CSE 306

Computer Architecture Sessional

Assignment-2: 32-bit Floating Point Adder Simulation

Section: A1

Group: 4

Group Members:

ID	Name
2005005	Anup Halder Joy
2005021	Afzal Hossan
2005022	Ekramul Haque Amin
2005024	Asif Karim
2005028	Md. Tamim Iqbal

Introduction:

A computer representation of a real number with a set number of digits before and after the decimal point is called a floating-point number. The digital circuit that adds and subtracts floating point numbers is called a floating-point adder. It can be applied to numerical values that are either too big or too small to be precisely represented by integer representations.

Three fields make up a number's floating-point representation: the sign bit, the exponent field, and the significand field. Positive or negative signs are represented by the sign bit. 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 number 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 significand, is made up of the digits that come after the decimal point. The sign bit, exponent field, and significand in this implementation use 1, 11, and 20 bits, respectively. Therefore, $2^{11-1}-1 = 1023$ would be the exponent bias for our case.

In order to add two numbers, a floating-point adder first aligns their decimal points before adding their significands. After the necessary shifts and related modifications (increment/decrement), the result's exponent is fixed, and normalization and rounding are performed. 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 computations can be completed far more quickly by a specialized floating-point adder than by the main CPU. Applications requiring high-precision floating point calculations, such data analysis and scientific simulations, may find this to be especially crucial.

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	11	20

Flowchart of the Addition/Subtraction Algorithm:

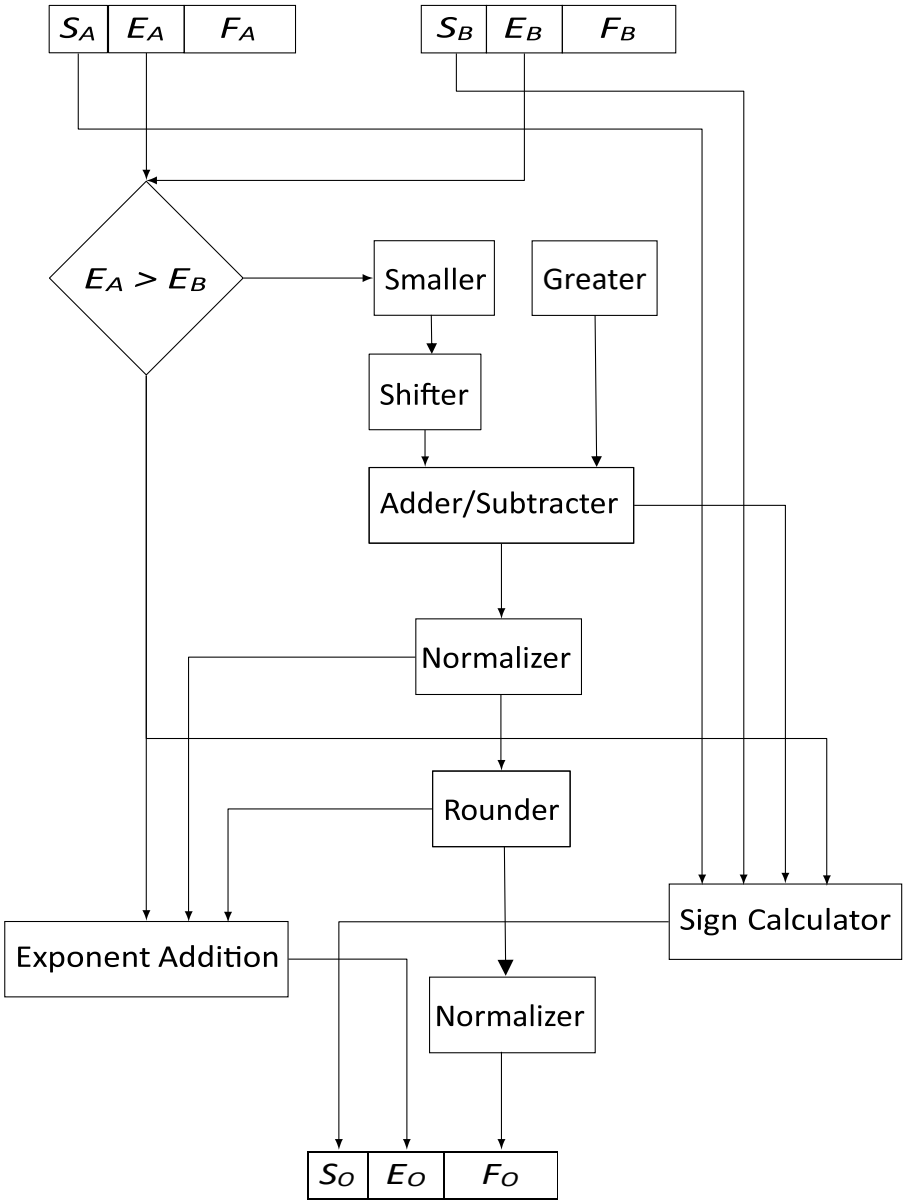

Fig: Flow chart of the addition/subtraction algorithm

High-level Block Diagram of the Architecture:

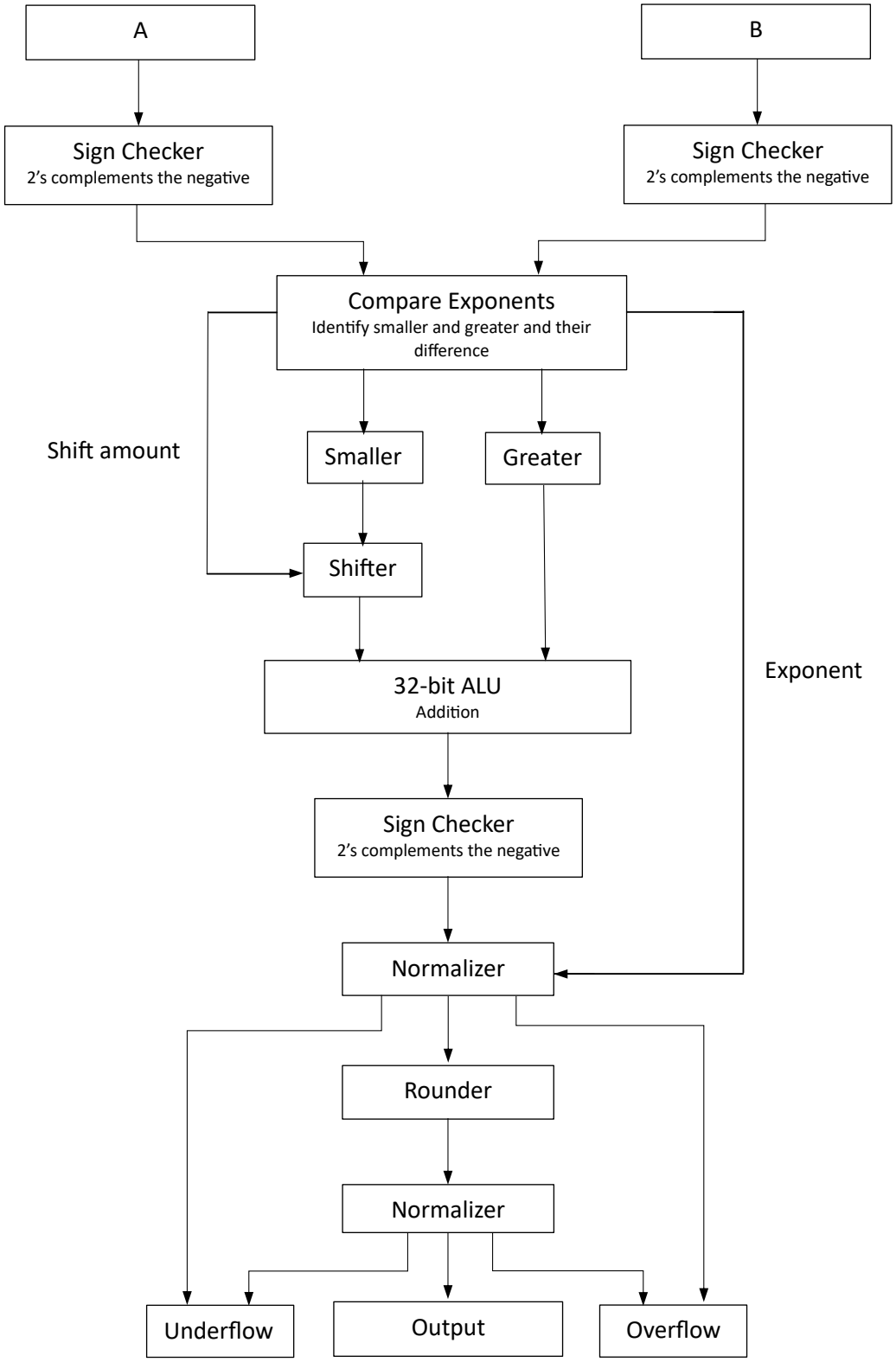

Fig: High-level Block Diagram of the Architecture

Floating Point Adder

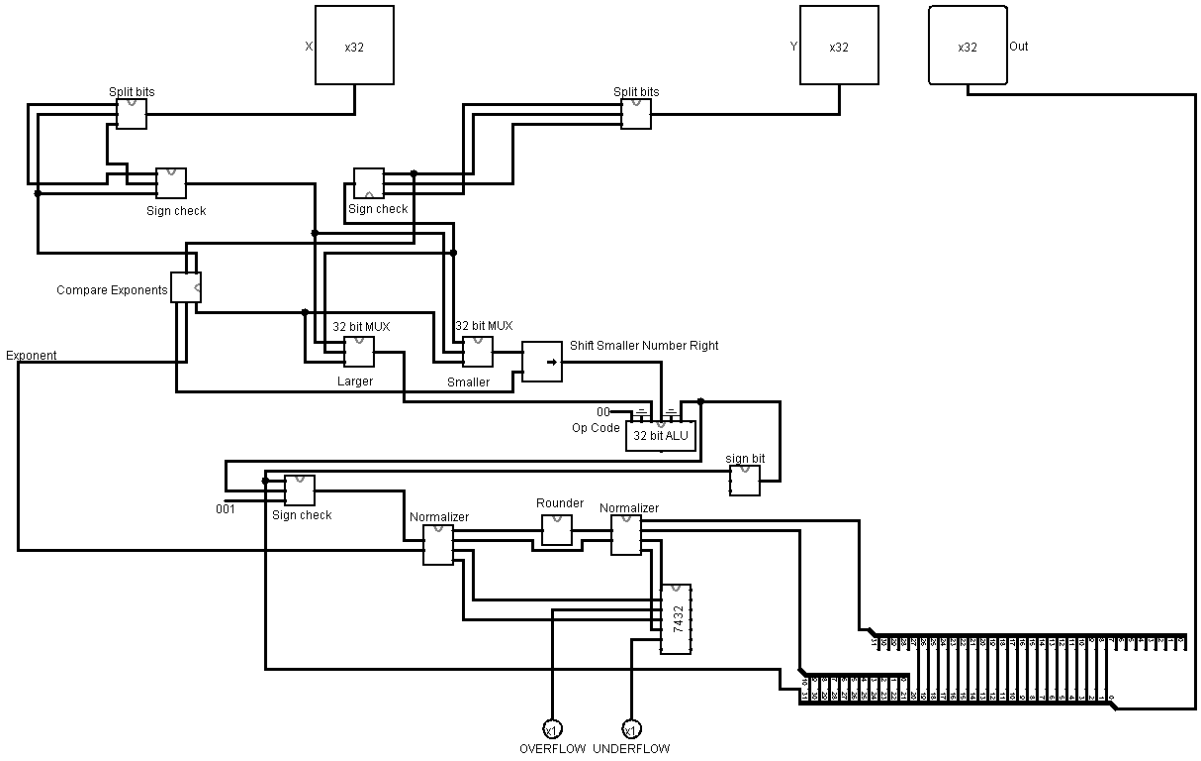

Fig: Floating Point Adder

Building Blocks:

- Check Sign
- Comparator
- Normalizer
- Rounder
- Bit Splitter

Detailed Circuit Diagram of the Important Blocks:

Check Sign

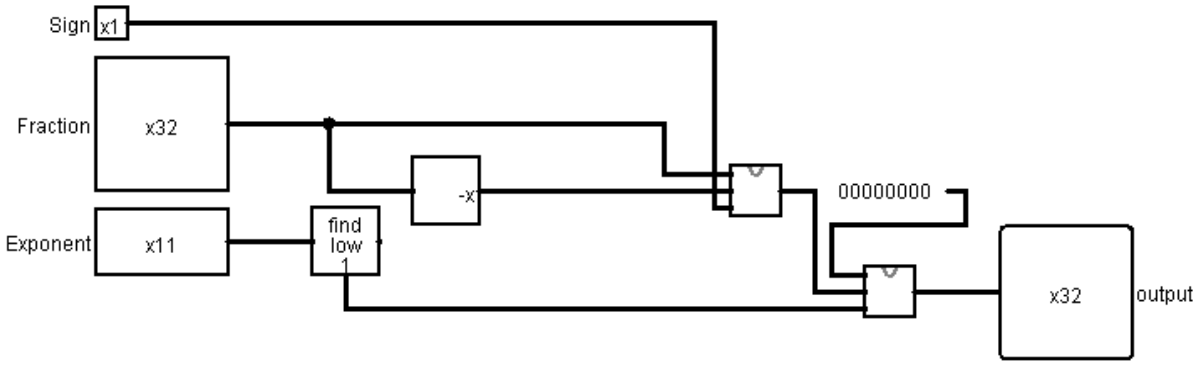

Fig: Check Sign

To add two numbers, we need to figure out the signs (positive or negative) of the numbers. If any of the number is negative i.e. sign bit is 1 then that number is in 2's complement form. So, we need to take 2's complement of that number and add it with other one. Otherwise, we can take the original number and add it. After taking an input, we are splitting it in 32 bits and separating the fraction (0^{th}-19^{th} bit), the exponent (20^{th}-30^{th} bit) and the sign bit (31^{th} bit). Then the original fraction part is put in a 32-bit MUX's 0^{th} input. Using a 32-bit negator, we have negated fraction and put the negative number in 1^{st} input of that MUX. The selector bit is the sign bit. If sign bit is 0 i.e. the number is actually positive, original number would be selected. If the sign bit is 1 then the number is actually negative and we are selecting its negated number (that is actually positive). The output of this MUX is then put into another 32-bit MUX's 1^{st} input and 0 has been put into the 0^{th} input of that MUX. The selector bit of this MUX is coming from a bit finder which actually evaluates if there is any 1 bit in the exponent. If there is 1 in exponent, the number would be selected. Otherwise, exponent is 0 which is only allowed in case of the fraction is also 0. That's why 0 has been selected from the MUX and outputted to be used in further operation.

