Project report
ECN-104

Abstract

An Arithmetic Logic-Unit (ALU) is an integral circuital component of a computer processor (CPU). It is used to perform various arithmetic such as addition, subtraction, multiplication, division and bitwise logic operations such as OR and AND etc. The memory stores the program's instructions and data.

This project aims to design and simulate a floating-point single-precision (32 bit) ALU which performs arithmetic operations and logic operations. All of these operations are realized using Verilog HDL, compiled, and simulated using Modelsim. The Floating-point numbers are represented in strict accordance with the IEEE standard 754.

Introduction

Today, for efficient computation of very large or small numbers, floating-point representation has become highly instrumental. If used a lot of power, time memory would be saved along with increase in accuracy and easy implementation. This representation is called floating-point as the values of the mantissa bits can 'float' along with the decimal point, based on the number's given value.

An Arithmetic-Logic Unit (ALU) is a combinational digital electronic circuit representing the fundamental building block of the Central Processing Unit (CPU) of a computer. It can carry out arithmetic operations like addition, subtraction, multiplication, and division, and various logic operations like AND, OR, NOT, XOR, etc.

A. Single Precision IEEE Format

Any floating-point number can be divided into three main components:

- ***Sign:*** That indicates the sign of the number. 0 is used for denoting positive and 1 for denoting negative numbers.
- ***Exponent:*** It contains the base number value.
- ***Mantissa:*** Sets the value of the number.

In a Single Precision Format, the bits will be divided in the following manner:

- The first bit (or the 31st bit) is set as the signed bit (S) of the number.
- The next 8 bits (from 30th to 23rd bit) represent the exponent (E).
- The remaining 23 bits (from 22nd to 0) are allotted the mantissa.

The Standard IEEE 754 specifies:

- The formats for binary and decimal floating-point data (for computational purposes).
- Various operations such as addition, subtraction, multiplication, etc.
- Inter-conversion between the integer-floating point formats.
- Different properties which need to be satisfied when rounding off numbers.
- Floating-point exceptions and their handling.

Binary Conversion of Decimal Number to Floating point

First step of this conversion is to convert the given decimal number to its binary equivalent via decimal to binary conversion. Now we have a number in binary sequence and we need to convert it into floating point. For this, firstly we need to find the number of bits before the decimal point. That would be equal to **(1 + exponent(e))** by the explanation in the previous conversion. Now we shift the decimal point to the position where there is only one bit to the left of the decimal point (which would always be 1). For example if the binary equivalent of decimal number is 100101001.01001, then exponent (e) is equal to 9 (number of bits to the left of the decimal point) - 1 = 8. And the binary equivalent now becomes 1.0010100101001. Now this exponent value plus the exponential bias (in this case exponential bias= 127) is converted into its binary equivalent of 8 bits and these 8 bits would be the 8 bits after the sign bit in the floating point representation.

Now we determine the sign bit. This can be done just by looking at the sign of the decimal number. If it is positive, then the sign bit will be 0 and if it is negative, the sign bit would be 1.

