I-Chip

Problem Statement-1

Report

Team name-Bajrang dal

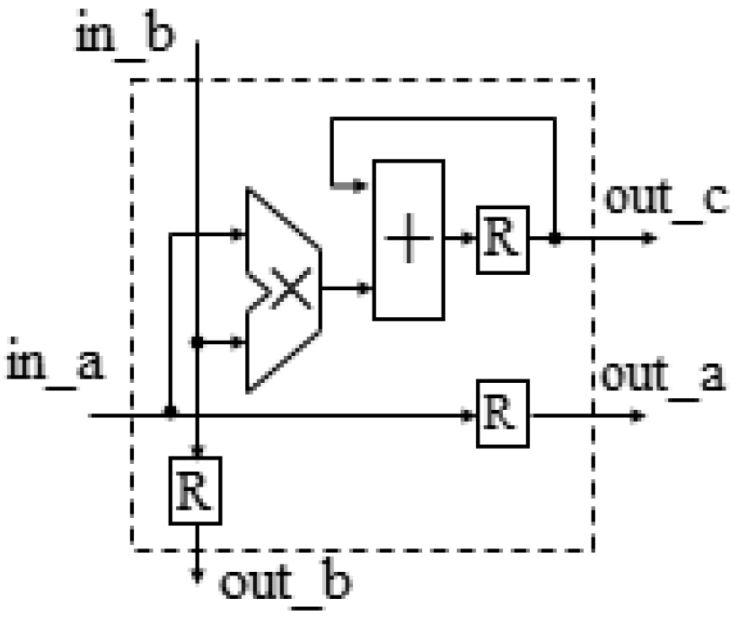
Team Members- Bhuvnesh Aggrawal , Dushyant Agrawal

Task :

The task at hand is to design and optimize a Processing Element (PE) that will serve as a fundamental building block for the AI accelerator.

Abstract :

The problem statement requires us to develop a Floating Point Multiplier, Floating Point Adder and then integrate all the components using registers to form a module as shown below:



Aim of the project is to design and simulation of single precision floating point ALU which is a part of math coprocessor. Floating point Complex numbers is an essential operation in DSP and communication applications with high speed and less power requirement. 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. 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

Introduction :

Floating-point addition is the most frequent floating-point operation and accounts for almost half of the scientific operation. Therefore, it is a fundamental component of math coprocessor, DSP processors, embedded arithmetic processors, and data processing units. These components demand high numerical stability and accuracy and hence are floating- point based.

II. IEEE 754 Precision Binary Format

The IEEE 754 Single Precision Binary Format is as shown below:

Fig (1): Single Precision Format

it consists of a one bit sign (S), an eight-bit exponent (E), and a twenty-three-bit fraction (M) or Mantissa.

Example:

0	1000 1000	11100111111000000000000

Fig (2): Single Precision Format of 975.75_{10}

General Flowchart for 32 bit floating point addition:

