CSE 210
Computer Architecture Sessional

Assignment-2
Floating Point Adder

Section - C2
Group - 08

Group Members:

1. 2105161 - Sadia Naushin
2. 2105163 - Dibbo Chowdhury
3. 2105168 - Nusrat Jahan Nidhi
4. 2105174 - Nujhat Sadia

1 Introduction

Floating-point representation is a method for representing real numbers, enabling the expression of both extremely small and extremely large values.
Disclaimer. There has been some changes in the circuit file since submission.But this report was not updated accordingly, though most of the logic are still the same.

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 (the term mantissa includes implicit significand bit). 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 9 bits and mantissa of 22 bits.

Bit Number	Portion	Number of bits
31	Sign (S)	1
30-22	Exponent	9
21-0	Fraction/Mantissa	22

Table 1: Bit Ranges of Floating-Point Number

Each component holds specific significance:

- **Sign bit:** Indicates the sign of the number, where 0 represents positive and 1 represents negative values.
- **Exponent:** Specifies the biased power of 2 by which the fraction is scaled.
- **Fraction:** Represents the fractional portion of the number, typically normalized to start with a leading 1.

The normalized significand lies within the range $1.0 \leq |\text{significand}| < 2.0$ and always includes an implicit leading 1 bit before the binary point. This "hidden bit" eliminates the need for explicit storage, as it is assumed to be present. The normalized significand corresponds to the fraction with the implicit "1." prepended.

The **bias**, a fixed constant, is used to represent the exponent as a signed integer, allowing for both positive and negative exponents. In this case, the bias value is 255.

1.2 Range

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 normalized positive number, the exponent is set to its maximum value of 254, and the mantissa is the maximum value for a 22-bit representation. The binary representation is $1.11111111111111111111 \times 2^{254}$.

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 $1.00000000000000000000 \times 2^{-254}$.

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.

1.3 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: 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 really 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.

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

$$0.\underbrace{00000000000000000001}_{\text{Fraction}} \times 2^{-254}$$

Which is equivalent to 1.0×2^{-276}. This is considerably smaller than the smallest positive normalized number, which is $1.00000000000000000000 \times 2^{-254}$. 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

Floating-point addition involves several key steps to ensure accurate computation:

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: Align Exponents To align exponents, the number with the smaller exponent is adjusted. Since B has a smaller exponent, its mantissa is shifted right, and its exponent is incremented to match A:

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

This requires an exponent comparator and a right shifter.

Step 2: Add Mantissas The mantissas are then added (considering the sign of B):

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

This step is executed using an Arithmetic Logic Unit (ALU).

Step 3: Normalize the Result If the result is not normalized, the mantissa is shifted, and the exponent is adjusted to restore a leading 1:

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

Normalization involves either right-shifting the mantissa while increasing the exponent or left-shifting while decreasing it.

Step 4: Check for Overflow/Underflow The result is checked for overflow or underflow, which may require special handling. In this case, neither occurs.

Step 5: Round the Result Rounding ensures the result adheres to the required precision. This is achieved using guard digits, round digits, and sticky bits, implemented via a rounding module.

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 Special cases such as denormalized numbers, infinity, and NaN are handled explicitly. In this example, no such cases arise.

Sign Determination: The result's sign is derived from the original numbers. After completing all steps, the final sum of A and B is:

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

This procedure ensures accurate floating-point addition by accounting for exponent alignment, mantissa addition, normalization, rounding, and 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

