Bangladesh University of Engineering and Technology

CSE306 : Computer Architecture Sessional

Assignment-2

Floating Point Adder

Section – B1

Group – 05

Participating Rolls:

1905071 – Tareq Ahmed
1905078 – Mehreen Tabassum Maliha
1905082 – Kazi Reyazul Hasan
1905084 – Wasif Jalal
1905088 – Mubasshira Musarrat

Date of Submission:
January 8, 2023

1 Introduction

A Floating Point Adder (FPA) is a combinational circuit which takes two floating point number as input and then it gives the sum of two given numbers as the output. In this assignment, we have to implement a 32-bit Floating Point Adder using a combinational circuit.

2 Problem Specification

In this assignment, we have 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 following representation:

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

Here, sign bit indicates whether the number is negative or positive with value 1 and 0 respectively. The exponent is 12 bits long and can keep a value between 0 and 4095. To avoid negative value, exponent is biased by 2047. Exponent value 0 and 4095 is reserved for representing denormalized number, underflow, overflow, Not a Number etc. Thus, actual range of exponent in this problem is -2047 to 2047. After biasing by 2047 (2^{11}–1), the biasing will be between 1 to 4094. We have to require to store the function in normal form. We can hide the only 1 before decimal point as it is always 1 in normal form. Now, we store 21 digit of Floating point after decimal point. So, the actual value of floating point will be —

$$N = (-1)^S \times 1.F \times 2^{-2047}$$

Here, S = Sign ; E = Exponent and F = Fraction In this assignment, two numbers will be given in this form and addition operation will be performed on them and the result will be in similar form.

3 Flowchart of the Addition/Subtraction Algorithm

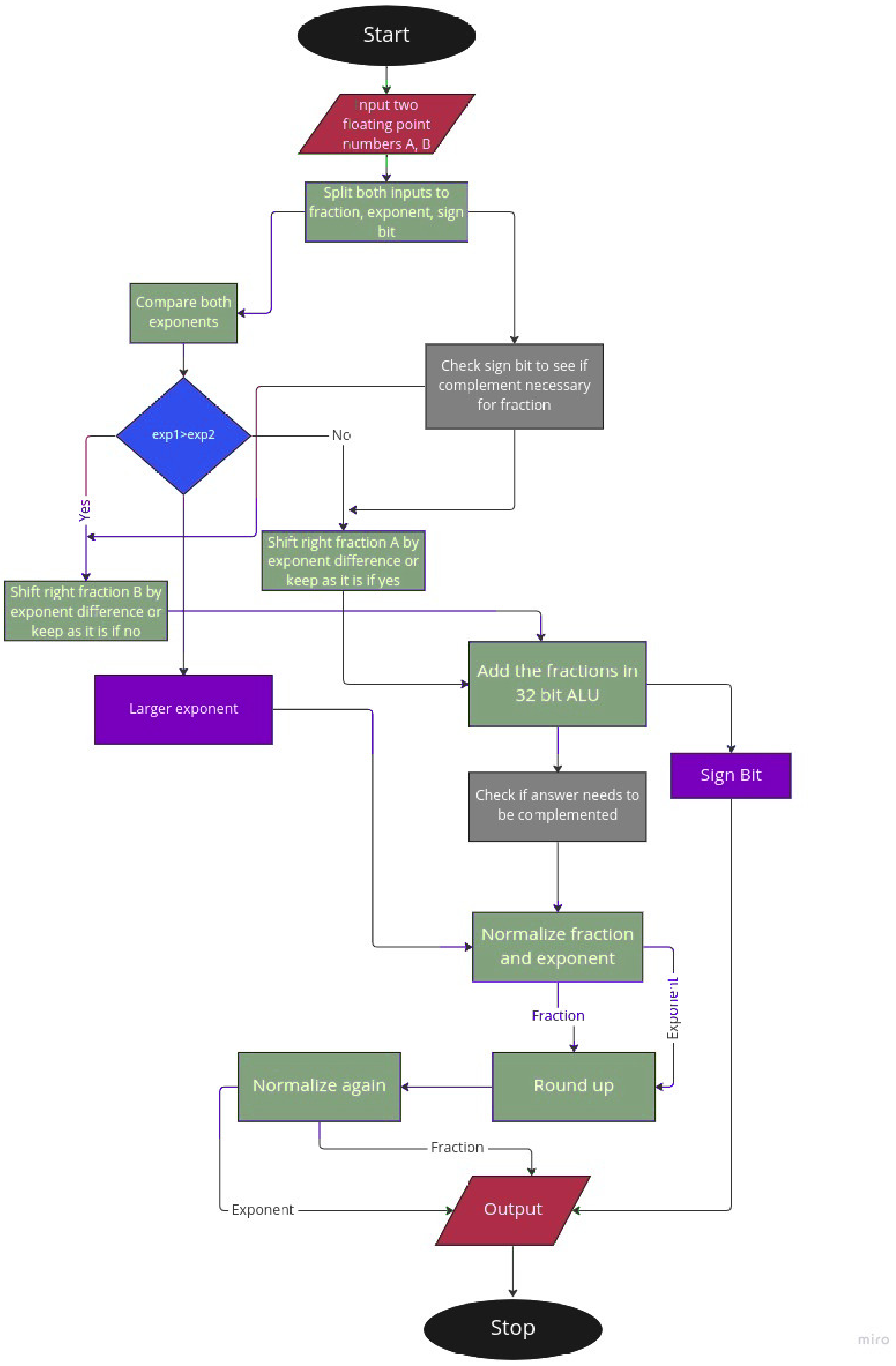

Start
Input two floating point numbers A, B
Split both inputs to fraction, exponent, sign bit
Compare both exponents
Check sign bit to see if complement necessary for fraction
exp1>exp2
No
Yes
Shift right fraction A by exponent difference or keep as it is if yes
Shift right fraction B by exponent difference or keep as it is if no
Add the fractions in 32 bit ALU
Larger exponent
Sign Bit
Check if answer needs to be complemented
Normalize fraction and exponent
Fraction
Exponent
Normalize again
Round up
Fraction
Exponent
Output
Stop
miro

