CSE 306
Computer Architecture Sessional

Assignment-2: 32-bit Floating Point Adder Simulation

Section - A1
Group - 03

Group Members:

i 2005003 - A.H.M Towfique Mahmud

ii 2005004 - Md Zim Mim Siddiqee Sowdha

iii 2005006 - Kowshik Saha Kabya

iv 2005018 - Munzer Mahmood

v 2005019 - Jarin Tasneem

January 21, 2024

1 Introduction to Floating Point Adder

A floating point number is a computer representation of a real number that has a set number of digits before and after the decimal point. The digital circuit that performs the addition of floating point values is called a floating point adder. It can be applied to numerical values that are either too big or too tiny to be precisely represented by integer representations.

The floating-point representation of a numerical value has three essential components: the sign bit, exponent field, and mantissa field. The sign bit indicates the positive or negative nature of the number. The exponent field plays a key role by allowing the multiplication of the significand with a power of the base, using a fixed number of bits in a biased form. This approach broadens the range of representable values. To determine the actual exponent, it is necessary to subtract the predetermined bias from the bits stored in the exponent field. Additionally, the mantissa represents the fractional part of the number, encompassing the digits following the decimal point. In our specific implementation, the sign bit, exponent field, and mantissa occupy 1, 11, and 20 bits, respectively. Consequently, the exponent bias for this system is calculated as $2^{11-1} - 1 = 1023$. This bias adjustment is critical for accurately interpreting the exponent value during floating-point arithmetic operations.

In the process of adding two numbers using a floating-point adder, the first step is aligning their decimal points. Subsequently, their mantissa are added together. The resulting exponent is fixed, leading to necessary shifts and adjustments, such as incrementing or decrementing, during the normalization and rounding procedures. The sign of the result is determined by the signs of the two numbers being added.

Floating-point adders play a pivotal role in diverse domains, offering precise handling of real numbers with a broad range of magnitudes. In scientific simulations and engineering calculations, these adders contribute to solving intricate mathematical models and simulating physical phenomena with heightened accuracy. Financial modeling benefits from their ability to perform exact calculations of interest rates and investment returns. In computer graphics, floating-point adders are crucial for rendering realistic scenes, while data analysis in fields like data science relies on their high precision. From high-performance computing and signal processing to applications in weather forecasting, medical imaging, and astrophysics, the versatility of floating-point adders extends across numerous domains, providing essential support for tasks that demand accurate numerical computations.

2 Problem Specification

The assignment involves the task of designing 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 bits	20 bits

Table 1: Problem Specification

3 Description and Circuit Diagram of Modules

Several libraries have been developed to enhance modularity in the design of the floating-point adder. Descriptions and applications of these libraries are provided below.

3.1 Preprocessor Module

The preprocessor library(PrepocessorCkt.circ) has several segments dedicated to helping with the prerequisite tasks of the addition of the significand. This involves comparing the exponent parts of the floating point numbers, shifting the significand accordingly, and outputting the shift amount based on this calculation. Before explaining the procedure of this step some points to be noted are following: Among the numbers which has a greater exponent value we are calling it A and the other B. So hereafter if we mention A that means the number with the greater exponent value. In case the exponent values of the 2 numbers are the same it doesn't matter which one is called A. Now let us explain the working principle of this portion.

3.1.1 Comparison of Exponents

By using an 11-bit adder, we are finding the difference between the two exponents. The C_{out} determines the greater exponent. If the C_{out} is 1 that means the number that was subtracted was smaller and we assign it to B and the other one to A. Also, the sum gives the exact difference between them which will help us to determine the shift amount later. If the C_{out} is 0 then we assign them to the alternative identities. Also, we used another 11-bit adder to find the 2's complement of the difference as the output from the adder is the 2's complement of the actual difference between them. Please note that the assignment process was not only for the exponent but also for the whole numbers even the sign bits and this assignment procedure was done simply by using a lot of 2X1 MUXs.

3.1.2 Calculation of Shift Amount

Now that we have determined the difference between the exponents of the two numbers, we can calculate the amount to be shifted for the significand part of B. As we are using the 32-bit arithmetic process for the significand we are using a 31-bit shifter. When the difference is under 32 the shifter simply left shifts significand(B) by the amount calculated earlier. In case the difference is more than 31 i.e. greater or equal to 32, then we are shifting significand(B) by 31 bits. This will not pose a problem as we will only take the significant

20 bits from the 32 bits. During this process, the 1 in the LSB will be ignored. This choice of shamt (shift amount) from 31 and the difference was selected using five 2X1 MUXs as 5 bit is needed to express the shamt.

