# CS 367 Project 2 - Fall 2021:

**Custom Floating-Point Library**

**Due: Friday, October 1st by 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:

Embedded systems often do not have hardware support for floating point that you can use when programming. A solution for this is to use a software floating point library, which will allow you to get the benefits of floating-point values without having the full support you normally have when programming. Another benefit of this is that we can also create custom floating-point encodings for different purposes, which is something that is done quite often in deep learning.

For this assignment, you will be finishing four functions in **swfp_lib.c** which will be used to provide custom floating-point support in non-standard sizes to other programs. In this Project, your library will be used in the AHA scripting calculator. This calculator uses your library to convert standard **double** type variables into custom **swfp_t** type encodings, and vice versa. It will also do some native math functions in this format, allowing both multiplication and addition efficiently with **swfp_t** values.

## 2. The AHA Scripting Calculator

We have already programmed AHA for you; all you have to do is provide the code for the **swfp** functions in **swfp_lib.c** that the calculator will use. This calculator is normally programmed using scripts, similar to how Python scripts are run. These AHA scripts will run the commands that will ultimately call your functions.

AHA does have the ability to run from the command line without any inputs, in which case it will let you type in your operations, one per line, for it to execute.

AHA uses single-letter, lowercase variables only, so you can only use **a** through **z**
AHA only has three different operators: `=`, `+`, and `*`
AHA also has two commands: **print** and **display**

To quit AHA, you simply have to press the ENTER key when done.

Here is an example **AHA** script with one statement per line: (sample.aha)

```
x = 0.14
print x
y = -5.15
print y
a = x + y
print a
z = x * y
print z
o = 1.0
print o
display o
```

This script inputs 11 commands into the AHA Scripting Calculator.

## 3. AHA Scripting Calculator Output

AHA outputs directly to the screen with the results of the operations or commands that are given. If you run a script, the output will be given all at once, as shown below with the output for the above (sample.aha) script.

```
zeus-1:handout$ ./aha < sample.aha
> > x =                   0.1398925781250000000000000000000000000000
> > y =                  -5.1484375000000000000000000000000000000000
> > a =                  -5.0078125000000000000000000000000000000000
> > z =                  -0.7202148437500000000000000000000000000000
> > o =                   1.0000000000000000000000000000000000000000
> o = 001110000000000
> Exiting
```

Only the **print** or **display** commands will print output to the screen, so these are the results shown for those commands in sample.aha

You can also run it directly from the command line without a script, just like with a Python interpreter. In this case, after every **print** or **display**, you'll get the output.

```
zeus-1:handout$ ./aha
> a = 1.5
> b = 3.2
> c = a + b
> print c
c =                   4.6992187500000000000000000000000000000000
> display c
c = 010010010110011
>
```

## 4. Specification for Project 2

We've already written the AHA Scripting Calculator for you and provided you with the stubs for the four functions you will be writing inside of **swfp_lib.c**

You will be completing this code, along with creating any number of helper functions that you would like to use, in order to implement these four functions. In this project, you will be working with our custom **swfp_t** type variables. These are custom types that are 32-bit signed ints in memory. Within these 32-bits, you will be encoding 15 bits of information in a very specific format. You are writing these functions to make a custom 15-bit floating point format for the AHA Calculator.

Since an **swfp_t** type is just a standard **int**, you can do operations on it just like you normally would with a signed int. (eg. shifting, masking, and other bitwise ops). Ultimately, you will be getting the S, exp, and frac information and storing them within your **swfp_t** value, just like we've been doing in class.

## AHA Representation of swfp_t Values

The **swfp_t** values are encoded using the following format within a signed 32-bit int:

| Unused Bits (**MUST BE 0s**) | | | | | | | | | | | | | | | | | S | exp | | | | frac | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | s | e | e | e | e | f | f | f | f | f | f | f | f | f | f |

This is the 15-bit Representation for **swfp_t** values:
1 bit for sign (s), 4 bits for exponent (exp) and 10 bits for fraction (frac).

Just like normal, we can support Normalized, Denormalized, and Special (NaN or ∞)

The smallest possible value (all bits = 0) is assumed to represent the value 0 as well as values very close to zero. The zero can use the sign bit to represent +0 and -0.

Rules describing the outcomes of arithmetic operations will be discussed later.

## Function Descriptions

**swfp_lib.c** has been given to you as your starting file. This contains as stub for all four functions you are required to implement. It would be a very wise decision to create many helper functions as a part of your design. They too must all be kept within **swfp_lib.c** as this is the only file you will be submitting.

