CSE 306
Computer Architecture Sessional

Assignment-2: 32-bit Floating Point Adder Simulation

Section - A1
Group - 01

Members of the Group:

i 1905001 - Mohammad Sadat Hossain

ii 1905002 - Nafis Tahmid

iii 1905004 - Asif Azad

iv 1905005 - Md. Ashrafur Rahman Khan

v 1905008 - Shattik Islam Rhythm

1 Introduction

A floating point number is a representation of a real number in a computer, with a fixed number of digits before and after the decimal point. A floating point adder is the digital circuit that performs addition and subtraction on floating point numbers. It may be used to represent numbers that are too large or too small to be represented accurately with integer representations.

The floating point representation of a number consists of three fields: the sign bit, the exponent field, and the mantissa field. The sign bit represents the sign, either positive or negative. The exponent field allows the significand to be multiplied by a power of the base, using a fixed number of bits in a biased form, in order to represent a wide range of values. To obtain the actual exponent, the bias has to be subtracted from the stored bits. The mantissa is the fractional part of the number, containing the digits after the decimal point. In this implementation, the sign bit, the exponent field and the mantissa take up 1, 12 and 19 bits respectively. Thus, the exponent bias for our problem would be $2^{12-1} - 1 = 2047$.

A floating point adder works by aligning the decimal points of the two numbers to be added and then adding the mantissas. The exponent of the result is fixed making required shifts and corresponding adjustments(increment/decrement) going through the process of normalization and rounding. The sign of the result is based on the signs of the two numbers being added.

Floating point adder is used in a variety of applications, including scientific and engineering calculations, financial modeling, and computer graphics. It is used in co-processors to perform fast, hardware-accelerated floating point arithmetic as they can be computationally intensive. A dedicated floating point adder can perform these calculations much faster than the main processor. This can be especially important in applications such as scientific simulations, data analysis, and other tasks that require high-precision floating point calculations.

2 Problem Specification

The assignment asks to design 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 the following representation:

Sign	**Exponent**	**Fraction**
1 bit	12 bits	19 bits

Table 1: Problem Specification

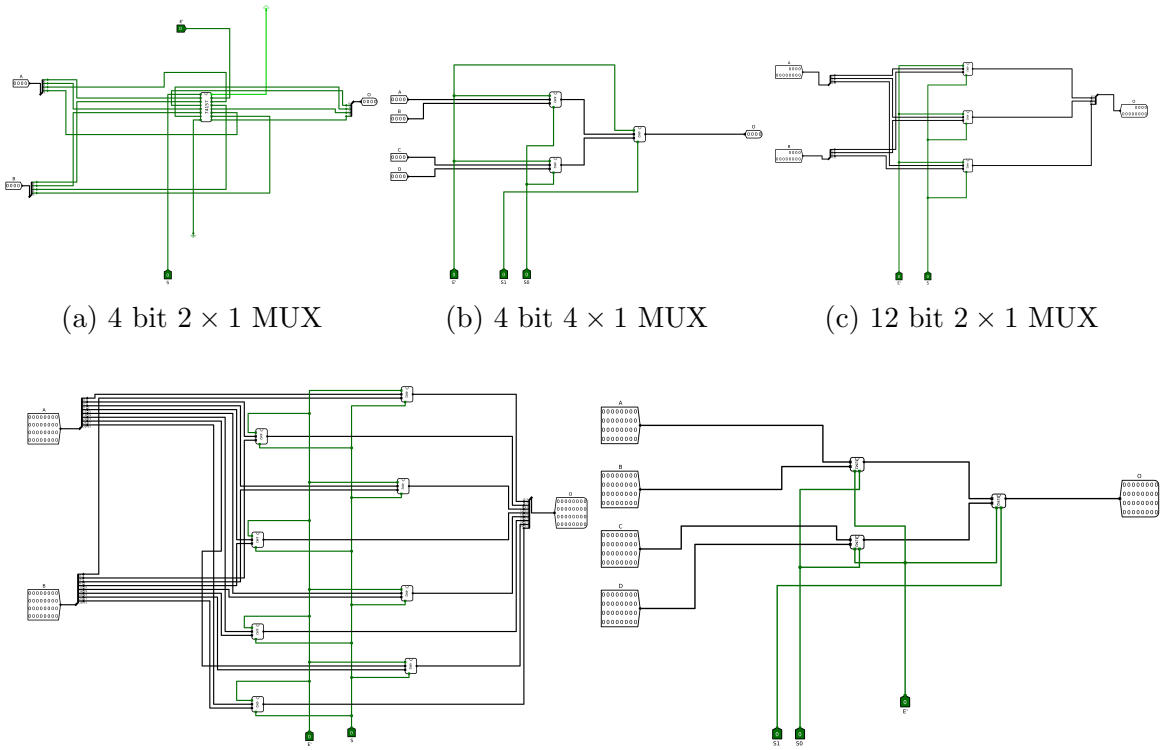
3 Description and Circuit Diagram of Modules

Several libraries have been implemented for the sake of modularity in the design of the floating point adder. Descriptions and usages of the libraries are given below:

3.1 Multiplexer Library

Multiplexer library(MuxLib.circ) has several circuits to help with multiplexing in the floating point adder. The important components are:

- Abstracted 4 bit 2 to 1 Mux (This circuit abstracts the IC 74157 for ease of use) [4Mux2X1]
- 4 bit 4 to 1 Mux [4Mux4X1]
- 12 bit 2 to 1 Mux [12Mux2X1]
- 32 bit 2 to 1 Mux [32Mux2X1]
- 32 bit 4 to 1 Mux [32Mux4X1]



(a) 4 bit 2×1 MUX (b) 4 bit 4×1 MUX (c) 12 bit 2×1 MUX

(d) 32 bit 2×1 MUX (e) 32 bit 4×1 MUX

Figure 1: Multiplexer Circuits

3.2 Comparator Library

To compare the exponenets of the inputs, a 12 bit comparator is constructed using IC 7485. This library(ComparatorLib.circ) holds the circuit 12BitMagnitudeComparator which takes two 12 bit numbers and compare them.

