CS 367 Project 2 – Fall 2024:  
MicroFP Software Floating-Point Library  
**Due: Friday, October 11th, 11:59pm**

|  |
| --- |
| **This is to be an individual effort. No partners. No Internet code/collaboration.  Protect your code from anyone accessing it. Do not post code on public repositories.**  **No late work allowed after 48 hours; each day late automatically uses one of your tokens.** |

**Core Topics: Floating Point, Bitwise Operators, C Programming Design (Helper Functions)**

# Project Overview

Micro-Ubiquitous Accounting Notary (**MUAN**) Programming Language is an expression-only, interpreted programming language designed to run on an **embedded system** without any hardware floating-point support. Many embedded systems typically do not have support for floats. You won’t have to worry about any of this, of course, but it means that you won’t be able to use any normal **float** or **double** variables.

**You may not use any C float or double types in your Solution**

**Your job is to implement MicroFP, which is a custom 9-bit floating-point library.**

You will be implementing functions to create the **MicroFP** 9-bit floating point library that MUAN will be using. You will be completing six functions in src/**microfp.c** for the API; however, it is **highly** recommended that you write many helper functions.

# The MUAN Programming Language

The MUAN programming language is already written; all you have to do is finish the API implementation for the six **MicroFP** functions in **microfp.c** that MUAN will use.

MUAN is a python-like interpretive programming language that will execute expressions using basic operators, constants, and variables. Like python, you can also write programs (eg. scripts) that you can run and see the output of.

Your MicroFP is a library that adds the ability to work with Floating Point values to the MUAN programming language. Since MUAN is designed for embedded hardware, there are no float or double data types available natively. Your library adds floating-point support to MUAN programs.

Please refer to the **MUAN Manual** document for information on how to program using MUAN.

This manual also has a large amount of sample inputs and output values.

# Specification for Project 2

We’ve already written the MUAN Programming language for you and provided you with the stubs (empty functions) for the six functions you will be writing inside of src/**microfp.c**

Complete this code, along with any number of helper functions that you would like to use, to implement these six functions. In this project, you will be working with our custom **microfp\_s** type variables. These custom types are **16-bit unsigned shorts** in memory. Within these 16-bits, you will be encoding our custom 9-bit floating point value.

Since a **microfp\_s** type is just a standard **unsigned short**, you can do operations on it just like you normally would with an unsigned short. (eg. shifting, masking, and other bitwise ops).

Ultimately, you will be getting the **S**, **exp**, and **frac** information and storing them within your **microfp\_s** value, just like we’ve been doing in class.

## MicroFP Representation (microfp\_s) Values

The **microfp\_s** values are encoded using the following format within an **unsigned 16-bit short**.

**Normalized and Special Encoding Format:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Must be Zeroes** | | | | | | | **S** | **exp** | | | **frac** | | | | |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **s** | **e** | **e** | **e** | **f** | **f** | **f** | **f** | **f** |
| *Bit* | *15* | *14* | *13* | *12* | *11* | *10* | *09* | *08* | *07* | *06* | *05* | *04* | *03* | *02* | *01* | *00* |

This is the 9-bit Representation for **microfp\_s** values:

1 bit for sign (s), 3 bits for exponent (exp) and 5 bits for fraction (frac)

MicroFP is a slightly simplified version of floating-point encoding. We can support **Negative, Normalized, Denormalized,** and **Special** (NaN or ) encodings as discussed in class.

MUAN does not do **Round to Nearest/Even**. All rounding will use **Round to Zero** **(Truncation)**. This means that all rounding will be by truncating any bits that do not fit within the frac field.

For example, if your mantissa was 1.01010101, your **frac** should have the following bits...

**01010101**

However, this would not fit in the 5-bit **frac** field of the MicroFP encoding. So, your rounding technique would be to only keep the first 5 bits and discard the remaining. This gives you...

**01010**

This would represent a rounded mantissa of 1.01010. This will **always** round towards zero.

If any function receives a **microfp\_s** with the upper bits set to anything except all 0s, you must safely ignore them. We will only test with values that have upper bits set to 0s, as is specified.

## Input Values to MicroFP

Since this is a program for a computer that does not have formal floating-point support, you may notice that there are no **float** or **double** types anywhere in the code for this program. In fact, **you will not be allowed to use any float or double types in your solution either.**

Because of this, the inputs to the functions you will be writing will be Integer types inside of a struct. These values will give you all the information that you need to use to implement your custom floating-point type (**microfp\_s**) values.

