International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 12 | Dec 2020 www.irjet.net p-ISSN: 2395-0072

Design and Simulation of 32-Bit Floating Point Arithmetic Logic Unit using VerilogHDL

Yagnesh Savaliya[1], Jenish Rudani[2]

[1]*Student, Dharmsinh Desai University (DDU), Nadiad, India*
[2]*Student, Dharmsinh Desai University (DDU), Nadiad, India*

Abstract - *Aim of the project is to design and simulation of single precision floating point ALU which is a part of math coprocessor. The main benefit of floating-point representation is that it can support a much wider range of values rather than fixed point and integer representation. Addition, Subtraction, Multiplication and division are the arithmetic operation in these computations. In this floating point unit, input should be given in IEEE 754 format, which represents 32 bit single precision floating point values. Main application of this arithmetic unit is in the math coprocessor which is generally known as DSP processor. In this DSP processor, for signal processing, value with high precision is required and as it is an iterative process, calculation should be as fast as possible. So a normal processor cannot fulfil the requirement and floating point representation came into the picture which can calculate this process very fast and accurately.*

Keywords: *Floating point, single precision, verilog, IEEE 754, math coprocessor, ALU*

1. INTRODUCTION

The floating point operations have found intensive applications in the various fields for the requirements or high precision operation due to its great dynamic range, high precision and easy operation rules. With the increasing requirements for the floating point operations for the high-speed data signal processing and the scientific operation, the requirements for the high-speed hardware floating point arithmetic units have become more and more exigent. The demand for floating point arithmetic operations in most of the commercial, financial and internet based applications is increasing day by day.[1] Floating point operations are hard to implement on reconfigurable hardware i.e. on FPGAs because of their algorithm's complexity. While many scientific problems require floating point arithmetic with upper level of accuracy in their calculations. Therefore verilog programming for IEEE single precision floating point unit have been explored. In this project, we are going to build a floating point arithmetic unit which is part of a math coprocessor and generally known as DSP processor. All the modules including Addition, Subtraction, Multiplication and Division are written using Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (Verilog HDL) and then compilation and simulation is performed using Altera QuartusII design software. This floating point arithmetic unit performs general arithmetic tasks such as addition, subtraction, multiplication and division. Which is also capable for giving output in special cases such as add by infinite, divide by zero etc. In this project, these four modules are built separately, Then In one common module named 'fpu' all are combined together and we can perform specific arithmetic operations with help of selection line.

The rest of the paper is organized as follows. Section 2 presents the general floating-point architecture. Section 3 explains the algorithms used to write Verilog codes for implementing 32-bit floating-point arithmetic operations: addition/subtraction, multiplication and division. The Section 4 of the paper details the Verilog code and behavior model for all above stated arithmetic operation. Section 5 of the paper shows the experimental results While section 6 concludes the paper with further scope of work.

2. Floating Point Architecture

Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 standard presents two different floating-point formats, Binary interchange format and Decimal interchange format. This paper focuses only on single precision normalized binary interchange format. Figure 1 shows the IEEE 754 single precision binary format representation; it consists of a one-bit sign (S), an eight-bit exponent (E), and a twenty-three-bit fraction (M) or Mantissa. [1]

32-bit Single Precision Floating Point Numbers IEEE standard are stored as:

- S EEEEEEEE
- MMMMMMMMMMMMMMMMMMMMMMM
    - S: Sign – 1 bit
    - E: Exponent – 8 bits
    - M: Mantissa – 23 bits Fraction

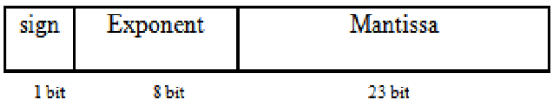

Fig -1: IEEE Floating Number Format

If E=255 and Significand is nonzero, then number is known as "Not a Number"(NaN)

If E=255 and Significand is zero and S is 1, then number is -Infinity (-∞)

If E=255 and F Significand is zero and S is 1, then number is Infinity (∞)

N = $(-1)^{s} * 2^{(E-Bias)} * (1.M)$;Where Bias=127

An extra bit is added to the mantissa to form what is called the significand. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number.

