CSE 306
Computer Architecture Sessional

Assignment-2
Floating Point Adder

Section - A1
Group - 01

Group Members:

1. 2005001 - Anik Saha
2. 2005012 - Abrar Jahin Sarker
3. 2005013 - Al Muhit Muhtadi
4. 2005017 - Abdullah Muhammed Amimul Ehsan
5. 2005023 - Jaber Ahmed Deeder

1 Introduction

Floating-point representation serves as a technique for expressing real numbers in a manner that accommodates an extensive range of values, encompassing both minuscule and colossal numbers.

1.1 Floating Point Representation

According to IEEE 754 standard, a floating-point number is delineated by a trio of elements: a sign bit, an exponent, and a fraction (alternatively termed mantissa or significand). The general structure adheres to the formula:

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

In our assignment,we have exponents of 11 bits and mantissa of 20 bits.

Bit Number	Portion
31	Sign (S)
30-19	Exponent
18-0	Fraction/Mantissa

Table 1: Bit Ranges of Floating-Point Number

Each component holds specific significance:

- Sign bit: Denoting the sign of the number, with 0 for positive and 1 for negative.
- Exponent: Denoting the exponent of 2 by which the fraction undergoes multiplication.
- Fraction: Representing the actual fractional segment of the number, often normalized to commence with a leading 1.

The normalized significand, residing in the range $1.0 \leq |\text{significand}| < 2.0$, consistently incorporates a leading pre-binary-point 1 bit. This obviates the need for explicit representation, commonly denoted as the hidden bit. The normalized significand is essentially synonymous with the fraction, featuring the restored "1." at its forefront.

The bias, a constant, is incorporated to facilitate the representation of the exponent using a signed integer, thereby accommodating both positive and negative exponents.Here the bias is 1023.

1.2 Range

- Extremely small numbers, in proximity to zero, manifest with a diminished exponent and a fraction featuring leading zeros.
- Extremely large numbers are portrayed with an elevated exponent.

In a floating-point adder with a 20-bit mantissa and an 11-bit exponent, the largest and smallest representable positive normalized numbers are determined by the range of exponents and the precision of the mantissa.

For the largest positive number, the exponent is set to its maximum value of 1023, and the mantissa is the maximum value for a 20-bit representation. The binary representation is $1.11111111111111111111 \times 2^{1023}$. In approximate decimal form, this represents a number of approximately 8.988×10^{307}.

For the smallest positive normalized number, the exponent is set to its minimum value of -1022, and the mantissa is the minimum value (just the implicit leading bit). The binary representation is $1.00000000000000000000 \times 2^{-1022}$, approximately equal to 2.225×10^{-308} in decimal.

These values encapsulate the range of positive normalized numbers that can be precisely represented by your floating-point adder with a 20-bit mantissa and an 11-bit exponent. It's crucial to note that these approximations are subject to the inherent limitations of representing real numbers in a finite binary format.

1.3 EEE 754 encoding of floating point numbers

Exponent	Fraction	Object Represented
0	0	0
0	Non Zero	± Denormalized Number
1-2046	Anything	± Floating Point Number
2047	0	± Infinity
2047	Nonzero	NaN(Not a Number)

Table 2: EEE 754 Encoding

1.4 Denormalized Number

The denormalized numbers in IEEE 754 floating-point representation are a special category of values that are smaller than normal numbers. They are used to allow for gradual underflow, where numbers get extremely close to zero with diminishing precision. Denormalized numbers have an exponent of all zeros and a hidden bit set to zero. The formula for denormalized numbers is:

$$\text{Denormalized Number} = (-1)^{\text{sign}} \times (0 + \text{Fraction}) \times 2^{-\text{exponent}_{\text{min}}}$$

Where: - sign is the sign bit (either 0 or 1), - Fraction is the fractional part of the number (including the hidden bit, which is 0 for denormalized numbers), - $\text{exponent}_{\text{min}}$ is the minimum representable exponent value.

Denormalized numbers have a smaller exponent range compared to normalized numbers, and they allow for a smooth transition to zero, ensuring that precision degrades gradually as the numbers get smaller.

For example, in single-precision format, the smallest denormalized number is represented as:

$$0.\underbrace{00000000000000000001}_{\text{Fraction}} \times 2^{-1022}$$

Which is equivalent to 1.0×2^{-1042}. This is considerably smaller than the smallest positive normalized number, which is $1.00000000000000000000 \times 2^{-1022}$. Denormalized numbers play a crucial role in maintaining precision for very small values close to zero in floating-point arithmetic.

1.5 Floating Point Number Addition

Performing floating-point addition involves several steps:

Example: Consider two normalized numbers in IEEE 754 single-precision format:

$$A = (-1)^0 \times 1.011 \times 2^3 \quad \text{and} \quad B = (-1)^1 \times 1.101 \times 2^2$$

Step 1: Aligning Exponents Let us Identify the number with the smaller exponent and adjust its mantissa and exponent to match the larger one. Since B has a smaller exponent, we need to align it with A. We shift the mantissa of B to the right and decrease its exponent by 1:

$$A = (-1)^0 \times 1.011 \times 2^3 \quad \text{and} \quad B = (-1)^1 \times 0.1101 \times 2^3$$

We need exponent comparator and Right shifter for this.

Step 2: Adding Mantissas Let us add the mantissas:

$$\text{Sum of Mantissas} = 1.011 + (-1)^1 \times 0.1101 = 0.1001$$

This step is performed using ALU(Arithmetic Logic Unit)

Step 3: Normalizing Result If the output is not normalised,we have to Normalize the result by adjusting the exponent and shifting the mantissa to have a leading 1:

$$\text{Normalized Result} = 1.001 \times 2^4$$

This normalization is performed either by shifting right and incrementing the exponent or shifting left and decrementing the exponent.

Step 4: Checking for Overflow or Underflow We have to check for overflow or underflow.If either occurs, it indicates an exception. In this case, there's no overflow or underflow.

Step 5: Rounding Result We have to the result based on the precision required.A rounder circuit can be designed Guard and round digits and sticky bits.A rounder module can be implemented for this purpose.

Step 6: Normalizing Rounded Result If the output is not in normalised form,we have to Normalize the result by adjusting the exponent and shifting the mantissa to have a leading 1

Step 7: Handle Special Cases In the realm of floating-point representation, some noteworthy cases deserve attention: denormalized numbers, infinity, and NaN. There are no special cases in this example.

