CSE 306
Computer Architecture Sessional

Assignment-2: 32-bit Floating Point Adder Simulation

Section - A2
Group - 02

Members of the Group:

i 2005035 - Md. Rafiul Islam Nijamy

ii 2005036 - Tawhid Muhammad Mubashwir

iii 2005037 - Zia Ul Hassan Abdullah

iv 2005045 - Prithu Anan

v 2005055 - Ha Meem

1 Introduction

A computer representation of a real number with a fixed number of digits before and after the decimal point (or radix point in a more general sense) is called a floating point number. A floating point adder is a digital circuit that works with floating point numbers to add and subtract. Numbers that are too large or too small to be accurately represented by integer representations can be represented using this method.

Three fields make up the floating point representation of a number: the sign bit, the exponent field, and the mantissa field. The sign bit represents the sign, either positive or negative. To express a wide range of values, the exponent field permits the significand to be multiplied by a power of the base using a fixed amount of bits in a biased form. The bias needs to be deducted from the stored bits in order to get the real exponent. The fractional portion of the number, or mantissa, is made up of the digits that come after the decimal point. The sign bit, exponent field, and mantissa in this implementation use 1, 11, and 20 bits, respectively. Thus, the exponent bias for our problem would be $2^{11-1}-1 = 1023$.

In order to add two numbers, a floating point adder first aligns their decimal points before adding their mantissas. After necessary shifts and related modifications (increment/decrement), the result's exponent is fixed. Normalization and rounding are performed afterwards. The signs of the two integers being added determine the sign of the outcome.

Applications for floating point adders include financial modeling, computer graphics, and calculations in science and engineering. Because co-processors can be computationally demanding, it is utilized in them to do quick, hardware-accelerated floating point arithmetic. These calculations can be completed far more quickly by a specialized floating point adder than by the main processor. Applications requiring high-precision floating point calculations, such data analysis and scientific simulations, may find this to be especially crucial..

2 Problem Specification

The task involves creating a circuit for a floating point adder that accepts two floating points as inputs, adds their sum, and outputs another floating point. Every floating point will have a length of 32 bits and be represented as follows:

Sign	**Exponent**	**Fraction**
1 bit	11 bits	20 bits

Table 1: Problem Specification

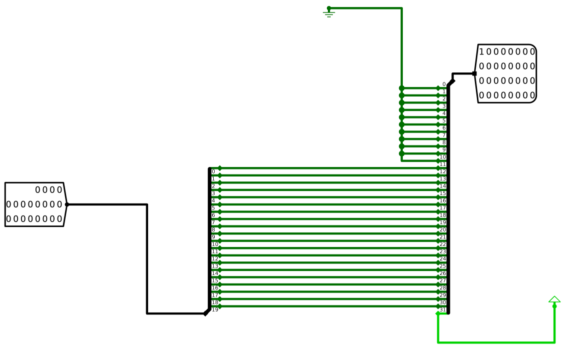
3 Description and Circuit Diagram of Modules

In order to maintain modularity in the floating point adder design, multiple libraries have been built and implemented. Descriptions and usages of the libraries are given below:

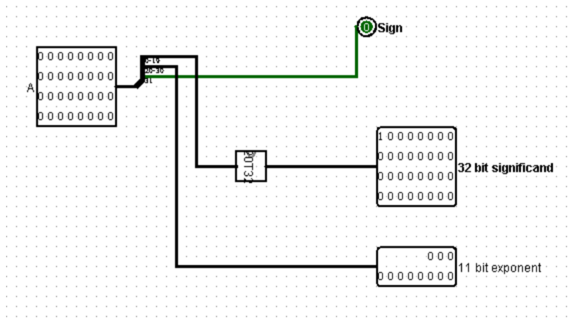
3.1 Processing the Input

This module(InputHandler.circ) helps to process the input for further operations in the floating point adder. It contains:

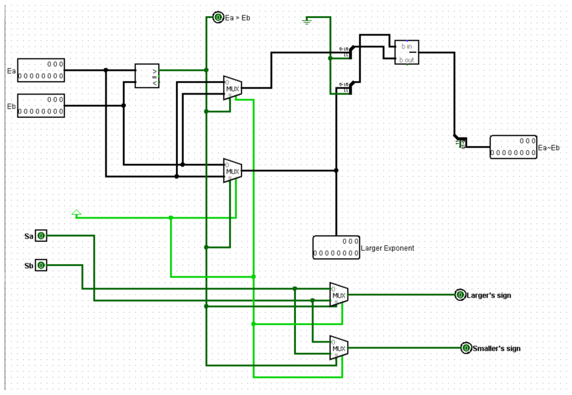
- A bit extractor that extends the 20 bit significand to 32 bit which includes a leading 1 at the 2*nd* bit [20to32]
- A circuit which splits the 32 bit floating point number to 11 bit exponent and 20 bit significand [Input Processor]
- Exponent differentiator that calculates the difference between the exponenets of two inputs [Exponent Differentiator]
- A circuit that determines which input is greater than the other, greater exponent, exponent difference etc for control decisions [Input]



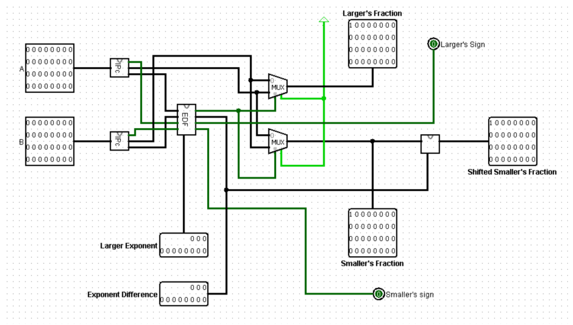
(a) 20 to 32 Bit Extractor



(b) Input Processor



(c) Exponent Differentiator



(d) Input Handler

Figure 1: Input Operator

3.2 Adder Library

An adder circuit (adder32bit.circ) has been included in the adder32bit library which performs the most crucial part of the FPA, which is to add the significands of the two given input floating point numbers.