3. Algorithm for Floating Point Arithmetic Unit

The algorithms using flow charts for floating point addition/subtraction, multiplication and division have been described in this section, that become the base for writing Verilog codes for implementation of 32-bit floating point arithmetic unit. Algorithm of each unit is shown below:

3.1 ADDER/SUBTRACTOR

The algorithm for floating point addition is explained through the flow chart in Figure 2. While adding the two floating point numbers, two cases may arise. [4] Case I: when both the numbers are of the same sign i.e. when both the numbers are either +ve or –ve. In this case MSB of both the numbers are either 1 or 0. Case II: when both the numbers are of different sign i.e. when one number is +ve and other number is –ve. In this case MSB of one number is 1 and the other is 0. [3]

3.1.1 When both numbers are of same sign

Step 1: - Enter two numbers N1 and N2. E1, S1 and E1, S2 represent exponent and significand of N1 and N2 respectively.

Step 2: - Is E1 or E2 =‟0‟. If yes; set hidden bit of N1 or N2 is zero. If not; then check if E2 > E1, if yes swap N1 and N2 and if E1 > E2; contents of N1 and N2 need not to be swapped.

Step 3: - Calculate difference in exponents d=E1-E2. If d = „0‟ then there is no need of shifting the significand. If d is more than „0‟ say „y‟ then shift S2 to the right by an amount „y‟ and fill the left most bits by zero. Shifting is done with a hidden bit.

Step 4: - Amount of shifting i.e. „y‟ is added to the exponent of N2 value. New exponent value of E2= (previous E2) + „y‟. Now the result is in normalize form because E1 = E2.

Step 5: - Check if N1 and N2 have different sign, if “no”;

Step 6: - Add the significands of 24 bits each including hidden bit S=S1+S2.

Step 7: - Check if there is carry out in significand addition. If yes; then add „1‟ to the exponent value of either E1 or new E2. After addition, shift the overall result of significand addition to the right by one by making MSB of S as „1‟ and dropping LSB of significand.

Step 8: - If there is no carry out in step 6, then the previous exponent is the real exponent.

Step 9: - Sign of the result i.e. MSB = MSB of either N1 or N2.

Step 10: - Assemble result into 32-bit format excluding 24th bit of significand i.e. hidden bit.

