# ECE 464 / 564 Project : Fall 2023: Quantum Computing Emulator

Change Log
- 8/21/23 original spec
- 10/20/2023 corrected a mistake concerning Q and M
- 10/29/2023 minor corrections
- 10/31/2023 reverted 464 sram layout and updated signal definitions

References:
- Quantum Computing for Computer Scientists by Noson S. Yanofsky (Author), Mirco A. Mannucci (Author). https://catalog.lib.ncsu.edu/catalog/NCSU5575047) *available as an ebook.
- Qiskit's learning quantum computing learning tutorial: https://qiskit.org/learn

This project is to be conducted individually. You can collaborate on the paper version of the design, including discussion of ideas, design approach, etc. However, you are forbidden to share code or to reuse code of others. We will be running code comparison tools on your submitted code.
The usage of AI tools such as GPT is allowed, but should be declared (delineated by comments) and the queries referenced and presented in your report. Comments should indicate the start and end of any GPT derived code. The student is responsible for all that is submitted. No points will be awarded if the AI generated content was faulty.

**EOL students are expected to do the ECE 464 project.**

**ECE 564**: Your task is to implement a nearly fully functional quantum emulator with complex values (it will be missing the measurement gates). The number of qubits would range from 1 to 4. On campus ECE 564 students can limit themselves to the ECE 464 project if they choose but at the cost of a 25% penalty.

**ECE 464 and EOL (students in section 601) students**: Your task is to implement a quantum computing emulator that does not contain imaginary numbers. The number of qubits would be fixed at 2.

This document will start with a brief background on quantum computing. It is not strictly necessary to understand this background but be advised that interviewers might ask. More will be presented in class.

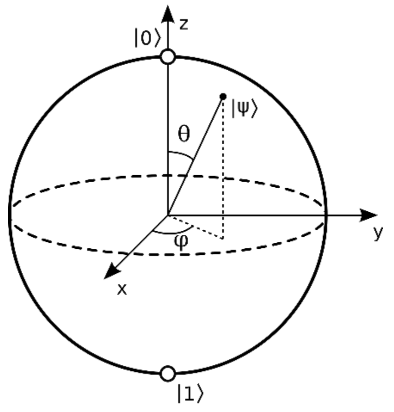
# Quantum Computing (QC)

Quantum computing is a type of computing that uses qubits, which can represent multiple states at once due to superposition. This allows quantum computers to perform certain calculations faster than classical computers. Qubits can also be entangled, meaning their states are instantaneously connected, enabling complex interactions. The power of quantum computers is that n qubits can represent 2**n states. Quantum computers excel at specific problems like factorization, simulation of quantum systems, and optimization. However, the technology is still in its early stages and faces challenges before becoming widely practical.

| Digital | Quantum State Vector |
|---|---|
| 0 | $\begin{bmatrix}1\\0\end{bmatrix}$ |
| 1 | $\begin{bmatrix}0\\1\end{bmatrix}$ |
| NA | e.g. $\begin{bmatrix}\frac{1}{\sqrt{2}}\\ i\,\frac{-1}{\sqrt{2}}\end{bmatrix}$ |

**The Bloch Sphere**

A very common way of graphically representing a qubit's state is to use the bloch sphere just like the one listed below:



This captures the probabilistic nature of a single qubit. When measured, the qubit will collapse to a classical bit – 0 or 1, depending with a probability captured in this concept.

The axes of this sphere are as follows:

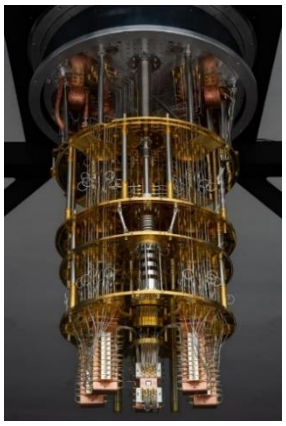
| Vector | Matrix correspondence | Measurement outcome |
|---|---|---|
| $\vert 0\rangle$ vector = North pole | $\begin{bmatrix}1\\0\end{bmatrix}$ | Gives us a “0” 100% of the time |
| $\vert 1\rangle$ vector = South pole | $\begin{bmatrix}0\\1\end{bmatrix}$ | Gives us a “1” 100% of the time |
| $\vert +\rangle$ vector (along the X axis) | $\begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\end{bmatrix}$ | Gives us a “0” 50% of the time and “1” 50% of the time |
| $\vert -\rangle$ vector (along the X axis) | $\begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{-1}{\sqrt{2}}\end{bmatrix}$ | Gives us a “0” 50% of the time and “1” 50% of the time |
| $\vert +\mathrm{i}\rangle$ vector (along the Y axis) | $\begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{i}{\sqrt{2}}\end{bmatrix}$ | Gives us a “0” 50% of the time and “1” 50% of the time |
| $\vert -\mathrm{i}\rangle$ vector (along the Y axis) | $\begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{-i}{\sqrt{2}}\end{bmatrix}$ | Gives us a “0” 50% of the time and “1” 50% of the time |

You may notice that all of the side vectors give us the same measurement outcome, so what’s the difference? It is simply the **interference** that a lot of quantum computing scientists use to make a lot of their algorithms work. It is mainly used to influence other qubits and open the qubit into getting influenced by other qubit’s interference… It is one of the cornerstones of what makes quantum computing so useful!

Nowadays, the two main methods of performing unsimulated or unemulated quantum computing is done through:

1- Superconducting universial gate quantum computers (e.g. IBM and Google)

2- Trapped ion (e.g.IonQ and Honeywell)



Rapid advances are being made in quantum computing. However they are still very noisy. An emulator allows the results to be produced by an ideal quantum computer to be predicted. You are going to build part of an emulator. So why build actual quantum computers when emulators are available? The reason is scalability, e.g. a 32-bit quantum computer requires 2**32 data entries. E.g. A 21 qubit emulator requires over 7 TB of memory. .

# The Quantum Circuit

A small quantum circuit would look something like this:

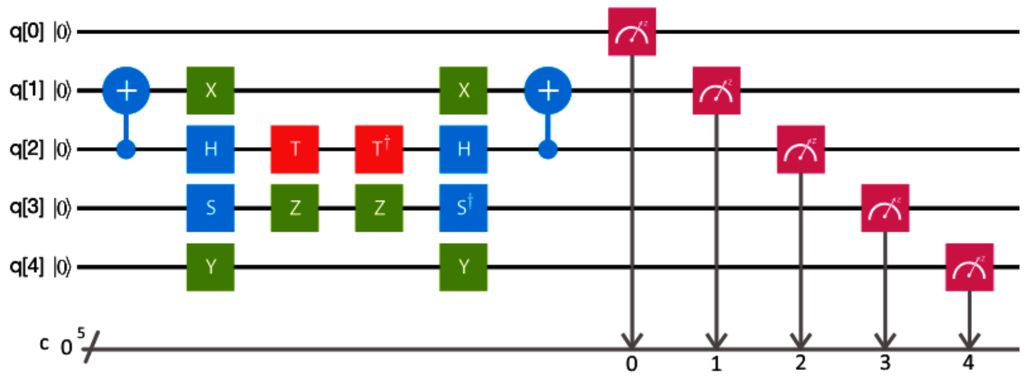


*Figure 1: A superconducting quantum computer, it's just a sample, so don't try to understand it*

It looks like a musical sheet rather than a electrical circuit, but that's for a reason. It's not really a circuit per say, it's a schedule of different operations being performed to different qubits. The last operation – that looks like a gauge – is the measurement operation that collapses the qubits down to classical bits.  Let's scale it down a notch:

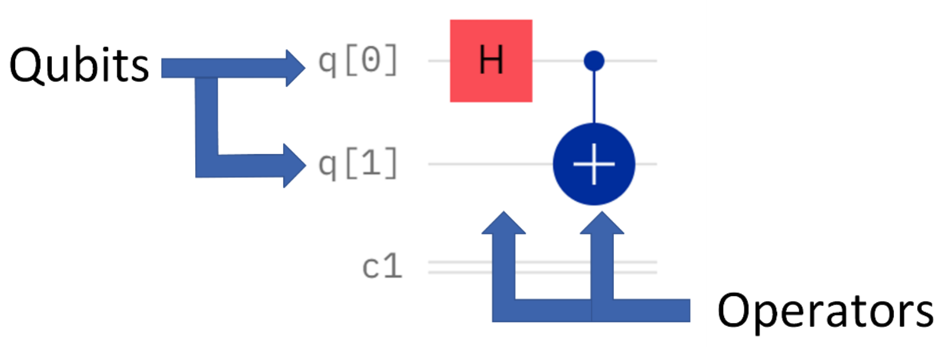


*Figure 2: This is the circuit that will correspond to the matrices in the following pages*

