

Design and Simulation of an IEEE Floating-Point Single-Precision (32 bit) ALU

Aaryan Garg (19116001), Divyank Beniwal (19116022),
Pereira Smith (19116047), Sagnik Majumdar (19116063)
under Dr. Bishnu Prasad Das.

ABSTRACT

An Arithmetic Logic-Unit (ALU) is an integral circuital component of a computer processor (CPU) which is used to perform various arithmetic and bitwise logic operations. The aim of this project is to design and simulate a floating-point single-precision (32 bit) ALU which performs arithmetic operations like addition, subtraction, multiplication, and division, and logic operations like NOT, AND, OR and XOR. All of these operations are realized using Verilog HDL, compiled, and simulated using Vivado. The Floating-point numbers are represented in strict accordance with the IEEE standard 754.

INTRODUCTION

Today, the use of floating-point representation has become highly instrumental in the efficient computation of very large or small numbers. The representation of such numbers is not impossible otherwise but can happen only at the cost of power and time consumption, accuracy, memory, and ease of implementation. We call this a floating-point representation because the values of the mantissa bits can ‘float’ along with the decimal point, based on the number’s given

value. We were inspired by the work of D. Jackuline (et al., 2011), Kiseon Cho (et al., 2002), Sukhmeet Kaur (et al., 2013), and thus, we chose an ALU as the project for our ECN-104 course. The importance of floating-point representation can be explained using an example - to represent large values like the distance between two distant galaxies, all 39 decimal values are needed to be placed down to the femtometer resolution, but if the same is represented in terms of light-years, then the computation will involve only 9 significant digits, and hence the remaining 30 digits (which contribute as pure noise) will be excluded from the calculations.

An Arithmetic-Logic Unit (ALU) is a combinational digital electronic circuit and represents the fundamental building block of the Central Processing Unit (CPU) of a computer. It can be used to carry out arithmetic operations like addition, subtraction, multiplication and division, and also 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:** Indicates the sign of the number. 0 is used for denoting positive and 1 for denoting negative numbers.
- **Exponent:** Contains the value of the base power.
- **Mantissa:** Basically 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.

B. Conversion of Decimal to Floating point

The method of conversion can be explained with the help of an example. Let the decimal number be $X(D) = 129.85$. Before converting it into a floating-point format, it is converted into its binary form, i.e $X(B) = 10000001.110111$

After the conversion, the radix point is shifted towards the left, until there remains only one bit to the left of the radix point, and that bit must be 1. Hence the final binary number is then represented as $X(F) = 1.000000111011100000000000$.

The number after the radix point is called the mantissa (23 bits) and the whole number is called the significand (24 bits, including the hidden bit). Now, we define a number 'x' which is basically the number of times the radix point is shifted. This value is then added to 127 to get the exponent value. In this example, x takes the value of 7, so the value of exponent becomes equal to $127+7= 134$, which when represented in the binary form, is 10000110. Also, the signed bit i.e MSB is 0 (the number being positive), and finally, the result is expressed in the 32-bit format which comprises the sign, exponent, and mantissa: 01000011000000011101110000000000.

C. Operations

a. Addition-Subtraction:

While adding or subtracting two floating-point numbers expressed in the Single precision IEEE-754 format, two possible cases may arise:

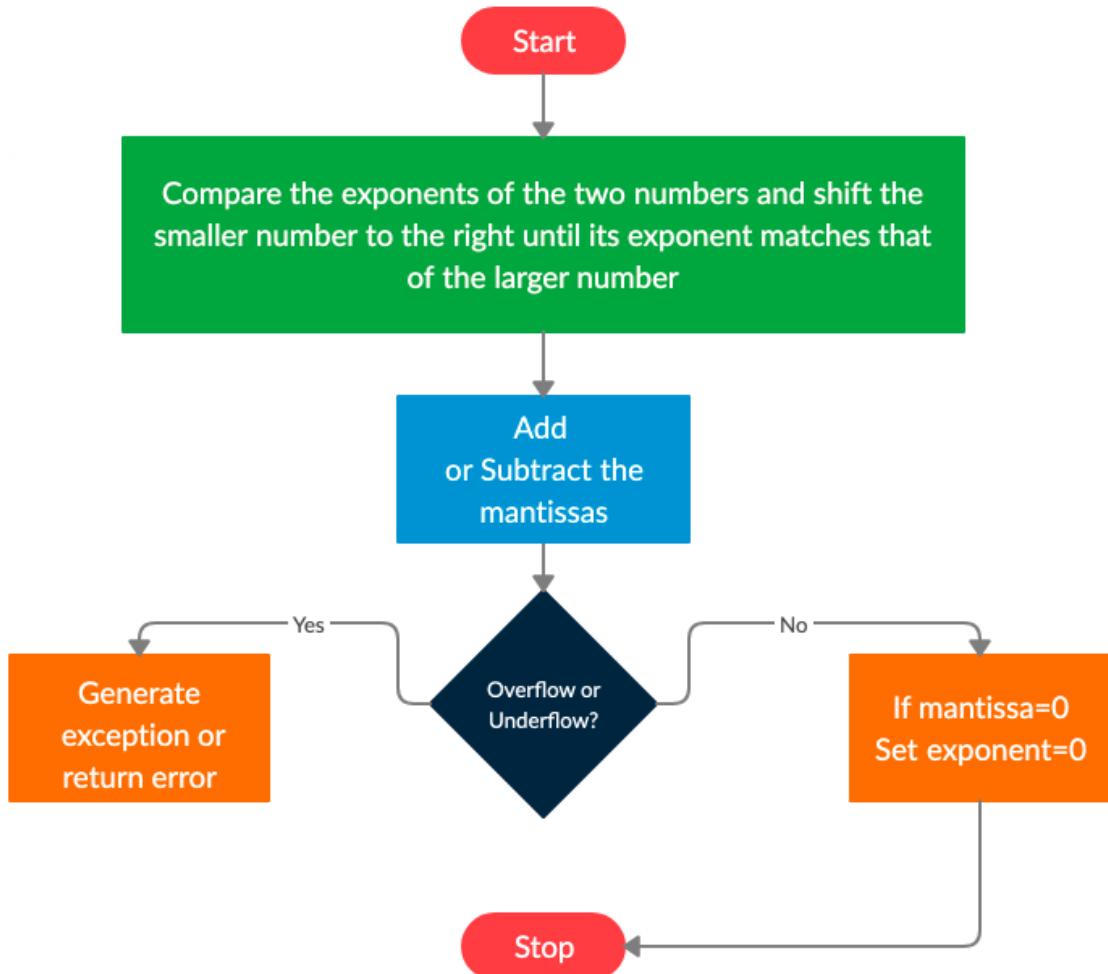
Case 1:

When both the numbers are of the same sign i.e. when both numbers are either +ve or -ve. In this case, the MSB of both numbers will either be 1 or 0.

Case 2:

When both numbers have different signs. In this case, MSB of one number will be 1 and that of the other will be 0. In this case, take the 2's complement of any one of the numbers and then add the numbers.

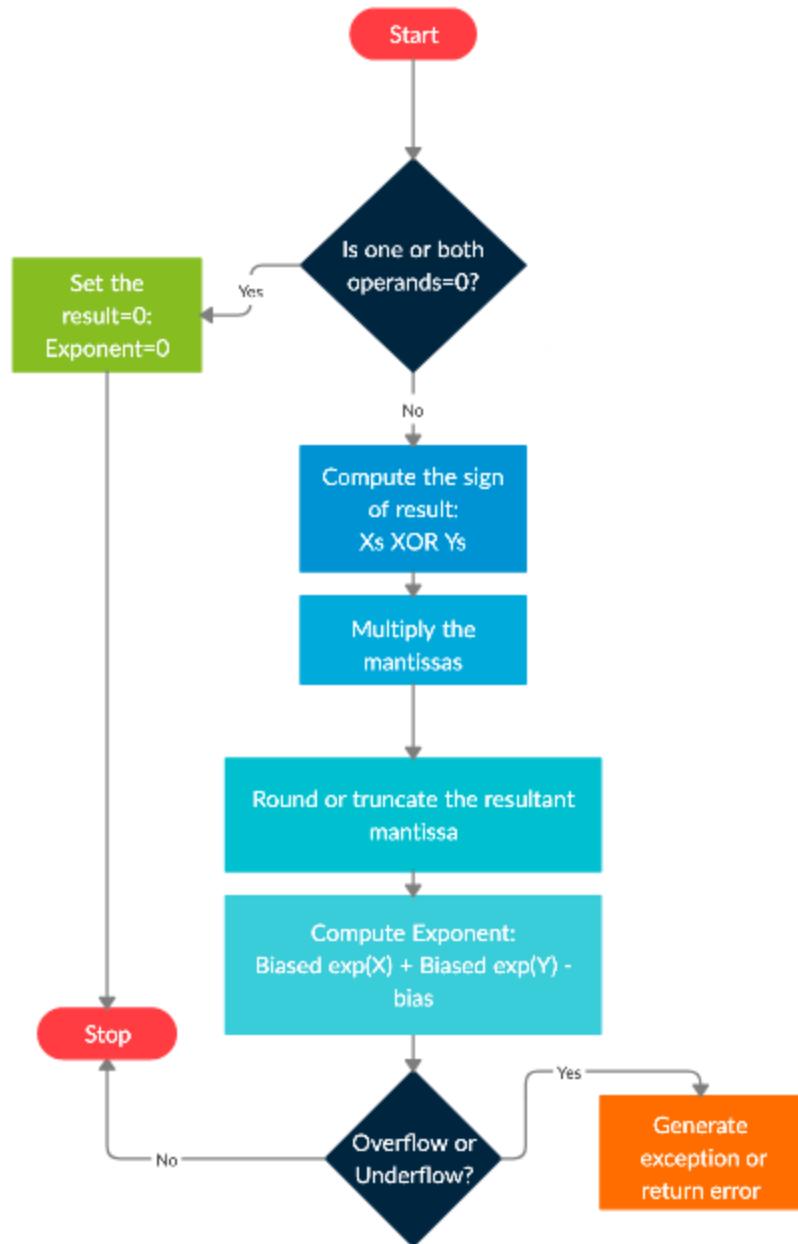
This is a simple flowchart depicting a simplified version of the algorithm used for the addition/ subtraction operations :



