BANGLADESH UNIVERSITY OF ENGINEERING & TECHNOLOGY

DHAKA, BANGLADESH

Course No. CSE 306

Course Title: Computer Architecture Sessional

Experiment No. 02

Name of the Experiment: Design and Implementation of a 32-bit Floating-point Adder

Date of Submission: 08/01/2023

Lab Section: A2

Lab Group: 04

Group Members: 1905035

1905043

1905045

1905050

1905060

Introduction:

Floating point adder is a combinational circuit which takes two floating point inputs and gives a floating-point output. It's implementation requires some basic n-bit adders, MUX, shifters, comparators and other basic gates.
Floating point adder is designed to perform floating point arithmetic, which is the most used way of approximating real number arithmetic for performing numerical calculations on modern computers. The universally accepted Floating Point representation is currently the IEEE 754 format.
The floating-point number representation is based on the scientific notation. In the scientific notation, the decimal point is not set in a fixed position in the bit sequence, but its position is indicated as a base power.

The floating-point representation of a number consists of four parts.

1. **Sign bit:** This is represented using a single bit, which indicates whether the floating-point number is positive or negative.
2. **Significand:** Significand sets the value of the number. To pack even more bits in the significand, IEEE 754 format makes the leading 1 of normalized binary number implicit. Hence the number is actually 24-bits long (Hidden 1 bit, and fraction 23 bits) in case of single precision numbers, and 53 bits long (Hidden 1 bit and fraction 52 bits) in case of double precision numbers.
3. **Exponent:** It contains the value of base power (biased). In Single precision floating point representation, the exponent is 8 bits, and in double precision floating point representation, the exponent is 11 bits.
4. **Base:** The base (or radix) is implied. It's common for all the binary representations, which is 2.

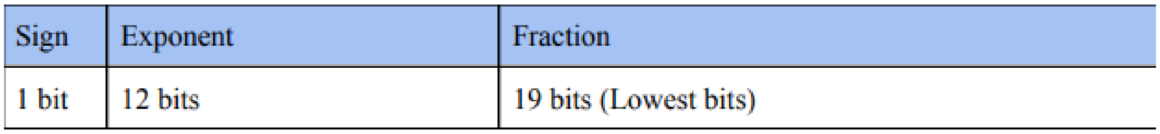
Generally, a floating-point number consists of 32 bits. We call this number as *Single Precision Number*. If a floating point number consists of 64 bits, we call this number as *Double Precision Number*. Here in this experiment, we are given the assignment to construct a 32-bit floating-point adder.

The step in designing a floating-point adder consists of:

1. Extracting sign, exponent and fraction of both the floating point inputs.
2. Handling special cases. These cases include:
   - **Operations with any of the operand equal to zero**. This case happens when exponent and fraction both are valued at 0.
   - **Operations including ±∞.** This case happens when exponent is all 1, and fraction consists entirely of 0s.
   - **Operations including NaN**. This case happens when exponent consists entirely of 1s, and fraction does not consist entirely of 0s, but is non-negative.
   - **Operations including Denormalized Numbers**. This case happens when exponent consists entirely of 0s, and fraction is non-zero. For simplicity, we ignore this case in this experiment.
3. Compare the exponents and find out the smaller exponent. We also need to find out their difference, which will indicate the required right shifting amount (*Larger exponent – Smaller Exponent)* amount to ensure that both numbers have the same exponent value.
4. Do necessary operations (2's complement) based on the sign bit of the inputs, and determine the output sign bit and do necessary operations to output the value.
5. Detect overflow, and normalize the output .