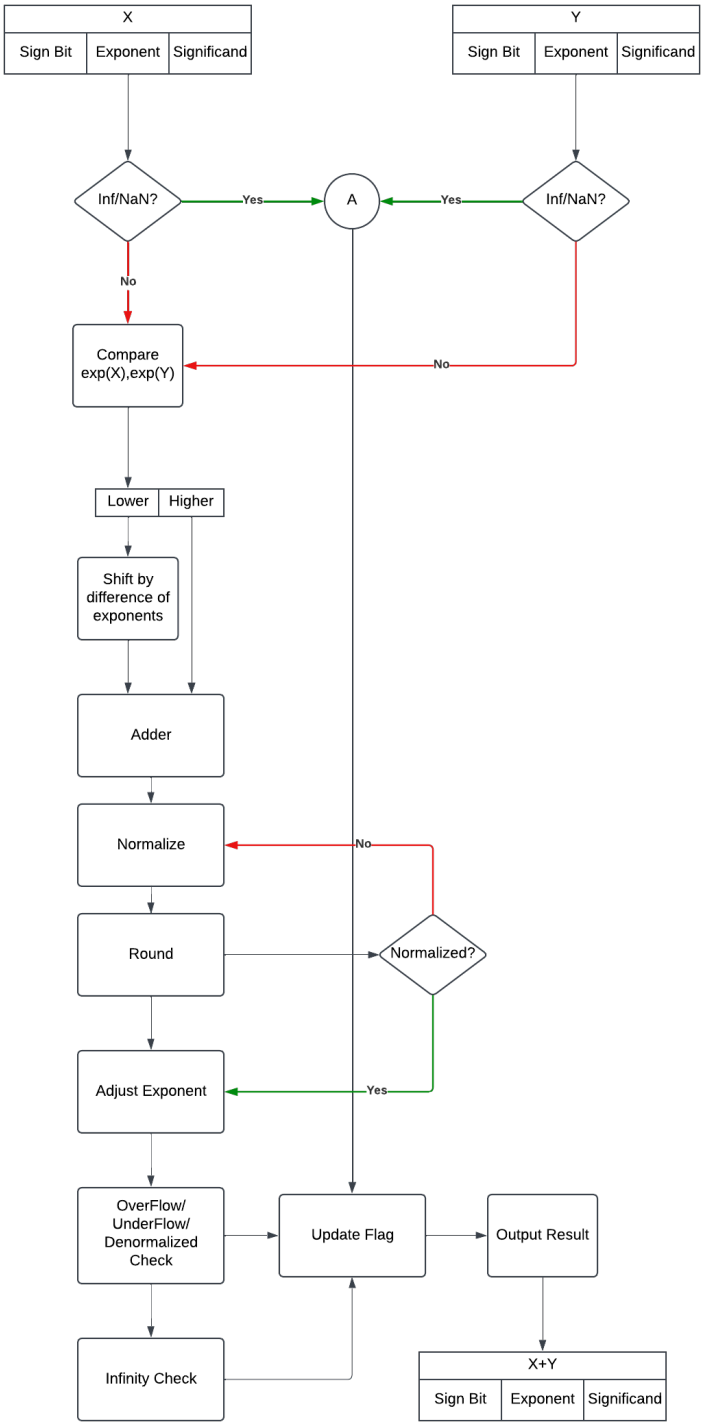


Figure 1: Flowchart of the addition/subtraction algorithm

4 High-level block diagram of the architecture

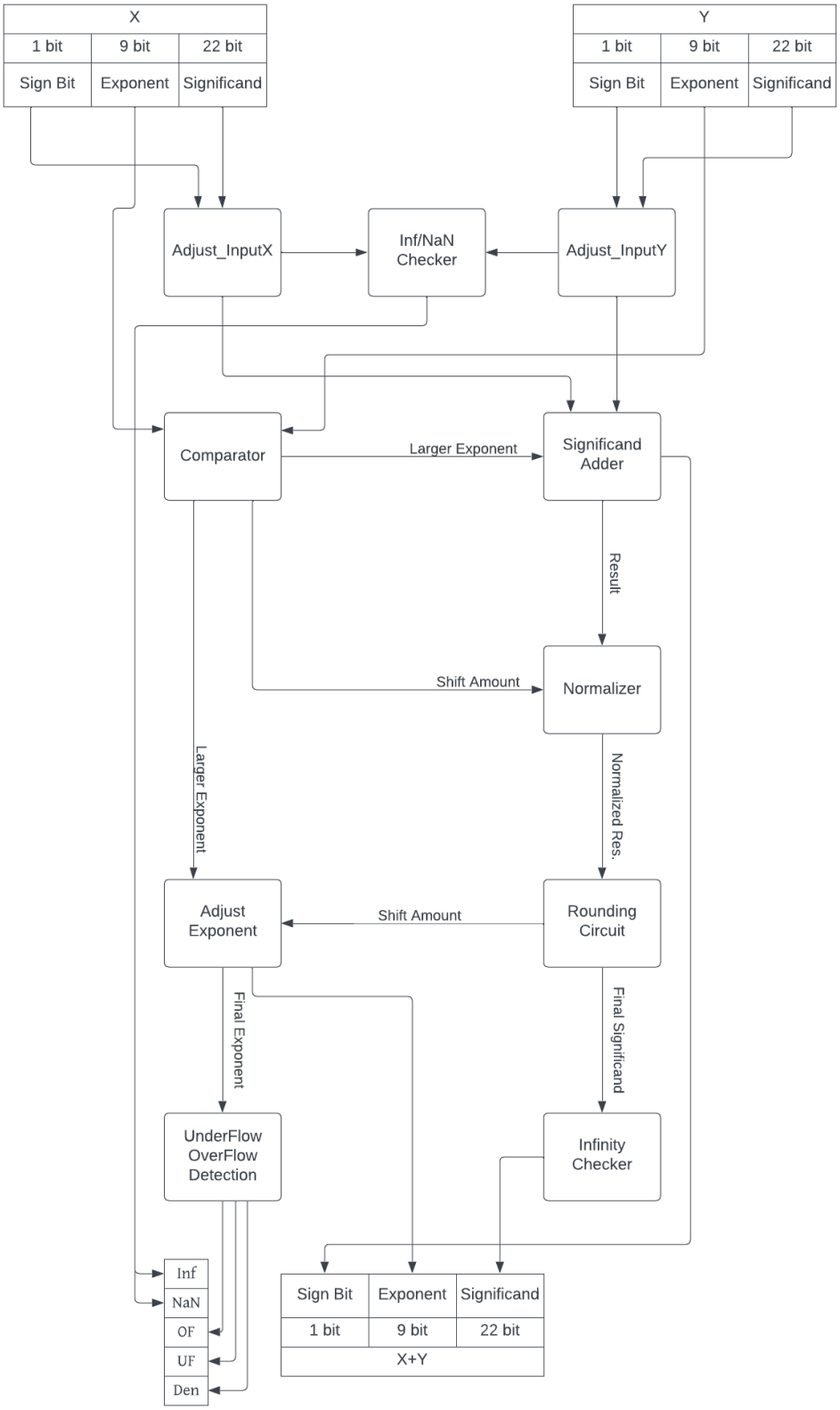


Figure 2: High-level block diagram of the architecture

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 Input Processors

The modular circuits in this library are as follows

- **Adjust Significand**: This module ensures that the input is correctly formatted, with a particular focus on handling negative values. When a negative input is detected, the module applies a 2's complement operation to convert it into a standardized format, making it ready for further processing. Also it handles the part for denormalization.
- **Adjust Exponent** : A component that computes the difference between the exponents of two input values and gives the final exponent as output.