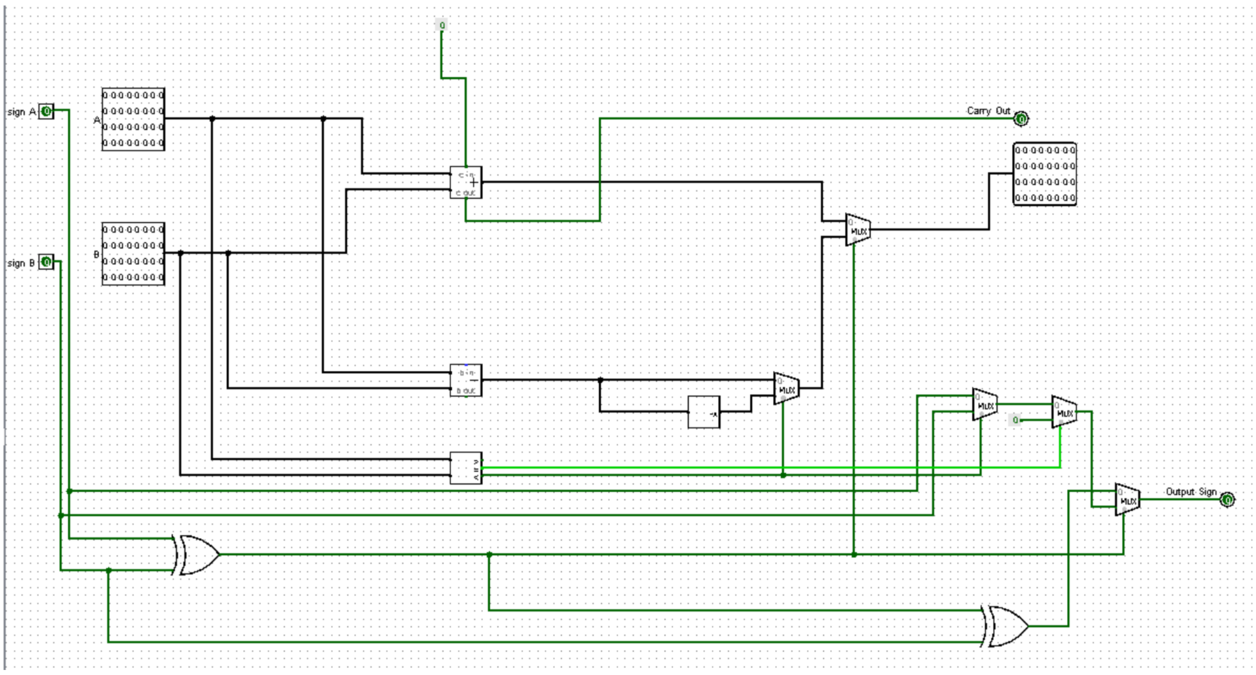
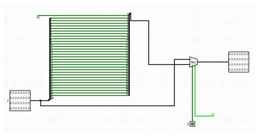


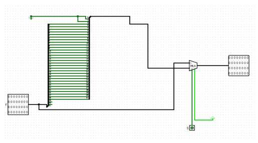
Figure 2: 32 Bit Adder Circuit

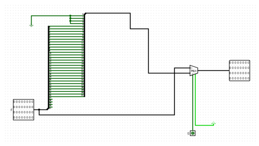
3.3 Shifter Library

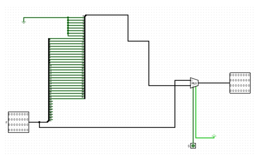
Shifters are required in floating point adders to shift the fraction in order to normalize or balance with the other operand. We have implemented shifters(leftShifterLib.circ and rightShifterLib.circ) with splitters and muxes from the Logisim built-in library. The circuits in the libraries are:

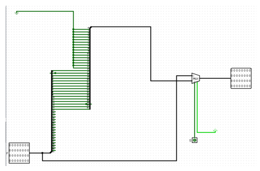
- 1, 2, 4, 8, 16 bits left shifter [1LeftShift, 2LeftShift, 4LeftShift, 8LeftShift, 16LeftShift]
- 1, 2, 4, 8, 16 bits right shifter [1RightShift, 2RightShift, 4RightShift, 8RightShift, 16RightShift]
- Arbitrary left and right shifter (Can shift any number of bits up to 31 bits) [ArbLeftShift, ArbRightShift]
- Right shifter that will make every bit 0 if it needs shfting more than 31 bits [RightShiftWithEmpty]

