CS 367 Project #1 - Fall 2019:
Floating Point Representation
Due: Friday, October 4 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.

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 code to do this, allowing you to convert the floating point numbers to a bit-level representation. You will also write code to perform addition and multiplication of floating point vals.

INPUT: You will read in a script for the MLKY scripting language, which we have implemented and provided to you as the starting code, and call your functions to implement floating point support for these scripts. The MLKY language is very simple with only 4 different kinds of statements: assignment, print, add and multiply. An example of a program is given below:

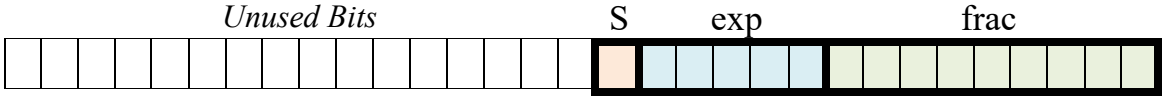
```
zeus-1:P1$ cat sampleprogram.mlky
x = 0.26
print x
y = -15.25
print y
a = x + y
print a
z = x * y
print z
```

OUTPUT: The output will be the current values of the given variables at the print statements. For the above script, the output would be:

```
zeus-1:P1$ ./fp < sampleprogram.mlky
> > x =     0.25976562500000000000000000
> > y =   -15.25000000000000000000000000
> > a =   -14.98437500000000000000000000
> > z =    -3.96093750000000000000000000
> Exiting
```

Some of this task is already done for you. We will 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 the functions (described next) that you will be implementing.

Encoding of our smaller Floating Point within a 32-bit **int** is in this format:



You are going to implement this 15-bit floating point representation, where 5 bits are for the exponent (exp) and 9 are for the fraction (frac).

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

- **Assignment statement** (`variable = value`) – this operation calls your function `compute_fp()`, which converts from a C float value to our 15-bit mini-float representation (which only uses the **15** lowest of the given 32 bits in an integer). The return value of the function will be the 32-bit integer that encodes the corresponding bit representation.

  For example, if a floating-point number is represented by the "exp" field expressed in bits as 00100, and the "frac" field expressed in bits as "0 0000 0001", then the integer that must be returned is the one that corresponds to the 32-bit pattern "0000 0000 0000 0000 0000 1000 0000 0001" – specifically 0x00000801. Observe how the "exp" and "frac" bits are preceded by a sequence of leading 0s to make the representation 32 bits that fit within an **int**.

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

  Given the number of bits, the rounding you will have to do for this representation may be substantial. In this assignment, we will simply **truncate** the fraction (i.e., **rounds down for positive** and **up for negative** values)

  For example, the closest representable value for `0.26` (rounding down) is `0.2597656250`, as can be seen in the program output.
  This means that when 0.26 is converted to the binary floating-point representation in our format, some precision is lost, and the resulting bit pattern corresponds to 0.2597656250 when printed by the get_fp() function below.
  - **For Underflows** (eg. Denormalized or 0), return 0

- **Print statement** (print variable) – uses your `get_fp()` function to convert from our mini-float representation to a regular C float value, and formats/prints it out nicely. Return the converted C float.
  - **For Infinity**, return the pre-defined float constant **INFINITY**
  - **For Underflows** (eg. Denormalized or 0), return 0

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

- **Add statement** – for this statement, you are going to take two values in our representation and use the same technique as described in class/comments to add these values and return the result converted back into our representation. *(i.e., if E1 > E2:Align M2, then M = M1+M2, E=E1, and adjust M & E as needed)*
    - **For Underflows** (eg. Denormalized or 0), return 0
    - **Rounding** will be through truncation (as with **compute_fp**)

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

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

- **Multiply statement** – for this statement, you are going to take two values in our representation and use the same technique as described in class/comments to multiply these values and return the result in our representation. *(i.e. M = M1*M2, E=E1+E2, S=S1^S2, and adjust M & E as needed)*
    - **For Underflows** (eg. Denormalized or 0), return 0
    - **Rounding** will be through truncation (as with **compute_fp**)

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

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

Constraints

- **Only one Special Number Type** (Infinity). For your `get_fp()` function, you will be returning the pre-defined float constant **INFINITY** for Infinity.
- **No Denormalized Numbers**. This assignment is only Signed and Special.
- **Negative Numbers must be handled**.
    - We have given you a function, `is_negative_zero(float val)` to help by checking if a float value is -0.0. It will return 1 if val is -0.0 and 0 otherwise.
    - This is because, in C, 0.0 == -0.0, even though they are different encodings and print differently. Use the provided function as needed to check for negative 0.

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 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, val=0.25976562500000000000000000
  M = 1.041016, val=0.26025390625000000000000000
  M = 1.042969, val=0.26074218750000000000000000
  ...
  ```

  The closest smaller **val** is `0.2597656250` – when I see this output, I know that value is being rounded and stored correctly.

- `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.

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.
        - `x = value` for some floating point value. Performs assignment.
        - `x = y + z` for any legal variable names
        - `x = y * z` for any legal variable names
- 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 the redirect operator.
  `./fp < samplescript.mlky`

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.