Now let's analyze the quantum circuit above. There are two qubits in this system and that means that the **initial state vector** of it will be $2^n = 2^2 = 4$ rows.

### Initial state vector

In the majority of cases the initial state of the state vector would be that all qubits are initialized to zero giving us this matrix on the left which corresponds to if the circuit is measured at the beginning:

$$\begin{bmatrix}1\\0\\0\\0\end{bmatrix} \rightarrow \begin{bmatrix}00\\01\\10\\11\end{bmatrix}$$

*The initial state vector matrix and its output representation*

This just means that when the quantum circuit is measured, it will give us a result of "00" 100% of the time. Think of it as the matrix that describes the probabilistic state of the system. The number is **NOT** the probability, it's the number that you square to get the probability. You will note that most if not all of the matrices here are unitary; meaning that if you square each number in the matrix and add them up, they'll add up to 1. This matrix/vector changes whenever an **operator** acts on any qubit in the system, except for the identity operator matrix(which passes the values as they are).
**Tldr:** think of it as the quantum circuit input.

**Operator Matrices:**

$$H = \begin{bmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}}\end{bmatrix} \quad \text{CNOT} = \begin{bmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{bmatrix}$$

Let's analyze the small quantum circuit from the previous page (Figure 2). The H matrix is called the Hadamard operator and it is commonly used to initiate the qubit into superposition and it is a single qubit operator, that's why it is a 2x2 matrix. The second matrix is called a Conditional Not and it is a two-qubit operator, that's why it is a 4x4 matrix.
A CNOT gate senses what's in the control qubit q[0] and applies a NOT gate to the qubit q[1].
Assuming that the initial state is **00** the chain of matrix multiplications that our previously mentioned quantum circuit will got through is:

$$\begin{bmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{bmatrix} \times \begin{bmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0\\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0\\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}}\end{bmatrix} \times \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\0\\0\\\frac{1}{\sqrt{2}}\end{bmatrix}$$

The result is that there's a 50% chance that we'll get 00 and 50% that we'll get 11. The value of the state vector is **not** the probability. You get the probability by absolute valuing + squaring it:

$$\left|\frac{1}{\sqrt{2}}\right|^2 = 0.5$$

*Note: You may observe that the CNOT gate in other forms based on which qubit is the control and which is the target.*

If you understood all of that, then great! You know just enough of quantum computing to do the project! For the sake of clearer understanding, we'll give the matrices that we just explained variable name:

$$\ldots \times \begin{bmatrix} w1 & w2 & w3 & w4 \\ w5 & w6 & w7 & w8 \\ w9 & w10 & \cdot\cdot & \cdot\cdot \\ \cdot\cdot & \cdot\cdot & \cdot\cdot & w16 \end{bmatrix} \times \begin{bmatrix} v1 & v2 & v3 & v4 \\ v5 & v6 & v7 & v8 \\ v9 & v10 & \cdot\cdot & \cdot\cdot \\ \cdot\cdot & \cdot\cdot & \cdot\cdot & v16 \end{bmatrix} \times \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \delta \end{bmatrix}$$

**For 564 students, you will have to multiply two complex floating point numbers.:**

1- Real multiplied with real
2- Real multiplied with imaginary (twice and added)
3- Imaginary with imaginary

1 and 3 would have to be summed together, since $i^2$ equals -1. As a reminder,

$$(A1 + i\ A2)*(B1 + i\ B2) = (A1*B1 – A2*B2) + i\ (A1 * B2 + A2 * B1)$$

**464 and EOL students**: will have just a regular matrix multiplication problem with floating point reals and no imaginary part.