b. Multiplication:

Take the two normalized operands and multiply the significands. Then add the exponents and determine the sign. Now, normalize the mantissa and update the exponent. Find the exception flags and also the special values for overflow and underflow.

This is a simple flowchart depicting a simplified version of the algorithm used for the multiplication operation :

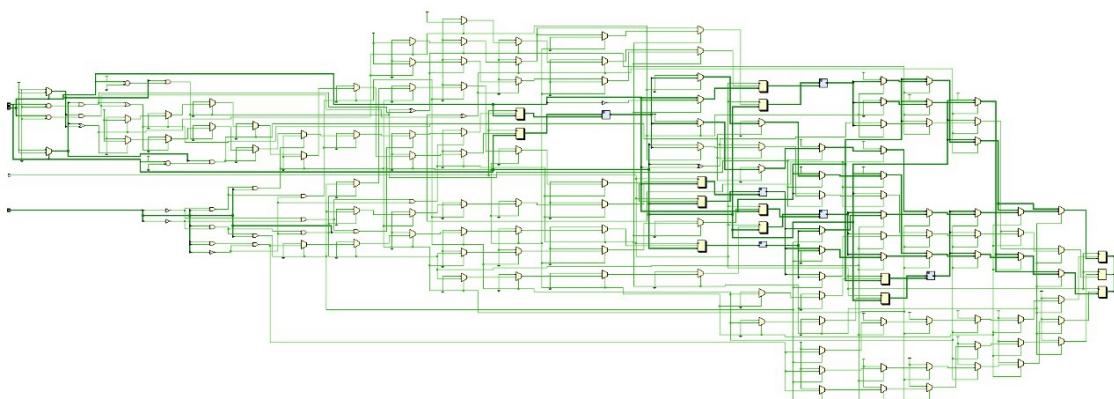


CHALLENGES FACED DURING IMPLEMENTATION

1. A number of Internet-related issues were faced by all of the members of the group, mostly due to unprecedeted network fluctuations and data limitations, and the Internet being a resource of utmost importance for this project, we were not able to set up Vivado on our systems and the group had to face a lot of challenges during the learning and simulation phases.
2. Choosing a proper working data-propagation model was another challenge.
3. 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.
4. 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.

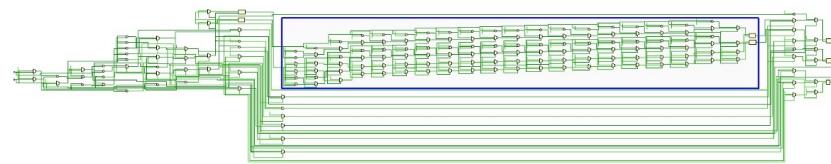
ARCHITECTURE OF THE DESIGN

A. Complete ALU:



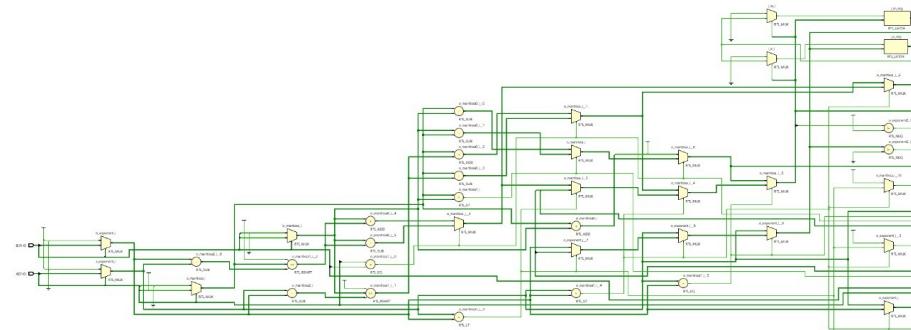
This is the Block Diagram of the complete ALU

B. Adder-Subtractor:

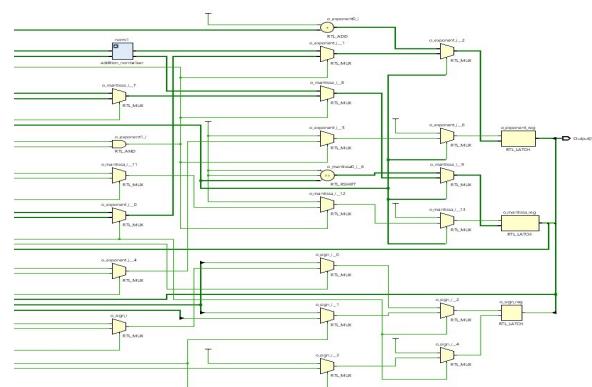


This is the block diagram of the Adder-Subtractor Module.

