CSE 210
Computer Architecture Sessional

Assignment-2
Floating Point Adder

Section - A2
Group - 01

Group Members:

1. 2105032 - Nawriz Ahmed Turjo
2. 2105047 - Himadri Gobinda Biswas
3. 2105048 - Shams Hossain Simanto

Date of Submission: 16 November 2024

1 Introduction

In this project, we have developed and implemented a 32-bit Floating Point Adder (FPA) circuit as a core component of our study in computer architecture. Floating Point Arithmetic (FPA) is essential in modern computing, as it allows for a vast range of values that can represent very small fractions as well as very large numbers. This capability is crucial for scientific calculations, engineering applications, financial modeling, and any task that requires precision beyond integers. Floating-point representation provides a standardized format for encoding numbers, typically consisting of three key components: a sign bit, an exponent, and a significand (or mantissa).

Our implementation conforms to the IEEE 754 standard, which is widely adopted for floating-point arithmetic in modern computing systems. The circuit uses a 32-bit architecture where:

❒ **Sign Bit**: 1 bit is dedicated to indicating whether the number is positive (0) or negative (1).

❒ **Exponent Field**: 9 bits are used to represent the power of the base (usually 2), stored in a biased format to support a large dynamic range, including both positive and negative exponents.

❒ **Mantissa (or Fraction) Field**: 22 bits capture the precision of the number, encoding its significant digits.

To calculate the actual exponent, the bias, $2^{9-1} - 1 = 255$, is subtracted from the stored value. This approach allows the representation of a wide range of values, making floating-point numbers suitable for both extremely large and very small magnitudes.

1.1 Floating-Point Addition Process

The floating-point adder is a specialized digital circuit designed to handle the complexities of arithmetic operations, including addition and subtraction. The process includes:

1. **Exponent Alignment**: The exponents of the two operands are compared. The significand of the smaller exponent is shifted right to align the two numbers on the same scale.

2. **Mantissa Operation**: Once aligned, the mantissas are added or subtracted based on the operation and the signs of the numbers.

3. **Normalization**: The result is adjusted to fit the normalized floating-point format by shifting the mantissa and modifying the exponent accordingly.

4. **Rounding**: The final value is rounded to fit within the available precision, ensuring minimal error in the representation.

5. **Sign Determination**: The sign of the result is derived based on the operation and the input signs.

This circuit is also designed to handle edge cases like underflow, overflow, and rounding errors. These challenges require careful binary arithmetic operations and logical circuits to ensure accuracy and efficiency, particularly when the exponents of the two operands differ significantly. In such cases, mechanisms to shift and align significands correctly and normalize the results are essential to maintain the IEEE 754 format.

1.2 Applications and Importance

Floating-point adders are indispensable in applications such as:

- ❑ **Scientific and Engineering Calculations**: Simulations, data analysis, and machine learning models require high precision and wide-range representation.
- ❑ **Financial Modeling**: Calculations involving significant digits and extreme magnitudes, such as interest rates or stock price predictions, depend on floating-point arithmetic.
- ❑ **Computer Graphics**: Real-time rendering, transformations, and lighting calculations rely on floating-point operations for accuracy and speed.

Modern processors often include dedicated hardware for floating-point arithmetic, such as co-processors or specialized CPU units. These hardware accelerators enable efficient and precise floating-point operations, reducing computational overhead in performance-critical applications.

1.3 Learning and Implementation Insights

By designing and implementing this FPA circuit, we gained a deeper understanding of the intricacies of floating-point arithmetic. This includes operations that are often abstracted away in high-level programming languages. Through this project, we explored the hardware-level operations and logic circuits necessary to achieve precise and efficient calculations. Additionally, we delved into optimization techniques for addressing common issues such as rounding errors, underflows, and overflows, which are crucial for ensuring consistent performance across a wide range of inputs.

The process of developing this 32-bit FPA circuit not only solidified our understanding of floating-point representation and binary arithmetic but also provided valuable insights into the challenges associated with precision and range in digital computing. Our implementation adheres to industry standards, offering an accurate and efficient tool for adding floating-point numbers while preparing us for more advanced digital design tasks in computer architecture.

The final result is a robust floating-point adder circuit that demonstrates the principles of efficient hardware design, paving the way for future work in high-performance computing and digital system design.

1.4 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)
30-22	Exponent
21-0	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 255.

1.5 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 22-bit mantissa and an 9-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 255, and the mantissa is the maximum value for a 22-bit representation. The binary representation is $1.1111111111111111111111 \times 2^{255}$. In approximate decimal form, this represents a number of approximately 6.432×10^{76}.

For the smallest positive normalized number, the exponent is set to its minimum value of -254, and the mantissa is the minimum value (just the implicit leading bit). The binary representation is $1.0000000000000000000000 \times 2^{-254}$, approximately equal to 3.454×10^{-77} in decimal.

These values encapsulate the range of positive normalized numbers that can be precisely represented by your floating-point adder with a 22-bit mantissa and an 9-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.

1.6 EEE 754 encoding of floating point numbers

Exponent	Fraction	Object Represented
0	0	0
0	Non Zero	± Denormalized Number
1-510	Anything	± Floating Point Number
511	0	± Infinity
511	Nonzero	NaN(Not a Number)

Table 2: IEEE 754 Encoding

1.7 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),
- "**exponent$_{\mathbf{min}}$**" 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:

$$0.\underbrace{0000000000000000000001}_{\text{Fraction}} \times 2^{-254}$$

Which is equivalent to 1.0×2^{-276}. This is considerably smaller than the smallest positive normalized number, which is $1.0000000000000000000000 \times 2^{-254}$. Denormalized numbers play a crucial role in maintaining precision for very small values close to zero in floating-point arithmetic.

