CSE 210

Computer Architecture Sessional

Assignment-2

Floating Point Adder(FPA)

Group Information: **Section:** C1 **Group:** 07

Members of the Group:

- 2105128 - Nakib Arman Arzon
- 2105138 - Dip Saha
- 2105142 -Md Mehedi Hasan Mim

November 16, 2024

1 Introduction

A floating point number is a number that has a decimal point and thus can store a fractional value. These numbers are stored differently than integers because to store a floating point number there are mainly 2 challenges. First of all, there is a challenge of storing the integer part and also storing the fractional part, hence more memory. Secondly, there are infinitely many floating point numbers even in a finite range, for example $[0, 1]$. So the floating point representation has to be clever enough to allow floating point numbers in a large range, as well as support good enough precision.

The floating point number representation uses the scientific notation of numbers. It has 3 parts - sign, significand and exponent. There are many different standards that decide how many bits each of these segments should store.
The *IEEE 754* standard specifies that there should be 1 sign bit, 8 exponent bits and 23 significand bits. The exponent is not directly stored, rather it is *biased* first by adding a bias to the actual exponent and than store that in memory. The significance of this process is to make the exponent unsigned, thus allowing easier circuitry to deal with the comparison of exponents of 2 numbers. The bias is chosen to be $2^{n-1} - 1$ where n is the number of bits in the exponent field.
In the scientific notation there is always a 1 in front of the radix point unless the number is 0. So storing that 1 is redundant, hence the significand bits always store the fractional part.

So the overall representation looks like:

$$(-1)^{\text{sign}} \times (1 + \text{significand}) \times 2^{\text{exponent}-\text{bias}}$$

Here the *exponent* portion represents the exponent stored in memory.

This representation has the power of encapsulating large range due to the use of exponents. On the other hand it can store numbers in great precision as effective exponents can be negative, and thus indicate really small numbers. But it should be kept in mind that there can be scenarios, specially for larger exponents, where the exact floating point number can not be stored. In those cases the memory stores the nearest floating point number.

2 Problem Specification

The assignment asked to create a floating point adder circuit that takes 2 floating point numbers and calculates their sum, and also notifies if there was any overflow or underflow. The representation was defined in the following manner-