Moreover, The sign of the result is determined based on the signs of the original numbers. So, after aligning exponents, adding mantissas, normalizing, and finalizing the result, the sum of A and B is approximately:

$$\text{Result} = 1.001 \times 2^4$$

This process ensures correct addition of floating-point numbers while considering the alignment of exponents, mantissa addition, normalization, rounding, and handling special cases.

2 Problem Specification

In this assignment, we were required to design a floating point adder circuit which takes two floating point numbers as inputs and provides their sum, another floating point number as output. Each floating point number will be 32 bits long with following representation:

Sign	Exponent	Fraction
1 bit	11 bits	20 bits

Table 3: Problem Specification

3 Flowchart of the addition/subtraction algorithm

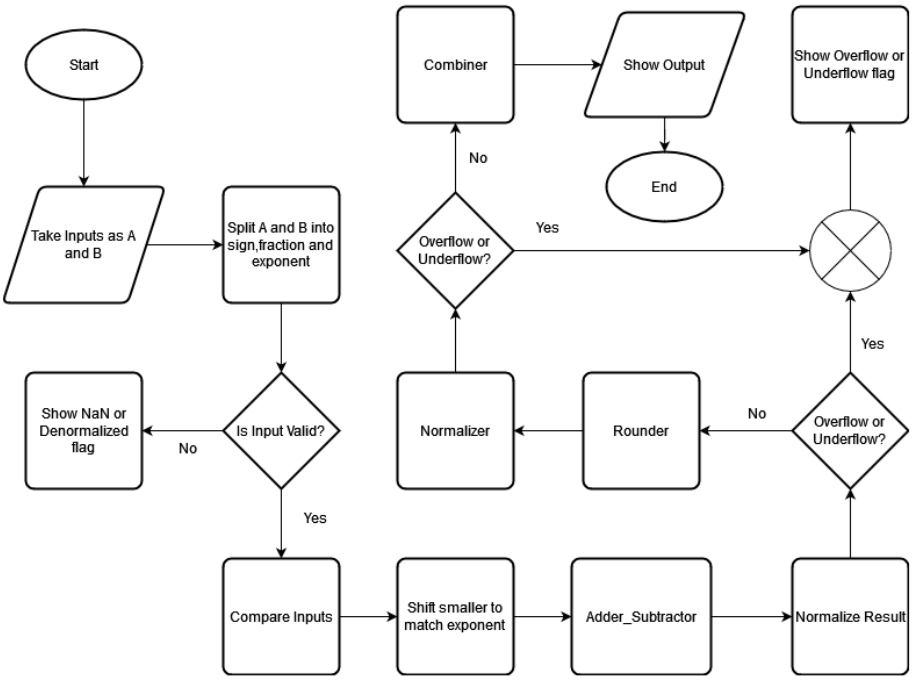


Figure 1: Flowchart of the addition/subtraction algorithm

4 High-level block diagram of the architecture

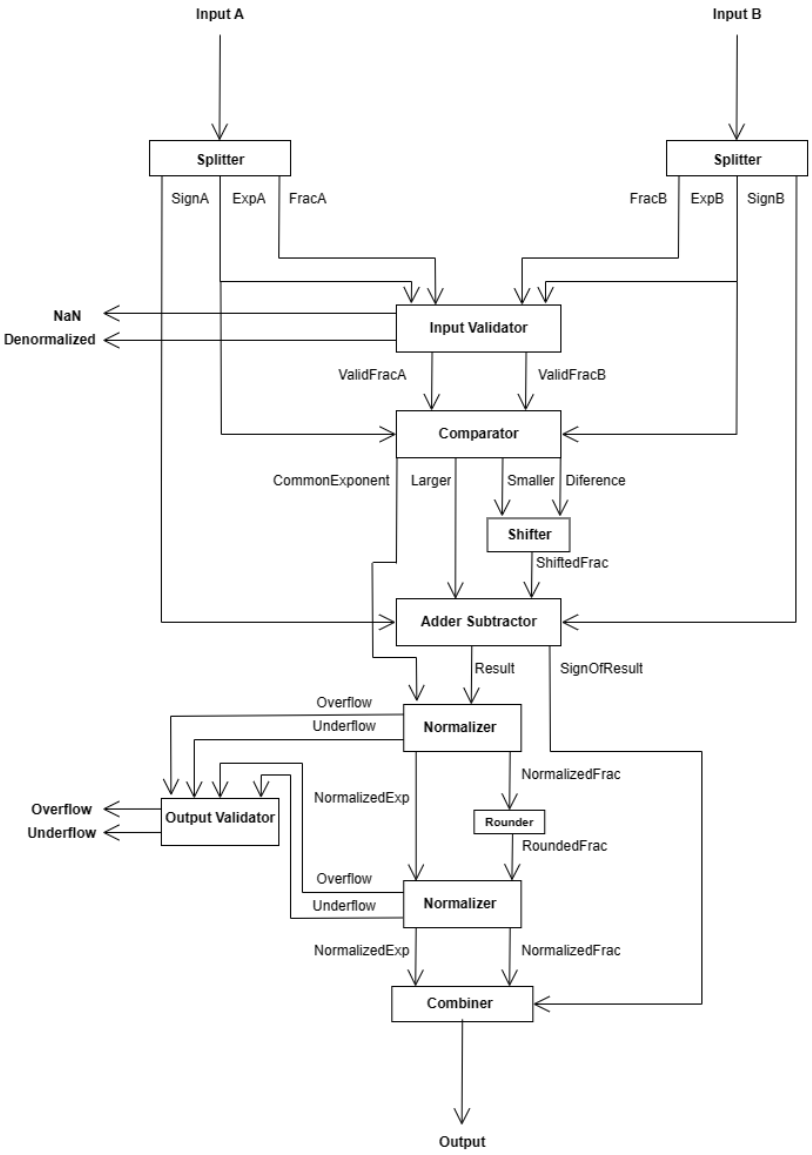


Figure 2: Flowchart of the addition/subtraction algorithm

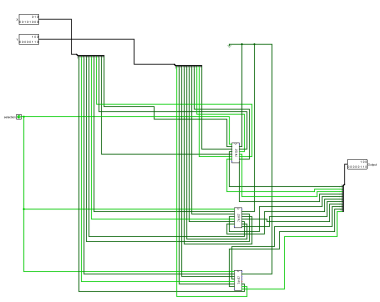
5 Detailed Circuit Diagram of the Important Blocks

Some libraries and circuits were implemented to enhance and simplify the final circuit design.Those are :

