CSE 210

Computer Architecture Sessional

Assignment: 32-bit Floating Point Adder

Section - A2

Group - 09

Group Members:

i. 2105056 - Shah Mohammad Abdul Mannan

ii. 2105052 - Md. Mehedi Hasan

iii. 2105050 - Dipit Saha

1 Introduction

Floating point numbers provide a mechanism for computers to represent real numbers through a format that maximizes the representable value range with a finite number of bits. Unlike fixed-point representations, which allocate a fixed number of digits before and after the decimal point, floating point representations utilize a format akin to scientific notation. This enables the decimal point to move, or "float," allowing for the efficient representation of very large or very small numbers. A floating point adder is a specialized digital circuit designed to perform arithmetic operations—particularly addition and subtraction—on floating point numbers, which are crucial in high-precision and wide-range numerical applications.

A floating point number consists of three key components:

- **Sign bit:** Determines the number's polarity (positive or negative).
- **Exponent:** Stored in a biased format to scale the mantissa by a power of the base, typically two. The actual exponent value is calculated by subtracting a bias from the stored value.
- **Mantissa (or Significand):** Contains the significant digits of the number, defining its precision.

In our design, the floating point format employs 1 bit for the sign, 9 bits for the exponent, and 22 bits for the mantissa, establishing an exponent bias of $2^{9-1} - 1 = 255$.

The process of adding two floating point numbers involves aligning their decimal points by adjusting the exponents, then adding the mantissas. Subsequent steps include normalizing the result and potentially rounding it to fit the fixed bit-width. The final sign of the result is influenced by the initial signs of the operands.

Floating point adders are indispensable in domains such as scientific computing, data analysis, and computer graphics, where precision is mandatory. By leveraging specialized hardware, these adders provide faster computation speeds than general-purpose processors. This project has implemented and tested a floating point adder using Logisim, providing detailed insights into its design and functionality.

2 Problem Specification

The task involves designing a floating-point adder circuit which takes two floating points as inputs and provides their sum, another floating point as output. Each floating point will be 32 bits long with following representation:

Sign	Exponent	Fraction
1 bit	9 bits	22bits

Table 1: Problem specification

3 Flowchart of the addition/subtraction algorithm

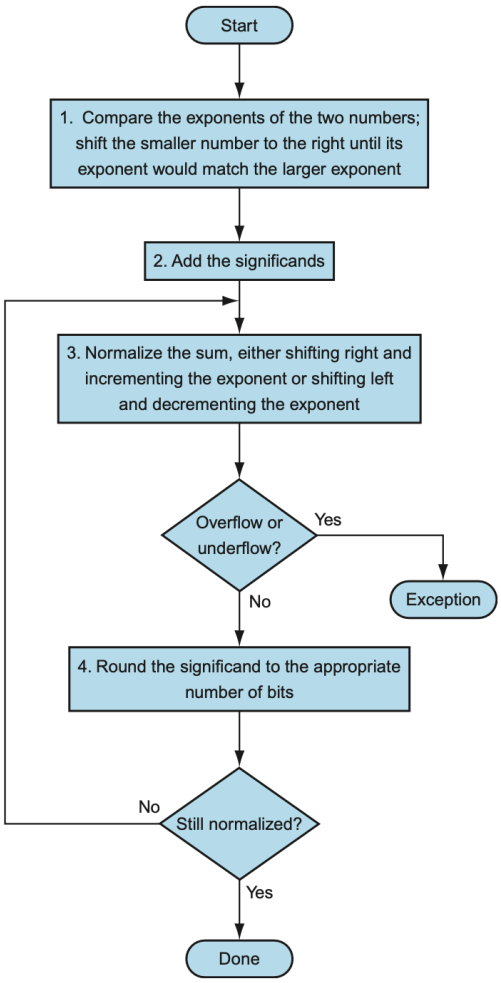

Figure 1: Flowchart of the addition/subtraction algorithm

4 High-level block diagram of the architecture

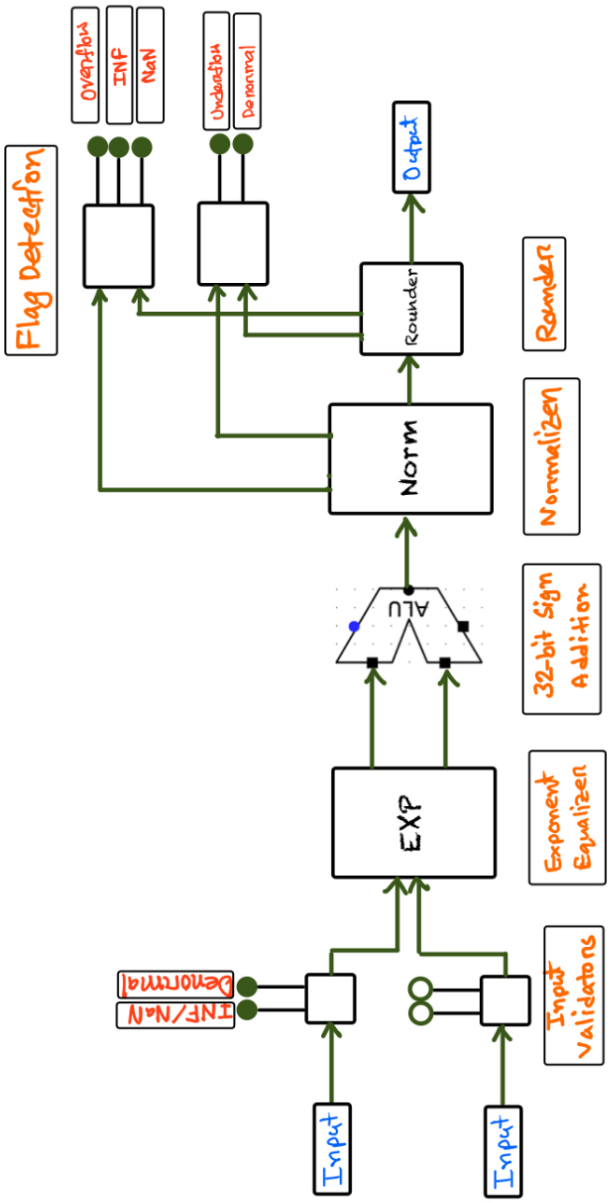

Figure 2: High-level overview flow diagram

5 Descriptions of Components

5.1 Input Validator

The Input Validator module ensures that any given input adheres to IEEE 754 floating-point standards. If invalid inputs such as NaN (Not a Number), denormalized numbers, or infinity are provided, the module raises exceptions and sets the corresponding error flags.

i. Check if the input represents NaN:

   - If `exponent == all ones` and `fraction != 0`, raise a NaN flag and terminate further processing.

ii. Check if the input is denormalized:

   - If `exponent == all zeros` and `fraction != 0`, raise a Denormal flag and terminate further processing.

iii. Check if the input is infinite:

   - If `exponent == all ones` and `fraction == 0`, raise an Infinity flag and terminate further processing.

iv. If none of the above cases are detected, pass the input to the subsequent modules for normal processing.

Input: A 32-bit floating-point number.
Output: Either a valid floating-point number or an exception with the corresponding error flag.

5.2 Exponent Equalizer

The purpose of the Exponent Equalizer is to align the exponents of two floating-point numbers by equalizing them to the larger exponent. This is necessary to facilitate accurate addition or subtraction of the significands.

i. Identify the number with the larger exponent (p) and the smaller exponent (q).

ii. Compute the difference between the exponents, $n = |exp_p - exp_q|$.

iii. Assign a pre-radix of 1 to the numbers unless they are 0, in which case the pre-radix will also be 0.

iv. Add 7 GRS bits and perform an n-bit right shift on q's combined pre-radix, significand, and GRS.

v. Update q's exponent to match that of p.

Input: Two normalized floating-point numbers.
Output: Two adjusted floating-point numbers where both have the same exponent.

5.2.1 Large-Small Finder

This component sorts two floating-point numbers based on their exponents to ensure that the larger exponent is identified correctly.

i. Calculate the difference, $h = keyB - keyA$.

ii. If $h < 0$, assign A as $larger$ and B as $smaller$; otherwise, assign B as $larger$ and A as $smaller$.

Input: Two 32-bit floating-point numbers (A and B) and their 9-bit keys (exponents).
Output: The numbers sorted into $larger$ and $smaller$ based on their exponents.

5.3 Normalizer

The Normalizer ensures that the result of an operation is in normalized form, which means its pre-radix is adjusted to 01. Depending on the input, it can perform either a right or left shift to achieve normalization.

i. Check the pre-radix bits to determine if normalization is needed.

ii. Perform right-shift normalization if required.

iii. Perform left-shift normalization if needed.

iv. Use a multiplexer to combine the outputs of the shift operations and produce the normalized result.