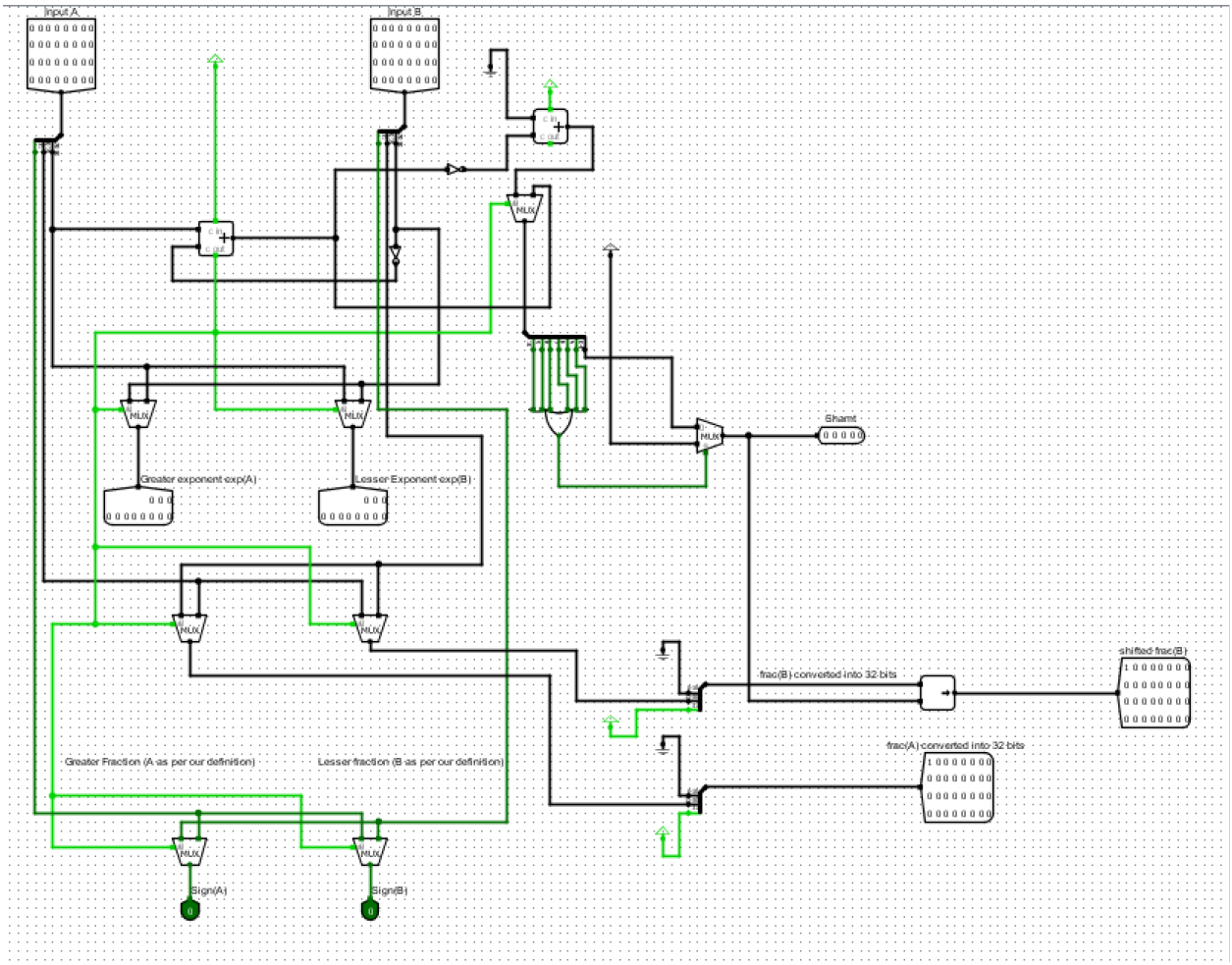

Figure 1: Preprocessing Circuit: Comparison of Exponents and Shifting

3.1.3 Shifting

Now that we know the shamt for significand(B) to be shifted, we can now simply shift that using the inbuilt shifter of Logisim. This type of shifter is very simple to build. For a shamt of 5 bits, this shifter is a combination of $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, and $2^4 = 16$-bit shifter. These shifters can be built by using 32 2X1 MUXs.

3.2 Significand Adder Module

Here we have implemented a 32-bit adder to calculate the addition or subtraction of two significands (SignificandAdder.circ). We have a total of four inputs here. Let us call them A and B. The sign bits and the fractional parts of A and B will be the input to this library. Notice that the fractions will be padded to 32 bits from the previous modules. Two adders have been used for two specific purposes. The first one is used simply for addition and subtraction. We control its operation with the help of a control bit that is derived from the xor of the sign bits. This bit acts as the carry input. The reason behind the second adder is to prevent any number from being in 2's complement form as in a floating point adder any negative number is represented using 1 as a sign bit. We have used 2 adders, 3 xor gates, 1 and gate, 1 not gate, and 2 special multiplexers. The multiplexers are used to create a 32-bit output from a single-bit input. The inputs are labeled as 'sign(A)', 'sign(B)', 'frac(A)' and 'frac(B)'. Output is labeled as 'Significand to be shifted'.

The carry-out from the first adder along with the control bit of the first adder(carry-in) will determine what action the second adder will take. For example, if subtraction between A and B is performed and A < B then the output will be in 2's complement form and the second adder takes 2's complement of that number again. The output of this module will be sent to the next module for shifting purposes.

