# Bangladesh University of Engineering and Technology

CSE 306
Computer Architecture Sessional

Offline 2: 32-bit Floating Point Adder Circuit

Section - B1
Group - 5

**2005061** - Farriha Afnan
**2005072** - Abrar Rahman Abir
**2005079** - Ananya Shahrin Promi
**2005087** - Iffat Bin Hossain
**2005089** - Wahid Al Azad Navid

January 14, 2024

# 1 Introduction

Adding floating-point numbers is a big deal in scientific calculations. It is tricky because we need to be precise with our numbers.
A floating-point number has three parts: a sign (plus or minus), an exponent (like a power), and a fraction (some bits after the dot). These parts come together using a special formula to create the floating-point number. This formula ensures that our number is in a standard form, and the exponent is always positive.

$$(-1)^{\text{sign}} \cdot (1 + \text{Fraction}) \cdot 2^{\text{Exponent}-\text{Bias}}$$

When we use a tool called a floating-point adder, we put in two floating-point numbers and get a sum as the result. But when we're adding, we have to be careful about the form of our numbers and make sure we're not losing any important details.

# 2 Problem Specification

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

| **Sign** | **Exponent** | **Fraction** |
|---|---|---|
| 1 Bit | 11 Bit | 20 Bit |

Table 1: Problem Specification

Input is in IEEE 754 Standard format of binary representation of the floating point number (sign bit, exponent bit, fraction bits). The number obtained after addition is required to be in normal form. Besides, we need to ensure that the number is rounded off properly.

# 3 Flowchart of the Addition/Subtraction Algorithm

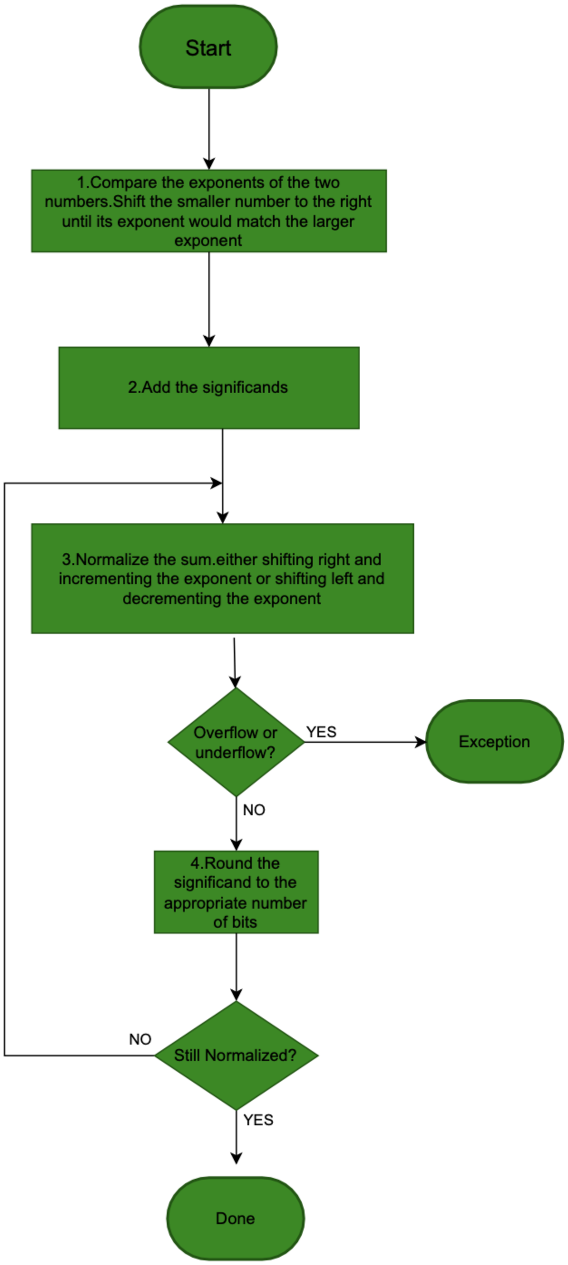

Start
1.Compare the exponents of the two numbers.Shift the smaller number to the right until its exponent would match the larger exponent
2.Add the significands
3.Normalize the sum.either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
YES
Exception
NO
4.Round the significand to the appropriate number of bits
NO
Still Normalized?
YES
Done

