Course ID: CSE 306

Course Title: Computer Architecture Sessional

Assignment 2: 32 bits Floating Point Adder Simulation

Section: A2

Group: 03

Group Members:

1) 1905034
2) 1905036
3) 1905038
4) 1905054
5) 1905059

Introduction:

Floating-Point Adder is a combinational circuit which takes two floating points as inputs and provides their sum which is another floating point as output. Its implementation requires some basic n bits adder, subtractor, shifter, multiplexer comparator and some other basic gates.

Floating-Point Adder is designed to perform "Floating Point arithmetic" which is by far the most used way of approximating real number arithmetic for performing numerical calculations on modern computers.

The floating-point numbers representation is based on the scientific notation: the decimal point is not set in a fixed position in the bit sequence, but its position is indicated as a base power.

All the floating-point numbers are composed by four components:

1] Sign: It indicates the sign of the number (0 positive and 1 negative)

2] Significant: It sets the value of the number

3] Exponent: It contains the value of the base power

4] Base: The base (or radix) is implied and it is common to all the numbers (2 for binary numbers)

The steps involved in the design of a Floating-Point Adder are as follows:

1] Extracting signs, exponents and fractions of both A and B numbers.

2] Treating the special cases:

- Operations with A or B equal to Zero
- Operations with $\pm\infty$
- Operation with NaN

For simplicity of our design, we are only dealing with the first case.

3] Finding out what type of numbers are given:

- Normalized
- Unnormalized

For simplicity of our design, we are assuming that numbers are given in normalized form

4] By comparing the exponent of the numbers, finding out their difference which is actually the required shifting amount and also the smaller & larger number.

5] Shifting the lower exponent number fractio to the right bits. Setting the output exponent as the highest exponent.

6] Working with the operation symbol and both signs to calculate the output sign and determine the operation to do.

7] Addition of the numbers and detection of overflow (carry bit)

8] Standardizing fraction shifting it to the left up the first one will be at the first position and updating the value of the exponent according with the carry bit and the shifting over the fraction.

Problem Specification:

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 bit long with following representation:

Sign	Exponent	Fraction
1 Bit	12 Bits	19 Bits

Flowchart:

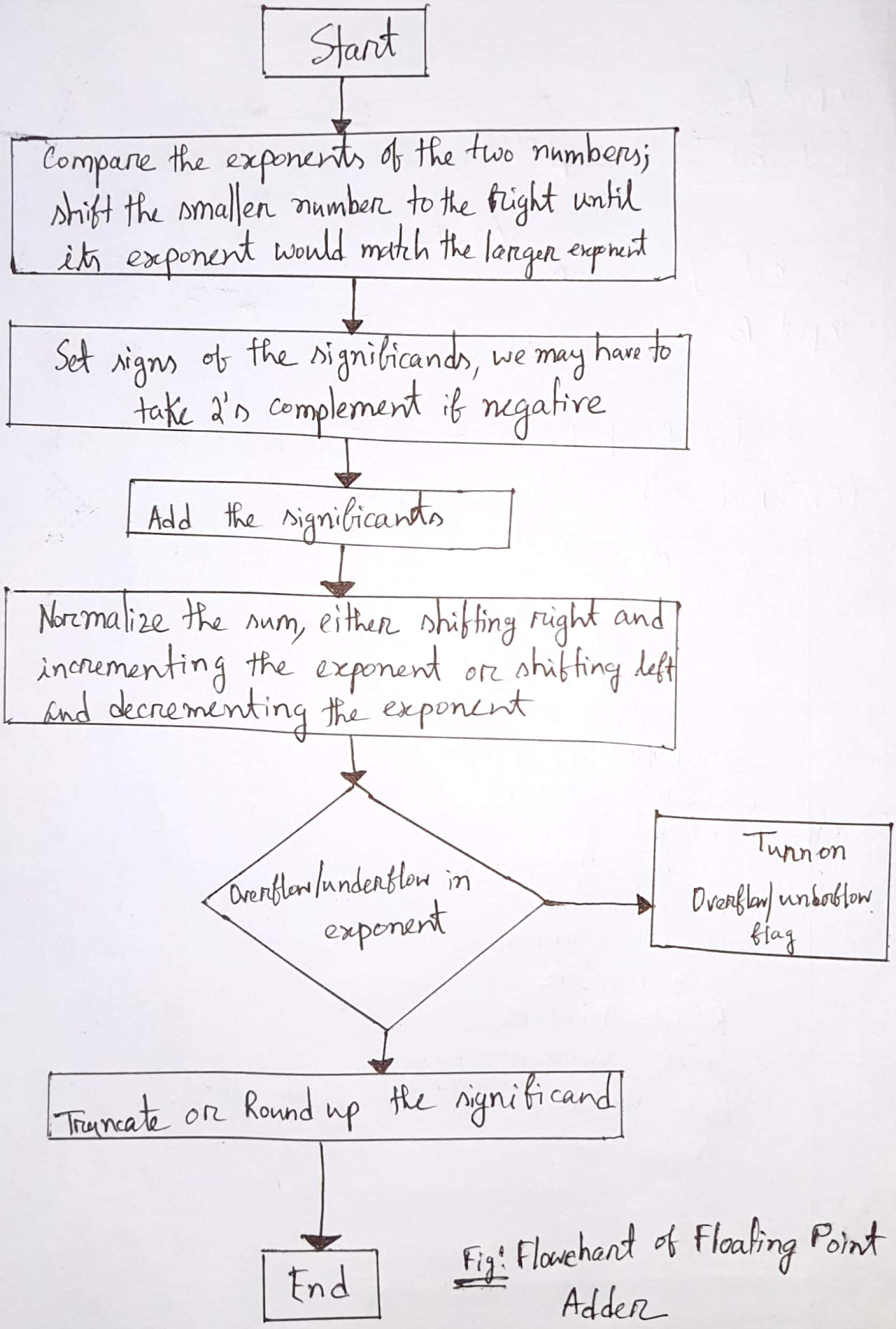

Fig: Flowchart of Floating Point Adder

Block Diagram:

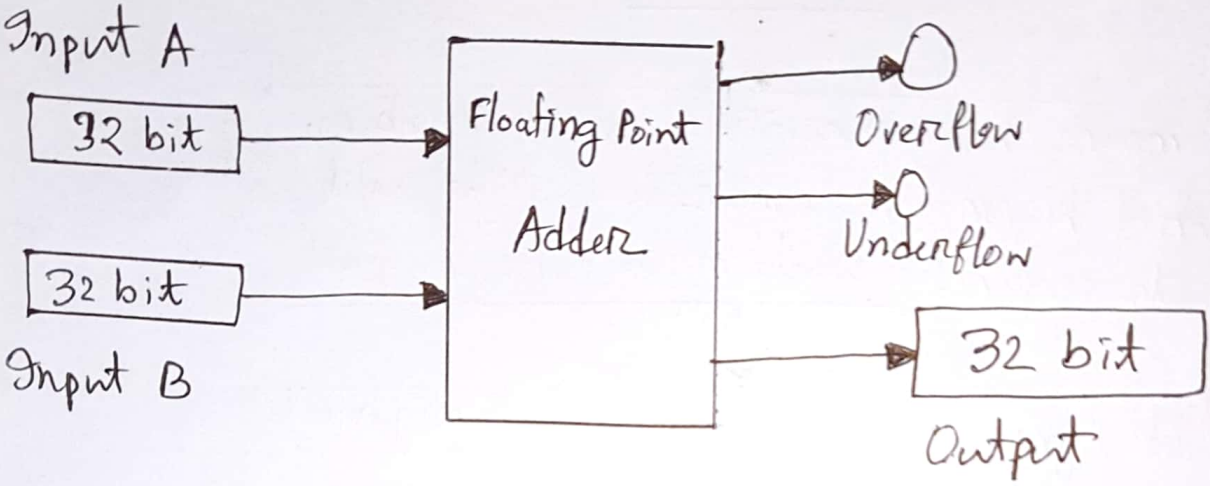

Fig: Block Diagram of Floating Point Adder

ICs used with count as a chart:

Component	Component Count
IC 7408	4
IC 7432	3
IC 7404	1
IC 7486	1
2X1 MUX	11
12 bit Comparator	4
19 bit Comparator	1
21 bit Adder	1
12 bit Adder	1
31 bit Adder	1
Subtractor	4
Right Shifter	3
Left Shifter	1
Bit Finder	2
Negator	1

Simulator:

Logisim Version 2.7.1

Discussion:

During Implementation of our circuit, we had to discard some design because those designs required a lot of ICs. We had to change our design several times. We directly uses modules provided ~~by~~ in LogiSim for our design to easily implement. We spent a lot of time in cross-checking corner cases, overflow, underflow conditions for each of sample test case. Sometimes, we encountered precision loss of minimum one or maximum three bits in output, because of rounding up and truncating the sum of two floating numbers. When two same floating points numbers but different sign bit are added, the sum did not equal to zero in our design's output. So, we need to ~~use~~ do extra checking and use some multiplexers and ICs to determine where those inputs are same with different sign bit. If it is true, the output would zero. Considering all those aspects, we finally implemented the most optimized design we could find.