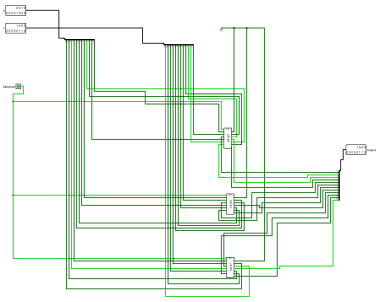
5.1 Multiplexer Circuits

The modular circuits in this library are as follows

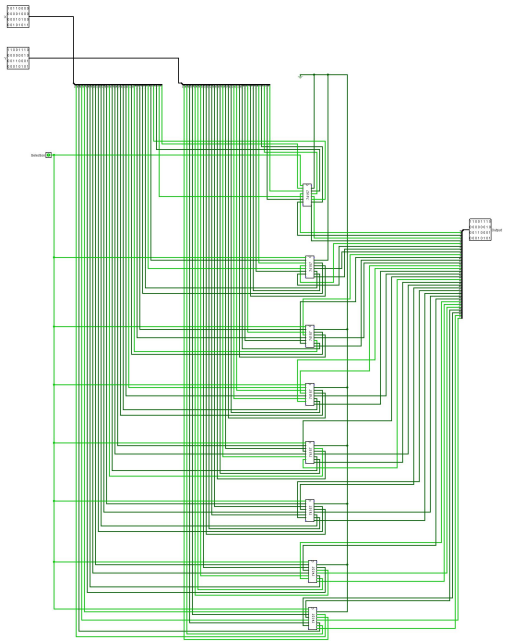
- 11 bit 2 to 1 Mux
- 12 bit 2 to 1 Mux
- 32 bit 2 to 1 Mux



(a) 11 bit 2×1 MUX



(b) 12 bit 2×1 MUX

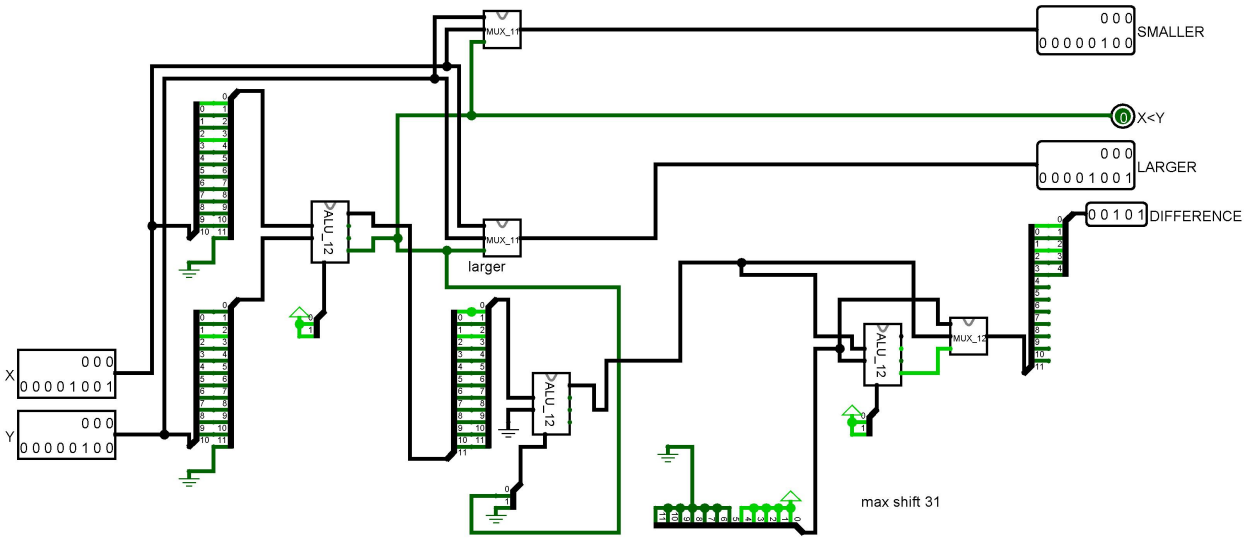


(c) 32 bit 2×1 MUX

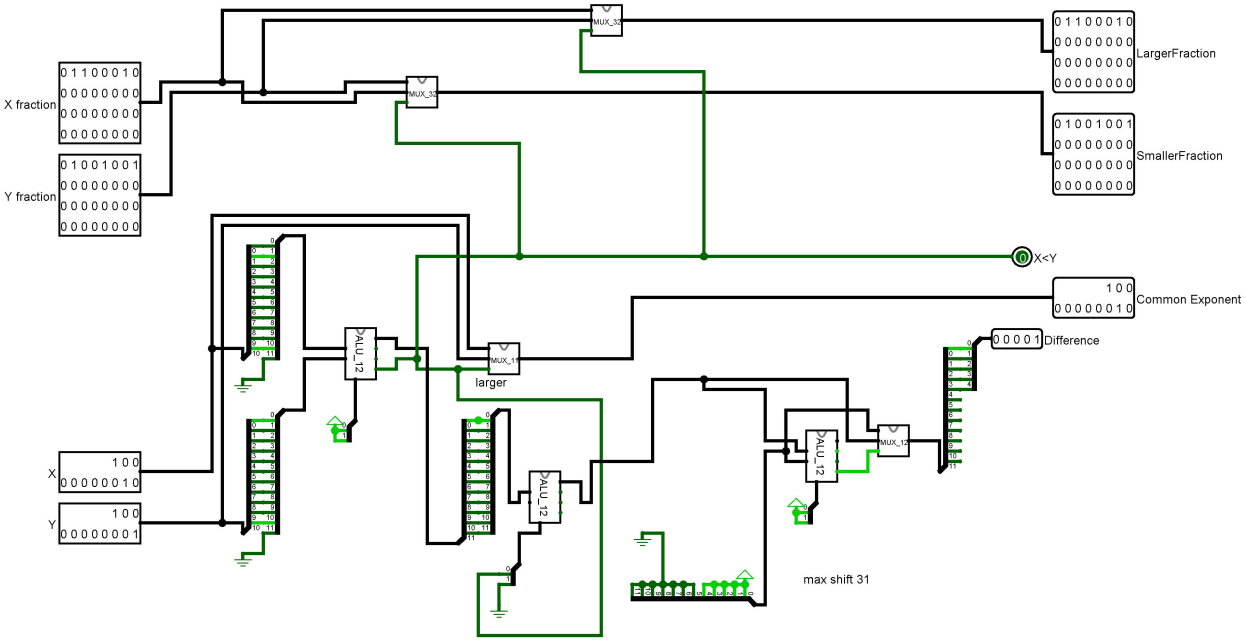
Figure 3: Multiplexer Circuits

5.2 Comparator Circuits

An comparator library was constructed to compare the exponenets . This library contains 2 circuits i.e. an 11 bit magnitude comparator and a comparator for small fractions.