Figure 1: Flowchart of the Algorithm

# 4 High-Level Block Diagram of the Architecture

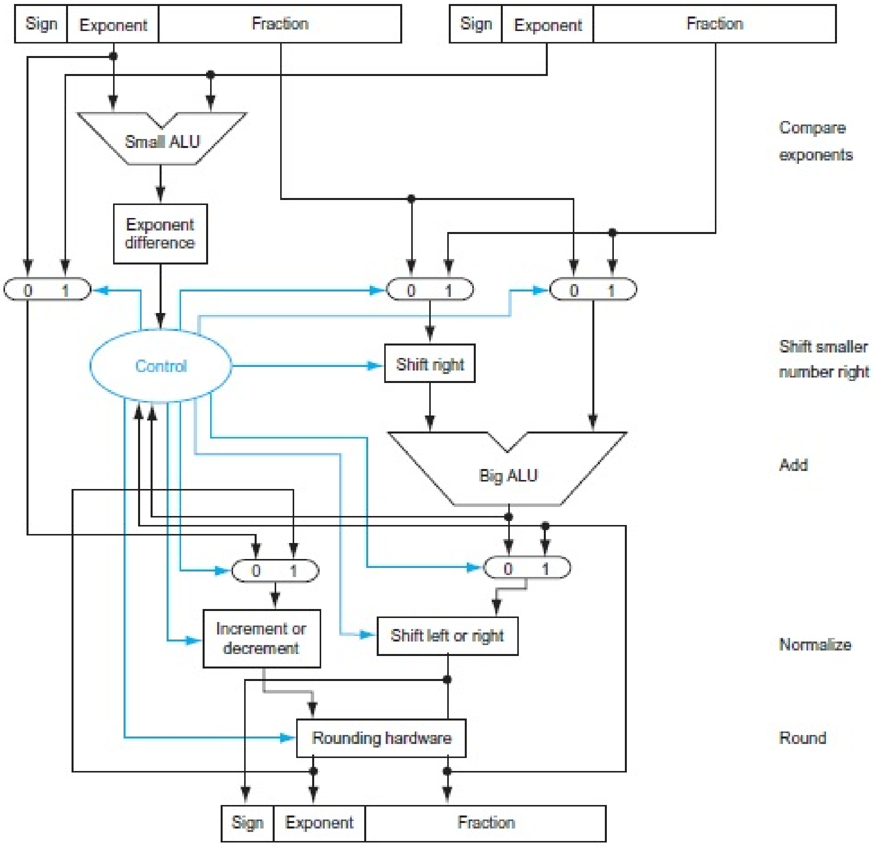


Figure 2: High-Level Block Diagram

# 5 Detailed Circuit Diagram of the Important Blocks

We will explain the design steps by breaking down and detailing each important component.

## 5.1 Multiplexer Library

The n-bit 2x1 MUX circuits take two n-bit inputs and output the one selected by a one-bit selector, constructed using cascading IC74157.

## 5.2 AND Library

The n-Bit $m$x1 AND circuits take $m$ n-bit inputs and output their bitwise AND. These are constructed using cascading IC7408.

## 5.3 OR Library

The n-Bit $m$x1 OR circuits take $m$ $n$-bit inputs and output their bitwise OR. These are constructed using cascading IC7432.

## 5.4 Exclusive OR Library

n-Bit $m$x1 XOR takes $m$ n-bit inputs and outputs their bitwise XOR. These are constructed using cascading IC7486.

## 5.5 NOT Library

n-bit NOT circuits take n-bit input and output its bitwise NOT. These are constructed using cascading IC7404.

## 5.6 Adder Library

n-Bit Adder takes two n-bit inputs and a carry-in bit and outputs their bitwise sum and the carry-out bit, constructed using cascading IC7483.

## 5.7 Value Add to LSB

It analyzes the 4 least significant bits of a number and outputs the value that needs to be added to the new LSB.

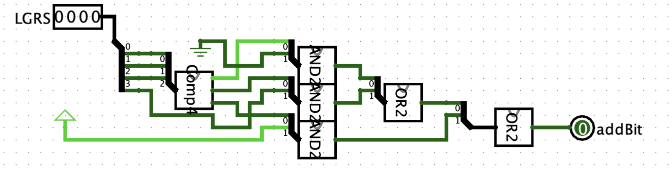


Figure 3: AdderBit

## 5.8 Compare Input with 4

This circuit takes a 3-bit input and outputs whether the input is greater, equal, or less than 4.

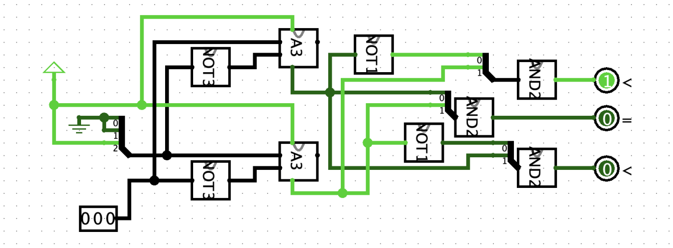


Figure 4: CompareWith4

## 5.9 Input Splitter

It splits incoming 32-bit data into 3 outgoing wires. First, two 32-bit inputs, $A$ and $B$ are inserted into InputSplitter. This will split the 32-bits into three individual lines of 1-bit (sign bit $A_s$ and $B_s$), 11-bit (exponent $A_e$ and $B_e$), and 20-bit (fraction part $A_f$ and $B_f$) each.