Comparator

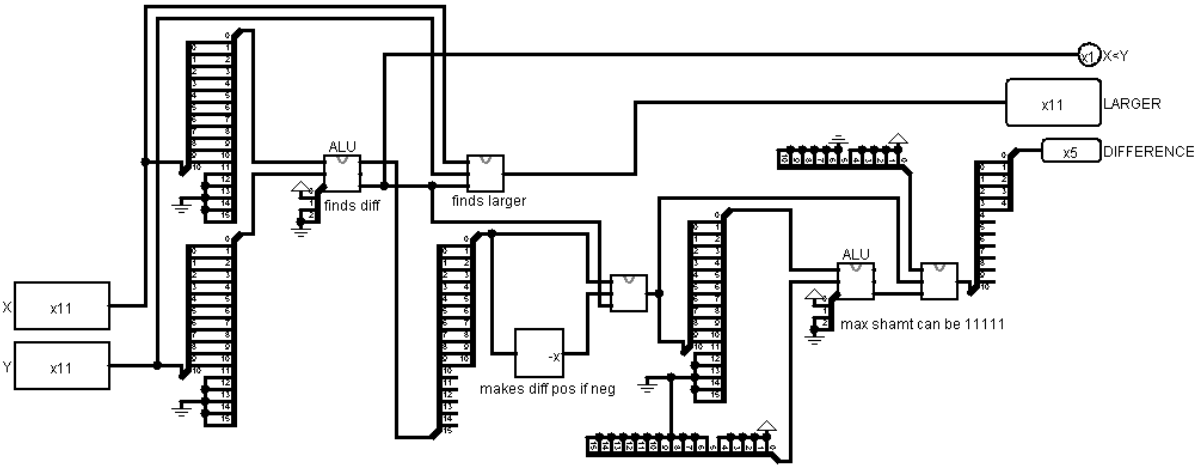

Fig: Comparator

The purpose of the comparator circuit is to compare the two exponents. It gives three output one is to show the larger exponent using a MUX and another indicator indicates whether exponent X is less Y or not and it comes from the 16-bit ALU sign flag. And if the exponents are different then we get the amount of shift from the third output which is the difference amount. The underline circuit of the comparator is the 16-bit ALU and MUX. In the floating-point representation exponent consists of 11 bits. So, we have to extend it to 16-bit. If we use the 001 as control bit of ALU then it works as an arithmetic subtractor. If the subtracted amount is neg then we have to make it positive and we choose this value using MUX. This subtracted amount represents the shifting amount of smaller exponent. Since we output and inputs are 32 bit the maximum amount of shift, we can handle is 31bit shifting. So, we compare whether the subtracted amount is less than 31 or not using a second 16bit ALU. If yes then we choose the subtracted amount otherwise we choose 31-bit shift. It is done by another MUX. And the required amount of shift is shown as DIFFERENCE.