(a) 11 Bit Magnitude Comparator



(b) Smaller Fractions Magnitude Comparator

Figure 4: Magnitude Comparators

5.3 Adder-Subtractor Circuit

A 32 bit adder-subtractor circuit was implemented to add or subtract 2 signed numbers

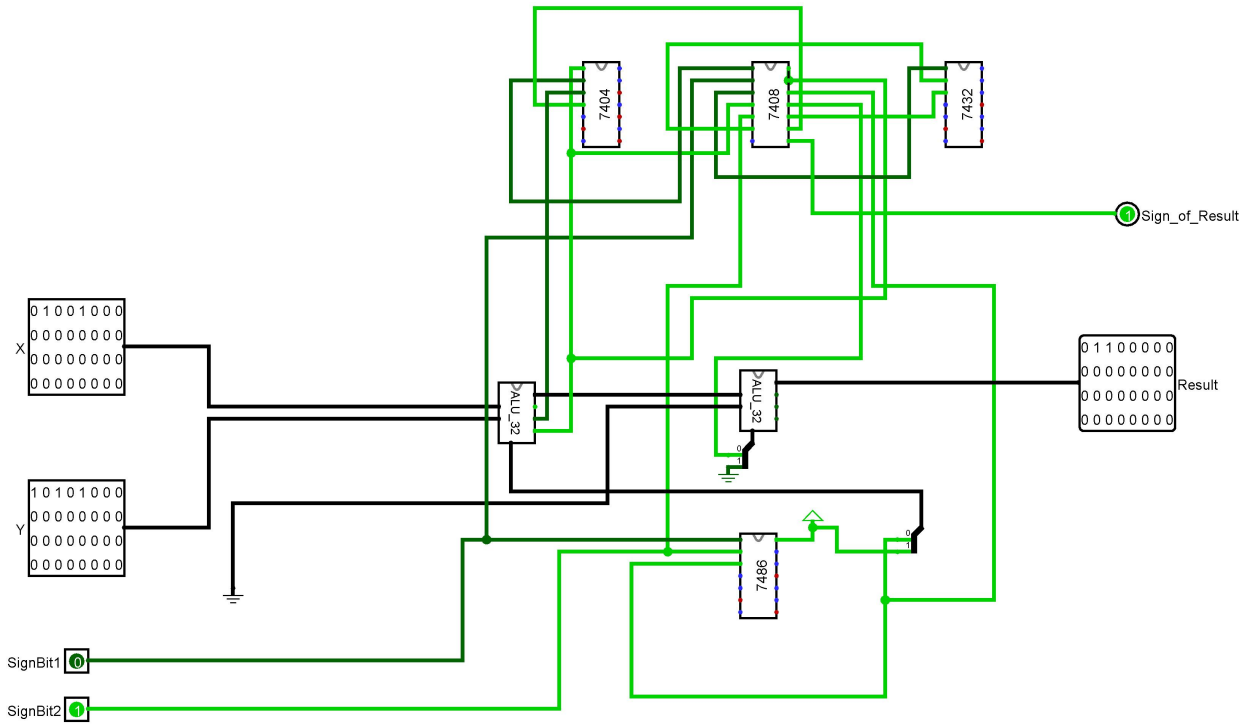
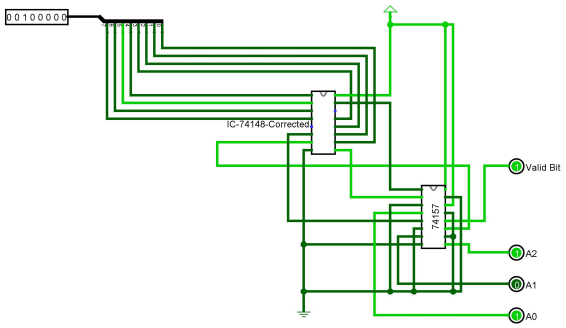


Figure 5: 32 bit Adder−Subtractor

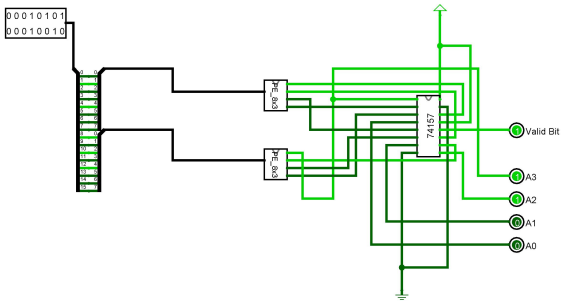
5.4 Encoder Circuits

To normalize a number, we need to locate the first set bit starting from MSB. 3 priority encoders was used for this purpose. Those are :

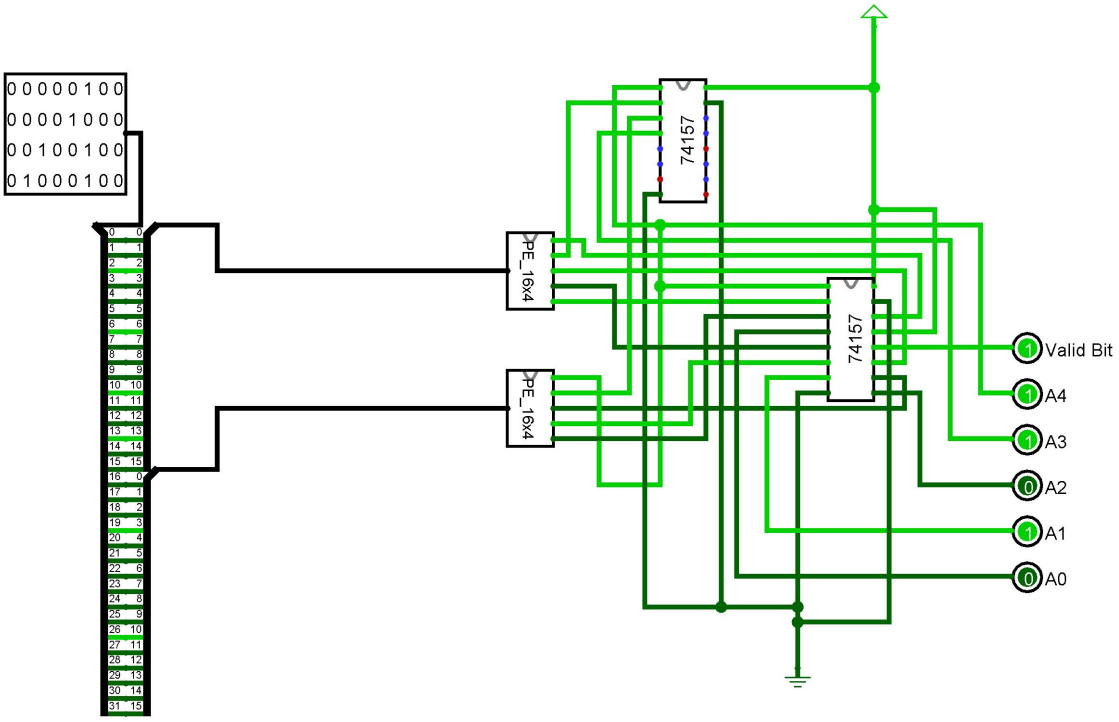
- 8 to 3 priority encoder
- 16 to 4 priority encoder
- 32 to 5 priority encoder