Sign	**Exponent**	**Significand**
1 bit	9 bits	22 bits

Table 1: Problem specification

3 Flowchart

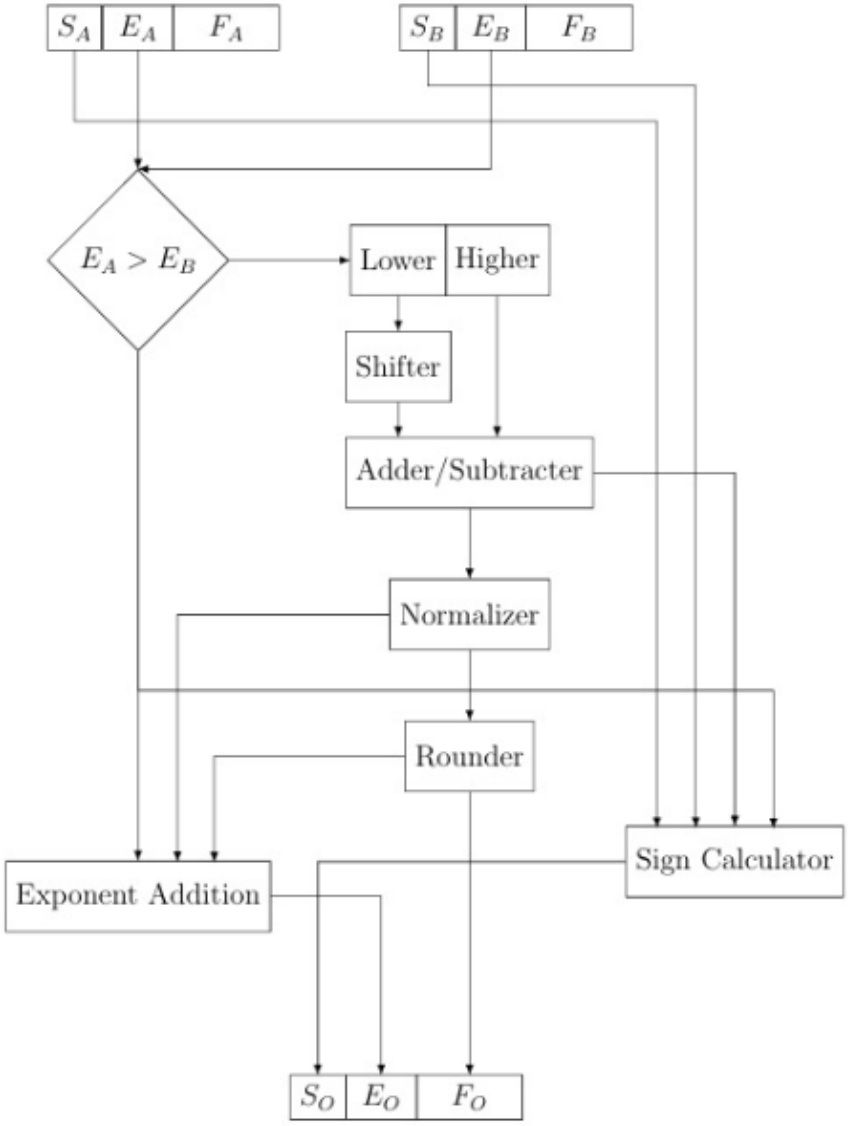

Figure 1: Flowchart of addition/subtraction algorithm

4 Description of Modules

Each high level module and how they work is listed below

4.1 Input

This module(Input.circ) helps to process the input for further operations in the floating point adder. It contains:

- An Input Splitter circuit which splits the 32 bit floating point number to a sign bit, 9 bit exponent and 22 bit significand. the 22 bit significand is again extended to 32 bits.

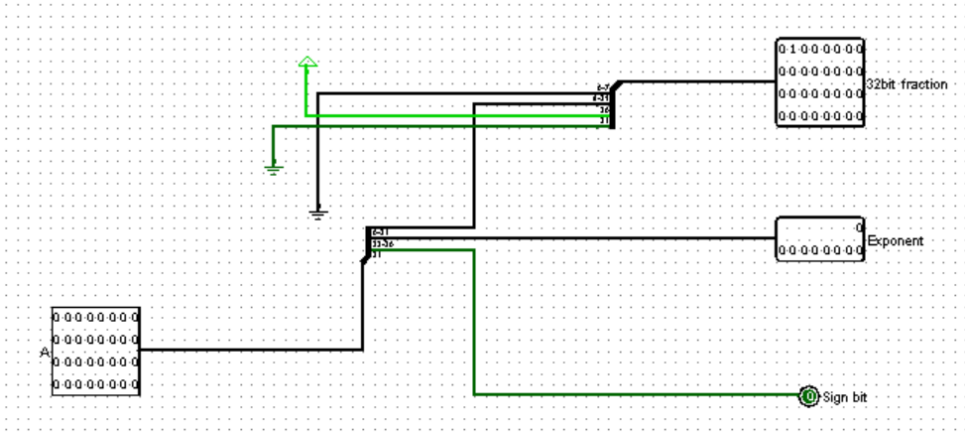

Figure 2: Input Spiltter

- Exponent Comparator that calculates the difference between the exponenets of two inputs