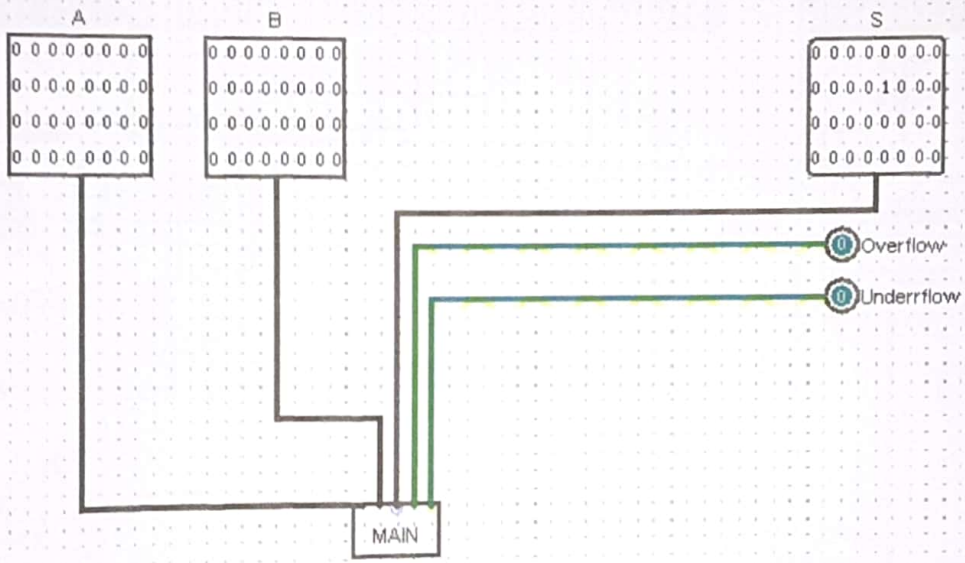

Fig: I/O

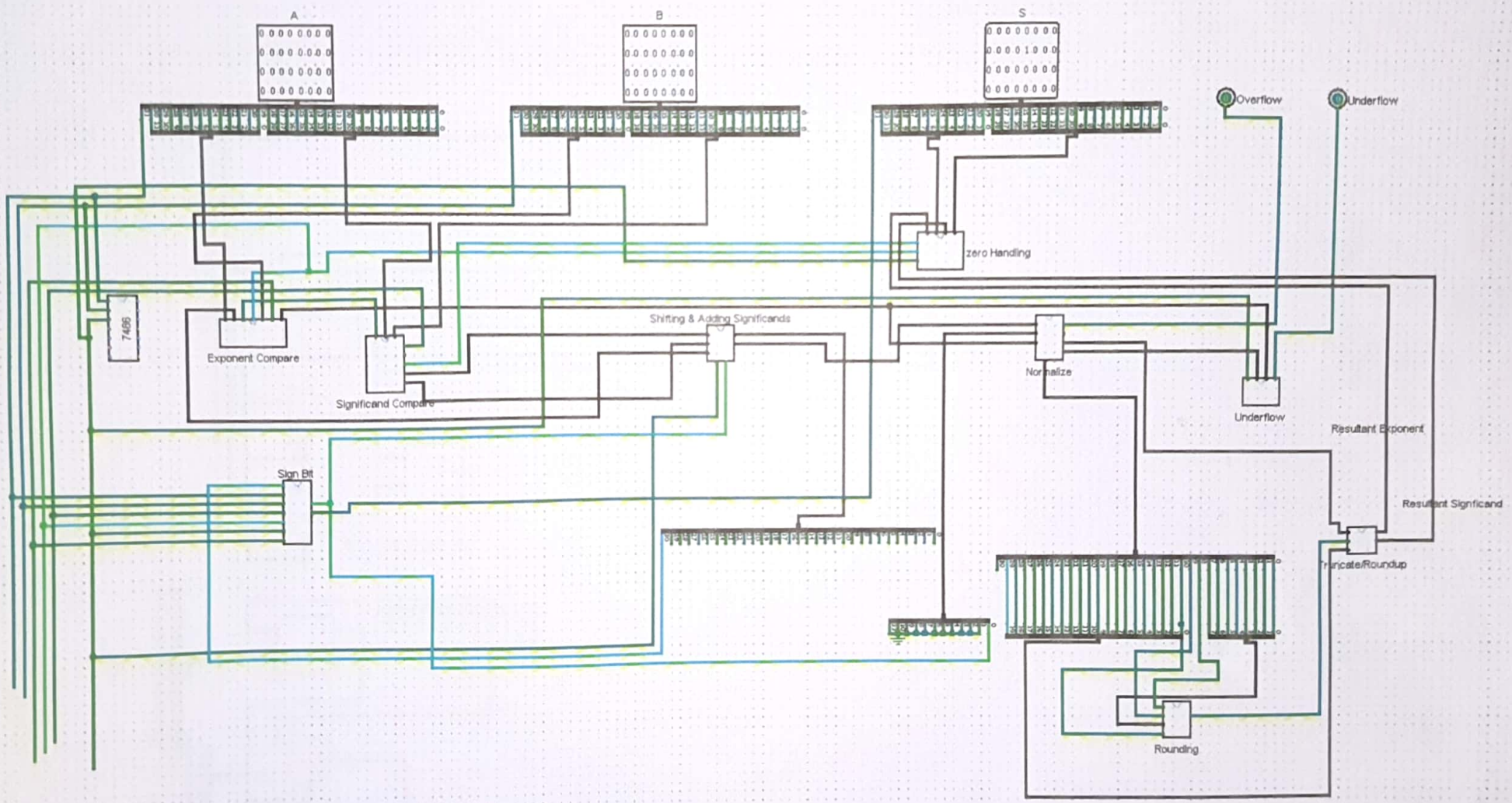