Input: A 2-bit pre-radix, a 9-bit exponent, a 22-bit significand, and 7 GRS bits.
Output: A normalized floating-point number with an updated pre-radix and overflow/underflow flags.

5.3.1 Right Shift Normalizer

This module adjusts the floating-point number to normalized form by shifting its bits to the right. It ensures that the pre-radix bits become 01 and updates the exponent accordingly.

i. Shift the combined exponent, significand, and GRS one bit to the right.

ii. Increment the exponent by 1.

Input: A 2-bit pre-radix, a 9-bit exponent, a 22-bit significand, and 7 GRS bits.
Output: A normalized floating-point number and an overflow flag.

5.3.2 Left Shift Normalizer

The Left Shift Normalizer shifts the significand and GRS bits to the left when normalization requires increasing the significant bits. The exponent is decremented by the same amount to maintain the value.

i. Determine how many bits need to be shifted left based on the significand and GRS bits.

ii. Shift the combined exponent, significand, and GRS bits to the left.

iii. Decrement the exponent by the number of bits shifted.

Input: A 2-bit pre-radix, a 9-bit exponent, a 22-bit significand, and 7 GRS bits.
Output: A normalized floating-point number and an underflow flag.

5.4 Rounder with GRS

The Rounder uses the GRS (Guard, Round, Sticky) bits to decide how to adjust the final result for greater precision. It ensures the output remains consistent with IEEE 754 rounding rules while handling overflows gracefully.

i. Analyze the GRS bits to determine the rounding direction.

ii. Adjust the significand based on the rounding decision.

iii. If rounding causes the significand to overflow, increment the exponent by 1.

iv. Check if the updated exponent exceeds the permissible limit, and if so, set the overflow flag.

Input: A 32-bit floating-point number with 7 GRS bits.
Output: A rounded 32-bit floating-point number and an overflow flag.

The table below demonstrates the behavior of the Rounder with different combinations of the G, R, and Sticky bits:

Table 2: GRS Bit Behavior and Rounding Value

G (Guard)	**R (Round)**	**Sticky (5 Bits)**	**Rounding Value**
0	0	00000	No rounding (retain original significand)
0	1	00000	Round down (no change in significand)
1	0	00000	No rounding (retain original significand)
1	1	00001 or more	Round up (increment significand)
0	0	11111	No rounding (sticky bits insignificant)
1	1	00000	Round up (significant GRS bits)

6 Additional Components

6.1 Arithmetic Logic Unit (ALU)

The ALU supports multiple operations as summarized in Table 3.

Mode	Description	32-bit Output	Extra Bit Output
00	Addition (2's complement)	$A + B$	C_{out}
01	Subtraction	$A - B$	B_{out}
10	Complement	A'	Z
11	Sign Magnitude Addition	$A + B$	C_{out}

Table 3: ALU Modes and Operations

6.2 Multiplexers

Multiplexers are built using 74157 ICs. A single 74157 can act as a 4-bit number selector. By combining two such selectors, a 32-bit number selector can be created.

6.3 Clampers

Clampers reduce a 32-bit number to a 5-bit output. The process involves using an ALU in mode '01' for comparison and then multiplying the result with an 8-bit number selector.

6.4 Shifters

6.4.1 Fixed Amount Shifters

Fixed amount shifters can be implemented by directly manipulating the positions of bits. Using splitters and mergers, any fixed amount shift can be created without the need for additional ICs.

6.4.2 Fixed Amount Shifters with Enable

To create variable fixed amount shifters, a 32-bit number selector is used. This allows for either no shift or a predefined shift amount to be applied.

6.4.3 Variable Amount Shifters

For arbitrary shift amounts, fixed shifters with enable functionality are used. A 5-bit binary input determines which shifters are activated in a series configuration to achieve the desired shift.

6.5 Bitwise AND

A 4-bit bitwise AND can be constructed using a single 74x08 IC. To create a 5-bit bitwise AND module, additional ICs can be daisy-chained. This module is used in various parts of the FPA.

6.6 Priority Encoders

Using a 74x148 IC, an 8-to-3 (8:3) priority encoder can be constructed. By daisy-chaining these encoders, larger encoders can be designed. Since the 74x148 IC operates with active-low logic, a complementary version using NOT gates is created to support active-high inputs. A 32:5 active-high priority encoder is used in our final implementation.