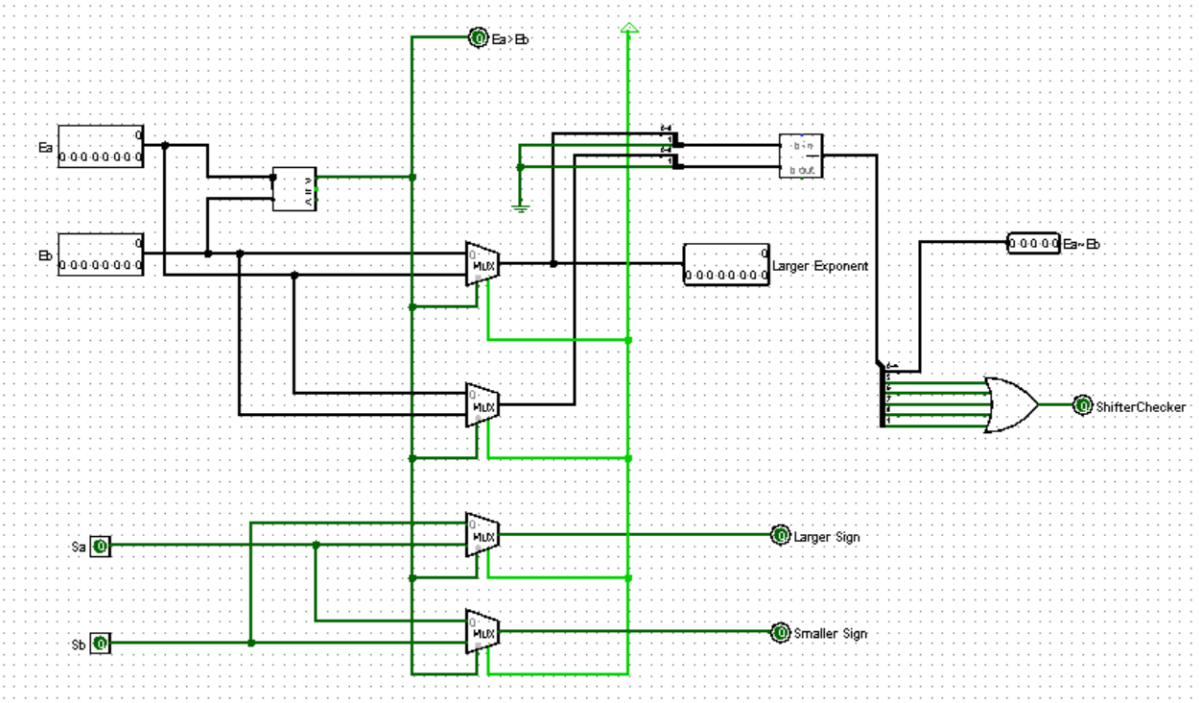

Figure 3: Exponent Comparator

- An Input Processor circuit that determines which input is greater than the other, greater exponent, exponent difference etc for control decisions

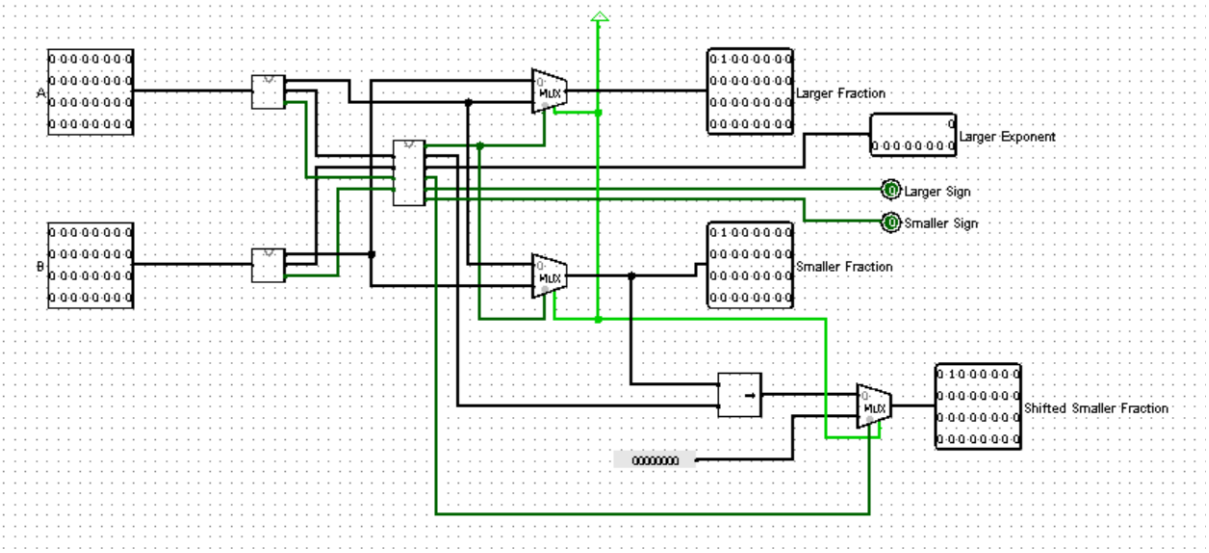

Figure 4: Input Processor

4.2 Adder Library

An adder circuit (Adder.circ) has been included in the Fraction Adder library which performs the most crucial part of the FPA, which is to add the significands of the two given input floating point numbers.