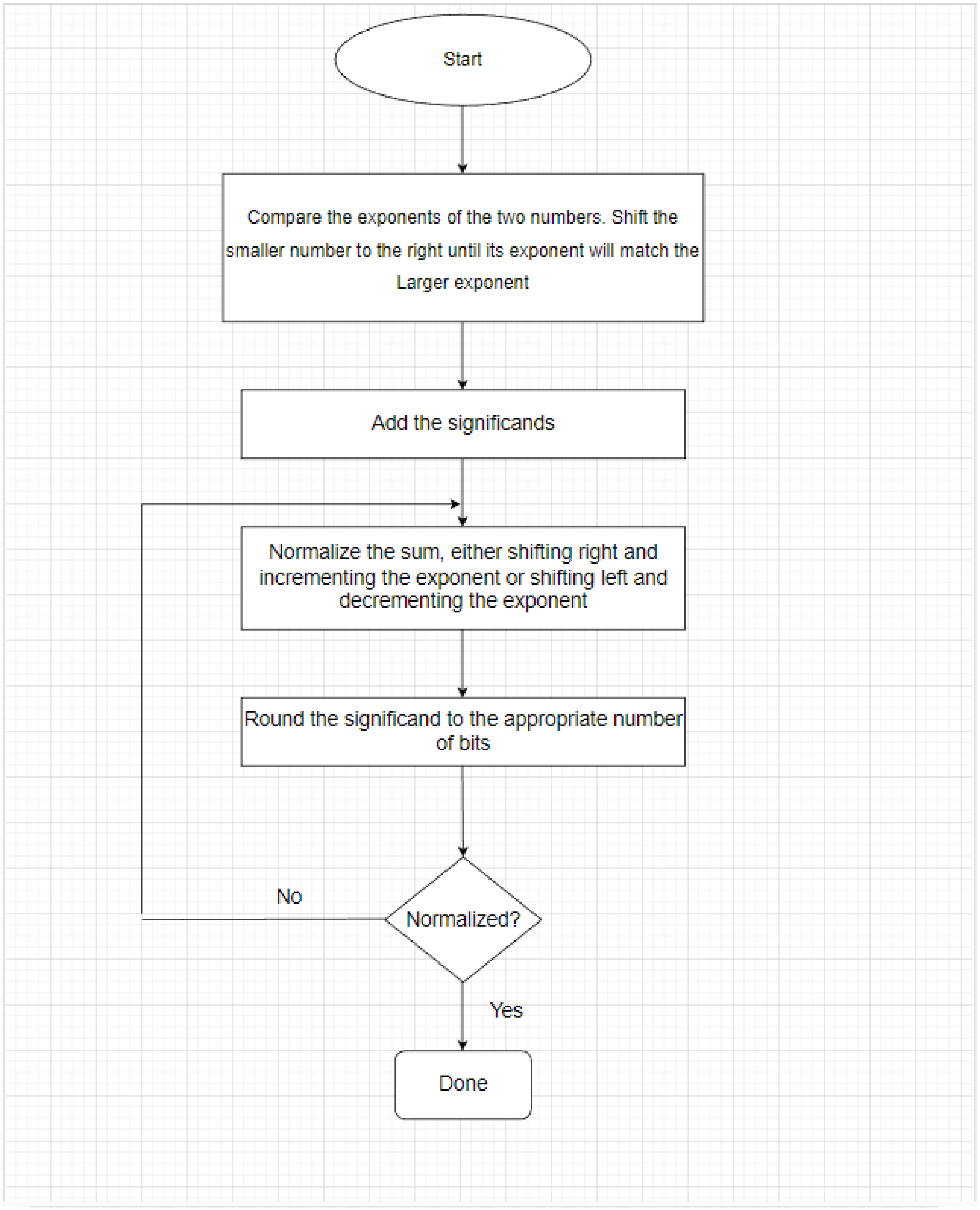
Problem Specification:

In this experiment, we were tasked to design a floating-point adder circuit which takes two floating point input and produces a floating point output. The input and output both are 32-bits with the following representation:



Sign	Exponent	Fraction
1 bit	12 bits	19 bits (Lowest bits)

Flowchart:


Start
Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent will match the Larger exponent
Add the significands
Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Round the significand to the appropriate number of bits
No
Normalized?
Yes
Done

Block Diagram:

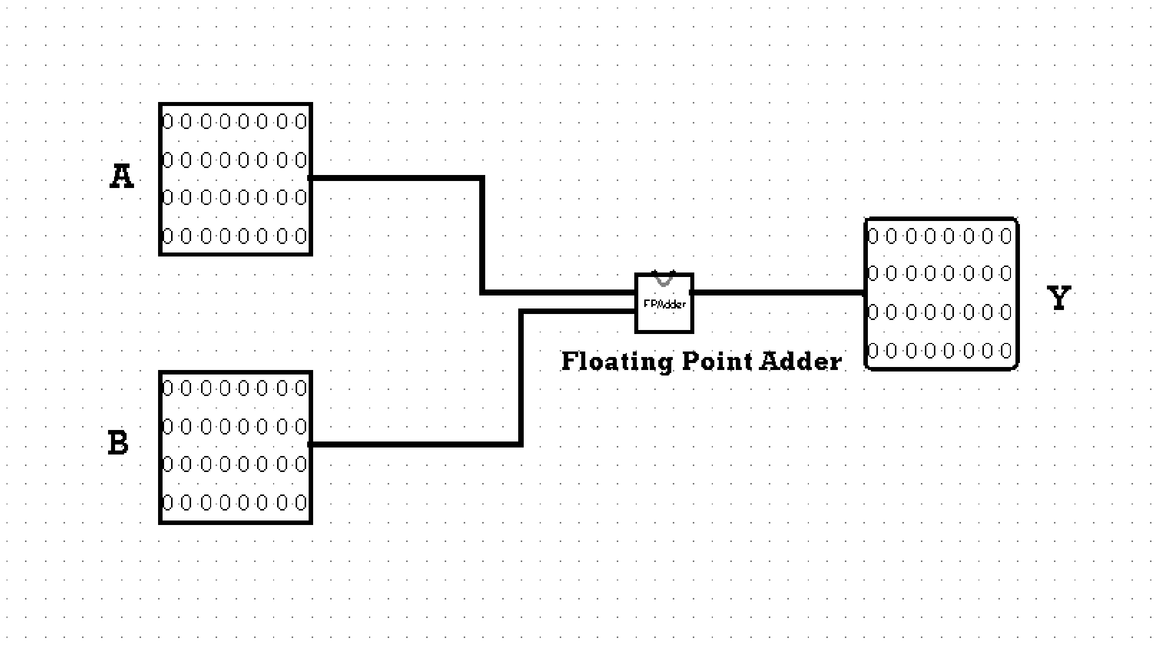


Figure 01: Block Diagram of Floating-Point Adder

Detailed Circuit Diagram:

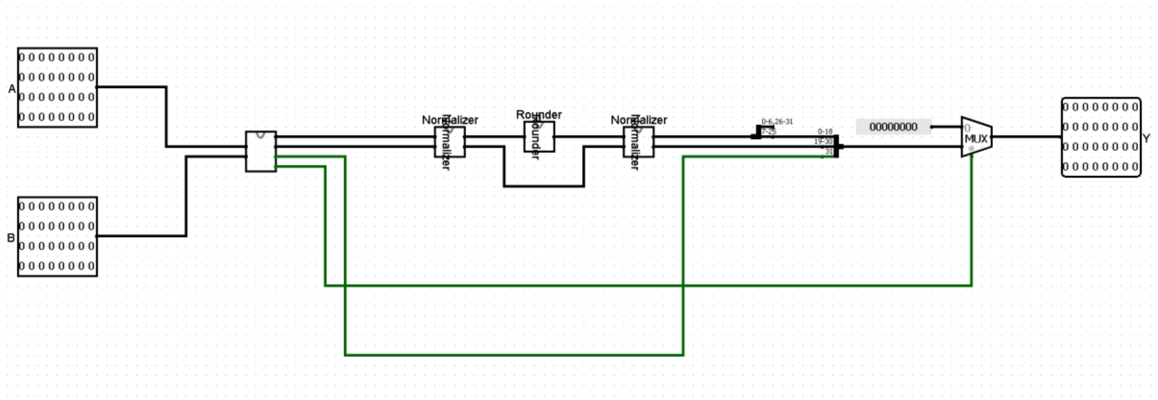


Figure 02: Circuit Diagram of Floating-point adder

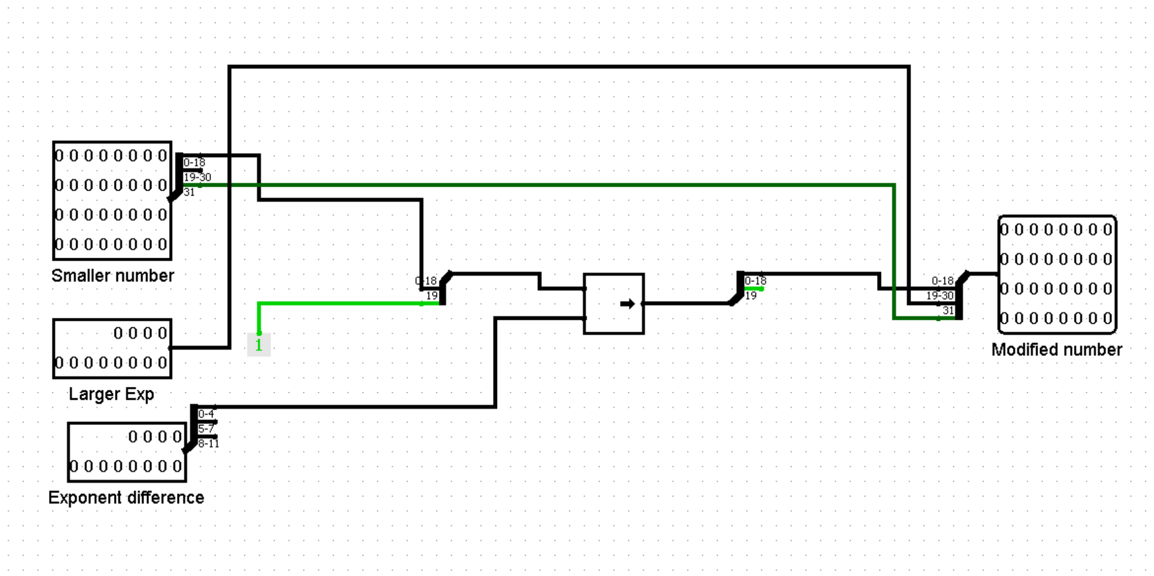


Figure 03: Fraction Alignment circuit of Floating-point adder

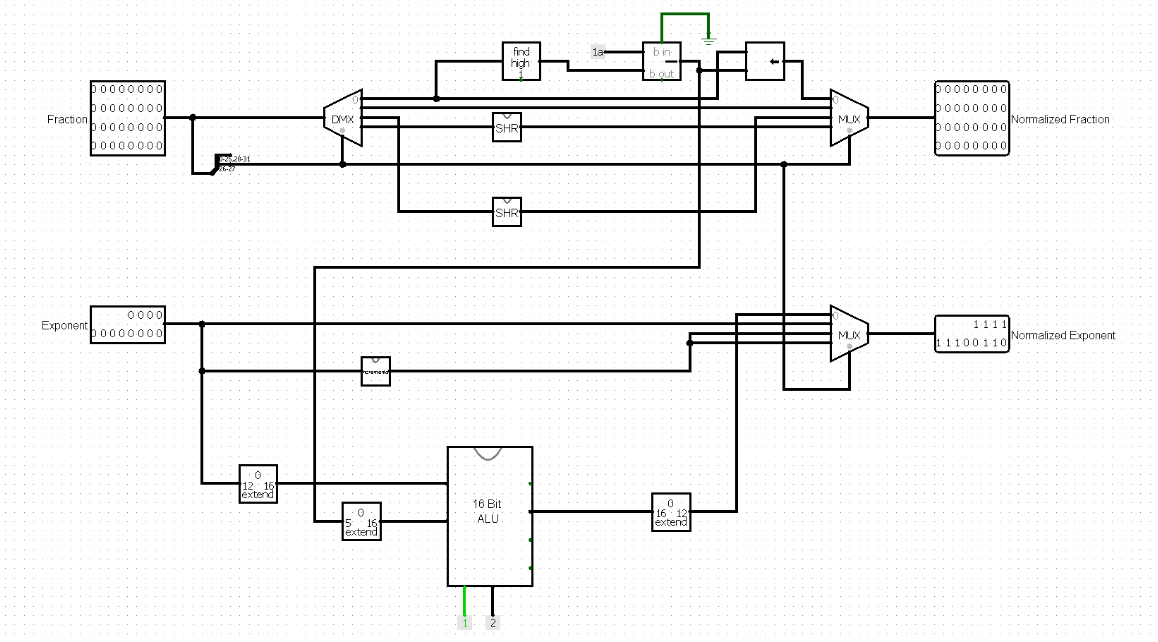


Figure 04: Normalizer circuit for Floating-point adder

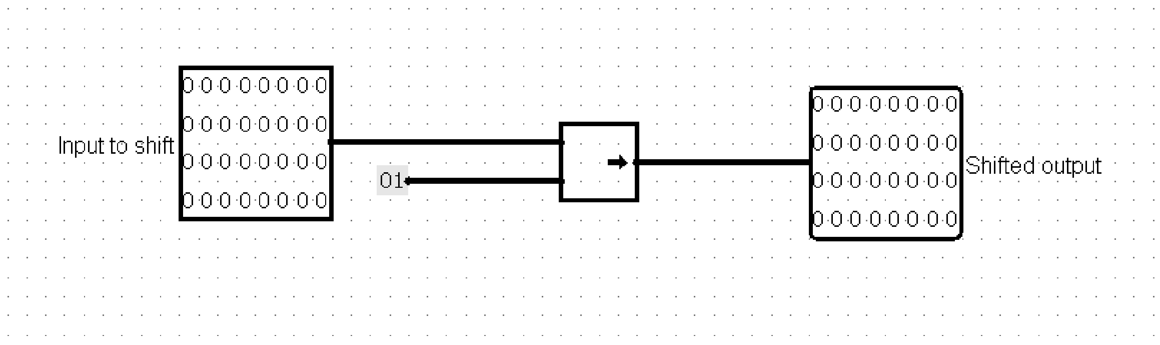


Figure 05: Right shifter

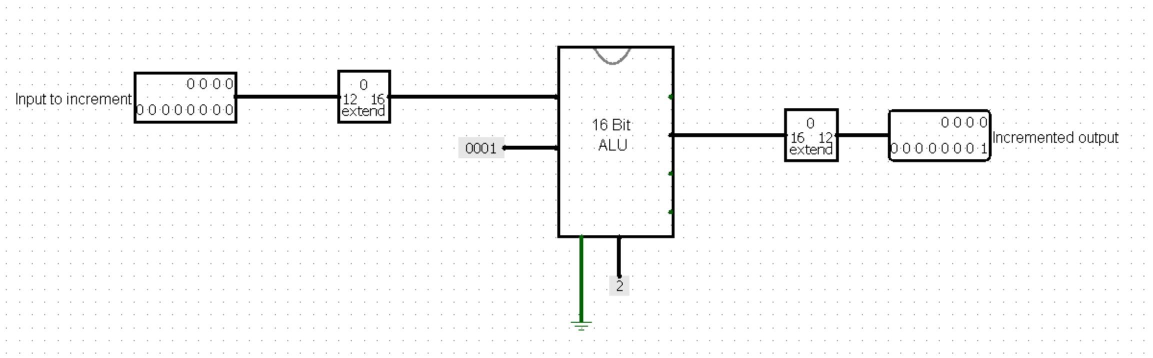


Figure 06: Incrementor circuit

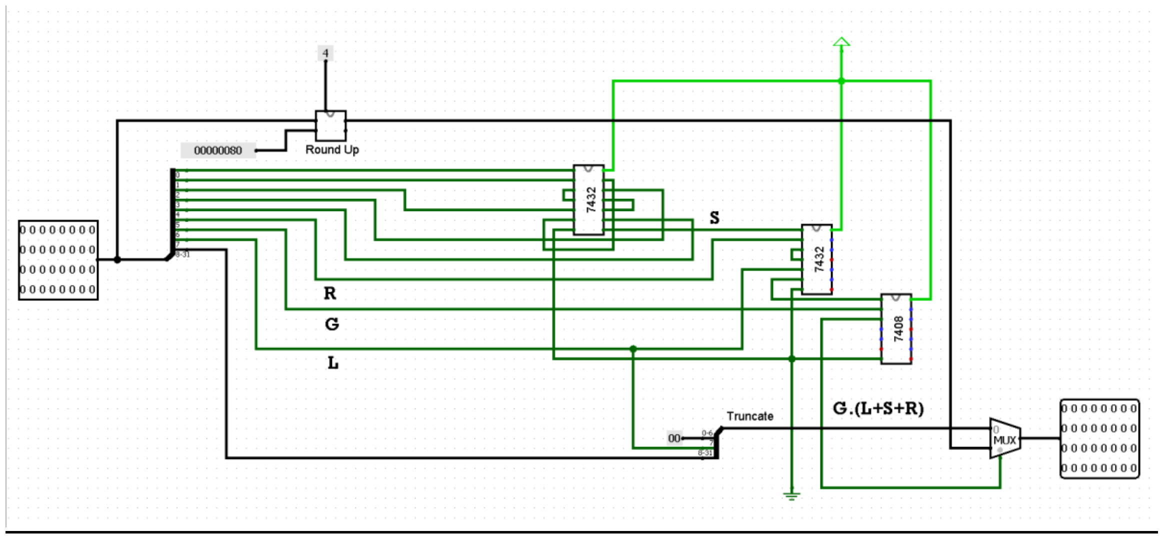


Figure 07: Rounder Circuit for Floating-point adder

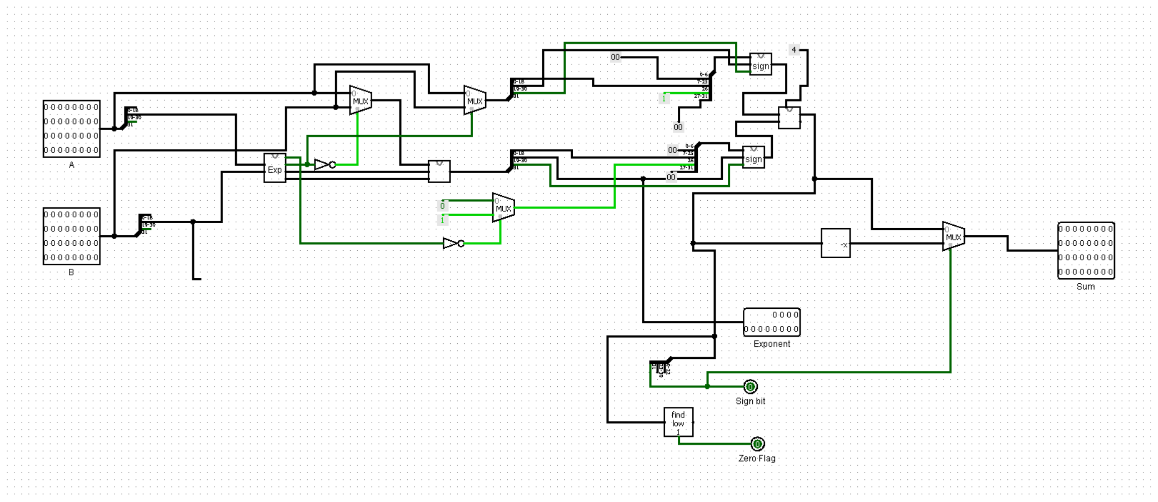


Figure 08: Exponent handler circuit for Floating-point adder

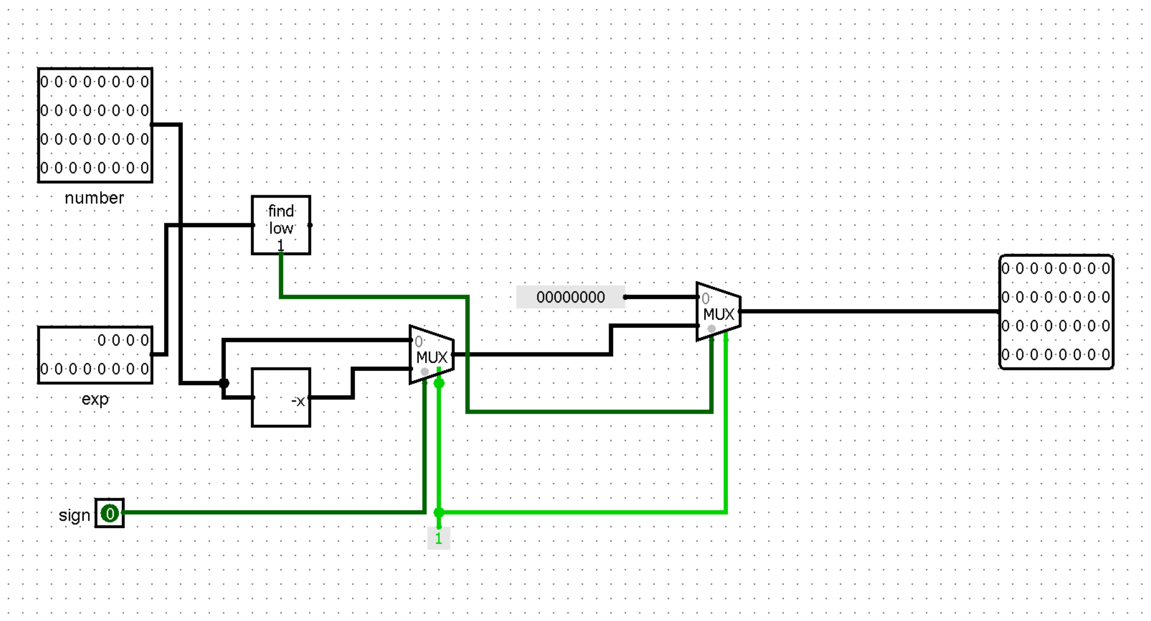


