CSE 210
Computer Architecture Sessional
Floating Point Adder

Section - A1

Group - 02

Group Members

- 2105007 - Nafis Nahian
- 2105008 - Sk. Ashrafuzzaman Nafees
- 2105009 - Abrar Zahin Raihan

1 Introduction

A floating-point number is a computer representation of a real number with a specific number of digits before and after the decimal point (or radix point in a broader context). A floating-point adder is a digital circuit designed to add and subtract floating-point numbers. It enables representation of values that are too large or small for integer formats.

Floating-point numbers have three primary fields: the sign bit, the exponent, and the mantissa. The sign bit indicates the sign, either positive or negative. The exponent field allows for a wide range of values by scaling the significand with a power of the base in a biased form. To find the true exponent, we subtract a bias from the stored exponent bits. The mantissa, representing the fractional part, contains the digits following the decimal point. In this implementation, the sign bit, exponent, and mantissa use 1, 9, and 22 bits, respectively. The exponent bias in this case is $2^{9-1} - 1 = 255$.

To add two numbers, a floating-point adder aligns their decimal points, then sums their mantissas. After required shifts and modifications (increment/decrement), the result's exponent is adjusted. Finally, normalization and rounding are performed. The outcome's sign depends on the signs of the two numbers.

Floating-point adders are widely used in fields like financial modeling, computer graphics, and scientific calculations, where fast, hardware-accelerated computations are essential. Co-processors often handle these demanding tasks, enabling rapid floating-point arithmetic. This is especially important in data analysis and scientific simulations that require high precision.

1.1 Floating Point Representation

According to the IEEE 754 standard, a floating-point number is represented by three main components: a sign bit, an exponent, and a fraction (also referred to as the mantissa or significand). The general structure of this format is defined by the equation:

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

In this assignment, we use an exponent of 9 bits and a mantissa of 22 bits.

Bit Number	Portion
31	Sign (S)
$30 - 22$	Exponent
$21 - 0$	Fraction/Mantissa

Table 1: Bit Ranges of Floating-Point Number

Each component has a specific role:

- **Sign bit**: Indicates the number's sign, with 0 for positive and 1 for negative.
- **Exponent**: Represents the power of 2 by which the fraction is multiplied.
- **Fraction**: The fractional part of the number, typically normalized to start with a leading 1.

The normalized significand, which lies in the range $1.0 \leq |\text{significand}| < 2.0$, includes an implicit leading 1 before the binary point. This leading bit, known as the hidden bit, is added automatically to the fraction. The normalized significand is essentially the fraction with "1." restored as its prefix.

A constant called the bias is added to the exponent to represent it as a signed integer, allowing both positive and negative exponents. Here, the bias is 255.

1.2 Range

- Extremely small numbers, close to zero, appear with a reduced exponent and a fraction that includes leading zeros.
- Extremely large numbers are represented with an increased exponent.

In a floating-point adder with a 22-bit mantissa and an 9-bit exponent, the largest and smallest positive normalized numbers are defined by the range of exponents and the precision of the mantissa.

For the largest positive number, the exponent reaches its maximum value of 255, and the mantissa is set to its maximum for a 22-bit format. This binary representation is 1.111111111111 1111111111 $\times 2^{255}$. In approximate decimal terms, this corresponds to a value around 1.158×10^{77}.

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

These values define the range of positive normalized numbers that can be accurately represented by a floating-point adder with a 22-bit mantissa and an 9-bit exponent. It's essential to recognize that these are approximate values, as finite binary formats impose limitations on the exact representation of real numbers.

1.3 IEEE 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	Non-Zero	NaN (Not a Number)

Table 1: IEEE 754 Encoding

1.4 Denormalized Number

In IEEE 754 floating-point representation, denormalized numbers are a special category for values smaller than normal floating-point numbers. They enable a gradual underflow, where numbers can approach zero with decreasing precision. Denormalized numbers have an exponent of all zeros and a hidden bit set to zero. The formula for representing 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 (0 for positive, 1 for negative),
- **Fraction** represents the fractional part of the number (without the hidden bit, which is 0 for denormalized numbers),

- $\text{exponent}_{\text{min}}$ is the smallest representable exponent.

Denormalized numbers have a more limited exponent range than normalized numbers, allowing for a smooth transition to zero as values decrease, thereby ensuring a gradual loss of precision.

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

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

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

1.5 Floating Point Number Addition

Performing floating-point addition involves multiple 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 First, 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 shift its mantissa to the right and increase its exponent by 1:

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

An exponent comparator and right shifter are used for this alignment.

Step 2: Adding Mantissas Next, add the mantissas:

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

This step is carried out using the Arithmetic Logic Unit (ALU).

Step 3: Normalizing the Result If the result is not in normalized form, we normalize it by adjusting the exponent and shifting the mantissa to obtain a leading 1:

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

Normalization involves shifting right and incrementing the exponent, or shifting left and decrementing the exponent as necessary.

Step 4: Checking for Overflow or Underflow Verify if the result has overflowed or underflowed. If either condition is met, it indicates an exception. In this example, there is no overflow or underflow.

Step 5: Rounding the Result Adjust the result based on the required precision. A rounding circuit can be designed with guard, round, and sticky bits for this purpose.

Step 6: Normalizing the Rounded Result If the rounded result is not in normalized form, normalize it by adjusting the exponent and shifting the mantissa as needed to include a leading 1.

Step 7: Handling Special Cases In floating-point representation, certain special cases must be handled, including denormalized numbers, infinity, and NaN. This example does not involve any of these cases.