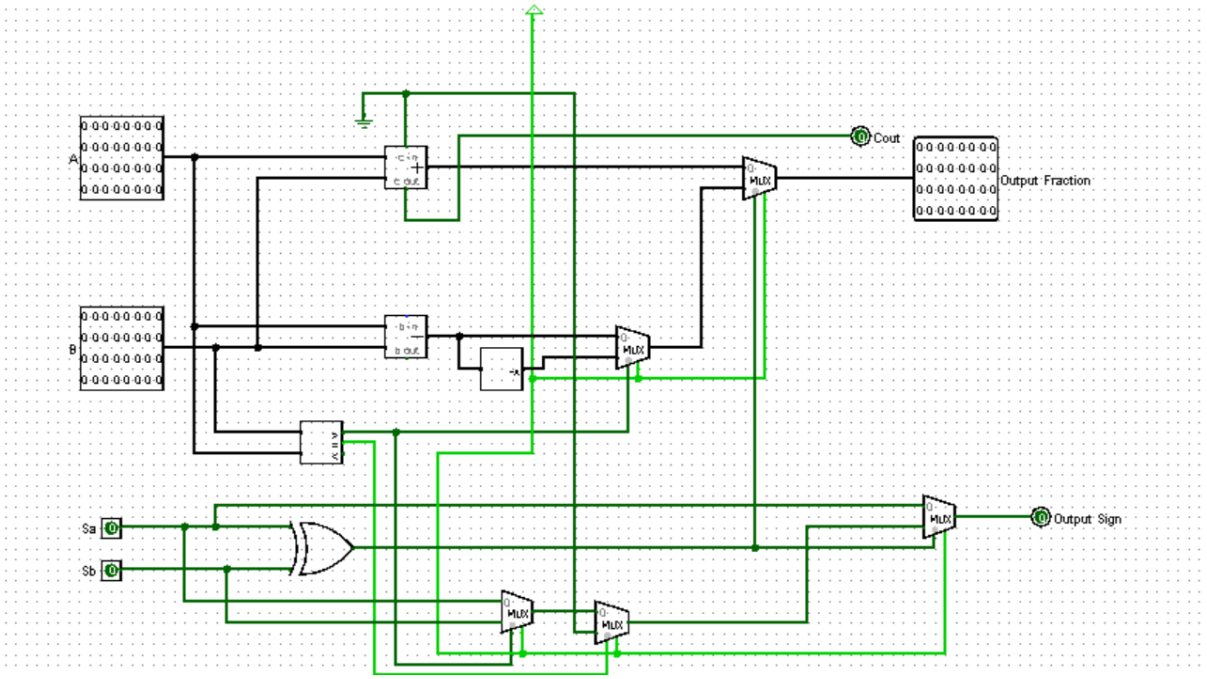

Figure 5: 32 Bit Adder Circuit

4.3 Normalizer Library

The circuit normalization included in this library (Normalizer.circ) normalizes the output by applying the necessary number of bit shifts with the aid of an encoder library. Additionally, it produces an 9-bit value that is added to the exponent to finish the normalizing process.

- **Normalizer:** Normalizes the input by applying necessary shifts.

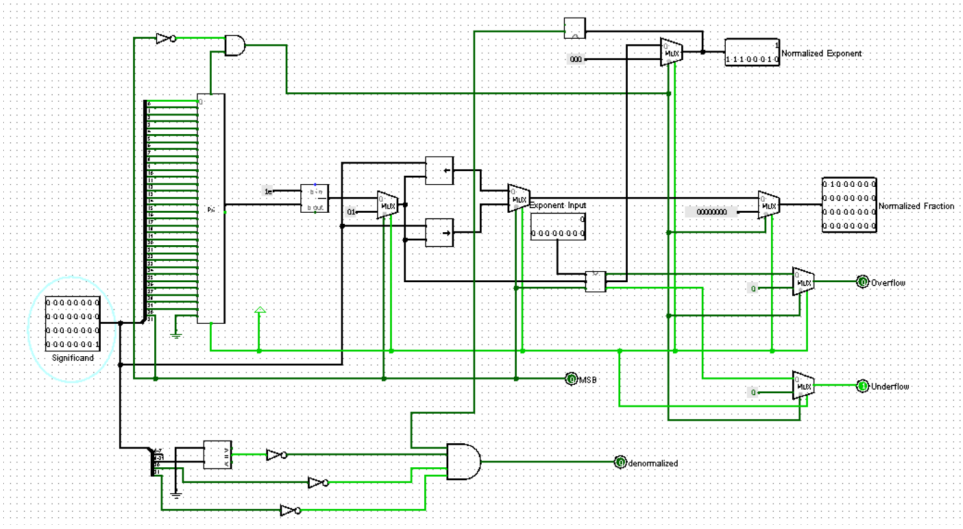

Figure 6: Normalizer

- **AdderSubtractor:** Adds or Substracts to the exponent according to the bits shifted.