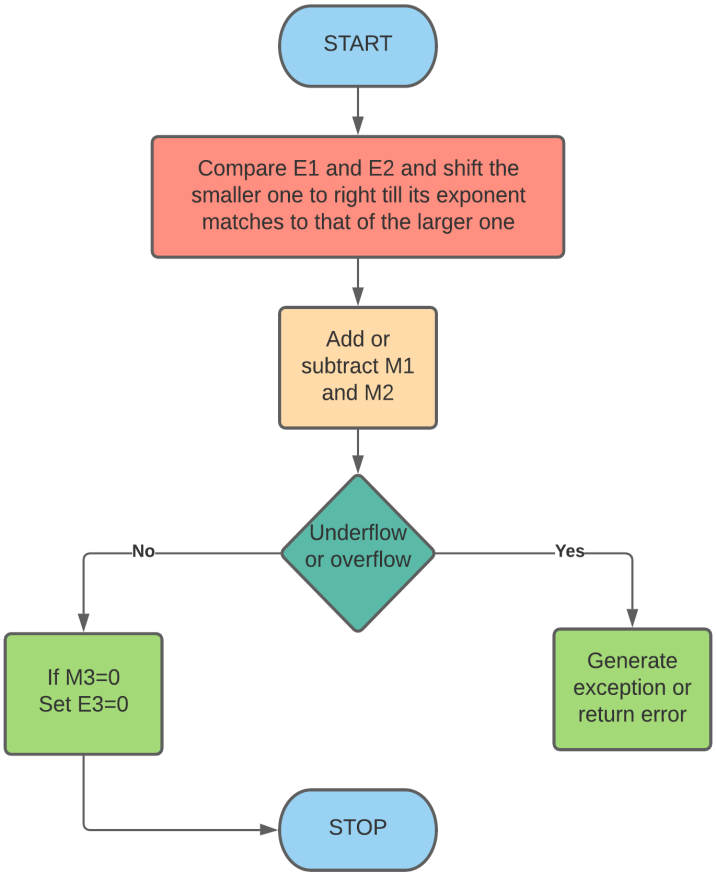
Now the last step is to determine the remaining 23 bits of the floating point. These can be determined by the bits after the decimal point in the remaining binary number. These are also called mantissa. The next 23 bits after the decimal point are therefore the last 23 bits of the floating point representation of the decimal number. If the binary number terminates before 23 bits, then all the remaining bits can be written as 0 so they don't contribute anything to the value. For example if the binary number after the second step is 1.00100110010000010001 then the last 23 bits of the floating point representation (mantissa) = 00100110010000100010000.

OPERATIONS

(1) Addition-Subtraction

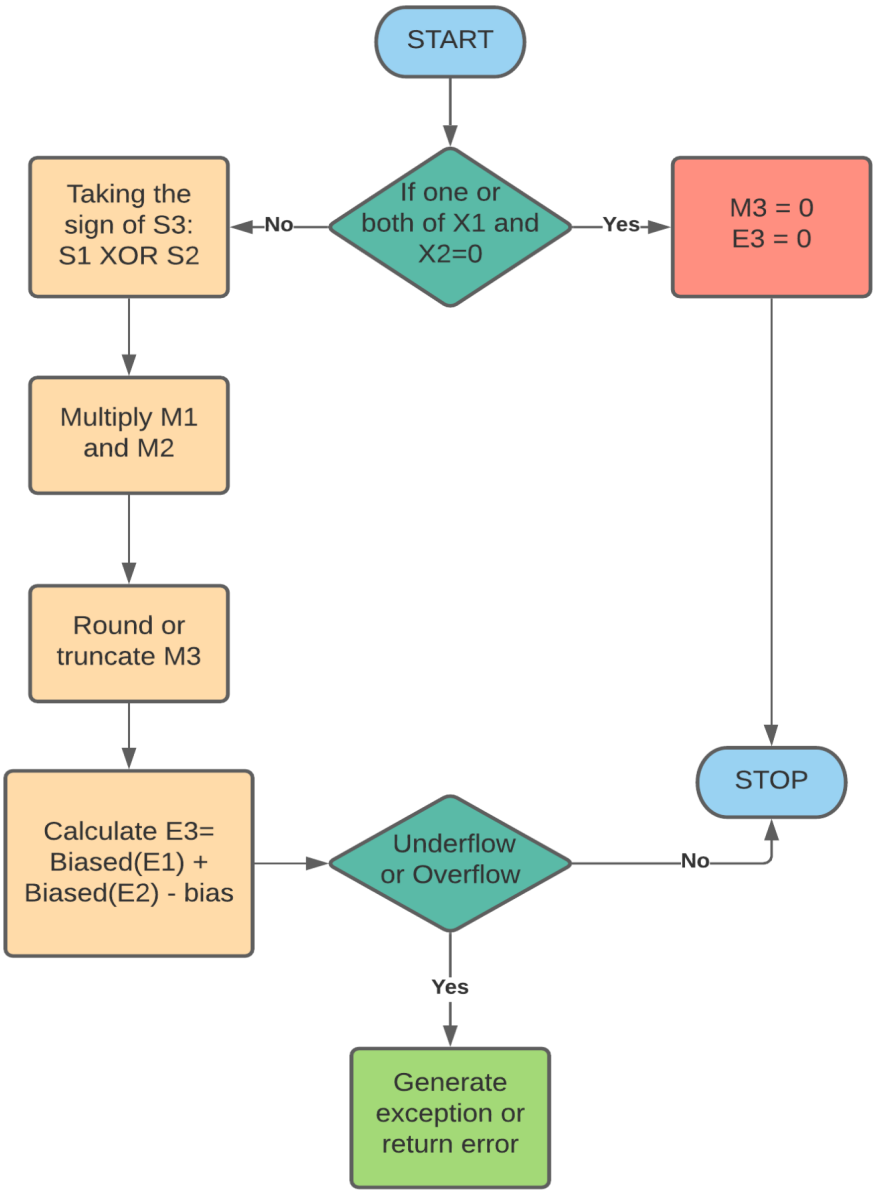
In addition or subtraction, first of all we check the leftmost bit which is called the **sign bit**. If the bit is 0, then the decimal equivalent of the binary number will be **positive**. But if the leftmost bit is 1, then the decimal equivalent of the binary number will be **negative**. The addition or subtraction of both positive or both negative numbers take place according to the normal addition or subtraction rules (where a carry of 1 and a sum of 0 is produced when two 1s are added).