1.8 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 normalized, 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 normalized 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	9 bits	22 bits

Table 3: Problem Specification

3 Flowchart of the addition/subtraction algorithm

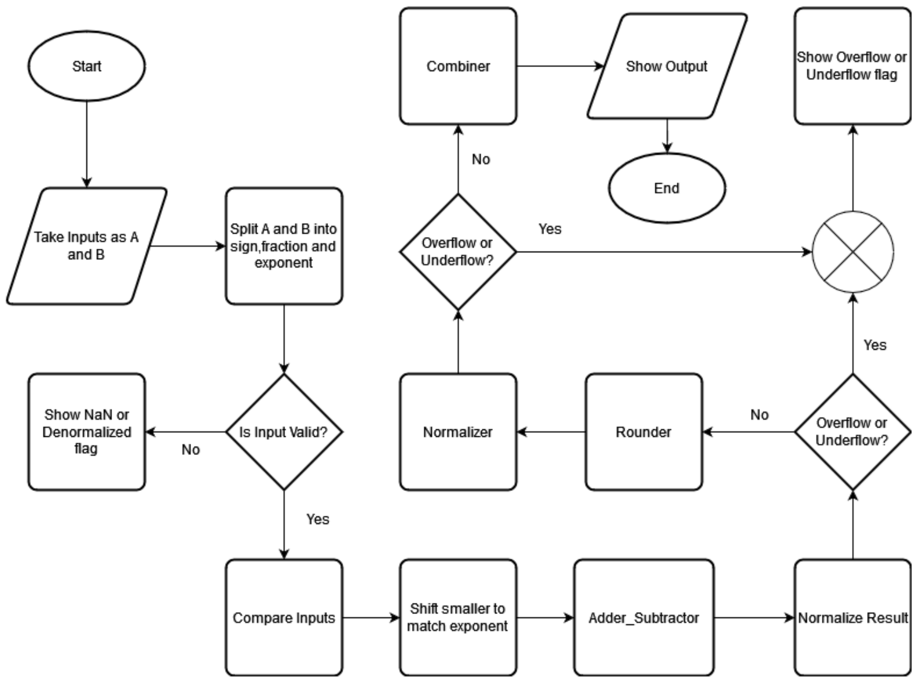


Figure 1: Flowchart of the addition/subtraction algorithm

4 High-level block diagram of the architecture

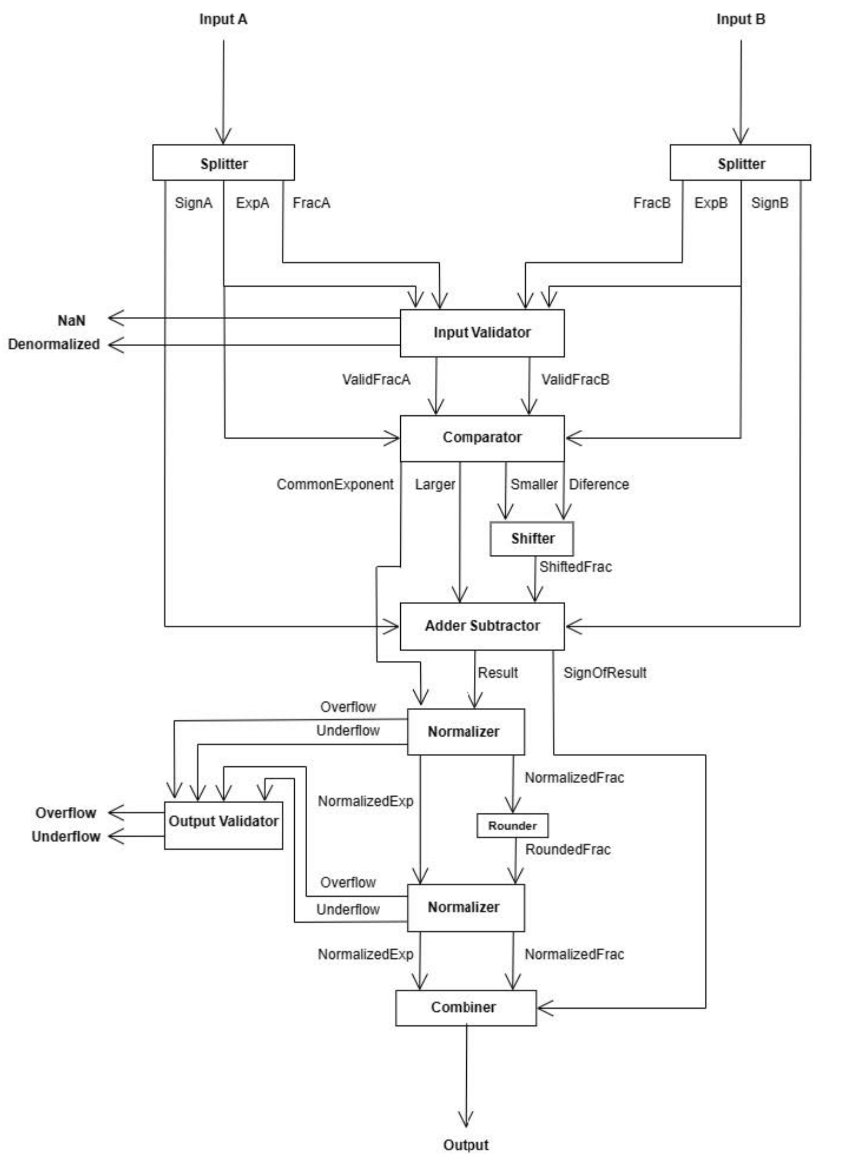


Figure 2: High-Level Block Diagram of the Architectrue

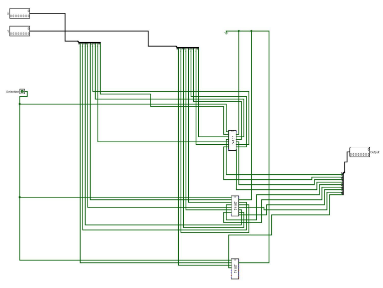
5 Detailed Circuit Diagram of the Important Blocks