**Number Struct Definition (inc/common\_structs.h):**

**typedef struct number\_struct {**

**int is\_negative; // 1 if Negative, 0 if Positive**

**int is\_infinity; // 1 if Infinity, 0 Otherwise**

**int is\_nan; // 1 if NaN, 0 Otherwise**

**unsigned short whole; // 16-bit Whole Portion of Number in Binary**

**// (eg. For 3.25, whole = 0x3)**

**unsigned short fraction; // 16-bit Fraction Portion in Binary**

**// (eg. For 3.25, frac = 0x4000, which represents .0100000000000000)**

**Factoring\_s conversion; // Used Internally by MUAN, Ignore this!**

**} Number\_s;**

Two of your functions (**toMicroFP** and **toNumber**) will have this struct passed into them by reference. **Do NOT free those when you are done with them**. (They are controlled by MUAN).

**Example**: Let’s say someone enters the value **3.0625** in MUAN.

MUAN will pass a Number\_s struct to your **toMicroFP** function with the following members set:

* is\_negative = 0
* is\_infinity = 0
* is\_nan = 0
* whole = 3
* fraction = 0x1000 *(in GDB, you can print in hex with* ***print/x*** *instead of print)*
  + 3.0625 is **11.000100** in Binary.
  + The fraction field is a 16-bit unsigned int that has the Binary value of the fraction.
  + So, **.0001** is the same as **.0001 0000 0000 0000** with trailing zeros.
    - fraction gives you these 16 bits to the right of the binary point.
    - **0001 0000 0000 0000** (0x1000) is this fraction as a 16-bit unsigned short.
  + So, fraction gives you the fraction component AS a whole 16-bit value.

## Binary Scientific Notation

In the previous example, we have 3.0625 entered, which gave us the following in number:

whole = **0x3** [Representing 0000 0000 0000 00**11**]

fraction = **0x1000**  [Representing 000**1** 0000 0000 0000]

Together, these represent **11.0001**, whose value is 3.065, but we don't have a leading 1 to start working with this as Normalized. To fix this, do the same technique we show in class by shifting the bits of the Mantissa.

11.0001000… The Whole number part is > 1

Shift to the Right Once

0**1**.**1**000**1**00… The Whole number is in range. **whole= 0x1 frac = 0x8800**

How can we do that in code? Remember each part is represented by an unsigned int. Think about using shifting and bitwise operations to shift both parts (whole and fraction) left or right.

## Infinity Values and Overflows on Input into MUAN

The number struct will give you three ints to tell you if the user entered negative or special values. Note, this is only for special values typed by the user in MUAN. So, if a user enters the following:

**¯\\_(ツ)\_/¯ $ value = inf**

MUAN will call your toMicroFP function with the number struct that has **is\_infinity** = 1 set.

However, you can still end up with infinity through an overflow. So, with the following:

**¯\\_(ツ)\_/¯ $ value = 300**

MUAN will call your toMicroFP function with the number struct that has **is\_infinity** = 0, because the user did not enter **inf**, however, that number will still end up overflowing to infinity, so you will end up returning the infinity floating point encoding. (Note: 16.0 is an overflow in MicroFP)

## Entering Negative Values

Pay special attention to the **is\_negative** flag and make sure to set your S bit accordingly.

¯\\_(ツ)\_/¯ $ **value = -2.0**

MUAN will pass you a Number struct that has **is\_negative** = 1 because the user entered a negative value. Always check **is\_negative** in **toMicroFP**, so you know what to set your S it to in the value.

# Function Descriptions

src/**microfp.c** has been given to you as your starting file. This contains a stub for all six required functions. You are strongly encouraged to create helper functions, constants, etc. in your design. They too must all be kept within **microfp.c** as this is the only file you will be submitting.

If any function receives a **microfp\_s** with the upper bits set to anything except all 0s, you must safely ignore them. We will only test with values that have upper bits set to 0s, as is specified.

You are not allowed to use any **float** or **double** types in this project.

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

**MicroFP Function: microfp\_s toMicroFP(Number\_s \*number)**

When MUAN gets any number, (example: apple = -1.25) it will call this function.

toMicroFP will take a Number Struct (with its **whole** and **fraction** parts) and encode the data into our custom 9-bit representation and return that value.