(Blue Part is the Addition normalizer)

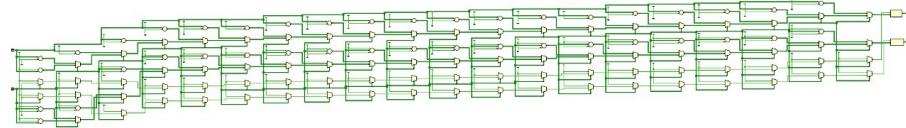


Left Close-up

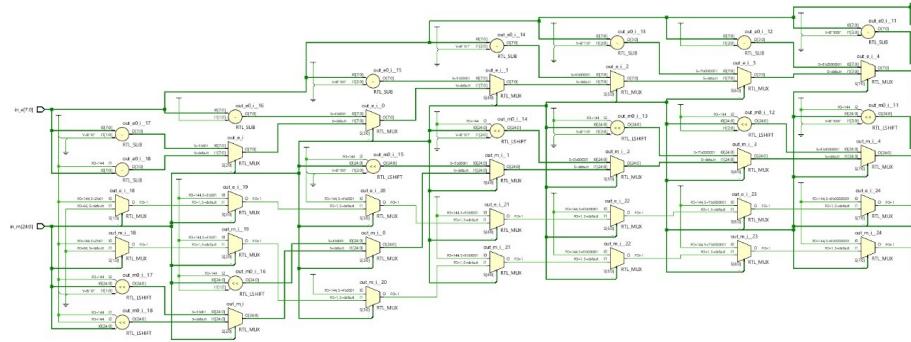


Right Close-up

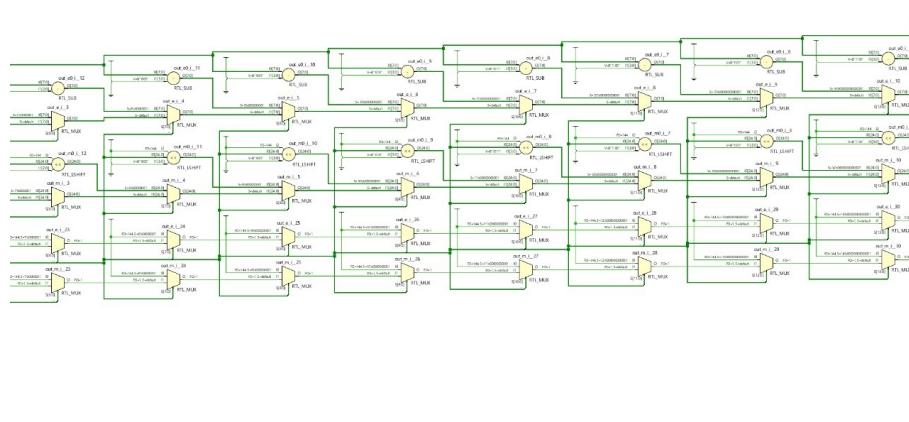
C. Addition Normalizer:



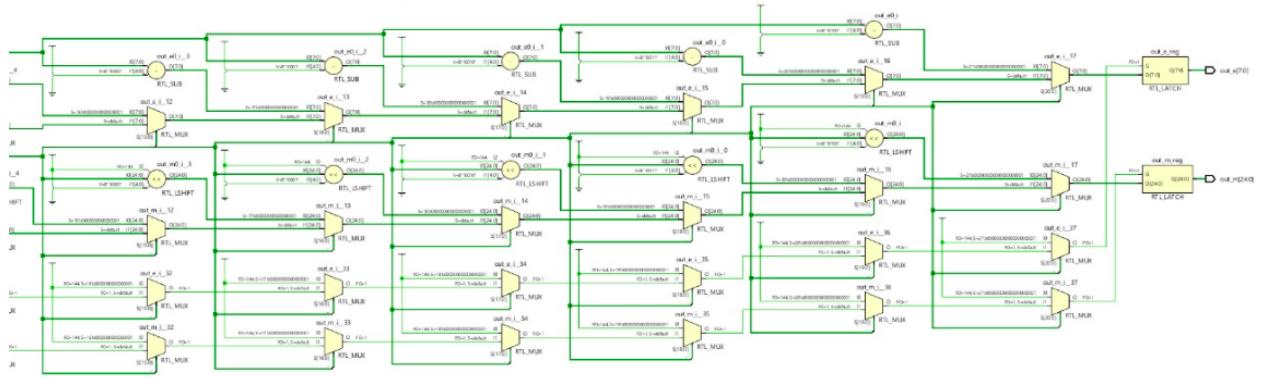
This is the block diagram of the Adder-Subtractor Module.