The sign of the result is determined by the signs of the original numbers. After completing exponent alignment, mantissa addition, normalization, rounding, and handling special cases, the sum of A and B is approximately:

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

This procedure ensures the correct addition of floating-point numbers by managing exponent alignment, mantissa addition, normalization, rounding, and handling of special cases.

2 Problem Specification

We designed a floating point adder circuit taking two floating point numbers as inputs. Each floating point number is 32 bits with the following representation:

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

Table 2: Problem Specification

In our assignment, we have an exponent of 9 bits and a mantissa of 22 bits.

Bit Number	**Portion**
31	Sign (S)
30-22	Exponent
21-0	Fraction/Mantissa

Table 3: Bit Ranges of Floating-Point Number

3 Flowchart of the Addition/Subtraction Algorithm

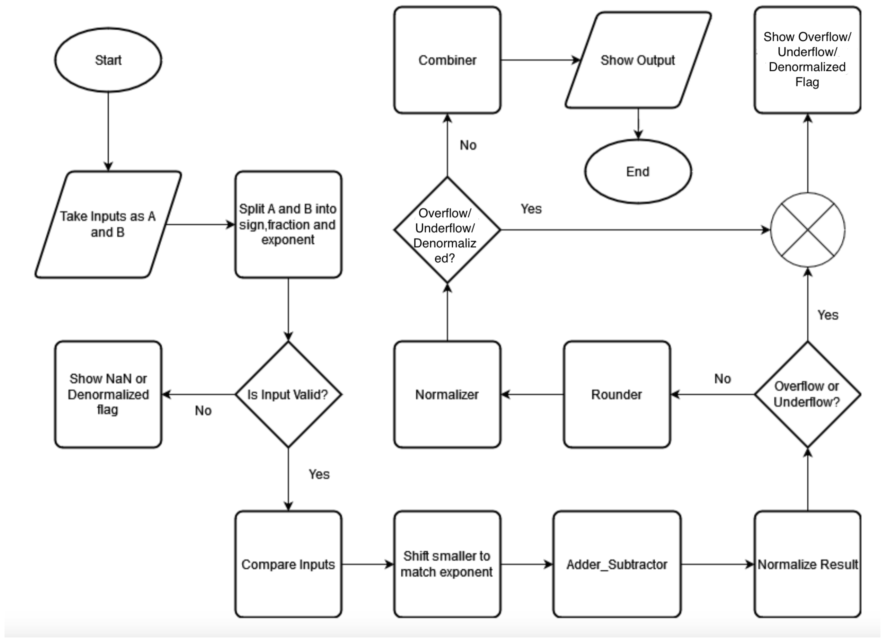

Start
Take Inputs as A and B
Split A and B into sign,fraction and exponent
Is Input Valid?
No
Show NaN or Denormalized flag
Yes
Compare Inputs
Shift smaller to match exponent
Adder_Subtractor
Normalize Result
Overflow or Underflow?
No
Rounder
Normalizer
Yes
Overflow/ Underflow/ Denormalized?
Yes
No
Combiner
Show Output
End
Show Overflow/ Underflow/ Denormalized Flag