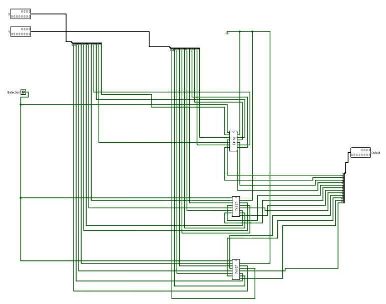
Some libraries and circuits were implemented to enhance and simplify the final circuit design. Those are as follows:

5.1 Multiplexer Circuits

The modular circuits in this library are as follows

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


(a) 9 bit 2×1 MUX


(b) 12 bit 2×1 MUX

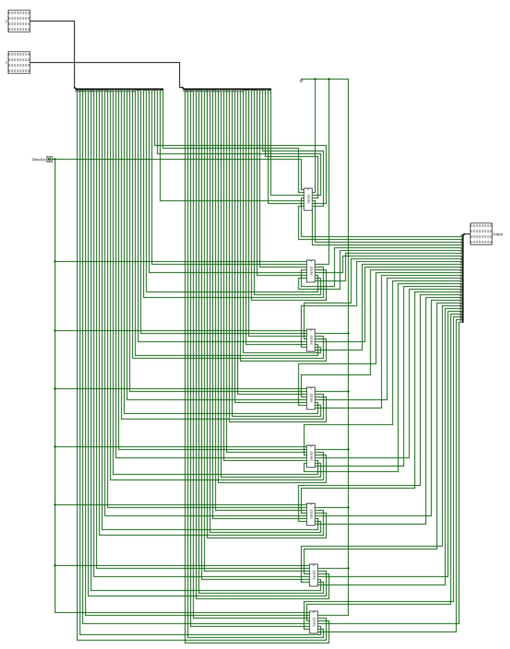
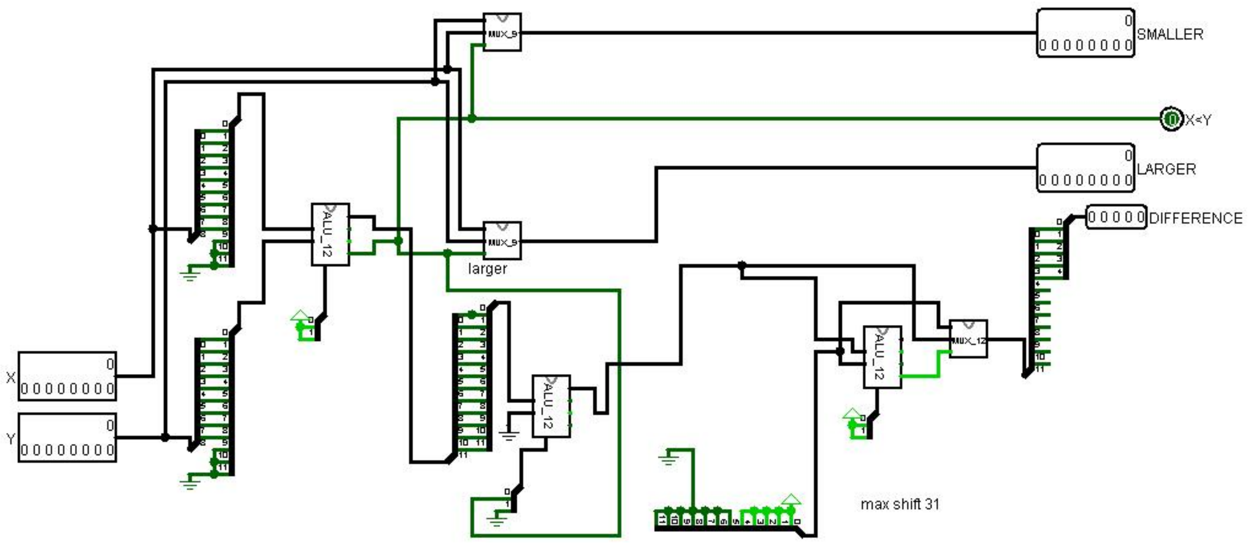