**Measurement:**

A very vital stage of quantum computing is measurement (seen in the very right end of Figure 1). This stage is left out for the sake of brevity and simpler grading. What you are producing in this project is the state vector and not the measurement.

To produce a measurement, you'd have to have a random number generator that would be called 1000+ times for each none zero vector. In quantum computing terms, each "call" is called a "shot". A histogram could be produced in the end to generate a visualization of this circuit's measurements.

**<u>This part is not required nor part of this project</u>**

# Project Specifications

- Inputs:
    - Number of qubits(matrix dimensions) + number of operator matrices
    - Initial state vector matrix
    - Operator matrices
- Output:
    - Final state vector matrix

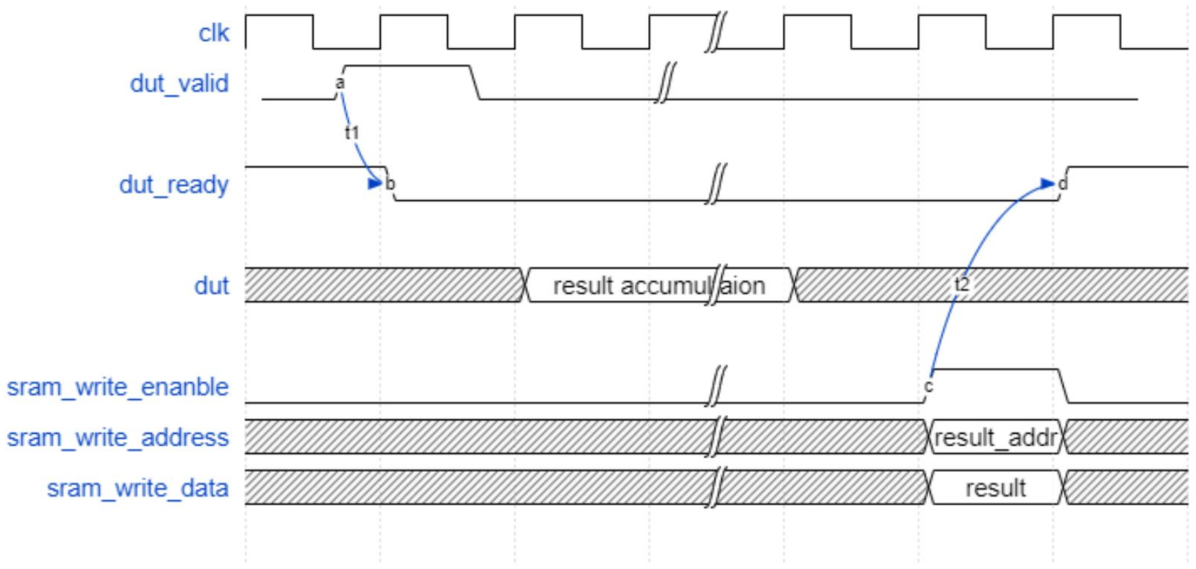
**Components:**

- **DUT** (device under test): Specified under the project folder as ***dut.v*** this is where you will write ALL of your code. Please don't touch any other file unless you've got explicit approval from your instructor.
- **SRAMS:** The following are all SRAMs and all have the same interaction procedures, but differing purposes. Please don't touch any of the SRAM source code unless you've got explicit approval from your instructor. There are 4 SRAMs that have already been instantiated for you in the testbench.sv file: **q_state_input, q_state_output, scratchpad, and q_gates**. You will interact with them using their specified wires in the testbench file too.
- **Designware Floating point units:** The documentation for them is included on Moodle. It is highly recommended that you only use the MAC units to minimize rounding errors, so please base your design on it, have a backup, then try optimizing with the adder and multiplier at your own risk.