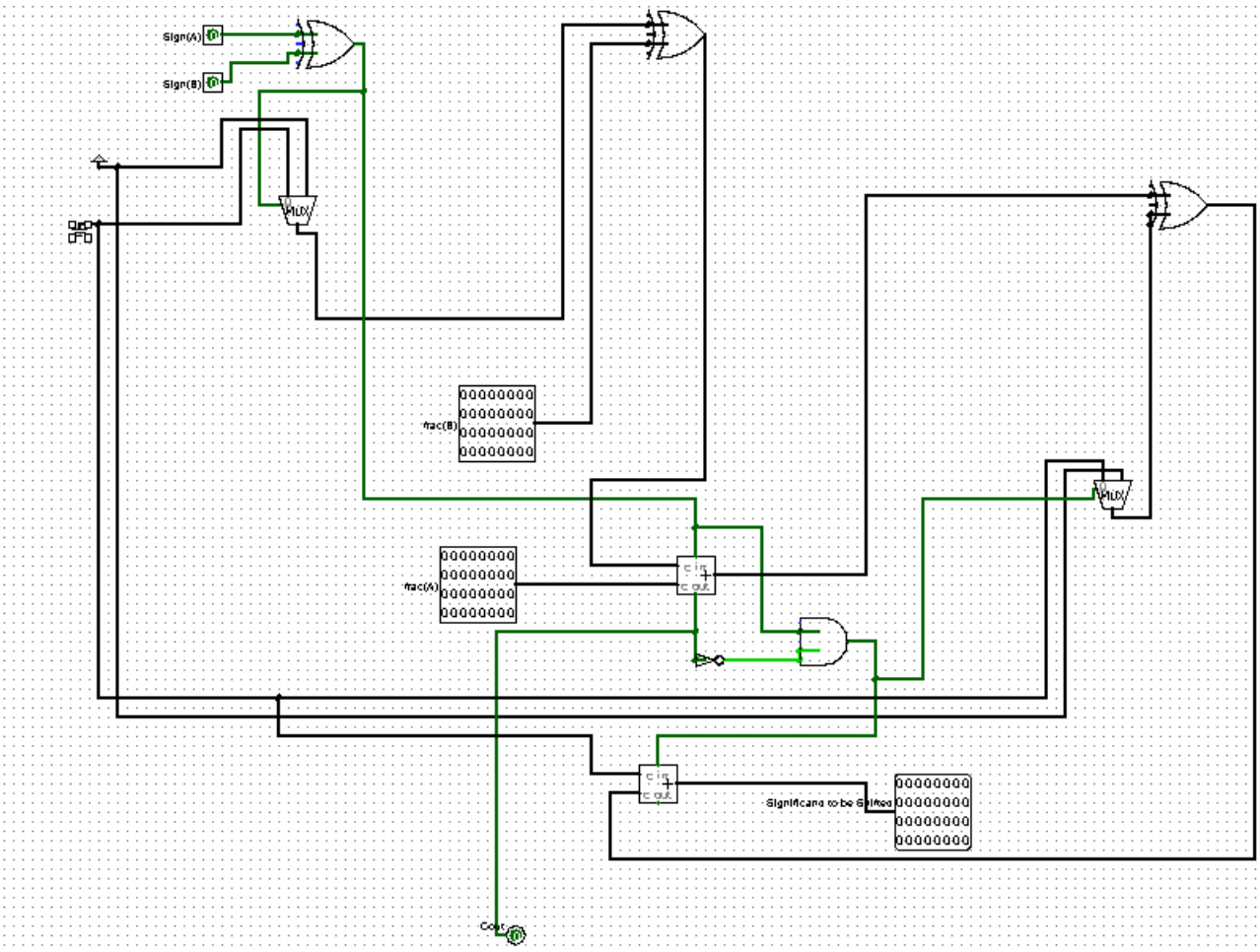

Figure 2: Significand Adder

3.3 Significand Normalization Module

In this module, we have normalized the 32-bit significand result coming from the previous significand adder module. To normalize the 32-bit input, at first, we need to figure out whether we have to perform a right or left shift.

To perform a left shift, the input 32-bit significand goes through a 32X5 priority encoder to figure out the position of 1st '1' in the number. This position works as the amount of left shift and becomes an input to a MUX that selects this shift amount when the left shift is needed. To perform a right shift, we have a right shifter available that shifts the input accordingly when needed.

From the previous adder module, we get a C_{out} that acts as an enable pin to our 32X5 priority encoder, the selector for our MUX dedicated for a left shift, and also as the shift amount of our right shifter. At once C_{out} can only contribute to the left-shifter being active or the right-shifter.