**microfp\_s** is a typedef for a 16-bit **unsigned short** in C

Once you have encoded the value into this **microfp\_s**, you will be returning it.

**Example**: The val -**1.25** has Sign = 1, exp = 011, frac = 01000

The 9-bit encoding should be: 1 011 01000

The full **microfp\_s** (16-bit) value will be: 0000 0001 0110 1000

Hex: 0x168 (You can confirm this with ref\_all\_values)

**So, how do you get these fields from the value?**

You are not allowed to use **float** or **double** when working within your function, but you can do the same operations we did in class. (Hint: Think of Binary Scientific Notation)

Think carefully in your design about how you want to shift the whole and fraction integers, and how you will need to move values between them, to get the right format for Binary Scientific Notation.

Remember also that you will need to have something to track your E component and that for each shift you do, you will need to adjust that E.

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.

**Example for 3.25:**

* number->whole is 0x3
  + This is 0000 0000 0000 00**11** in Binary
* number->fraction is 0x4000 *(the bits on the right of the binary point)*
  + This is **01**00 0000 0000 0000 in Binary

Together, they represent **11.01** in Binary

To get this in Binary Scientific Notation for Normalized Encoding, you can shift!

* Be careful here because these are two 16-bit unsigned integers!

|  |  |  |
| --- | --- | --- |
| **whole** | **.** | **frac** |
| 0000 0000 0000 001**1** *Shift Right Once* |  | 0**1**00 0000 0000 0000 |
| 0000 0000 0000 0001 |  | 10**1**0 0000 0000 0000 |

Note that the Least Significant bit of **whole** became the Most Significant bit of **frac**

Logically, with the **whole** and **frac** parts, this represents the following binary value.

**0000 0000 0000 0001 . 1010 0000 0000 0000**

Which is just 1.101 in Binary. Also, since we shifted right once, we’ll add one to E.

This is now 1.101 \* 21, which is the proper value in Binary Scientific Notation.

**Rounding:**

* Follow the Rounding Rules (next section).

**Return your microfp\_s value or any microfp\_s value for NaN if number is NULL.**

**Special Rules:**

* **For Underflows** (eg. exp would be 0 with 1 M < 2 in the Normalized Range):
  + Encode the value using the **Denormalized** encoding.
* **Negatives**
  + Note that you can get a negative value on input, and it must be handled.
* **Special Values**
  + Note that you may get either Infinity (**is\_infinity**) or Nan (**is\_nan**) passed in.
    - These need to be handled according to the standard encoding.
  + If **both** flags are True, encode the number as NaN.
* **For Overflows** (eg. exp is too large for Normalized):
  + Encode the **microfp\_s** variable as the special value as appropriate.
  + Specials are as we covered with the exp set to all 1s and frac set accordingly.

**MicroFP Function: int toNumber(Number\_s \*num, microfp\_s value)**

When MUAN gets the **print** function, example: print(foo), it will call this function.

toNumber will convert our 9-bit representation (value) into the S, M, and E and the put the resulting components into a Number\_s struct that’s passed in by reference.

**Do NOT free or malloc on num, this is managed by MUAN and is already allocated for you!**

Extract the **S**, **exp**, and **frac** portions of the microfp\_stype and then convert them back into normal number form (just the whole and fraction values without any multipliers) these fields of the Number\_s struct. **num** is guaranteed that all values inside were initialized to 0s.

Fill in these Fields of **num**:

* **is\_negative**
  + Set this to 1 if the value is negative, or 0 if positive.
* **is\_infinity**
  + Set this to 1 is the value represents Infinity, or 0 otherwise.
* **is\_nan**
  + Set this to 1 if the value represents NaN, or 0 otherwise.
* **whole**
  + Set this to the whole number of your converted value. (Shift until E = 0 first!)
  + You can ignore this field if the number was infinity or NAN
* **fraction**
  + Set this to the binary representation of the bits that would be on the right of the binary point in the converted value. (Shift until E = 0 first!)
  + You can ignore this field if the number was infinity or NAN
* **conversion** –Do not Modify this Struct, it is used by MUAN. Leave As-Is

Example: If your **microfp\_s** value passed into **toNumber** was **0x1A2**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Must be Zeroes** | | | | | | | **S** | **exp** | | | **frac** | | | | |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | **0** | **1** | **0** | **0** | **0** | **1** | **0** |

This microfp\_s value encodes -11 \* **1.00010** \* 22. We need E = 0 to get the original number back.