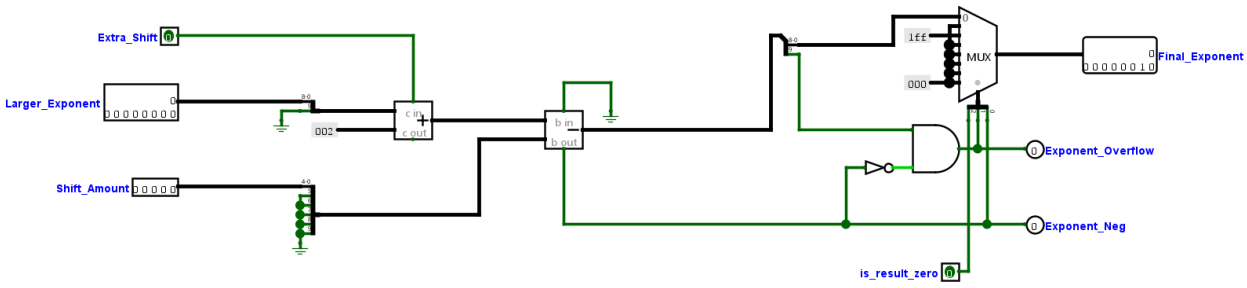


Figure 3: Adjust Exponent

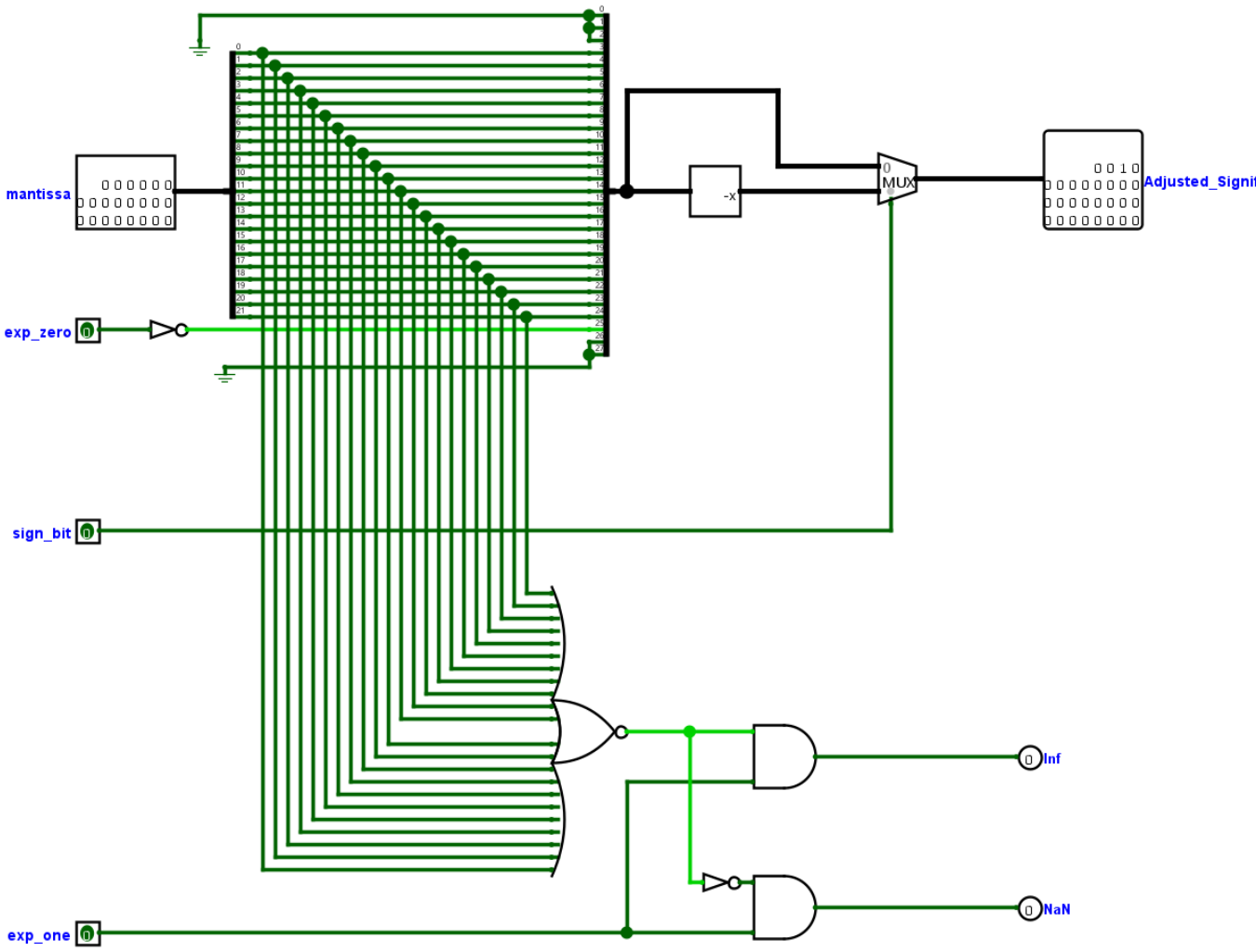


Figure 4: Adjust Significand

5.2 Comparator Circuit

The Comparator module is designed to compare two floating-point inputs by evaluating their exponents.Based on the outcomes of these comparisons, the module generates control signals that facilitate the subsequent stages of the addition process.