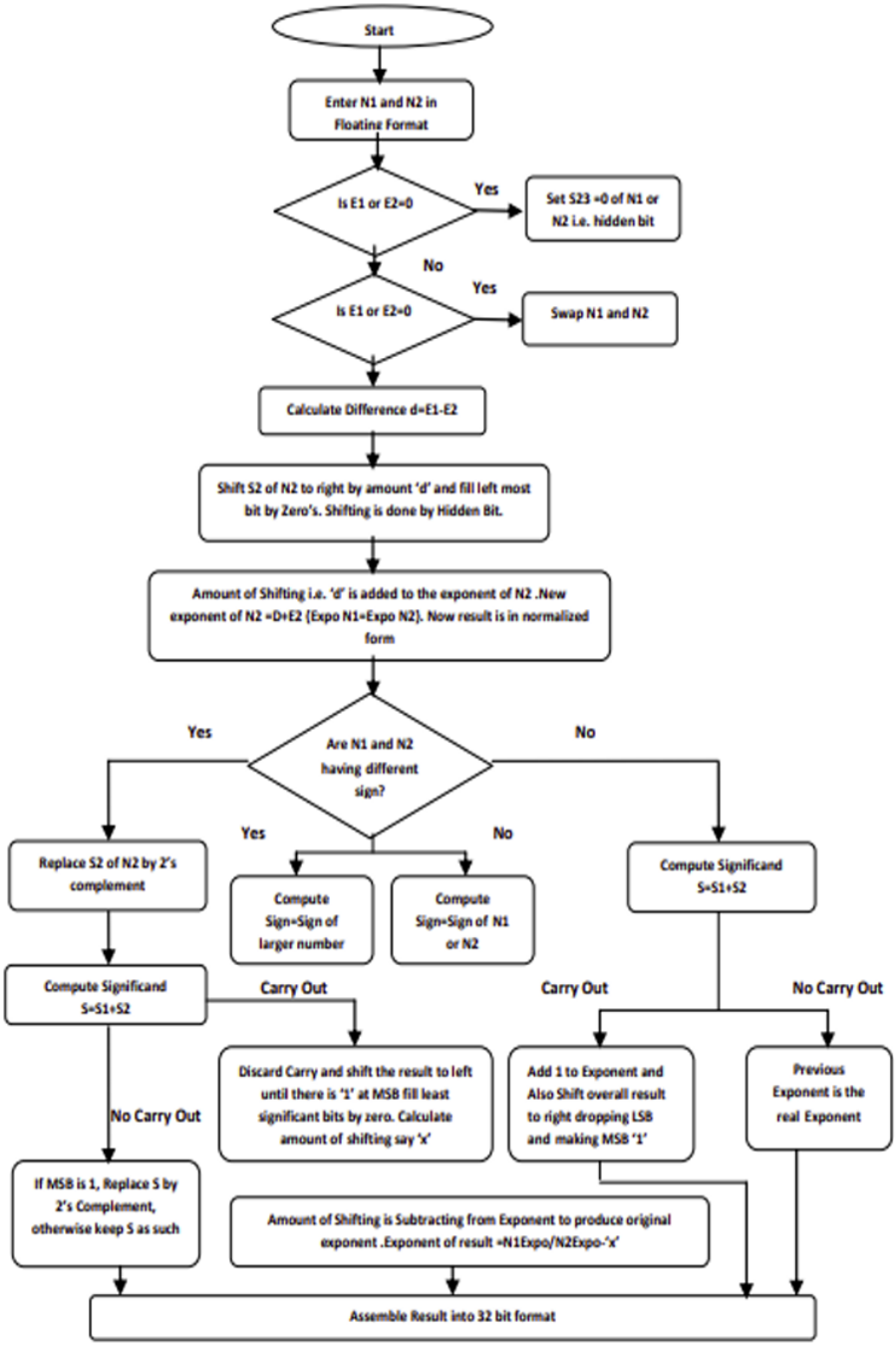


Fig (3) : Flow chart of Floating-point Addition/Subtraction

General Flowchart for 32 bit floating point multiplication :