Write the code for these functions, **using bitwise operators** for encoding/decoding.

**AHA Operator:** = `(example: a = 1.23)`

When someone uses the = operator, AHA will call your function:

```
swfp_t double_to_swfp(double val);
```

**double_to_swfp** will convert a standard C **double** into our custom 15-bit representation (using the 15 lowest bits of a signed int) and return that value.

**swfp_t** is a typedef for a signed 32-bit **int** in C

Once you have encoded the value into this **swfp_t**, you will be returning it.

**Example**: The val -1.25 has Sign = 1, exp = 0111, frac = 0100000000

The swfp_t value should be: `0000 0000 0000 0000 0101 1101 0000 0000`
(Hex: `0x5D00` )

**So, how do you get these fields from the value?**
Set up the problem just like we did in class. You can use **double** when working within your function, so you can perform the same operations we did on the board.

The key idea is to get your value into the right range while adjusting E. This will give you the ability to determine the S, E, and M components first.
*(hint: You can't shift a double, but you can do the same operations other ways)*

Note that frac is big (10-bits), however it's not big enough to store all possible values, so we will need to do rounding. The good news is that for this project, you will use **truncation** for rounding. (Only the first 10 bits of frac are used).

As an example, when **254.2** is entered into AHA, it will round down to **254.125**

We can determine what things **should** round to by looking at the output of a helper program, **all_values**. This program prints out every possible value that can be represented in AHA, along with the Mantissa and binary representation.

The relevant output is below:

```
  M = 1.985352 (b1.1111110001), val=254.125000000000000000000000000000000000
  M = 1.986328 (b1.1111110010), val=254.250000000000000000000000000000000000
  M = 1.987305 (b1.1111110011), val=254.375000000000000000000000000000000000
```

So, we can see that **254.125** and **254.250** are both representable, but **254.2** is not. So, since we're rounding by **truncation**, we'll round towards zero.

Truncation works for both positive and negative values by **rounding-to-zero**

So, the closest value rounding towards zero would be **254.125**, which is what AHA expects when **254.2** is entered. Likewise, if we had entered **-254.2**, rounding towards zero would give us **-254.125**.

**Special Rules:**

- Do not worry about -0 on input.
    - C is very bad at recognizing -0 being passed in. If you get an input of 0, you can assume it is positive 0.
- **For Underflows** (eg. exp ≤ 0):
    - This supports Denormalized values, to continue by encoding it as a Denormalized value with your **swfp_t** variable.
- **For Overflows** (eg. exp is too large for Normalized):
    - Encode the **swfp_t** variable as the special value $\infty$ *or* $-\infty$ as needed.

**AHA Command: print** `(example: print a)`

When someone uses the **print** command, AHA will call your function:

```
double swfp_to_double(swfp_t val);
```

**swfp_to_double** will convert our 15-bit representation into a standard C **double**

Extract the S, exp, and frac portions of the **swfp_t** type and then convert them back into a normal C **double** value. You can work with double types when doing these operations.

**Special Rules:**

- **If val is Special and is $\infty$ or $-\infty$ in your swfp_t encoding:**
    - Return the **DOUBLE_INF** constant.
        - This constant has the **double** value for infinity.
        - Make sure to negate this before returning for $-\infty$
    - Note: the **DOUBLE_INF** constant will **not** work with your **swfp_t** floating-point representation. It will only work for **double** types.
- **If val is Special and is NaN in your swfp_t encoding:**
    - Return the **DOUBLE_NAN** constant.
        - This constant has the **double** value for NaN.
        - It will print out as **-nan**, that's ok!
            - NaN isn't a number, so if it may print as **nan** or **-nan**
    - Note: the **DOUBLE_NAN** constant will **not** work with your **swfp_t** floating-point representation. It will only work for **double** types.

**AHA Command: +** `(example: a = b + c)`

When someone uses the + operator, AHA will call your function:

```
swfp_t swfp_add(swfp_t val1, swfp_t val2);
```

**swfp_add** will add two values together and the return the result as a **swfp_t** value.

Extract the S, exp, and frac portions of each one of the two value arguments, convert them into S, E, and M, then add them together using the technique covered in class:

val1: S1, M1, E1
val2: S2, M2, E2
sum: S, M, E

**Align the Mantissas:**

You need both Es to be the same, so pick one of your Ms and adjust it. Once both E1 and E2 are equal, you can now add your Ms.

**M = M1 + M2**

**E = E1**

For the Sign, you will need to determine what it should be.

