CSE 210

Computer Architecture Sessional

Assignment-2: Floating Point Adder

Section - A2
Group - 02

Members of the Group:

1. 2105045 - Sadia Binte Sayad
2. 2105058 - Elias Mainur
3. 2105060 - Jonayed Mohiuddin

1 Introduction

A floating-point adder is a critical component in computer arithmetic, designed to perform addition operations on floating-point numbers. Floating-point addition (FPA) is an essential operation in computer arithmetic, allowing for precise addition of real numbers across a broad range of values. Floating-point numbers are represented in a format similar to scientific notation, with three main components: *sign*, *significand* (or *mantissa*), and *exponent*, each associated with a specific *base*. The base, typically 2 in binary systems, defines the system's numerical scale and significantly impacts the representation's range and precision. Various standards define the specific bit allocation for each of these segments. The widely used *IEEE 754* standard, for example, allocates 1 bit for the sign, 8 bits for the exponent, and 23 bits for the significand in a 32-bit floating-point number.

In this format, the exponent is not stored directly. Instead, it is *biased* by adding a constant, known as the bias, to the actual exponent value before storage. This biasing allows the exponent to be stored as an unsigned integer, simplifying the circuitry required for comparing exponents of two numbers. The bias value is calculated as $2^{n-1} - 1$, where n is the number of bits allocated to the exponent field. For an 8-bit exponent, the bias is $2^7 - 1 = 127$. Thus, an exponent of zero is stored as 127, a positive exponent of 1 as 128, and so on.

Biasing plays a crucial role in simplifying floating-point arithmetic and ensuring compatibility across different computing platforms.

In the scientific notation, there is always a 1 in front of the radix point unless the number is 0. So storing that 1 is redundant; hence the significand bits always store the fractional part.

The overall representation is as follows:

$$(-1)^{\text{sign}} \times (1 + \text{significand}) \times 2^{(\text{exponent} - \text{bias})}$$

Here, the *exponent* portion represents the exponent stored in memory.

Floating-point representation enables computers to process both extremely large and tiny numbers, which would be difficult to manage with integer-only arithmetic. Floating-point arithmetic includes handling special values like infinity, NaN (Not a Number), and subnormal numbers. These cases provide a robust framework for managing edge cases in numerical computations without crashing programs. It should be kept in mind that there can be scenarios, especially for larger exponents, where the exact floating-point number cannot be stored. In those cases, the memory stores the nearest floating-point number. Since floating-point numbers have limited precision, rounding errors are inherent in FPA operations.

2 Problem Specification

The assignment asked us to create a floating-point adder circuit that takes two floating-point numbers and calculates their sum. The representation was defined as follows:

Sign	Exponent	Significand
1 bit	9 bits	22 bits

Table 1: Problem specification

3 Flowchart

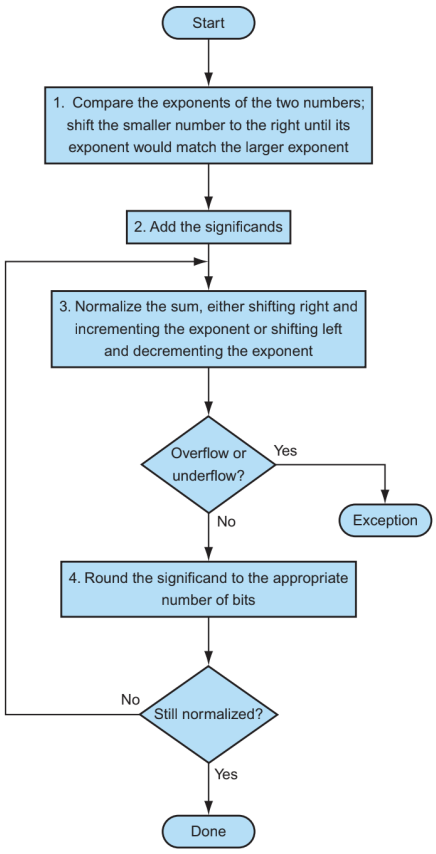


Figure 1: Floating-point addition flowchart

4 High Level Block Diagram

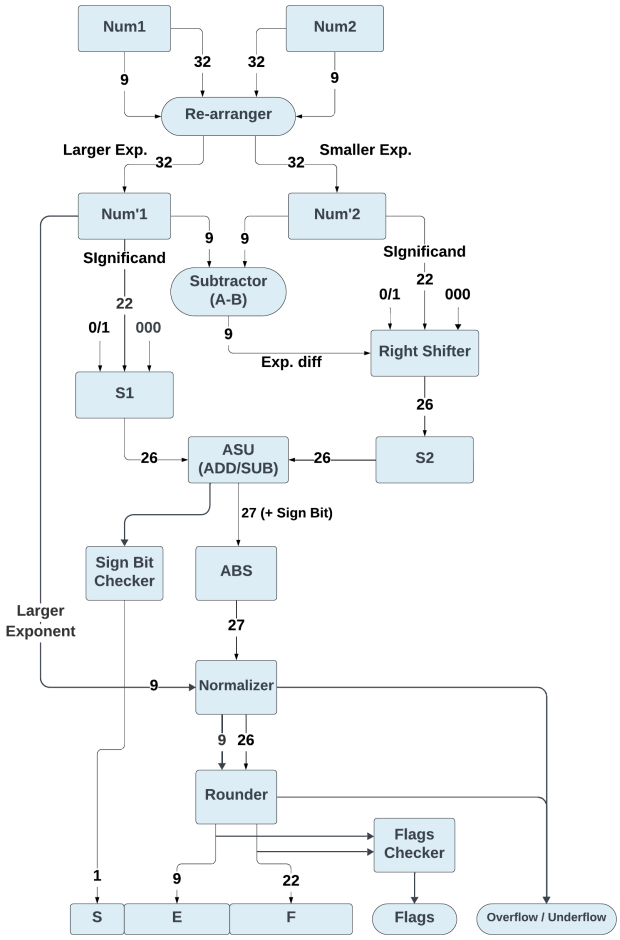


Figure 2: High-level block diagram

5 Description of Modules

Each high level module and how they work is listed below

5.1 Shifters

In designing shifters for normalization and exponent equalization, flexibility and resource efficiency were prioritized. Instead of creating individual shifters for every possible shift amount, only five modules were developed, capable of shifting by powers of two: 1, 2, 4, 8, and 16. Each shifter has two inputs—the *number* to be shifted and an *enable* signal—and is equipped with a multiplexer (MUX) at the output. This MUX ensures that the shifter outputs either the shifted value (if *enable* is active) or the original number (if *enable* is inactive).

To shift numbers by an arbitrary amount, the fixed shifters are connected in a chain. The *shift amount*, specified as a binary number, determines which shifters are activated. For instance, each bit in the binary *shift amount* corresponds to the *enable* input of a specific shifter, allowing the desired shift to be efficiently computed using a combination of the fixed shifters.

• **Shift Adapter:** In the assignment the exponents are 9 bits so their difference will also be a 9-bit number. However, only 5 shifters are available. To bridge this gap, an adapter is required to process the 9-bit difference and generate a suitable enable string for the right shifter.

This adapter performs two main tasks:

1. It ensures that the shift amount does not exceed the maximum meaningful value for a given bit-width (e.g., it does not allow a 27-bit number to be shifted more than 27 bits to the right). If the exponent difference exceeds the maximum shiftable range, the adapter sets all shifters' enable signals to high, activating all shifters simultaneously.
2. If the exponent difference is within range(e.g., 27), the adapter extracts the lower 5 bits of the difference and directly uses them as the enable string, efficiently mapping the exponent difference to the shifter controls.