Normalizer

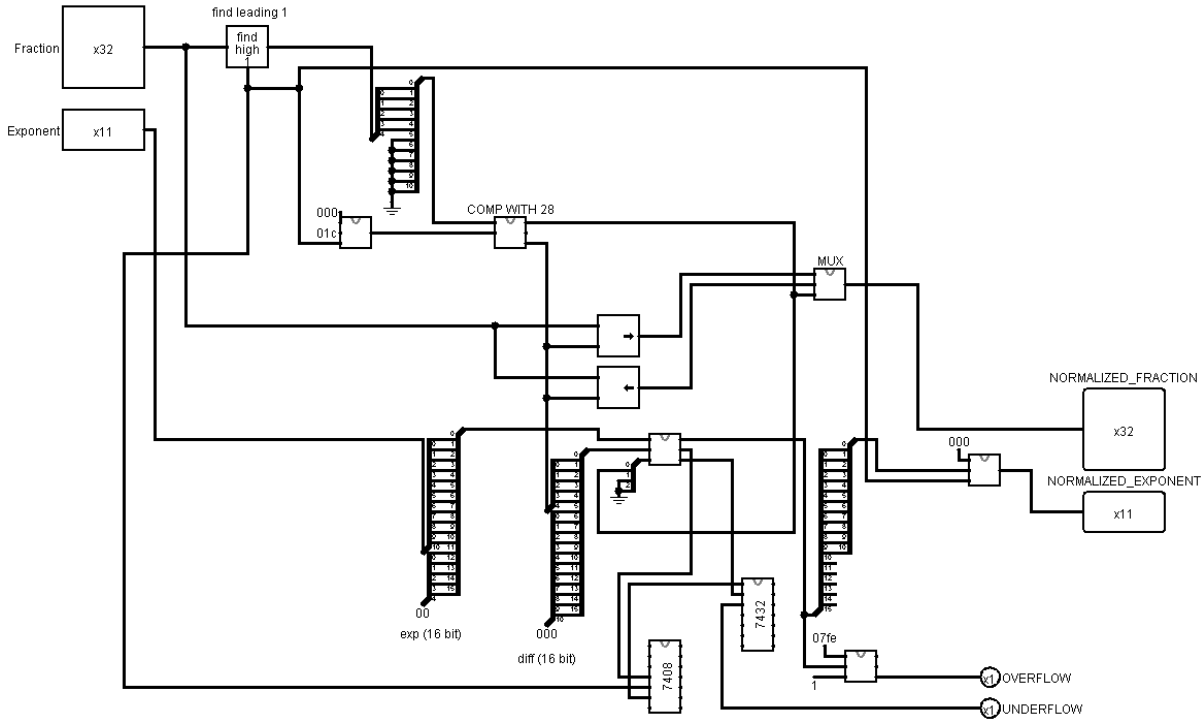

Fig: Normalizer

In order to normalize followed by addition operation, we employed a bit finder to identify the position of the most significant set bit, which is the leftmost one. The result can be up to 31(0th to 31th), hence consisting of 5 bits (0 to 25 − 1). If the fraction part contains set bit, then we compare the position of leftmost set bit with 28 to determine the shift amount as we set the left most bit at 28th position. We use “Comparator” to determine the difference. Then shift the fraction part accordingly. Firstly, we shift the fraction both left and right. Then using a MUX, we select which one to be taken according to the comparison. At the same time, we adjust the exponent part (converted to 16 bits) using a ALU according to the comparison. If the exponent becomes greater than the maximum valid exponent for 11 bits (0x7e) then the overflow flag is set. On the other hand, if the exponent becomes zero or negative then the underflow flag is set.

Rounder

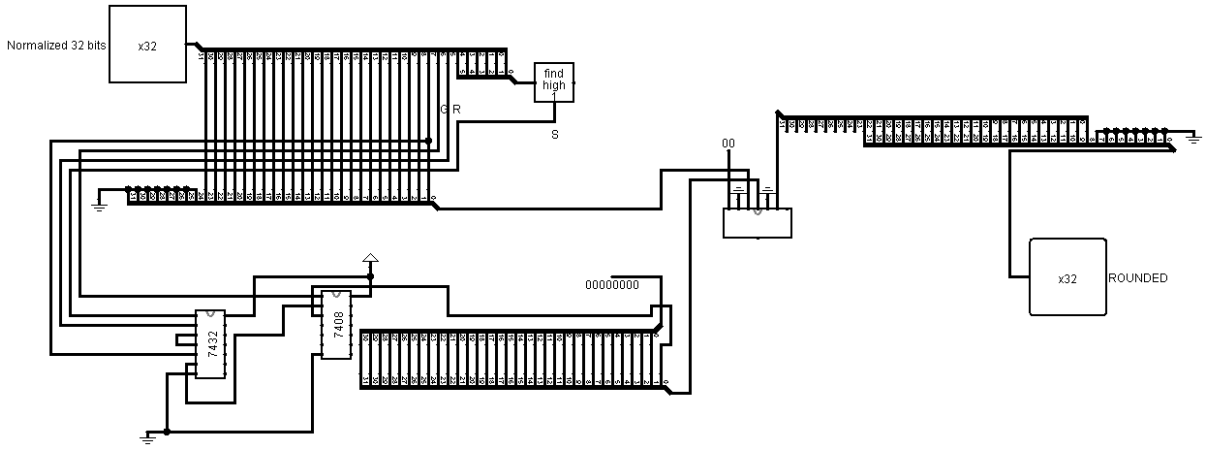

Fig: Rounder

We have a 20-bit mantissa. but we have done the operation on a 32-bit adder. We have to take only 20 bits. So, we have to round the result by some instance. In order to do so we consider the 7th bit as Guard bit, 6th bit as round bit and if any of the bit right to the round bit is set Sticky bit will be set. And the LSB of the 20-bit mantissa (8th bit) is M. Then we have to do the following operation.

M	G	R	S	Action
X	0	X	X	Truncate
0	1	0	0	Truncate
1	1	X	X	Round up
X	1	1	X	Round up
X	1	X	1	Round up

K-map for rounding up is give below:

RS / MG	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	1	1	1	1
10	0	0	0	0

Roundup=GM+GR+GS

=G(M+R+S)

If Roundup is 1, we add 1 to the 8th bit of the result. Otherwise, we just truncate.

Bit Splitter:

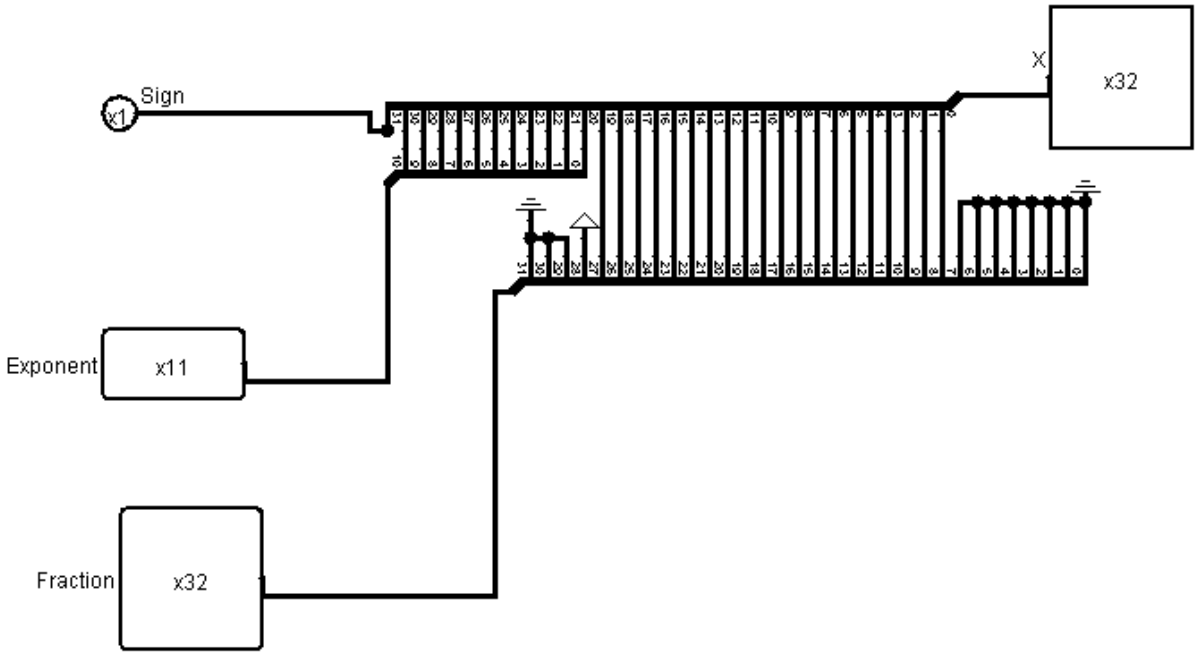

Fig: Bit Splitter

We know that there are three portions in a floating-point number: sign, exponent and fraction. Bit splitter splits these portions so that they can be used in action.

ICs Used with Count as a Chart:

IC	Quantity
32-bit ALU	2
16-bit ALU	10
Bit Finder	6
Negator	6
Shifter	5
IC 74157	119
IC 7432	4
IC 7408	3

Simulator used Along with the Version Number:

Logisim 2.7 has been used for simulating the floating-point adder circuit.

Discussion:

In this assignment our task of designing and simulating a Floating-Point Adder (FPA) using Logisim 2.7.1 presented both challenges and learning opportunities throughout the process.

We used smaller circuit to design a full FPA circuit. We used 32-bit ALU downloaded from an online source. Incorporating a pre-designed 32-bit ALU from an online source brought its own set of challenges. Ensuring seamless compatibility, adapting interfaces, and maintaining overall system coherence required thorough testing and debugging.

Again, working with Logisim 2.7.1 presented challenges due to version-specific features and potential bugs. Ensuring compatibility and mitigating any unexpected software-related issues required continuous process like repeatedly closing a file and opening again.

In this assignment, the specified input format, comprising 32 bits with 1 sign bit, 11 bits for the exponent, and 20 bits for the fraction, presents a standard structure for floating-point numbers. One notable decision in the design process was to extend the precision of the fraction component. While the initial input allocated 20 bits for the fraction, the decision was made to convert this fraction to a 32-bit representation. By converting the 20-bit fraction to 32 bits, the Floating-Point Adder could perform arithmetic operations with increased precision. It helped us in rounding the output.

Another challenge was to check the circuit input/output and check if it's working fine or not. We used different programs written in C or Python to compute output for sample input.

This project has been an enriching journey through the intricacies of floating-point arithmetic, digital circuit design, and the strategic manipulation of number representations. The challenges faced, be it in designing a 16-bit ALU, integrating an external 32-bit ALU, configuring an 11-bit MUX, or managing the extended and rounded fraction, have deepened our understanding of the complexities inherent in digital system design.

In conclusion, this endeavor not only honed our technical skills but also fostered a greater appreciation for the delicate balance required when navigating precision and computational efficiency in the realm of floating-point arithmetic. The insights gained from this project will undoubtedly contribute to our future endeavors in digital system design and computational modeling.