Figure 1: Flowchart of the Addition/Subtraction Algorithm

4 Design Steps

4.1 Input and Output Handling

- Firstly, we take two 32-bit normalized inputs and split it into three parts. First 19 bits are for the fraction, 12 bits for exponent and the last bit as sign representation.
- At the very beginning, we check if one of the two inputs exactly means 0. For this, we used a 0 check component. If any situation occurs where one of the inputs is in fact 0, we use a 32-bit MUX (two for two cases) to simply show the other input as output because $A + 0 = A$.
- Now that we are clear of the 0 as output situation, we then move on to the normal addition process and check the sign bit and take a decision over a 22-bit MUX whether we need to complement the fraction before addition or not.

4.2 Comparing Exponents and Aligning Fractions

- Meanwhile, our 12-bit exponent is fed to the comparator, which gives us exactly three outputs.

  1. The larger exponent
  2. Whether $exp_1 < exp_2$
  3. The highest visualizable difference between the exponents (11111 is max)

  We won't be taking any difference greater than 11111 because this difference is used to shift and align exponents later on and for a 32-bit number, a shift greater than 11111 basically means the number will be all 0's.
- The ($exp_1 < exp_2$) output from the comparator helps us distinguish the larger and the smaller input of the two inputs, where we use two 22-bit MUXs (19 bits for fraction, one bit for marking the implicit 1 and two bits extra to store any sort of carry) and that comparator output as a selector bit.
- Now, the smaller input is fed into a 32-bit shift right component (not 22-bit because we don't want to lose the extra shifted bits which we will use for rounding later on) and the amount of shift will be the output of the highest difference between the exponents which also comes out from the comparator.

4.3 The Actual Addition

- After the shifting is done and the exponents are aligned, we take the two fraction parts (larger one directly from the MUX, smaller one from the shifter), make it 32 bit if necessary and use both as inputs for a 32-bit ALU for addition.
- After addition we check the sign bit to see if complementing is necessary. Another check with a MUX is if the addition came to be 0. If so, normalizing, rounding is unnecessary and we can simply treat that as output fraction, adding the sign bit and exponent.

4.4 Normalizing the Output

- Continuing with normal flow, now we will be required to normalize the processing output. The larger exponent and ALU output (complemented or not) is fed to the normalizer. For this, we will be remembering our implicit 1 which we planned to use as a marker, its index was on 29 when we sent the fractions for addition. Now, after addition, there can be two cases.

1. Carry, which will result the leading 1 to be at index 30
2. Leading 1 will move to an index lower or equals to 29

For the first case, a single right shift will be necessary to get our 1 marker back to 29 position. For the second case when it is lower than 29, we need the position of the leading 1. For that, we used a bit finder and we subtracted its index from 29 to find out how many shifts we need exactly to get our marker 1 back to 29 index. After that, a MUX checks if the first case is true or the second and outputs value accordingly. Now we have our fraction. We have to change our exponent accordingly based on the difference found from ALU.

4.5 Rounding and Re-normalizing

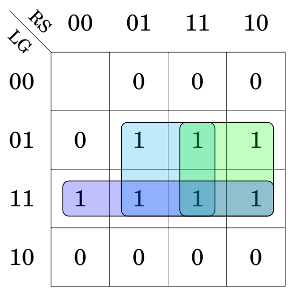
- For rounding, we take the input from the normalizer as input. As for our design, our 8th bit is sticky bit, 9th bit is round bit, 10th bit is ground bit and 11th bit is our LSB. We use a simple logic $G(S+R+L)$ for the rounding part. The LSB is necessary to ensure one case where G is 1, S is 0, R is 0 and the number is odd (we need to round it up to even). Rest is straightforward logic where G always has to be 1 to round up. The rounder gives us a 1 or 0 at LSB and all other 0's. The expression Y is the output of the rounder, which is to be added to the truncated mantissa for rounding.

4.5.1 Derivation of the expression Y

The truth table for binary fraction is as follows for a fraction where G, R, S are the three most significant of the bits that will be truncated, and L is the least significant of the bits that will be kept. ($(G,R,S) = (1,0,0)$ defines the midpoint between 001 and 111):

G	R	S	Y	
			L=0	L=1
0	X	X	0	
1	0	0	0	1
1	0	1	1	
1	1	X	1	

Upon plotting a K-map of Y from the truth table, we obtain —



LG \ RS	00	01	11	10
00		0	0	0
01	0	1	1	1
11	1	1	1	1
10	0	0	0	0

Therefore, the expression turns out to be —

$$Y = LG + SG + RG$$
$$= G(L+R+S)$$

- Now we take another 32-bit ALU to add the rounder output with our normal output.
- After that another normalizing step is necessary. Finally the output from normalizer can be treated as our desired output, adding the sign bit from the first ALU.