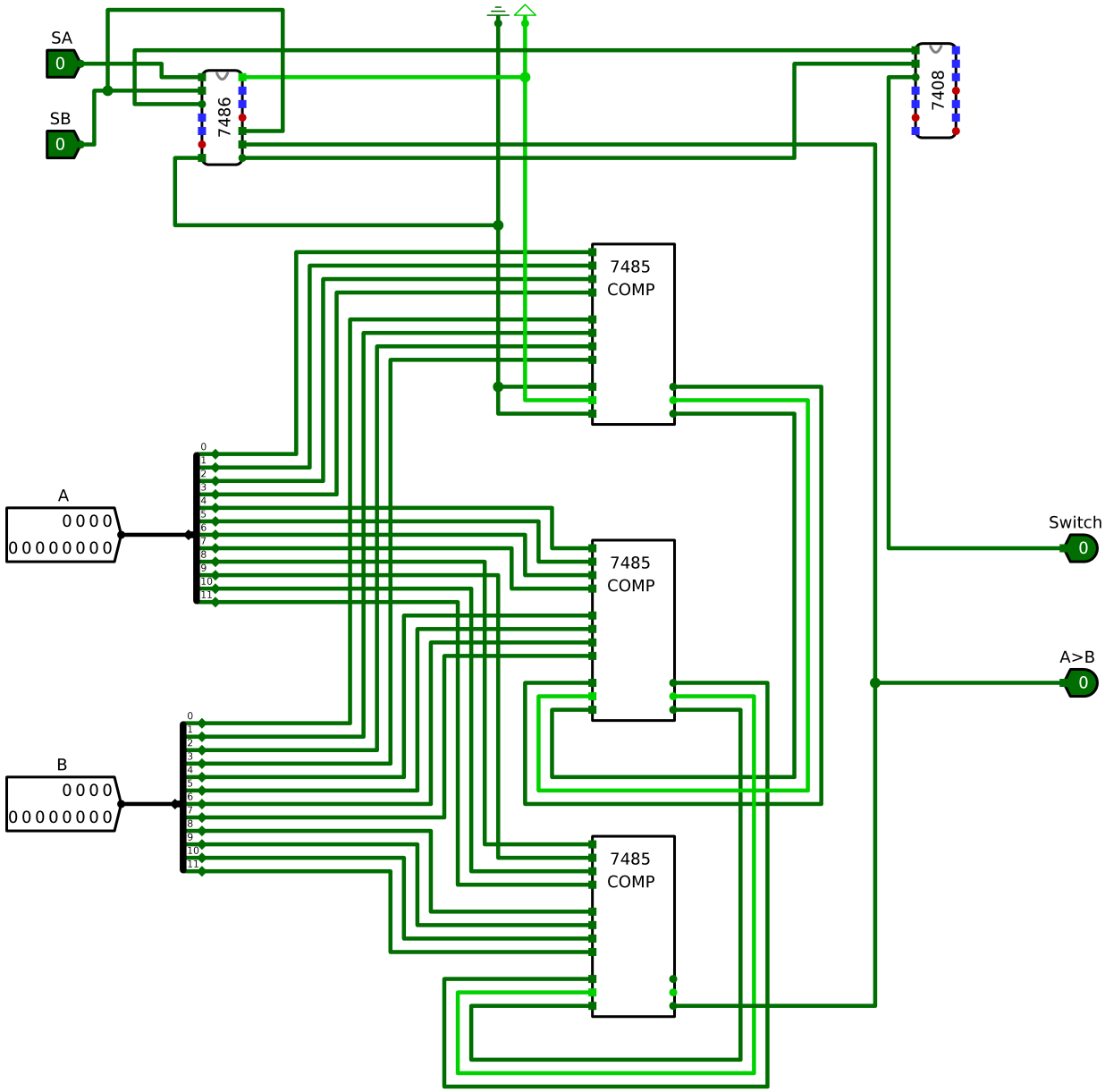
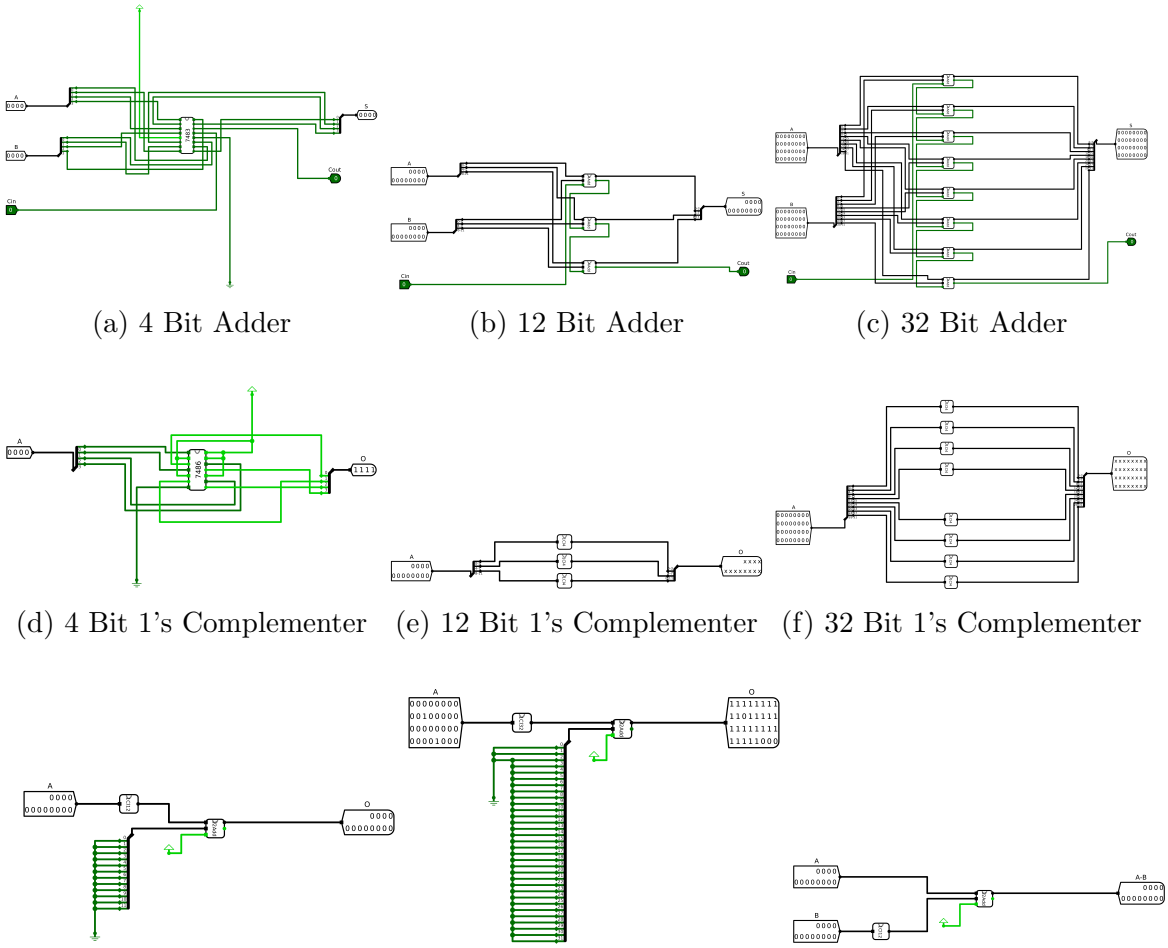


Figure 2: 12 Bit Magnitude Comparator

3.3 Adder Library

In floating point adder, we need adders and subtractors in several places. To make the work easier, an adder library(AdderLib.circ) is constructed. It includes:

- 4, 12, 32 bit adder [4Adder, 12Adder, 32Adder]
- 4, 12, 32 bit 1's complementer [1Compl4, 1Compl12, 1Compl32]
- 12, 32 bit 2's complementer [2Compl12, 2Compl32]
- 12 bit subtractor [12Sub]



(a) 4 Bit Adder (b) 12 Bit Adder (c) 32 Bit Adder

(d) 4 Bit 1's Complementer (e) 12 Bit 1's Complementer (f) 32 Bit 1's Complementer

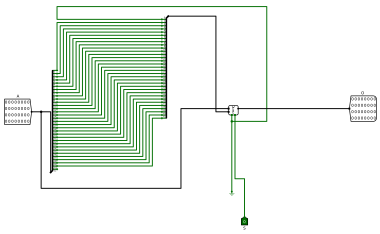
(g) 12 Bit 2's Complementer (h) 32 Bit 2's Complementer (i) 12 Bit Subtractor

Figure 3: Adder Circuits

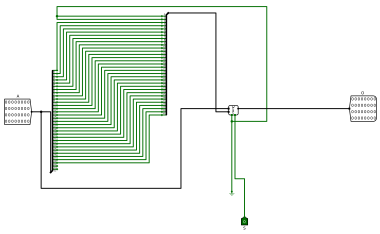
3.4 Shifter Library

In floating point adder we need shifters to shift the fraction in order to balance with the other operand or normalize. We have implemented shifters(ShiftLib.circ) with splitters and muxes implemented in multiplexer library. The circuits in the library are:

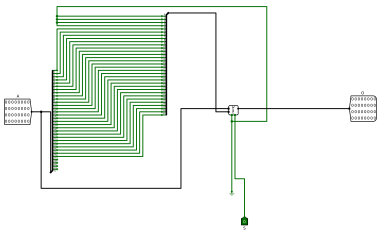
- 1, 2, 4, 8, 16 bits left shifter [1LeftShift, 2LeftShift, 4LeftShift, 8LeftShift, 16LeftShift]
- 1, 2, 4, 8, 16 bits right shifter [1RightShift, 2RightShift, 4RightShift, 8RightShift, 16RightShift]
- Arbitrary left and right shifter (Can shift any number of bits up to 31 bits) [ArbLeftShift, ArbRightShift]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