5.2 Re-arranger

To add two numbers, their exponents must be equal so that their significands can be properly aligned before passing them to the adder. Achieving this requires calculating the difference between their exponents and right-shifting the smaller number accordingly. This process demands a mechanism to identify which number has the smaller exponent.

The **Re-arranger module** facilitates this by taking two primary numbers A and B and two comparator values X and Y. Based on the relationship between X and Y, it determines and outputs the smaller and larger numbers. If $X > Y$, the module designates A as the smaller number and B as the larger number.

Conversely, if $X < Y$, the roles are reversed.This is implemented by using a subtractor circuit as a comparator to evaluate X and Y. The comparison result is then passed to two multiplexers, which reassign A and B as smaller or larger based on the outcome, ensuring the correct arrangement of the numbers.

5.3 Priority Encoder

We need to use a 32-bit priority encoder for normalization. However, 8×3 or 10×4 priority encoders are the only ones available. Therefore, we first cascade two 74148 (8×3) priority encoders to create a 16×4 priority encoder before creating our 32-bit priority encoder. A second 16×4 priority encoder is made in a similar manner. Now we cascade our 16×4 priority encoders to create 32×5 priority encoder. Therefore, we can use four 74148 (8×3 Priority encoder) ICs to create a 32×5 priority encoder.

5.3.1 Cascading two 8×3 priority encoders

74148 has 8 input pins, 3 output pins, 1 enable input (EI), 1 enable output (EO) and 1 group signal (GS) pin. We have two ICs of 74148 (IC_1 and IC_2). Input 8 to input 15 is connected to IC_1 and input 0 to input 7 is connected to IC_2. EI of IC_2 is connected to the EO of IC_1. Output 3 is obtained from IC_1's GS. Output-2 is taken from O_2 (IC_1) AND $O_1(IC_2)$. In a similar manner, output-1 and output-0 are also obtained by applying the AND operation to IC_1 and IC_2's outputs. Another 16×4 priority encoder is created using two 74148 ICs (IC_3 and IC_4).

5.3.2 Cascading two 16×4 priority encoders

It's the same procedure as before. Instead of using the 74148 IC as our base component, we use the 16×4 encoder and cascade 2 of them to achieve a 32×5 encoder.

5.3.3 32-bit input high encoder

This is a wrapper around the 32 bit encoder. To enable active high input, we first invert the input. Our module has an additional pin that we refer to as 1 based indexing. The output is incremented when this pin is set high.

5.4 Normalisation

After addition the sum could become denormalized. In that case we have to normalize the result. Let's say the sum looks like this: $d_1 d_0 . d_{-1} d_{-2} ... d_{-22}$. Then the normalization process can be described in 3 cases -

$d_1 d_0 = \mathbf{01}$ The number is already normalized

$d_1 d_0 = \mathbf{1}\times$ Shift the number 1 bit to the right (hence increment the exponent)

$d_1 d_0$ **= 00** The number needs to be left shifted to bring a 1 after the radix point to the front of the number

The last case is a little different than the other cases since it requires finding the first 1 after the radix point when scanned from left to right. If the first 1 is found at position d-n than the number has to be shifted n bits to the left. Note that in this example, $d_i = 0; \forall i < n$. The bits to the right of the first 1 do not matter. So it is clear that the first 1 has more priority than the other 1s that follow. And we also want the position of the first 1 in binary so that this can be directly fed into the enables of the shifters as described in Section 5.2. This process can be done by using a priority encoder.

But the basic 32 bit priority encoder (Section 5.3) has active low input and active low output. But the bits of our significand are active high. So instead, we use the 32-bit input high encoder (Section 5.3). However, there's no need to invert the outputs. For normalization, lets say we have 1 in the $22_t h$ bit. So, the number needs to be shifted by 1 bit to the left. Consequently, our priority encoder should give us 00001. We send our $22_t h$ bit to our priority encoder's input-31. The output we get is 00000 but our required output is 00001. That can be achieved by setting the 1 based indexing pin high.