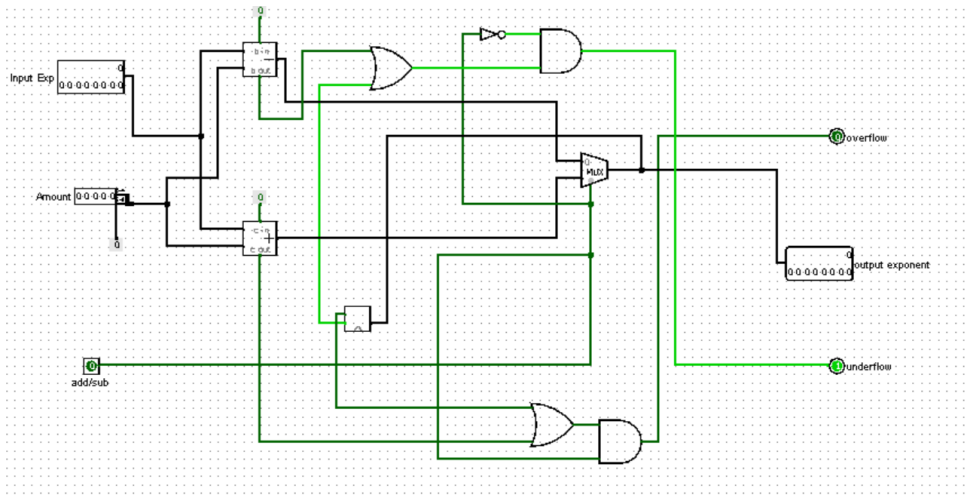

Figure 7: Adder Subtractor Circuit

- **ZeroOneChecker:** Checks whether all the bits in exponent aro 0 or 1

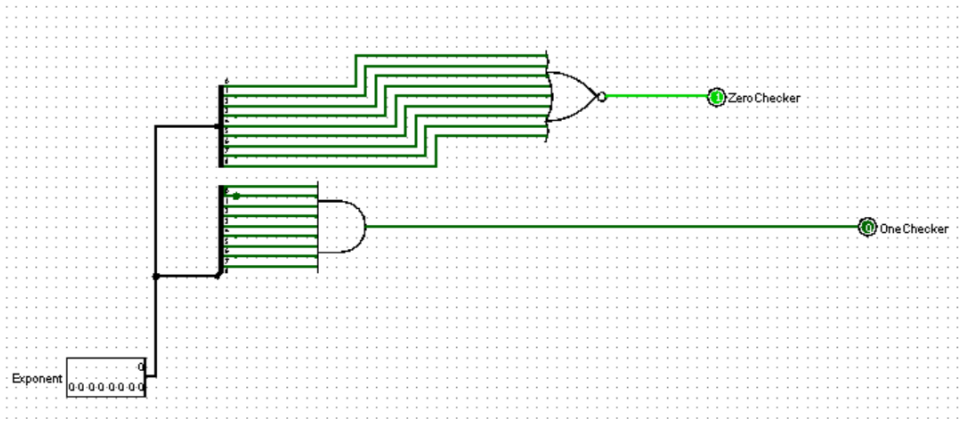

Figure 8: Zero One checker

4.4 Rounder

The rounder is responsible for rounding the result and storing only the limited number of bits (in this assignment, 20 bits) in the significand segment of the result.
Initially before exponent equalization, three 0 bits are added to the right of the number. But after the whole addition process, we can not store those bits as our memory is limited. But those bits help us store the closest result by approximating it. The 3 bits are called the guard bit, the round bit and the sticky bit (G, R, S).

By the *IEEE 754* standard, the approximation is achieved in the following manner-

G	R	S	Process	Comment
0	0	0	Truncate	Add 0 to the actual significand
0	0	1		
0	1	0		
0	1	1		
1	0	0	Round to even	Add 1 if the last bit of actual significand is 1
1	0	1	Round up	Add 1 to the actual significand
1	1	0		
1	1	1		

Table 2: Rounder Truth Table

From this table this is apparent that we only add 1 when the guard bit (G) is 1 **AND** either of the round bit (R) **OR** the sticky bit (S) **OR** the last bit of the significand is 1. So we add 1 when $G(R + S + d_0) = 1$. In other words, we add $G(R + S + d_0)$ to our 22 bit significand.