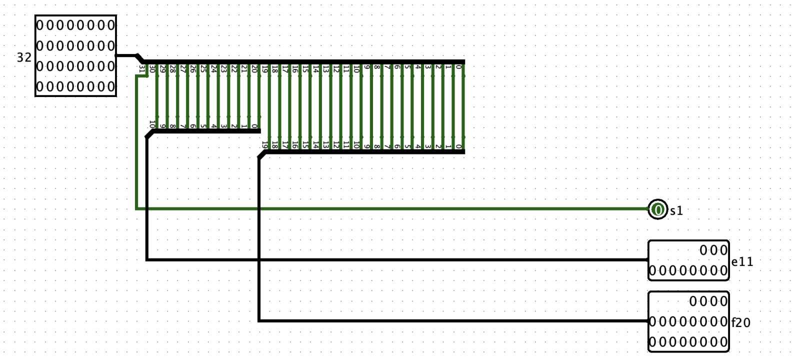


Figure 5: InputSplitter

## 5.10 Find Control Values

Control finds the difference between $A_e$ and $B_e$, $A_f$ and $B_f$. This is used to output the absolute difference between $A_e$ and $B_e$ (denoted as $D_e$), the larger value between absolute values of $A$ and $B$ ($S_{|A|>=|B|}$), and whether the absolute values of $A$ and $B$ are identical ($S_{|A|=|B|}$). These values are used as selector bits for some MUXs. They are also used to shift fraction values. Using 2x1 MUXs and $S_{|A|>=|B|}$, we calculate $G_s, G_e, G_f, L_s, L_f$, such that, $G = max(|A|, |B|)$ and $L = min(|A|, |B|)$. We need to consistently perform a right shift on the addend with the lower exponent to align it with the exponent of the larger value. Therefore, it's crucial to identify which addend is larger and which one is smaller.

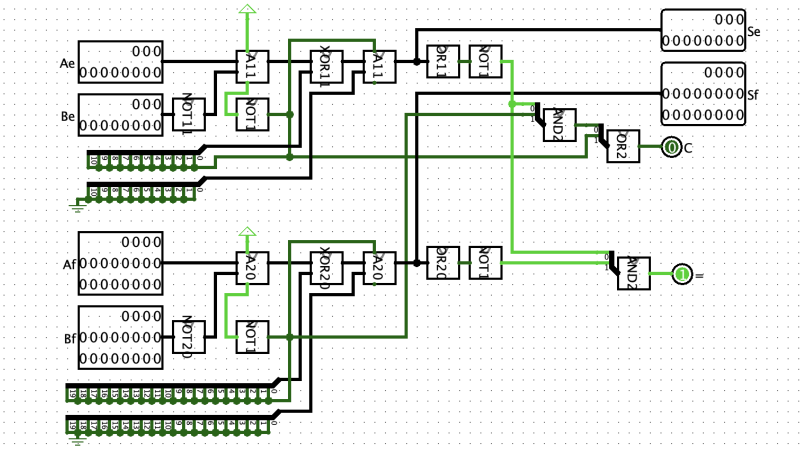


Figure 6: Control

Another key insight is that, before addition, both addends should be considered to have the same exponent value, specifically that of the larger addend. Consequently, the exponent value of the smaller addend becomes unnecessary for future calculations.

## 5.11 Modify Significand

It converts incoming 20-bit data into 32-bit output data, with MSB 1 and the LSB's 0.

Before proceeding with the addition, it is essential to determine whether to perform an addition or subtraction operation by evaluating the XOR value of $G_e$ and $L_e$. All the current representations of the fraction parts lack a leading 1. To address this, we employ the makeSignificand circuit, which adds a leading one at the leftmost bit of each fraction and converts the 21-bit fractions into 32-bit values.

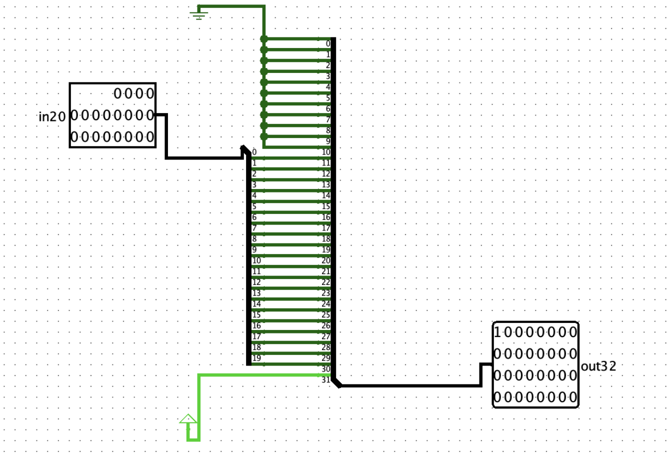


Figure 7: MakeSignificand

## 5.12 Left Shifter

It takes a 32-bit input and an 11-bit input (shift amount). CustomLeftShifter outputs the input left shifted by that amount, with the shifted bits replaced by the shift bit.