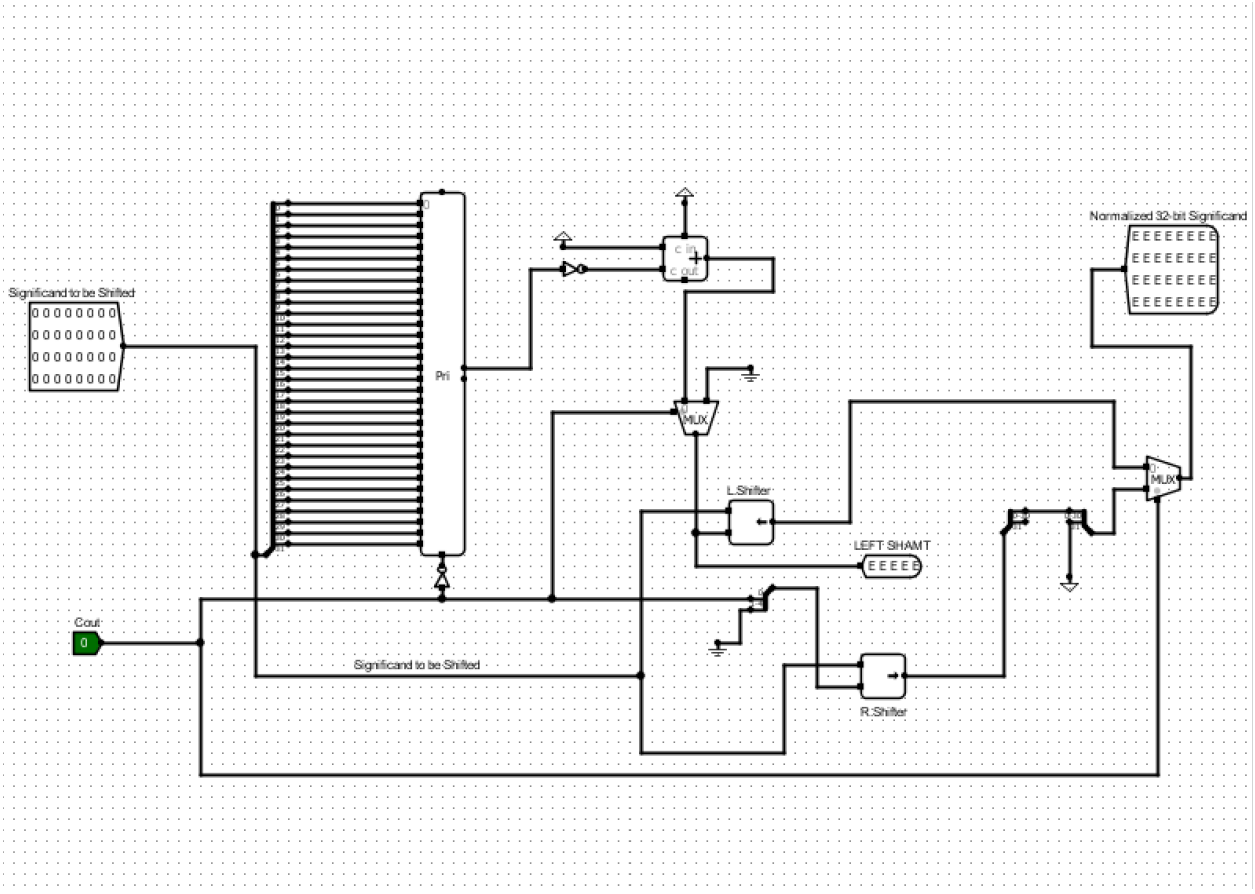

Figure 3: Significand Normalizer

There can be two cases. Since C_{out} is the selector of the 2x1 MUX used here,

- If C_{out} is 0, the 0th-bit input from the 2X1 MUX that is the left-shift amount is selected

and the left shifter shifts the input by that amount. At the same time, the amount for right-shift is 0, so the right shifter doesn't have a role here and the significand becomes normalized after being left-shifted.

- If C_{out} is 1, the 1st-bit input from the MUX that is 0, is selected and the left-shifter shifts the input by the amount 0, basically no left-shift happens. At the same time, the right shift amount is determined by C_{out} that is 1, and the significand ultimately gets normalized after being right shifted only once.

3.4 Exponent Normalization Module

After normalizing the significand, in this module, we normalize the greater exponent produced by the preprocessor module and get the normalized exponent as well as the underflow and overflow flag.

The preprocessor module compares the two exponents of two inputs and then tells us which one is the greater one. We take that greater exponent as well as the shift amount from the previous module as our input. Using a 11-bit adder we subtract the shift amount from the greater exponent and use the output of this adder to get our final normalized exponent. We use another 11-bit adder that takes one input from the result of our previous adder and another input from the ground. The C_{in} of this adder comes from the previous module C_{out} and adds 1 while the significand was normalized doing a right shift.