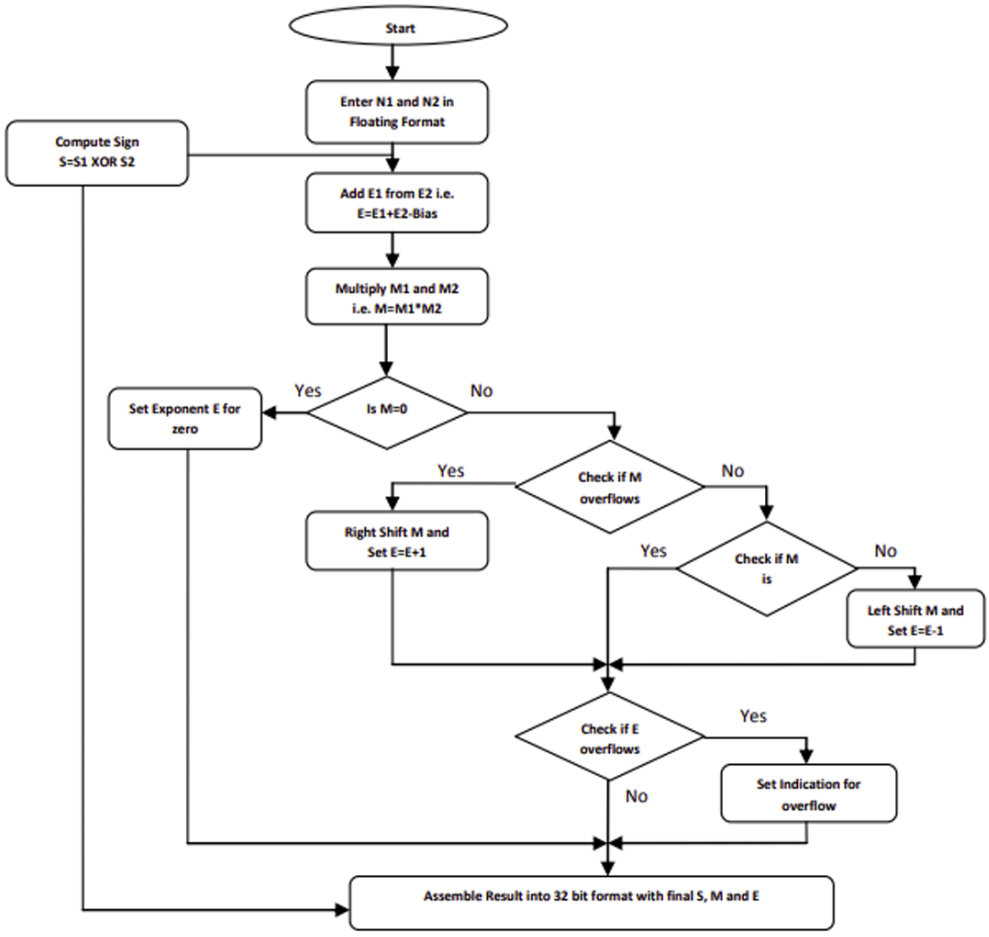


Fig (4) : Flow chart of Floating point Multiplication

Design Methodology :

Floating Point Adder:

The general pipeline for the adder is given in Fig (3), and the way we have implemented it using behavioral flow of verilog is as follows:

Case 1 : If exponent of both numbers are equal,

Compare there sign

▼ If sign of both the number is equal add there mantissa part with the hidden 1

- if the result of addition can be normalized then normalize the

▼ If the sign of both the number is different

Compare there fraction parts:

▼ If fraction of both the numbers are equal:

mantissa part and add one to the exponent and output the result.

- if the result of the addition is already normalized the directly output the result.

1. The final result will be simply 0.

▼ If fraction of first is greater than second:

1. final sign will be the sign of first number.
2. subtract the fraction of second number from the first.
3. normalize the obtained result and properly manage the exponent with the normalization.
4. output the obtained result.

▼ If fraction of second number is greater than the first:

1. final sign will be the sign of second number.
2. subtract the fraction of first number from the second.
3. normalize the obtained result and properly manage the exponent with the normalization.
4. output the obtained result.

Case 2 : If exponent of first number is greater than the second,

Compare there sign

▼ If sign of both the number is equal

1. sign of final result will be the sign of first number.
2. Right shift the mantissa part of second number by exp_a-exp_b.

▼ If the sign of both the number is different

1. sign of final result will be the sign of first number.
2. Right shift the mantissa part of second number by exp_a-exp_b.

3. add the obtained mantissa's and normalize the final output.
4. output the result.

3. Subtract the mantissa of first number by the shifted mantissa of the second number.
4. normalize the result and manipulate the final exponent correspondinly.
5. output the result thus obtained.

Case 3 : If exponent of second number is greater than the first ,

▼ If sign of both the number is equal

1. sign of final result will be the sign of second number.
2. Right shift the mantissa part of first number by exp_b-exp_a.
3. add the obtained mantissa's and normalize the final output.
4. output the result.

▼ If the sign of both the number is different

1. sign of final result will be the sign of second number.
2. Right shift the mantissa part of first number by exp_b-exp_a.
3. Subtract the mantissa of second number by the shifted mantissa of the first number.
4. normalize the result and manipulate the final exponent correspondinly.
5. output the result thus obtained.

Floating Point Multiplier:

The general pipeline for the multiplier is given in Fig (4), and the way we have implemented it using behavioral flow of verilog is as follows:

1. Sign of the final result will be XOR of the sign bits of the first and the second number .
2. Set the final exponent to sum of the exponents of the two number subtracted with 127.
3. Next multiply the mantissa of both the numbers with the hidden one , the result will be a 48 bit number.
4. Then normalize the 48 bit number with the following cases:
   ▼ Case 1 : If the MSB bit is 1

   Add 1 to the result exponent and set the normalized fraction bit to the result.

▼ Case 2 : If the 2nd to MSB is 1

Set the final fraction accordingly with the MSB of final fraction being the 2nd bit of the received result.