(c) 32 bit 2×1 MUX

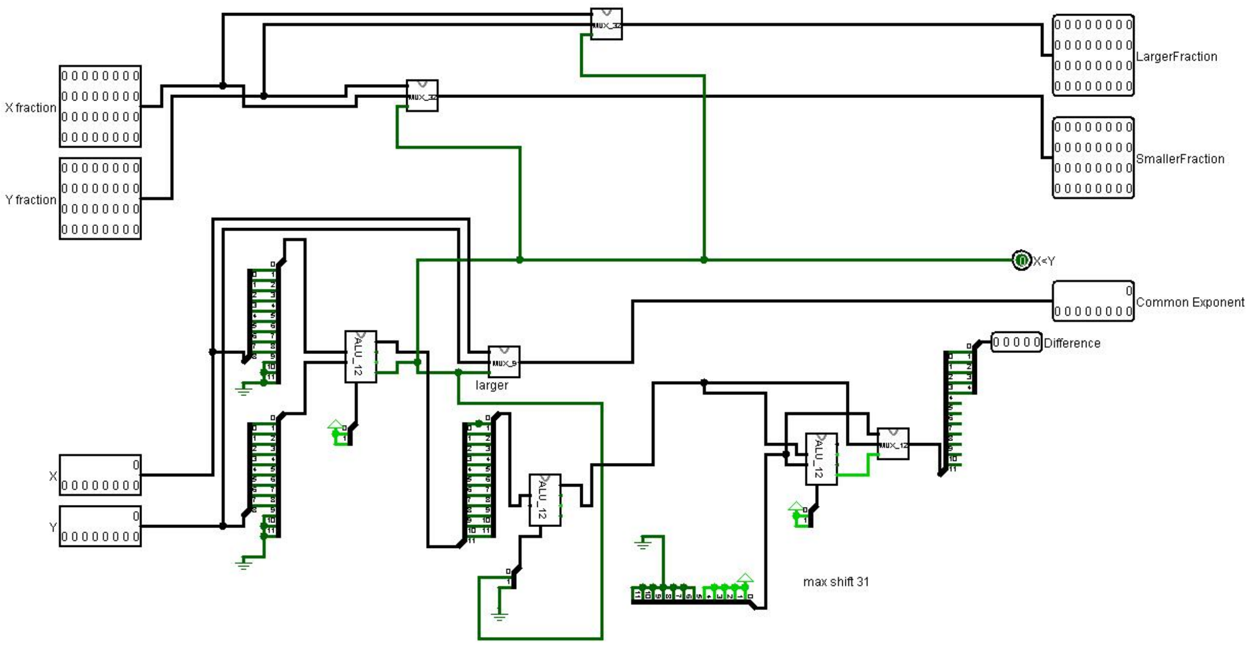
Figure 3: Multiplexer Circuits

5.2 Comparator Circuits

An comparator library was constructed to compare the exponenets. This library contains 2 circuits i.e. an 9 bit magnitude comparator and a comparator for small fractions.



(a) 9 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

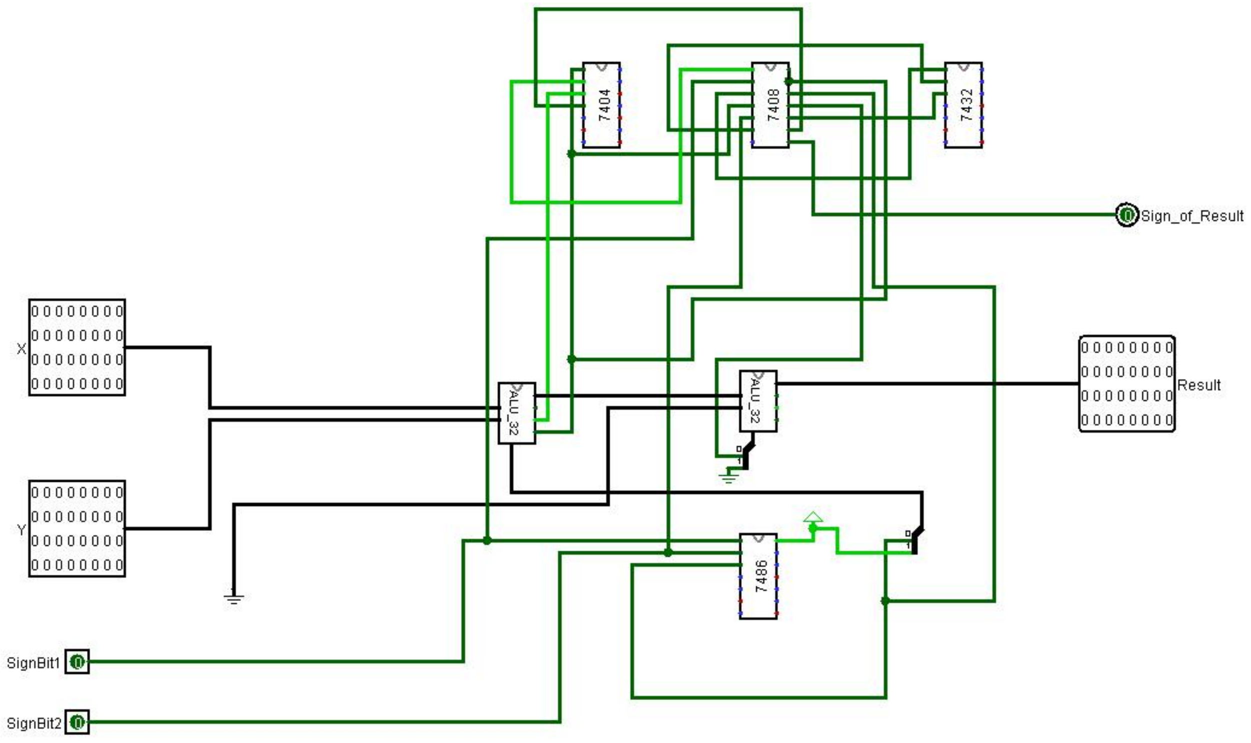
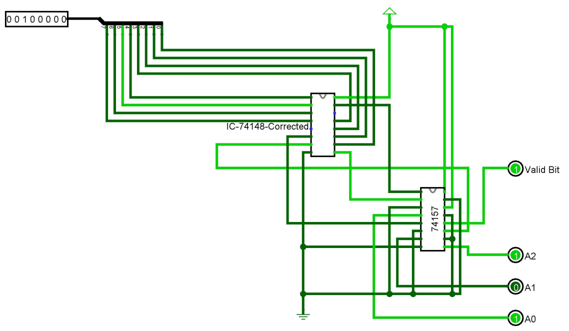


Figure 5: 32 bit Adder−Subtractor

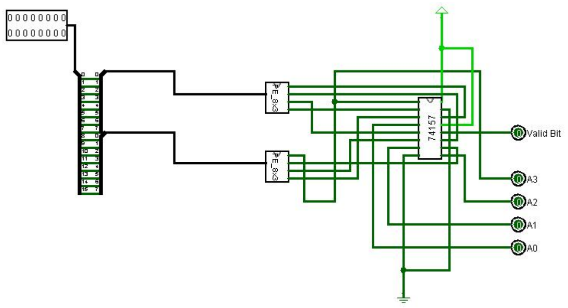
5.4 Encoder Circuits

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 :