7 Detailed Circuit Diagrams

7.1 FPA Interface

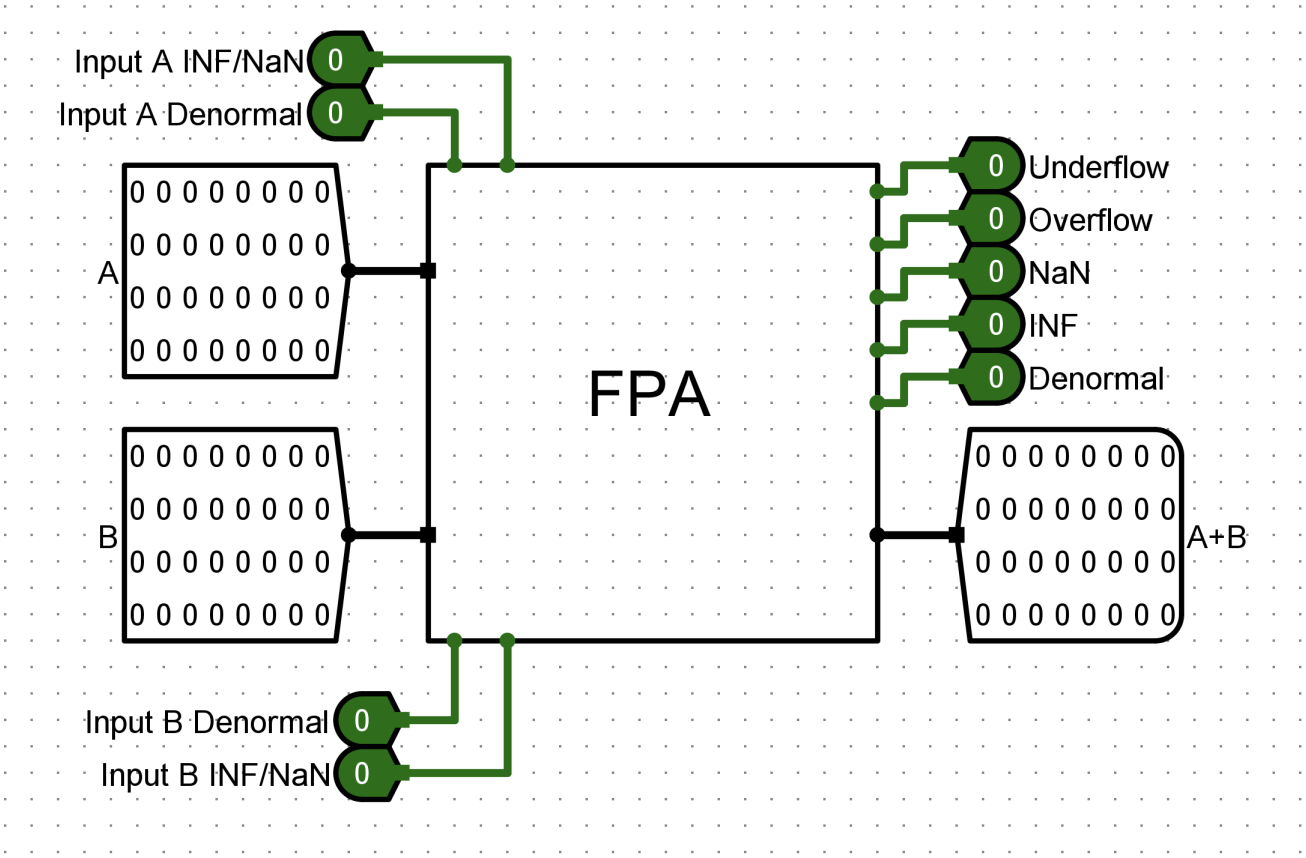

Figure 3: FPA Interface

7.2 FPA Circuit

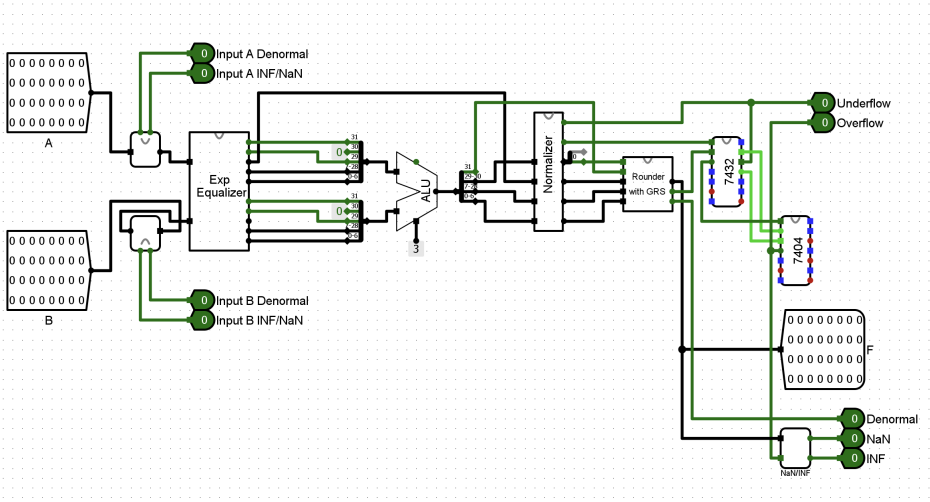

Figure 4: FPA with Flag

7.3 Invalid Input Handler

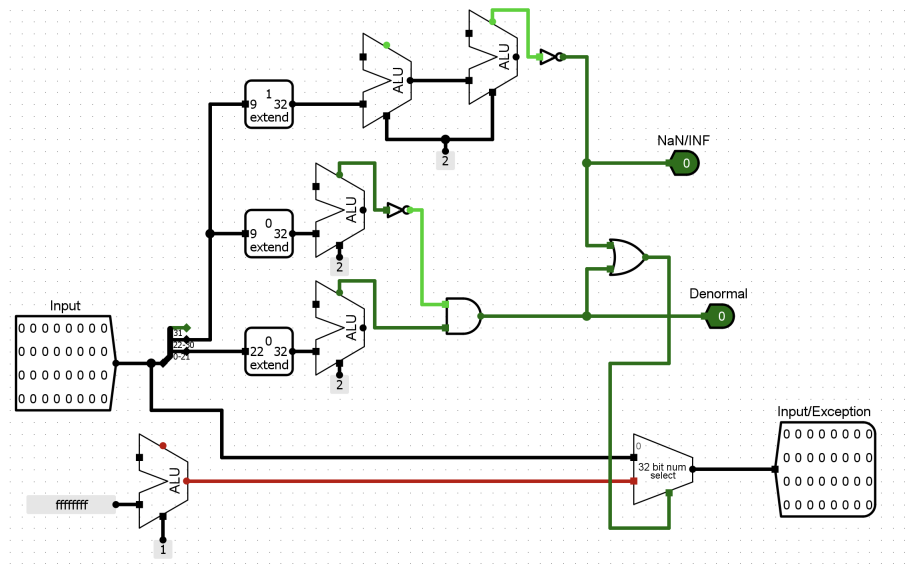

Figure 5: Invalid Input Handler

7.4 Exponent Equalizer

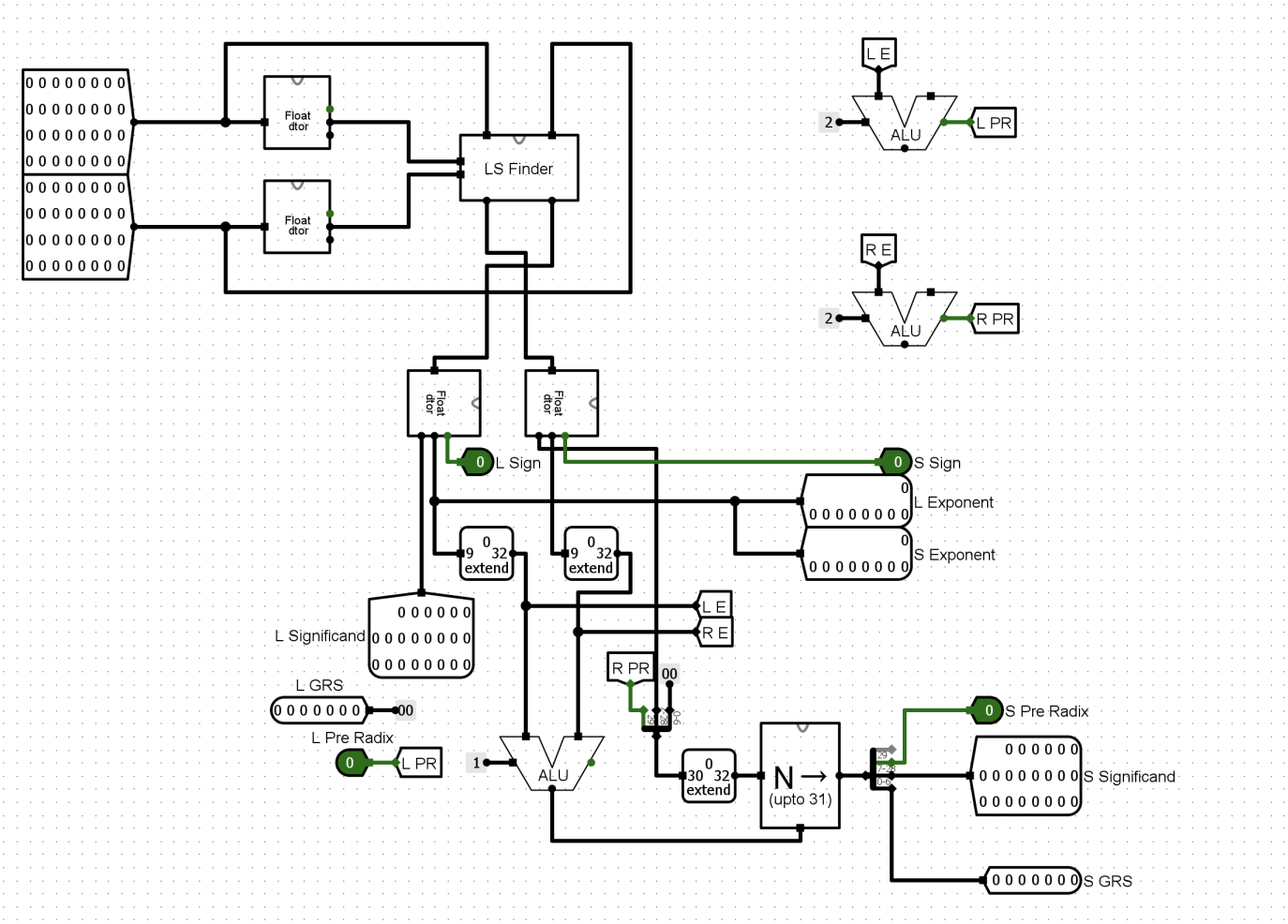

Figure 6: Exponent Equalizer

7.4.1 Float Breaker

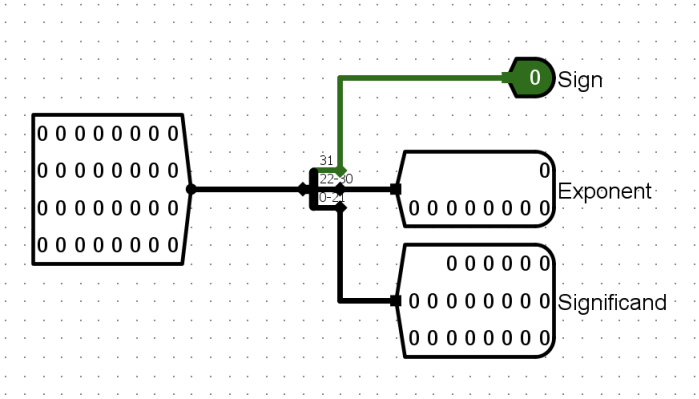

Figure 7: Float Breaker

7.4.2 LS Finder

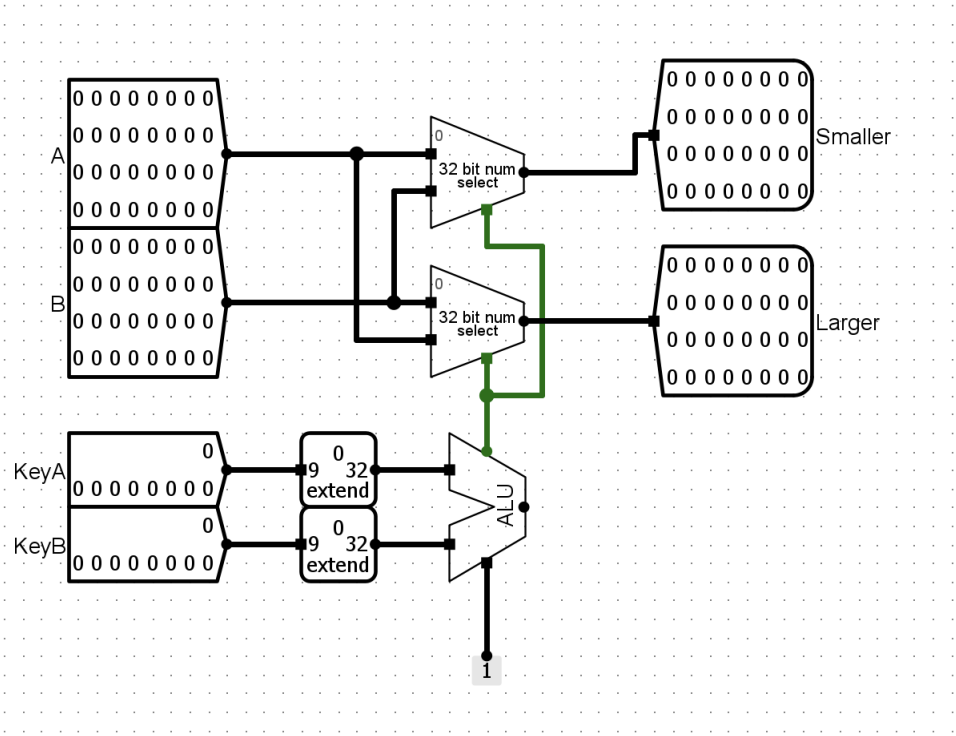

Figure 8: LS Finder

7.5 Normalizer

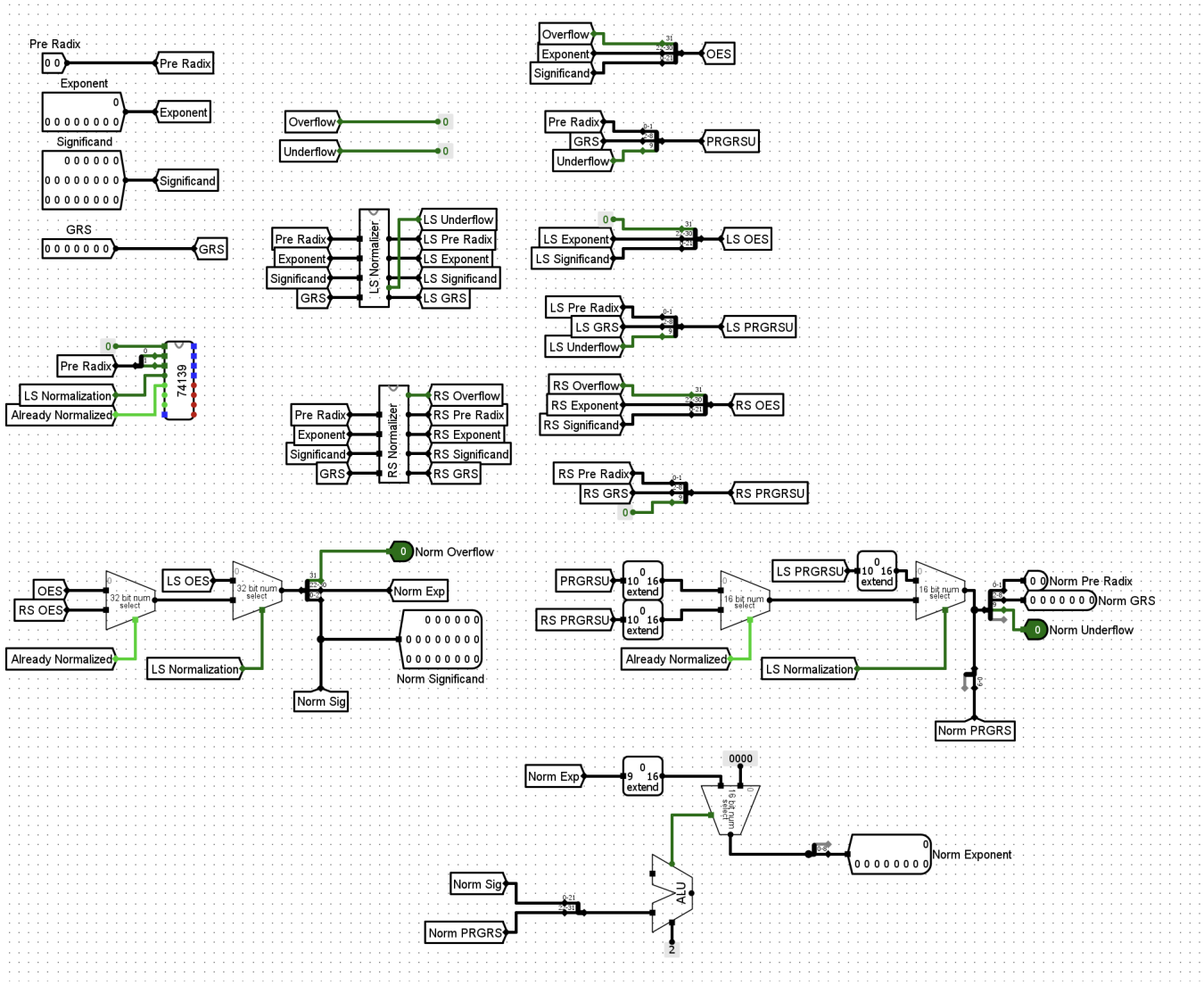

Figure 9: Normalizer

7.5.1 Right Shift Normalizer

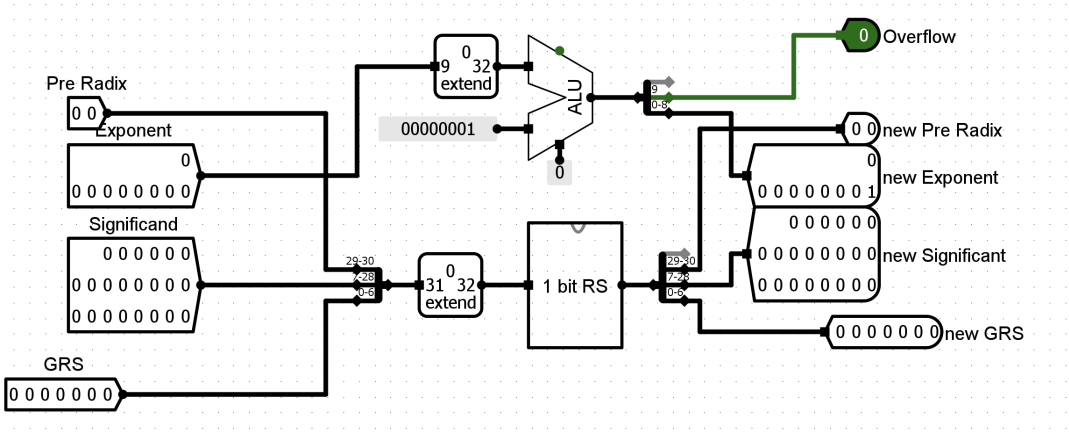

Figure 10: Right Shift Normalizer

7.5.2 Left Shift Normalizer

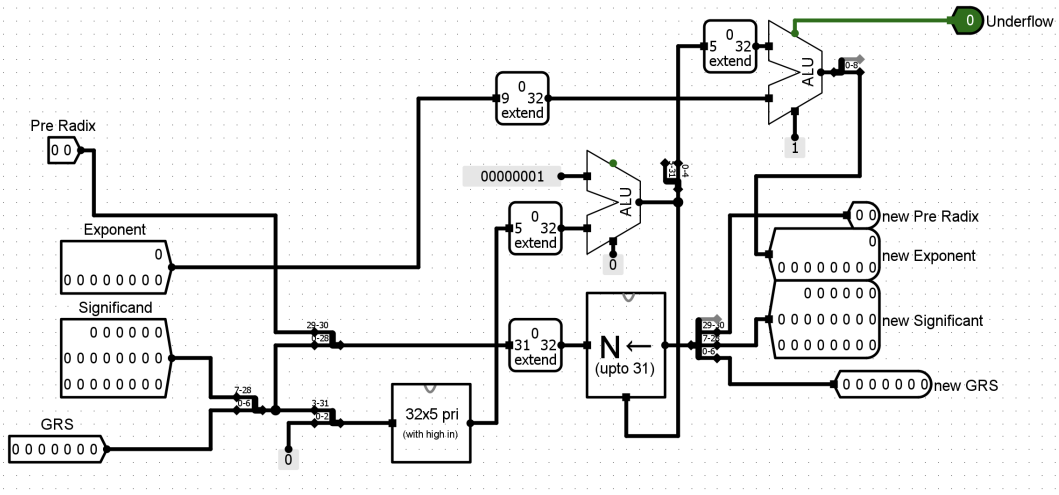

Figure 11: Left Shift Normalizer

7.6 Rounder

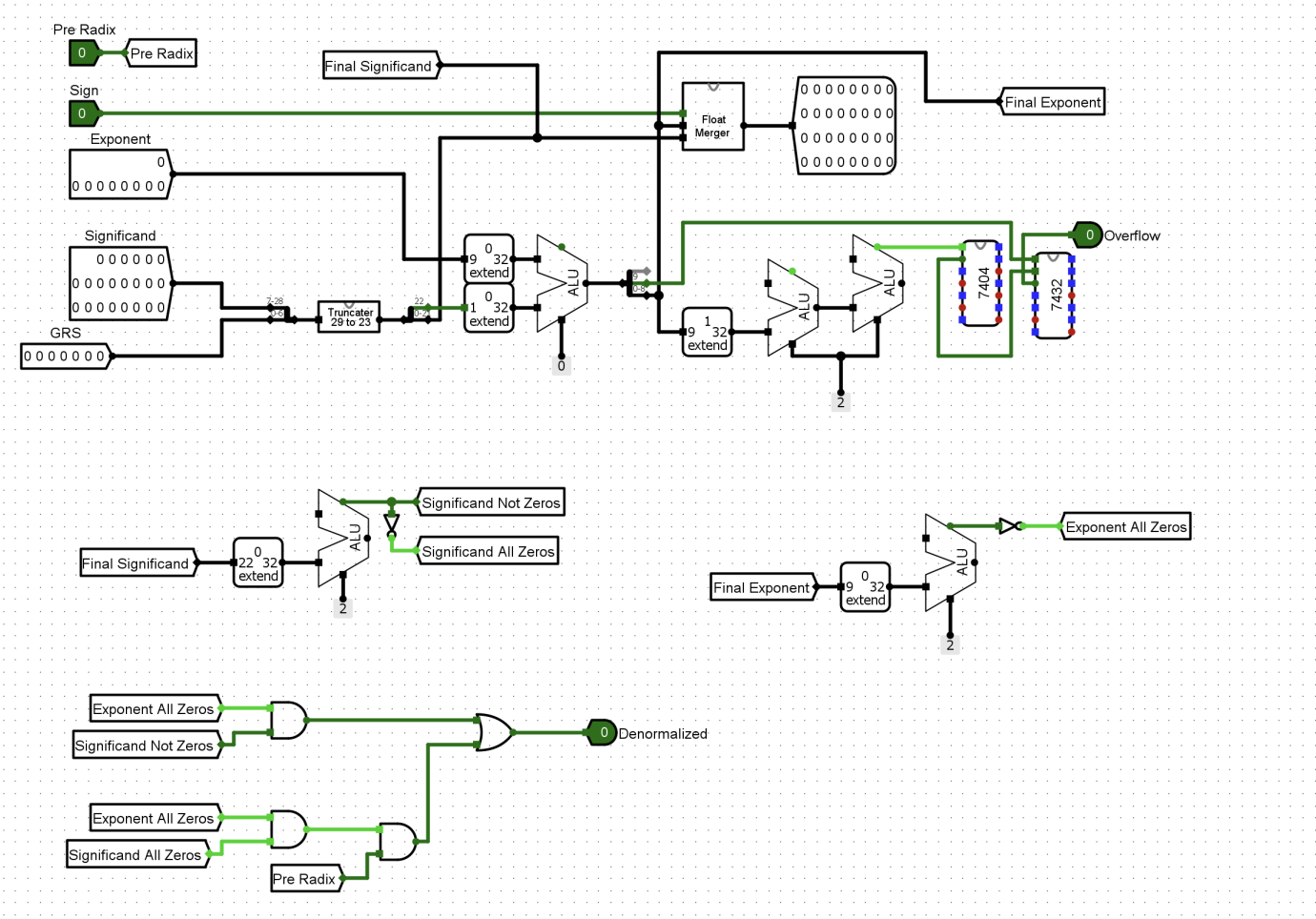

Figure 12: Rounder

7.6.1 Float Merger