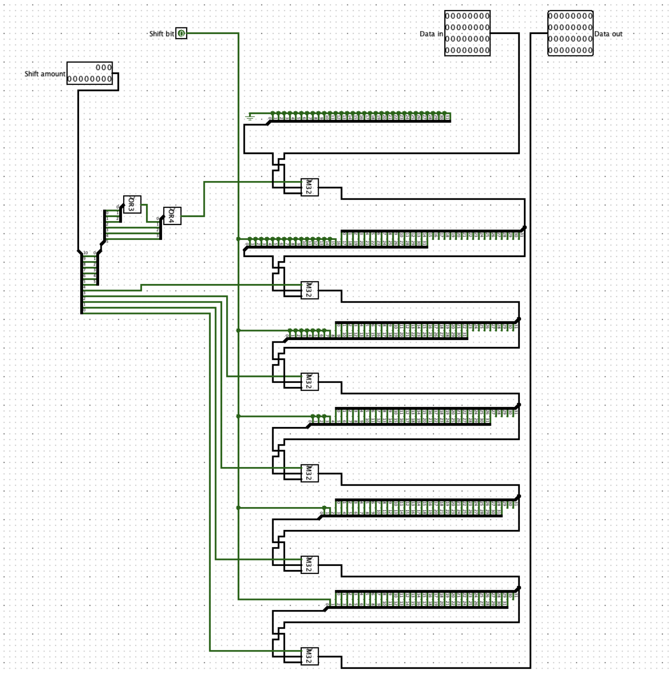


Figure 8: CustomLeftShifter

## 5.13 Right Shifter

This circuit takes a 32-bit input and an 11-bit input (shift amount) and outputs the input right shifted by that amount, with the shifted bits replaced by the shift bit. Thus, inserting $L_f$ and $D_e$ will shift $L_f$ by an amount of $D_e$, making $G_e = L_e$.

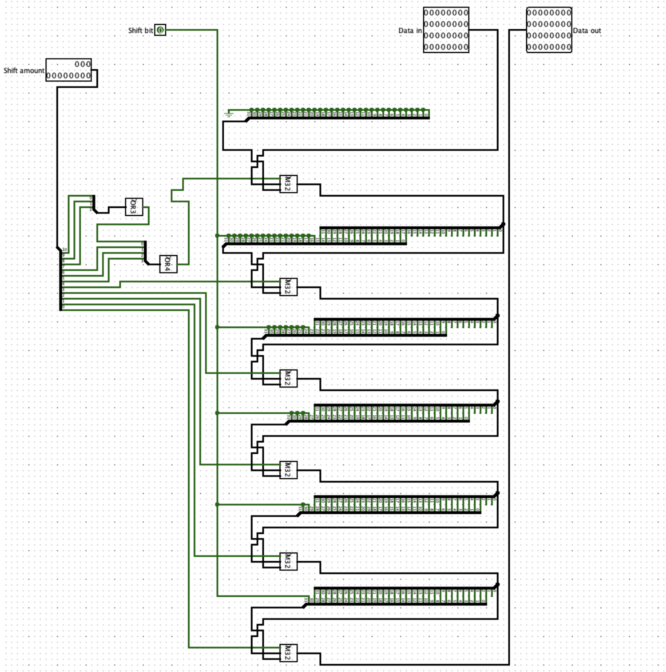


Figure 9: CustomRightShifter

## 5.14 Inverter

It takes a 32-bit input and a selector and outputs the same input if the selector is set to 0, otherwise outputs the bitwise not (1's complement) of the input.

In subtraction operations, regardless of the signs, we consistently subtract $L$ from $G$. This approach eliminates the need to invert the result after addition, and the sign of the result remains $G_s$. To achieve this, we utilized the SelectInverse circuit, which outputs the 2's complement of the right-shifted $L_f$ only when $G_e \oplus L_e$ is 1; otherwise, it outputs $L_f$ as it is. Moreover, $C_{in}$ should be 1 during the subtraction operation.

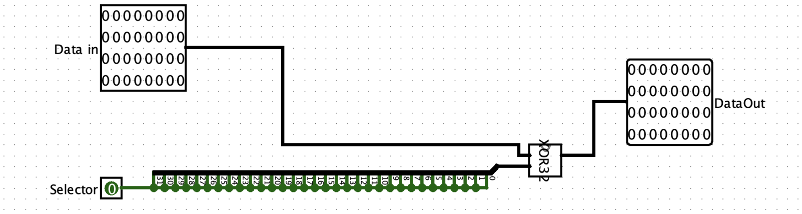


Figure 10: SelectInverse

## 5.15 Increment Exponent

We use IncrementExponent to achieve this $F_e = G_e + 1$.

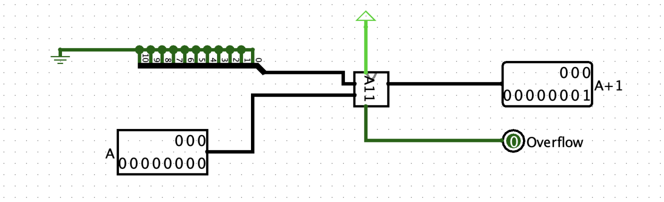


Figure 11: IncrementExponent

## 5.16 Priority Encoder

PriorityEncoder takes a 32-bit input and outputs the position of the first 1 in the input and add one with this.