Left Close-up



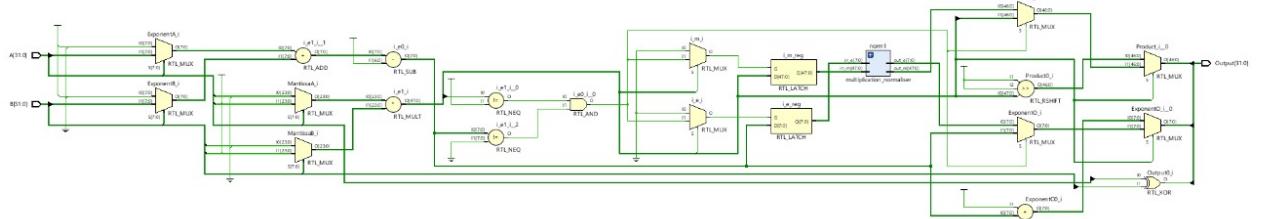
Centre Close-up



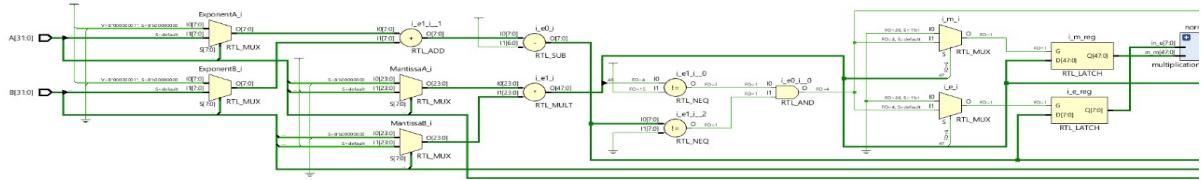
Right Close-up

D. Multiplier:

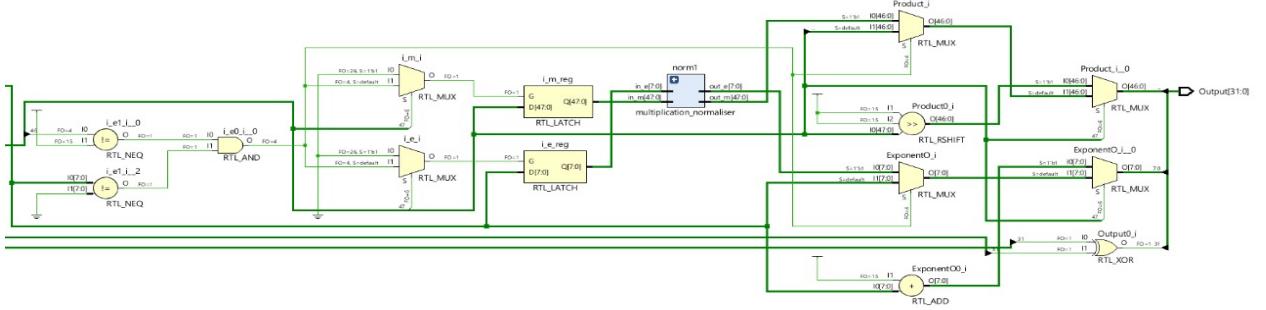
Multiplies the mantissa of A with the mantissa of B. The speed of the operation is determined by the 24*24 bit multiplier which is used to calculate the resulting 48-bit mantissa. Since this is the most intensive operation, it is the slowest hence, the rate-determining step. The resulting sign of the numbers is determined by a simple XOR operation.