Since E is 2, we need to shift the M left twice. (both whole and frac). Note that the two most-significant bits of **frac** become the two least-significant bits of **whole**.

|  |  |  |
| --- | --- | --- |
| **whole** | **.** | **frac** |
| 0000 0000 0000 000**1**  *Shift Left Twice* |  | 0001 0000 0000 0000 |
| 0000 0000 0000 0100 |  | 0100 0000 0000 0000 |

We end up with **100.01** in binary, which represents **4.25** in Decimal.

With the S bit, this would have the value of **-4.25** overall.

Your toNumber function must take this microfp\_s value and convert it back to the original number format with E = 0, so that you would set your Number\_s struct with these values:

* **is\_negative = 1**
* **is\_infinity = 0**
* **is\_nan = 0**
* **whole = 4** (which is already what you have after the shifting in whole)
  + **4** is 0000 0000 0000 0**1**00 in binary as an unsigned short.
* **fraction = 0x4000** (which is already what you have after the shifting in frac)
  + **0x4000** is 0100 0000 0000 0000 as an unsigned short.

You will not need to return num, since it’s passed in by reference, so when you set the values and return, MUAN already has access to it. **Do not free num**.

**Return 0 on success or -1 if the number argument was NULL.**

**MicroFP Function: microfp\_s mulMicroFP(microfp\_s val1, microfp\_s val2)**

This will multiply two microfp\_s values and then return the result as a formatted microfp\_s.

**Multiplication Examples in MUAN: foo \* bar**

**bar \* -3.5**

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **val1:** | S1, M1, E1 |  | **product**: | **S = S1 ^ S2** |
|  |  |  |  | **M = M1 \* M2** |
| **val2:** | S2, M2, E2 |  |  | **E = E1 + E2** |

**Hint: See Hint on Mathematical Operations in the Next Section**

Encode S, M, E of product back into **microfp\_s** format and return that resulting **microfp\_s** value.

**Special Rules for Multiplication:**

* **Rounding:**
  + Follow the Rounding Rules (next section).
* **For Underflows** (eg. exp would be 0 with 1 M < 2 in the Normalized Range):
  + Encode the value using the **Denormalized** encoding.
* **For Overflows** (eg. exp is too large for Normalized):
  + Encode the **microfp\_s** variable as the special value as needed.
* **Sign Considerations:**
  + Always follow the multiplication sign rules.
* **Arithmetic Special Cases:** See the next section on **Arithmetic Rules**

**You are not allowed to use double or float data types at any point in these functions**.

**MicroFP Function: microfp\_s addMicroFP(microfp\_s val1, microfp\_s val2)**

This adds two microfp\_s values and then returns the result as a properly formatted microfp\_s.

**Addition Examples in MUAN: foo + bar**

**2.5 + 3.5**

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:

value1: S1, M1, E1

value2: S2, M2, E2

result: S, M, E

**Align the Mantissas:**

You need both Es to be the same, so pick one of your values and adjust it.

Once both E1 and E2 are equal, you can now add your Ms to get the

resulting M of the sum. The E of the result will then be 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 S, M, and E of result, encode them into **microfp\_s** format and return the result.

**Hint: See Hint on Mathematical Operations in the Next Section**

Encode the S, M, E of sum back into **microfp\_s** format and return that resulting **microfp\_s** value.

**Special Rules:**

* **Rounding:**
  + Follow the Rounding Rules (next section).
* **For Underflows** (eg. exp would be 0 with 1 M < 2 in the Normalized Range):
  + Encode the value using the **Denormalized** encoding.
* **For Overflows** (eg. exp is too large for Normalized):
  + Encode the **microfp\_s** variable as the special value as needed.
* **Arithmetic Special Cases:** See the next section on **Arithmetic Rules**

**You are not allowed to use double or float data types at any point in these functions**.

**MicroFP Function: microfp\_s subMicroFP(microfp\_s val1, microfp\_s val2)**

This will perform the following subtraction: **val1 – val2**

**Subtraction Examples in MUAN: foo - bar**

**2.5 - 3.5**

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:

value1: S1, M1, E1

value2: S2, M2, E2

result: S, M, E

**Align the Mantissas:**

You need both Es to be the same, so pick one of your values and adjust it.