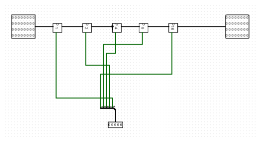

(a) 1 Bit Left Shifter

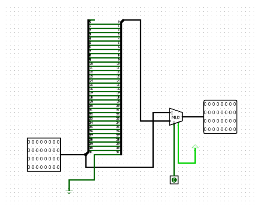

(b) 2 Bit Left Shifter


(c) 4 Bit Left Shifter

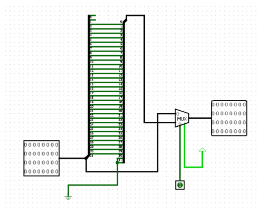

(d) 8 Bit Left Shifter


(e) 16 Bit Left Shifter

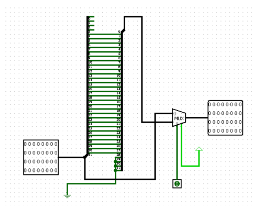

(f) Arbitrary Left Shifter



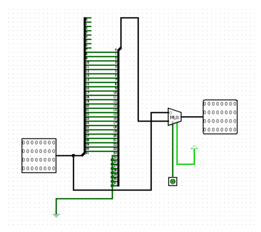
(g) 1 Bit Right Shifter



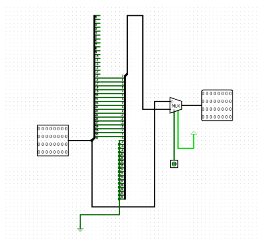
(h) 2 Bit Right Shifter



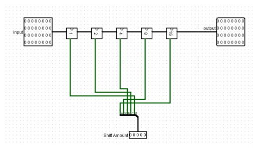
(i) 4 Bit Right Shifter



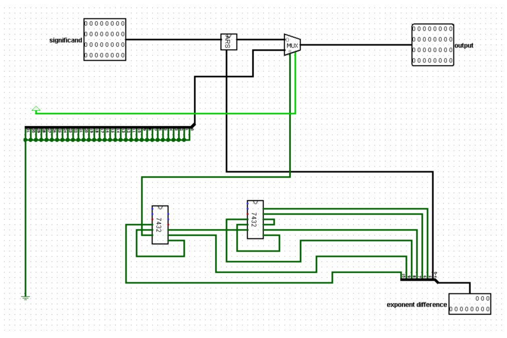
(j) 8 Bit Right Shifter



(k) 16 Bit Right Shifter



(l) Arbitrary Right Shifter



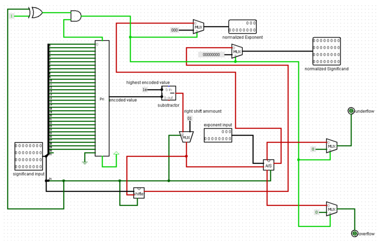
(m) Right Shift With Empty

Figure 3: Shifter Circuits

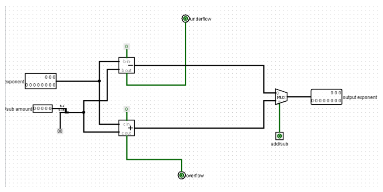
3.4 Normalizer Library

The circuit normalization included in this library (NormalizerLib.circ) normalizes the output by applying the necessary number of bit shifts with the aid of an encoder library. Additionally, it produces an 11-bit value that is added to the exponent to finish the normalizing process.

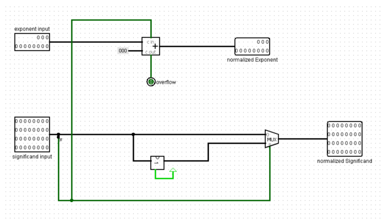
- Normalizer
- Shifter
- Adder-Subtractor
- Rounded-Normalizer
- Zero-Checker
- One-Checker



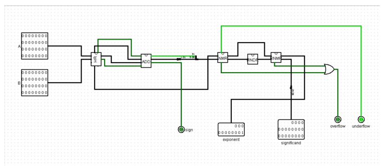
(a) Normalizer



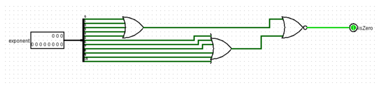
(b) Shifter



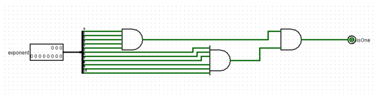
(c) Adder-Subtractor



(d) Rounded-Normalizer



(e) Zero-Checker



(f) One-Checker

Figure 4: Normalizer Circuits

3.5 Rounding Circuit

The name of this module is Rounder.circ. It contains a comparator and and a full adder and does the rounding of the mantissa.