This is the block diagram of the Multiplier

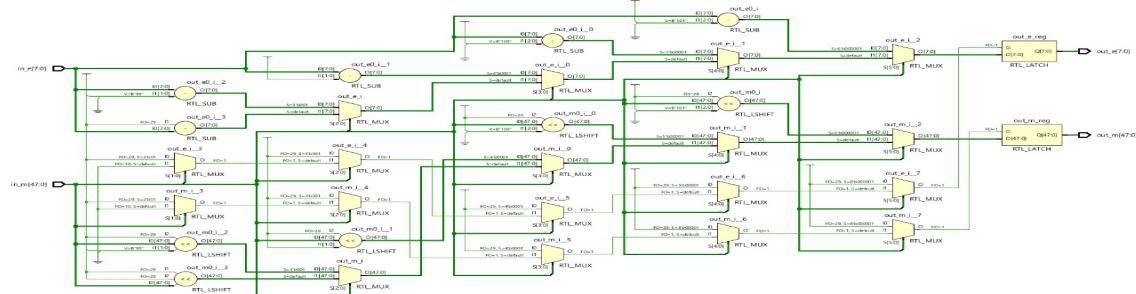


Left Close-up



Right Close-up

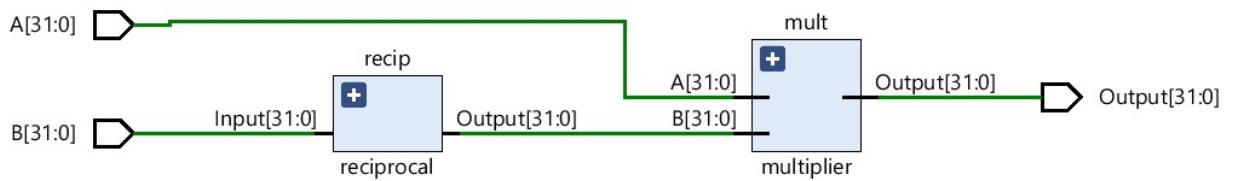
E. Multiplication-Normalizer:



This is the block diagram of the Multiplication Normalizer

F. Divider:

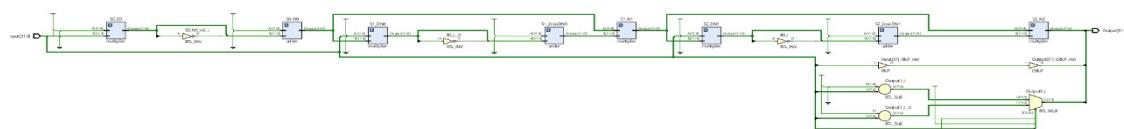
We first shift the dividend by 1 towards the left. Then, we subtract the divisor. We don't restore if the carry is 1. If the carry is 0 i.e. answer is negative then we restore by adding back to the divisor. We then place the carry as the Least Significant Bit of the intermediate answer. We do this procedure 24 times.



This is the block diagram of the Divider

G. Reciprocal:

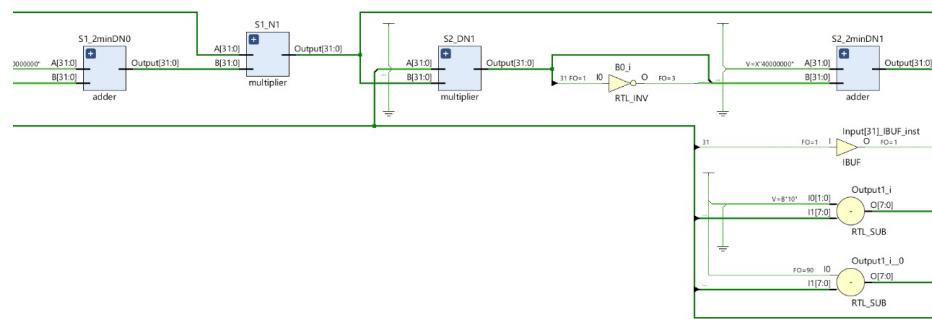
This module was used to calculate the reciprocal of B. Together with the Multiplier Module, it gives the Divider module. After each operation, its output had to be checked and handled for overflow and underflow.



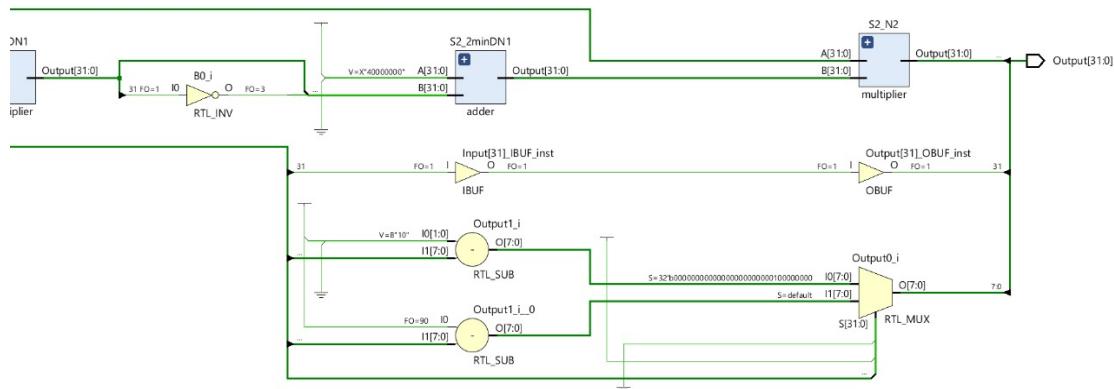
This is the block diagram of the Reciprocal Module



Left Close-up



Centre Close-up



Right Close-up

The entire design is implemented in the following steps:

- Conversion of the Floating Point Number into an IEEE-754 floating-point format.
- Shifting the bits of the operand with a lower exponent to match that of the other operand.
- Performing the operation.
- Normalize the output obtained.
- Detecting and handling the exceptions encountered.