Fig: Main

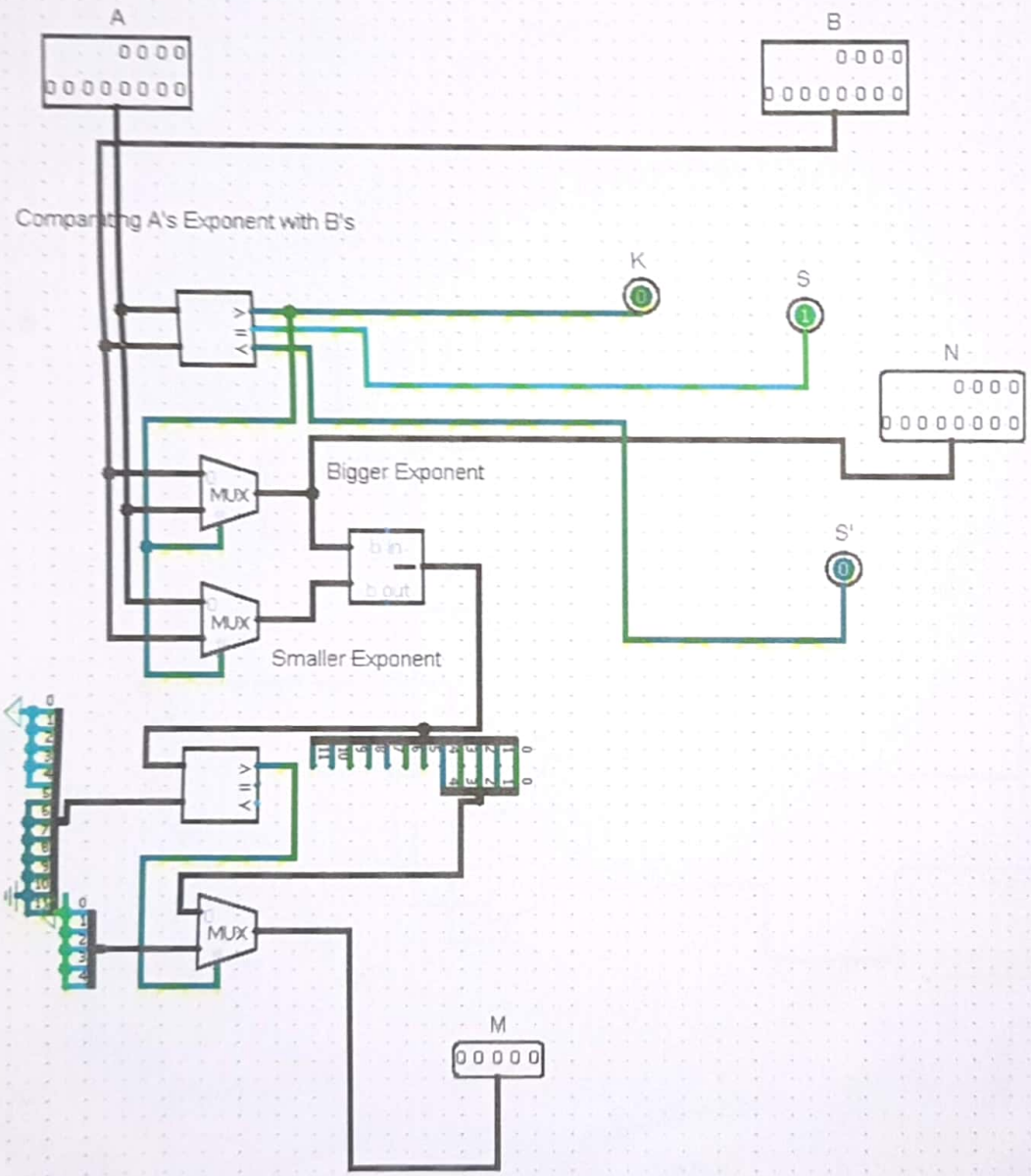

Fig: Exponent Compare

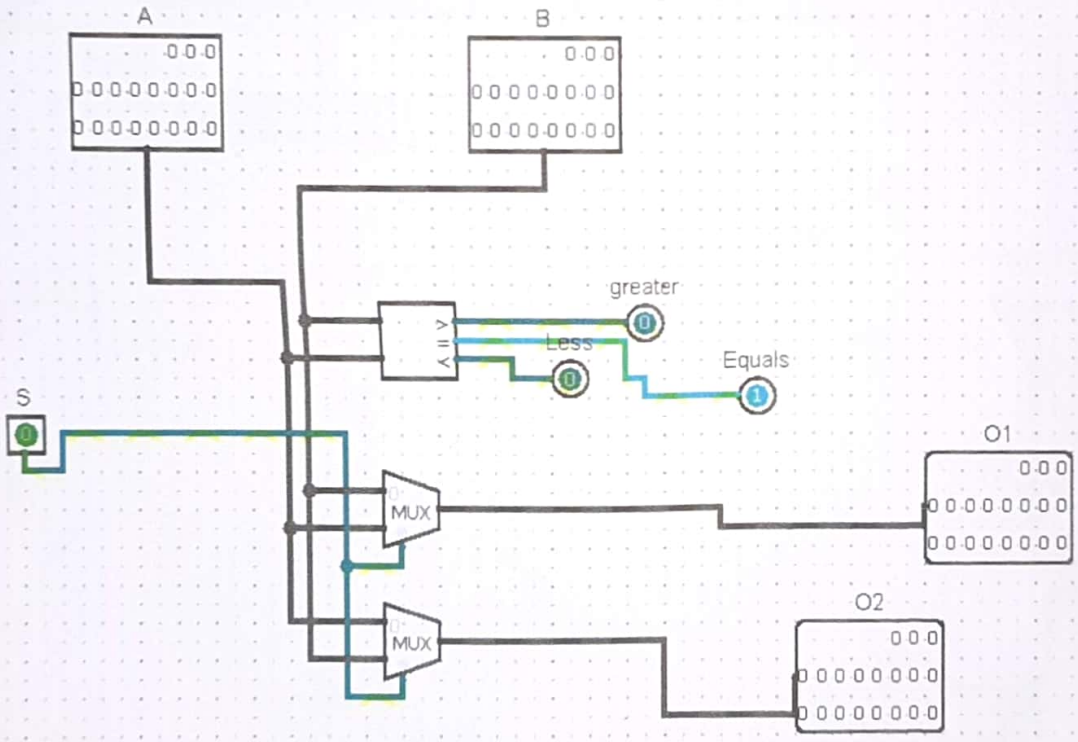