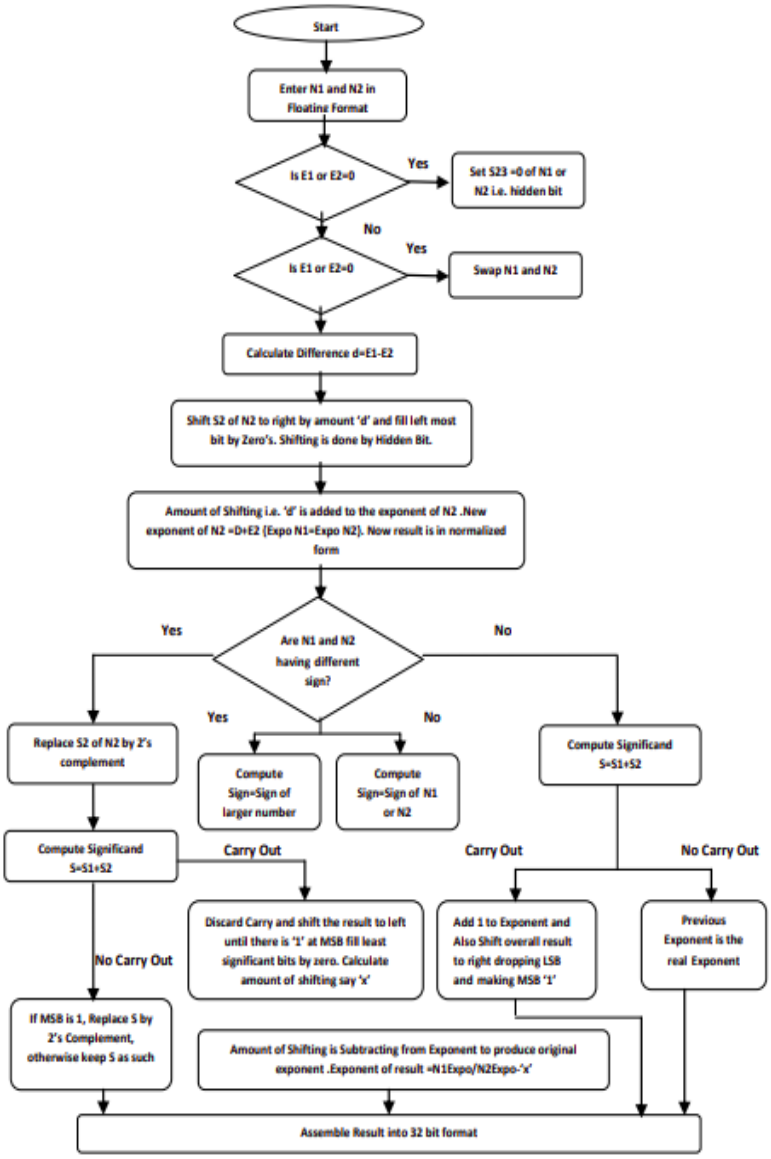

Fig -2: Flow chart of Floating-point Addition/Subtraction

3.1.2 When both numbers are of different sign

Step 1, 2, 3 & 4 are the same as done in case I.

Step 5: - Check if N1 and N2 have different sign, if "Yes";

Step 6: - Take 2"s complement of S2 and then add it to S1 i.e. S=S1+ (2"s complement of S2).

Step 7: - Check if there is carry out in significand addition. If yes; then discard the carry and also shift the result to the left until there is „1" in MSB and also count the amount of shifting say "z".

Step 8: - Subtract "z" from exponent value either from E1 or E2. Now the original exponent is E1-" z". Also append the "z" number of zeros at LSB.

Step 9: - If there is no carry out in step 6 then MSB must be "1" and in this case simply replace "S" by 2's complement.

Step 10: - Sign of the result i.e. MSB = Sign of the larger number either MSB of N1or it can be MSB of N2.

Step 11: - Assemble result into 32-bit format excluding 24th bit of significand i.e. hidden bit.

3.2 MULTIPLIER

The algorithm for floating point multiplication is explained through the flow chart in Figure 3. Let N1 and N2 are normalized operands represented by S1, M1, E1 and S2, M2, E2 as their respective sign bit, mantissa (significand) and exponent. Basically, the following four steps are used for floating point multiplication.

1. Multiply significands, add exponents, and determine sign

M = M1*M2

E = E1+E2-Bias

S = S1 XOR S2

2. Normalize Mantissa M (Shift left or right by 1) and update exponent E

3. Rounding the result to fit in the available bits

4. Determine exception flags and special values for overflow and underflow.

Sign Bit Calculation: The result of multiplication is a negative sign if one of the multiplied numbers is of a negative value and that can be obtained by XORing the sign of two inputs. Exponent Addition is done through unsigned adder for adding the exponent of the first input to the exponent of the second input and after that subtract the Bias (127) from the addition result (i.e. E1+E2-Bias). The result of this stage can be called an intermediate exponent.

Significand Multiplication is done for multiplying the unsigned significand and placing the decimal point in the multiplication product. The result of significand multiplication can be called as intermediate product (IP). The unsigned significand multiplication is done on 24 bits.