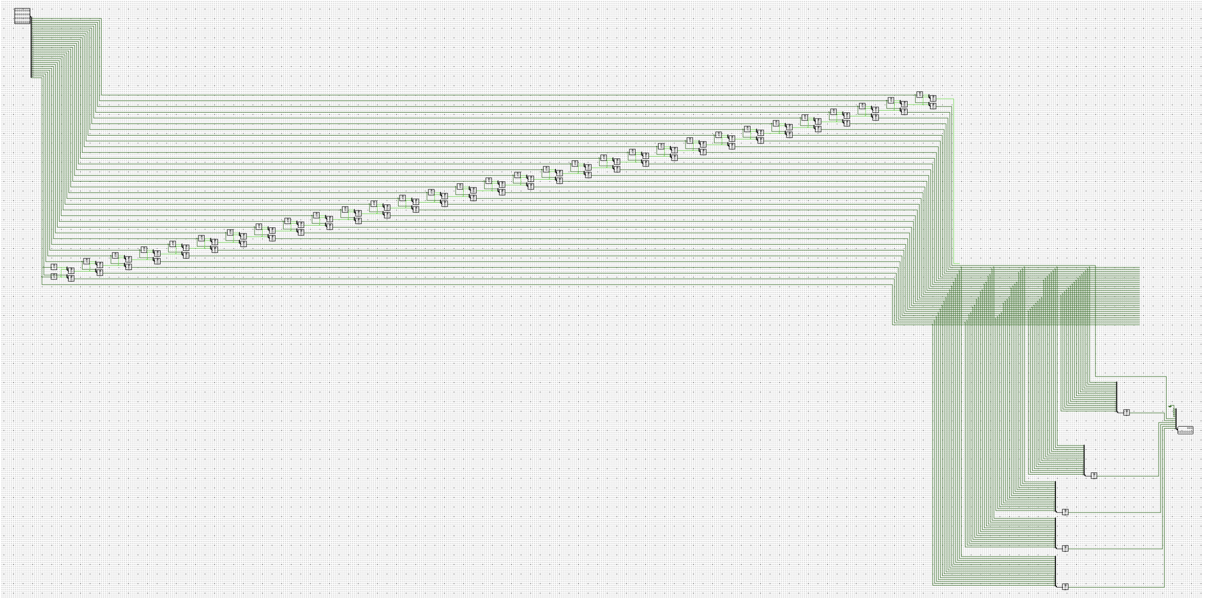


Figure 12: PriorityEncoder

## 5.17 Normalizer

When there is a carry-out, indicating that the leftmost 1 is already beyond the result fraction (we denote it as $F_{f32}$), and thus, no further right shifting is needed. In the absence of a carry-out, we must left-shift the result until the leftmost 1 is outside the 32 bits. Regardless of whether $C_{out}$ of $A_{32}$ is 1, we transmit $F_{f32}$ and $F_e$ into the Normalizer. Here, using the PriorityEncoder, we ascertain the required shift amount $D_{norm}$. We then pass $D_{norm}$ and $F_{f32}$ to the CustomLeftShifter and subtract $D_{norm}$ from $F_e$. Using a pair of MUX, we decide based on $C_{out}$ of $A_{32}$ whether to output the original value or the shifted one. Thus, we obtain the final normalized, unrounded $1 + 11 + 32 = 44$-bit result.

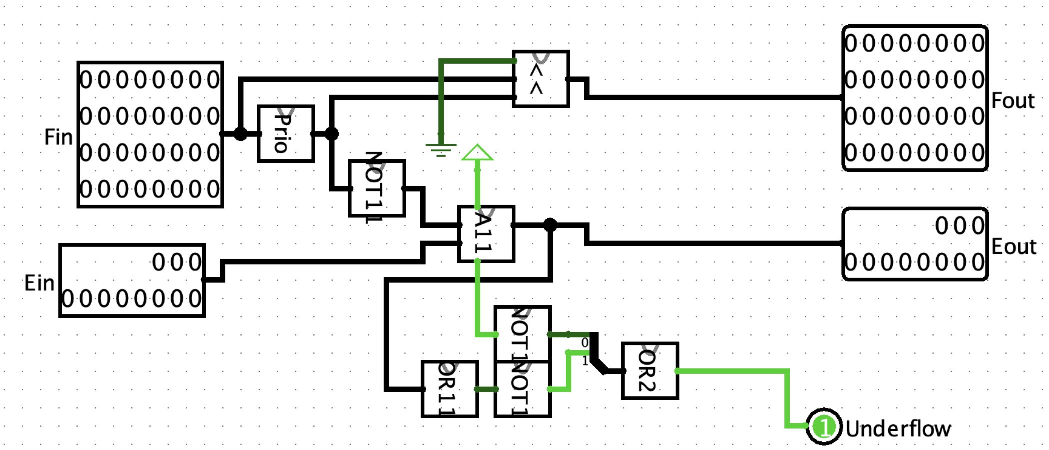


Figure 13: Normalizer

## 5.18 32-bit Zero

It gives an all-zero 32-bit output.

