

# CS 367 Project #1 - Spring 2019:

## Floating Point Representation

### Due: Wednesday, February 27 11:59pm

This is to be an individual effort. No partners.

No late work allowed after 48 hours; each day late automatically uses up one of your tokens.

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 numbers, using the techniques discussed in class.

**INPUT:** You will read in a 'program' and call your functions to implement these programs. The language is very simple with only 4 different kinds of statements: assignment, print, add and multiply. An example of a program is given below:

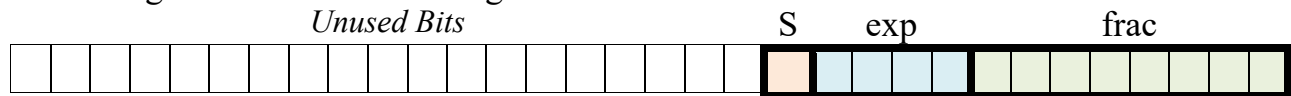
```
zeus-1:P1$ cat sampleprogram
x = 0.26
print x
y = 15.25
print y
a = x + y
print a
z = x * y
print z
```

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

```
zeus-1:P1$ ./fp < sampleprogram
> > x = 0.2597656250
> > y = 15.2500000000
> > a = 15.5000000000
> > z = 3.9609375000
> Exiting
```

Some of this task is already done for you. We will provide a program that reads in the given programs, 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 13-bit floating point representation, where 4 bits are for the exponent (exp) and 8 are for the fraction (frac).**

Using bit-level operators, you will write the code for the functions (shown below) to help implement the program statements:

- **Assignment statement** (`variable = value`) – this operation calls your function `computeFP()`, which converts from a C float value to our 13-bit mini-float representation (which only uses the **13** 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 0100, and the “frac” field expressed in bits as “0000 0001”, then the integer that must be returned is the one that corresponds to the 32-bit pattern “0000 0000 0000 0000 0000 0100 0000 0001” – specifically 0x00000401. 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 computeFP(float val) { }  
// input: float value to be represented  
// output: 32-bit integer that encodes the input float value in our  
// IEEE-like format
```

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., round down).

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 `getFP()` function below.

- **Print statement** (`print variable`) – uses your `getFP()` 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, you can simply return -1)

```
float getFP(int val) { }  
// Using the defined representation, compute and  
// return the floating point value
```

- **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.  $M = M1 * M2$ ,  $E = E1 + E2$ , and adjust  $M$  &  $E$  as needed)

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 addVals(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., if  $E1 > E2$ : Align  $M2$ , then  $M = M1 * M2$ ,  $E = E1$ , and adjust  $M$  &  $E$  as needed)

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

## Assumptions

To make your life a little easier, we are going to make the following assumptions:

- No negative numbers. The sign bit can be ignored (always has a 0 value).
- Only one Special Number (Positive Infinity). For your `getFP()` function, you will be returning a -1 for Infinity.

Note that your program should be able to process/manipulate both normalized and denormalized numbers, as long as they are non-negative.

## Getting Started

First, get the starting code (`prog1_s19.tar`) from the same place you got this document. Once you un-tar the handout on zeus (using `tar xvf prog1_s19.tar`), you will have the following files:

- `fp_funcs.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, both denormalized and normalized! 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=0.259765625000000
M = 1.042969, val=0.260742187500000
M = 1.046875, val=0.261718750000000
...
```

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

- **sampleprogram1**, **sampleprogram2**, **sampleprogram3** – some sample input files. The first two test basic normalized value storage and arithmetic. The third one tests Special and Denormalized cases.
- **fpParse.h**, **fp.h**, and **fp.1** – 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

- Program Files – The accepted syntax is very simplistic and it should be easy to write your own programs to test your code (which we strongly encourage). Variable names are single lower case letters.
- There are 4 different statement types:
  - **print x** - where x is a variable.
  - **x = value** - for some floating point value. This performs assignment
  - **x = y + z** - for any legal variable names
  - **x = y \* z** - for any legal variable names
- There are 26 valid variable names. These are single-character, lowercase letters and range from a to z. No other variable name is valid.

## Submitting & Grading

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

- **30 points** - code & comments. Be sure to document your design clearly in your code comments. This score will be based on reading your source code.
- **70 points** – correctness. We will be building your code using the `fp_funcs.c` code you submit along with our code.
  - If your 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 compile 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.
  - We will use a series of programs to test your solution. Make sure you check that your code performs the correct storage and operation for each of the different cases that it should.