val=56.125000000000000000000000000000000000000
M = 1.757812 (b1.11000010), val=56.250000000000000000000000000000000000000
```

This shows that 56.125 and 56.25 are representable with our 15-bit floating-point representation, however, 56.24 is not. When rounding, you will use **round-to-zero** (**truncation)**, which will round to the value that's closest to 0.

This means that when 56.24 is converted to the binary floating-point representation in our format, some precision is lost, and the resulting bit pattern corresponds to 56.125, which is the closer value to 0.

Handling cases in `compute_fp()`

- **For Underflows** (eg. Denormalized or 0), compute the Denormalized representation for the value.
  - There are no underflows for Denormalized values. If your number is smaller than the smallest representable Denormalized value, then it will round-to-zero and become either 0 or -0.
  - 0 is a Denormalized Value.
- **For Overflows** (eg. exp is too large for Normalized), return the `fp_gmu` floating-point representation of the special value ∞ *or* $-\infty$ as needed.

Print statement (`print variable`)

This uses your `float get_fp()` function to convert from our `fp_gmu` floating-point representation to a regular C `float` value and returns it as a C `float`.

```
float get_fp(fp_gmu val) {}
```

- **For** $\infty/-\infty$, return the pre-defined C `float` constant **INFINITY**
  - In C, this is simply one of these two following lines exactly:
    `return INFINITY;`
    `return -INFINITY;`
  - Note: the INFINITY constant will **not** work with your `fp_gmu` floating-point representation. It is how C `float`s store infinity.
- **For NaN,** return the pre-defined C `float` constant **NAN**
  - In C, this is simply this line exactly:
    `return NAN;`
  - There is no such thing as negative Not a Number. So, it doesn't matter what the sign bit is, you will return NAN.

Add statement (`variable = variable + variable`)

```
fp_gmu add_vals(fp_gmu source1, fp_gmu source2) {}
```

For this statement, you are going to take two values in our `fp_gmu` floating-point representation and use the same technique as described in class to add these values and return the result converted back into our representation.

> Align M2 by making E2 equal to E1 (or vice versa), then M = M1+M2, E = E1, then encode your S, M, and E back into a fp_gmu value.

- **Rounding** will be through round-to-zero (as with **compute_fp**)
- **For Overflows** (eg. exp is too large for Normalized), return the `fp_gmu` floating-point representation of the special value $\infty \; or \; -\infty$
- **For Underflows**: Handle these as Denormalized Values.
- **Special Cases:** See the next section on **Arithmetic Rules**

When implementing this statement, **DO NOT** convert the numbers back to C floats, add them directly as C floats, and then convert to the new representation (doing so will not bring any credit).

You must work with the S, M, and E components of the representations as described above. You may, however, work with M as a C `float`.

Multiply statement (`variable = variable * variable`)

```
fp_gmu mult_vals(fp_gmu source1, fp_gmu source2) {}
```

For this statement, you are going to take two values in our `fp_gmu` representation and use the same technique as described in class to multiply these values and return the result in our representation.

M = M1*M2, E=E1+E2, S=S1^S2, then encode S, M, E back to a `fp_gmu` value.

- **Rounding** will be through round-to-zero (as with **compute_fp**)
- **For Overflows** (eg. exp is too large for Normalized), return the `fp_gmu` floating-point representation of the special value ∞ *or* $-\infty$
- **For Underflows**: Handle these as Denormalized Values.
- **Special Cases:** See the next section on **Arithmetic Rules**

When implementing this statement, **DO NOT** convert the numbers back to C floats, multiply directly as C floats, and then convert to the new representation (doing so will not bring any credit).

You must work with the S, M, and E components of the representations as described above. You may, however, work with M as a C `float`.

Display statement (`display variable`)

Nothing to do here, it's all done for you!

5. Arithmetic Rules

Here are a list of rules for when you are multiplying or adding values.

1. **NaN** *(You may use either 0 or 1 for the Sign Bit. It will be Ignored!)*
   - If you add $\infty + -\infty$ (either order), return your `fp_gmu` NaN.
   - If you multiply ∞ (or -∞) by 0 return your `fp_gmu` NaN.
   - If either argument is NaN, return your `fp_gmu` NaN
2. **Infinity**
   - If you add any value (non NaN, -∞) to ∞, return ∞.
   - If you add any value (non NaN, ∞) to $-\infty$, return $-\infty$.

- If you multiply any non-Zero, non NaN value to $\infty \; or \; -\infty$, return $\infty \; or \; -\infty$ as appropriate based on the sign rules for multiplication

3. **Zero**
   - If you multiply 0 or -0 by non NaN, non Infinity, return either 0 or -0 *(based on the normal sign rules for multiplication)*
   - If you add -0 + -0, you will return 0.
   - If you add -0 + 0, or vice versa, you will return 0.
   - If you add 0 or -0 to any value that is non NaN, non Infinity, you will return that other value.
   - If you multiply 0 * -0, or vice versa, you will return either 0 or -0 *(based on the normal sign rules for multiplication)*

6. Project Constraints

There are Two Special Number Types: Infinity and NaN.

- These will be implemented using a proper special number pattern in your `fp_gmu` floating-point representation. (Remember $\infty \; and \; -\infty$)
- There is only one NaN, regardless of sign, it's not a number.
- For your `get_fp()` function only, you will be converting your `fp_gmu` floating-point representation value back into a standard C `float` type.
  - If your `fp_gmu` representation is storing $\infty \; or \; -\infty$, then you will return using a pre-defined C float constant, **INFINITY**. (or -INFINITY)
    - `return INFINITY;` // or `return -INFINITY;`
    - This INFINITY only works with C `float`s, not with our representation, so it's only used to return from `get_fp()` when the input is ∞
  - If your `fp_gmu` is storing NaN, then you can return NAN;
    - `return NAN;`
    - There is no negative NaN. If the sign is a 1, still `return NAN;`

Denormalized Numbers are part of this assignment.

- When working with arithmetic, it is possible to start with a Normalized number and end with a Denormalized number.
- Always check and encode accordingly.

You may Not use any math.h functions, including pow()
You may Not use any unions in your code.

Negative Numbers must be handled.

- All values (including ∞) will be handled properly with negatives.
- `compute_fp`, `mult_vals`, and `add_vals`, will need to support -0 encoding.
- -0 in your integer-based floating point representation is the expected return value for underflows that occur with a negative value.

(eg. if exp is too small when encoding a negative number, underflow to -0)

When working in your functions, you may work with M as a C float.

- You can use C `float`s in your code to do your work generally.
- The only restriction is that in `add_vals` and `mult_vals`, you can't covert the entire inputs to C `float`s, then just add/multiply them together and convert them back.
- You have to do the operations on the S, M, and E components.
- You can still use normal C `float`s in those functions for your work on M.

7. Getting Started

First, get the starting code (`project1_handout.tar`) from the same place you got this document. Once you un-tar the handout on zeus (using `tar xvf project1_handout.tar`), you will have the following files:

- `fp_functs.c` – **This is the only file you will be modifying (and submitting).** There are stubs in this file for the functions that will be called by the rest of the framework. Feel free to define more functions if you like but put all of your code in this file!

- `Makefile` – to build the assignment (and clean up).

- `README` – read it.

- `fp_program.c` – This is the main program for the assignment. Do **not** change it. It implements a recursive descent parser to read in the program files, determine what each line is supposed to do, and call your functions to convert, add and multiply.

- `all_values` – This is a program we wrote to make debugging easier for us. It prints out all legal values in our representation. This will help you determine what values you should be seeing.

For each E (the E is given and the exp equivalent is given in binary), `all_values` lists all possible values that can be represented (vals). For each val, you get the M in decimal and in binary for convenience.

For example, in the sample program we assign `0.26` to x. This number is not a valid **val** in the output for this program, as shown in the snippet from the all_values output below.