Fig: Significand Compare

fig: Shifting & Adding Significands

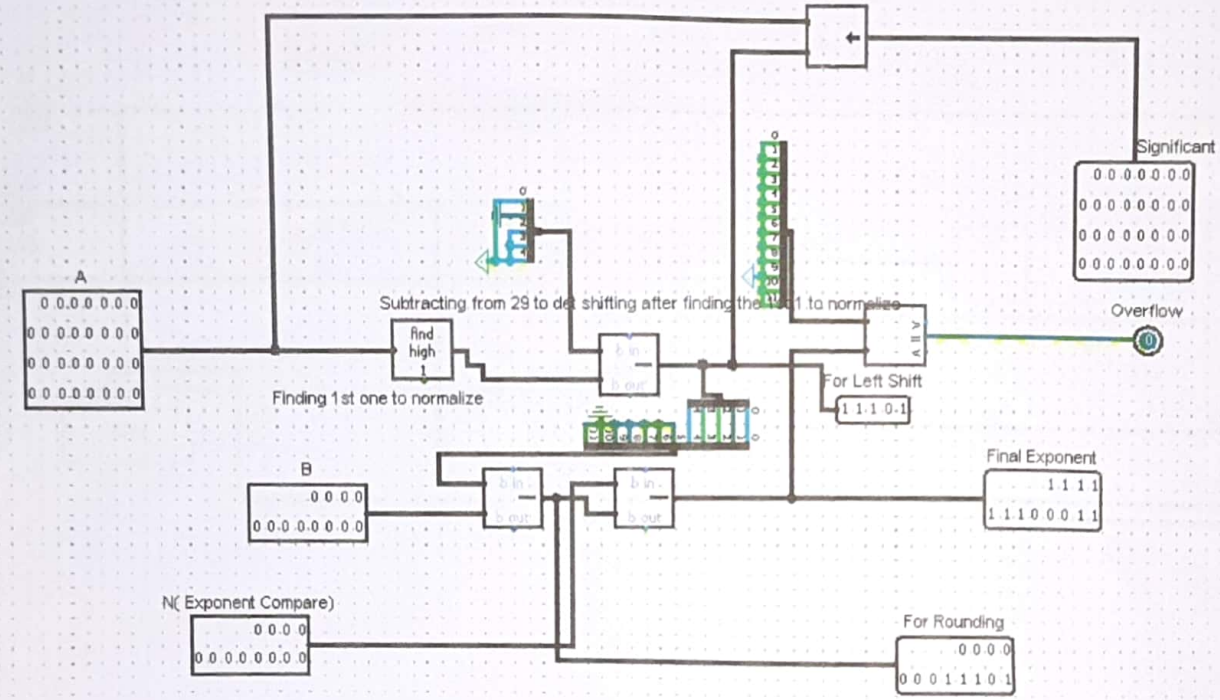