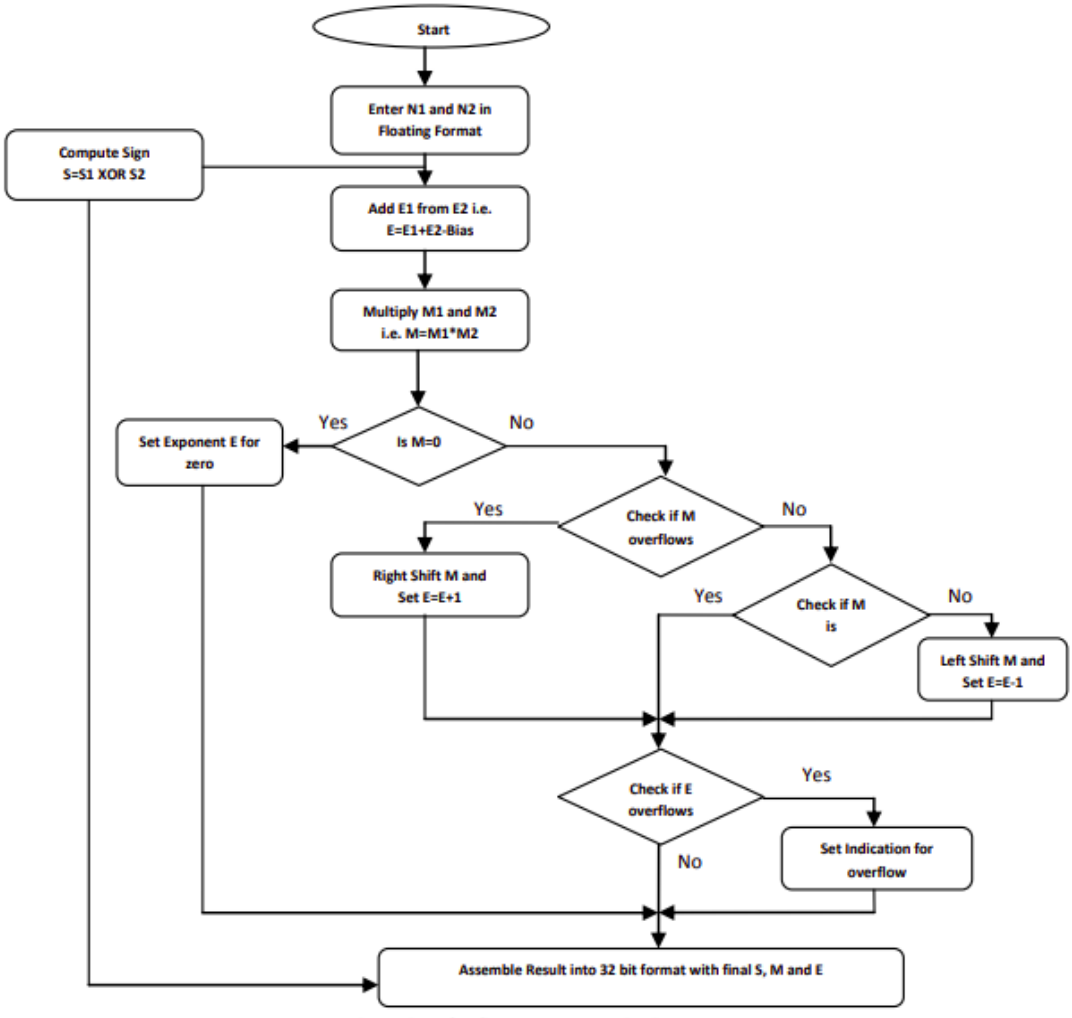

Fig - 3: Flow chart of Floating point Multiplication

The result of the significand multiplication (intermediate product) must be normalized to have a leading "1" just to the left of the decimal point (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47. If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift is needed. If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1. Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. The exponent of the result must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one. An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition can be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it is an underflow that can never be compensated; if the intermediate exponent = 0 then it is an underflow that may be compensated during normalization by adding 1 to it. When an overflow occurs an overflow flag signal goes high and the result turns to ±Infinity (sign determined according to the sign of the floating-point multiplier inputs). When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating-point multiplier inputs). [5]

3.3 DIVIDER

The algorithm for floating point multiplication is explained through the flow chart in Figure 4. Let N1 and N2 are normalized operands represented by S1, M1, E1 and S2, M2, E2 as their respective sign bit, mantissa (significand) and exponent. If let us say we consider x=N1 and d=N2 and the final result q has been taken as "x/d". Again, the following four steps are used for floating point division.