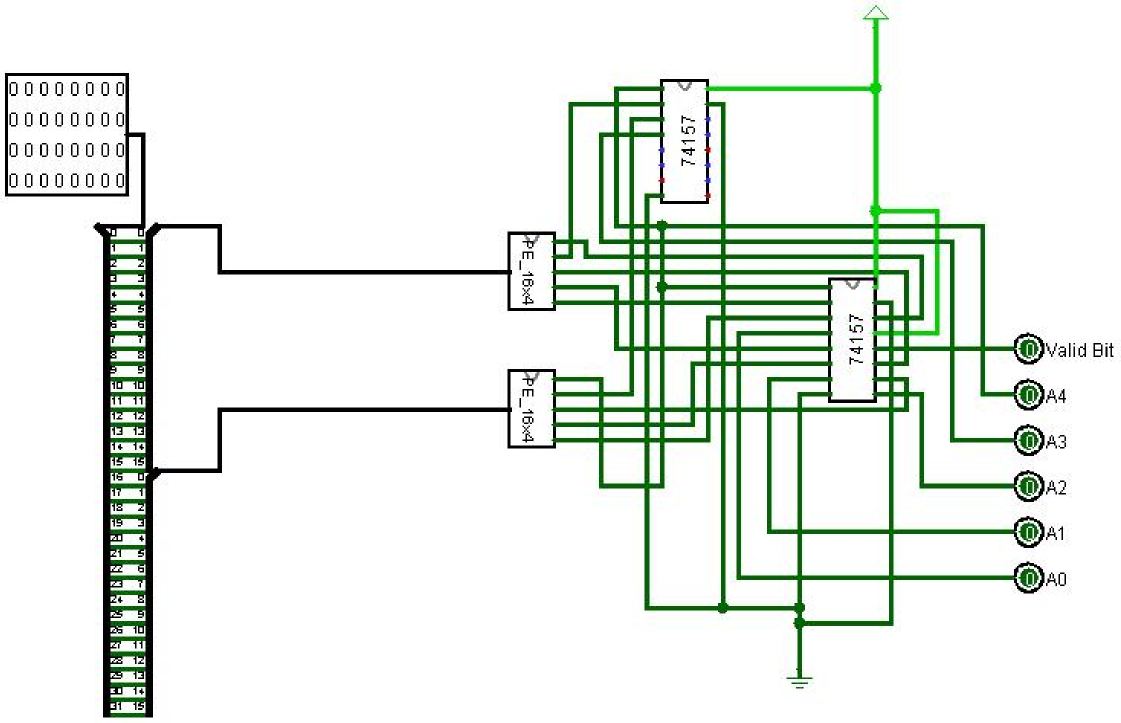
- 8 to 3 priority encoder
- 16 to 4 priority encoder
- 32 to 5 priority encoder



(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.