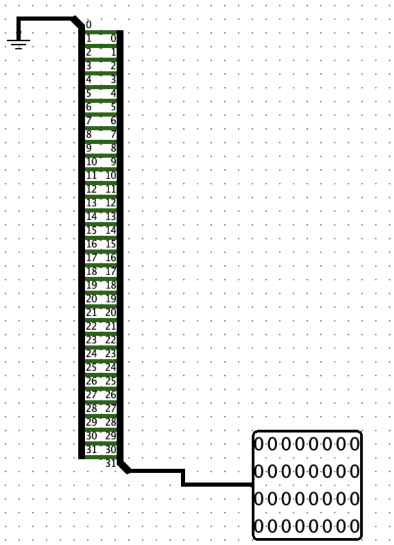


Figure 14: Zero32Bit

## 5.19 Rounder

This part rounds the mantissa. It determines the bit position in the 20-bit fraction where rounding needs to occur. Use a process, like a set of 3-bit truncators, to round the fraction based on the identified position. If the bits beyond the rounding position are significant, add 1 to the rounded fraction. Adjust the exponent based on the rounding operation, considering any carry from the fraction rounding. Check for possible overflow due to rounding and adjust the exponent and fraction accordingly. Present the final result with the sign, adjusted exponent, and rounded fraction.

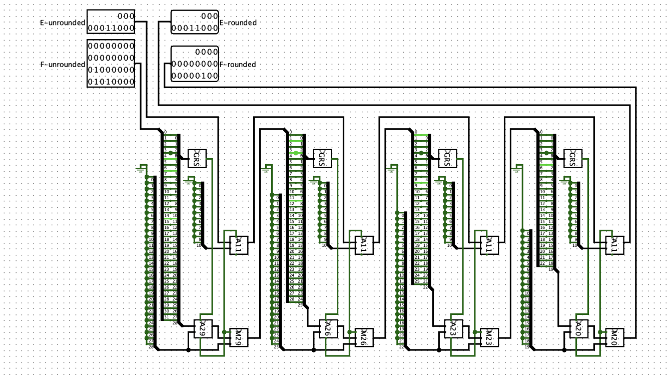


Figure 15: Rounder

## 5.20 Output Joiner

Finally, this circuit combines 3 incoming 1-bit, 11-bit, and 20-bit data into a single 32-bit output.

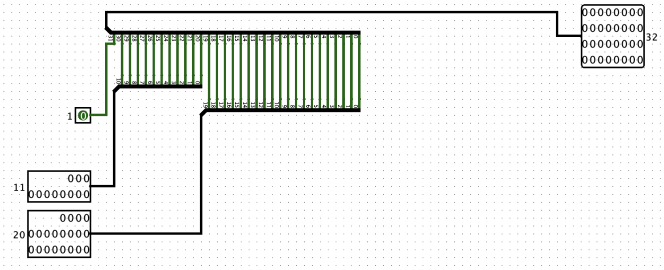


Figure 16: Output Joiner

## 5.21 Floating Point Adder

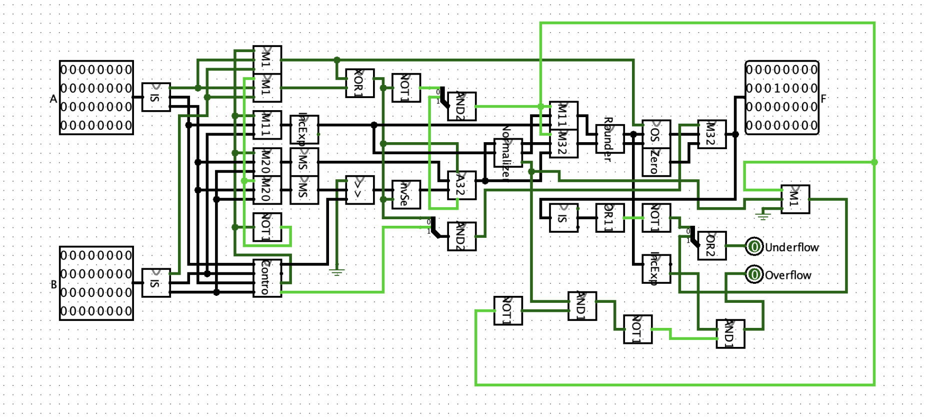


Figure 17: Floating Point Adder (FPA)