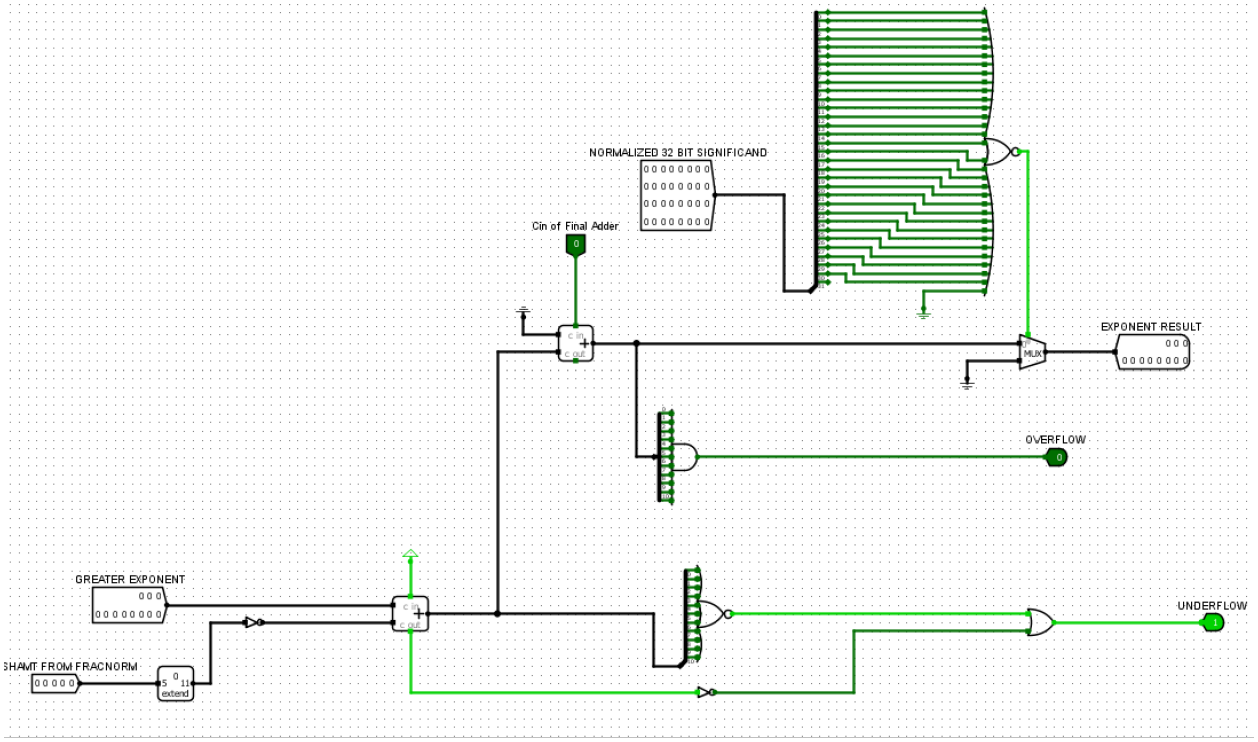

Figure 4: Exponent Normalizer

The result and the C_{out} of the first adder give us the underflow flag while the final normalized exponent reveals the overflow flag. If the normalized exponent <=0, we get an

underflow flag and if the exponent >= 2^{11} -1, we get an overflow flag.

3.5 Rounding Module

- Input: 32-bit normalized significand
- Output: 32-bit rounded significand and 1-bit carry-out

This module (RoundingCkt.circ) is designed to derive the desired 20-bit fraction from the extended 32-bit significand received from the adding and normalizing module. In this 32-bit number, the most significant bit (bit 31) is always 1 and it represents the whole number part of the floating point number. Bits 30-11 will hold the desired 20-bit fraction. Bits 10-0 are used for rounding. We followed the procedure below to carry out the rounding of the input:

- If bit 10 (Guard bit) is 0, there is no need for rounding - bits 10-0 will be truncated.
- Else if bit 10 is 1, we look at bits 9-0.
  - If at least 1 bit among bits 9-0 is 1, we will round up the fraction by adding 1 to it.
  - Else if all of bits 9-0 are 0, then we will round to even. To do so, we look at bit 11.
    - If bit 11 is 0, there is no need for rounding - bits 10-0 will be truncated.
    - Else if bit 11 is 1, we will round up the fraction by adding 1 to it.

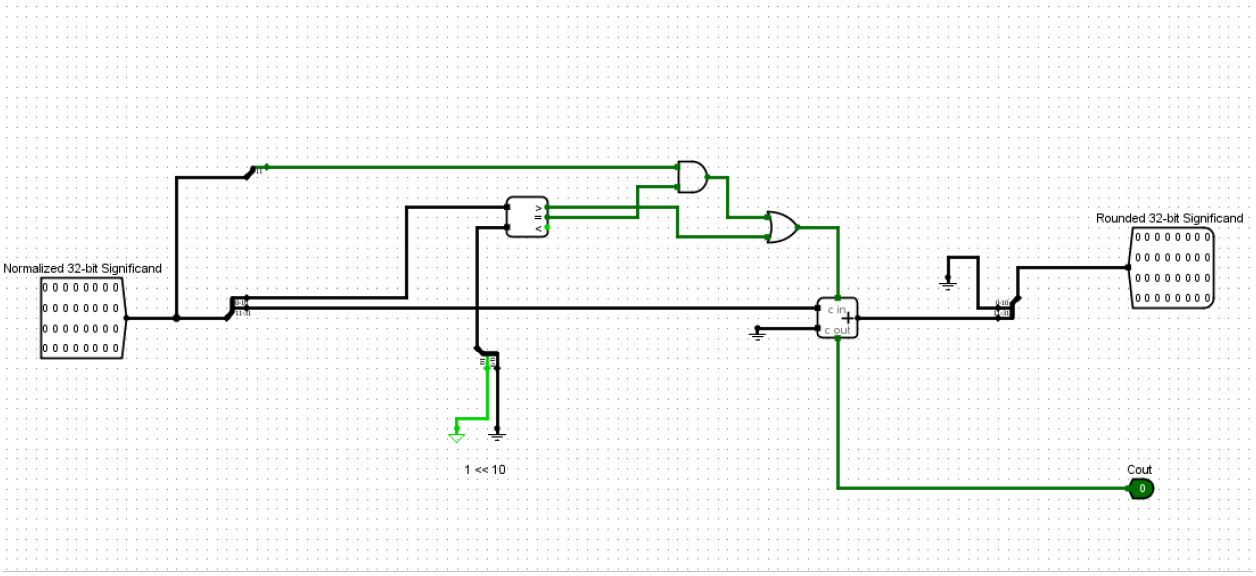

Figure 5: Rouding Circuit

From the procedures, it can be easily deduced that we will round up the fraction if either of the following two conditions is satisfied:

- Condition 1: Bits 10-0 is greater than 1 ¡¡ 10 (one 1 followed by 10 0's)

- Condition 2: Bits 10-0 is equal to 1 ¡¡ 10 (one 1 followed by 10 0's) and bit 11 is 1

To implement the logic above, we first compared the bits 10-0 of the input with the fixed value 1 << 10 (one 1 followed by 10 0's) using an 11-bit comparator. We calculated the logical AND of the "=" output of the comparator with the bit 11 of the input - this bit is 1 if and only if Condition 1 above is satisfied. We then calculated the logical OR of this bit with the ">" output of the comparator - this result covers both the conditions stated above. This result is sent as the carry-in for a 21-bit adder.