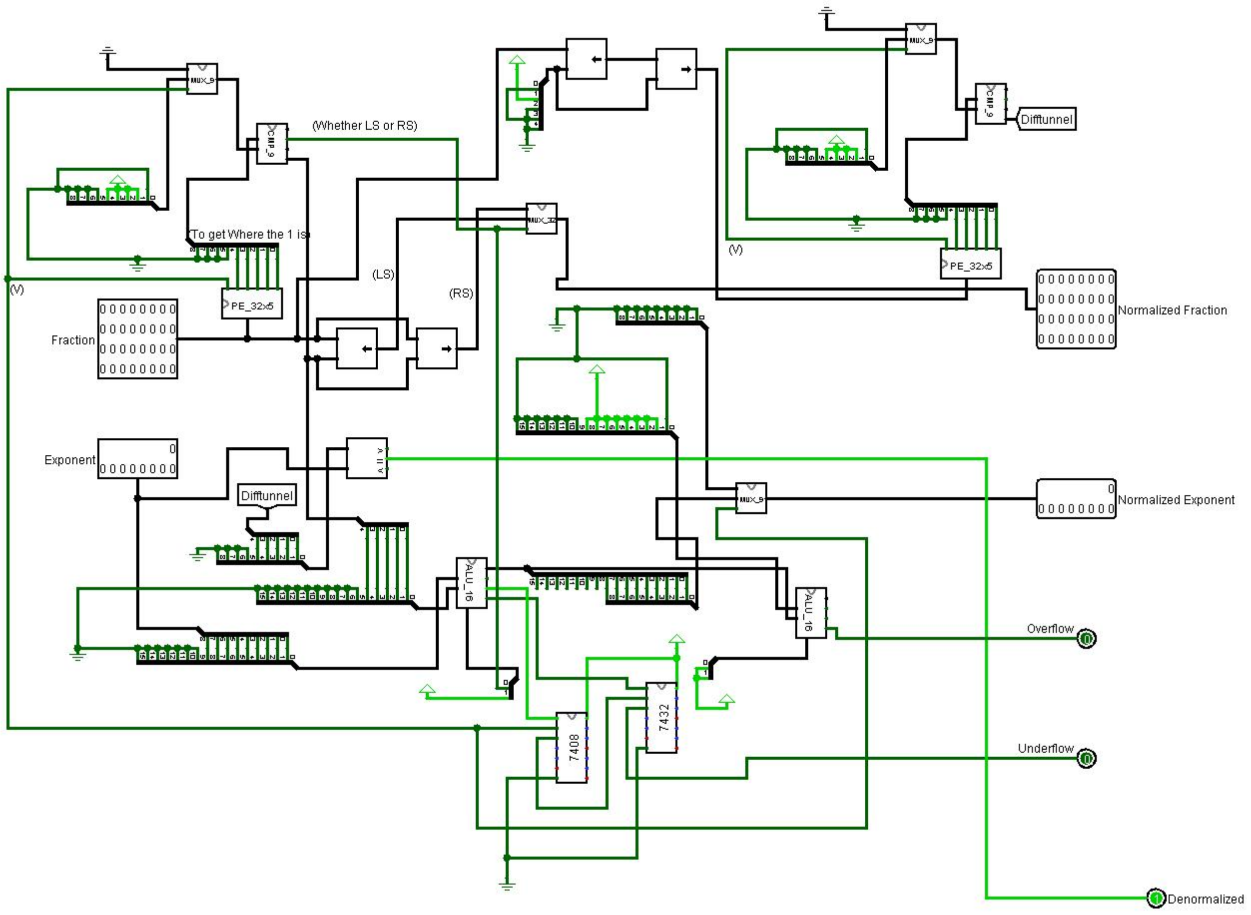


Figure 7: Normalizer Circuit

5.6 Rounder Circuit

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

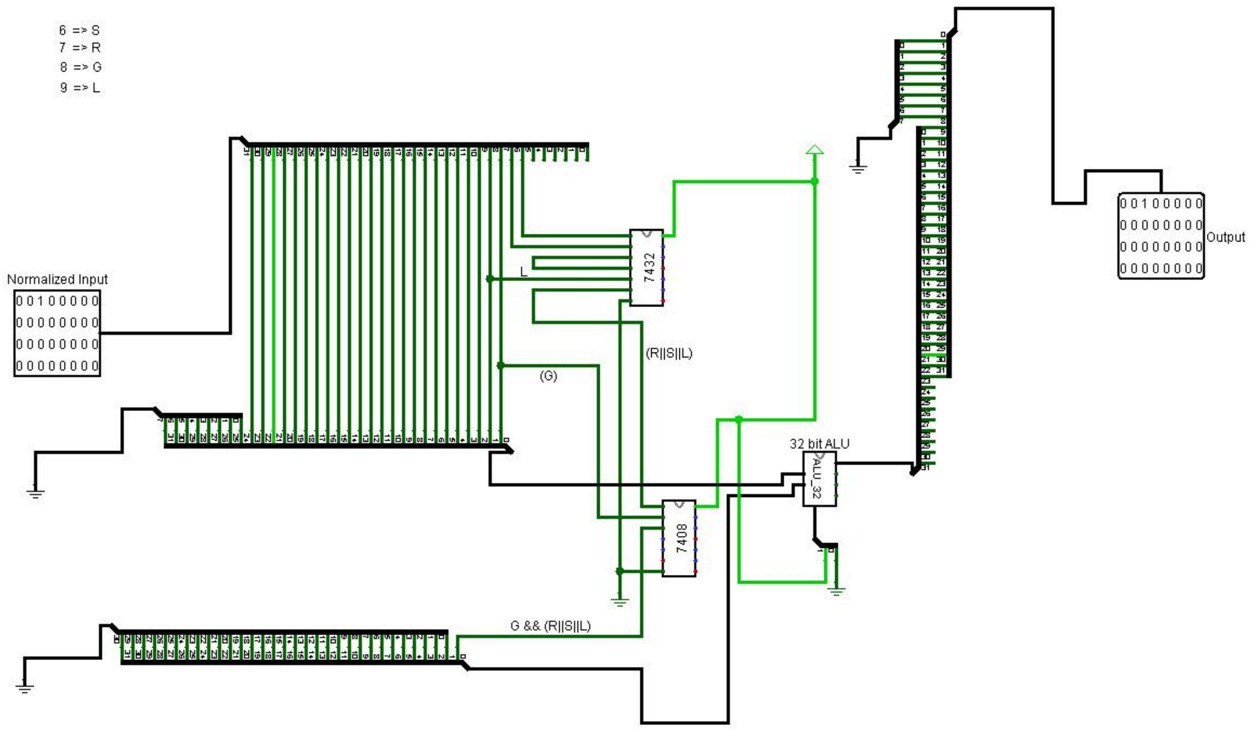
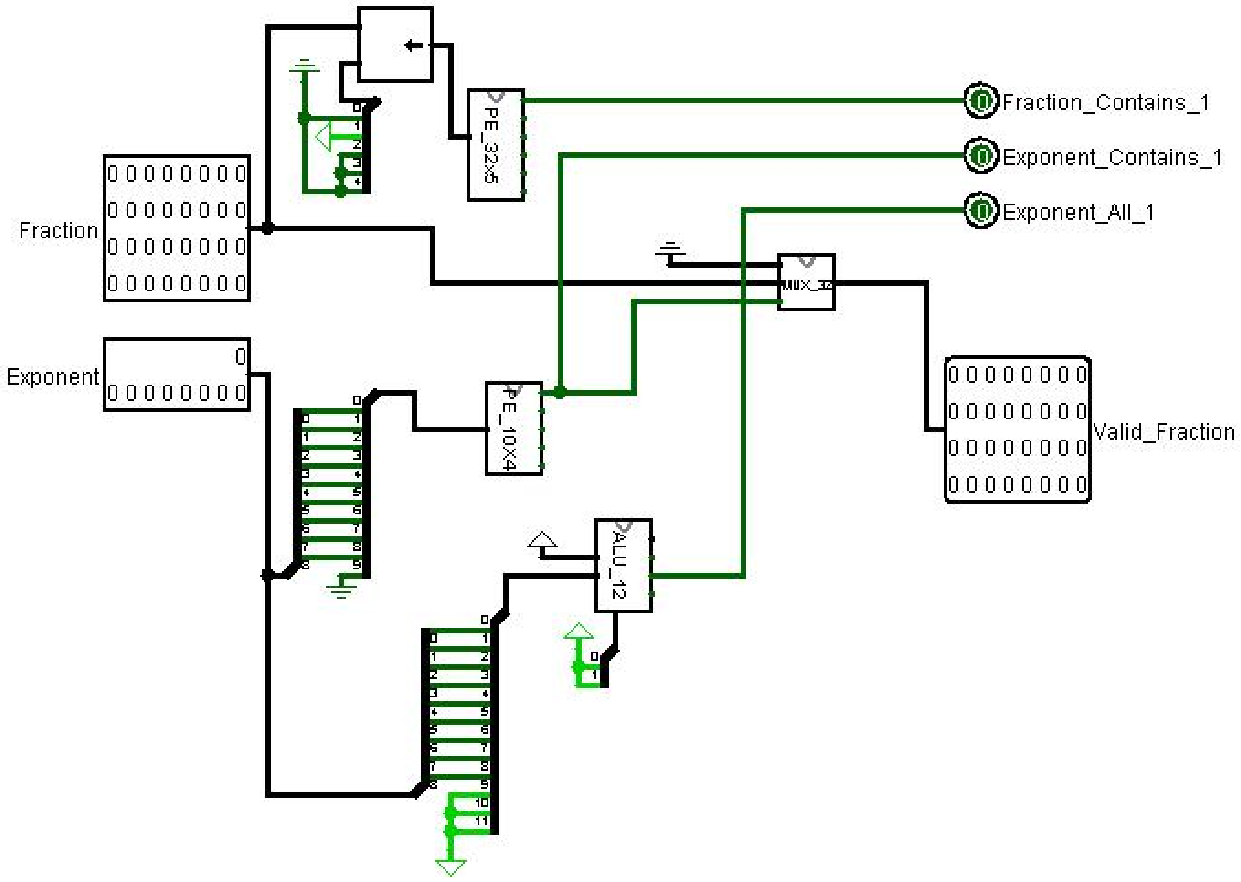