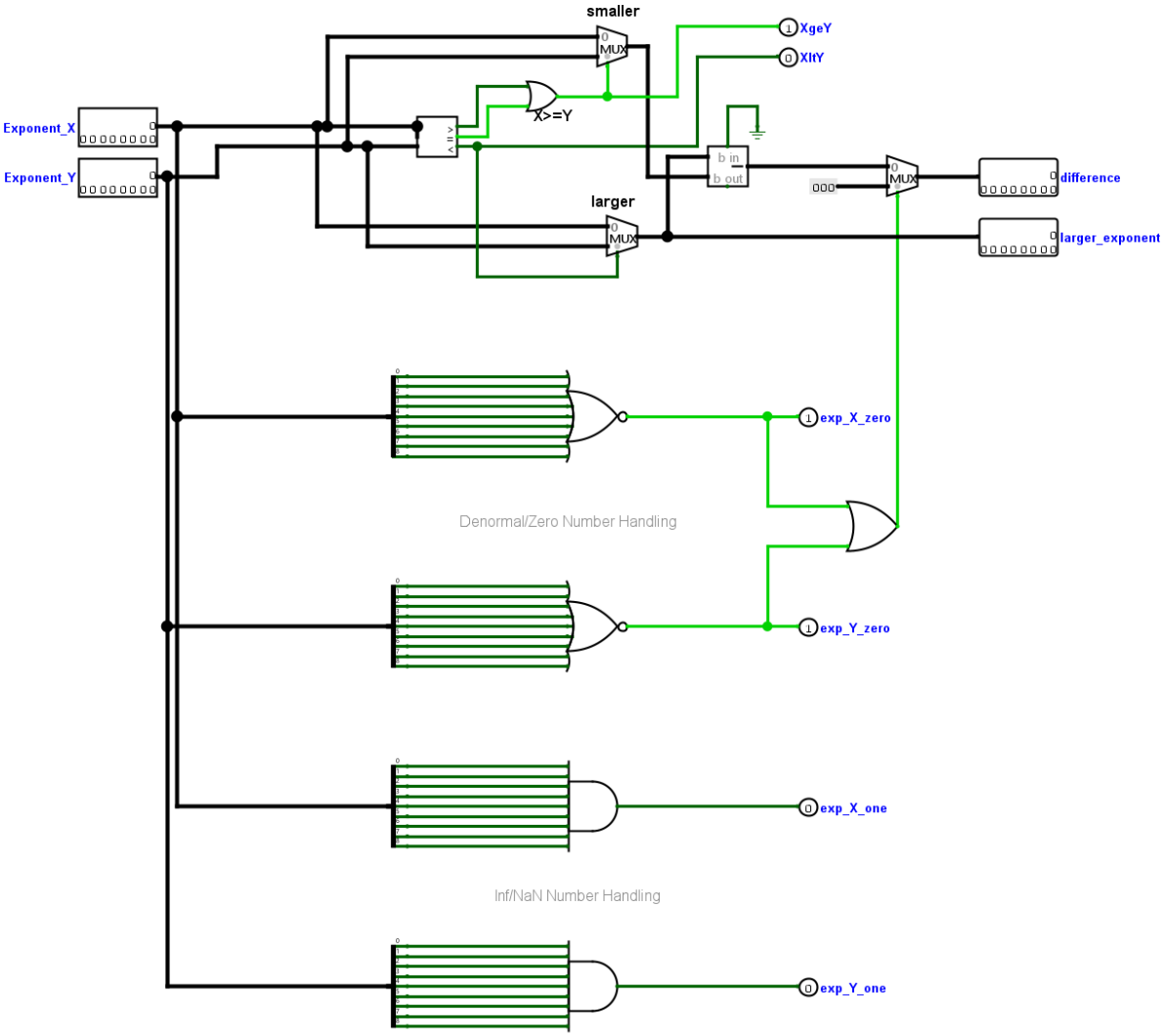


Figure 5: Comparator Cirtuit

5.3 Normalizer Circuit

The normalizer circuit adjusts the mantissa and exponent to prepare the number for use in other circuits. .It normalizes the Adjusted Significand , calculates the Shift Amount, and indicates if the mantissa is zero with the Fraction Zero bit. Also if handles the case of denormal plus denormal number.