The inputs for the 21-bit adder are bits 31-11 of the input significand, and all 0's. If carry-in is 1, then an increment takes place - otherwise, nothing happens. We take the resultant 21 bits, pad it with 11 lower bits (all 0's), and set it as the 32-bit rounded output for the module. Moreover, we also send the carry-out of the adder as an output.

Note that, if the carry-out is 1, it means the answer is needed to be normalized again. Therefore, the 32-bit output and the 1-bit carry-out will pass through a significand normalization module and an exponent normalization module once again.

3.6 Floating Point Adder

This module (A1_Group3.circ) is the final module that combines other modules to completely implement a floating point adder. The circuit takes two floating point numbers as input, gives a resulting floating point number, and checks for overflow and underflow.

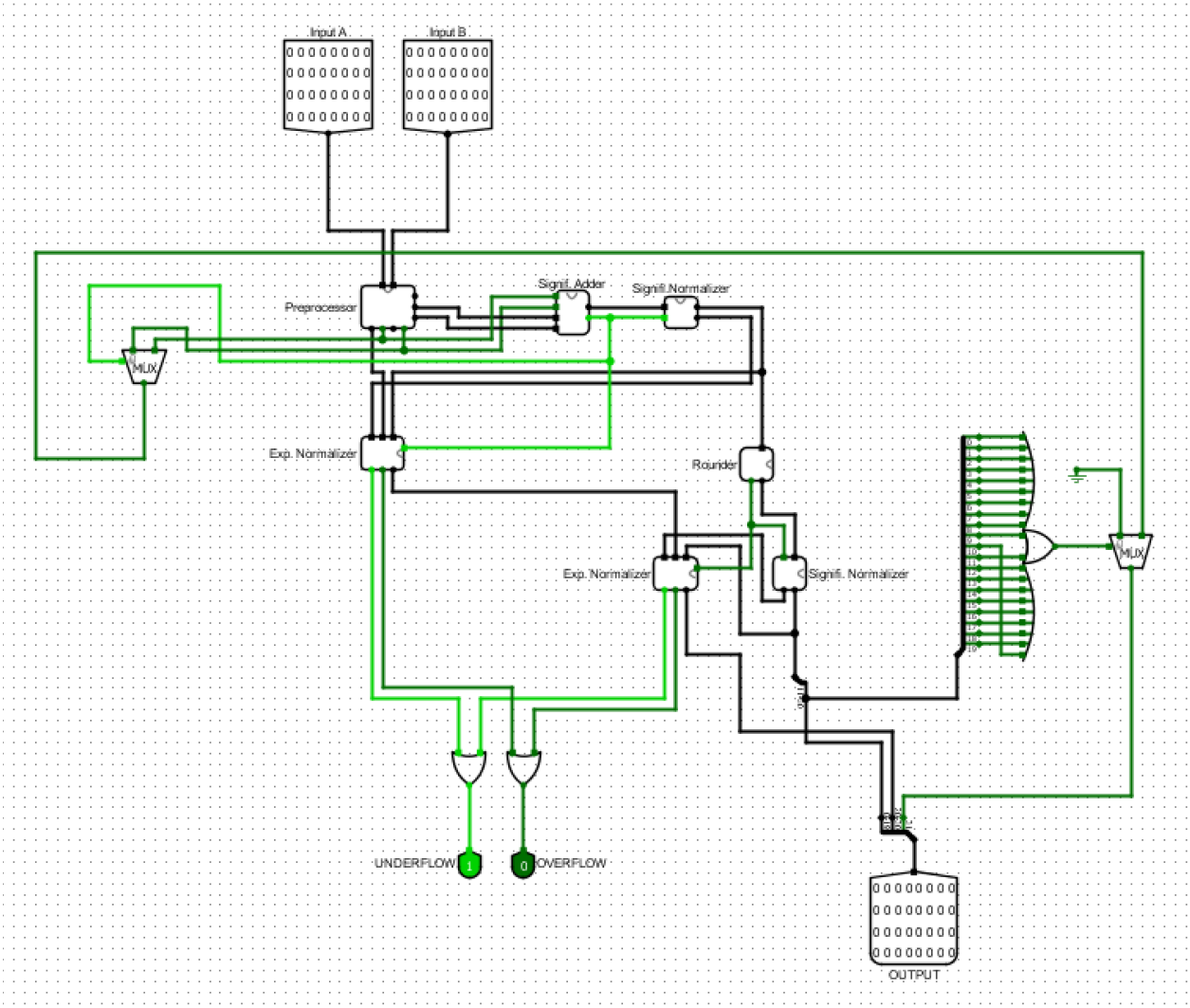

Figure 6: The FPA

3.7 Third Party Libraries

Third-party libraries 7400-lib.circ and logi7400ic.circ are used to incorporate 7400 series ICs in the floating point adder implementation.