Once both E1 and E2 are equal, you can now subtract for M1 – M2 to get the

resulting M of the difference. The E of the result will then be equal to E1 and E2.

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 S, M, and E of result, encode them into **microfp\_s** format and return the result.

**Hint: See Hint on Mathematical Operations in the Next Section**

Encodes S, M, E of difference into **microfp\_s** format and returns that resulting **microfp\_s** value.

**Special Rules:**

* **Rounding:**
  + Follow the Rounding Rules (next section).
* **For Underflows** (eg. exp would be 0 with 1 M < 2 in the Normalized Range):
  + Encode the value using the **Denormalized** encoding.
* **For Overflows** (eg. exp is too large for Normalized):
  + Encode the **microfp\_s** variable as the special value as needed.
* **Arithmetic Special Cases:** See the next section on **Arithmetic Rules**

**You are not allowed to use double or float data types at any point in these functions**.

**MicroFP Function: microfp\_s negMicroFP(microfp\_s value)**

This will negate a microfp\_s value.

**Negation Examples in MUAN: -bar**

**-(x + 4)**

Change the sign of the input value and then output that result in **microfp\_s** format.

Return the **microfp\_s** result.

**You are not allowed to use double or float data types at any point in these functions**.

**The remaining MUAN Commands are already written for you.**

**MUAN Function: display(variable)**

**display Example in MUAN: display(bar)**

This command shows the binary representation of your **microfp\_s** values.

It’s great for debugging! It’s also written for you, so no work is needed.

**Example:**

¯\\_(ツ)\_/¯ $ bar = 1.25

¯\\_(ツ)\_/¯ $ display(bar)

MicroFP Value in Binary: 0 011 01000 (0x068)

¯\\_(ツ)\_/¯ $

This represents a **microfp\_s** value with these bits:

**0 011 01000**

S = 0, exp = 011, frac = 01000

This would be: -10 \* 1.01000 \* 20 in Binary, which is 1.25 in Decimal.

This shows the bits of the variable you set with your earlier functions, as well as a quick representation in hex (0x1a0) to help you compare with the **ref\_all\_values** program:

**E= 0 exp=(b011)**

**...**

**M = 1.250000 (b1.01000), val=1.250000000000000000 [0x068]**

**The ref\_all\_values Program is Described in the References Manual**

**MUAN Function: print(variable)**

**print Example in MUAN: print(bar)**

This command shows you the value of a variable or an expression.

This will call your **toNumber** function whenever it’s called.

**Example:**

¯\\_(ツ)\_/¯ $ bar = 1.25

¯\\_(ツ)\_/¯ $ print(bar)

bar = 1.25

¯\\_(ツ)\_/¯ $ print(3.0 - 1.5)

Value = 1.5

¯\\_(ツ)\_/¯ $

**MUAN Operator: quit** or **exit**

**exit Examples in MUAN: quit**

**exit**

You can use either **exit** or **quit** to quit the program. Easier than leaving Python!

**MUAN Operator: help**

**help Examples in MUAN: help**

This simply prints out some helpful command references with examples.

# **Rounding Rules**

Note that frac is small (5-bits) and not big enough to store all possible values, so we will need to do rounding using **round-to-zero (Truncation)**, which means for positive values, we always round the number down by truncating (removing) the bits that do not fit.

As an example, **microfp** can represent **13.0** and **13.25**, but nothing in between. When **13.20** is entered into MUAN, it will round down (to nearest) down to **13.0**. When **-13.20** is entered into MUAN, it will round to zero “up” to **-13.0**. Note that this is effectively the same as rounding the absolute value of any number down.

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

...

M = 1.625000 (b1.10100), val=13.000000000000000000 [0x0d4]

M = 1.656250 (b1.10101), val=13.250000000000000000 [0x0d5]

M = 1.687500 (b1.10110), val=13.500000000000000000 [0x0d6]

...

So, we can see that **13.0** and **13.25** are both representable, but **13.20** is not.

Note again that the **ref\_all\_values** program only shows positive values. Whether the number was positive or negative, it would not make any difference, since **13.20** would round down to the next valid value of **13.0** and **-13.20** would round to the next value closer to zero, which would be **-13.0**. In both cases, we ended up with the number **13.0**, just with a different sign at the end.