Example, 5.0 + -3.0 will be positive, while -5.0 + 3.0 will be negative.

Once you have the S, M, and E of the sum, then encode them back into the **swfp_t** format and return that result.

You are allowed to use **double** in C when working with your **M1**, **M2**, and **M** components while doing the work like we did in class.

You are **not allowed** to convert your values back to **double** types, add them, then convert the results into **swfp_t** types. You need to do the work with the Ms, Es, and S components directly.

**Special Rules:**

- **Rounding:**
  - The result may not fit evenly into your format anymore! Make sure to round (**truncation**) to fit into the frac field.
- **For Underflows** (eg. exp $\leq 0$):
  - This supports Denormalized values, to continue by encoding it as a Denormalized value with your **swfp_t** variable.
- **For Overflows** (eg. exp is too large for Normalized):
  - Encode the **swfp_t** variable as the special value $\infty$ *or* $-\infty$ as needed.
- **Sign Considerations:**
  - Adding positive and negative values may require subtraction.
- **Arithmetic Special Cases:** See the next section on **Arithmetic Rules**

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

**AHA Command: *** `(example: a = b * c)`

When someone uses the * operator, AHA will call your function:

```
swfp_t swfp_mul(swfp_t val1, swfp_t val2);
```

**swfp_mul** will multiply two values and the return the result as a **swfp_t** value.

Extract the S, exp, and frac portions of each one of the two value arguments, convert them into S, E, and M, then add them together using the technique covered in class:

| | |
|---|---|
| val1: | S1, M1, E1 |
| val2: | S2, M2, E2 |
| product: | S, M, E |

**S = S1 ^ S2**
**M = M1 * M2**
**E = E1 + E2**

Once you have the S, M, and E of the product, then encode them back into the **swfp_t** format and return that result.

You are allowed to use **double** in C when working with your **M1**, **M2**, and **M** components while doing the work like we did in class.

You are **not allowed** to convert your values back to **double** types, add them, then convert the results into **swfp_t** types. You need to do the work with the Ms, Es, and S components directly.

**Special Rules:**

- **Rounding:**
  - The result may not fit evenly into your format anymore! Make sure to round (**truncation**) to fit into the frac field.
- **For Underflows** (eg. exp $\leq 0$):
  - This supports Denormalized values, to continue by encoding it as a Denormalized value with your **swfp_t** variable.
- **For Overflows** (eg. exp is too large for Normalized):
  - Encode the **swfp_t** variable as the special value $\infty$ *or* $-\infty$ as needed.
- **Sign Considerations:**
  - Even when working with $\infty$, follow the multiplication sign rules.
- **Arithmetic Special Cases:** See the next section on **Arithmetic Rules**

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

**AHA Operator: display** `(example: display a)`

This command shows the binary representation of your **swfp_t** values.
It's great for debugging! It's also written for you, so no work is needed.

**Example:**

```
> c = 1.25
> display c
c = 001110100000000
```

This represents a swfp_t value with these bits:

**0 0111 0100000000**

S = 0, exp = 0111, frac = 0100000000

This shows the bits of the variable you set with your earlier functions.

## Special Arithmetic Rules

Use these rules for special cases when doing arithmetic:

1. **Addition Special Rules** *(Arguments in the table can be in either order)*

| If one argument is… | And the other argument is… | Then return … |
|---|---|---|
| ∞ | ∞ | ∞ |
| ∞ | −∞ | **NaN** |
| ∞ | NaN | **NaN** |
| ∞ | Any Value (Not ∞,−∞, or NaN) | ∞ |
| −∞ | −∞ | −∞ |
| −∞ | NaN | **NaN** |
| −∞ | Any Value (Not ∞,−∞, or NaN) | −∞ |
| 0 | -0 | **0** |
| -0 | -0 | **-0** |
| 0 | Any Non-Zero Value (Not ∞,−∞, or NaN) | **That Value** |
| -0 | Any Non-Zero Value (Not ∞,−∞, or NaN) | **That Value** |

2. **Multiplication Special Rules** *(Arguments in the table can be in either order)*

| If one argument is… | And the other argument is… | Then return … |
|---|---|---|
| ∞ | ∞ | ∞ |
| ∞ | NaN | **NaN** |
| ∞ | Any Value (Not ∞,−∞, or NaN) | ∞ |
| −∞ | NaN | **NaN** |
| −∞ | Any Value (Not ∞,−∞, or NaN) | −∞ |
| ∞ or −∞ | 0 or -0 | NaN |

**Remember the sign rules for any Multiplication!**
(Multiplying by ∞, -∞, 0 or -0 is handled using normal multiplication sign rules)

## Project Constraints

**There are Two Special Number Types: Infinity and NaN.**

- These will be implemented using a proper special number pattern in your **swfp_t** floating-point representation. (Remember $\infty$ $and - \infty$)
- There is only one NaN, regardless of sign, it's not a number.
  - Any pattern which matches a NaN is considered equivalent.
  - Your **swfp_t** inputs should be able to recognize any bit representation of NaN.
  - Your **swfp_mul** and **swfp_add** returns may use any valid NaN bit representation.