## 5.22 Full Circuit

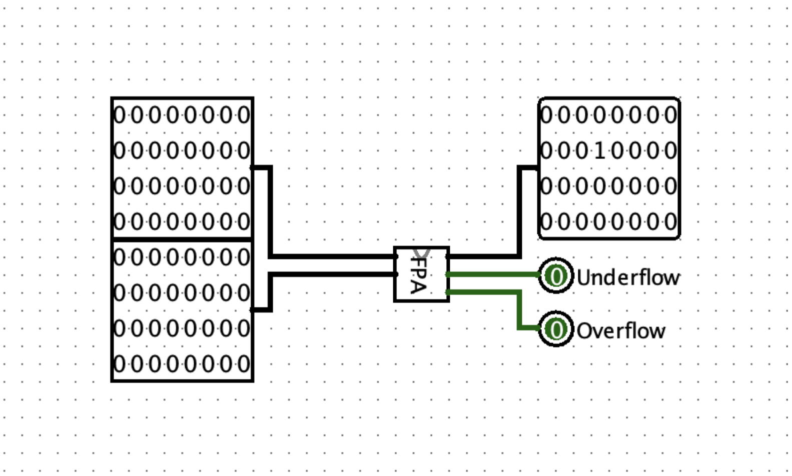


Figure 18: Complete Circuit

# 6 Flow Chart And Working Methodology

## 6.1 Flow Chart of work

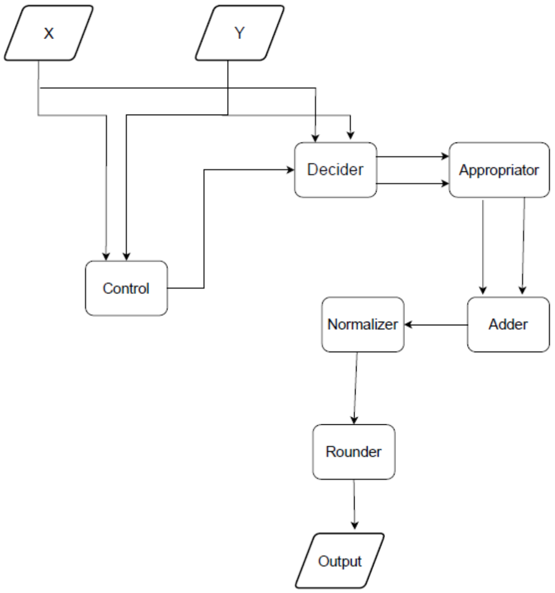


Figure 19: Flow Chart of work

## 6.2 Comparing the Exponents and Aligning Radix Point

To perform the addition of two floating-point numbers, it is essential to align their radix points. This alignment is achieved by shifting the input with the smaller exponent to the right, ensuring it aligns with the larger input. To facilitate this process, a comparator is employed to compare the exponents of the two input numbers. Subsequently, an 11-bit subtractor is utilized to calculate the difference between these exponents. The result of this subtraction operation determines the amount of rightward shifting required to align the radix points, enabling the subsequent addition of the floating-point numbers.

## 6.3 Shifting

The usage of multiplexer-based shifters facilitates all shifting operations. We employ five 32-bit multiplexers to achieve bit shifts of any arbitrary amount up to 31. These multiplexers can individually shift by 1, 2, 4, 8, and 16 bits. By combining them, we attain flexibility for shifts of up to 31 bits. However, when aligning two fractions, scenarios may arise where more than 31 shifts are necessary, resulting in all bits being set to 0. In such cases, the input to the shifter circuit should be the difference of the exponents. Specifically, the lower 5 bits are utilized for shifting only when all the upper 7 bits are clear. This condition ensures that the shift amount does not exceed 31 ($2^5 - 1$).

## 6.4 Normalization

For the normalization followed by an addition operation, we incorporated a priority encoder to identify the position of the most significant set bit, specifically the leftmost one. This involves utilizing four 8-to-1 priority encoders.

The result now spans from 0 to 2047 ($0_{th}$ to $2047_{th}$), given the 11-bit exponent and 20-bit fraction, forming a 5-bit representation (0 to $2^5 - 1$). The lower 3 bits of the result correspond to the outputs of the encoders. The upper 2 bits, along with the outputs of the encoders, are selected using their valid bits.

If the leftmost encoder receives a valid input, the other encoders are bypassed, setting the upper 2 bits to '00'. Consequently, the generated decimal numbers will range from 0 to 7. When the second encoder is selected, the upper 2 bits will be '01'; '10', and '11' for the third and fourth encoders, respectively.

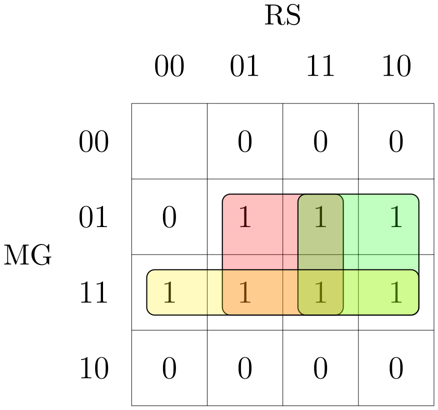
The selection bit of the multiplexer is determined by a 4-to-2 priority encoder, with inputs being the valid bits of the aforementioned priority encoders.

## 6.5 Rounding

For the addition and subtraction of mantissa, a 32-bit adder is employed. However, due to space constraints, only 19 bits can be retained, necessitating rounding in specific scenarios. The $20^{th}$ bit is designated as the Guard bit (G), and the $21^{st}$ bit is the Round bit (R). If any of the bits to the right of the Round bit is set, the Sticky bit (S) is set; otherwise, it is clear. The cases are as follows:

| **$19^{th}$ bit** | **G** | **R** | **S** | **Action** |
|---|---|---|---|---|
| X | 0 | X | X | Truncate |
| 1 | 1 | X | X | Round up |
| 0 | 1 | 0 | 0 | Truncate |
| X | 1 | 1 | X | Round up |
| X | 1 | X | 1 | Round up |

In the case where the $19^{th}$ bit is 0, no action is taken (truncation). Otherwise, rounding up is required. For simplification, a Karnaugh Map (K-map) is employed with the $19^{th}$ bit denoted as M:



| MG \ RS | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 |  | 0 | 0 | 0 |
| 01 | 0 | 1 | 1 | 1 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 |

The rounding decision is determined by the flag (flag $= G \cdot (M + R + S)$). If the flag is set, 1 is added at the $19^{th}$ position; otherwise, bits beyond the $20^{th}$ position are truncated.