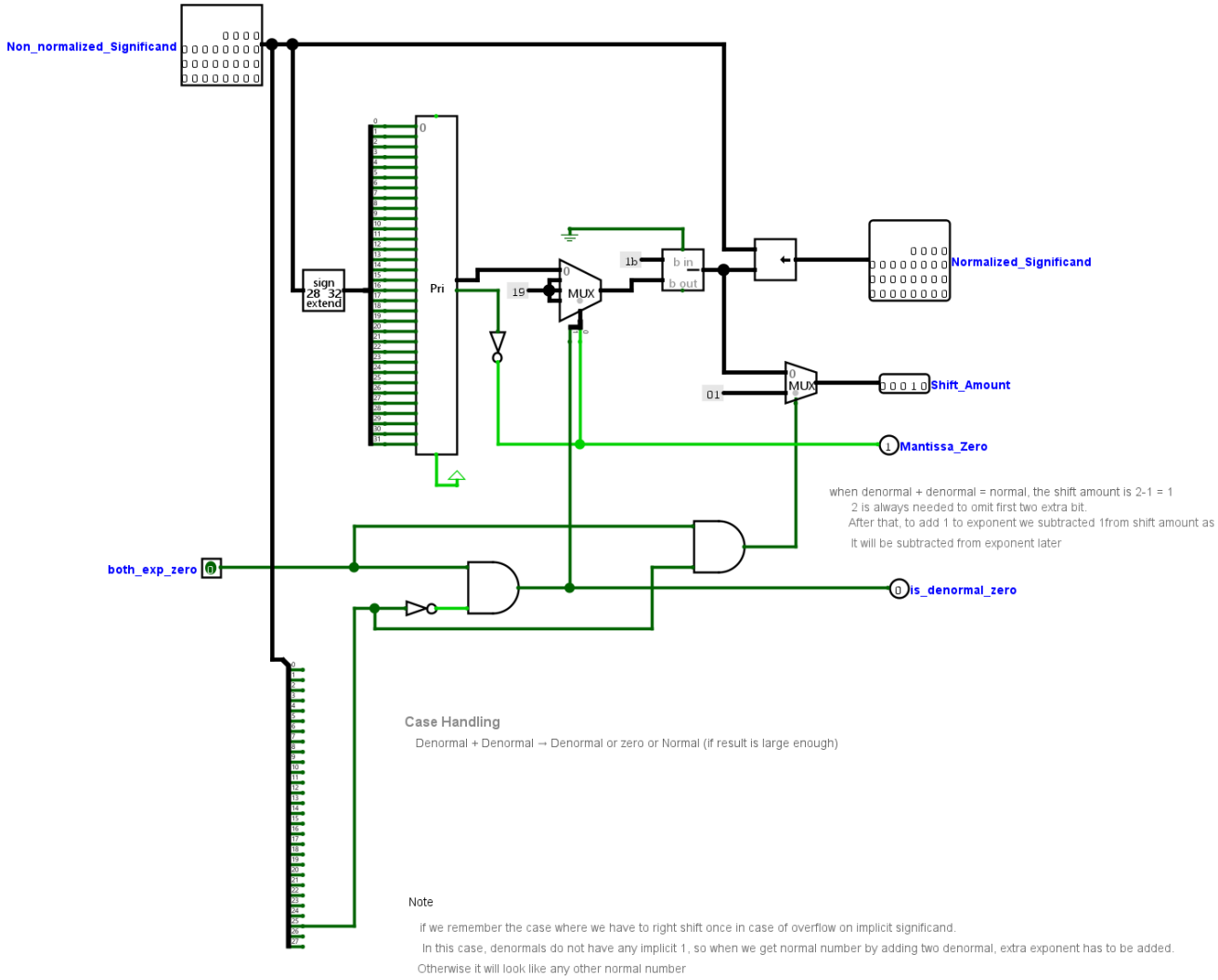


Figure 6: Normalizer Circuit

5.4 Rounder Circuit

The rounder circuit modifies the mantissa when rounding is needed for desired precosion It also handles the denormal number as rounding is not needed for those.