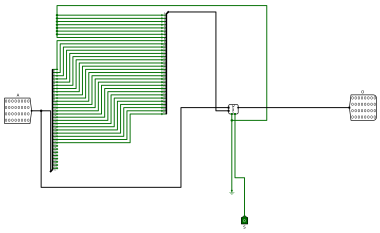
(a) 1 Bit Left Shifter



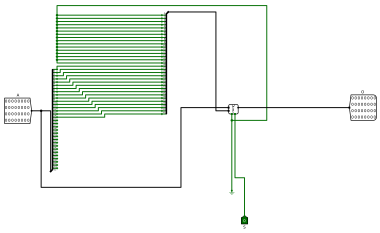
(b) 2 Bit Left Shifter



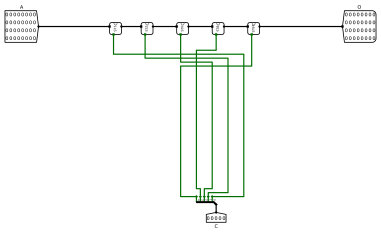
(c) 4 Bit Left Shifter



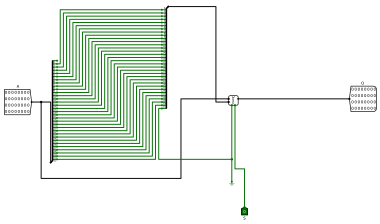
(d) 8 Bit Left Shifter



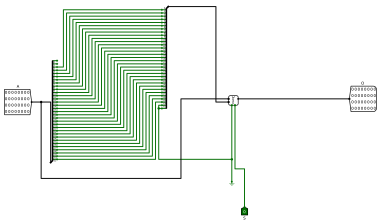
(e) 16 Bit Left Shifter



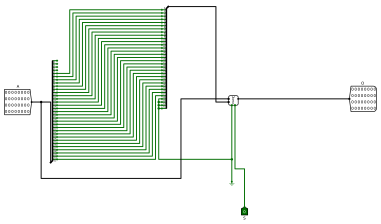
(f) Arbitrary Left Shifter



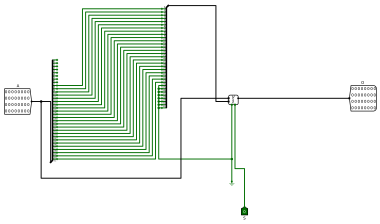
(g) 1 Bit Right Shifter



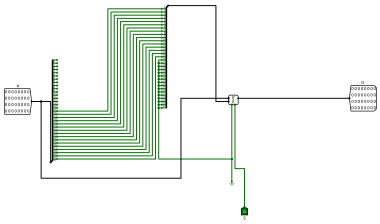
(h) 2 Bit Right Shifter



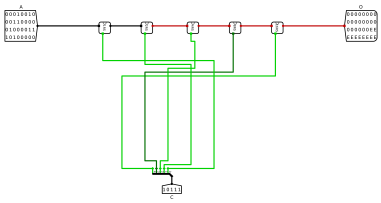
(i) 4 Bit Right Shifter



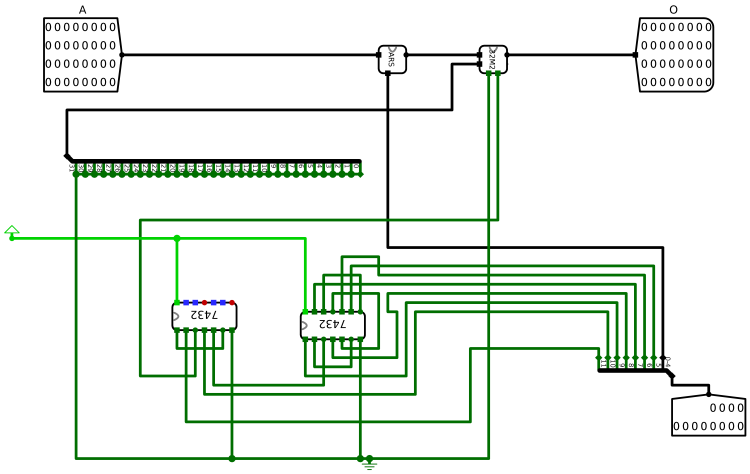
(j) 8 Bit Right Shifter



(k) 16 Bit Right Shifter



(l) Arbitrary Right Shifter



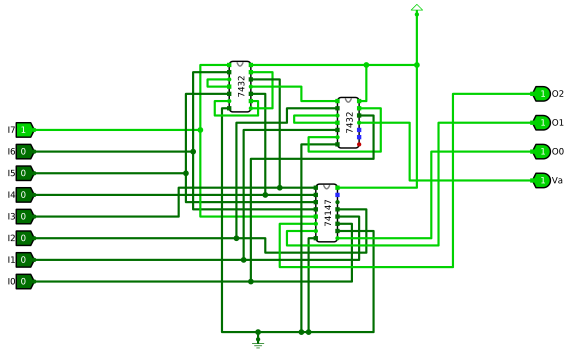
(m) Right Shift With Empty

Figure 4: Shifter Circuits

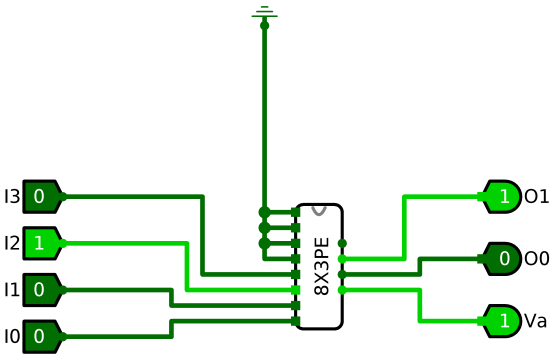
3.5 Encoder Library

To normalize the final result, we need to locate the first set bit from left to right. It can be done easily using priority encoder. To help with this task, this library(EncoderLib.circ) is implemented. The contents of this library are:

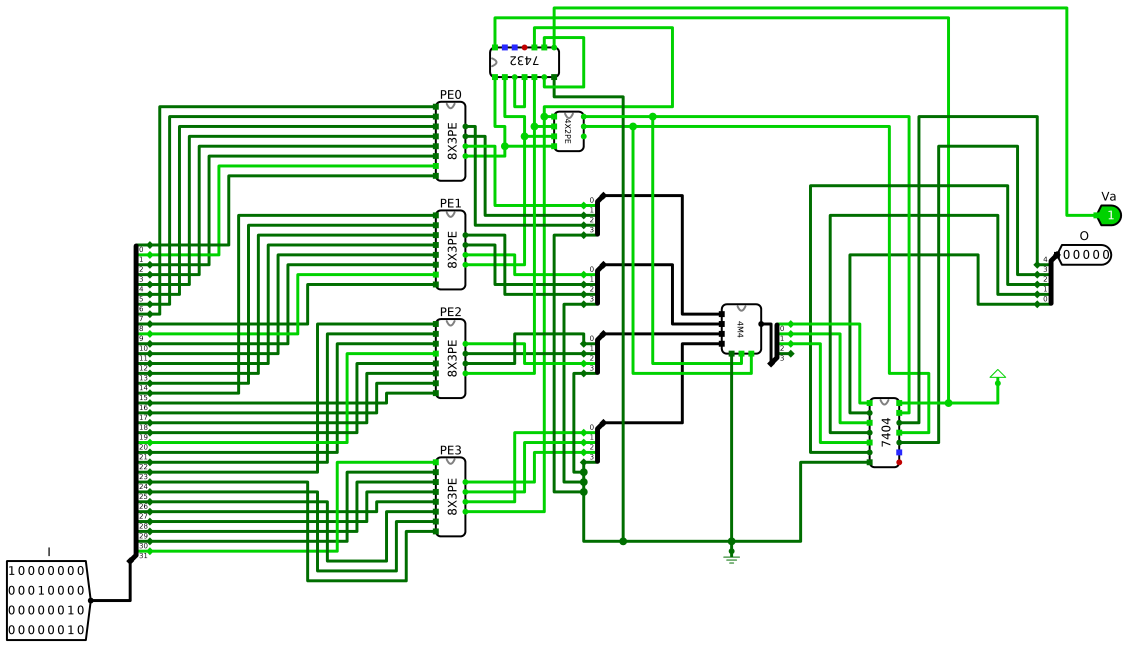
- 8 to 3 priority encoder (Constructed using 74147 IC) [8x3 Priority Encoder]
- 4 to 2 priority encoder (Constructed using 8 to 3 priority encoder of this library) [4x2 Priority Encoder]
- Circuit that finds the required number of bit shifts to normalize (Constructed using the aforementioned encoders) [ShiftFinder]



(a) 8×3 Priority Encoder



(b) 4×2 Priority Encoder



(c) Shift Finder

Figure 5: Encoder Circuits

3.6 Normalizer Library

This library(NormalizerLib.circ) contains the circuit Normalization which normalizes the output with required number of bit shifts with the help of encoder library. Also it generates the 12 bit number which will be added to the exponent to complete normalization.