Figure 09: Sign Handler

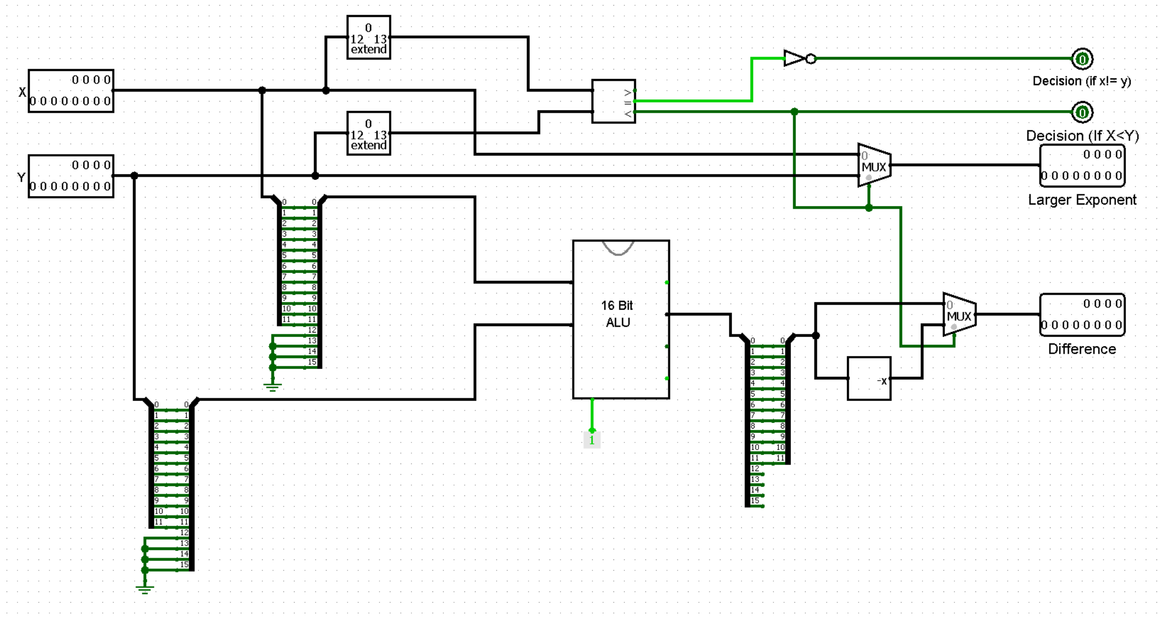


Figure 10: Exponent Handler

ICs used with count as chart:

Component	Component Count
IC 7408	1
IC 7432	1
NOT gate	1
32-bit 2x1 MUX	6
1-bit 2x1 MUX	1
12-bit 2x1 MUX	1
20-bit right shifter	1
32-bit right shifter	1
32-bit 1x4 DEMUX	1
32-bit 4x1 MUX	1
12-bit 4x1 MUX	1
5-bit subtractor	1
32-bit left shifter	1
16-bit ALU	3
32-bit ALU	2
32-bit negator	2
12-bit negator	1
13-bit comparator	1

Simulator:

Logisim Version 2.7.1

Discussion:

While implementing the circuit, we had to change our design several times. Some designs required a lot of ICs. To optimize the number of ICs, we had to discard those designs.

Again, we invested enough of our time to cross and validate the results with a correct dataset consisting of corner cases to make sure no design fault occurred. As per specification of this assignment, we did not check extreme cases (Overflow, underflow, denormalized number). However, because of truncating and rounding up the sum, sometimes we encountered precision loss of minimum one or maximum two bits. Also, if the difference between the exponents of the two inputs is too high, we are losing precision during fraction alignment of the smaller input.

Since the problem specification dictates that we design a floating-point adder where the floating point representation consists of a 12 bit exponent instead of IEEE 754 standard 8 bit exponent, and a 19 bit fraction instead of the standard 23 bits, our fraction covers a larger range of numbers, however, at the cost of smaller precision.

As this assignment scope is strictly limited to software simulation, we used Logisim modules in the implementation of our circuit to make our design cleaner, modular, robust and comprehensible, and consequently easier to analyze and debug.

Also, in some cases, we reused logic gate outputs to minimize the number of ICs used. Considering all these aspects, we finally implemented the most optimized design we could find.