Figure 8: Rounder

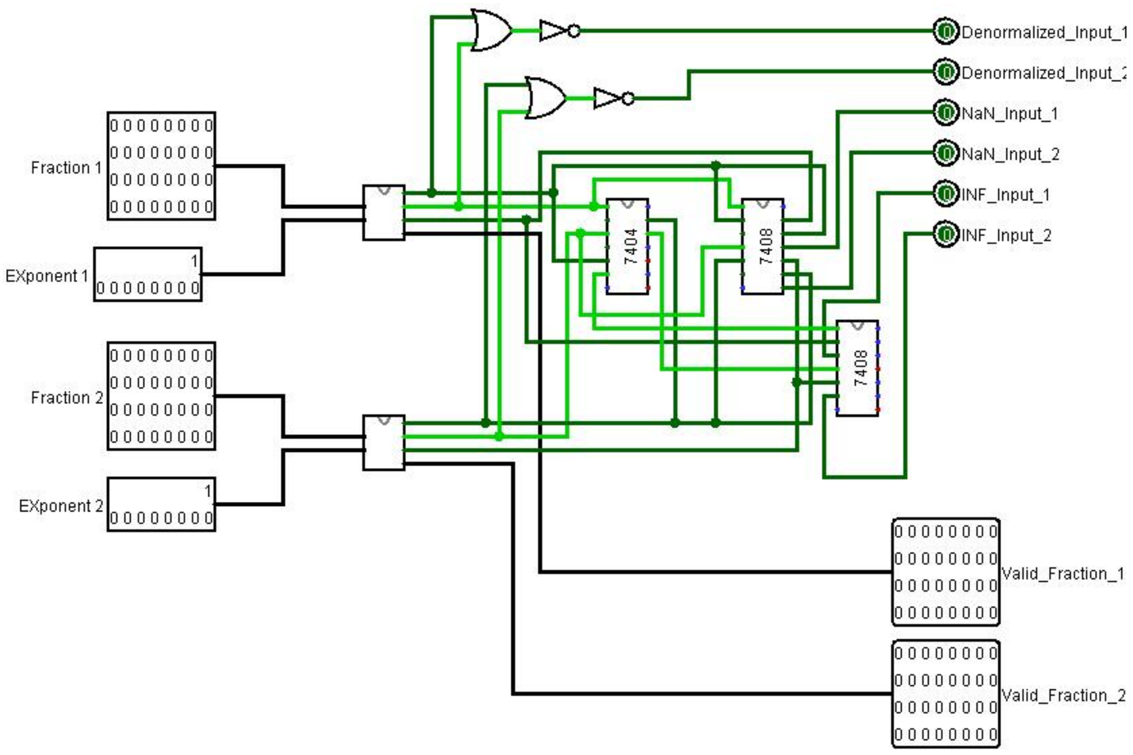
5.7 Input Pre-processor Circuits

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 Pre-processor Circuits

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 ALUs were also used in the design. All of them are of identical construct.