Figure 1: Flowchart of the addition/subtraction algorithm

4 High-level Block Diagram of the Architecture

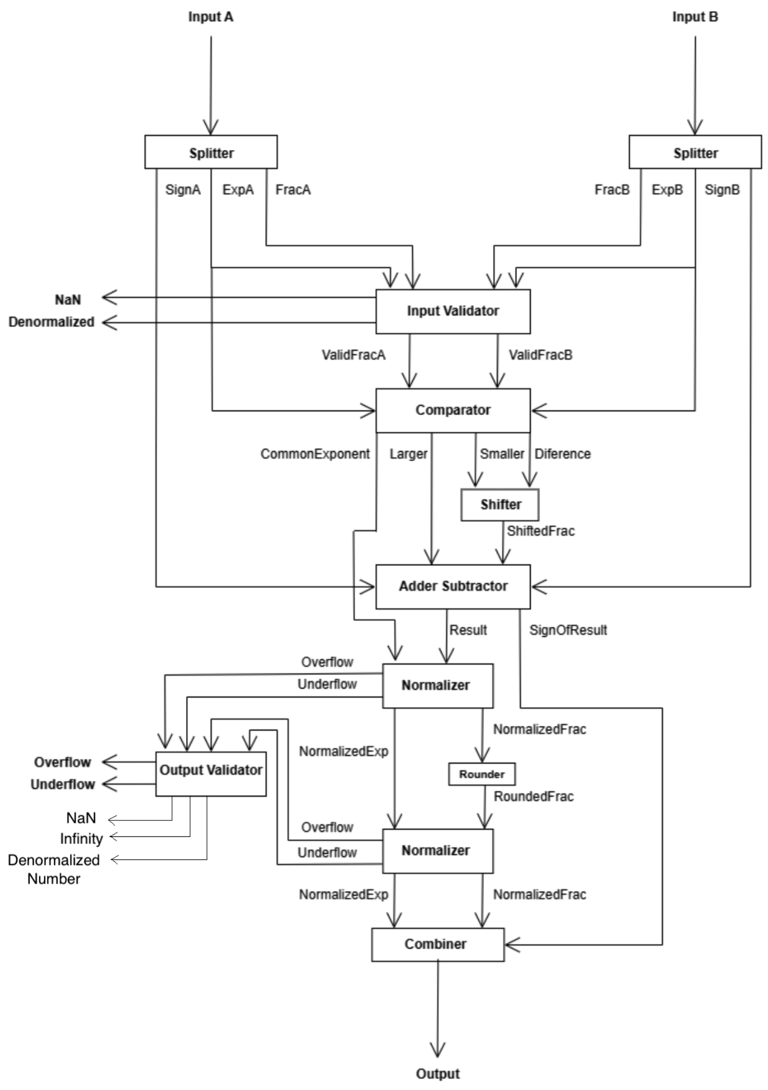


Figure 2: High-level block diagram of the architecture

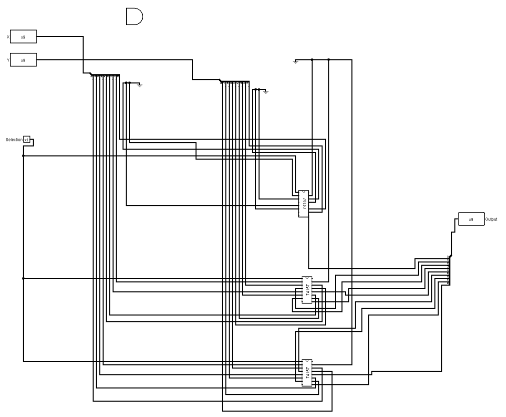
5 Detailed Circuit Diagram of Important Blocks

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

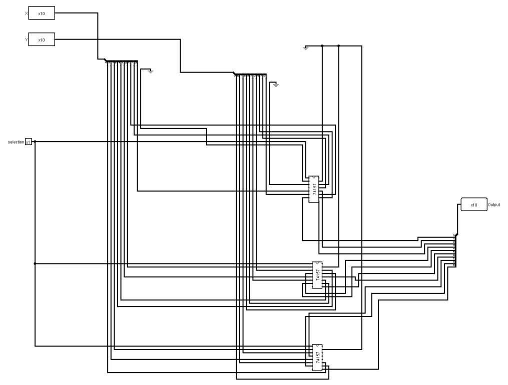
5.1 Multiplexer Circuits

The modular circuits in this library are as follows

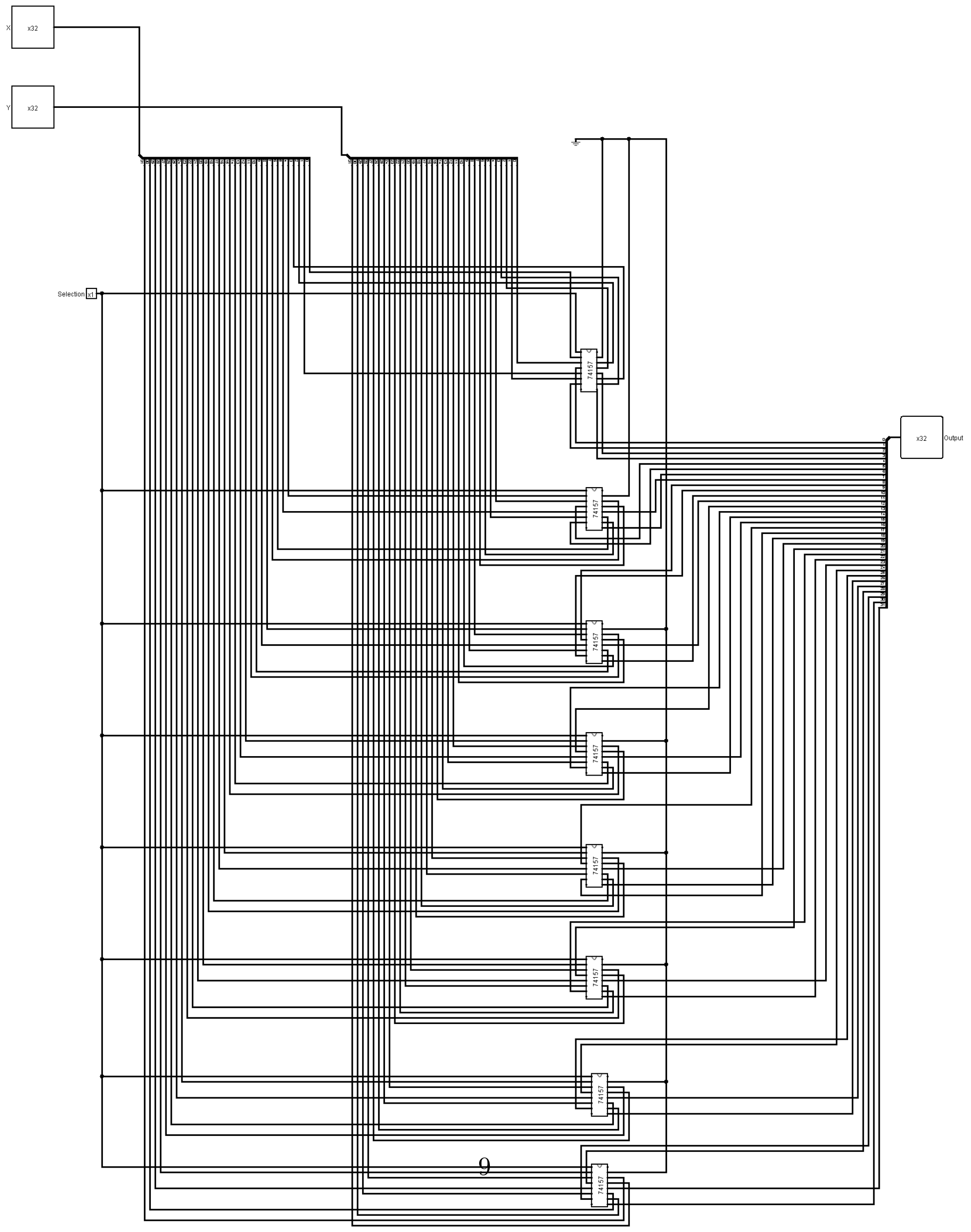
- 9 bit 2 to 1 MUX
- 10 bit 2 to 1 MUX
- 32 bit 2 to 1 MUX