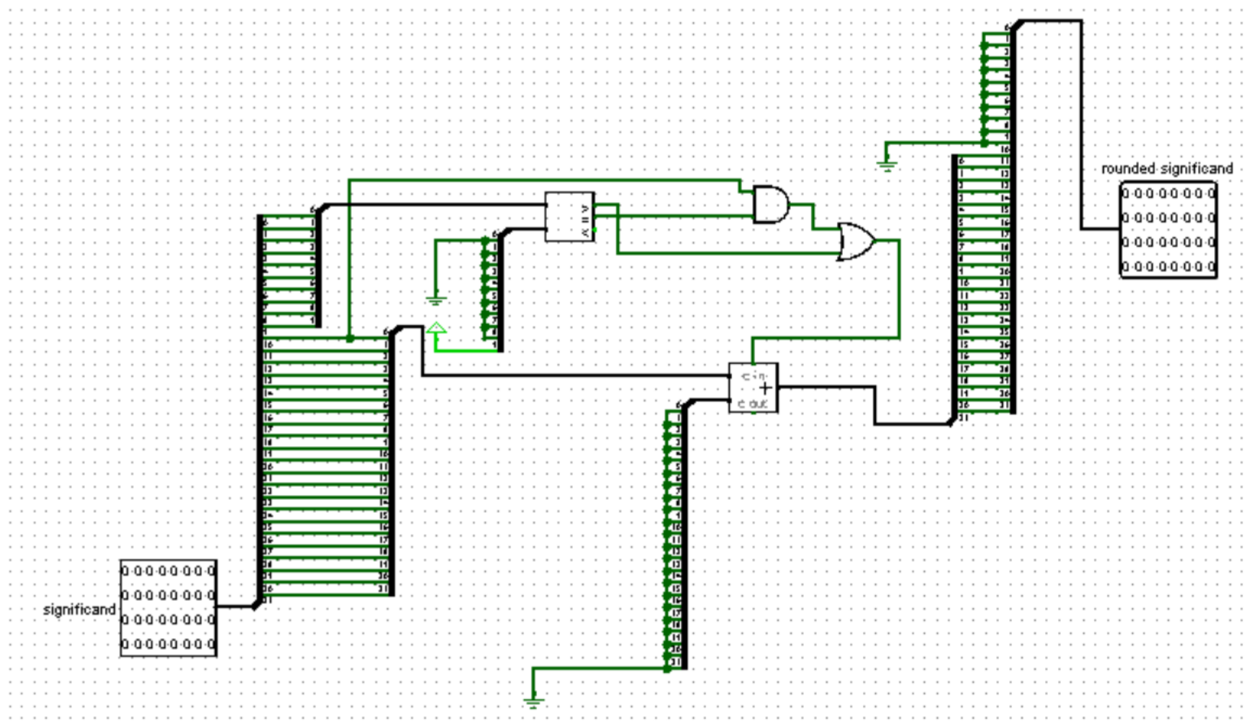


Figure 5: Rounder

3.6 Floating Point Adder

The last module, called FPA.circ, uses the libraries and other modules to fully create a floating point adder. It has the real floating point adder, or circuit FPA. The output processor circuit that merges the sign, exponent and significand of the result is also included in this module.