▼ Case 3 : If the first 1 appears after the 2nd bit the the received result

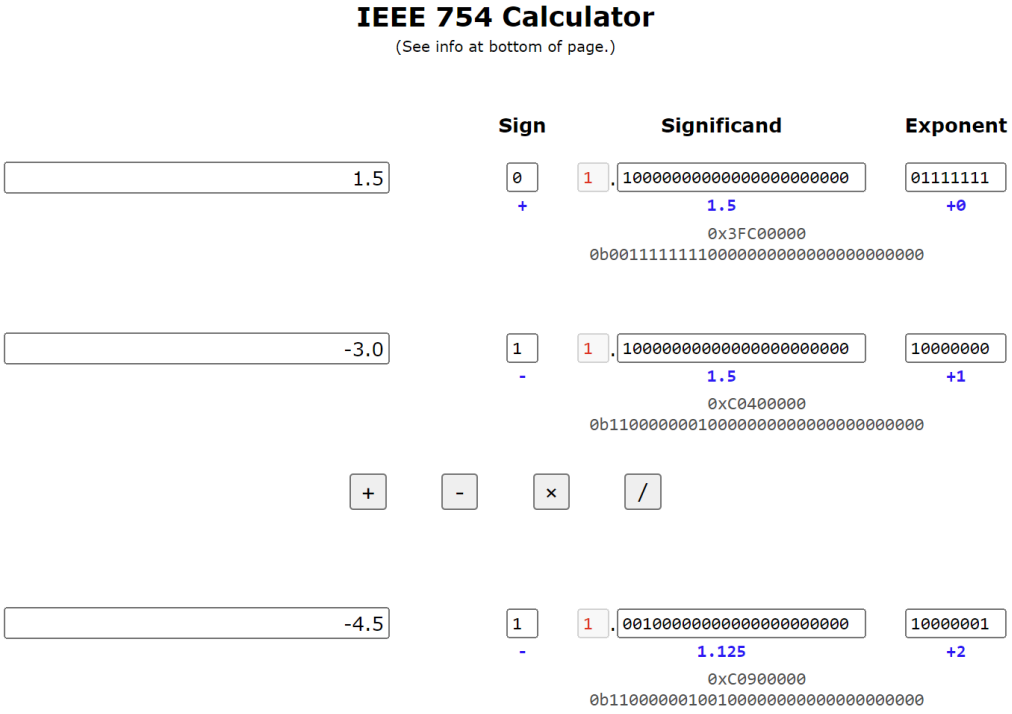
Then we have to normalize the fraction by subtracting 1 from the result exponent.

5. At last check for any overflow or underflow case then output the result after rounding the result.

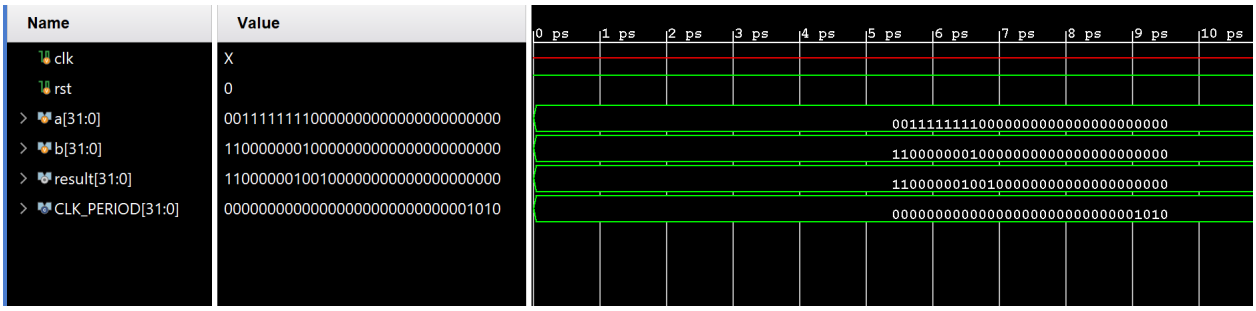
After making these two modules we were only required to combine them to make the Processing Element(PE), and we have implemented this in the complete_module file. In this file we have two input wires(input a and input b) and 3 output wires(output a, output b and output c).