**DUT Signals**

The IO are as follows:

- Clk: Clock from the test fixture. Make sure this corresponds to the name of the clock in the synthesis script.
- reset_n: (Test fixture -> DUT) Reset from the test fixture. This will go active low at the start of the simulation run.
- dut_ready: (DUT -> Test fixture) Signals that the DUT is ready to do its computation. DUT should hold the dut_ready low when a dut_valid high is detected until it has fully populated the result value back in the *q_state_output* SRAM.
- dut_valid: (Test fixture -> DUT) Signals that a valid input can be computed from the SRAM.


clk
dut_valid
dut_ready
dut
sram_write_enanble
sram_write_address
sram_write_data
a
b
c
d
t1
t2
result accumul/aion
result_addr
result

**SRAM Contents:**

All SRAMs are holders of double precision floating point numbers(**64 bits each**). Each transfer AND address holder has a length of 128 bits. The color scheme corresponds to the matrix in the "Operator Matrices" section.

- **q_state_input:** Contains the number of operator matrices(M) and the dimensions of the q_gates $2^Q$ . The input matrix should have the dimensions of $2^Q$x1.

| **564 Project** | | |
|---|---|---|
| Address | 127:64 | 63:0 |
| 00 | Q: ***number of Qubits*** | M***: Number of operator(yellow) matrices*** |
| 01 | a(real) | a(imaginary) |
| 02 | b(real) | b(imaginary) |
| .. | … | … |
| .. | d(real) | d(imaginary) |

| **464 Project** | | |
|---|---|---|
| Address | 127:64 | 63:0 |
| 00 | Q: ***number of Qubits*** | M***: Number of operator(yellow) matrices*** |
| 01 | a(real) | Zero fill |
| 02 | b(real) | Zero fill |
| .. | … | Zero fill |
| .. | d(real) | Zero fill |

- **q_state_output:** Contains your solution and your solution would be evaluated according to the data in this matrix. It should have the dimensions of $2^Q$x1.

| **564 Project** | | |
|---|---|---|
| Address | 127:64 | 63:0 |
| 00 | x(real) | x(imaginary) |
| 01 | y(real) | y(imaginary) |
| .. | … | … |
| .. | $\delta$ (real) | $\delta$ (imaginary) |

| **464 Project** | | |
|---|---|---|
| Address | 127:64 | 63:0 |
| 00 | x(real) | Zero fill |
| 01 | y(real) | Zero fill |
| .. | … | Zero fill |
| .. | $\delta$ (real) | Zero fill |

- **q_gates:** Contains the values of the **operator matrix,** and contains $(2^Q)^2$*M elements

| **564 Project** | | |
|---|---|---|
| Address | 127:64 | 63:0 |
| 00 | v1(real) | v1(imaginary) |
| 01 | v2(real) | v2(imaginary) |
| .. | … | … |
| .. | $w$16 (real) | $w$16 (imaginary) |
| .. | .. | .. |

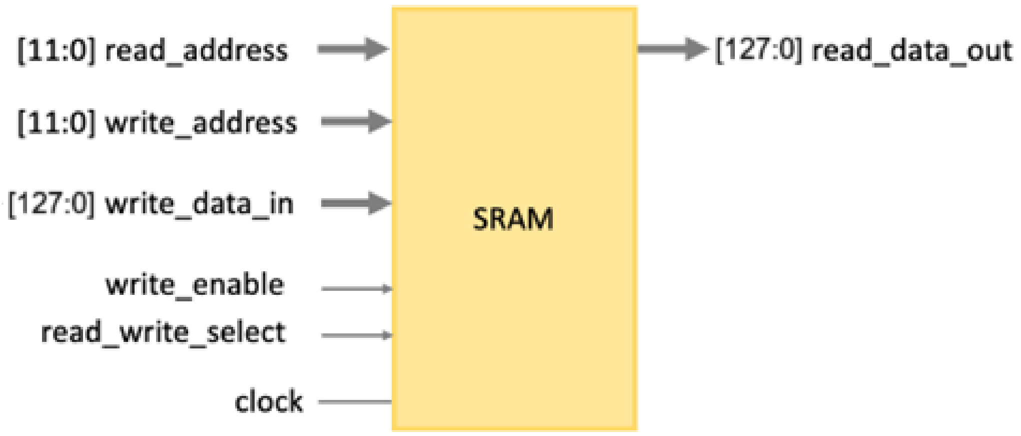
| **464 Project** | | |
|---|---|---|
| Address | 127:64 | 63:0 |
| 00 | v1(real) | Zero fill |
| 01 | v2(real) | Zero fill |
| .. | … | Zero fill |
| .. | $w$16 (real) | Zero fill |
| .. | .. | Zero fill |

- **scratchpad:** This SRAM is there purely for your own usage, you can use it to offload some of the memory. This would result in lower area, hence giving you a better grade. However it should be noted that you can totally disregard it if you're struggling to complete the project.