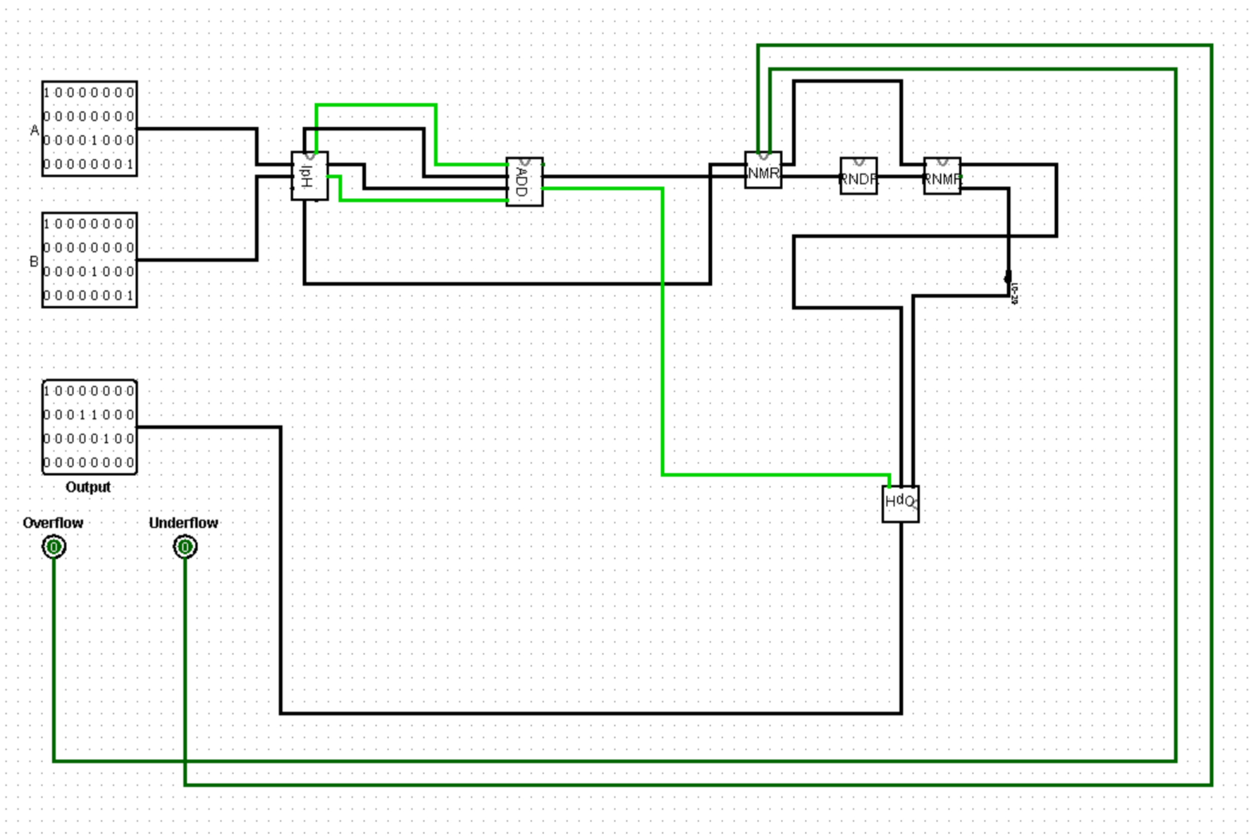


Figure 6: The FPA

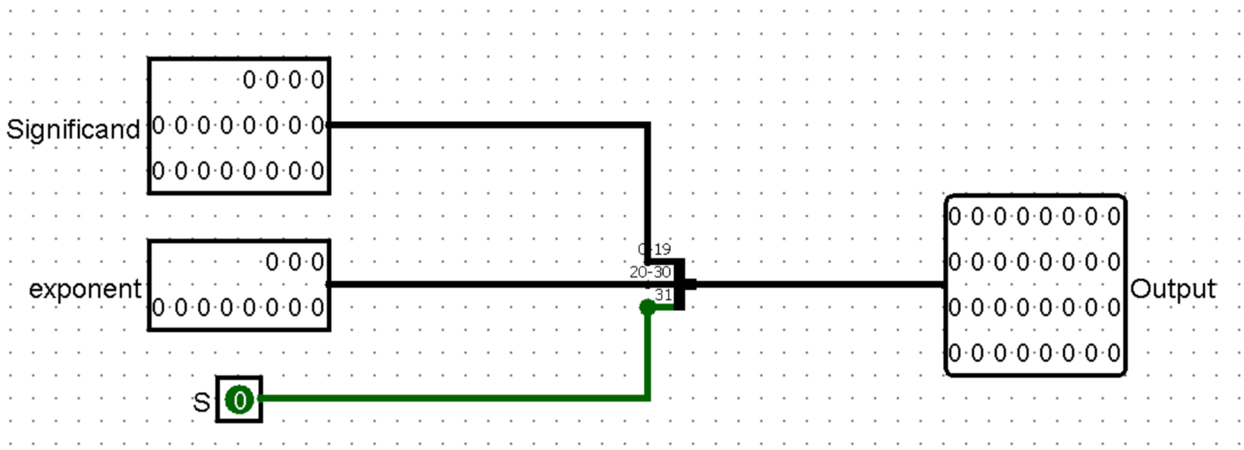


Figure 7: Output Processor of the Result

3.7 Third Party Libraries

Third party libraries 7400-lib.circ and logi7400dip.circ are used to incorporate 7400 series ICs in the floating point adder implementation.