5 High-level Block Diagram of the Architecture

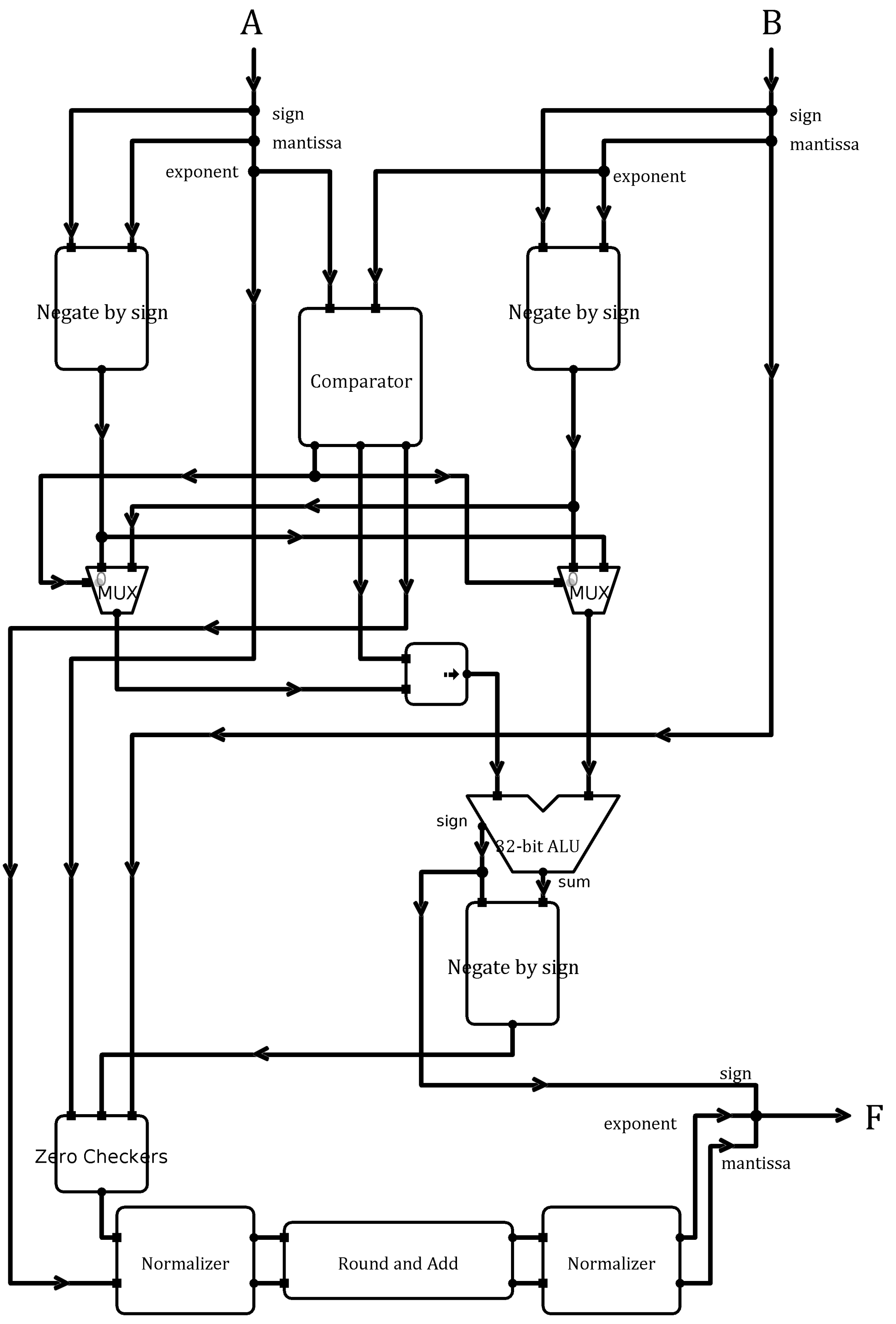


Figure 2: High-level Block Diagram of the Architecture

6 Detailed Circuit Diagram and Description of Modules

6.1 Multiplexers and Helpers

To make the task of multiplexing simpler, several 2-to-1 line multiplexers for specific bit widths were designed. The 12-bit MUX was made by cascading three quad 2-to-1 MUX circuits. Then the 32-bit MUX was made by cascading three 12-bit MUX circuits, and then it was fed 10 low inputs to emulate another 22-bit MUX circuit. A small circuit was built to check inputs and outputs for zero.