**See the Attached Manual for ref\_all\_values for more Information**

# Hint on Mathematical Operations

## One Valid Approach to Performing the Operations

Instead of trying to do an operation on M, when M is represented by **whole** and **fraction** parts, one trick you can use would be to shift each M to the left (updating the corresponding Es)

k-times, where k is the number of shifts needed so that both values have only **whole** components and no **fraction**. Once you have this, you can do the math using C directly as ints.

Let's look at an example using Subtraction.

**val1 is 1.01100 \* 2-1** (.6875 in decimal) **and val2 is 1.00010 \* 2-1** (.53125 in decimal)

If you shift each Mantissa left 4 times, then you end up with this:

**val1 is 10110.0 \* 2-5 and val2 is 10001.0 \* 2-5**

Now the whole number for val1 is **10110**, which as an int is 22

Now the whole number for val2 is **10001**, which as an int is 17

Since we still have the same Es, you can subtract the wholes in C as **int**s and get 22 - 17 = **5**

So, your answer has a whole value of 5 as a temporary **int**, which is stored as 101.0 in Binary.

This would then be **101.0 \* 2-5** as the result of the subtraction with whole = **5** and exp = **-5**

Now you can put it back in normalized format by shifting right to get a whole number of 1 again…

**result is 1.01000 \* 2-3** (.15625 in decimal, which is the proper answer to 0.6875 - 0.53125)

And now you can continue with your encoding.

This is not required to do, however, this hint makes it far easier for you to do the mathematical operations using C in int format by first getting both values as whole numbers.

# **Special Arithmetic Rules**

Use these rules for special cases when doing arithmetic:

(X represents any Real number that is not 0, NaN, or

1. **Addition/Subtraction Special Rules***(Arguments can be in any order of mathematical equivalence)*
2. **Multiplication Special Rules** *(Args can be in any order of mathematical equivalence)*

**Remember the sign rules for any Multiplication!**

(Multiplying by , - 0 or -0 is handled using normal multiplication sign rules)

**Note: The sign has no effect if the result is NaN. NaN may be represented with either sign.**

# Project Constraints

**You may Not #include <math.h> or use any math.h functions, including pow()**

**You may Not use any double or float types anywhere in your code.**

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

* + These will be implemented using the standard special number pattern in your **microfp\_s** floating-point representation. (Remember )
  + There is only one NaN, regardless of sign, it’s not a number.
    - Any pattern which matches a NaN is considered equivalent.
    - **Your microfp\_s inputs should be able to recognize any value bit representation of NaN.**
    - Your functionsmay use any valid NaN representation.

**Rounding is by Round to Zero**

* + Numbers round towards Zero by truncating any bits that do not fit in frac.

**Negative Numbers must be handled.**

* + All values (including ) will be handled properly with negatives.
  + All functions should support -0 values being passed in as arguments.

# 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 key folders and key files:

* **Makefile** – Run **make** to build the assignment (and **make** **clean** to clean up).
* **src/**
  + **microfp.c** – **This is the only file you will be modifying (and submitting).**   
    Feel free to define more functions if you like but put all of your code in this file!
  + **test\_microfp.c** - Framework for a unit tester you can use.
* **inc/**
  + **microfp\_precision.h** **– Do Not Modify.**
    - This does have some nice constants that you may wish to use if you like.
  + **common\_structs.h – Do Not Modify.** This has the struct definitions.
  + **MUAN\_settings.h:** You may modify this. This defines what the prompt looks like and whether or not you want to see the colors. Feel free to change as you like.
* **scripts/**
  + **sample.muan** – This is the provided sample MUAN script. You can write your own for testing!
* **ref\_all\_values** – This prints out all legal values in our representation.

# Note on Types:

Remember that in C, all integer types are just collections of bits. So, an **unsigned short** is just 16-bits. We can interpret this as an Integer, of course, but we can also just use it as a container and work with those individual bits.

The **microfp\_s** type is an unsigned short internally, but you can think of it as a 16-bit container. Remember if you do any shifting, C will perform right-shifts on the signed/unsigned type rules.