1. Divide significands, subtract exponents, and determine sign

M=M1/M2

E=E1-E2

S=S1XORS2

2. Normalize Mantissa M (Shift left or right by 1) and update exponent E

3. Rounding the result to fit in the available bits

4. Determine exception flags and special values

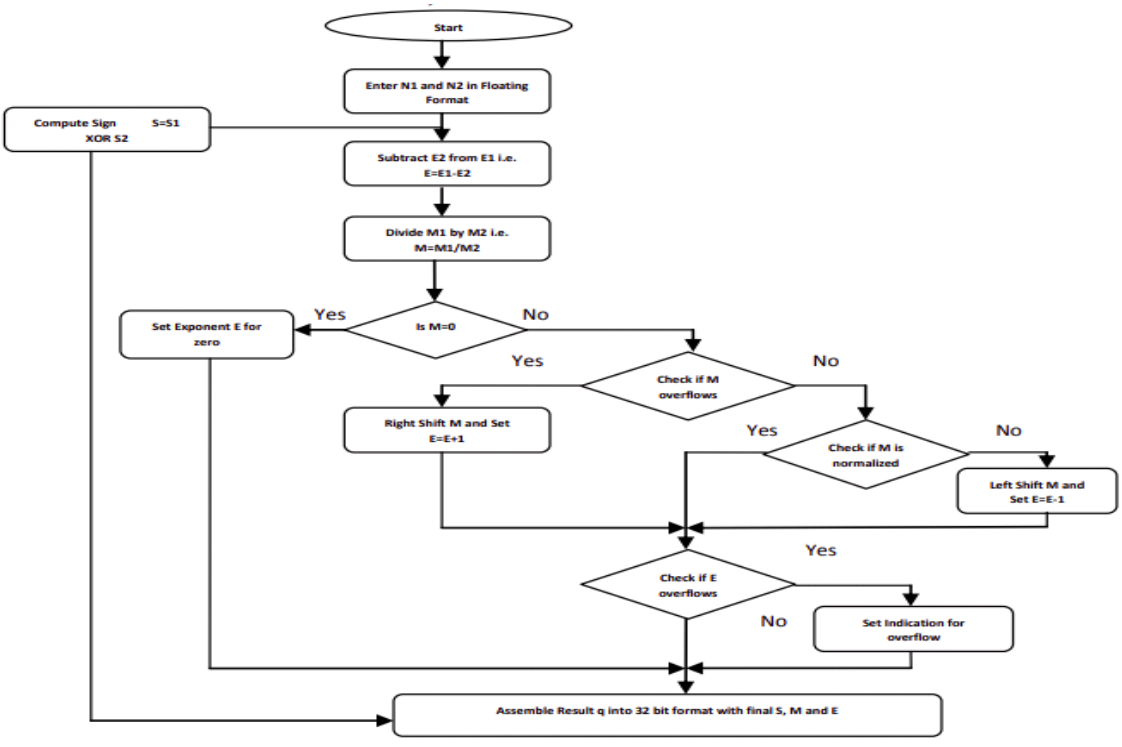

Fig - 4: Flow Chart for floating point Division (q = x/d; N1=x and N2=d)