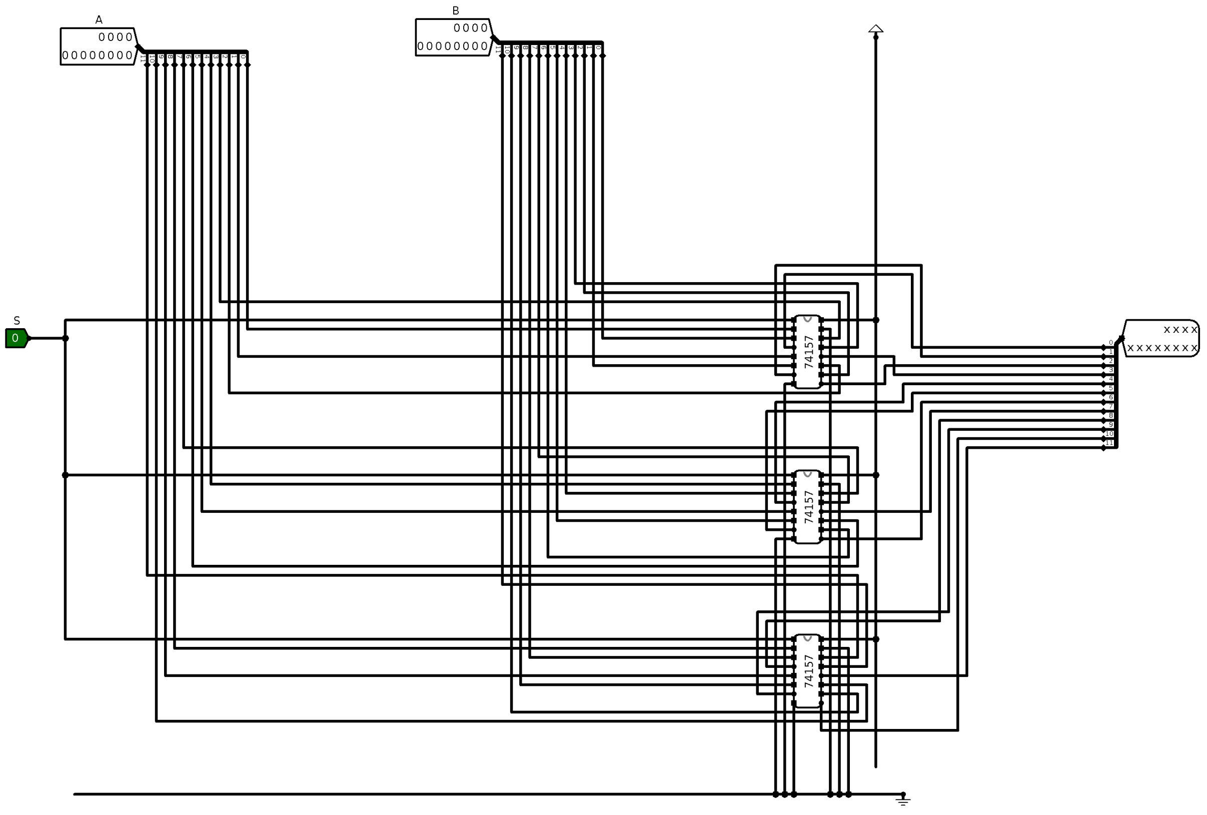


Figure 3: 12-bit 2-to-1 line multiplexer circuit

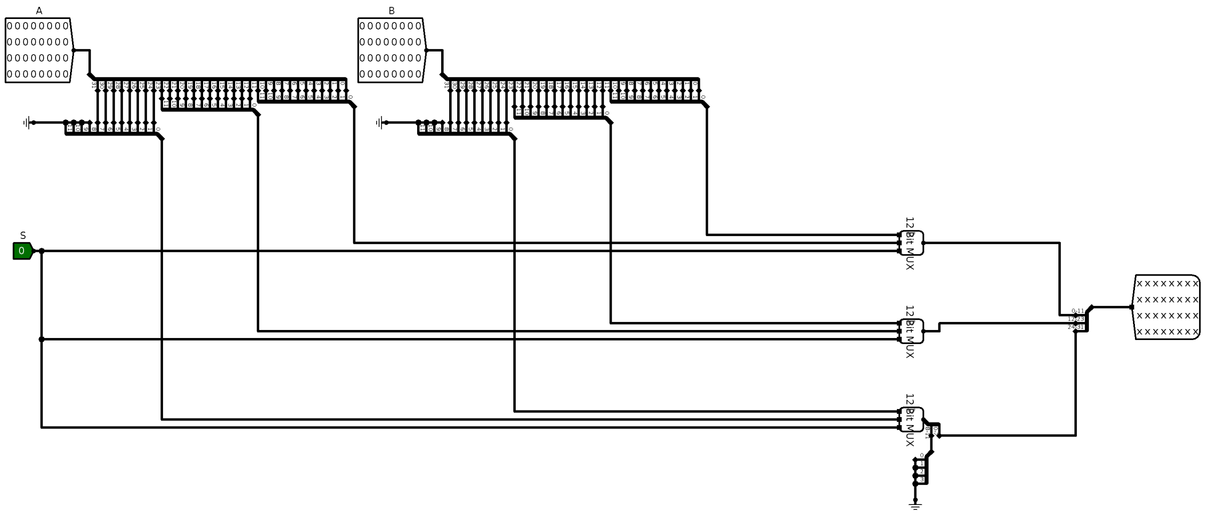
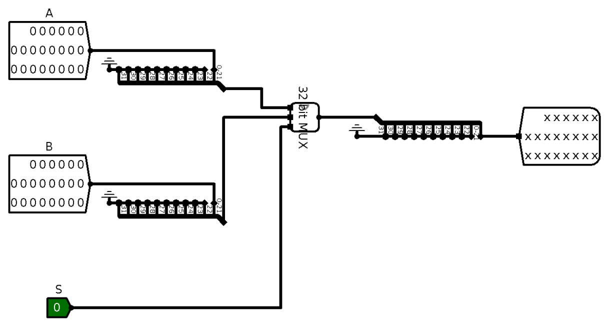
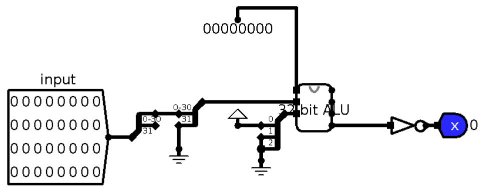