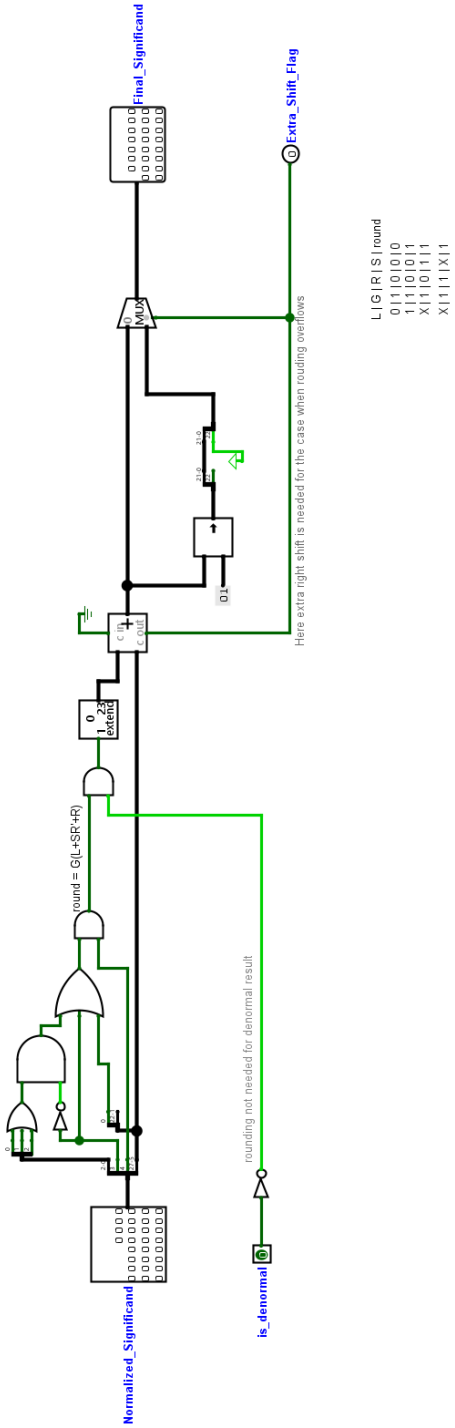
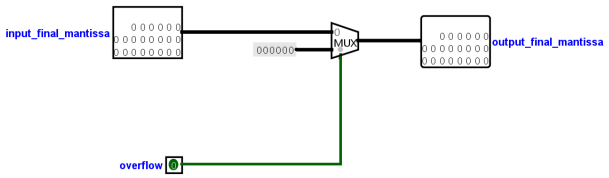


Figure 7: Rounder

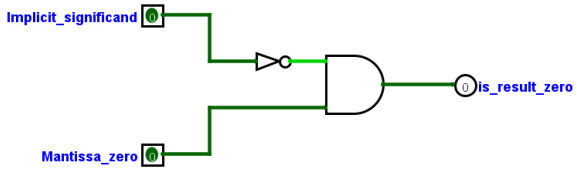
5.5 Checkers

The modular circuits in this library are as follows

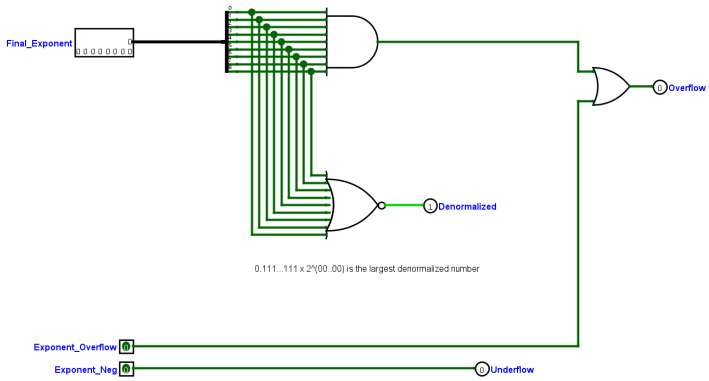
- **Infinity Checker:** Checks whether the output is infinity
- **Zero Checker:** Checks weather the result is zero
- **Underflow-Overflow Checker:** Checks underflow, overflow of the output



