1 Introduction

A floating point number is a number that has a decimal point and thus can store a fractional value. These numbers are stored differently than integers because to store a floating point number there are mainly 2 challenges. First of all, there is a challenge of storing the integer part and also storing the fractional part, hence more memory. Secondly, there are infinitely many floating point numbers even in a finite range, for example $[0, 1]$. So the floating point representation has to be clever enough to allow floating point numbers in a large range, as well as support good enough precision.

The floating point number representation uses the scientific notation of numbers. It has 3 parts - sign, significand and exponent. There are many different standards that decide how many bits each of these segments should store.
The *IEEE 754* standard specifies that there should be 1 sign bit, 8 exponent bits and 23 significand bits.
The exponent is not directly stored, rather it is *biased* first by adding a bias to the actual exponent and than store that in memory. The significance of this process is to make the exponent unsigned, thus allowing easier circuitry to deal with the comparison of exponents of 2 numbers. The bias is chosen to be $2^{n-1} - 1$ where n is the number of bits in the exponent field.
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.

So the overall representation looks like:

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

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

This representation has the power of encapsulating large range due to the use of exponents. On the other hand it can store numbers in great precision as effective exponents can be negative, and thus indicate really small numbers. But it should be kept in mind that there can be scenarios, specially for larger exponents, where the exact floating point number can not be stored. In those cases the memory stores the nearest floating point number.

2 Problem Specification

The assignment asked to create a floating point adder circuit that takes 2 floating point numbers and calculates their sum, and also notifies if there was any overflow or underflow. The representation was defined in the following manner-

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

Table 1: Problem specification

3 Flowchart

4 High Level Block Diagram

5 Description of Modules

Each high level module and how they work is listed below

5.1 Multiplexers

There are several multiplexer circuit each working with different input sizes. The multiplexers that were required for this assignment are - 5×1, 8×1, 28×1, 32×1. They were created by chaining several 74157 MUX ICs.

5.2 Shifters

In order to shift the numbers we needed different kind of shifters for both direction. In normalization and also in exponent-equalization, the numbers needed to be shifted by arbitrary amount. So shifters that can support arbitrary amount shifting were required.

Instead of making all possible shifters (1 bit shifter, 2 bit shifter, 3 bit shifter and so on..), the design was cleverly crafted so that there are lesser number of ICs used. Only 5 different shifters were created (1, 2, 4, 8 and 16). Each of them having 2 inputs - *the number* and *enable*. They are designed so that when the *enable* is set to 1, the number is shifted by the amount specified by the module, otherwise the number is not changed and passed to the output. This *enable* feature was achieved by introducing a MUX in the output so that when *enable* is 1, the shifted number is sent to the output, otherwise the original number is sent to the output.

Now that the fixed amount of shifters (that can only shift by an amount that is a whole power of 2) are ready, we can finally create arbitrary shifters. The arbitrary right shifter (same logic for left shifter) takes 2 inputs - *the number to shift* and *the shift amount*. Since the shift amount is specified by a binary number, we can leverage the fact that in a binary number, each bit corresponds to the availability of a certain power of 2. So connecting each bit to the corresponding shifter's enable and chaining such shifters will suffice to shift by any arbitrary amount.

5.2.1 Shift Adapter

Since the exponents are 11 bits each, their difference will also be 11 bits. But we only have 5 shifters as described in Section 5.2. So we need some sort of an adapter that take the 11 bit difference and generate a suitable enable string for the right shifter. This adapter basically compares the highest possible enable value (for example it does not make sense to shift a 32 bit number more than 32 bit to the right) and the exponent difference. If the exponent difference is larger than expected than it sets all the shifters' enable to high, thus enabling

all shifters. Otherwise it reads the last 5 bits of the exponent difference and sends them as the enable string.