Figure 4: 32-bit 2-to-1 line multiplexer circuit



(a) 22-bit 2-to-1 line multiplexer circuit

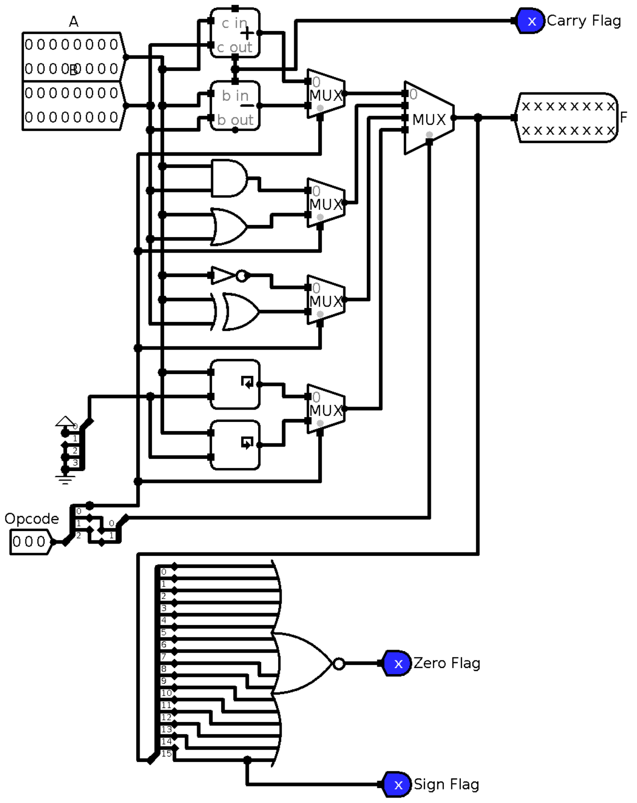


(b) Zero checker circuit

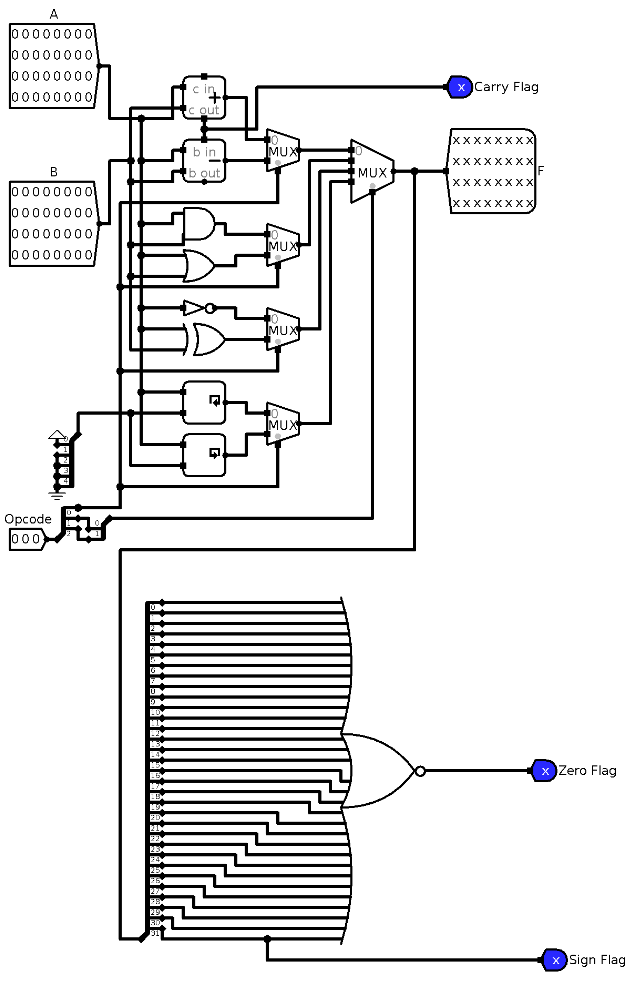
Figure 5: 22-bit MUX and Zero Checker

6.2 Arithmetic Logic Units

Both 16-bit and 32-bit ALUs were used for arithmetic across the circuit. They are identical to each other in design except for bit width, and support serveral operations with opcode selection. Only the addition and subtraction operations were used in the Floating Point Adder.



(a) 16-bit ALU



(b) 32-bit ALU

Figure 6: Arithmetic Logic Unit circuits

6.3 Comparator

The Comparator finds out the difference between two 12-bit exponents by using two 16-bit ALUs. Then it produces outputs using combinational logic as described in Section 4.2.