Fig: Normalize

Fig: Rounding

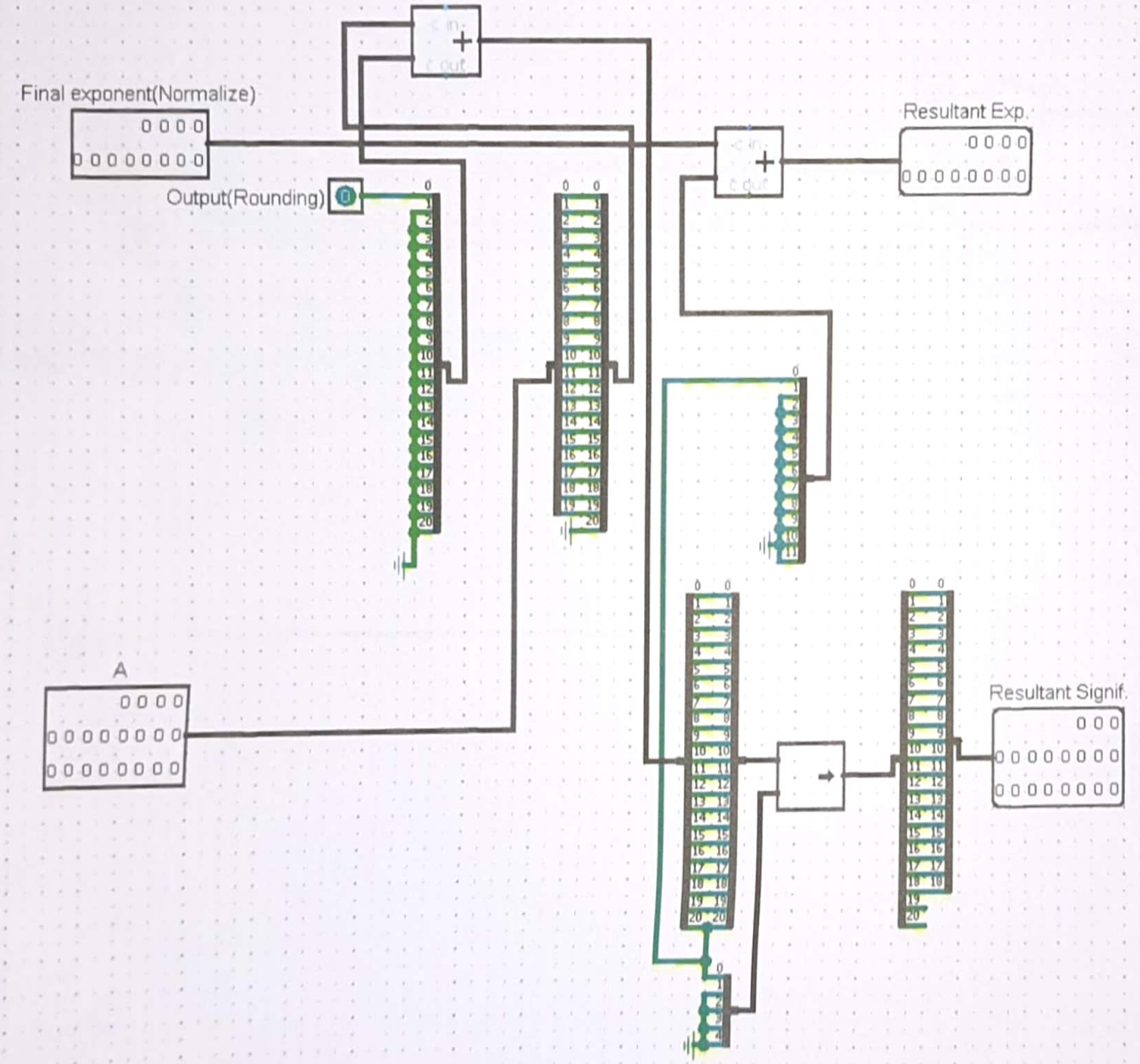