(a) 8×3 Priority Encoder



(b) 16×4 Priority Encoder



(c) 32×5 Priority Encoder

Figure 6: Encoder Circuits

5.5 Normalizer Circuit

The normalizer circuit does necessary change in mantissa and exponent to change the number in operable form for other circuits.

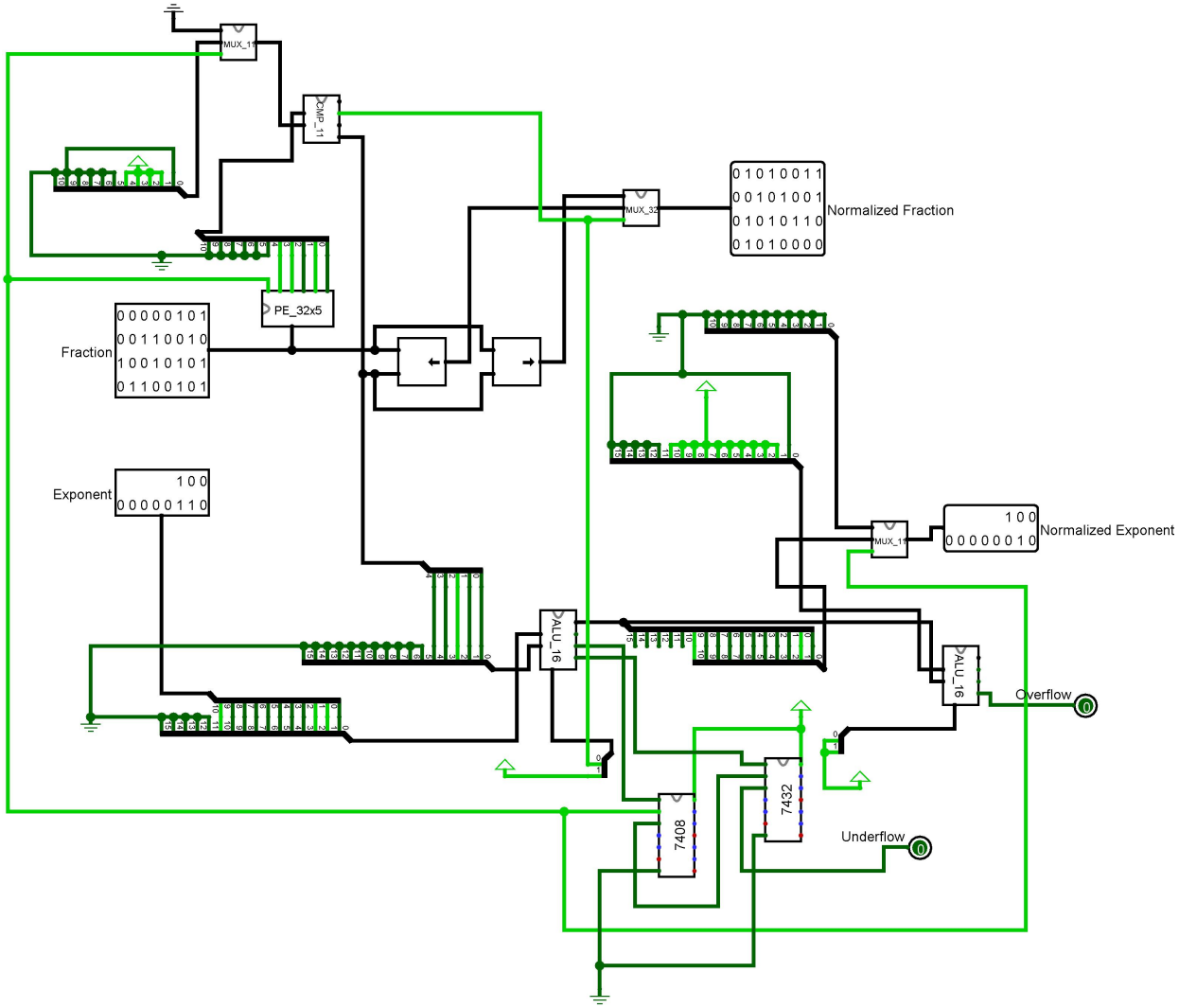


Figure 7: Normalizer Circuit

5.6 Rounder Circuit

This circuit does the rounding of the mantissa in the result.