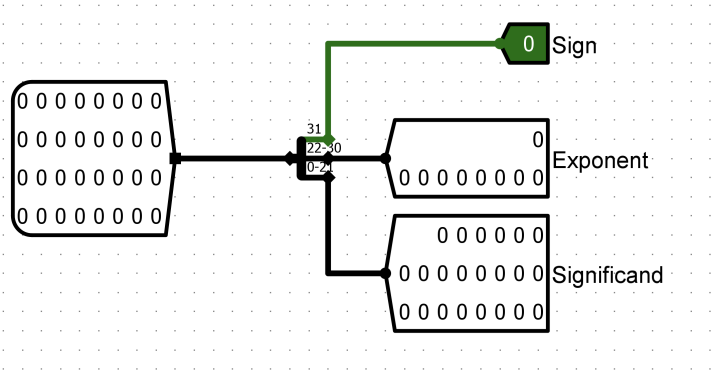

Figure 13: Float Merger

7.6.2 Truncator

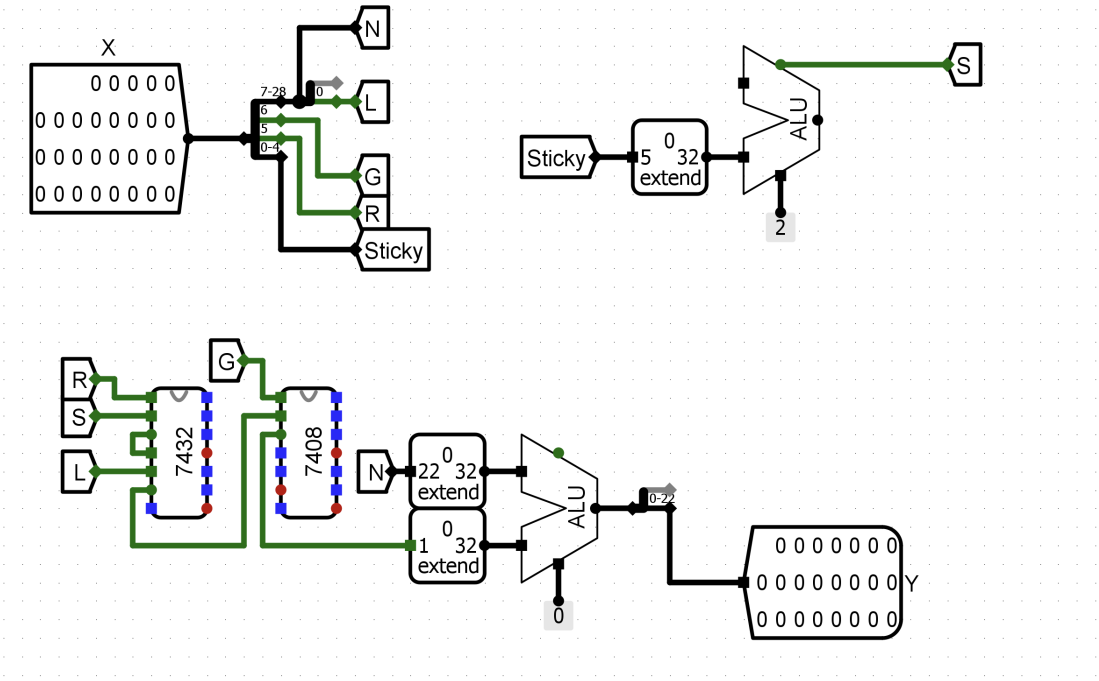

Figure 14: Truncator

7.7 Clamper

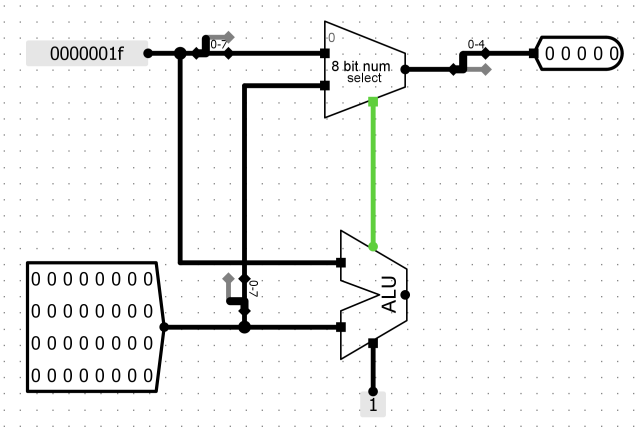

Figure 15: Clamper

7.8 32-bit ALU

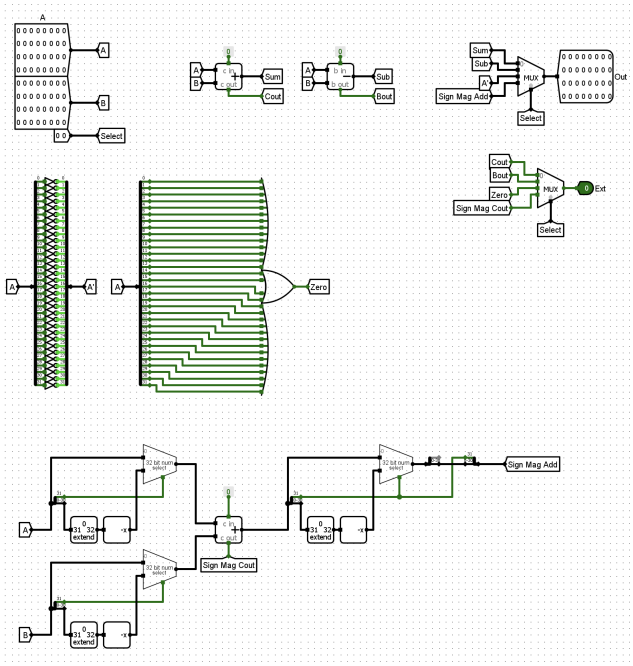

Figure 16: 32-bit ALU

7.9 Bit Selector

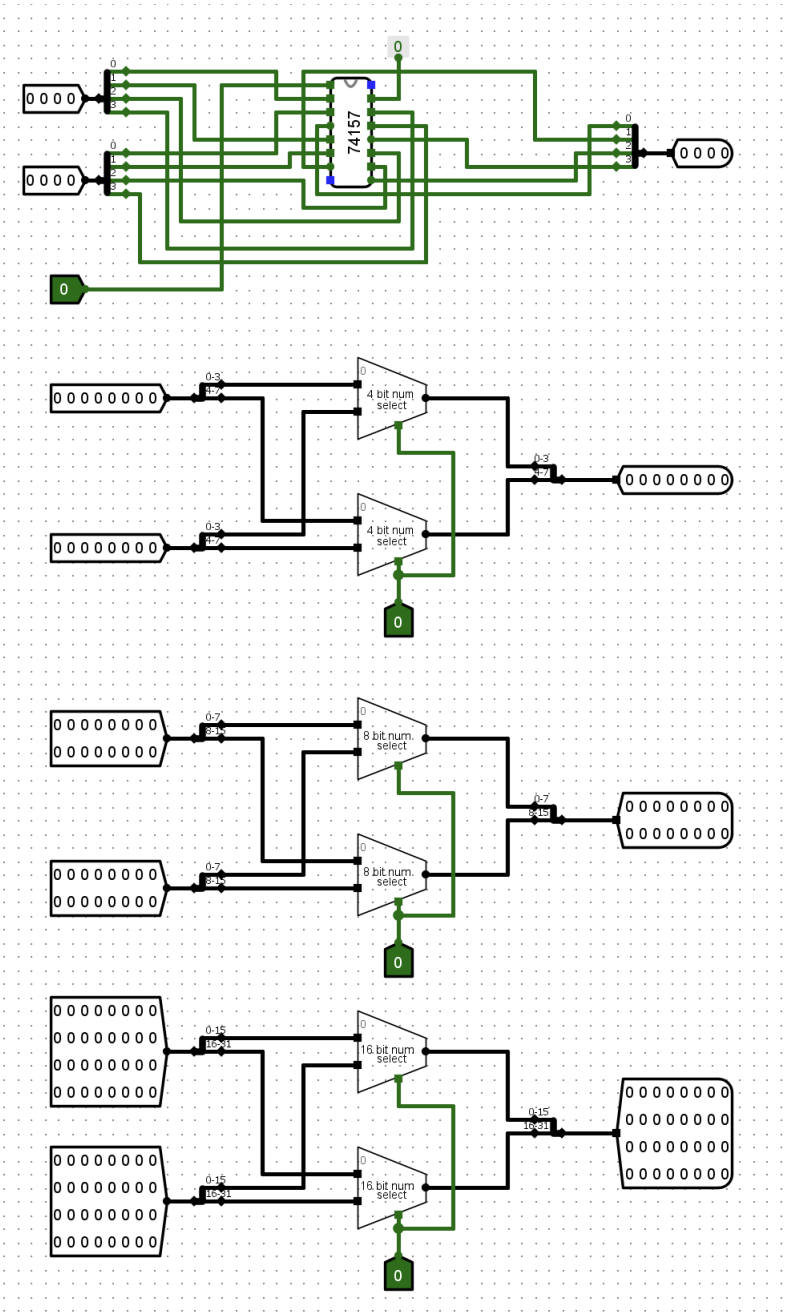

Figure 17: Bit Selector

7.10 Shifters

7.10.1 Fixed Bit Shifters

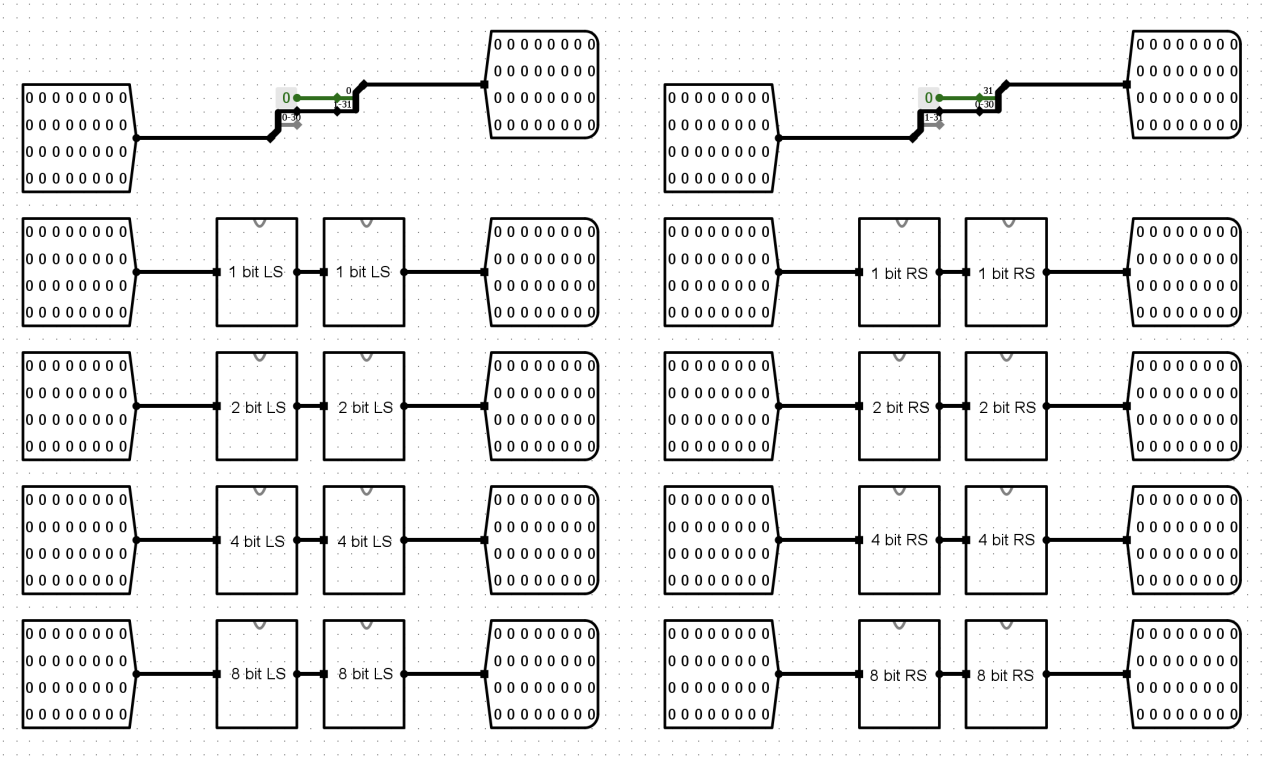

Figure 18: Shifters

7.10.2 Fixed Bit Shifters with Enable

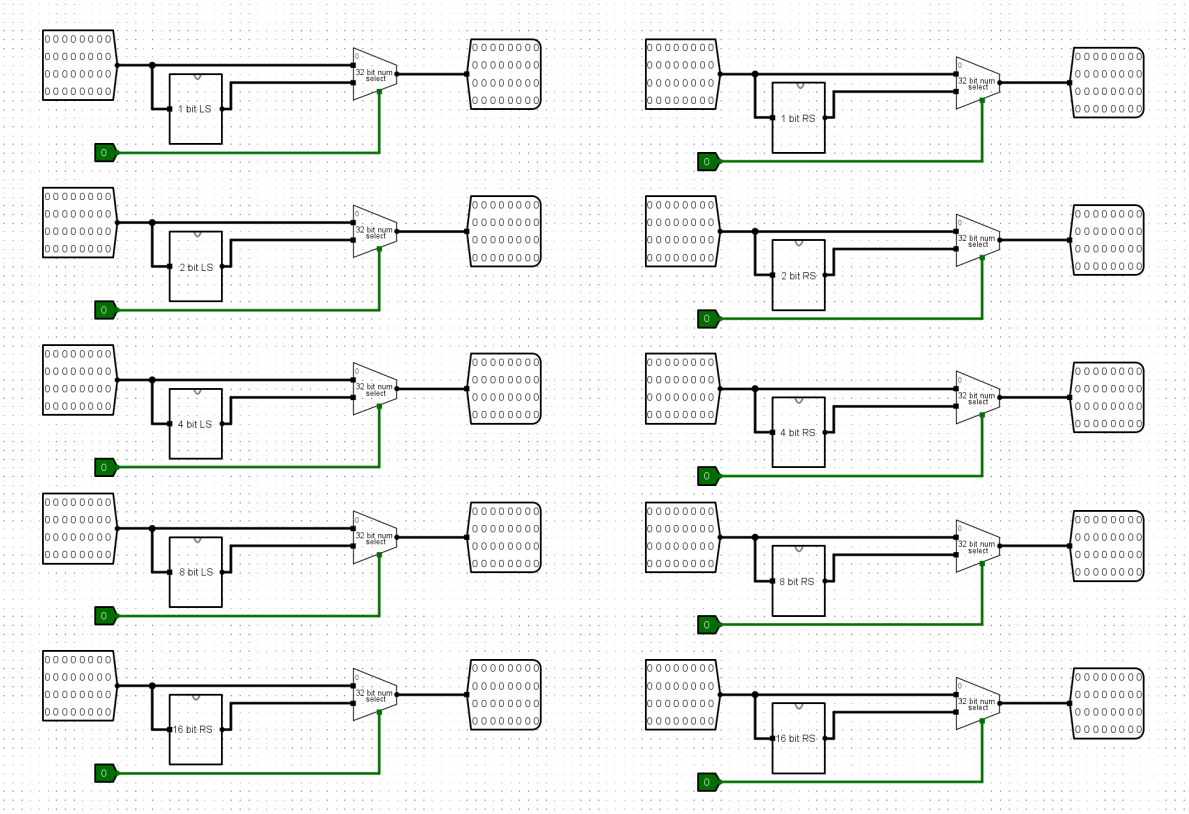

Figure 19: Fixed Bit Shifters with Enable

7.10.3 Variable Bit Left Shifter

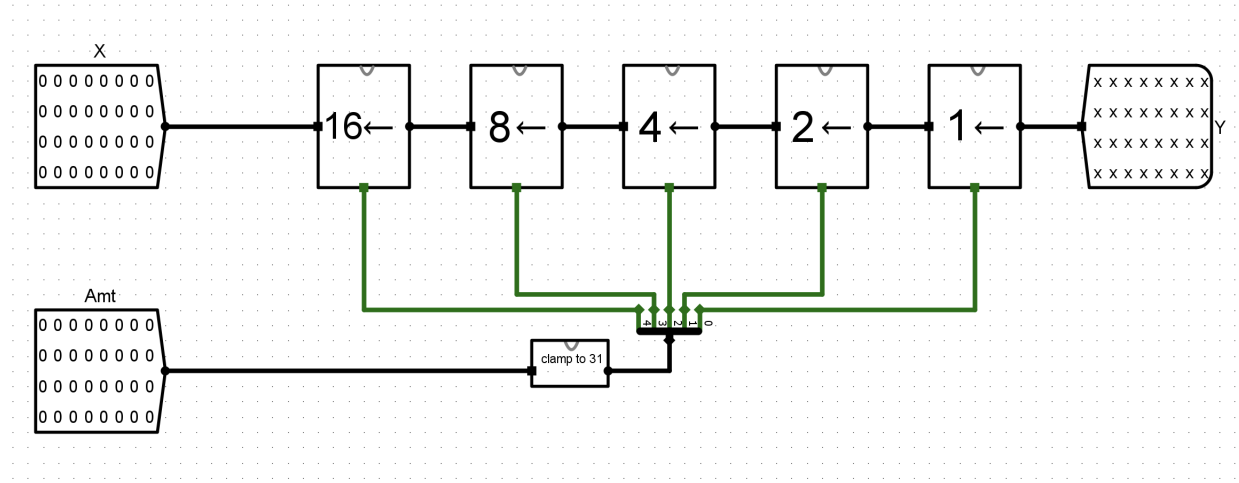

Figure 20: Variable Bit Left Shifter

7.10.4 Variable Bit Right Shifter

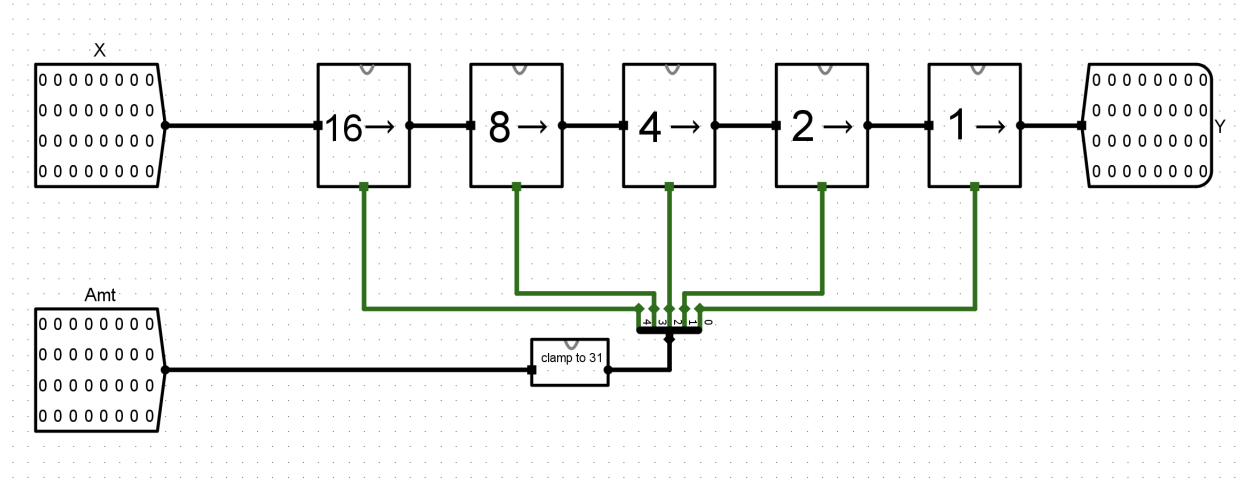

Figure 21: Variable Bit Right Shifter

7.11 Bitwise AND

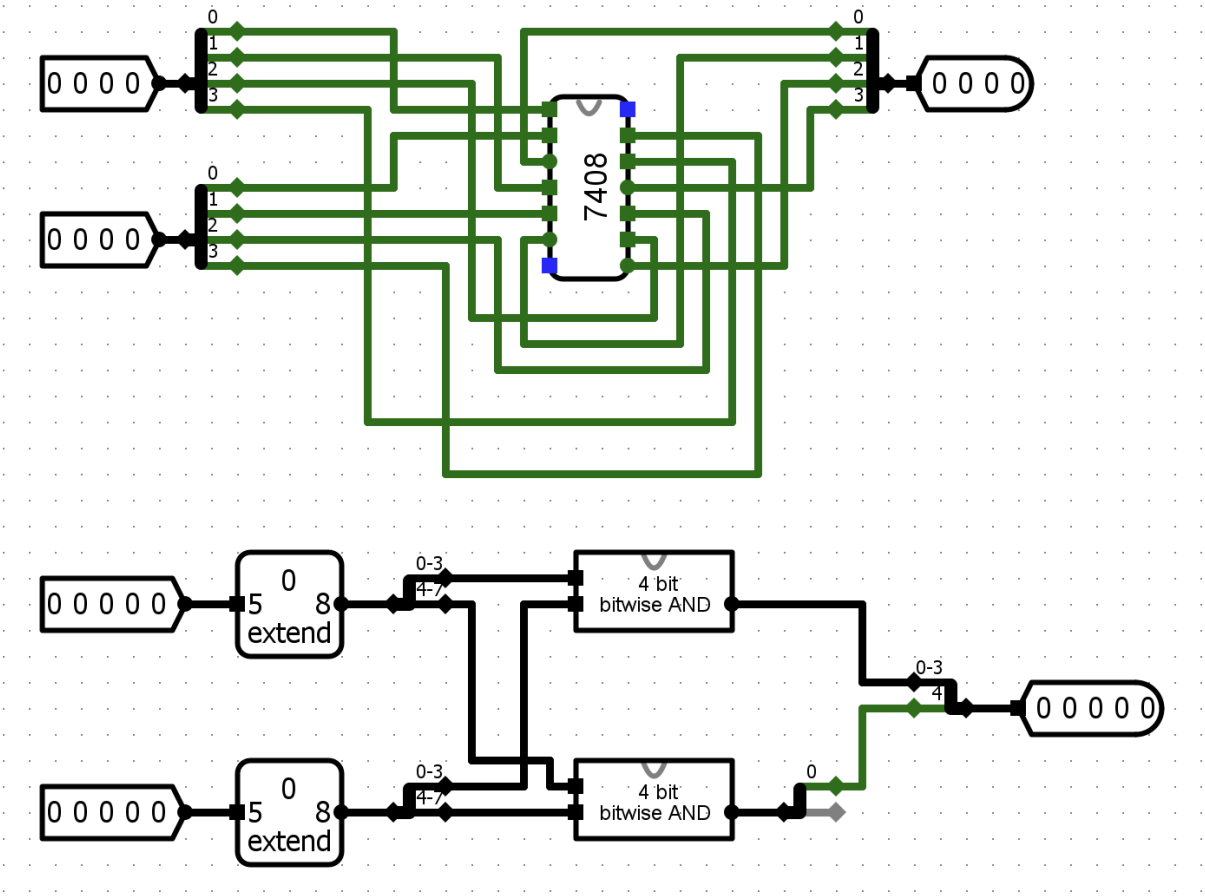

Figure 22: Bitwise AND

7.12 Priority Encoder