Fig: Truncate / Round Up

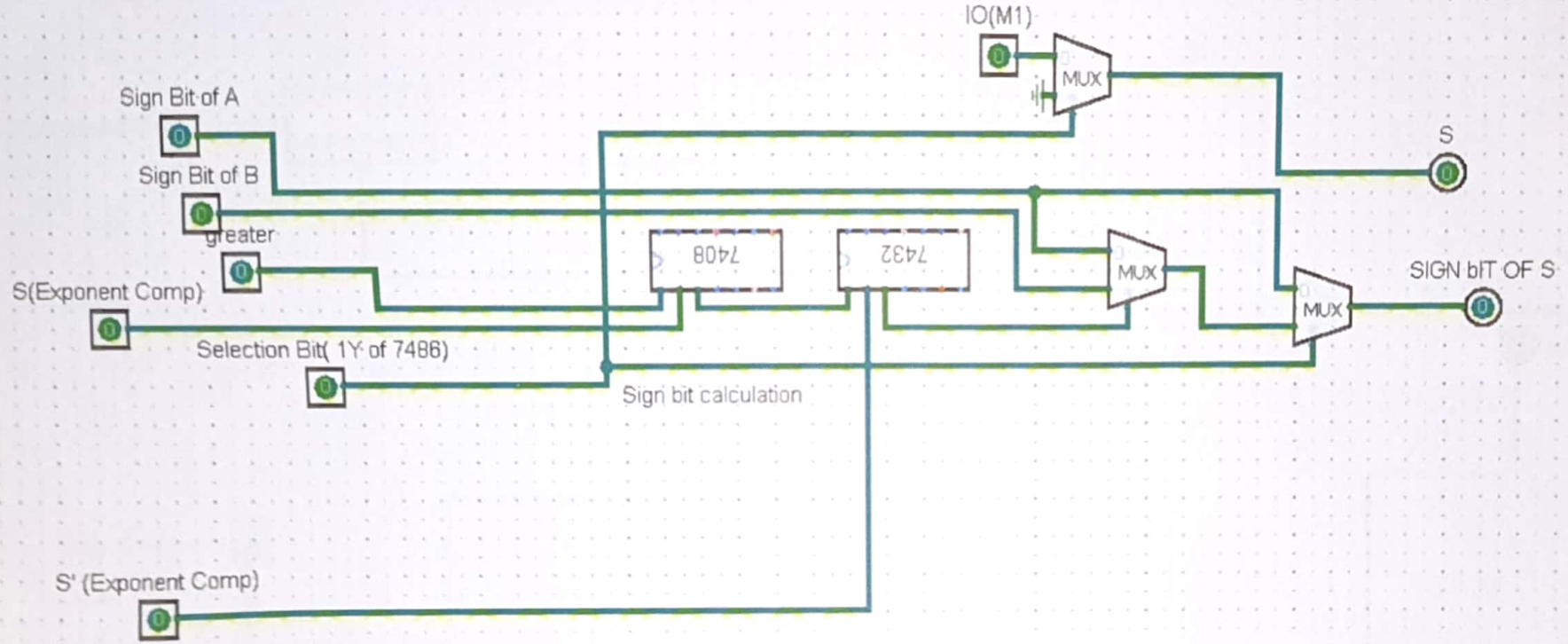

Fig: Sign Bit