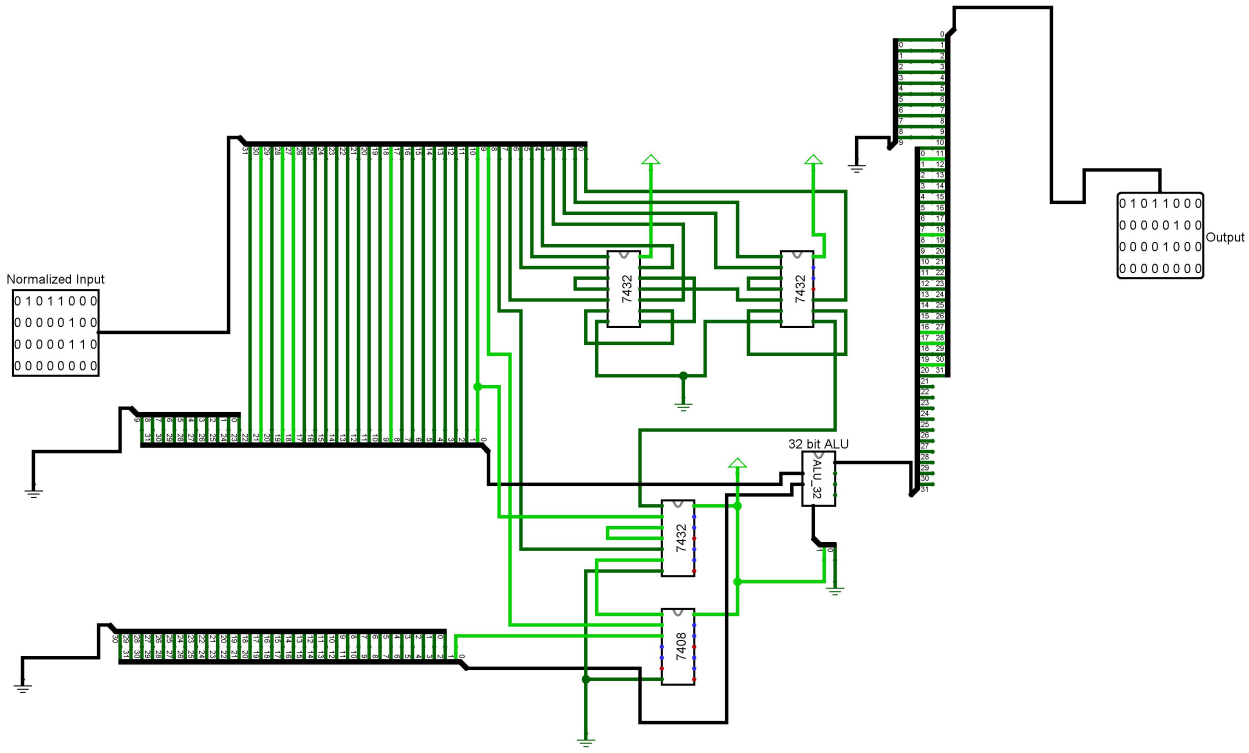
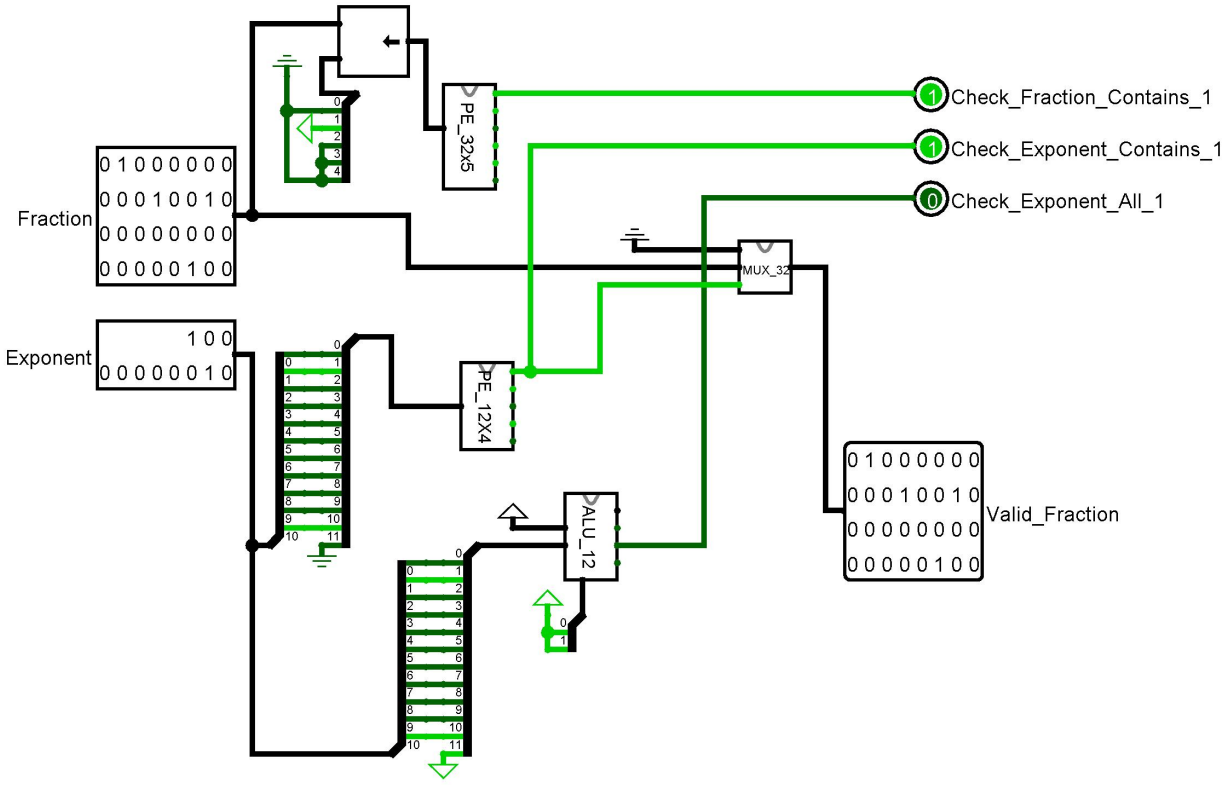