4 Flowchart of the Addition/Subtraction Algorithm

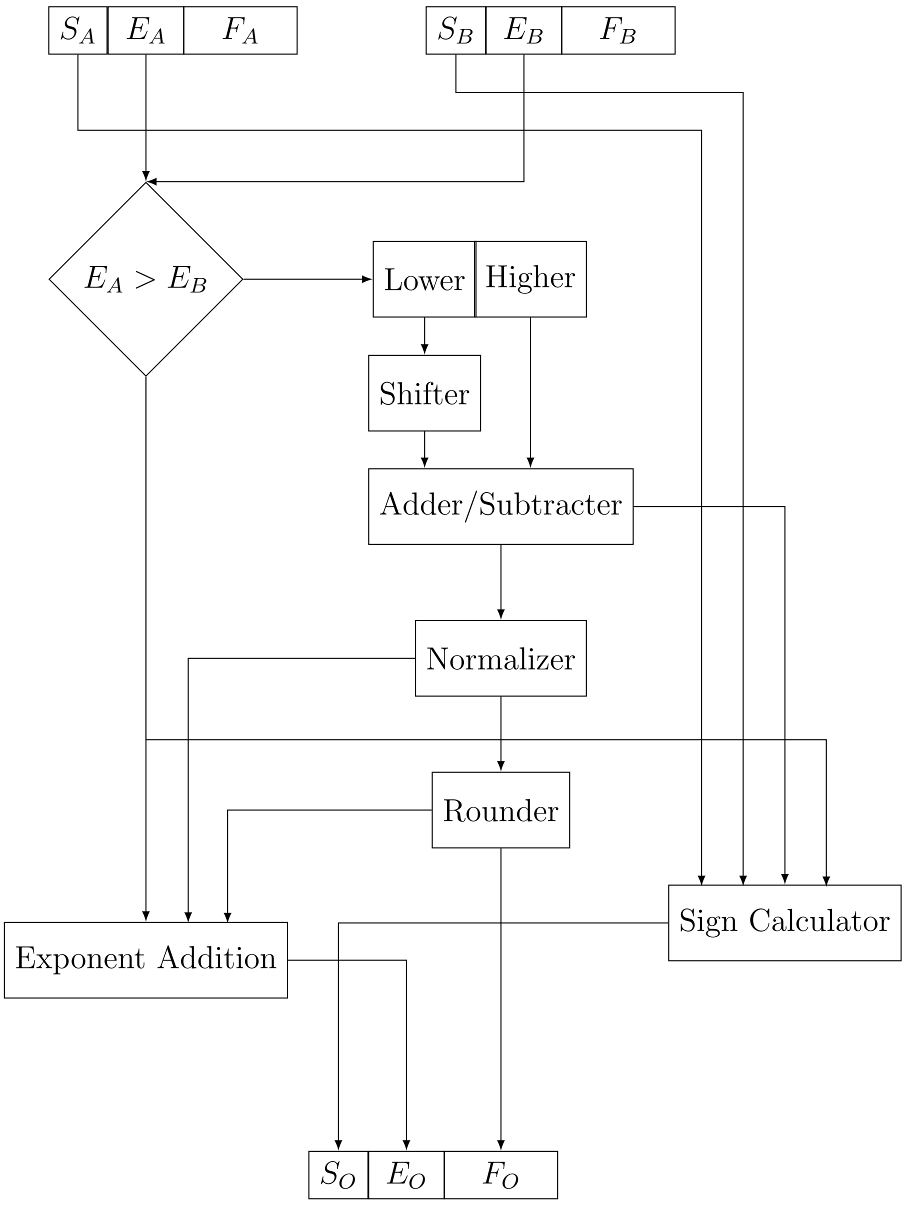


Figure 8: Flow chart of the addition/substraction algorithm

5 High-level Block Diagram of the Architecture

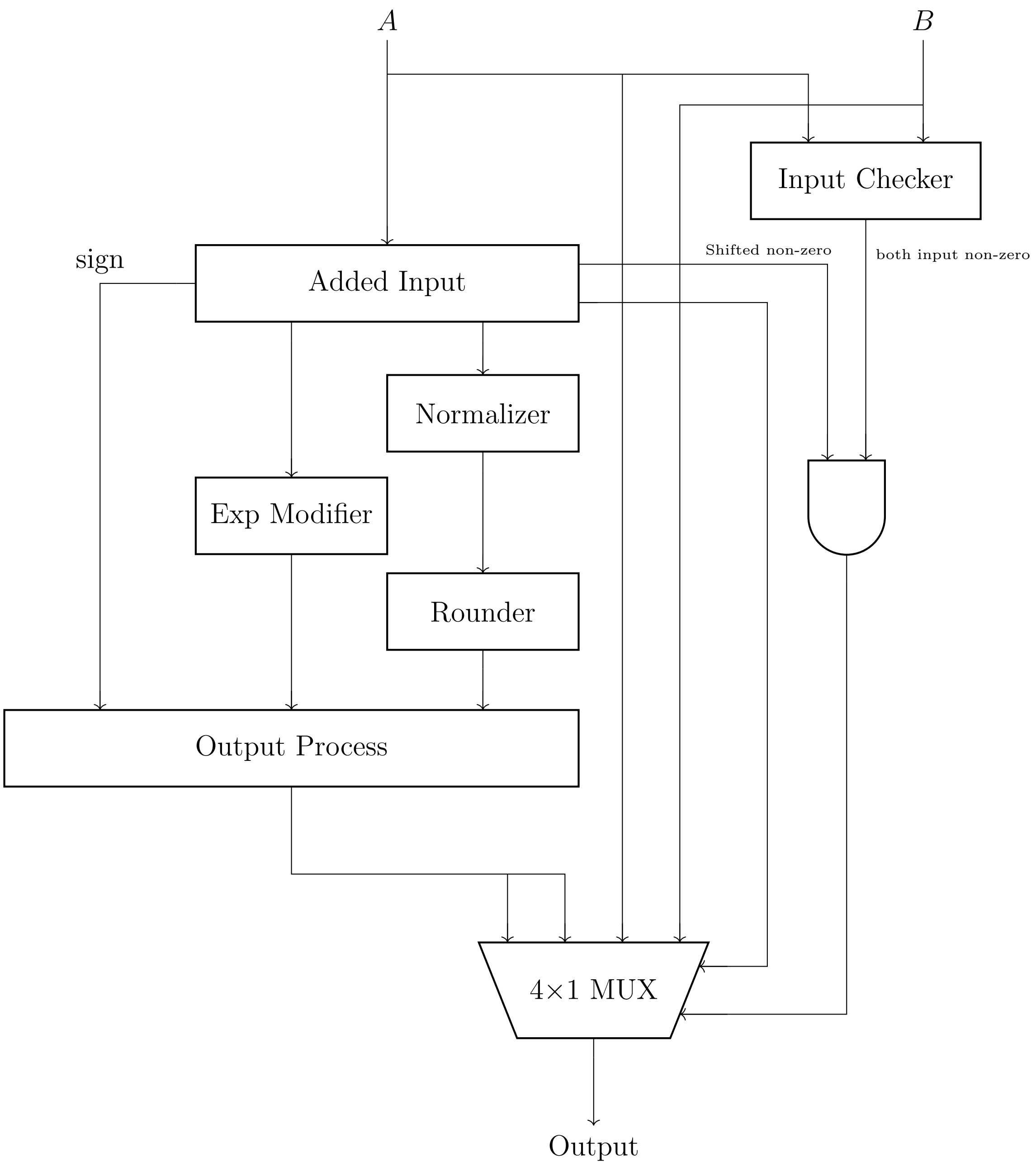


Figure 9: Block Diagram of FPA

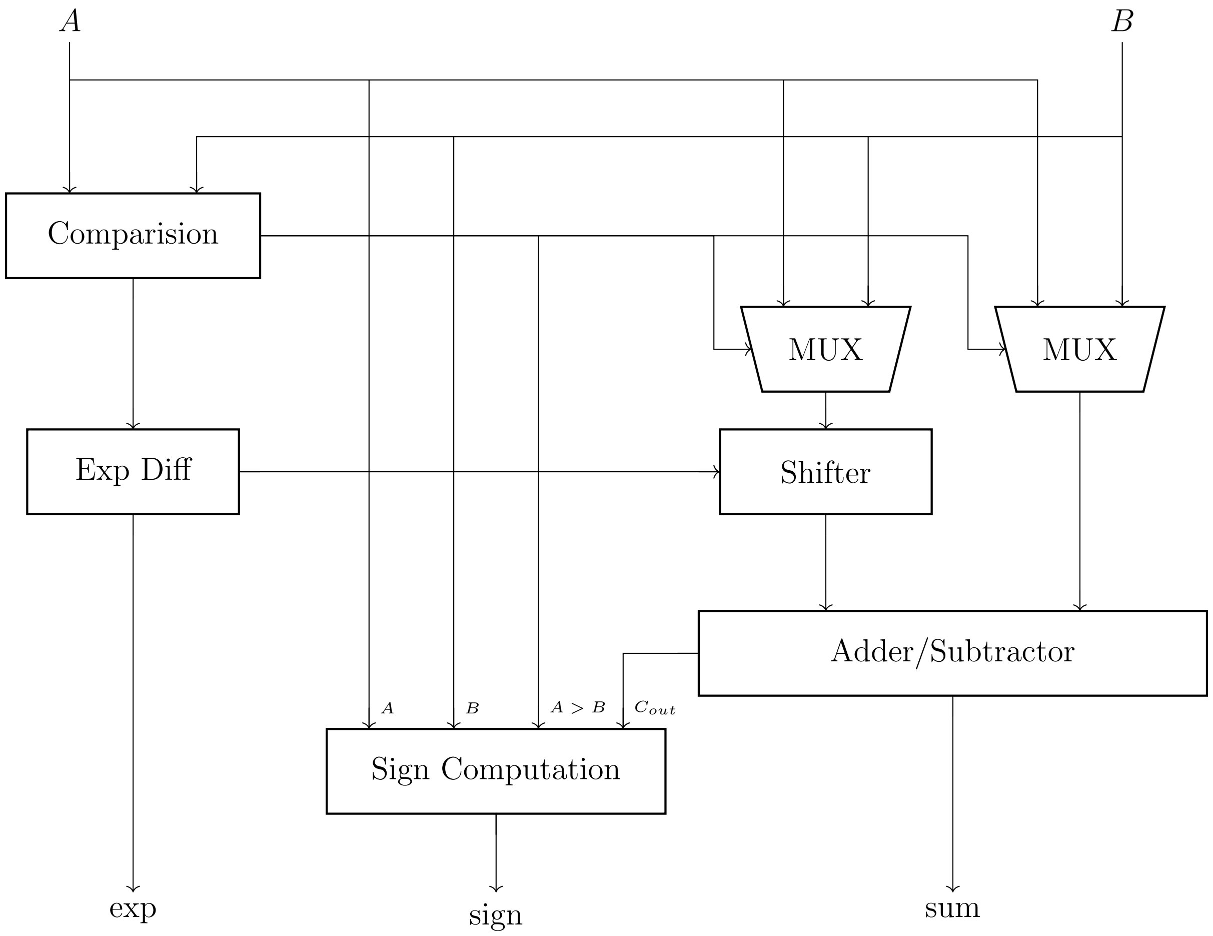


Figure 10: Block Diagram of Added Input

6 Comprehensive Design Description

6.1 Comparing the Exponents and Aligning Radix Point

The radix points need to be lined up in order to add two floating-point values. To align the input with the larger input, this is usually accomplished by moving the input with the smaller exponent to the right. We have computed the difference between the two inputs using a 12-bit subtractor and compared their exponents using a comparator.

6.2 Shifting

All shifting operations have been carried out using shifters based on multiplexers. We have employed five 32-bit multiplexers to obtain bit shifts of any arbitrary value up to 31. The multiplexers have the ability to shift 1, 2, 4, 8, and 16 bits in turn. Combining them yields shifts of any length up to 31, as 3f and 3l illustrate. It could take more than 31 shifts to align two fractions, in which case all bits would be 0. Furthermore, the exponent difference ought to be the shifter circuit's input. Thus, after all seven of the higher bits are clear, the lower five bits are used for shifting. The shift amount does not exceed $31(2^5 - 1)$ just in that scenario. Otherwise, the fraction will be 0(all bits cleared), which is done in 3m.