But if both the numbers are of different sign, then we need to take 2's complement of one of the numbers, then add it with another number and then again take 2's complement of the result. In a mathematical way, A+(-B) = A-B= 2^n-(2^n-A+B).



(2) Multiplication

Binary Multiplication also takes place as per normal multiplication convention in which the two numbers are listed one below other and each bit is multiplied to the number one by one. Just some points are different here. No borrow or carry method is applicable here. Also, we need to find the exception flags and also the special values for overflow and underflow.



Challenges faced during implementation

As everything is online which means the internet has much effect on the project. A number of Internet-related issues were faced by all of the members of the group, mostly due to unprecedented network fluctuations and data limitations, and the Internet being a resource of utmost importance for this project, we were not able to set up Modelsim on our systems and the group had to face a lot of challenges during the learning and simulation phases. Choosing a proper working data-propagation model was another challenge. Novice errors in writing the Verilog code due to a lot of compile-time errors were quite common initially and the code didn't work as expected sometimes. Underflow and Overflow troubled the Multiplier and Divider modules quite a lot initially, but later their effect was reduced to a large extent by scaling the exponents.

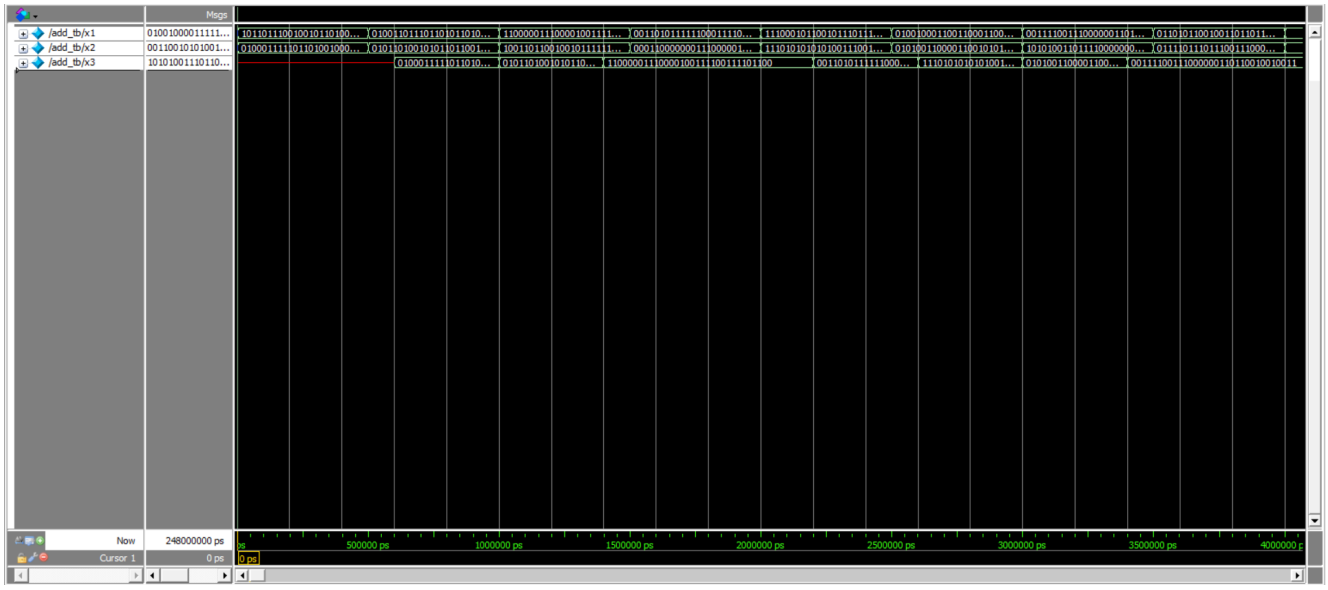
Simulation Results:

Operation	**Tests done**	**Tests passed**	**Tests failed**	**Accuracy**
Addition	500	496	4	99.2%
Subtraction	500	494	6	98.8%
Multiplication	500	374	126	74.8%

Division	500	368	132	73.6%
AND	500	500	0	100%
OR	500	500	0	100%
NOT	500	500	0	100%

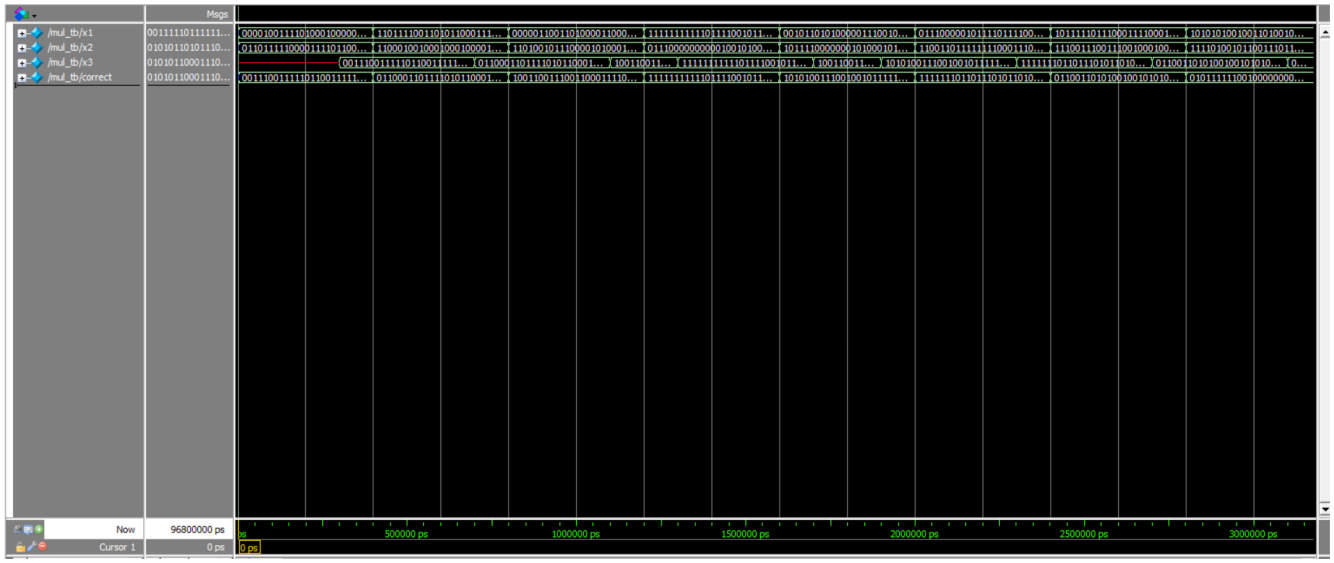
Simulation

1. **Addition**

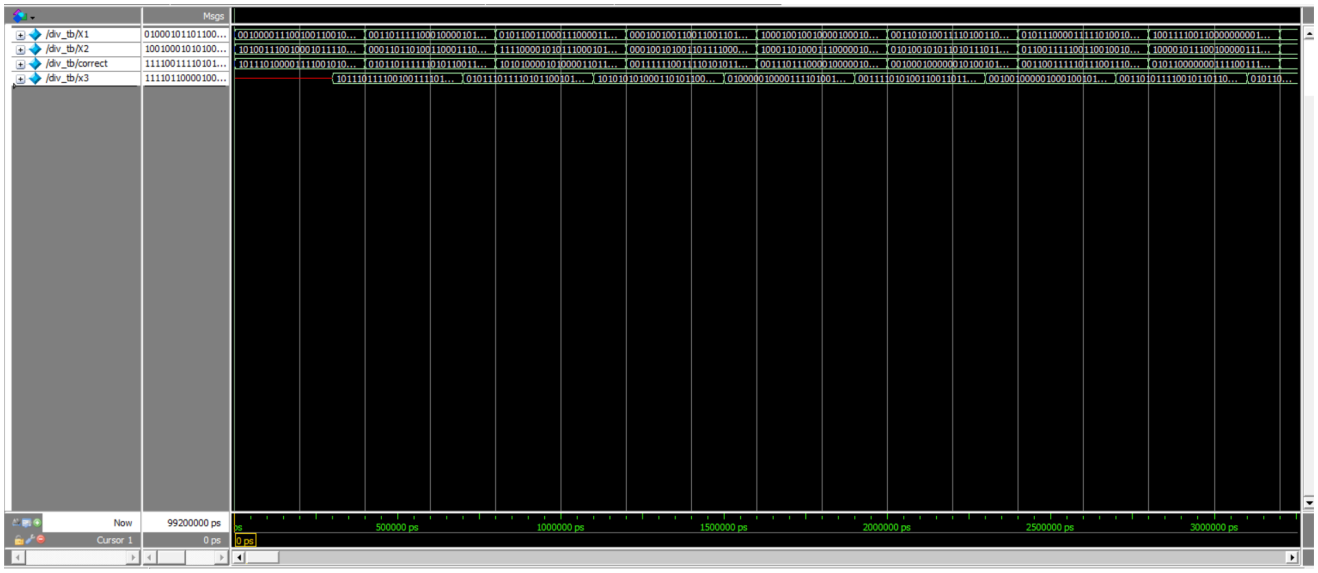


2. Subtraction

3. Multiplication

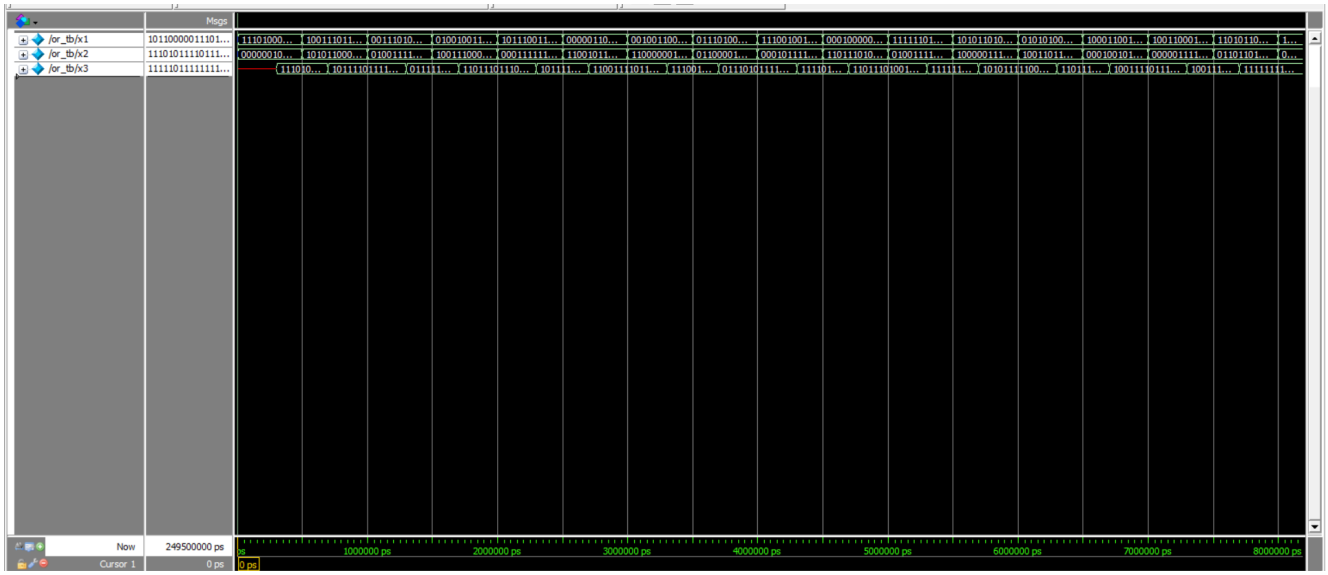


4. Division

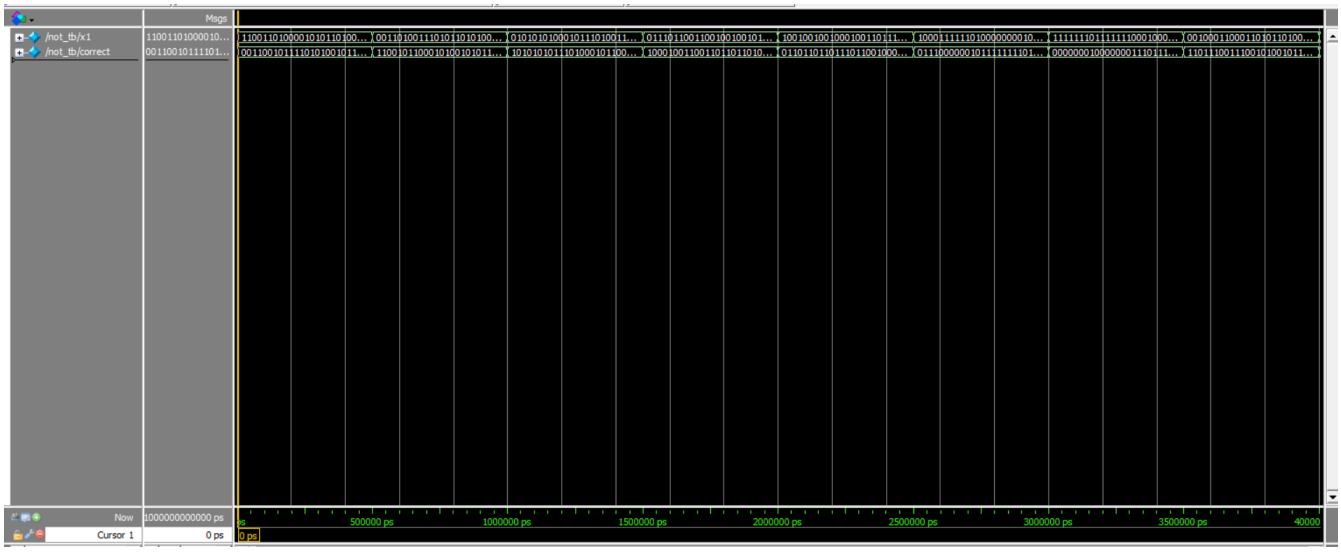


5. AND

6. OR


Msgs
/or_tb/x1
/or_tb/x2
/or_tb/x3
Now
249500000 ps
Cursor 1
0 ps
1000000 ps
2000000 ps
3000000 ps
4000000 ps