4. EXPERIMENTAL RESULTS

4.1 RTL View

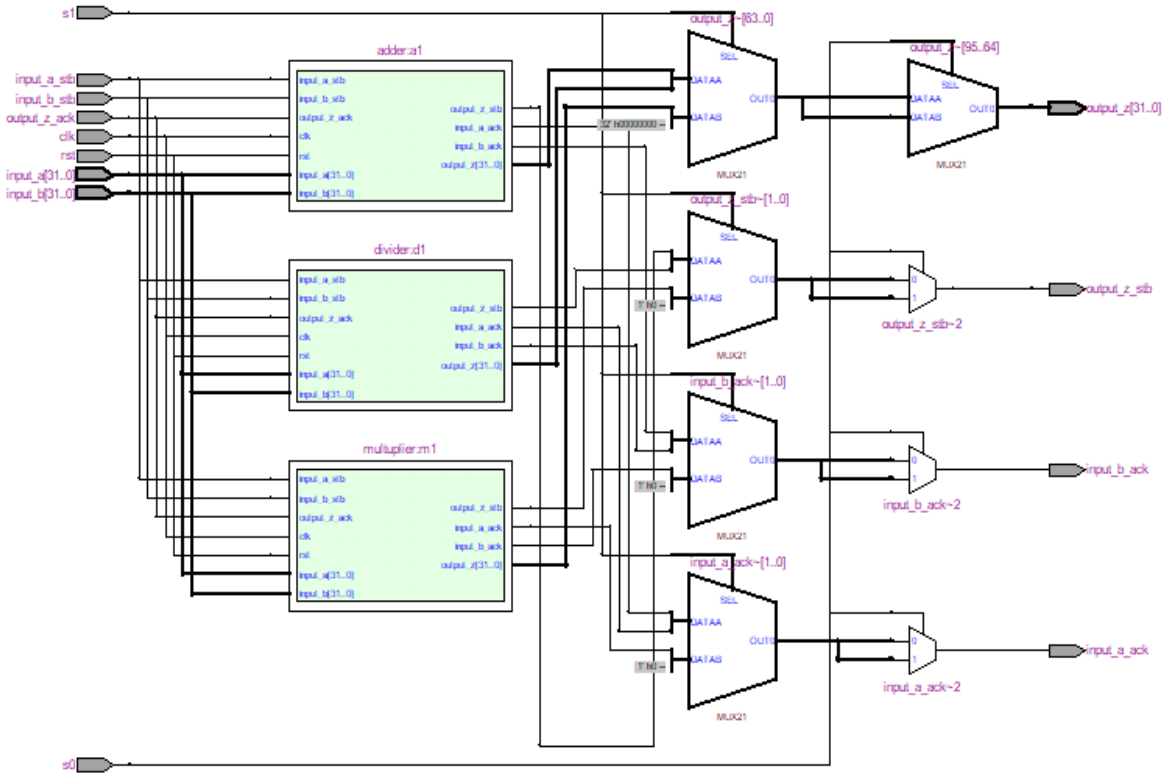

Fig - 5: RTL View

4.2 Output waveforms

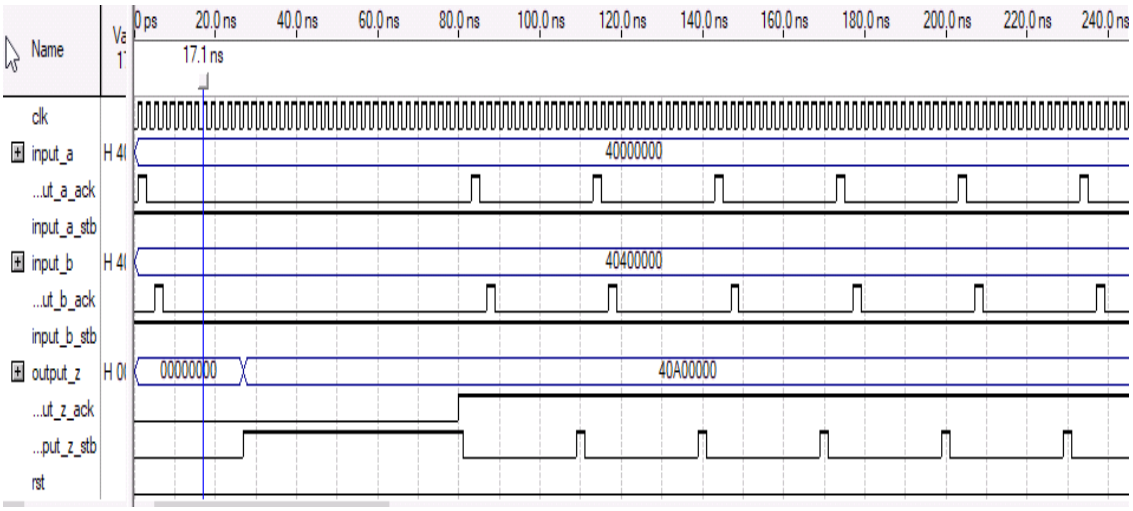

Fig -6: Simulation result of addition of 2 and 3 in floating point format

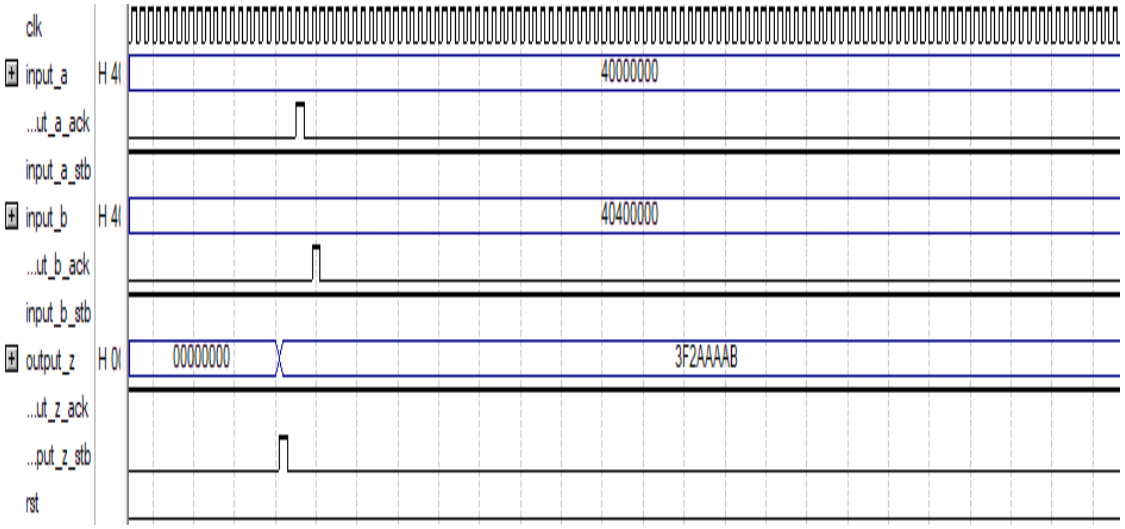

Fig -7: Simulation result of subtraction of 2 and 3 in floating point format

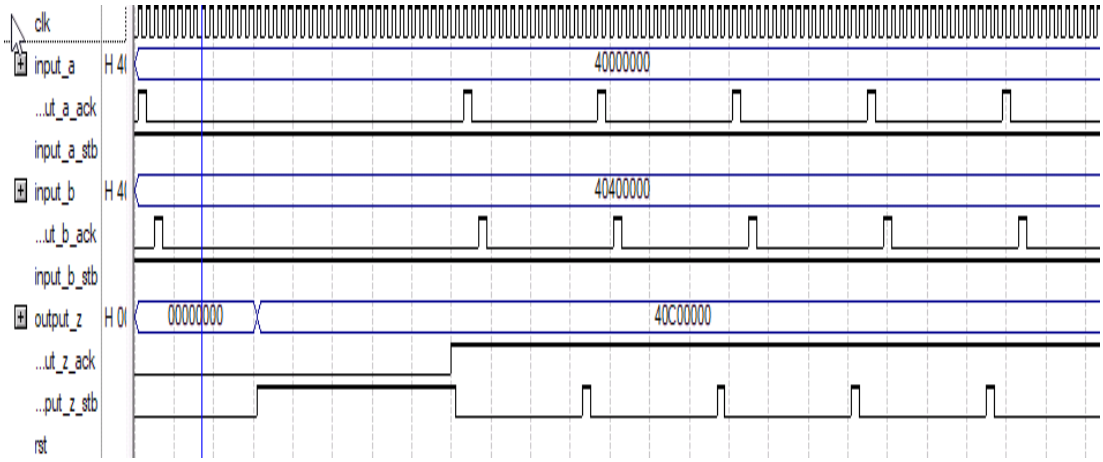

Fig -8: Simulation result of multiplication of 2 and 3 in floating point format

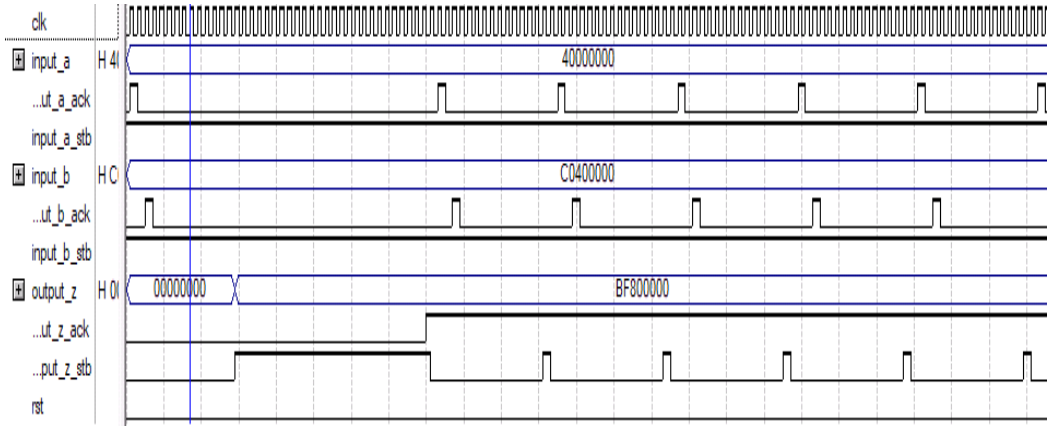

Fig - 9: Simulation result of division of 2 and 3 in floating point format

Table - 1: Tabular Result

Operation	Selection Line	Input 1	Input 2	Output
Addition	00	32'h4000000	32'h4040000	32'h40a00000
Subtraction	00	32'h4000000	32'hc0400000	32'hbf800000
Multiplication	01	32'h4000000	32'h4040000	32'h40c00000
Division	10	32'h4000000	32'h4040000	32'h3f2aaaab

5. Conclusion and Future Scope of Work