# Testing your Code

## Testing with MUAN

First, you can run your code in **MUAN**, which will let you enter expressions and see the outputs in the full program that your MicroFP is designed for. The best way to test in here is to enter a value using variable assignment and then to use the **display** function on it. The **display** function is pre-written and will tell you exactly what the value was converted to by your **toMicroFP** function, making it easy to compare with the values from **ref\_all\_values**, without needing to have **toNumber** written at all.

Once you have that working, you can then pass values into the **print** function to print them out to the screen. All values passed into **print** will have been converted by your **toMicroFP** function earlier, but now **print** will call your **toNumber** function to print them out as full values again. So, you can use **print** to test your **toNumber**, once you know your **toMicroFP** function is working.

After that, you can test the remaining functions using the above.

## Testing with a Unit Tester

We also included a simple unit testing program with the handout. This is the same style of tester that you had available to you for Project 1. You can build this testing program with:

**make tester**

You can then run the program with:

**./tester**

The file src/**test\_microfp.c** currently has a main function that calls a single testing function as an example of what you could put in here. That testing function just calls **negMicroFP** with a known encoded value and then checks out what it returns. This is just a demo of using the tester.

You can modify this tester a will. It is **much** easier to run **GDB** using the tester than MUAN.

## Testing Tips for Mathematical Operations

So, how do we know what the results should be?

The trick here is to do the math with a calculator, however, at every step, you need to look up each value in the expression using **ref\_all\_values** first.

So, let's say we want to multiply: 9.1 \* 0.24

Neither of these are in ref\_all\_values! So, we need to look up the next lower values.

…

**M = 1.125000 (b1.00100), val=9.000000000000000000 [0x0c4]**

**M = 1.156250 (b1.00101), val=9.250000000000000000 [0x0c5]**

…

…

**M = 0.937500 (b0.11110), val=0.234375000000000000 [0x01e]**

**M = 0.968750 (b0.11111), val=0.242187500000000000 [0x01f]**

…

So, we know that if we entered 9.1 and 0.24, we will actually be rounding to **9.0** and **0.234375**

Now we can multiply these two with a calculator and we get **2.10938**. We know this is the proper, mathematical result we're expecting, however, our miniFP will be rounding this too! So, now we have to look 2.10938 up in ref\_all\_values to see what we should expect.

…

**M = 1.031250 (b1.00001), val=2.062500000000000000 [0x081]**

**M = 1.062500 (b1.00010), val=2.125000000000000000 [0x082]**

…

So, the rounded value that we can expect would be 2.0625. This is the result we should get.

Let's now try this inside of **MUAN** to test our code.

¯\\_(ツ)\_/¯ $ print(9.1 \* 0.24)

**Value = 2.0625**

That is how we can check to see what we're expecting!

**We can also implement this in our test\_microfp.c file as well using the hex values:**

**M = 1.125000 (b1.00100), val=9.000000000000000000 [0x0c4]**

**M = 0.937500 (b0.11110), val=0.234375000000000000 [0x01e]**

**M = 1.031250 (b1.00001), val=2.062500000000000000 [0x081]**

So, if we pass in **0x0C4** as the first val and **0x01E** as the second val to mulMicroFP(), we should get back **0x081** as the result. We can check this in our tester, like so...

**microfp\_s val = mulMicroFP(0x0c4, 0x01e);**

**if(val == 0x081) {**

**printf("Great success!\n");**

**}**

**else {**

**printf("Expected 0x081, got 0x%x\n", val);**

**}**

# Submitting & Grading

Submit this assignment electronically on Canvas. Note that the only file that gets submitted is **microfp.c.** Make sure to put your name and G# as a commented line in the beginning of your source file in the comment that is already provided for you at the top of the 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 Canvas! 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, so please double-check that you have submitted on time.   
  
Questions about the specification should be directed to the CS 367 Piazza forum.

**A full Rubric is available on Blackboard for the Project as well with more details.**

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 **microfp.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.

## Valgrind Notes:

This program leaks more than if the Titanic crashed into the Poseidon in an Aquaman movie.

**No Valgrind will be Run and No Leak Checks will be Made.**

# Changelog

**v1.0: Sep 19: Release** (Build: **2910c72a** in develop)

**v1.01: Sep 22:** Fixed a pair of typos on Page 7 in the Example

* The value passed in to toNumber should have been **0x1A2** since this was Negative.
* The Sign bit in the depicted encoding should have been a **1** since this was Negative.
* Both have been corrected as this was -4.25, so the S bit should have been a 1