4 Flowchart of the Addition/Subtraction Algorithm

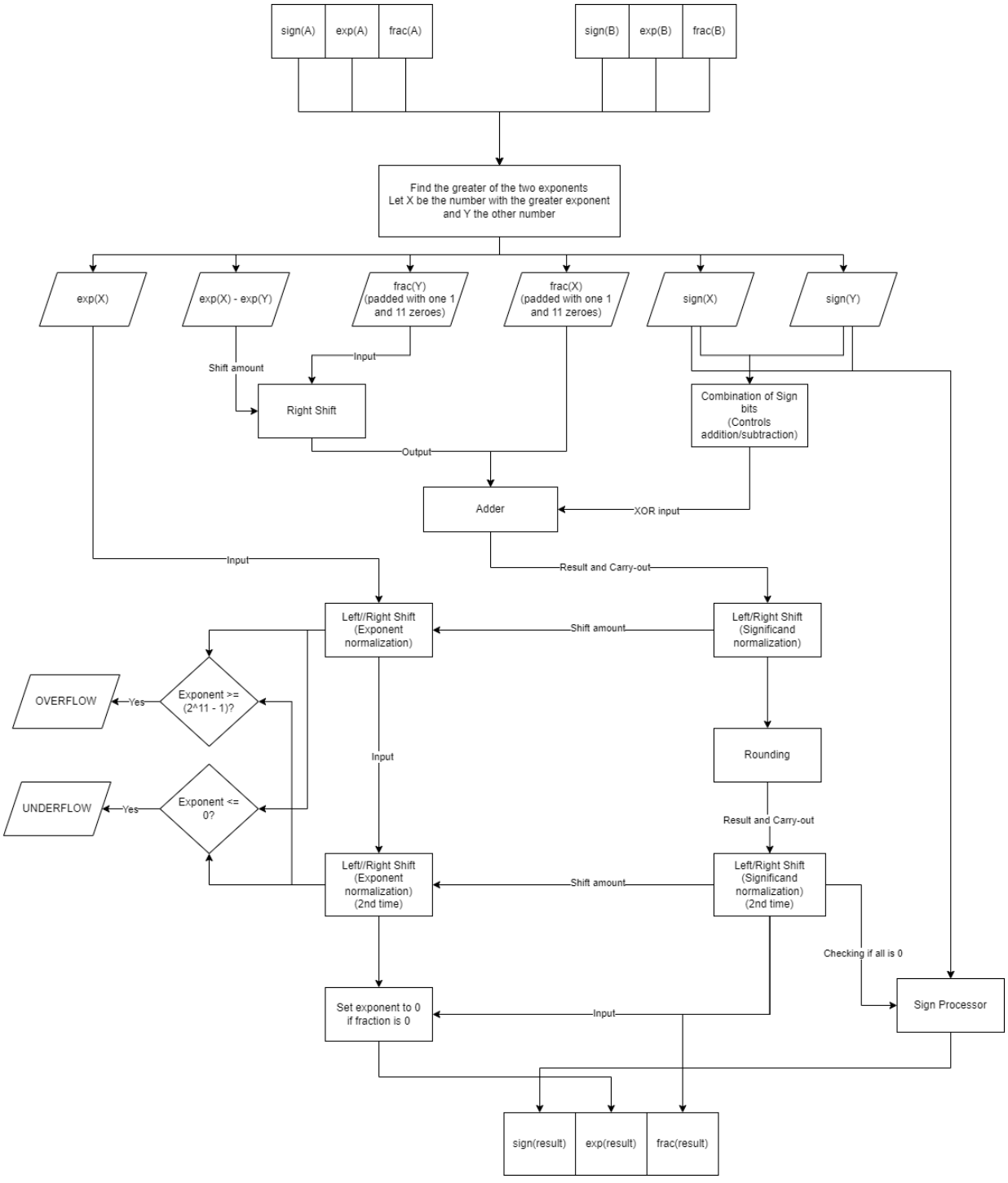

Figure 7: Flow chart of the adder

5 High-Level Block Diagram of Floating Point Adder

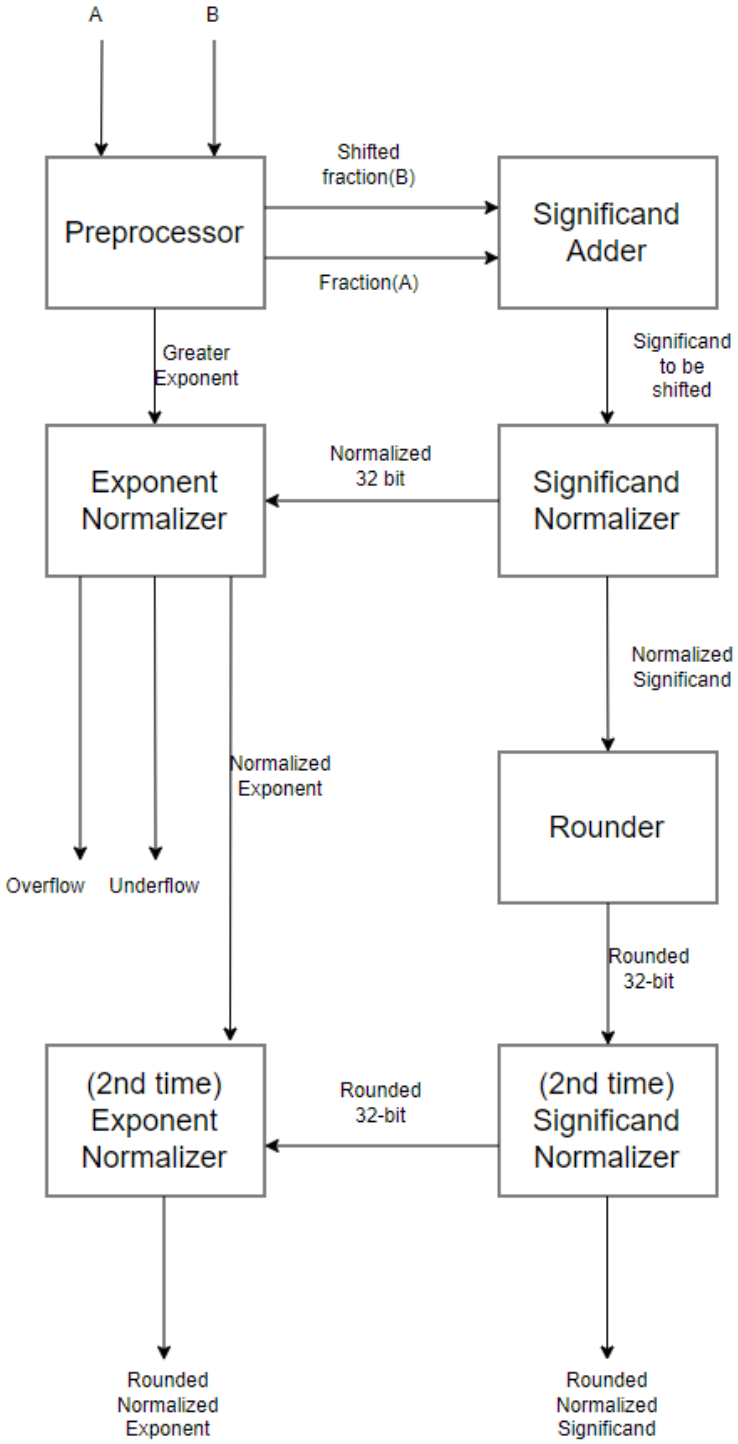

Figure 8: Block Diagram of Floating Point Adder

6 ICs Used with Count as a Chart

IC	Quantity
IC 7404	10
IC 7408	6
IC 7432	28
IC 7483	43
IC 7486	16
IC 74157	59
Total	162

Table 2: ICs Used with Quantity

NOTE: Bit Shifter, Priority Encoder, and Comparator modules were used from the built-in library of Logisim, hence proper IC count for them was not feasible.

7 Simulator used Along with the Version Number

Logisim 2.16.1.4 has been used for simulating the floating point adder circuit.

8 Discussion

In this assignment, we have successfully implemented a floating point adder (FPA) for 32-bit floats. We divided the overall process of adding two numbers into five major modules, and each individual in our group worked on one specific module.

While normalizing the significand through shifting left, the primary challenge was to find the first 1 from the left. We achieved this by using a priority encoder. We also noticed that whenever we needed to shift right here, it would occur only once. Therefore, we designed the module in a way so that if the left shift is not necessary, then without any other prior calculation the significand will be right shifted. This saved us extra additions to the circuit.

Apart from straightforward addition, we were instructed to enable rounding in the design. To ensure that, throughout the whole work cycle of the FPA, we considered the significand part to be a 32-bit number instead of the given 21-bit version. This ensured a higher precision in the result. Before terminating a process and producing an output, we used proper rounding techniques to produce the output in the desired format.