## 6.6 Computing Sign Bit

When determining the sign bit, several considerations come into play. If the signs of the two inputs are the same, the sign of the output will match either of the signs of the inputs.

In cases where the signs of the inputs differ, additional factors must be taken into account. Firstly, the sign of the output from the adder can be calculated using the equation:

$$\text{sign} = (S_A \oplus S_B)\overline{C_{out}} \tag{1}$$

Here, $C_{out}$ is the carry-out of the adder.

However, this sign may not always be accurate due to the subtraction order (subtracting the input with the smaller exponent from the input with the higher exponent). In instances where the signs of the inputs are opposite, i.e., if the input with the greater exponent is negative and the other one is positive, Equation 1 gives the opposite of the correct sign. To address this, we introduce a variable called the switch. The switch bit alters the sign bit in these specific cases, and its equation is given by:

$$\text{switch} = (S_A \oplus S_B)(\text{Comp} \oplus S_B) \tag{2}$$

Here, Comp is the output of the comparator circuit ($Exp_A > Exp_B$). The formula for the actual sign bit is then:

$$\text{actualSign} = \text{sign} \oplus \text{switch} \tag{3}$$

Equation 3 is used when the signs of the inputs are different. Otherwise, the sign of the first input is directly applied to the sign of the output. This selection is made using a multiplexer.

## 6.7 Handling Zero or Small Input

If either of the inputs is zero, the other input will be passed directly to the output. Additionally, if the exponent of one of the inputs is significantly smaller than the other (a difference of more than 31), the input with the larger exponent is passed directly to the output.

### 6.8 Increasing Precision

In addition, we did not set aside a bit to capture the overflow bit. Instead, we captured the bit directly from the carry-out. Besides, in the case of subtraction, we did not use a separate bit for the sign. Instead, the carry-out of the adder was used to determine the sign of the result. If the carry-out is 1, the result of subtraction is positive, otherwise negative. It is an exception when one of the summands is zero.

## 7 ICs used with count as a chart

In this section, the integrated circuits (ICs) utilized in the implementation, along with their respective quantities, are presented in a concise chart:

| IC Number | IC Name | Quantity |
|---|---|---|
| SN74HC08N | Quad 2-Input AND | 90 |
| SN74HC32N | Quad 2-Input OR | 36 |
| SN74HC86N | Quad 2-Input XOR | 17 |
| SN74HC04N | 1-Input Hex-Inverter | 68 |
| SN74HC83N | 4-bit Binary Full Adder | 79 |
| SN74HC157N | Quad 2 to 1 MUX | 156 |

Table 2: ICs Used with Quantity

## 8 Simulator Used along with the Version Number

**Software:** Logisim
**Version:** Logisim-win-2.7.1

## 9 Discussion

For this assignment, our objective was to design a 32-bit FPA circuit capable of performing the addition of two floating point numbers operations. The software implementation presented challenges, particularly in strategically planning the placement of various modules.

- The entire setup was built using 7400-library integrated circuits, each serving specific functions.

- Our entire circuit is combinatorial, meaning it doesn't use any memory components like registers or flip-flops—just basic logic gates.

- To ensure both addends have the same exponent, we used a Custom Right Shifter. This device checks each bit of the shift amount value and decides whether to send the shifted or unshifted input value. For shift amounts greater than 31, we take the OR of the seven most significant bits, sending a full zero value if the result is 1.

- Inside the Normalizer, a Priority Encoder determines the position of the leftmost 1 in the 32-bit fraction. This value is used in CustomLeftShifter to shift the fraction by that amount and is subtracted from the exponent value.

- In the Rounder, we have cascaded 3-bit truncators, each truncating 3 bits from the fraction input. The exponent value is also rounded accordingly using adders.

- To avoid data loss and preserve even the lowest bit of information We considered the carry-out after addition as the leading bit of the result, allowing us to shift the sum leftwards without losing data.

- We handled addition or subtraction decisions using control, using the sign bit of the larger addend as the sign bit of the sum. For subtraction, the addend with the smaller absolute value is subtracted from the larger one.

- Rounding was meticulously done using cascading rounder devices.

- We put together individual components with smaller ones, although we didn't focus much on minimizing the number of ICs used.

- MUXs were extensively used for selecting and sorting values. Carry-out bits from different stages acted as selector bits for MUX in numerous instances.

- The circuit assumes that the addends will always be in normalized form, meaning no Zero, denormalized, NaN, or infinity values should be provided as inputs.