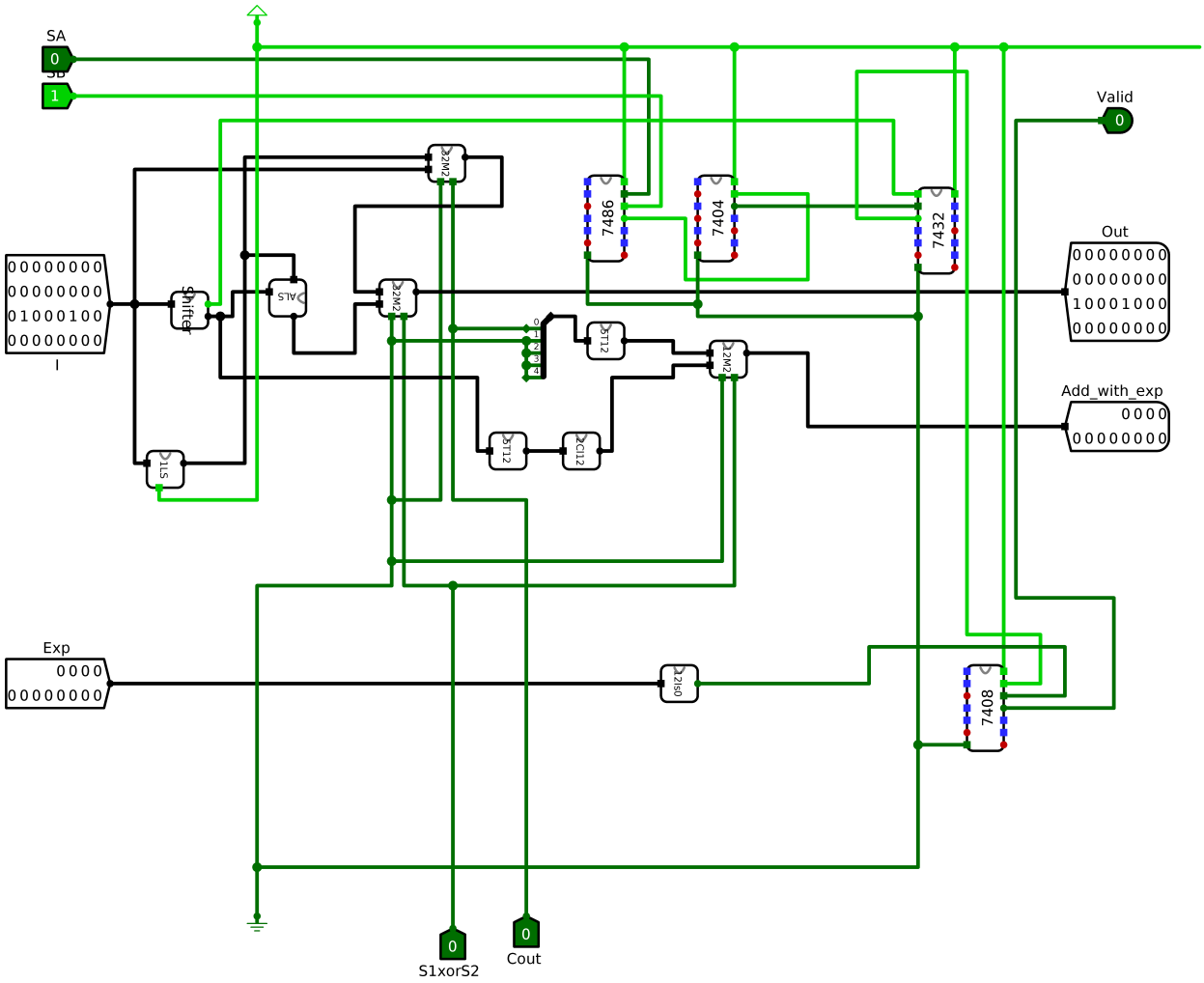


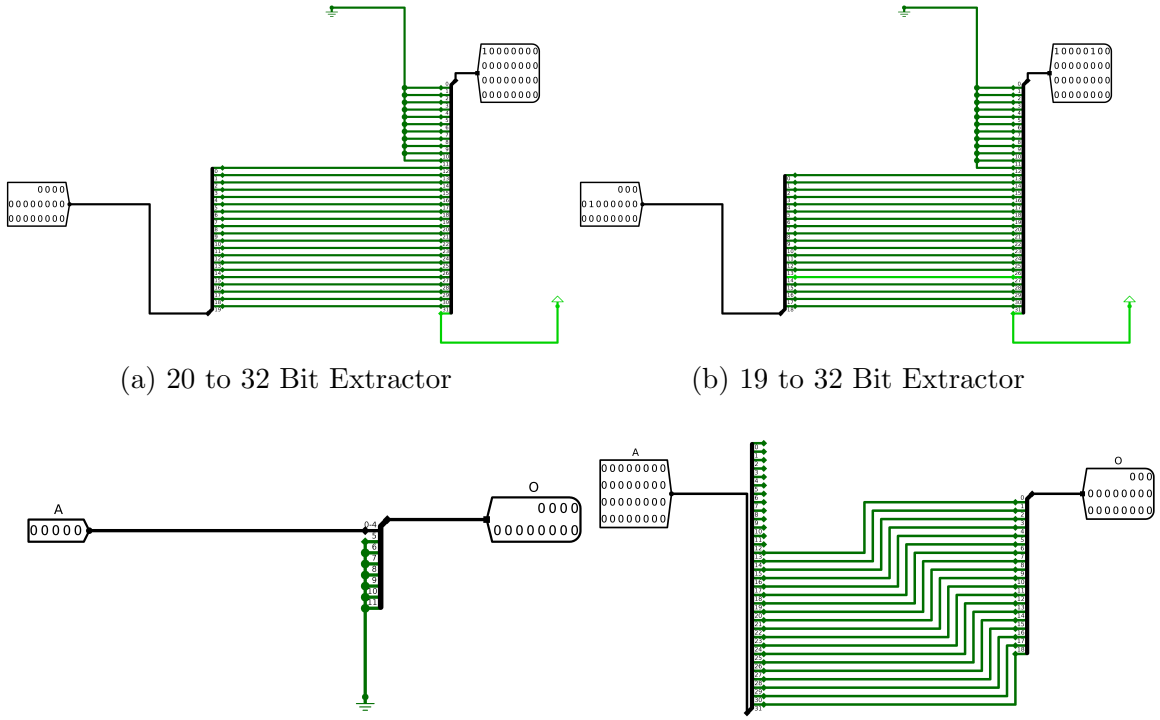
Figure 6: Output Normalizer

3.7 Miscellaneous Library

This library(MiscLib.circ) contains many circuits that helps to increase modularity and ease of implementation:

- Bit extenders or bit extractors (20 to 32 bit, 19 to 32 bit, 5 to 12 bit, 32 to 19 bit) [20to32, 19to32, 5to12, 32to19]
- 0 checkers to see if all the bits are 0 [4Is0, 12Is0, 32Is0]
- Input processor that separates the different parts of floating point representation [InputProcess]

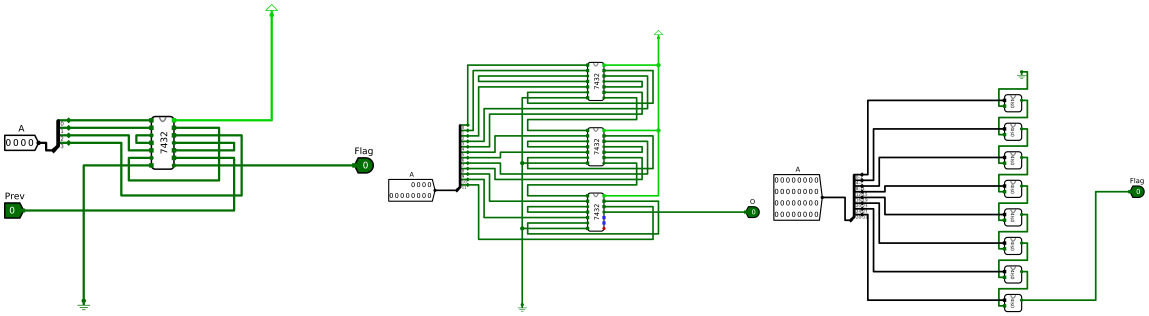
- Output processor that combines the different parts of floating point number [Output-Process]
- Checker that checks whether there is an input with zero in exponenet [InputCheck]
- Circuit that changes the sign bit to 0, if any number representing zero is input with a 1 as sign bit [0handler]
- 4 bit OR, 32 bit OR [4OR, 32OR]
- Circuit that determines if rounding up is needed [Rounding]



(a) 20 to 32 Bit Extractor (b) 19 to 32 Bit Extractor

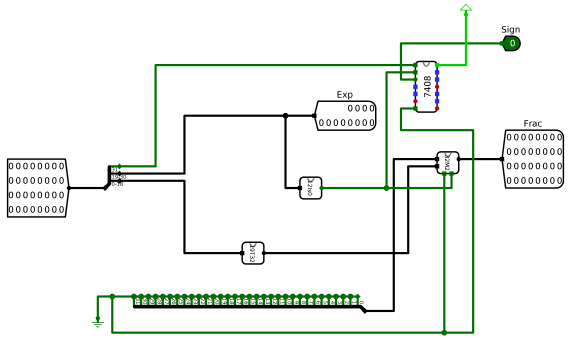
(c) 5 to 12 Bit Extractor (d) 32 to 19 Bit Extractor