**Denormalized Numbers are part of this assignment.**

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

**You may Not use any math.h functions, including pow()**

**You may Not use any unions in your code.**

**You may Not work directly with the bits of a C double.**

**Negative Numbers must be handled.**

- All values (including $\infty$) will be handled properly with negatives.
- **swfp_to_double**, **swfp_mul**, and **swfp_add** all need to support -0 values being passed in as arguments.
- **swfp_mul**, and **swfp_add** need to support -0 values being returned.
- **double_to_swfp** will not have to handle -0 being passed in.

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

- You can use C **doubles** in your code to do your work generally.
- The only restriction is that in **swfp_mul**, and **swfp_add**, you can't convert the entire inputs to C **doubles**, 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 C doubles in those functions for your work on M.

## 5. Input Checking

With **swfp_t** types, we have to do rounding on a lot of our values because they aren't directly representable. In float types in C, the same thing has to happen.

When you enter a value when running **aha** you're actually entering it as a C double, which means that some of the values you type in won't be exactly the same when they get passed into the **double_to_swfp** function. Most of the time the rounding is so close that it won't matter, but there may be occasions where you enter a value and get an unexpected result.

The first thing to do when debugging is to run a provided program, called **input_checker**. This program will be created when you use make and will let you enter a value just like you would on **aha**, but this will show you exactly which value will be passed into your **double_to_swfp** function for that input.

This shows that if you do the following:

```
kandrea@zeus-1:handout$ ./aha
> a = 123.45678
> print a
a =                 123.43750000000000000000000000000000000000
>
```

You can see exactly what value will be passed into your `compute_fp` function.

```
kandrea@zeus-1:handout$ ./input_checker
.---------------------------------------------------
| Input Checker for Project 2 (Floating Point)
+---------------------------------------------------
| When you type in a value to AHA, it will be
|  converted to a double before it is passed in
|  to your double_to_swfp function.  Since doubles
|  also round, the number passed in to double_to_swfp
|  may be different than your input!
|
| This program shows you the actual values passed
|  into double_to_swfp when you give an input to AHA.
| Enter a value to check, or 0 to quit.
+---------------------------------------------------
| Enter a value to check: 123.45678
| You entered:       123.45678
| Calling double_to_swfp(123.4567799999999485680746147409081459045410)
+---------------------------------------------------
| Enter a value to check:
```

So, we can see here that your **double_to_swfp** function will get `123.456779999999994856807461474090814590454 1` passed into it. Usually, this is fine, but for some values it may account for any discrepancies you may see.

## 6. Getting Started

First, get the starting code (`project2_handout.tar`) from the same place you got this document. Once you un-tar the handout on zeus (using `tar xvf project2_handout.tar`), you will have the following files:

- **swfp_lib.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).

- **fp_program.c** – This is the main code for the **aha** program. 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 `0.14` 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 = 1.119141 (b1.0001111010), val=0.139892578125000000000000000000000000
  M = 1.120117 (b1.0001111011), val=0.140014648437500000000000000000000000
...
```

  The closest values to **0.14** are `0.139892578125` **and** `0.1400146484375`. Since we're using round-to-zero (truncation), we round down to `0.139892578125` 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 **double** in C) and you'll have to find a way to get the first 10 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 to -127.6875 |
| `y = -1.0` | This is directly representable. |
| `z = x + y` | The true answer is -128.6875, which is **not** representable. |
| `print z` | This will print -128.625 (after rounding) |

- **sample.aha** – The sample script used in this document.
- **input_checker** – A program to see what's being passed into **double_to_swfp**
- **fp_parse.h, fp.h, 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.

## 7. Implementation Notes

- **aha** 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).
    - **aha** only uses single-letter, lowercase variable names.
    - **aha** 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 **aha** from the command line without inputting a script file, you can end the session by pressing enter with no input.
- To run **aha** with a script file, use redirects. `./aha < sample.aha`

## 8. Submitting & Grading

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

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 **swfp_lib.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.