(a) 9 bit 2×1 MUX



(b) 10 bit 2×1 MUX

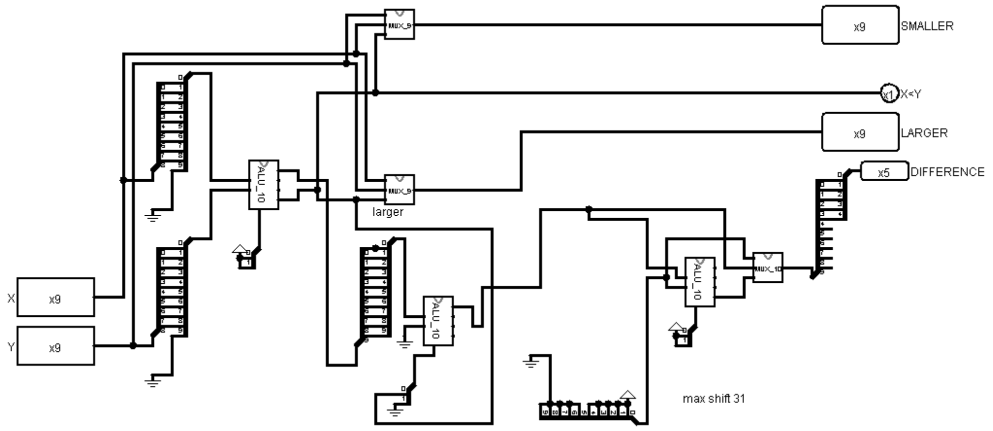


(c) 32 bit 2×1 MUX

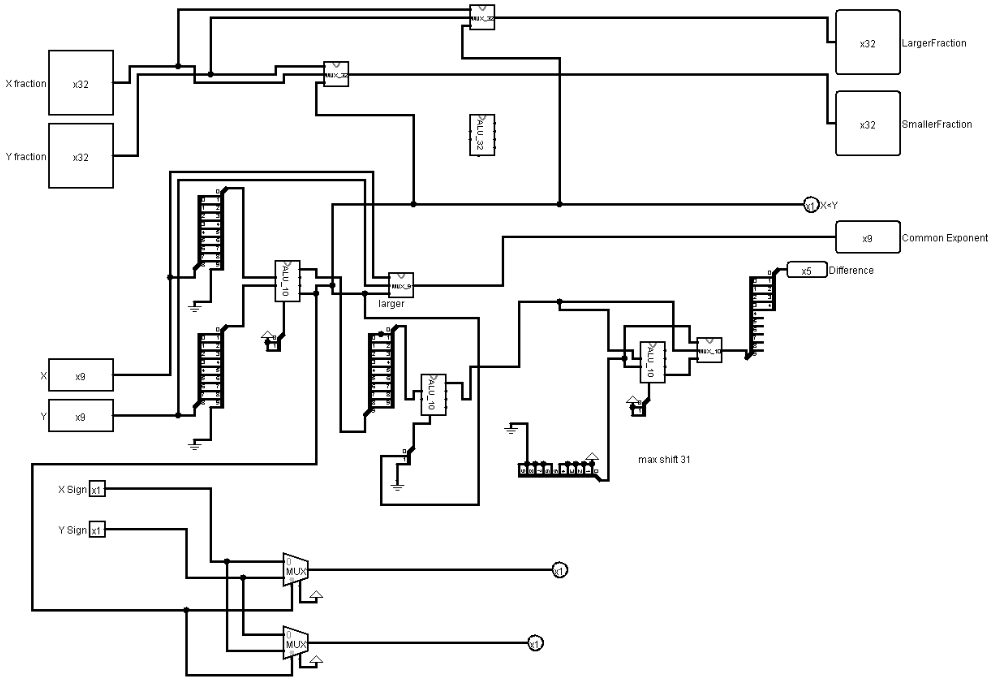
Figure 3: Multiplexer Circuits

5.2 Comparator Circuits

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



(a) 9 bit magnitude comparator



(b) Smaller Fractions Magnitude Comparator

Figure 4: Magnitude Comparator

5.3 Adder-Subtractor Circuit

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

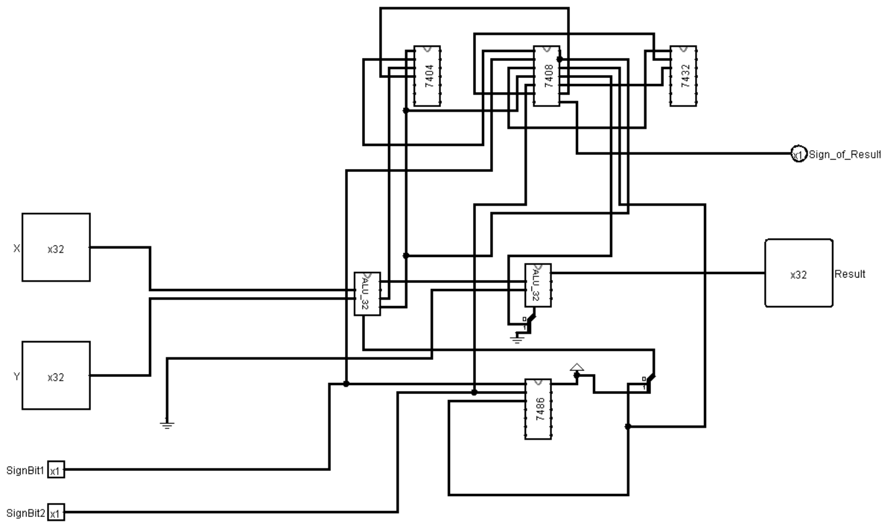
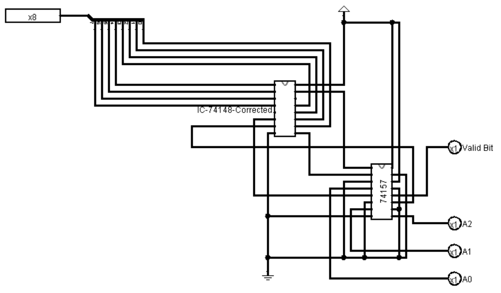


Figure 5: 32 bit Adder-Subtractor

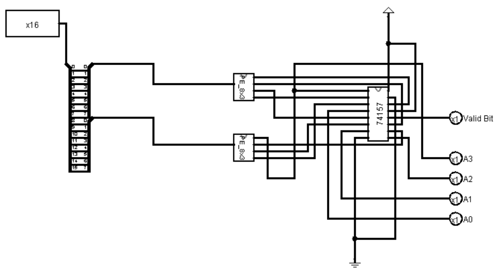
5.4 Encoder Circuits

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

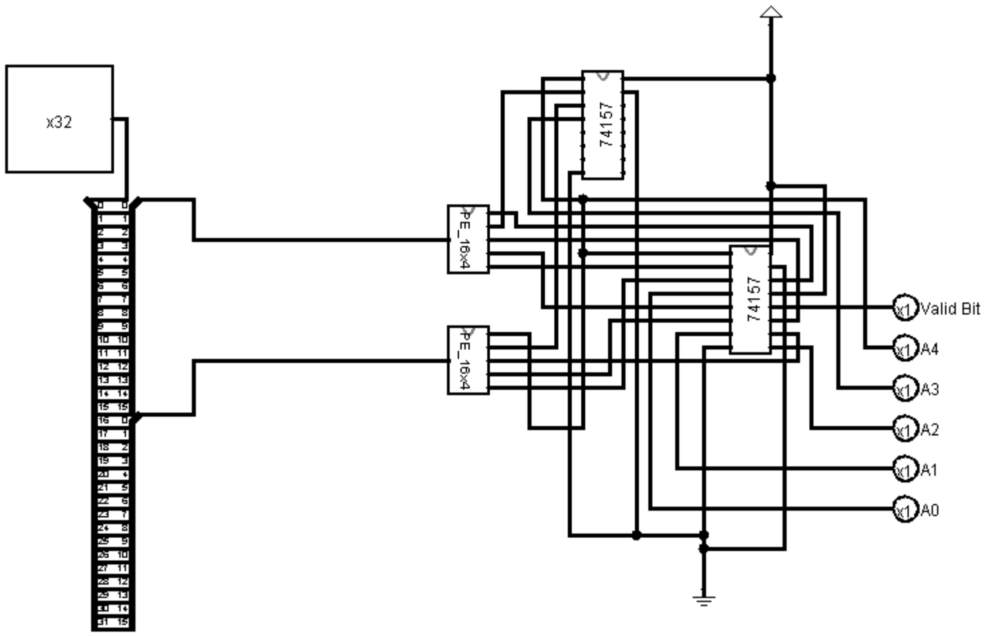
- 8 to 3 priority encoder
- 16 to 4 priority encoder
- 32 to 5 priority encoder



(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.