5.3 Re-arranger

The exponents of the numbers need to be equal in order for the significands to be passed to the adder. To do that, we first need to find the difference of their exponent and shift the smaller exponent number to the right. So we clearly need a distinction based on the exponent. The *Re-arranger* module takes 2 numbers as the base number(A, B) and 2 other decider numbers(P, Q) and re-arrange the number based on the decider numbers. It outputs 2 numbers - *smaller* and *larger*. If $P > Q$ then smaller $= A$ and larger $= B$. The alternative case happens when $P < Q$.

This is achieved by using a subtractor circuit as a comparator and comparing the *deciders*. Then the comparison result is fed into 2 multiplexers that finally re-arrange the *base* numbers and output *smaller* and *larger*

5.4 Priority Encoder

We need to use a 32-bit priority encoder for normalization (explained further in Section - 5.5). 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.

5.4.1 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).

5.4.2 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 (described in Section 5.4.1) and cascade 2 of them to achieve a 32×5 encoder.

5.4.3 32-bit input high encoder

This is a wrapper around the 32 bit encoder described in Section 5.4.2. 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.

5.5 Normalization

After addition the sum could become denormalized. In that case we have to normalize the result. Let's say the sum looks like this: $d_1 d_0 . d_{-1} d_{-2} ... d_{-20}$. Then the normalization process can be described in 3 cases -

$d_1 d_0 = 01$ The number is already normalized

$d_1 d_0 = 1\times$ 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 5.4.2) 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 5.4.3). However, there's no need to invert the outputs. For normalization, lets say we have 1 in the 20^{th} 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 20^{th} 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.

5.6 Rounder

The rounder is responsible for rounding the result and storing only the limited number of bits (in this assignment, 20 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 20 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.

6 Complete Circuit Diagram

7 ICs Used with Count as a Chart

IC	Quantity
ALU	9
IC 74157	110
IC 7432	14
IC 7486	8
IC 7404	8
IC 74148	4
IC 7408	6
IC 7402	1
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

All of us have put a great deal of effort in order to complete this assignment. We tried our best to reduce the number of ICs and also tried to make a simplified design overall so that the real life implementation becomes easier. The use of ALU was preferred over creating own adders or comparators as that has the potential of saving a lot of ICs.

The assignment asked to only handle the floating point domain, but we also handled the case when the number is zero. According to the *IEEE 754* standard, the floating point number's exponent can not be all 0s nor all 1s. These 2 cases are reserved. They indicate different meanings. For instance, both the significand and exponent being 0 indicate 0.0. Although the assignment did not ask to handle the reserved cases, we decided to tackle the zero's case as well. For example with the assignment requirement implemented without handling these special cases, the adder will fail to provide $a - a = 0$ or $a + 0 = 0 + a = a$. But we took great care to implement this special case of the input or output being 0.

The adder of our implementation is not perfect. Since the rounding buffer contains only 1 sticky bit. By adding more sticky bits it is possible to have a greater buffer of result, and thus better precision. Having said that, we have found the precision level of our implementation is good enough for most of the

input cases.

We have thoroughly checked many input cases and to test it we have created a *ANSI C* program (`https://github.com/Irtiaz/Floating-Point-Adder-Simulator`) to verify our results. But the program was hard to work with since it ran on the terminal. So we built a *Typescript, React* web app (`https://floating-point-adder.netlify.app/`) that helped us quickly test our adder against many different test cases.

Overall the entire design process along with the implementation proved to be a very challenging but interesting task. We have learned a lot from this assignment.

10 Contribution of Each Member

2005063 - Abid Hasan Khondaker

- Developed the normalizer circuit
- Developed shifter circuits

2005070 - Md Irtiaz Kabir

- Designed the whole adder and assembled components
- Developed the tools for testing results

2005077- Sadatul Islam Sadi

- Developed the priority encoder
- Developed *ABSer*

2005080 - Mohammad Ninad Mahmud

- Developed rounder
- Developed overflow and underflow detector and relevant circuits

2005083 - Jarin Tasnim

- Developed shifters and MUX circuits
- Tested the adder thoroughly