Figure 8: Rounder

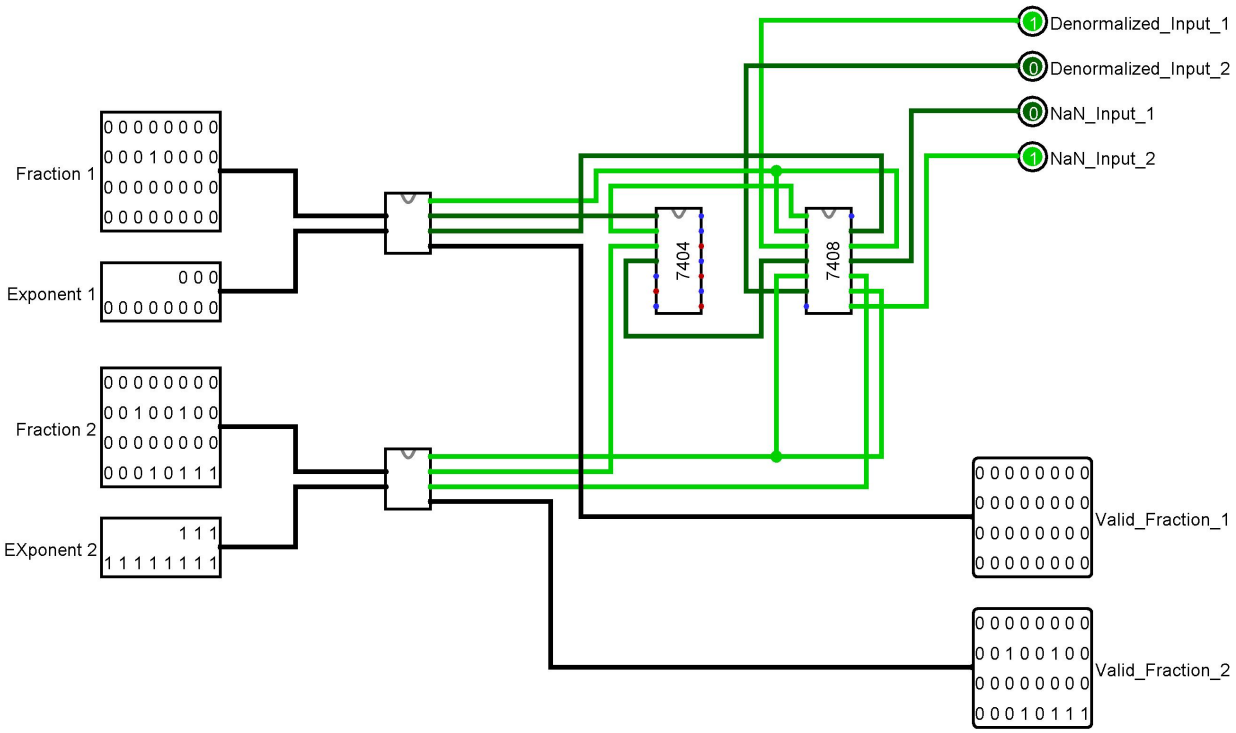
5.7 Input Pre-processor Circuits

This module validates the input given by user . It contains:

- A Single Input Checker that shows the validity of a single floating point number input
- A Input Validator containing 2 single input checkers to validate the whole input



(a) Single Input Checker



(b) Input Validator

Figure 9: Input Processing Utility

5.8 Arithmetic Logic Unit

The 32 bit Arithmetic Logic Unit is used in several places in the circuit to perform add operation. Additionally, 12 and 16 bit ALU's were also used in the design. All of them are of identical construct.

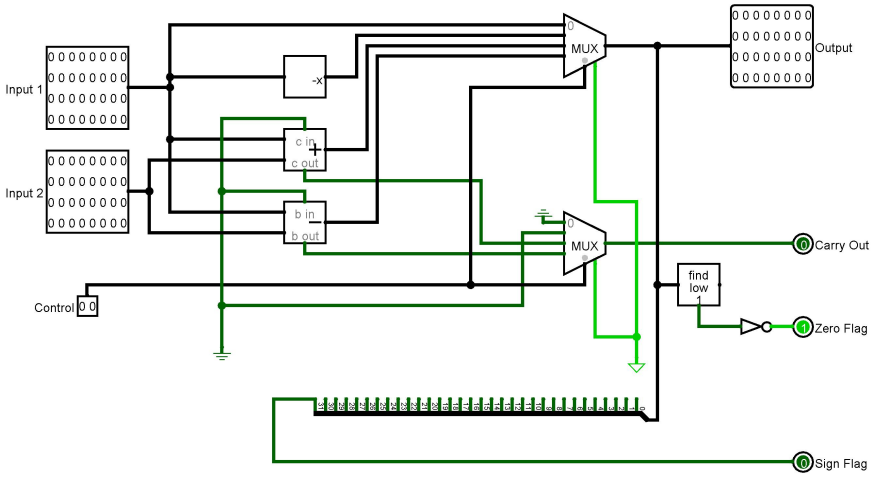


Figure 10: 32 bit ALU

5.9 Floating Point Adder

This is the actual floating point adder circuit which includes all the other libraries and circuits.

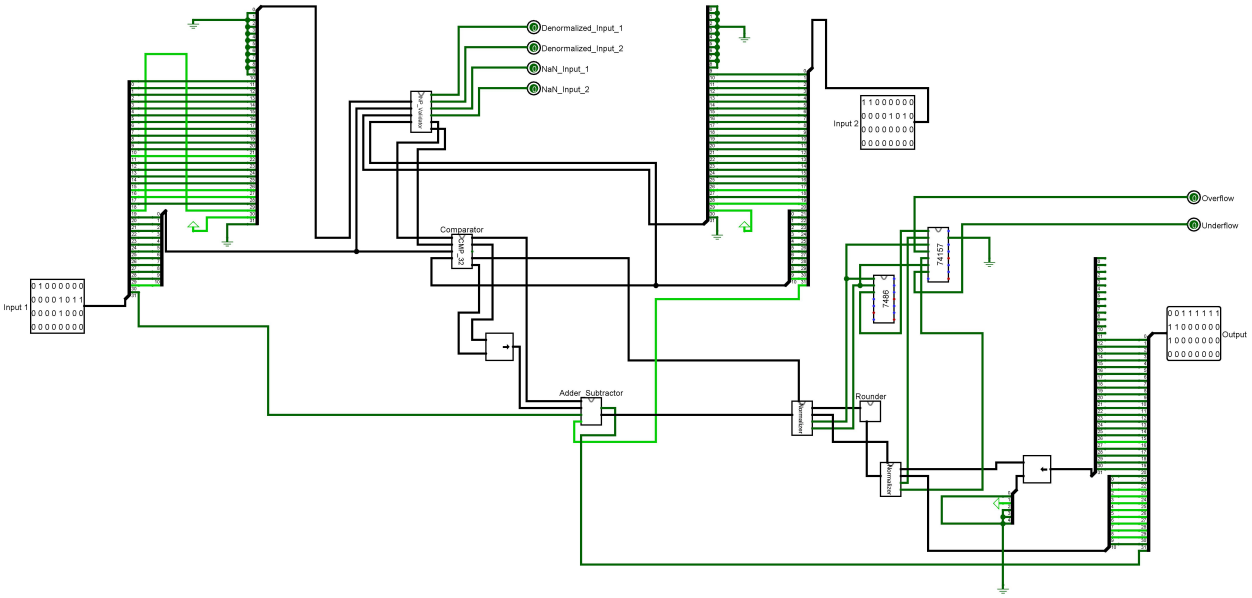


Figure 11: Floating Point Adder

5.10 Miscellaneous

This is a circuit tester used for checking whether the output matches the desired answer or not. Using a RAM provided by the simulator software, the circuit checks our output against pre-generated outputs. It is a sequential circuit which increments the memory address by a counter at every clock pulse and matches output.