5000000 ps
6000000 ps
7000000 ps
8000000 ps

7. NOT


Msgs
/not_tb/x1
/not_tb/correct
Now
1000000000000 ps
Cursor 1
0 ps
500000 ps
1000000 ps
1500000 ps
2000000 ps
2500000 ps
3000000 ps
3500000 ps

Main Code

https://github.com/anshul-11/32-bit-ALU-ECN-project

Conclusion

The Floating-Point Single-Precision Arithmetic-Logic Unit (ALU) has been discussed extensively in-depth and a suitable algorithm has been developed to successfully carry out various arithmetic operations namely addition, subtraction, multiplication and division and logic operations namely Bitwise NOT, Bitwise AND, Bitwise OR. The algorithm has been implemented in a pipelined way so as to minimize the delay and maximize computational efficiency.

References

Salah Hasan Alkurwy, “A novel approach of multiplier design based on BCD decoder”

A. Kant and S. Sharma,"Applications of vedic multiplier designs - A review"

Soumya Havaldar, K S Gurumurthy,“Design Of Vedic IEEE 754 Floating Point Multiplier”

Nisha singh and R Dhanabal, “Design Of Single Precision Floating Point Arithmetic Logic Unit Multiplier”

The Team

Sanchit Garg (20116086)
Anshul Gusain (20116014)
Yash Hirani (20116108)
Yash Mittal (20116109)
Ujjwal Kukreti (20119053)