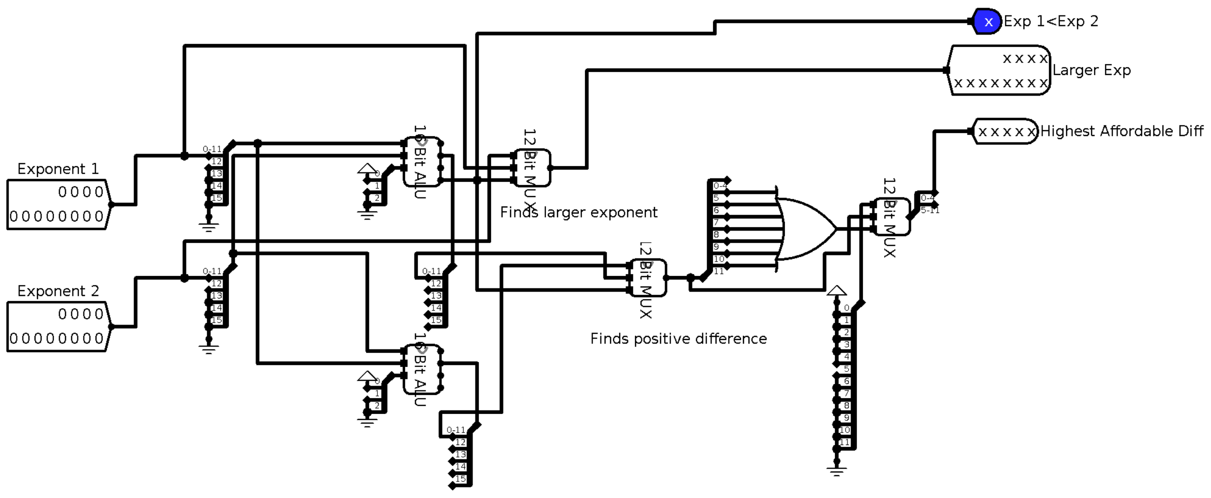


Figure 7: Comparator circuit

6.4 Normalizer

The Normalizer uses two 16-bit ALUs for arithmetic. The flow of logic in the circuit is described in Section 4.4.

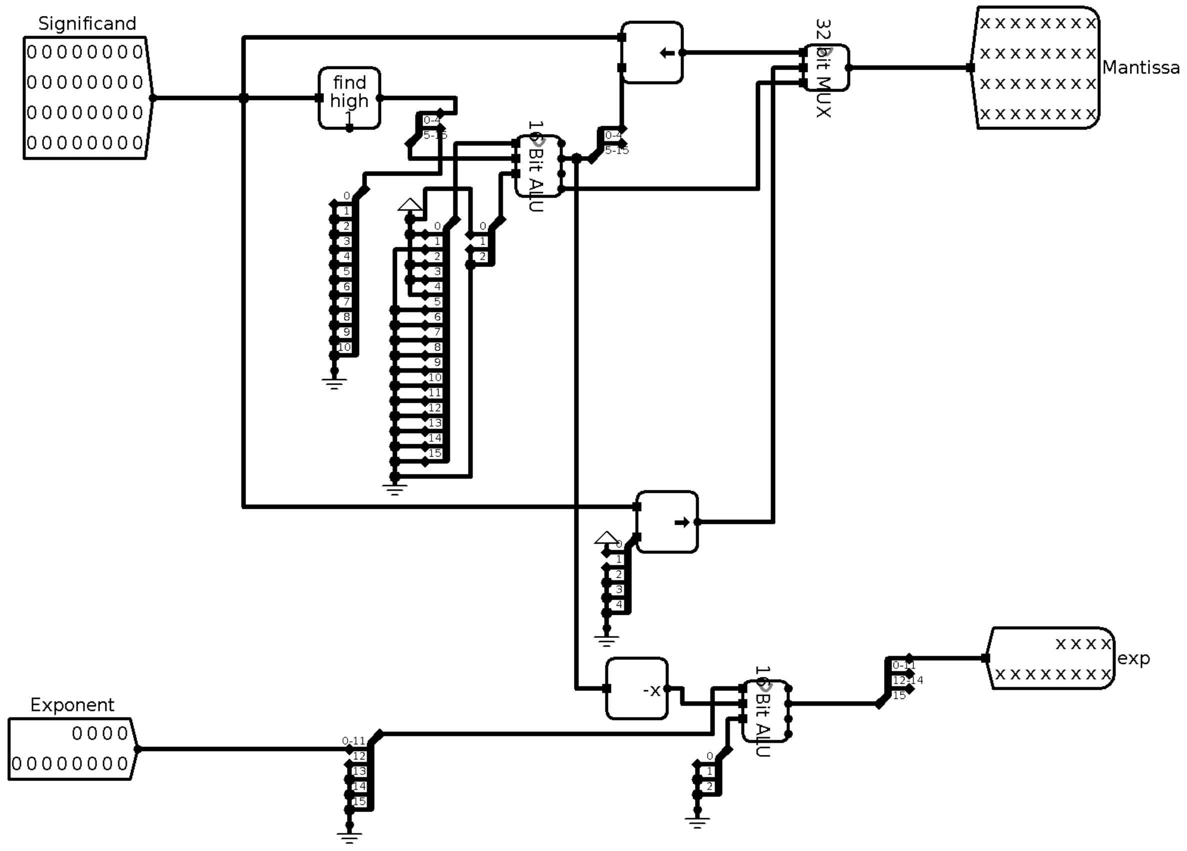


Figure 8: Normalizer circuit

6.5 Rounder

The Rounder just implements a Boolean expression with basic logic gates. Its detailed design is discussed in Section 4.5.