Separating: We separate the sign bit, the exponent bits, and the mantissa.

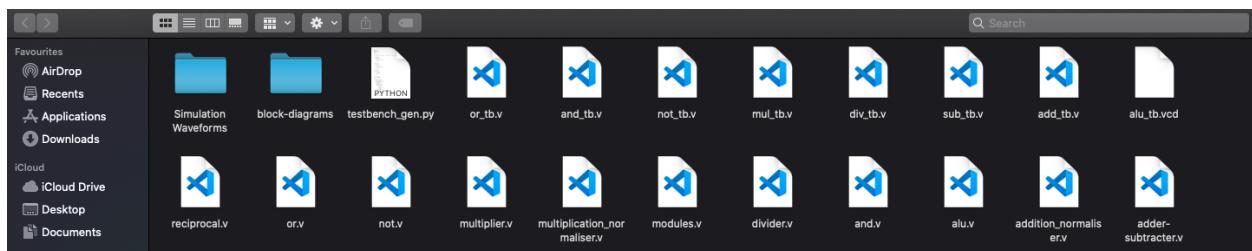
Swapping: We have written our code assuming that A always has the larger exponent. So, we have to swap A and B if B turns out to have the larger exponent. This reduces the complexity of the code and makes it easier to understand.

Exponent-Matching: We calculate the difference in exponents of A and B. Then, we shift B's mantissa by the difference in exponents. We then put three extra bits- The Sticky Bit, The Rounding Bit, and the Guard bit to both the mantissas so as to increase the accuracy of the operation. We calculate the Sticky Bit by performing "OR" on any bits dropped during the bit-shifting procedure.

Normalize: We normalize the resulting mantissa by detection of the leading one by the locally one-dimensional method. We adjust the resulting exponent accordingly. Then, we give the result in the IEEE-754 format.

CODE STRUCTURE:

We have created separate modules for each type of operation and normalization. Then, we have included them in the main file "alu.v" via "modules.v". As for the testbenches, we have made a python file called "testbench_gen.py" which when run generates a testbench file in Verilog for the specified operation and the no. of test cases.



The Folder Structure

Here are some snippets of the code:

```
module adder(A, B, Output);
    // Inputs
    input [31:0] A, B;

    // Outputs
    output [31:0] Output;

    // Wires
    wire [31:0] Output;
    wire [7:0] o_e;
    wire [24:0] o_m;

    // Registers
    reg a_sign;
    reg [7:0] a_exponent;
    reg [23:0] a_mantissa;
    reg b_sign;
    reg [7:0] b_exponent;
    reg [23:0] b_mantissa;
    reg o_sign;
    reg [7:0] o_exponent;
    reg [24:0] o_mantissa;
    reg [7:0] diff;
    reg [23:0] tmp_mantissa;
    reg [7:0] tmp_exponent;
    reg [7:0] i_e;
    reg [24:0] i_m;

    // Main
    addition_normaliser norm1
    (
        .in_e(i_e),
        .in_m(i_m),
        .out_e(o_e),
        .out_m(o_m)
    );

    assign Output[31] = o_sign;
    assign Output[30:23] = o_exponent;
    assign Output[22:0] = o_mantissa[22:0];

    always @ ( * ) begin
        a_sign = A[31];
        if(A[30:23] == 0) begin
            a_exponent = 8'b00000001;
            a_mantissa = {1'b0, A[22:0]};
        end else begin
            a_exponent = A[30:23];
            a_mantissa = {1'b1, A[22:0]};
        end
        b_sign = B[31];
        if(B[30:23] == 0) begin
            b_exponent = 8'b00000001;
            b_mantissa = {1'b0, B[22:0]};
        end
    end