```
...
M = 1.039062 (b1.00001010), val=0.259765625000000000000000000000000000000
M = 1.042969 (b1.00001011), val=0.260742187500000000000000000000000000000
...
```

The closest values to **0.26** are `0.259765625` and `0.2607421875`. Since we're using round-to-zero (truncation), we round down to `0.259765625` as seen in the sample output.

Of course, doing this in decimal is very hard. It's a lot easier once you're working in the code to do the rounding from the binary directly. You'll have your M (which you can work with as a `float` in C) and you'll have to find a way to get the first 8 bits of the fraction part of M as an integer for encoding in frac.

Sample Test with easier test values for rounding:

`x = -127.74`	This will round down to -127.5
`y = -1.0`	This is directly representable.
`z = x + y`	The true answer is -128.5, which is representable.
`print z`	This will print -128.5

- `samplescript.mlky` – The sample script used in this document.

- `fpParse.h` , `fp.h`, and `fp.l` – You can ignore these files - They are the Lex specification which tokenizes input and sends it to the recursive descent parser in the main program.

8. Implementation Notes

- `MLKY` Script Files – The accepted syntax is very simplistic and it should be easy to write your own scripts to test your code (which we strongly encourage).
  - `MLKY` only uses single-letter, lowercase variable names.
  - `MLKY's` only commands are:
    - `print x` where x is a variable to print the value.
    - `display x` where x is a variable to display the bit representation
    - `x = value` for some floating point value. Performs assignment.
    - `x = y + z` for any legal variable names to add variables.
    - `x = y * z` for any legal variable names to multiply variables.
- If you run fp from the command line without inputting a script file, you can end the session by pressing enter with no input.
- To run fp with a script file, use redirects. `./fp < samplescript.mlky`

9. Submitting & Grading

Submit this assignment electronically on Blackboard. Note that the only file that gets submitted is **`fp_functs.c`**. Make sure to put your name and G# as a commented line in the beginning of your program `fp_functs.c`

You can make multiple submissions; but we will test and grade ONLY the latest version that you submit (with the corresponding late penalty, if applicable).

Important: Make sure to submit the correct version of your file on Blackboard! Submitting the correct version late will incur a late penalty; and submitting the correct version 48 hours after the due date will not bring any credit.

Questions about the specification should be directed to the CS 367 Piazza forum.

Your grade will be determined as follows:

- **20 points** - code & comments. Be sure to document your design clearly in your code comments. This score will be based on reading your source code.
- **80 points** – correctness. We will be building your code using the `fp_functs.c` code you submit along with our code.
  - If you program does not compile, **we cannot grade it**.
  - If your program compiles but does not run, **we cannot grade it.**
  - We will give partial credit for incomplete programs that build and run.
  - You will not get credit for a particular part of the assignment (multiplication for example), if you do not use the required techniques, even if your program performs correctly on the test cases for this part.