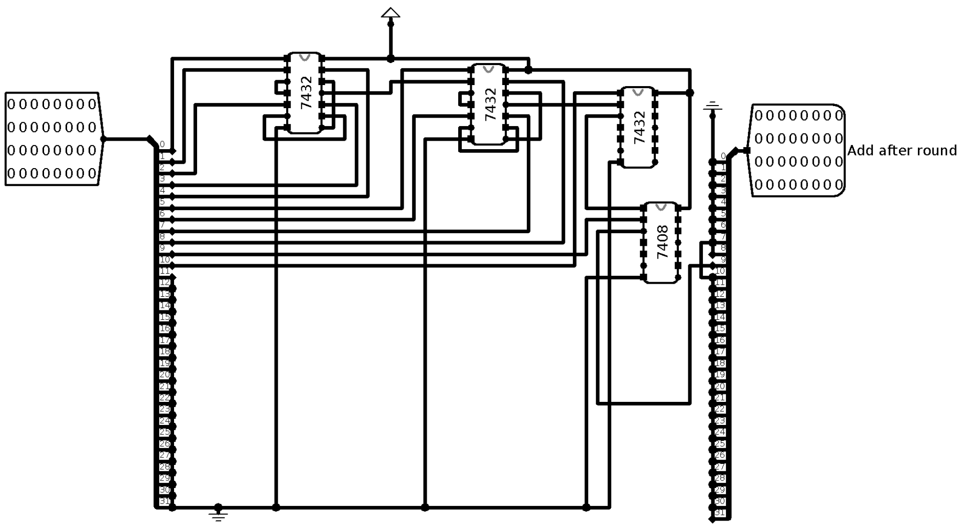


Figure 9: Rounder circuit

6.6 Decimal Viewer

We designed an extra tool to help us instantly visualize the values of the output and inputs in decimal notation, so that the arithmetic can be verified quickly. Binary fractions represented as integers (just by removing the radix point) need to be multiplied by $(10_{(10)})^n$ to shift their decimal point rightwards (so that the result is printable as an integer in decimal, with n digits after decimal), and then divided by $(10_{(2)})^r$ (where r is the number of bits after the radix point in the binary fraction). This arithmetic is straightforwardly implemented with multipliers and dividers in this module, and then the outputs are displayed as signed decimal by using probing apparatus of the simulator.

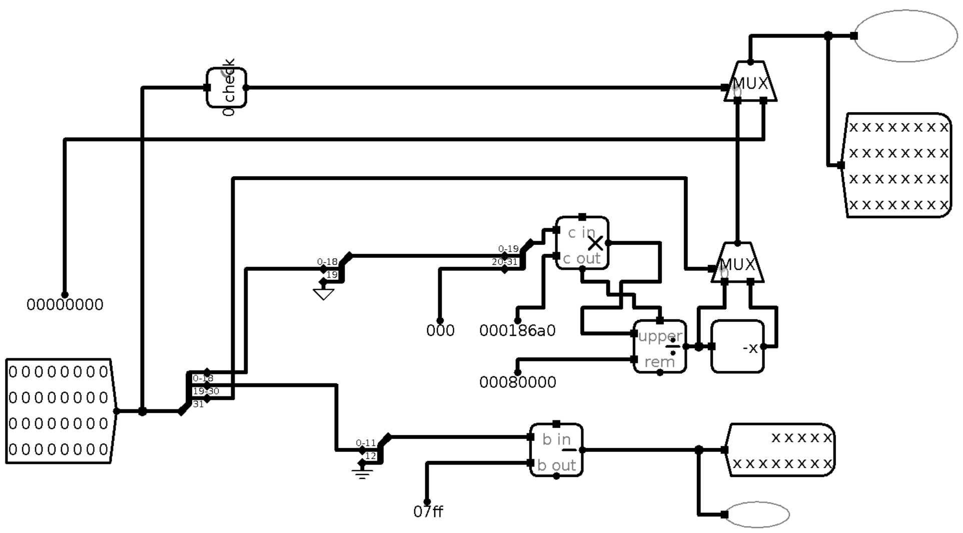


Figure 10: Decimal Viewer circuit

6.7 Assembled Floating Point Adder

Assembling all the parts discussed so far as indicated in Section 4, the full circuit of the Floating Point Adder is obtained. It will receive two floating point numbers of the assigned standard in normalized form and produce a normalized floating point number in the same format as output. It also shows the three parts of the output separately along with a decimal value.