After shifting the significand to the left, the exponent must also be reduced by the shift amount. The normalizer is the main module that handles the overflow and underflow. Basically, after left shifting, if all of the bits of the exponent are 1, or there is a carry generated while incrementing the exponent, this is considered to be an overflow. As neither fall on the domain of floating point number (Note that an exponent having all bits set is reserved). And after left shifting, if a borrow is generated while subtracting from the exponent, an underflow occurs and gets reported.

5.5 Rounder

The rounder is responsible for rounding the result and storing only the limited number of bits (in this assignment, 22 bits) in the significand segment of the result.
Initially before exponent equalization, three 0 bits are added to the right of the number. But after the whole addition process, we can not store those bits as our memory is limited. But those bits help us store the closest result by approximating it. The 3 bits are called the guard bit, the round bit and the sticky bit (G, R, S).

By the *IEEE 754* standard, the approximation is achieved in the following manner-

G	R	S	Process	Comment
0	0	0	Truncate	Add 0 to the actual significand
0	0	1		
0	1	0		
0	1	1		
1	0	0	Round to even	Add 1 if the last bit of actual significand is 1
1	0	1	Round up	Add 1 to the actual significand
1	1	0		
1	1	1		

Table 2: Rounder Truth Table

From this table this is apparent that we only add 1 when the guard bit (G) is 1 **AND** either of the round bit (R) **OR** the sticky bit (S) **OR** the last bit of the significand is 1. So we add 1 when $G(R + S + d_0) = 1$. In other words, we add $G(R + S + d_0)$ to our 22 bit significand.

The number can become denormalized after rounding up or rounding to even. In that case we have to shift the number 1 bit to the right and hence increment the exponent. Because the exponent is being incremented, there can be scenarios where the exponent is too large, thus an overflow. So the rounder also reports such overflows.

5.6 Flags

We have implemented 6 flags to detect problematic outputs. They are given below:

- De-norm flag is used to detect when the exponent bits are zero but fraction is non-zero.
- NaN flag is used to detect when the exponent bits are all ones but fraction is non-zero.
- INF flag is used to detect when the exponent bits are all ones but fraction bits are zero.
- Zero flag is used to detect when the exponent bits are zero and fraction bits are zero.
- Overflow happens when the exponent becomes larger than the maximum representable exponent.
- Underflow happens when the unbiased exponent becomes less than the smallest representable exponent.

6 Complete Circuit Diagram

6.1 Re-arranger

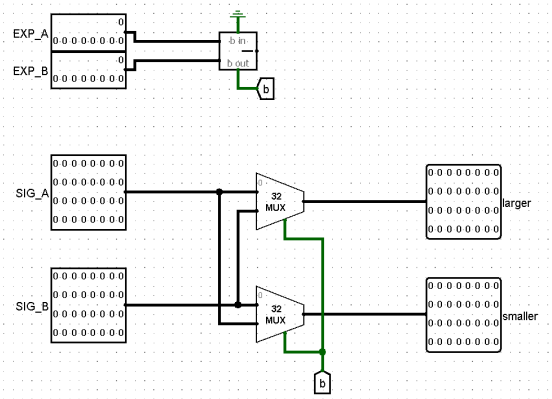
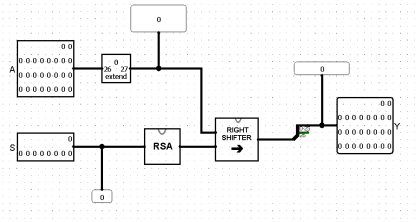
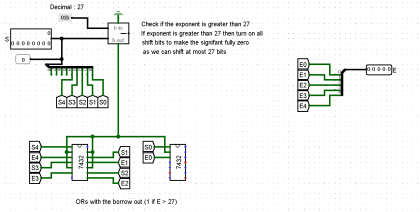