Figure 7: Bit Extractors

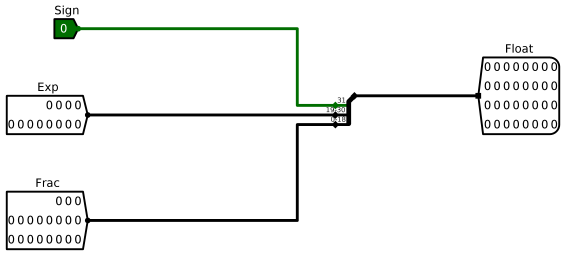


(a) 4 Bit Zero Checker (b) 12 Bit Zero Checker (c) 32 Bit Zero Checker

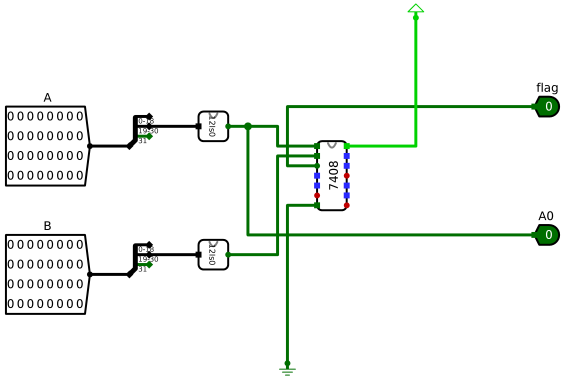
Figure 8: Zero Checkers



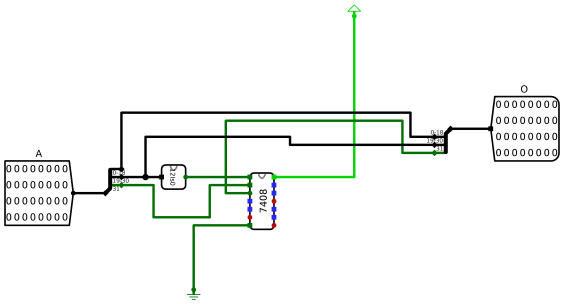
(a) Input Processor



(b) Output Processor

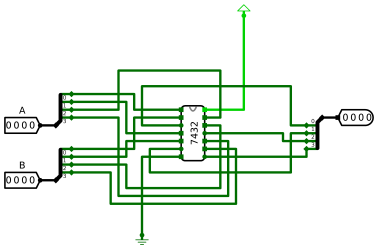


(c) Input Checker

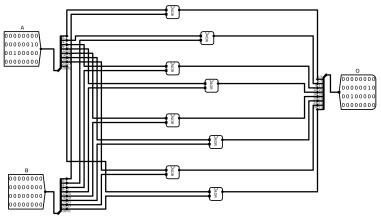


(d) Zero Handler

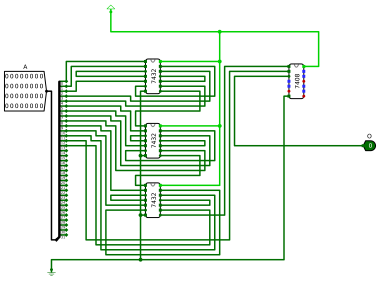
Figure 9: Input Output Processors



(a) 4 Bit OR



(b) 32 Bit OR



(c) Rounding Determiner

Figure 10: Other Useful Circuits

3.8 Rounding Circuit

The name of this module is RoundingMachine.circ. It contains the circuit RoundingMachine that uses the Rounding circuit in the miscellaneous library and does the rounding of the mantissa.

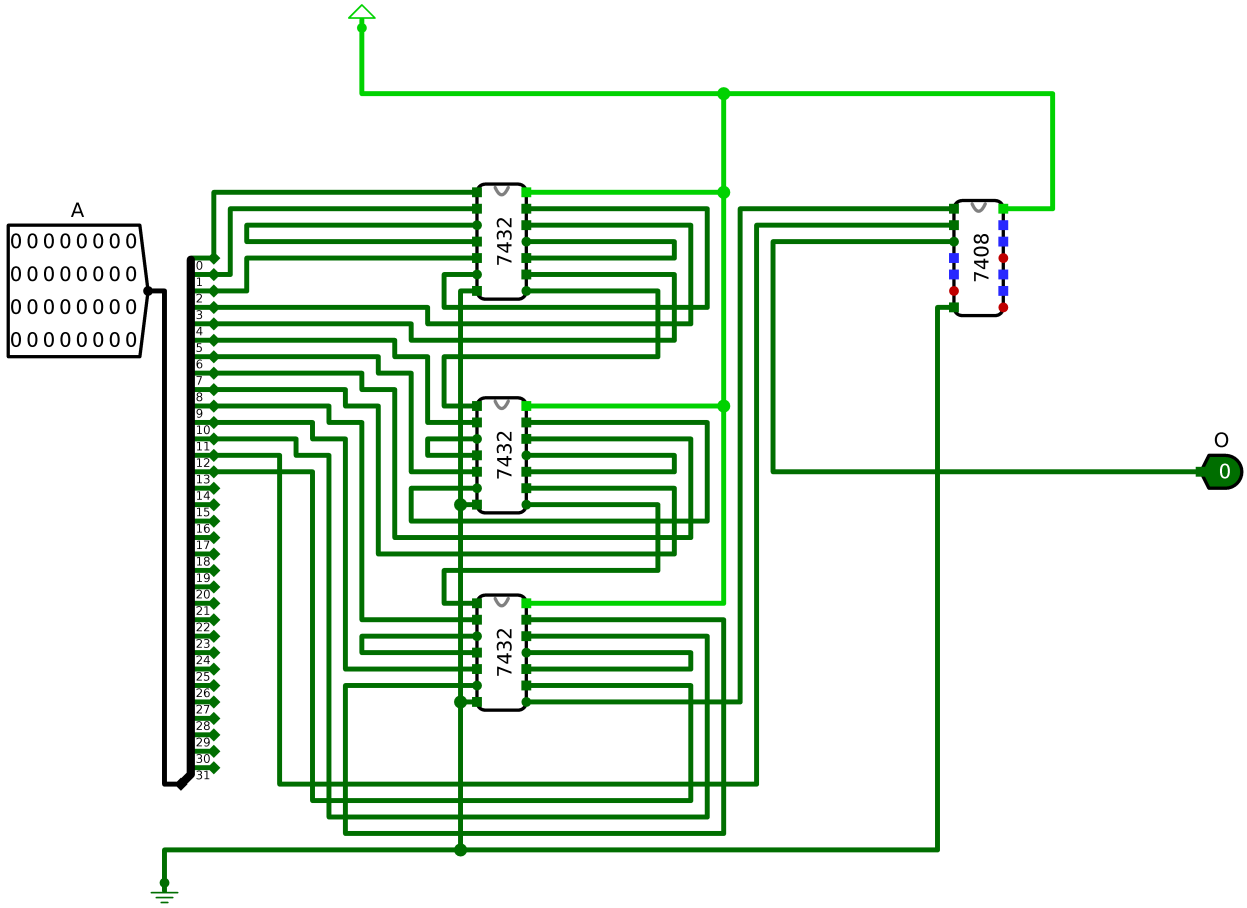
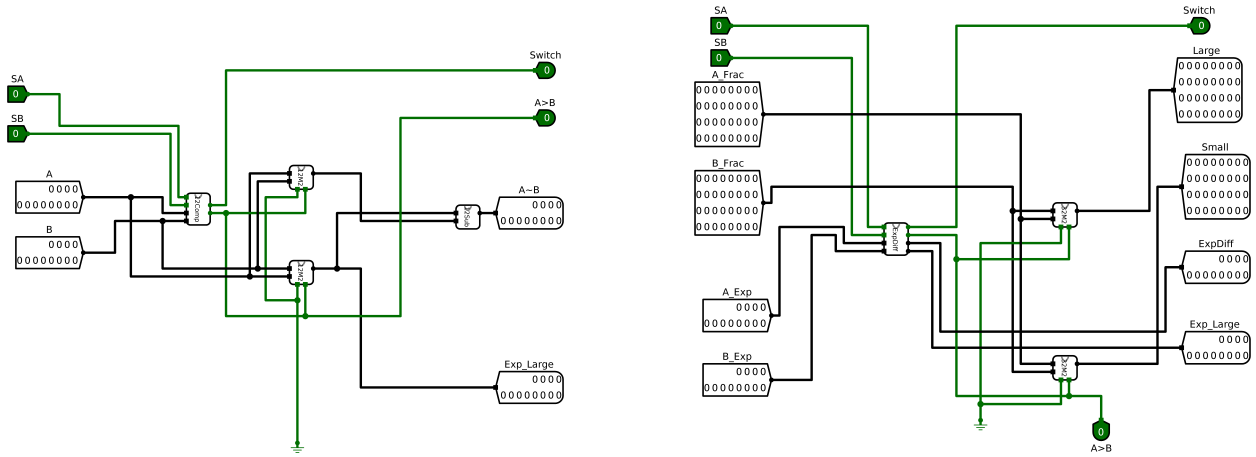


Figure 11: Rounder

3.9 Processing the Input

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

- Exponent differentiator that calculates the difference between the exponenets of two inputs [ExpDiff12]
- A circuit that determines which input is greater than the other, greater exponent etc for control decisions [InputHandle]



(a) Exponent Differentiator (b) Input Handler

Figure 12: Input Operator

3.10 Adder Module

This module(Added.circ) encapsulates all the works from taking input up to adding/subtracting. This module includes the circuit AddedInp which does this job.