This project presents an implementation of an efficient 32-bit floating-point arithmetic unit using Verilog with aim of analyzing the problem during implementation and understanding the way to overcome the problem in order to enhance the system performance.

To perform any arithmetic operation, users have the knowledge about floating point representation as per IEEE 754 standard to provide input to the FPU. So, for future work, we can make one converter which can convert the decimal number into the IEEE 754 format and give these values as the input of FPU. After performing the operation, FPU gives the output in terms of IEEE 754 format. So, one converter can convert this format into decimal representation and give the output as a decimal number system. So, the users who are not aware about IEEE 754 format can also use this arithmetic unit.

6. References

[1] Reshma Cherian#, Nisha Thomas*, Y.Shyju# "Implementation of Binary to Floating Point Converter using HDL"pp. 461-64,©2013 IEEE

[2] Sunita.S.Malaj, S.B.Patil, Bhagappa.R.Umarane, "VHDL Implementation of Interval Arithmetic Algorithms for Single Precision Floating Point Numbers" International Journal of Scientific & Engineering Research Volume 4, Issue3, March-2013.

[3] "Design and Implementation of IEEE-754 Addition and Subtraction for Floating Point Arithmetic Logic Unit",V.vinay chamkur

[4] Preeti Sudha Gollamudi, M. Kamaraju, " Design of High performance IEEE-754 single precision (32 bit) floating point adder using VHDL. IJERT, Vol.2 Issue 7, pp. 2264-75, July-2013.

[5] Guillermo Marcus, Patricia Hinojosa, Alfonso Avila and Juan Nolazco-Flores "A Fully Synthesizable Single-Precision,Floating Point Adder/Substractor and Multiplier in VHDL for General and Educational Use," Proceedings of the Fifth IEEE International Caracas Conference on Devices, Circuits and Systems, Dominican Republic, Nov.3-5, 2004.