6.3 Normalization

We used a priority encoder to determine the location of the most significand set bit, which is the leftmost one, in order to normalize and then do an addition operation. To do this, four 8 to 1 priority encoders are employed. The result consists of 5 bits $(0 - 2^5 - 1)$, so it can have up to 31 values (from $0_{th} to 31_{th}$). The result's lower three bits are the encoders' output. Using their valid bits, the encoders' upper two bits and outputs are chosen. Other encoders won't be selected and the top two bits will be 00 if the leftmost encoder receives a valid input. Therefore, the decimal range of the integers that are formed is 0 to 7. Likewise, if the second encoder is chosen, the top two bits will be 01; the third and fourth ones will be 10 and 11, respectively. A 4 to 2 priority encoder, whose inputs are the valid bits of the previously stated priority encoders, passes the selection bit of the multiplexer to it. In the event that neither of the priority encoders accepts an input, the fraction is comprised entirely of zeros. This can happen in two cases:

1. The result is of the structure $1.00 * 2^x$
2. The result is 0.0

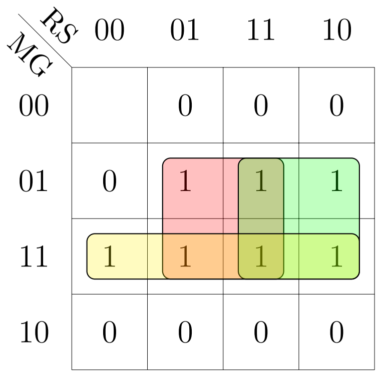
If two numbers of type $1.00 * 2^{x-1}$ with equal exponents are added, the first instance may manifest. Thus, we can verify if the signs in the two inputs match. If so, we shall proceed with the method and raise the exponent by one. In reality, the outcome is 0 if the signs are opposite. Thus, 0.0 is the output.

6.4 Rounding

We used a 32-bit adder to add and subtract the mantissas. We may, however, occasionally need to round the result because we can only hold 20 bits. $21st$ bit is regarded as the guard bit, and $22nd$ as the round bit. Sticky bit will be set if any of the bits to the right of Round bit are set. If not, it's evident. We must take into account these situations:

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

Actually X100 is the case for round to even. So, if 20*th* bit is 0, we need to do nothing, hence truncation. Otherwise, we will round up. For simplification we will use K-map. $(20thbit = M)$



$$flag = GS + GR + MG = G(M + R + S)$$

If flag is set, 1 is added at the 20*th* position. Otherwise, the bits beyond the 21*st* position are truncated.

6.5 Computing the Sign Bit

When computing the sign bit, there are a few things to keep in mind. The sign of the output will be either of the inputs' signs if the signs of the two inputs are the same. In the event that they differ, we must take two factors into account. First, we must determine the adder's output sign. It can be calculated using the equation:

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

Where C_{out} is the carry out of the adder. Since we constantly deduct the input with the smaller exponent from the input with the higher exponent, this sign could not always be accurate. 1 yields the opposite result to the right one when the signs of the inputs are opposite, that is, if the input with the larger exponent is negative and the other one is positive. We have included a variable switch to address this. In some situations, this switch bit will change the sign bit. When the inputs' sign bits are opposite and the input with the larger exponent is negative, our sign bit will fail. The switch will then become 1 and change the sign in that scenario. The switch bit equation is:

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

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

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

If there are differences in the signs of the inputs, equation 3 will be applied. If not, the output's sign will be directly affected by the first input's sign. A multiplexer will be used to make this choice.

6.6 Handling Zero or Small Input

The other input will be sent directly to the output if one of the inputs is zero. Furthermore, the input with the larger exponent passes directly to the output if the exponent of one of the inputs is substantially smaller than the other (a difference of greater than 31).

6.7 Increasing Precision

Furthermore, we opted not to reserve a bit for capturing the overflow bit. Rather, we took the bit directly out of the carry out. Furthermore, we did not use a separate bit for the sign in the subtraction instance. Rather, the sign of the result was ascertained by utilizing the adder's carry out. Subtraction yields a positive outcome if the carry out is 1, and a negative result otherwise. If one of the summands is zero, there is an exception. As a result, as mentioned in 6.6, this situation was handled independently. As a result, the precision is increased by 2.

7 ICs Used with Count as a Chart

IC	**Quantity**
IC 7404	1
IC 7408	2
IC 7432	2
IC 7486	1
IC 74157	6
Total	12

Table 2: ICs Used with Quantity

It is worth noting that the IC count seems relatively smaller because we have used the built-in library of logisim for adders and comparators and thus neither did we have to design them from scratch using more basic ICs nor did we have to include their IC count to the total tally.

8 Simulator used Along with the Version Number

Logisim 2.7.1 has been used for simulating the floating point adder circuit.

9 Discussion

We tried our utmost best to ensure the novelty of our implementation of the FPA. Designing the entire circuit from scratch would have been a cumbersome task. So, we decided on building separate modules first and then combining them together to obtain the actual working circuit. This approach also helped us to properly divide workload among ourselves.

We faced a handful of challenges along the way. The first notable challenge was to figuring out how to construct a shifter that could shift a number by any arbitrary amount. We did this by serially joining together 1, 2, 4, 8 and 16 bit shifters together where each of them corresponded to one of the input bits of the shift amount.

Secondly, handling negative numbers as input was also somewhat of a thought provoking task. Empirically we found out the result depended upon the sign of the inputs and their relation of magnitude. This gave rise to only 4 possible cases which we handled using adder, subtractor, complementer, XOR and muxes.

Accounting for overflow and underflow was the third challenge. We did not reserve a separate bit to record overflow or the sign of the result, as explained in 6.7. As a result, we were able to increase our implementation's precision by 2 digits. To get over the restriction of having no additional bit, we used the carry out from the adder.

We used built-in circuits from logisim library in unison with our own implemented circuits to ensure the most minimalist yet fully functional design possible from our part.

All things considered, designing and implementing the floating point adder was an interesting task that provided us with a better understanding of the inner workings of a Floating Point Adder.