Figure 3: Re-arranger Circuit

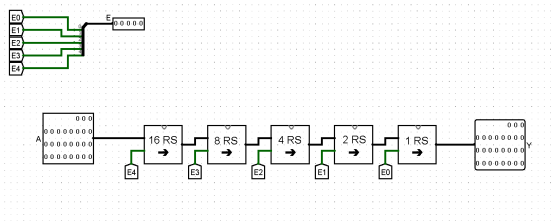
6.2 Right Shifters



(a) Right Shift Wrapper



(b) Right Shift Adapter



(c) Right Shifter

Figure 4: Shifters Circuit

6.3 Adder Subtracter Unit (ASU)

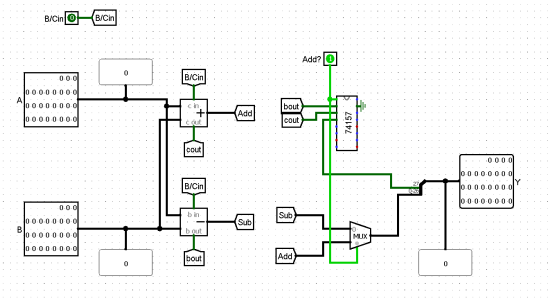


Figure 5: Adder Subtracter Circuit

6.4 Absolute Calculation Unit

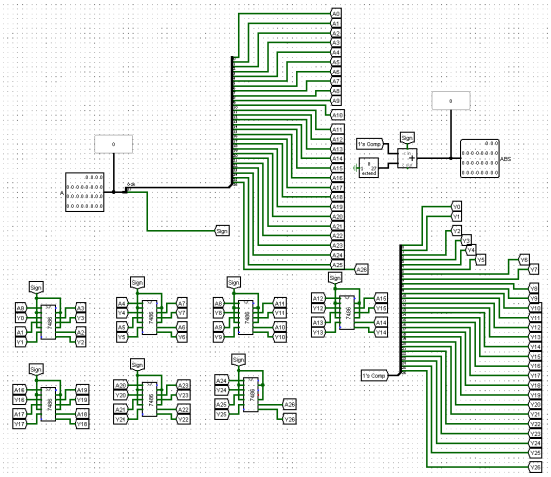
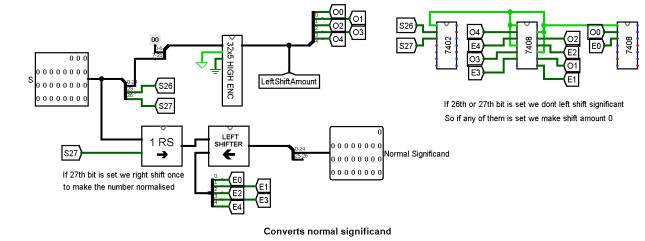
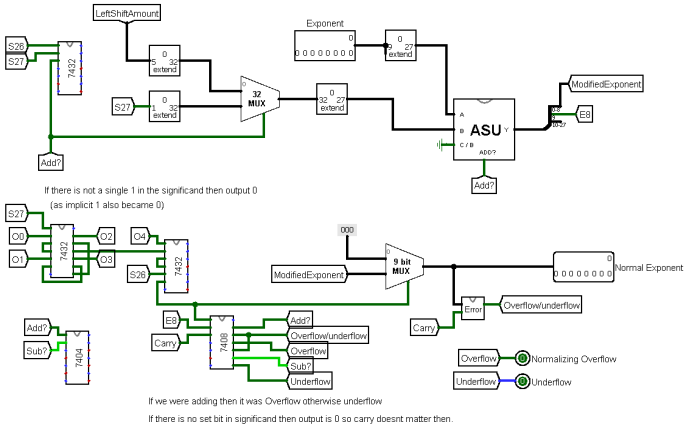