(a) Infinity Checker



(b) Zero Checker



(c) Underflow-Overflow Checker

Figure 8: Necessary Checkers

5.6 Floating Point Adder

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

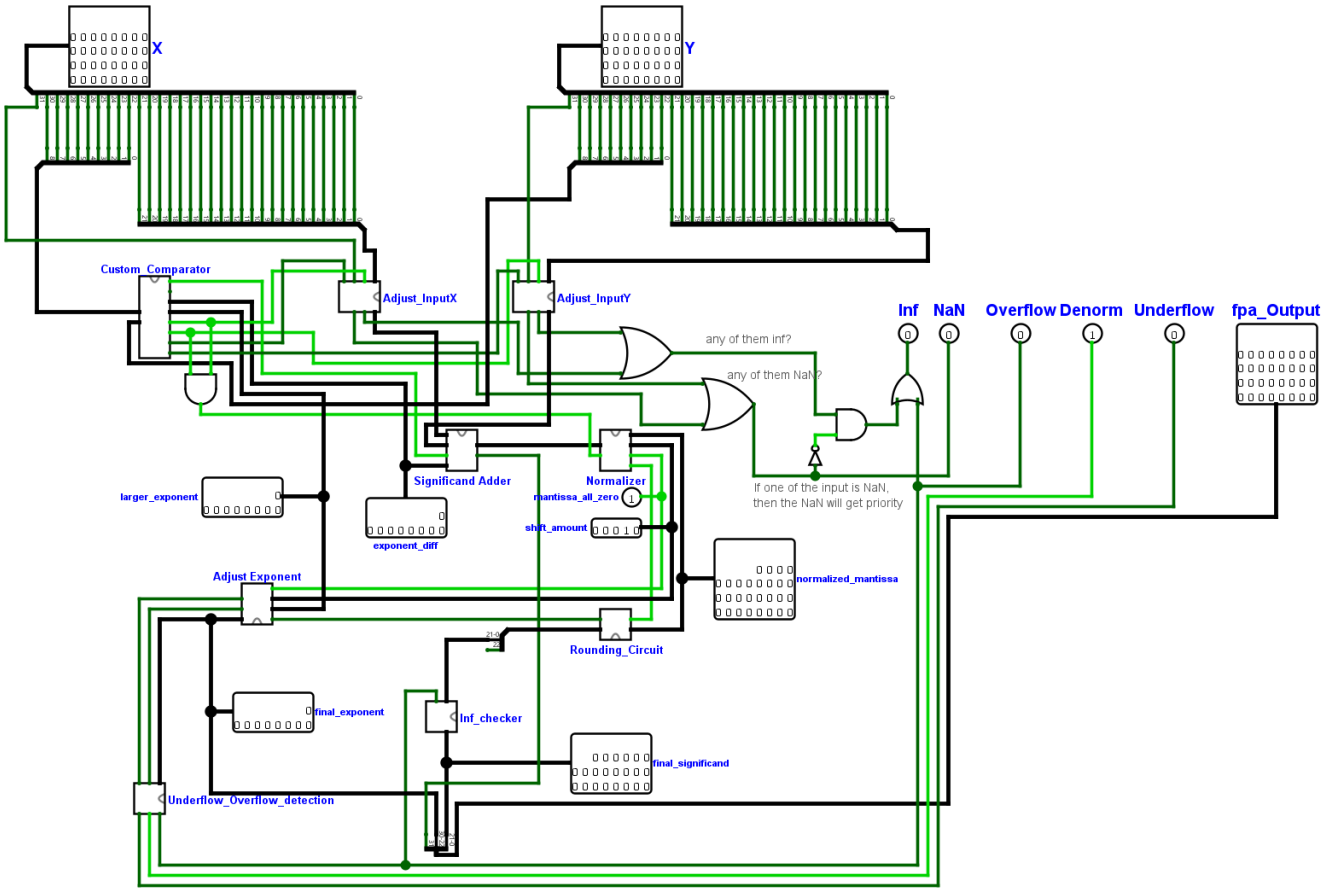


Figure 9: Floating Point Adder

5.7 Main Circuit

This is the main circuit with output and necessary flags

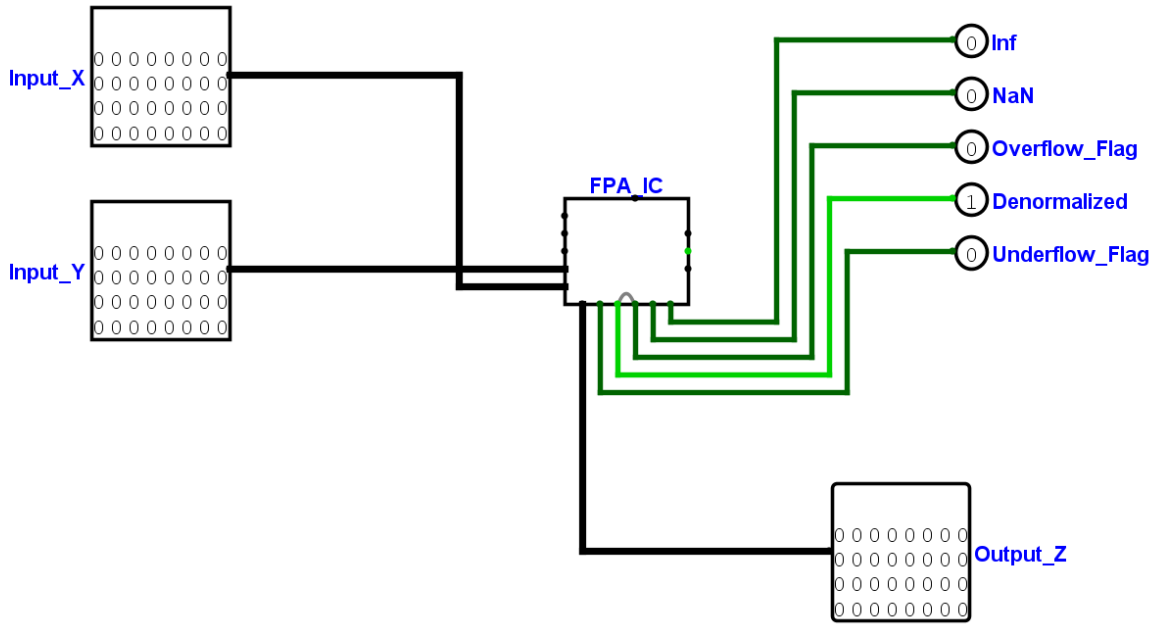


Figure 10: Main Circuit

6 Comprehensive Design Description

6.1 Calculating exponents and adjusting mantissa for addition

To perform floating-point addition, we are splitting 32 bit input into 3 parts:

1. **Sign Bit:** The first bit of our input indicates whether the input is positive or negative.
2. **Exponent:** Then 9 bit represents our exponent.
3. **Mantissa:** The last 22 bits represent the fraction part

First, 22 bit significand is adjusted into 32 bit by adding rounding bits, implicit one and zeros.