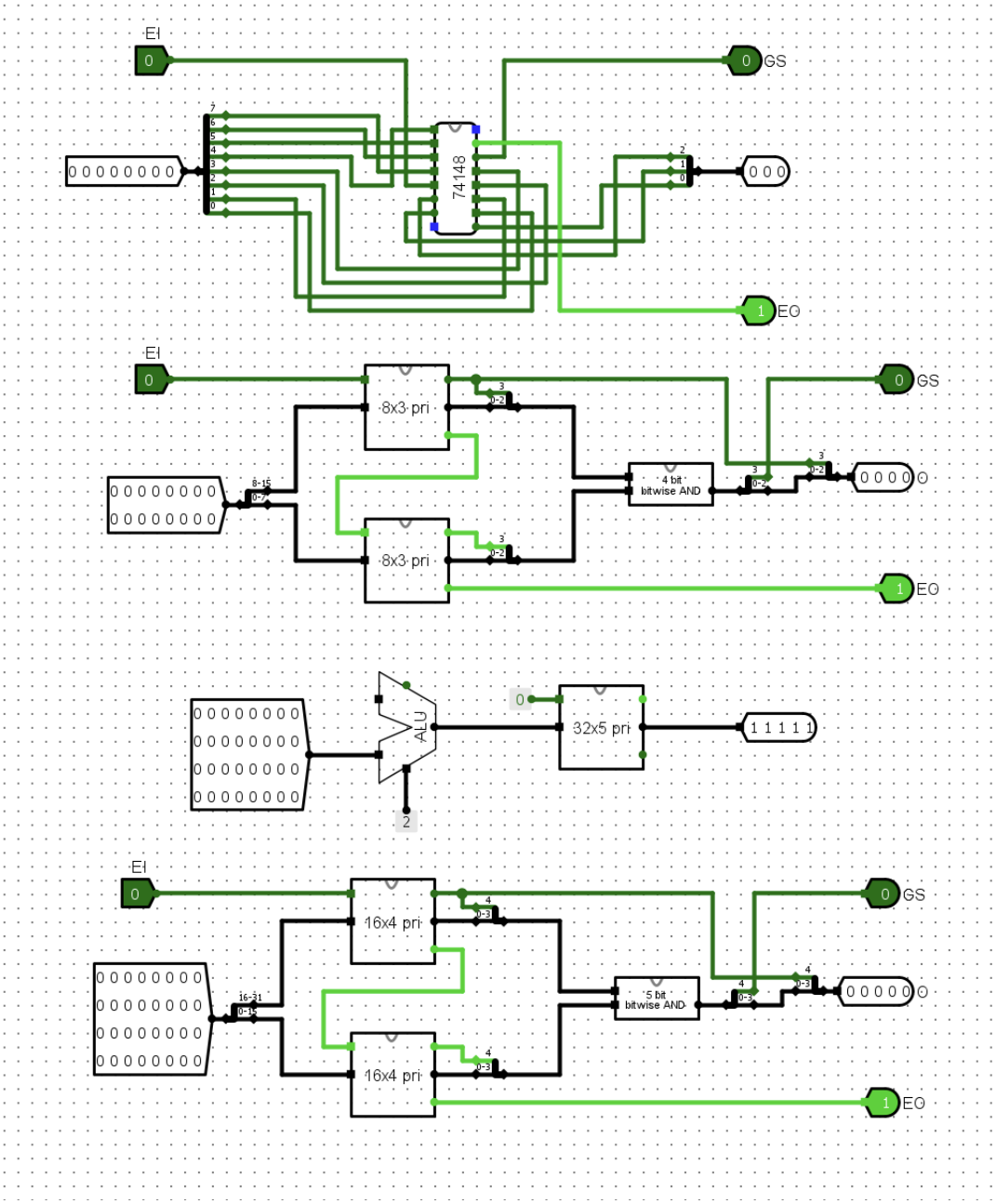

Figure 23: Priority Encoder

8 Requirements

8.1 Simulator used along with the version number

We used Logisim-ITA (win-2.16.1.4).

8.2 ICs and Gates

IC/Gate	Count	Size (bits)
74x157 (MUX 2*1)	125	32
32-bit ALU	26	32
74x148 (Priority Encoder)	4	32
(OR Gate)	8	2
(AND Gate)	11	2
74x04 (NOT Gate)	7	1
74x139 (Decoder/Demux)	1	4

Table 4: IC and Gate Counts Used in the Design

9 Contribution of members

1. **Design:** 2105052, 2105056
2. **Software Implementation:**
   - Input Processor: 2105050, 2105052
   - Significand Adder/Subtactor: 2105050, 2105052, 2105056
   - Normalizer: 2105050, 2105052
   - Rounder: 2105056
   - INF, NaN, Denormalized Flags: 2105050, 2105056
   - FPA & Output Processor: 2105050, 2105052, 2105056
3. **Testing:** 2105050, 2105056
4. **Debugging:** 2105052, 2105056
5. **Report Writing:**
   - 2105050: Section 1, 2, 7
   - 2105052: Section 5, 6, 8
   - 2105056: Section 3, 4, 5