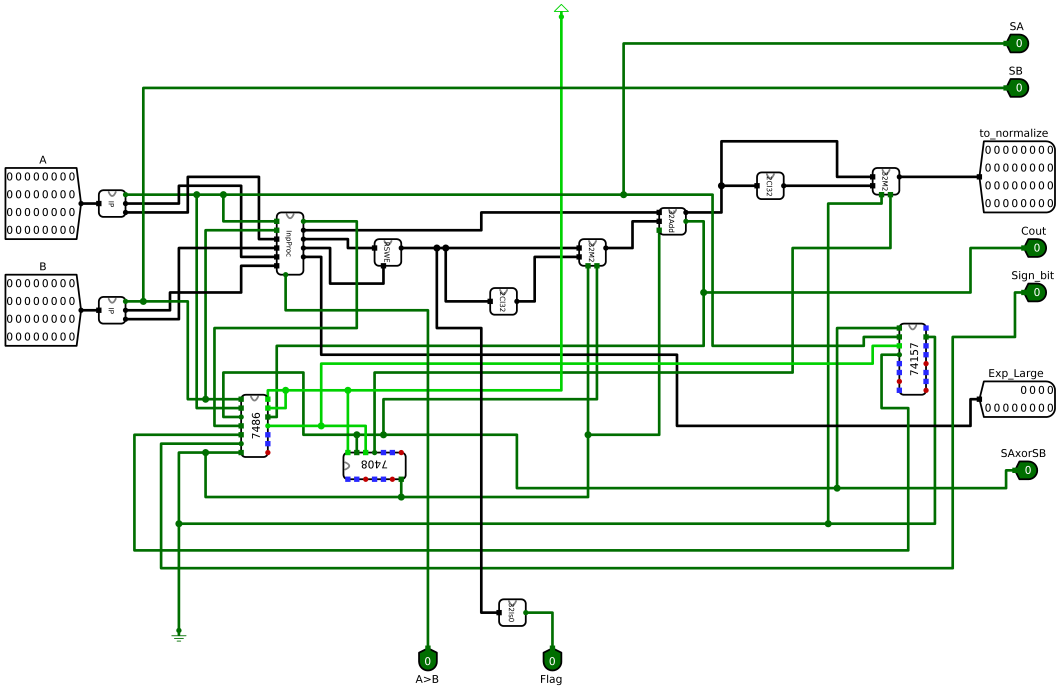


Figure 13: Adder

3.11 Floating Point Adder

This module(FPA.circ) is the final module that combines other modules and uses the libraries to completely implement a floating point adder. It has the circuit FPA that is the actual floating point adder. The module includes another circuit main which demonstrates the work of FPA abstracting internal componenets.