The number can become denormalized after rounding up or rounding to even. In that case we have to shift the number 1 bit to the right and hence increment the exponent. Because the exponent is being incremented, there can be scenarios where the exponent is too large, thus an overflow. So the rounder also reports such overflows.

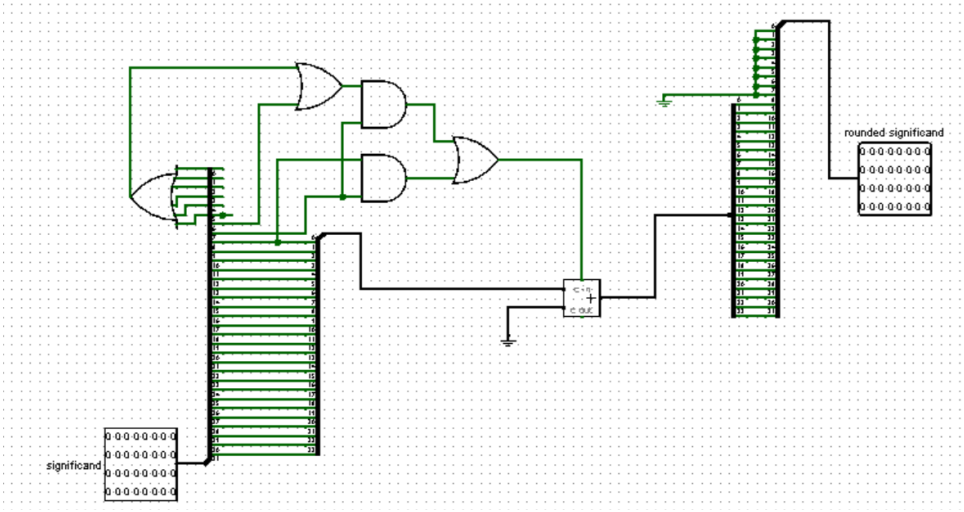

Figure 9: Rounder Circuit

4.5 Rounded Normalizer

The circuit Rounded Normalizer included inside the library (RoundedNormalizer.circ) normalizes the rounded output.

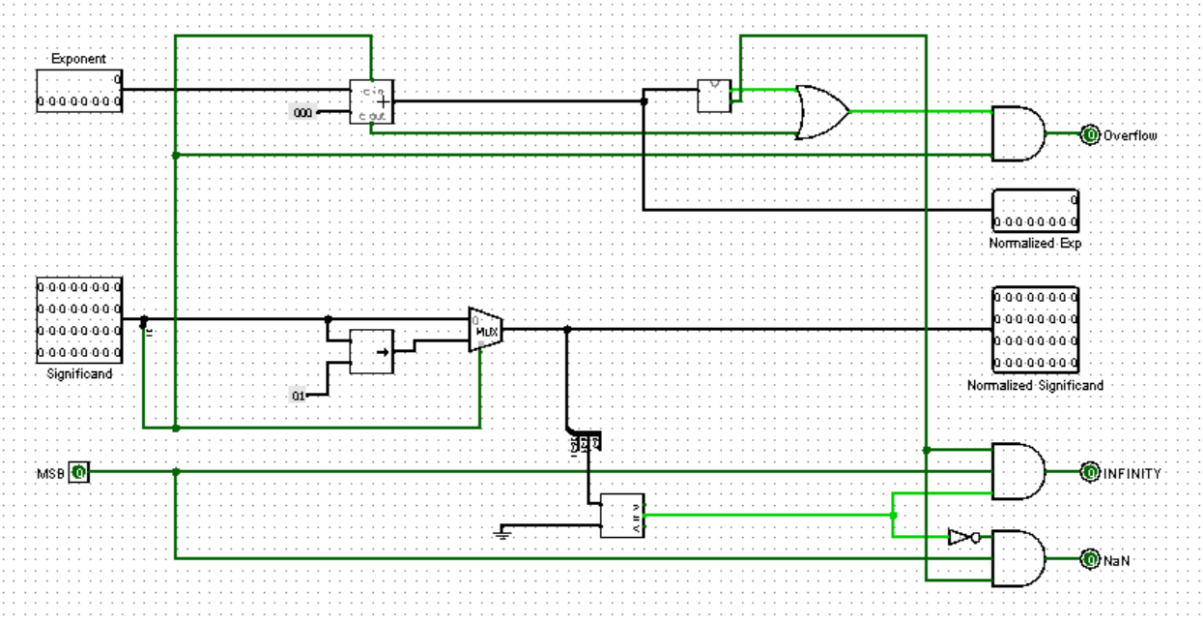

Figure 10: Rounded Normalizer

4.6 Floating Point Adder

The last module, called FPA.circ, uses the libraries and other modules to fully create a floating point adder. It has the real floating point adder, or circuit FPA.