Using our comparator, we find out the larger exponent and then then we shift the fraction with smaller exponent. Addition is followed by this.

6.2 Normalization

Normalization involves identifying the position of the first set bit using a priority encoder. The shift amount is calculated by subtracting this position from 32. The significand is shifted to the left by the computed shift amount, and the exponent is decremented accordingly. If a carry is generated during addition, the exponent is incremented by one, and the significand is shifted one position to the left to maintain proper normalization.

6.3 Rounding

Rounding ensures that the final result fits within the precision constraints of 22 bits. The 23rd bit serves as the guard bit, and the 24th bit is the round bit. A sticky bit is set if any bits beyond the round bit are non-zero. The rounding decision depends on the following cases:

1. If the guard bit is 0, truncation occurs.
2. If the guard bit is 1 and either the round bit or sticky bit is 1, rounding up is performed.
3. For round-to-even behavior, rounding is performed only if the 22nd bit is 1 when the lower bits indicate rounding is required.

Table 4: Rounding Actions Based on Guard, Round, and Sticky Bits

L	G (Guard)	R (Round)	S (Sticky)	Round
0	1	0	0	0
1	1	0	0	1
×	1	0	1	1
×	1	1	×	1

A Karnaugh map simplifies the rounding logic, resulting in the equation:

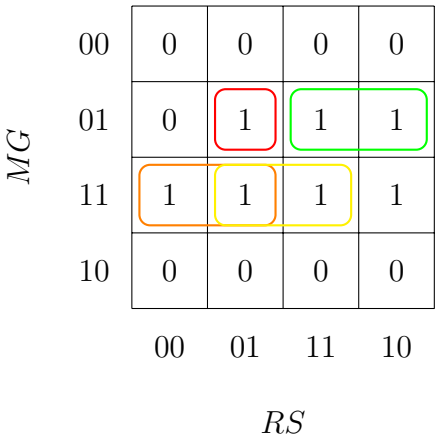


Figure 11: K-map for flag evaluation

$$\text{flag} = GS + GR + MG = G(M + R + S)$$

If the flag is set, 1 is added to the 22nd bit. Otherwise, bits beyond the 23rd position are truncated.

6.4 Flag Checking

The Flag Checking process determines the final status of the floating-point result based on specific conditions such as underflow, overflow, and exponent or significand abnormalities. Flags indicate scenarios like denormalized values, infinity (INF), not-a-number (NaN), underflow, or overflow. This mechanism ensures accurate classification and handling of edge cases in floating-point operations.

Table 5: Flag Checking

Result = 0?	Underflow	Overflow	Significand	Exponent	Flag
0	×	×	0	0	Zero
×	1	×	Not Zero	0	Denormalized
×	×	×	0	511	INF
×	×	×	Not Zero	511	NaN
×	1	×	Not Zero	0 to 510	Underflow
×	×	1	Not Zero	0 to 510	Overflow

7 ICs Used with Count as a Chart

IC	Quantity
IC 7402	17
IC 7404	3
IC 7408	10
IC 7432	4
IC 7483	9
IC 74151	1
IC 74153	1
IC 74157	4
Shifter (Left)	1
Shifter (Right)	1
Priority Encoder	1
Comparator	1
Total	53

Table 6: ICs Used with Quantity

8 The Simulator Used along with the Version Number

Logisim-evolution-3.9.0 has been used for simulaing the floating poitn adder circuit.

9 Contribution

Roll No	Contribution
2105161	Simulation,Report
2105163	Design,Simulation,Report
2105168	Simulation, Report
2105174	Simulation, Report

Table 7: Contribution of Each Member

9 Discussion

This assignment delved into the fundamental concept of *Floating Point Addition*, adhering to the IEEE 754 standard, a critical formatting standard for modern computing systems. Through this project, we gained valuable insights into designing circuits that align with this widely adopted standard.

The initial phase of the assignment involved creating helper libraries to house circuits of similar types with varying bit capacities.We used the default adder as adder , subtractor and negator.

The next phase focused on building the core components of the Floating Point Adder, including the Adder Subtractor, Adjust input , Comparator , Normalizer, and Rounder circuits.

Handling denormal numbers proved to be one of the most challenging aspects of the design. These special cases required us to update the design 7 times, each revision followed by exhaustive testing against all possible edge cases to ensure correctness. This iterative approach was essential to achieving a robust and reliable design.

Designing the final circuit required integrating the pre-built components while carefully managing control flow and ensuring all components worked seamlessly. At various stages of the design, we had to make critical decisions, balancing simplicity, efficiency, and resource usage. Overflow, Underflow, and Error checker flags were incorporated to handle such edge cases.

Extensive manual testing was carried out after each design update to validate functionality and ensure the circuit met the IEEE 754 standard. This rigorous testing process and careful design refinement have given us confidence in delivering a fully optimized and functional Floating Point Addition circuit.