10 Discussion

We designed and constructed the floating-point adder (FPA) by segmenting the architecture into distinct modules, each dedicated to handling specific functionalities. This modular strategy not only streamlined the design process but also enhanced clarity and manageability of the circuit's overall functionality. This approach facilitated troubleshooting and iterative improvements throughout the development cycle.

During the development of the FPA, we faced multiple challenges, particularly with Logisim's limitations and the inherent complexities of floating-point arithmetic. A significant hurdle was the built-in shifter's limitation to a 5-bit shift amount. This constraint required us to develop a custom solution for handling larger exponent differences by manually managing the shift operations and truncating extraneous bits to prevent data overflow.

Another challenge involved accurately determining the larger of two input values, critical for correct addition and subtraction operations. We implemented a comparison mechanism that first evaluates the exponents and, if they are equal, compares the mantissas. The outcome of this comparison not only determined the larger value but also influenced the resulting sign after the arithmetic operation.

The arithmetic operations within the ALU were dynamically chosen based on the signs of the inputs; identical signs triggered an addition, while differing signs necessitated a subtraction. Managing this dynamic selection efficiently required careful integration of control logic within the ALU.

Moreover, handling special cases such as overflow and underflow presented additional complexity. We utilized the carry-out bit from the adder and a dedicated checker circuit to detect overflow conditions, whereas underflow was monitored through the borrow-out bit from the subtractor alongside a zero-check mechanism.

We also incorporated flags to indicate invalid or special input cases, such as denormalized numbers and operations resulting in NaN (Not a Number), enhancing the robustness and reliability of the FPA by ensuring such anomalies were clearly flagged and handled appropriately.

Integrating custom-designed modules with Logisim's built-in components allowed us to optimize the circuit design, striking a balance between custom functionality and the efficiencies offered by standard components. This integration was crucial in achieving a high-performance, reliable floating-point adder.

Reflecting on the project, the design and implementation of the FPA provided us with profound insights into the computational intricacies and challenges of digital circuit design, particularly in the context of floating-point arithmetic. This experience has significantly enriched our understanding and skills in both theoretical and practical aspects of computer architecture.