For overflow and underflow flags, we deduced that working with the exponents was sufficient. Necessary checking was done in every module where an underflow or overflow situation may occur. Moreover, we handled the sign bit of the result separately, using the output derived from the significant adder.

Thorough testing was done independently for each module, and then once again after the merging was done. Along with random cases, we were especially careful not to miss any corner cases. The sign and exponent portion of the result were properly handled when the result became absolute zero.

The implementation of the FPA helped us understand the underlying workings of floating point addition. We were also able to apply our theoretical knowledge of digital design and divide a large task into smaller, manageable portions.

9 Contribution

i **2005003**

- Logical design of the preprocessor circuit
- Software implementation of the preprocessor circuit
- Checking and debugging the preprocessor circuit and the final circuit
- Respective section in the assignment report

ii **2005004**

- Logical design of the rounding circuit
- Software implementation of the rounding circuit
- Checking and debugging the rounding circuit and the final circuit
- Respective section in the assignment report

iii **2005006**

- Logical design of the significand/fraction normalization circuit
- Software implementation of the significand/fraction normalization circuit
- Checking and debugging the significand/fraction normalization circuit and the final circuit
- Respective section in the assignment report

iv **2005018**

- Logical design of the significand/fraction adder circuit
- Software implementation of the significand/fraction adder circuit
- Checking and debugging the significand/fraction adder circuit and the final circuit
- Respective section in the assignment report

v **2005019**

- Logical design of the exponent normalization circuit
- Software implementation of the exponent normalization circuit
- Checking and debugging the exponent normalization circuit and the final circuit
- Respective section in the assignment report