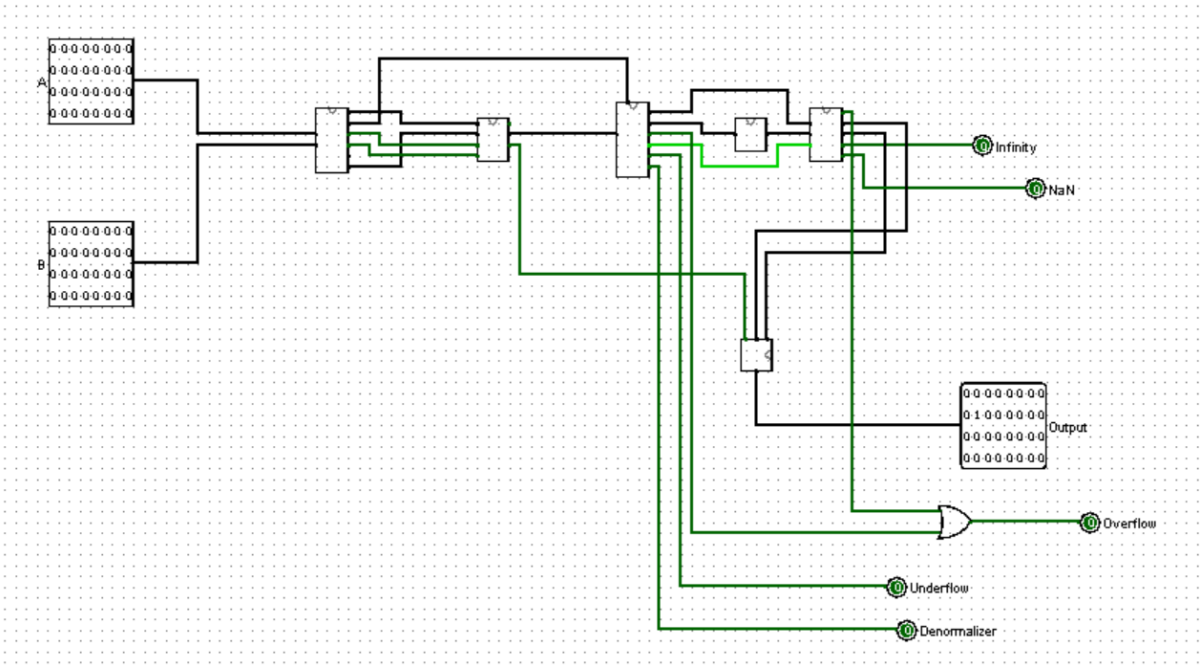

Figure 11: The Main FPA Circuit

The output merger circuit that merges the sign, exponent and significand of the result is also included in this module.