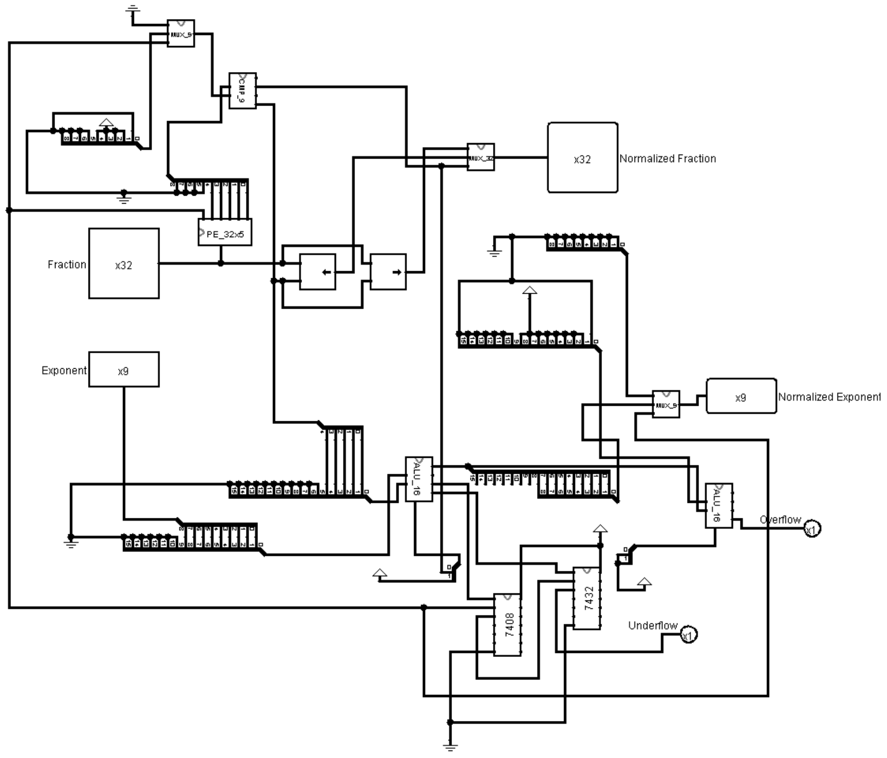


Figure 7: Normalizer Circuit

5.6 Rounder Circuit

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

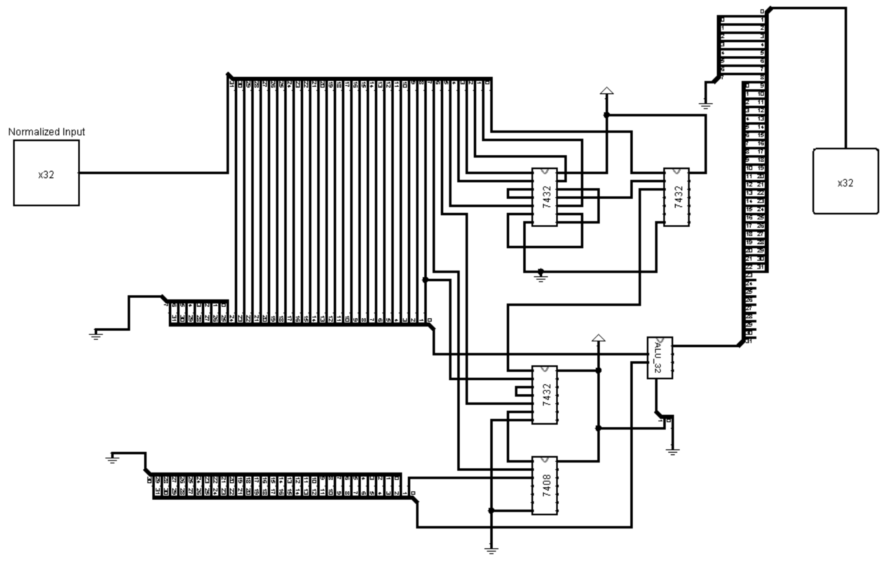
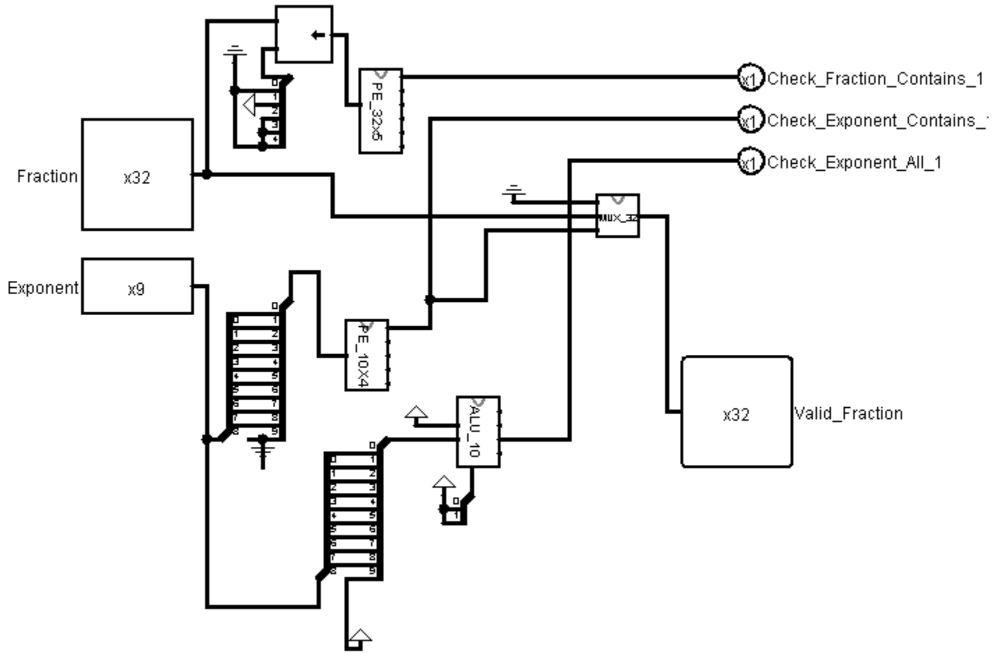