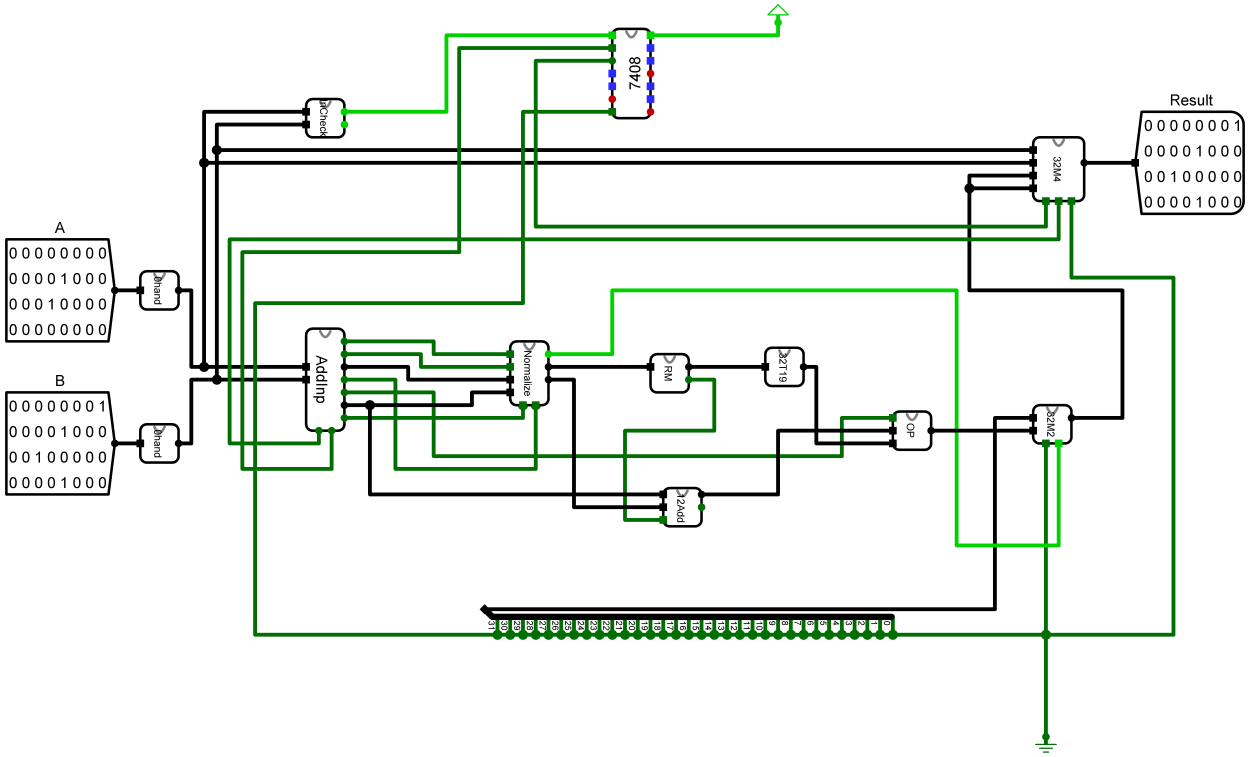


Figure 14: The FPA

3.12 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

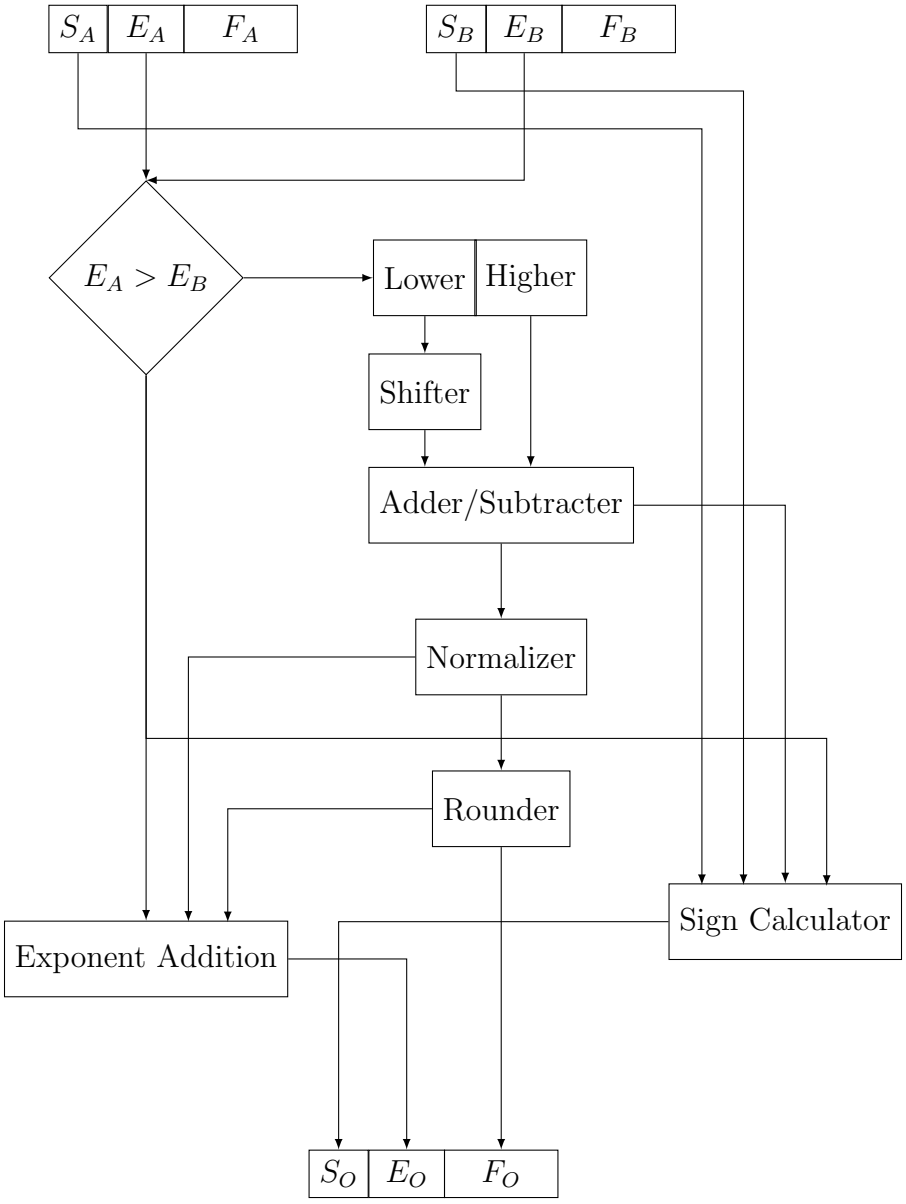


Figure 15: Flow chart of the addition/substraction algorithm

5 High-level Block Diagram of the Architecture

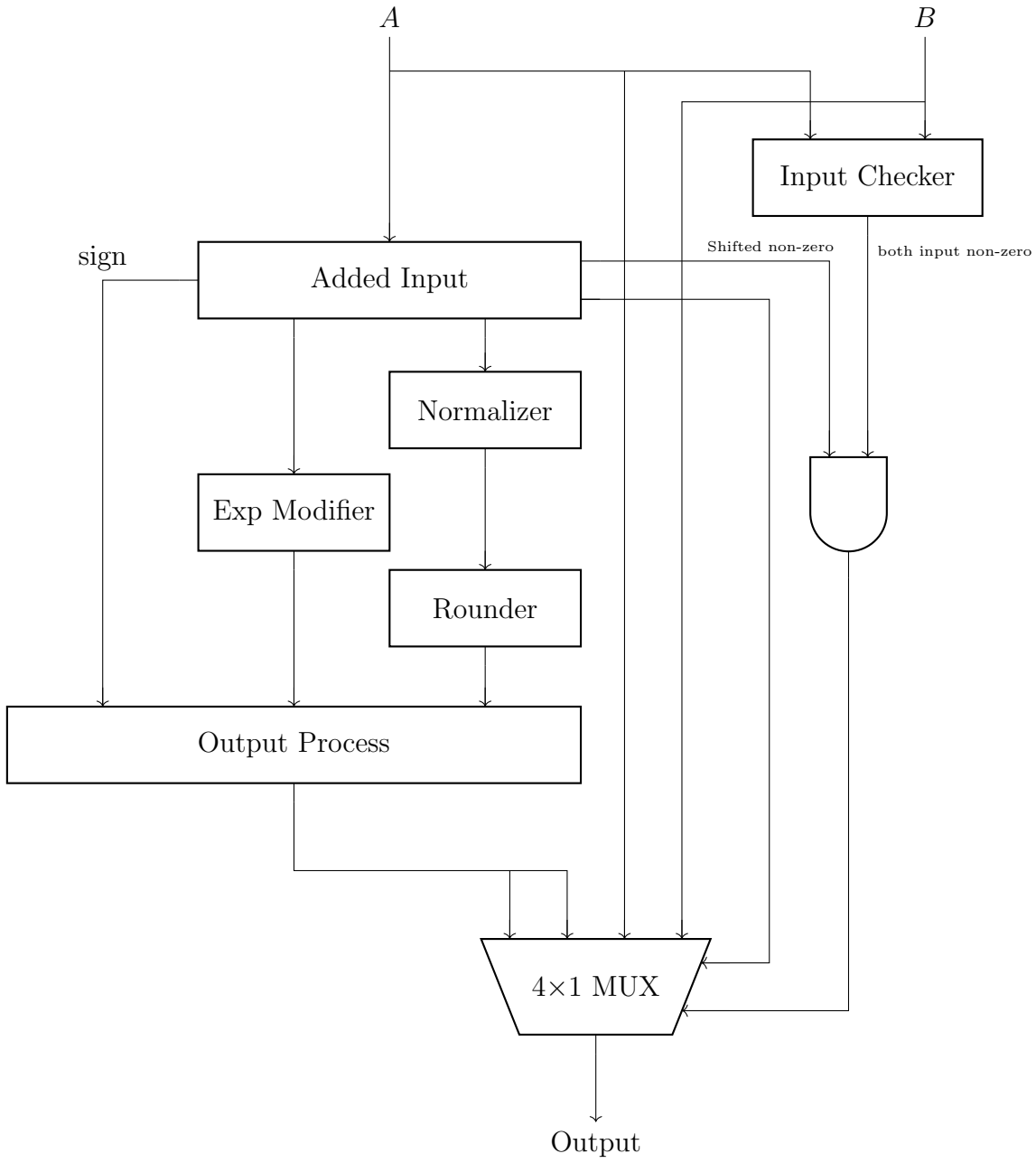


Figure 16: Block Diagram of FPA

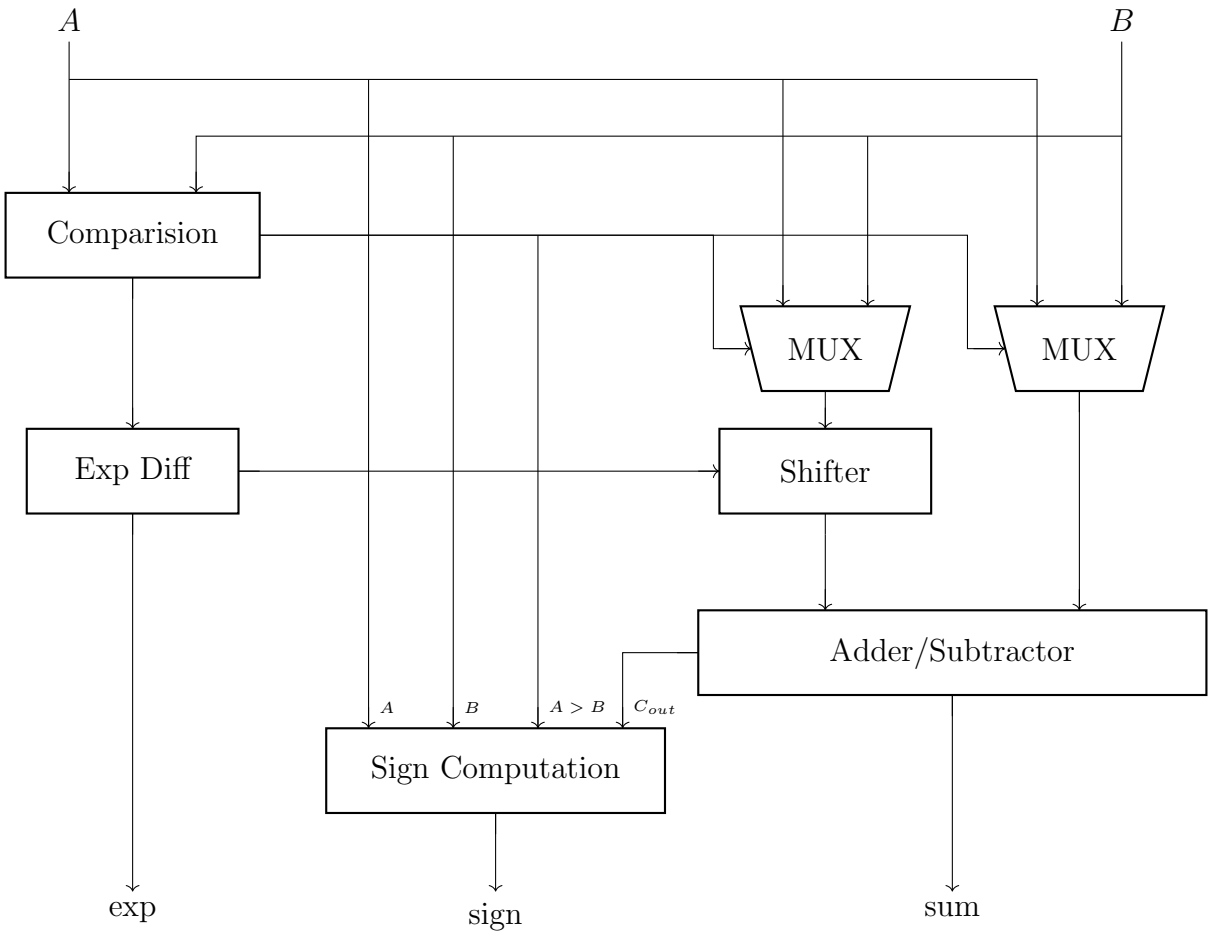


Figure 17: Block Diagram of Added Input

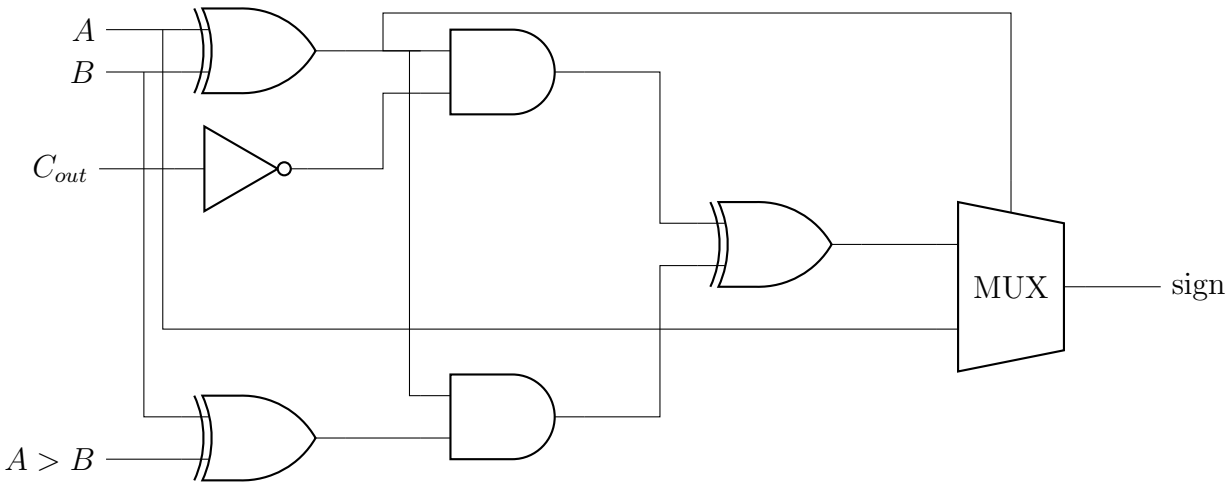


Figure 18: Block Diagram of Sign Computation

6 Comprehensive Design Description

6.1 Comparing the Exponents and Aligning Radix Point

To add two floating-point numbers, the radix points must be aligned. This is typically done by shifting the input with the smaller exponent to the right to align it with the larger input. We have used a comparator to compare the exponents of the two inputs and a 12-bit subtractor to calculate the difference between them.

6.2 Shifting

Multiplexer-based shifters have been used to perform all shifting operations. To obtain bit shifts of any arbitrary amount up to 31, we have used five 32-bit multiplexers. The multiplexers can shift respectively 1, 2, 4, 8 and 16 bits. Combining them, we can get any number of shifts up to 31, as is shown in 4f and 4l. In case of aligning two fractions, we may need more than 31 shifts which result in all bits being 0. Moreover, the input to the shifter circuit should be the difference of the exponents. Therefore, the lower 5 bits are be used for shifting when all the upper 7 bits are clear. Only in that case, the shift amount does not exceed $31(2^5 - 1)$. Otherwise, the fraction will be 0(all bits cleared), which is done in 4m.

6.3 Normalization

In order to normalize followed by addition operation, we employed a priority encoder to identify the position of the most significant set bit, which is the leftmost one. Four 8 to 1 priority encoders are used to implement this. The result can be upto $31(0_{th}to31_{th})$, hence consisting of 5 bits$(0 - 2^5 - 1)$. Output of the encoders are lower 3 bits of the result. Upper two bit and outputs of the encoders are selected using their valid bits. If left most encoder gets a valid input, other encoders will not be selected and upper two bits will be 00. So the range of the numbers generated will be from 0 to 7 in decimal. Similarly, when the second encoder is selected upper two bits will be 01; 10 and 11 for third and fourth ones respectively. The selection bit of the multiplexer is passed from a 4 to 2 priority encoder whose inputs are the valid bits of the aforementioned priority encoders. If inputs to neither of the priority encoders are valid, it means the fraction contains only 0s. This can happen in two cases.

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

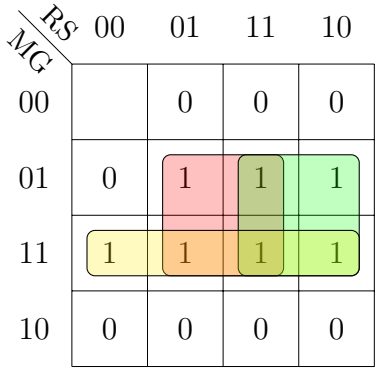
The first case can appear if two numbers of type $1.00 * 2^{x-1}$ and equal exponents are added. So, we can check whether the signs of both input are same. If they are, we will continue the algorithm and increase the exponent by 1. If the signs are opposite, the result is actually 0. So, output is 0.0.

6.4 Rounding

In order to add and subtract the mantissas, we employed a 32-bit adder. However, we can only retain 19 bits, so we may need to round the result in some instances. We consider $20th$ bit as Guard bit, $21th$ as Round bit. If any of the bits on the right of Round bit is set, Sticky bit will be set. Otherwise, it is clear. We need to consider these cases:

$19th$ **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 19th bit is 0, we need to do nothing, hence truncation. Otherwise, we will round up. For simplification we will use K-map. ($19thbit = M$)



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

If flag is set, 1 is added at the $19th$ position. Otherwise, the bits beyond the 20th position are truncated.

6.5 Computing the Sign Bit