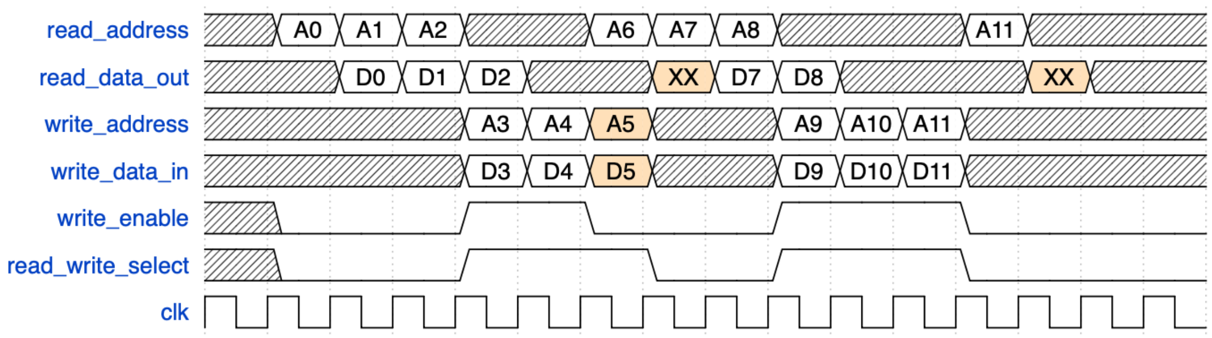
**Hierarchy**

I suggest that you organize your design as one module. There's no need for multiple module files.

**SRAM interface**



The SRAM is word addressable and has a one cycle delay between address and data. When writing to the SRAM, you would have to set the "write_enable" to high. The SRAM will write the data in the next cycle.

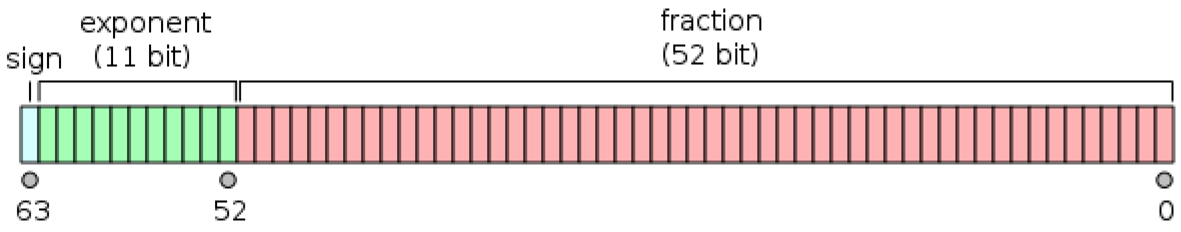


As shown in the example above, since "write_enable" is set to low when A5 and D5 is on the write bus, D5 will not be written to the SRAM. Also, because "read_write_select" is set to high, the read request for A6 will not be valid. (IN THE PROJECT WE HAVE REMOVED read_write_select).

Note that the SRAM cannot handle consecutive read after write (RAW) to the same address (shown as A11 and D11 in the timing diagram). You would have to either manage the timing of your access, or write the data forwarding mechanism yourself. As long as the read and write address are different, the request can be pipelined.

**Double Precision Floating Point**

You don't have to worry a lot about this format since you will have your own floating point unit (FPU), but to give you a quick background the format in which double precision floating point is represented is as follows:



The order of multiplies and adds matter. What matters however is that the operations are performed **left to right** for all addition procedures. Because of how the fraction part works in double precision floating points, there will always be an error factor: "machine epsilon". In order to match up with the golden results, we ask that you perform additions in left to right order.

For more info on floating point: [https://en.wikipedia.org/wiki/Double-precision_floating-point_format](https://en.wikipedia.org/wiki/Double-precision_floating-point_format)