Figure 8: Rounder Circuit

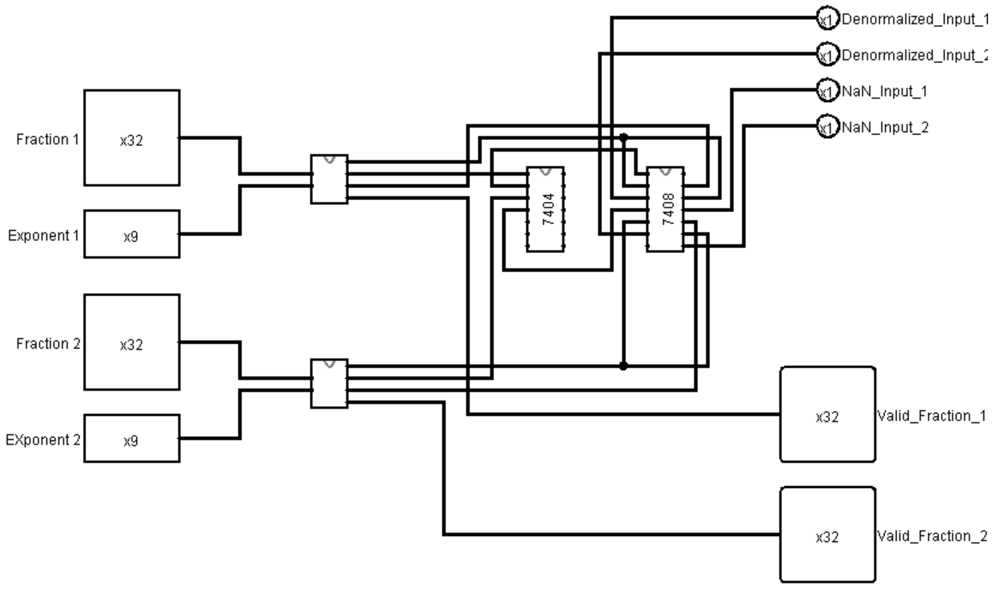
5.7 Input Pre-processor Circuits

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

- A Single Input Checker that shows the validity of a single floating point number input
- An 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, 10 and 16 bit ALU's were also used in the design. All of them are of identical construct.