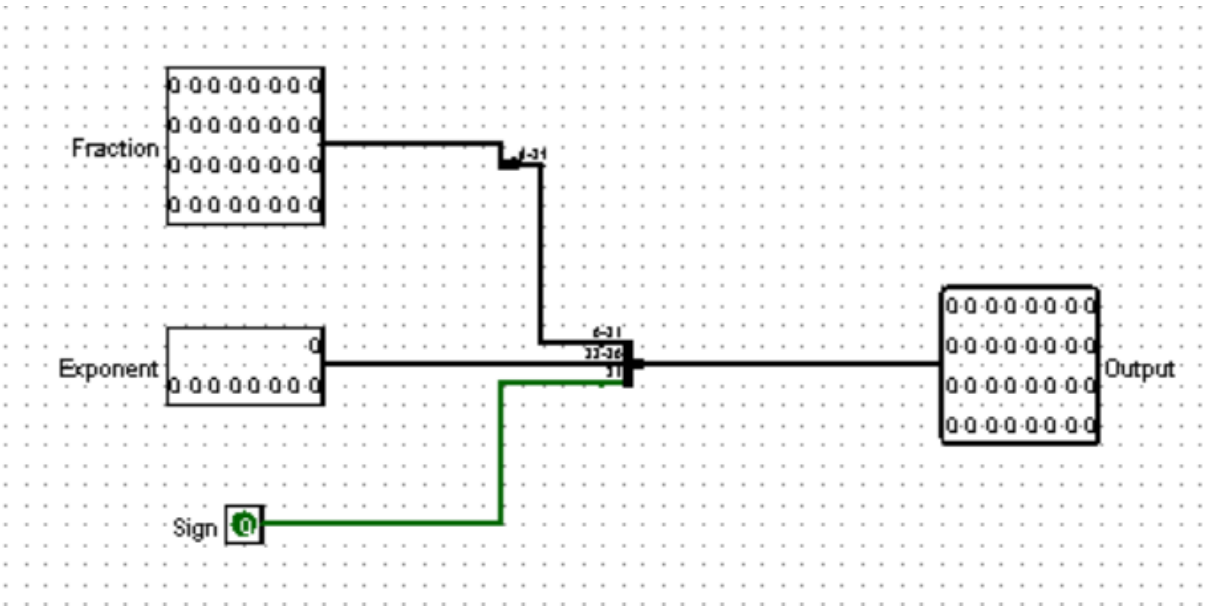

Figure 12: Output Processor of the Result

5 High Level Block Diagram

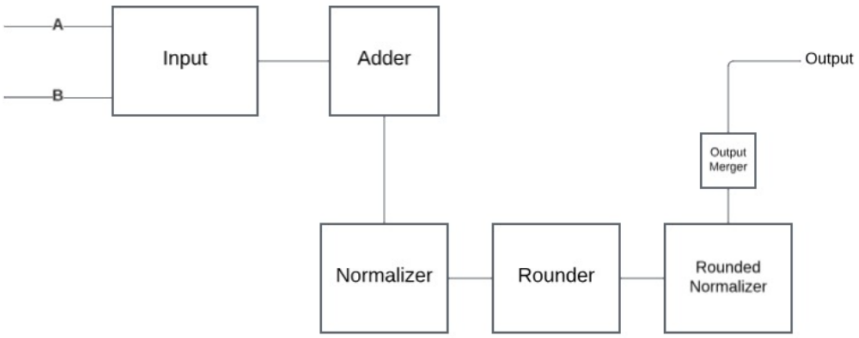

Figure 13: Block Diagram

6 ICs Used with Count as a Chart

IC	Quantity
IC 7483	8
IC 74157	74
IC 7432	4
IC 7486	1
IC 7404	1
IC 74148	4
IC 7408	6
IC 7402	3
Shifter	4
Comparator	4
Total	109

Table 3: ICs Used with Quantity

7 Simulator Software with Version

Logisim 2.7.1 has been used for this assignment.

8 Discussion

All of us have put a great deal of effort in order to complete this assignment. We tried our best to reduce the number of ICs and also tried to make a simplified design overall so that the real life implementation becomes easier. The use of ALU was preferred over creating own adders or comparators as that has the potential of saving a lot of ICs.

The assignment asked to only handle the floating point domain, but we also handled the case when the number is zero. According to the *IEEE 754* standard, the floating point number's exponent can not be all 0s nor all 1s. These 2 cases are reserved. They indicate different meanings. For instance, both the significand and exponent being 0 indicate 0.0. Although the assignment did not ask to handle the reserved cases, we decided to tackle the zero's case as well. For example with the assignment requirement implemented without handling these special cases, the adder will fail to provide $a - a = 0$ or $a + 0 = 0 + a = a$. But we took great care to implement this special case of the input or output being 0.

The adder of our implementation is not perfect. Since the rounding buffer contains only 1 sticky bit. By adding more sticky bits it is possible to have a greater buffer of result, and thus better precision. Having said that, we have found the precision level of our implementation is good enough for most of the input cases.

Overall the entire design process along with the implementation proved to be a very challenging but interesting task. We have learned a lot from this assignment.

9 Contribution of Each Member

2105128 - Nakib Arman Arzon

- Developed Input Library
- Designed the whole adder and assembled components
- Designed the INF, NAN and Denormalized flag

2105138 - Dip Saha

- Developed rounder
- Tested the adder thoroughly
- Generated the Report

2105142 -Md Mehedi Hasan Mim

- Developed the normalizer circuit
- Developed overflow and underflow detector and relevant circuits
- Developed Rounded Normalizer circuit