Simulation and Verification :

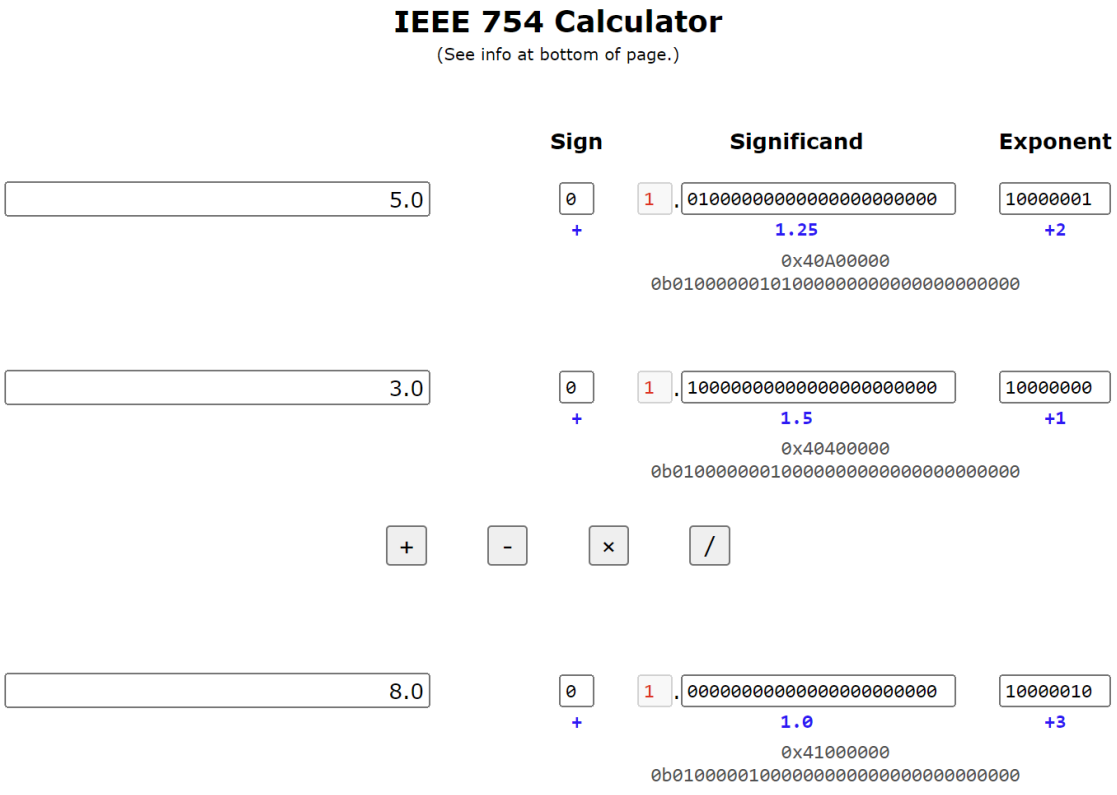
Verification Of the Multiplier:



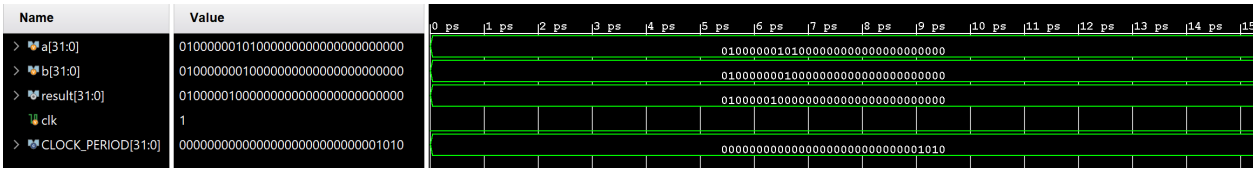
Output of the Verilog Design:



Verification of the Adder:



Output of the Verilog Design:



Conclusion :

We have implemented the adder and multiplier as separate modules for reusability and readability. The modules are designed in VHDL and simulated with Xilinx ISE14.5. As a result, a 32-bit floating-point multiplier and adder is designed and implemented on the Xilinx platform using VHDL code. These are multipliers and adders also manage overflow and underflow circumstances. The final module was also successfully implemented by combining the multiplier and adder modules.

References:

https://digitalsystemdesign.in/floating-point-addition-and-subtraction/

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.iosrjournals.org/iosr-jvlsi/papers/vol5-issue1/Version-1/G05115053.pdf

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.irjet.net/archives/V7/i12/IRJET-V7I12262.pdf