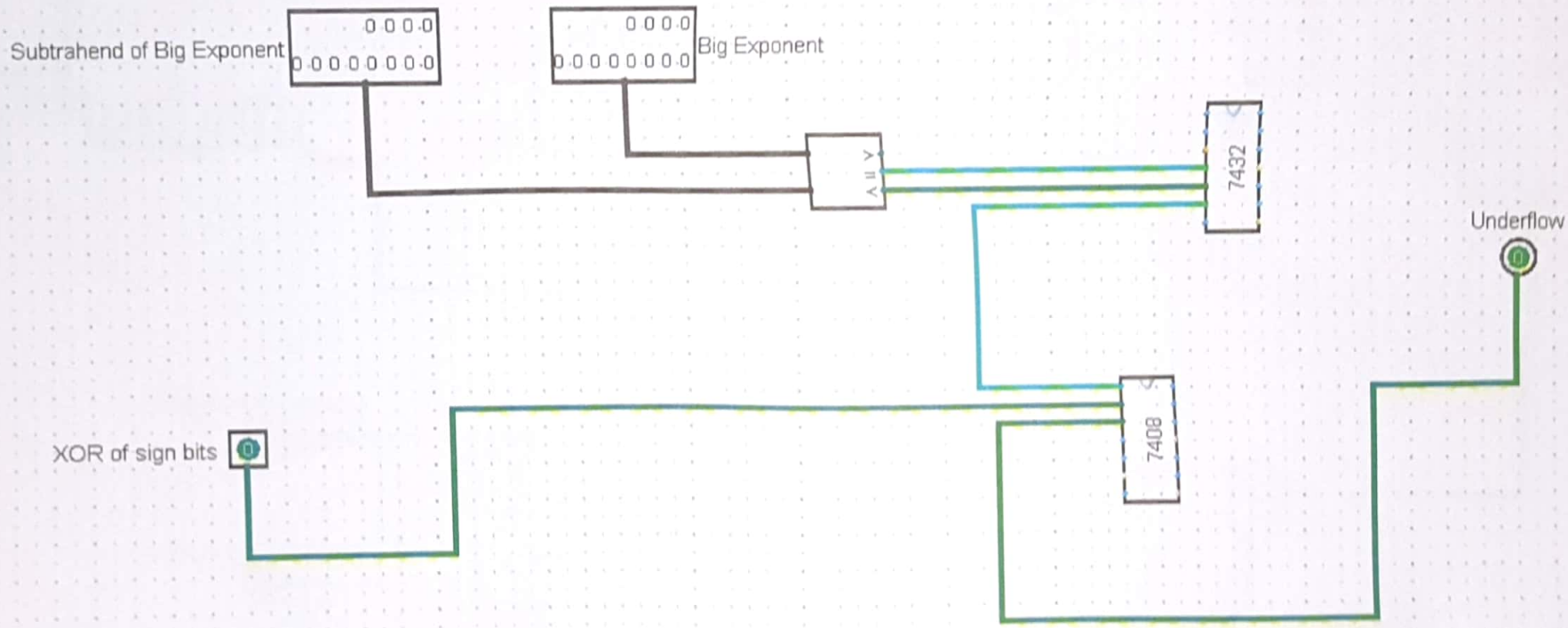

Fig: Underflow

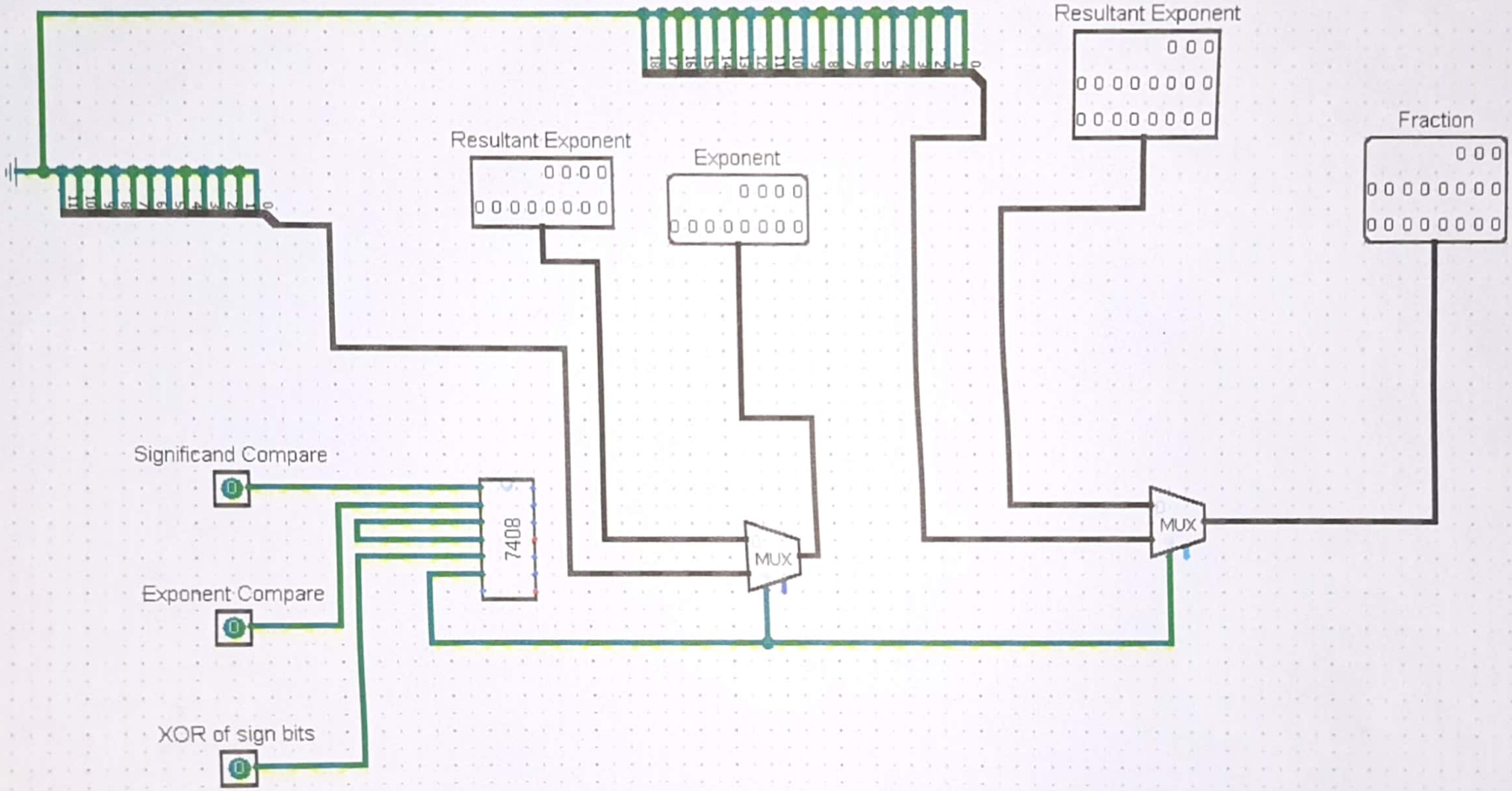

Fig: Zero_handling