There are certain considerations to take into account when calculating the sign bit. If the signs of the two inputs are the same, then the sign of the output will be either of the signs of the inputs. If they are not the same, we have to consider two things. Firstly, we need to find the sign of the output of the adder. 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. However, this sign may not always be correct because we always subtract the input with the smaller exponent from the input with the higher exponent. When the signs of the inputs are opposite, i.e, if the input with greater exponent is negative and the other one is positive, 1 gives opposite to the correct. To fix this we have introduced a variable switch. This switch bit will alter the sign bit in those cases. Our sign bit will fail when the sign bits of inputs are opposite and the input with greater exponent is negative. In that case switch will become 1 and will alter the sign. Equation of switch bit 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:

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

Equation 3 will be used in case the signs of the inputs are different. Otherwise, The sign of the first input will be directly applied to the sign of the output. This selection will be made using a multiplexer.

6.6 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.

6.7 Increasing Precision

For 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 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. Therefore, this case was handled separately as stated in 6.6. As a result, the precision is increased by 2.

7 ICs Used with Count as a Chart

IC	Quantity
IC 7404	2
IC 7408	10
IC 7432	46
IC 7483	41
IC 7485	3
IC 7486	25
IC 74147	5
IC 74157	205
Total	337

Table 2: ICs Used with Quantity

8 Simulator used Along with the Version Number

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

9 Discussion

In this assignment, we put a concerted effort to ensure the novelty and utmost efficiency of the design. In a lot of instances, we took the difficult route of implementing the modules by ourselves or coining an algorithm just to avoid using additional bits. This might have cost us huge efforts, but we ended up achieving a design we can claim to be as novel as it can get.

As explained in 6.7, we did not reserve a separate bit for capturing overflow or the sign of the result. This allowed us to improve the precision of our implementation by 2 digits. However, it took some time to come up with the proper algorithm to correctly compute the result. We utilized the carry out from the adder to work around this limitation of no extra bit.

We also put a great deal of effort into fine-tuning our design and thoroughly testing it. To be absolutely spot on with our testing, we wrote a C++ program to mimic our algorithm and then match its outputs with the ones of the adder. In several stages, it helped us rectify parts of our algorithm.

In addition to the technical aspects of our implementation, we also took great care to ensure the self-contained nature of our floating point adder. That is to say, rather than relying on external libraries, we implemented all the modular components of our adder by ourselves. This not only allowed us to fully understand the inner workings of our tool, but also made it more reliable and easy to maintain. From shifting bits using multiplexer to finding the leftmost set bit using priority encoders, we did it all.

Interestingly, this did pose a challenge at one point. As part of the report, we are expected to count the number of different ICs used in the design. But because we did so many abstractions, it was virtually impossible to count the number by hand. Module A might contain three instances of module C, module C might again contain two B's, another module D might comprise of A's, it is a humongous task to keep this track manually. We revisited the famous Depth First Search (DFS) algorithm to solve this problem. We constructed a graph considering the modules as vertices and their inter-dependencies as edges, and ran the DFS algorithm on it. This was another interesting bit of this assignment, that we really enjoyed doing. The source code can be found here.

Overall, the design and implementation of the floating point adder was an interesting task and we learned many new things along the way.