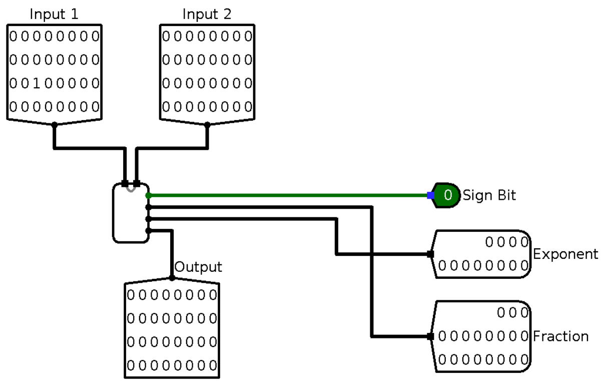


Figure 11: A schematic preview of the Floating Point Adder

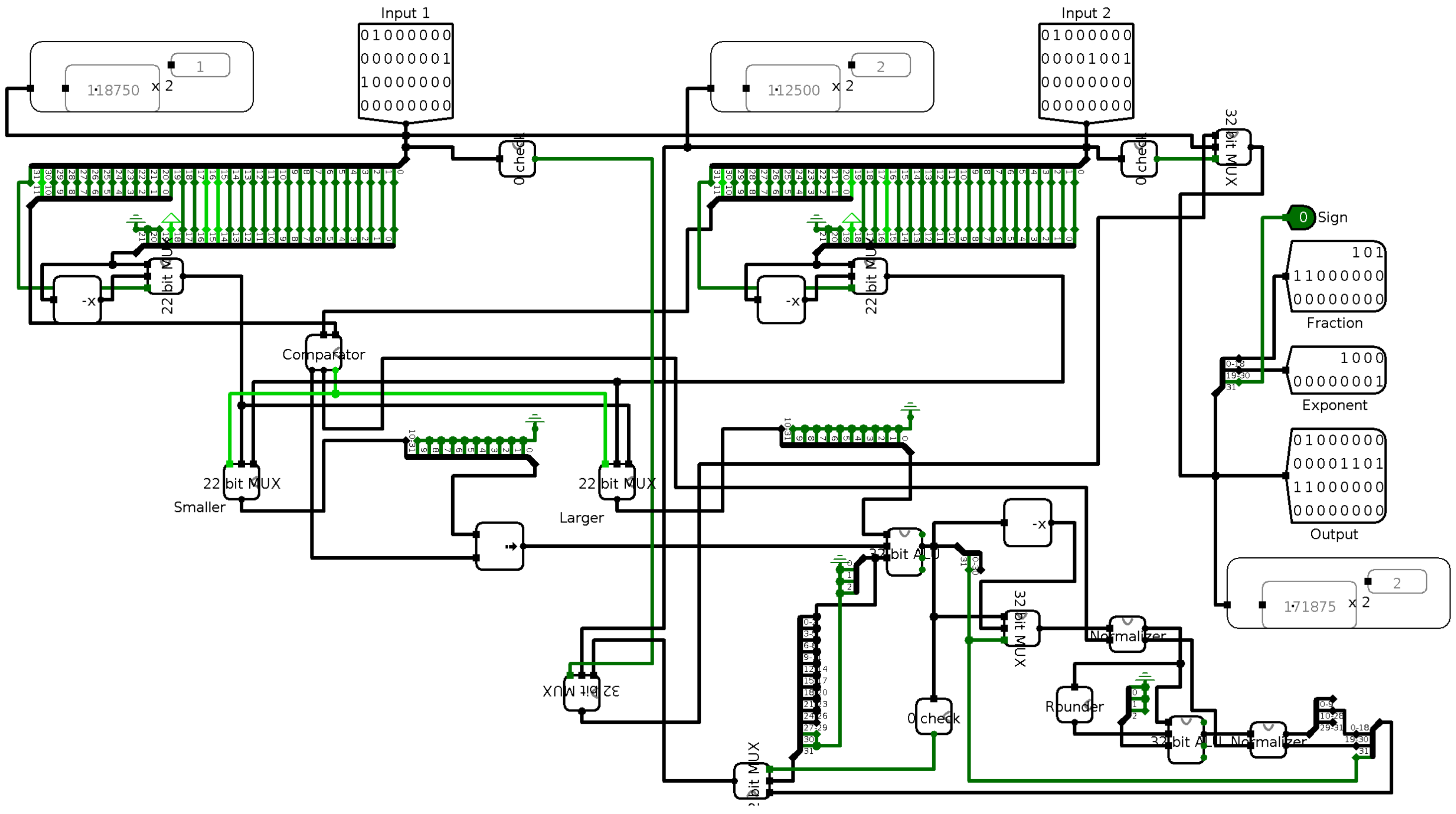


Figure 12: Full Circuit Diagram

7 Required Components

Component Name	Description	Count
NOT Gate	Multiple-input built-in	20
AND Gate	Multiple-input built-in	14
OR Gate	Multiple-input built-in	15
NOR Gate	Multiple-input built-in	14
XOR Gate	Multiple-input built-in	14
Multiplexer	16-bit 2-to-1	16
	16-bit 4-to-1	4
	32-bit 2-to-1	14
	32-bit 2-to-1	3
Adder	16-bit Binary Full Adder	8
	32-bit Binary Full Adder	6
Subtractor	16-bit Binary	8
	32-bit Binary	6
	13-bit Binary	3
Multiplier	32-bit to 64-bit Binary	3
Divider	64-bit to 32-bit Binary	3
Negator	16-bit 2's Complement	2
	22-bit 2's Complement	2
	32-bit 2's Complement	4
Shifter	32-bit Arithmetic Left Shifter	2
	32-bit Arithmetic Right Shifter	3
Bit Finder	Highest-order 1 index finder	2
IC-7408	Quad 2-Input AND Gate	1
IC-7432	Quad 2-Input OR Gate	3
SN74HC157N	Quad 2-to-1 Multiplexer, non-inverting	99

8 Simulator Name

Logisim ITA (Version 2.16.1.4)

9 Discussion

1. The floating point adder simulated here mainly has four core components.

    - Comparator
    - 32 bit ALU
    - Normalizer
    - Rounder

    The flow of input-output is explained in detail in design steps part. Given 19 bits for fraction, a rough precision of 5 numbers (in decimal unit) after decimal point was achievable mathematically and our floating point adder mostly maintains that except a few very extreme cases.

2. None of the components designed are very operation heavy. The operations are equally distributed to the components.

3. The ALU mainly provides different types of operations as a whole. However, the features of ALU used in the circuit were limited to addition and subtraction. The ALU itself was desined for versatile operations. The design contains a 16-bit ALU and a 32-bit ALU which was simulated similarly.

4. Total three MUXs are designed for convenience of selection. The 12-bit MUX is designed on IC-level and the other two are cascaded.

5. The splitters are an integral part of the total circuit for maintaining input and output bits properly.

6. A Decimal Viewer was designed for convenience as well. This is not a part of the core circuit operation but as floating point representation is quite intuitive, being able to see the converted decimal value of input-output eases difficulty of understanding.

7. Output was separated into three parts like given inputs to clearly differentiate the result values.

8. The floating point adder is designed assuming there will no extreme cases of underflow and overflow as mentioned in specifications.

9. Two extra bits and a bit for implicit 1 is taken for fraction addition. The extra bits do not cause any precision loss visually and is safe to take for making life a bit easier.

10. There are some extra components like Check 0 which also play crucial part to make calculations easier for the core components.

11. The design itself is broad and expendable. Some components like Rounder is designed by ICs, whereas modules are used for parts like Normalizer to keep the design clean.

12. Overall the circuit was tested with various input combinations and the answers were accurate. It can be then concluded that the simulation works as expected.