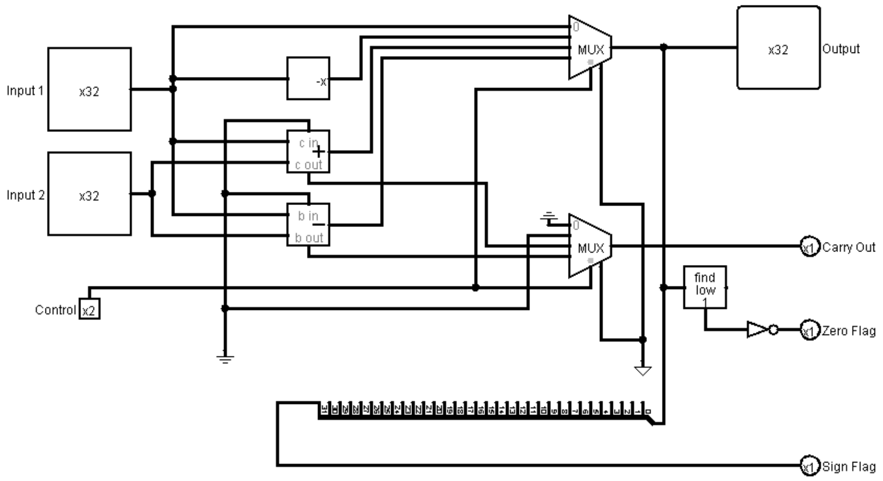


Figure 10: 32 bit ALU

5.9 Infinity Output Checker

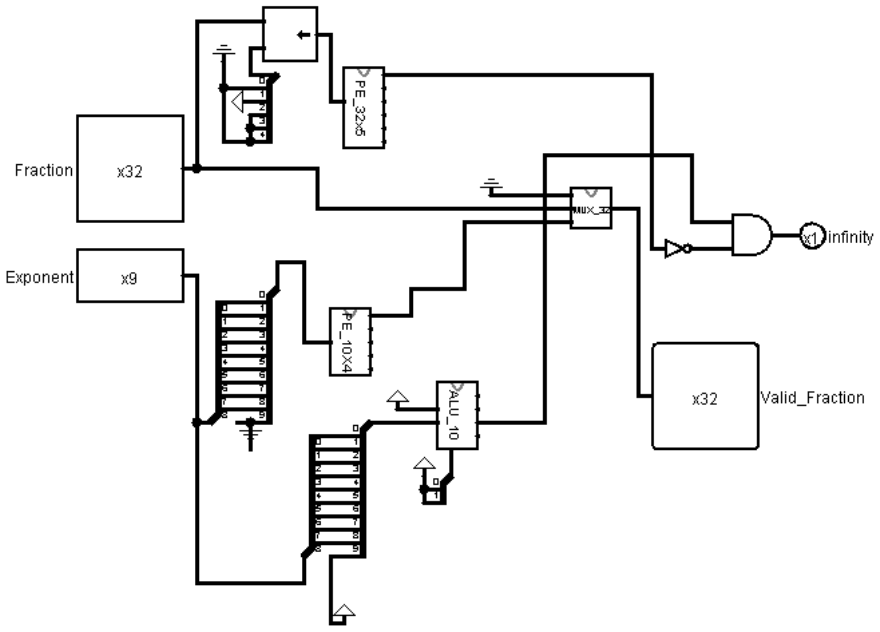


Figure 11: Infinity Output Checker

5.10 Denormalized Output Checker

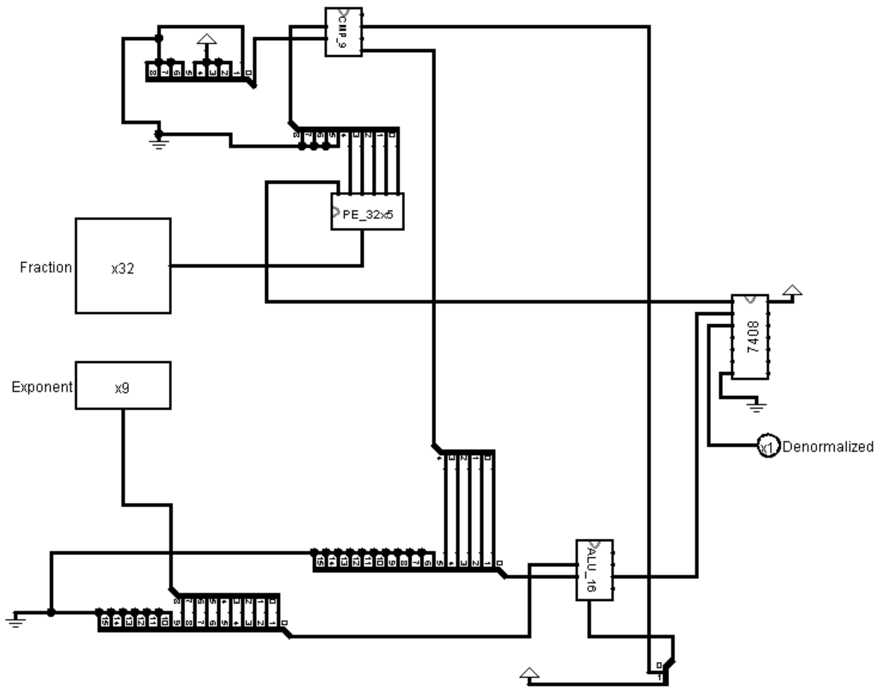


Figure 12: Denormalized Output Checker

5.11 Floating Point Adder

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

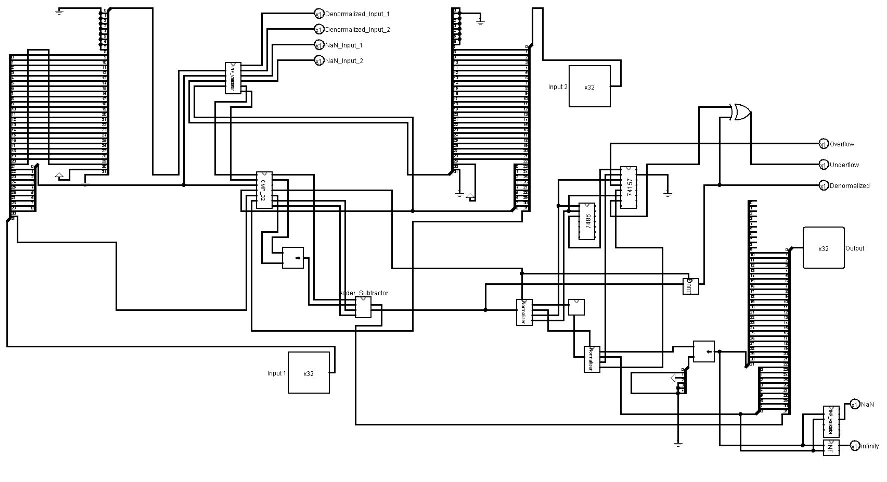


Figure 13: Floating Point Adder

6 ICs Used with Count

IC	**Quantity**
IC 7404	3
IC 7408	6
IC 7432	7
IC 7486	3
IC 74148	20
IC 74157	123
ALU (10 bit)	11
ALU (16 bit)	4
ALU (32 bit)	3
Shifter (Left)	5
Shifter (Right)	3
Total	**188**

Table 4: ICs Used with Quantity

7 Simulator Used

Logisim - 2.7.1

8 Contributions

- **Comparator**: 2105008, 2105009
- **Adder-subtractor, Encoder, MUX, ALU**: 2105007, 2105008, 2105009
- **Normalizing**: 2105008
- **Rounding**: 2105007, 2105008
- **Input Processing**: 2105007
- **Output Processing**: 2105008
- **Report**: 2105009

9 Discussion

This assignment introduces a fundamental concept in Computer Arithmetic, known as *Floating Point Addition*, in accordance with the IEEE 754 standard—a widely adopted floating-point representation in modern computer systems.

The process begins with designing auxiliary libraries containing circuits of similar types and varying bit capacities. For Encoder and Multiplexer libraries, we employed cascading techniques to achieve a comprehensible and maintainable design. The ALU library, which comprises three distinct circuits, was constructed using the default *Adder* tools provided by the simulation software, in line with the given specifications. The Comparator library relies on the

ALU library for subtraction operations and subsequently processes the ALU output to provide comparison data.

The core components of the Floating Point Adder include the Adder Subtractor, Input Validator, Normalizer, and Rounder, each built on top of the auxiliary libraries. During design, we focused on minimizing the number of ICs by carefully considering various design choices. For instance, we initially overlooked the transfer operation within the ALU, which, although seemingly trivial, resulted in additional multiplexers. Recognizing this inefficiency, we incorporated the transfer operation in the ALU. The input validator utilizes encoders and the ALU to perform an error pre-check, identifying IEEE-specified ranges. Accordingly, it flags outputs as either *Denormalized Number*, *NaN*, or valid. In the Rounder, to account for appended bits '01' in front, we treated 22 bits as the mantissa, with the last two designated as *Guard* and *Round* bits, followed by additional Sticky bits.

The final circuit design involved merging pre-built components while carefully managing control flow. Normalization was repeated post-rounding, as in certain cases, the Normalizer output could be denormalized after passing through the Rounder. Overflow, underflow, and error checker flags were displayed as outputs in the final circuit. Since we prefixed the mantissa with '01' bits to simulate the hidden set bit, we used a left shifter to disregard these bits in the output.