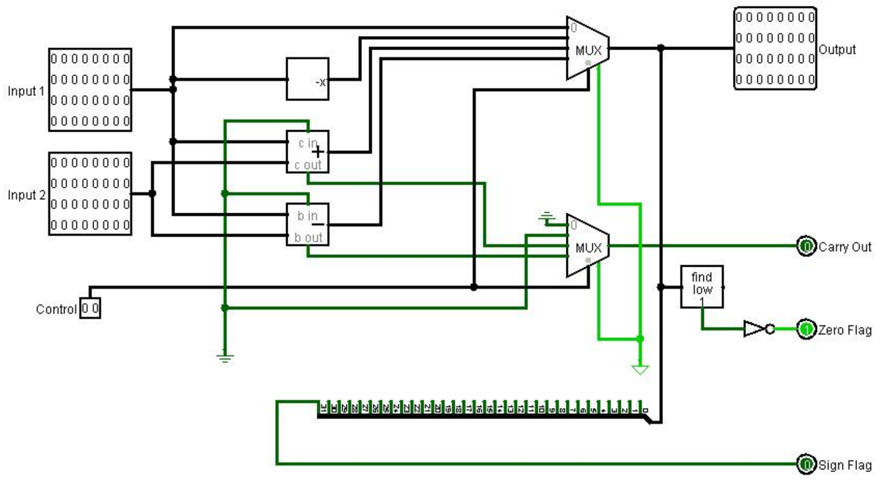


Figure 10: 32 bit ALU

5.9 Floating Point Adder

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

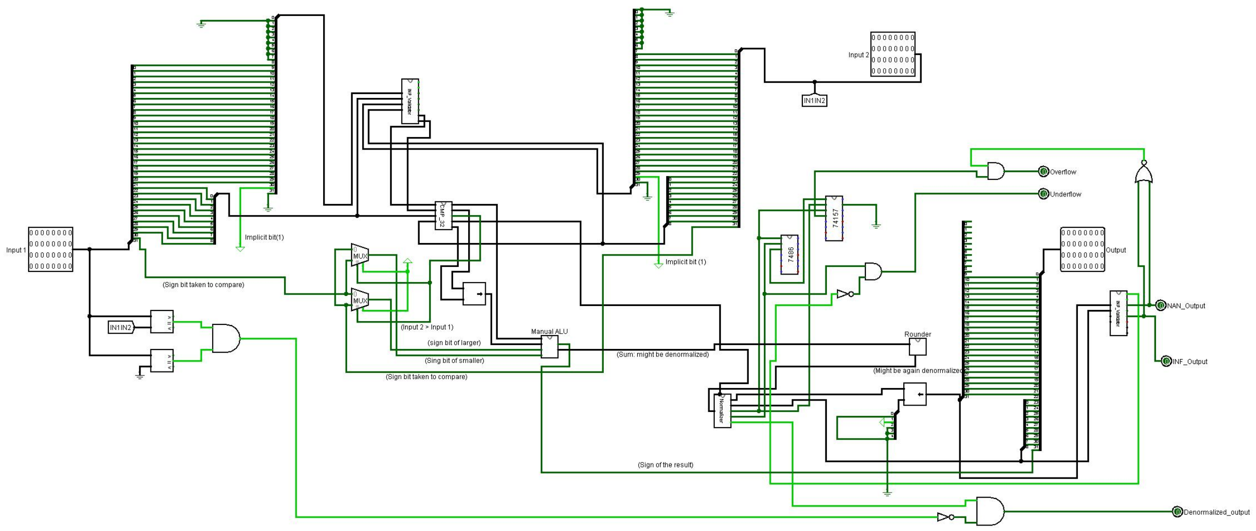


Figure 11: Floating Point Adder

6 ICs Used with Count as a Chart

IC	**Quantity**
IC 7402	1
IC 7404	2
IC 7408	6
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	186

Table 4: ICs Used with Quantity

7 The Simulator Used along with the Version Number

Logisim - 2.7.1

8 Discussion

In this assignment, we put a concerted effort to ensure the accuracy and utmost efficiency of the design. First of all, we built the 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. Then we focused on building different modules of the FPA separately using the aforementioned libraries. This approach helped us to maintain a nice flow and also properly divide workload among ourselves.

Although we had thoroughly understood the theory of floating-point arithmetic (FPA), soon we came to know that only theoretical knowledge is far from enough to actually implement the whole circuit as we had to overcome a lot of hurdles in various minuscule details of the circuit components and wirings.

The first time we got stuck was the case when X and Y had different signs and Y had different signs and $|Y| > |X|$. This case was showing an output with correct value but wrong sign. After long observation, we found that we had sent the factions to the comparator without checking their sign bits which resulted in our FPA always assuming $X > Y$. So we debugged this logic using two multiplexer components provided by the simulation software which ensured we always take the correct sign bit.

Designing the normalizer was another intricate and time-consuming task. We had faced various edge cases in this module which created issued and had to be rectified later. We first decided to keep flags for Denormalized and INF inputs, but later we reverted from that decision as we were notified that no invalid inputs will be given for testing. Also the case when exponent and fraction all are 0 will not be considered as denormalized was handled properly.

For the rounder circuit, the theory of G, R, S, L bits was learnt comprehensively and implemented with great care. However, the rounding buffer uses only a single sticky bit. By incorporating additional sticky bits, the buffer could hold more information, thereby enhancing the precision of the result. But through testing we ensured that our implementation does not need more sticky bits. The overflow, underflow and NaN output flags were also challenging to implement and we so we built them with utmost caution.

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. 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.

During the whole design process, we put utmost attention to minimizing the number of ICs by carefully altering myriad design decisions. The total IC count actually covers the complete hierarchy of components as we only outsourced ALU (for addition and subtraction, as told in the specification) and all other helping sub-components were implemented manually.

Overall, understanding, designing and implementing the floating point adder was a thrilling task for all of us. It provided us with great insights practical knowledge about the precise processing of Floating Point Arithmetic in computers.

9 Contribution of Each Member

Design

- ✐ Normalizer - 2105047, 2105048
- ✐ Rounder - 2105032, 2105048
- ✐ Floating point adder - 2105032, 2105047, 2105048
- ✐ Utility - 2105032
- ✐ Optimiztion - 2105032
- ✐ Flags - 2105032
- ✐ Comparator - 2105047
- ✐ MUX - 2105047
- ✐ Manual ALU(for addition and subtraction)- 2105048
- ✐ Encoder- 2105048
- ✐ Debugging, Testing - 2105047, 2105048
- ✐ Wire Labelling - 2105048

Report

- ➥ Section: 1, 2, 3, 5, 7 - 2105032
- ➥ Section: 5, 6, 8 - 2105047
- ➥ Section: 4, 8 - 2105048