Figure 6: ABS Circuit

6.5 Normaliser



(a) Significand Normaliser



(b) Exponent Normaliser

Figure 7: Normaliser Circuit

6.6 Rounder

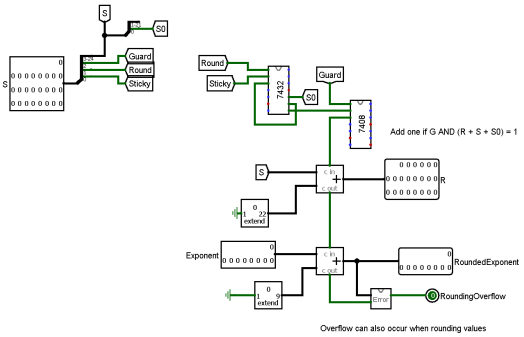


Figure 8: Rounder Circuit

6.7 Flags Checker

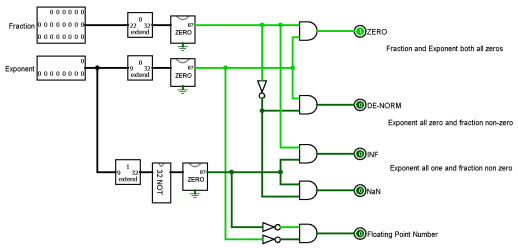


Figure 9: Flags Checker Circuit

7 ICs Used with Count as a Chart

IC	Quantity
IC 74157	112
IC 7432	14
IC 7486	7
IC 7404	7
IC 74148	4
IC 7408	7
IC 7402	1
Adder	5
Subtracter	5
Total	160

Table 3: ICs Used with Quantity

8 Simulator Software with Version

Logisim 2.7.1 has been used for this assignment.

9 Discussion

We tried our best to reduce the number of ICs and also aimed to create a simplified design overall to facilitate easier real-life implementation. The design was divided into smaller submodules for clarity and scalability. The submodules include:

- **Normalization and Rounding Module**
- **Shifter Module**
- **Priority Encoder Module**
- **Re-arranger Module**

In the Shifter Module we implement the circuit for the right shift and left shift that can shift the number by 0 to 27 bits.In the Re-arranger Module we implement the circuit that can generate the larger number and the smaller number in basis of two exponents of the two numbers by using subtraction operation.In the priority encoder module we have implement the priority encoder circuit that does the left shift operation for the normalizatioin when there was 0 before the fraction part.Like `00.001001` then we have to first see where the first one after the fraction part and then we have to left shift the 1 by 3 amount and this 3 was generated by the priority encoder.In the Normalization part we use our own custom priority encoder to normalize a number that have 0 before the fraction part the the priority encoder decides how much amount of shift we needed to normalize the number and decreases the exponent by the same amount of shift.In the case of 10 before the fraction part we just right shifted the number and increses the exponent by 1 bit.In the rounder module We use this equation:

$$F = G(R + S + S_0)$$

Where S_0 is the Lsb.

The assignment focused solely on handling the floating-point domain, but we extended our implementation to also address the case when the number is zero. According to the *IEEE 754* standard, the exponent in a floating-point number cannot be entirely 0s or entirely 1s, as these scenarios are reserved for specific representations. For example, when both the significand and exponent are 0, the number represents 0.0. Although handling such reserved cases was not explicitly required by the assignment, we chose to account for the zero case. Without this consideration, the adder would fail in scenarios like $a - a = 0$ or $a + 0 = 0 + a = a$. By implementing this special case, we ensured the correct behavior of the adder when zero appears as an input or output.

Our adder implementation is not without limitations. Currently, the rounding buffer uses only a single sticky bit. By incorporating additional sticky bits, the

buffer could hold more information, thereby enhancing the precision of the result. However, through testing, we determined that the precision level of our implementation is sufficient for the majority of input cases.

10 Contribution of Each Member

2105045 - Sadia Binte Sayad

- Designed Rounder Circuit
- Designed Abs Circuit

2105058- Elias Mainur

- Designed Right Shifter Circuit
- Designed Re-arranger Circuit

2105060 -Jonayed Mohiuddin

- Designed Normalizer Circuit
- Designed Priority Encoder and Assembled Components