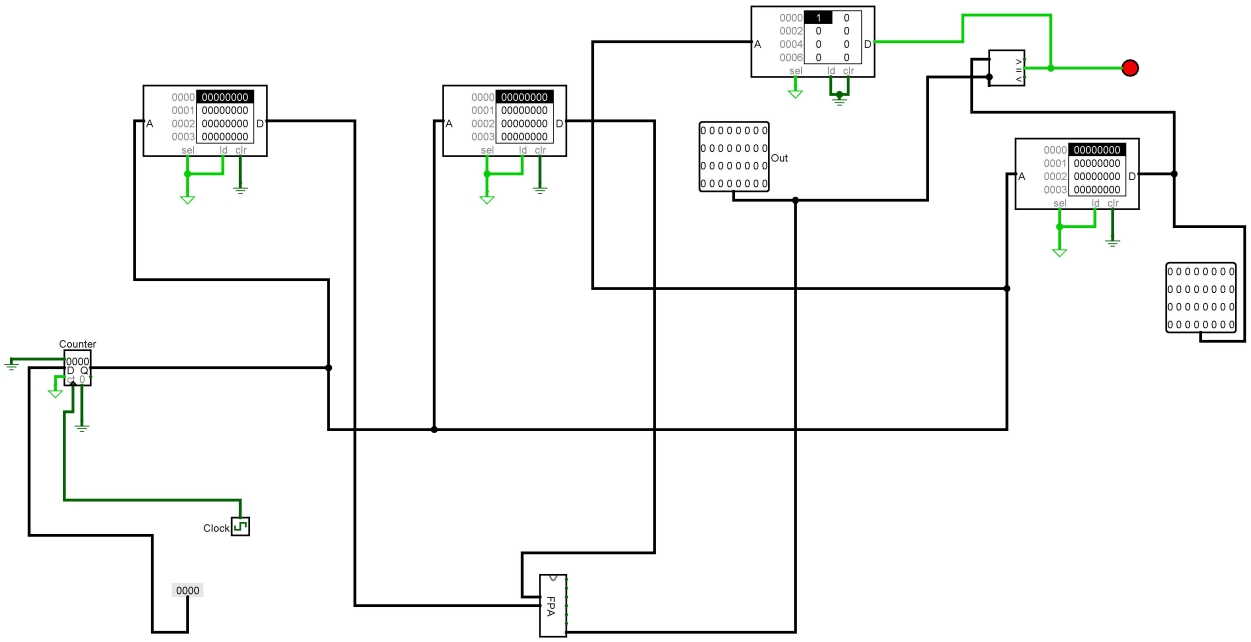


Figure 12: Circuit Tester

6 ICs Used with Count as a Chart

IC	Quantity
IC 7404	2
IC 7408	5
IC 7432	6
IC 7486	2
IC 74148	20
IC 74157	123
ALU (12 bit)	11
ALU (16 bit)	4
ALU (32 bit)	3
Shifter (Left)	5
Shifter (Right)	3
Total	184

Table 4: ICs Used with Quantity

7 The Simulator Used along with the Version Number

Logisim - 2.7.1

8 Discussion

This assignment familiarizes us with a crucial aspect of Computer Arithmetic, named 'Floating Point Addition' in the light of IEEE 754 standard, a widely adopted floating-point formatting standard in modern computer systems.

First comes the process of building helping libraries containing circuits of similar genre and varying bit capacity. As of Encoder and Multiplexer libraries we used an ample amount of cascading to get to an understandable and maintainable design. The ALU library consisting of 3 different circuits, was built using default 'Adder' tools provided by the simulation software, as per the given specification. Comparator library saliently relies upon the ALU library for performing subtraction. Afterwards, it processes the output of the ALU to deliver information regarding comparison.

Next comes the major building blocks of the Floating Point Adder, Adder Subtractor, Input Validator, Normalizer and Rounder which are in turn built upon the aforementioned libraries. While designing these, we put utmost attention to minimizing the number of ICs by carefully altering myriad design decisions. For instance, previously we lacked the prospect of doing transfer operation in the ALU. Though it may seem unnecessary, in essence, it was costing us several extra Multiplexers. Upon realizing the same, we introduced the transfer operation in ALU. In the input validator, we implemented an error pre-checking mechanism, which, primarily with the help of Encoders and ALU, detects what IEEE specified range the inputs fall between. Accordingly, we showed output flags for whether the input is a 'Denormalized Number' or 'NaN' or valid. As for Rounder, omitting the appended '01' in front, we considered 20 bits as actual mantissa, the two after them as 'Guard' and 'Round' bits, and all after them as Sticky bits.

Designing the final circuit required merging the pre-built components while carefully handling the control flow. We performed Normalization again after Rounding, since, in an extreme case, the output from the Normalizer may turn out to be denormalized after being processed by the Rounder. Overflow, Underflow and Error checker flags were shown as outputs in the final circuit. As we appended two bits '01' in front of the mantissa to imitate the hidden set bit, while showing the output, we had to ignore those two bits using a left shifter.

To verify our outputs, we wrote a python script to generate bulk random inputs and corresponding outputs. Afterwards, we used the provided RAM in the Logisim simulation software to build a sequential circuit driven by clock pulses. At every clock pulse, it enhances the address by 1, locates the next position, reads the data, performs the simulation and writes it back to another RAM. We then read back that RAM and match the outputs with our pre-calculated outputs.

Since the design process incorporates a deep hierarchy of components, counting the total number of ICs required for the whole design was a challenging task to do by hand. Therefore, we first converted our design into a Directed Graph format, which indicated which components hold which other components as children at what count. We then wrote another python script to run a Depth First Search on the directed graph to count the number of ICs through bottom-up Dynamic Programming.

While building through, at numerous junctions, we had to choose among several different design decisions and we hope we managed to take the optimal path in most of those cases. Extensive randomized and automated testing was performed afterwards, which further enkindles the hope that ours has been a fully working optimized design.

We wish to learn a lot more about the miniscule details of the design and we hope to extend our design to incorporate further operations in future.