```

```

    b_mantissa = {1'b0, B[22:0]};
end else begin
    b_exponent = B[30:23];
    b_mantissa = {1'b1, B[22:0]};
end
if (a_exponent == b_exponent) begin // Equal exponents
    o_exponent = a_exponent;
    if (a_sign == b_sign) begin // Equal signs = add
        o_mantissa = a_mantissa + b_mantissa;
        //Signify to shift
        o_mantissa[24] = 1;
        o_sign = a_sign;
    end else begin // Opposite signs = subtract
        if(a_mantissa > b_mantissa) begin
            o_mantissa = a_mantissa - b_mantissa;
            o_sign = a_sign;
        end else begin
            o_mantissa = b_mantissa - a_mantissa;
            o_sign = b_sign;
        end
    end
end
end else begin //Unequal exponents
    if (a_exponent > b_exponent) begin // A is bigger
        o_exponent = a_exponent;
        o_sign = a_sign;
        diff = a_exponent - b_exponent;
        tmp_mantissa = b_mantissa >> diff;
        if (a_sign == b_sign)
            o_mantissa = a_mantissa + tmp_mantissa;
        else
            o_mantissa = a_mantissa - tmp_mantissa;
    end else if (a_exponent < b_exponent) begin // B is bigger
        o_exponent = b_exponent;
        o_sign = b_sign;
        diff = b_exponent - a_exponent;
        tmp_mantissa = a_mantissa >> diff;
        if (a_sign == b_sign) begin
            o_mantissa = b_mantissa + tmp_mantissa;
        end else begin
            o_mantissa = b_mantissa - tmp_mantissa;
        end
    end
end
if(o_mantissa[24] == 1) begin
    o_exponent = o_exponent + 1;
    o_mantissa = o_mantissa >> 1;
end else if((o_mantissa[23] != 1) && (o_exponent != 0)) begin
    i_e = o_exponent;
    i_m = o_mantissa;
    o_exponent = o_e;
    o_mantissa = o_m;
end
end
endmodule

```

These snippets are from the Adder-Subtractor Module

Similar to this module, we have made many modules, each for different operation namely –

- adder-subtractor.v
- addition_normaliser.v
- multiplier.v
- multiplication_normaliser.v
- divider.v
- and.v

- OR.V
- reciprocal.v

Similarly, we have a generated testbench file in verilog for each operation. These include **add_tb.v**, **sub_tb.v**, **mul_tb.v**, **div_tb.v**, **and_tb.v**, **or_tb.v**, and **not_tb.v**

```
timescale 1 ns/1 ps      You, 5 hours ago • Changed test bench module name
`include "alu.v"

module add_tb ();
    reg clock;
    reg [31:0] a, b;
    reg [2:0] op;
    reg [31:0] correct;

    wire [31:0] out;
    wire [49:0] pro;

    alu U1 (
        .clk(clock),
        .A(a),
        .B(b),
        .OpCode(op),
        .O(out)
    );
    /* create a 10Mhz clock */
    always
    #100 clock = ~clock; // every 100 nanoseconds invert
    initial begin
        $dumpfile("alu_tb.vcd");
        $dumpvars(0,clock, a, b, op, out);
        clock = 0;
    end
    op = 3'b000;

    /* Display the operation */
    $display ("Opcode: 000, Operation: ADD");
    /* Test Cases*/
    a = 32'b000101000010100011000110010111011;
    b = 32'b10101011100101001000111101000001;
    correct = 32'b101010111001000101110110000101000;
    #400 //1.1623247e-14 * -1.0555793e-12 = -1.043947e-12
    begin
        $display ("A      : %b %b", a[31], a[30:23], a[22:0]);
        $display ("B      : %b %b", b[31], b[30:23], b[22:0]);
        $display ("Output : %b %b", correct[31], correct[30:23], correct[22:0]);
        $display ("Correct: %b %b", correct[31], correct[30:23], correct[22:0]); $display();
    end
    a = 32'b0010011011001010101010010111010;
    b = 32'b0101101001001111010110110000011;
    correct = 32'b0101101001001111010110110000011;
    #400 //1.4062502e-15 * 1.4614093e+16 = 1.4614093e+16
    begin
        $display ("A      : %b %b", a[31], a[30:23], a[22:0]);
        $display ("B      : %b %b", b[31], b[30:23], b[22:0]);
        $display ("Output : %b %b", correct[31], correct[30:23], correct[22:0]);
        $display ("Correct: %b %b", correct[31], correct[30:23], correct[22:0]); $display();
    end
end
```

This is a snippet from the addition operation test bench file - add_tb.v

All the code for the project can be found here: <https://github.com/Aaryan-Garg/ALU>

SIMULATION RESULTS

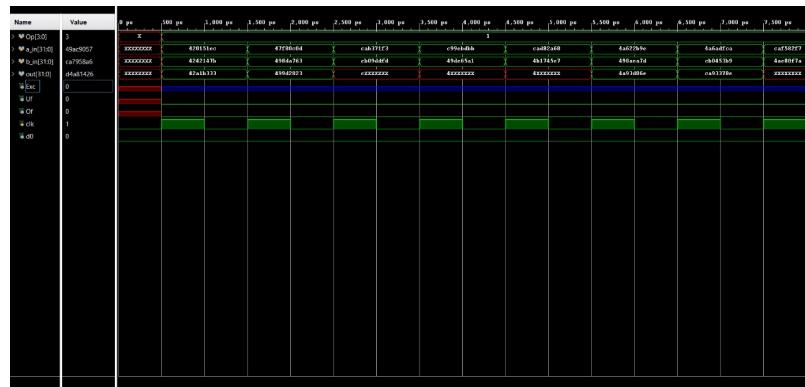
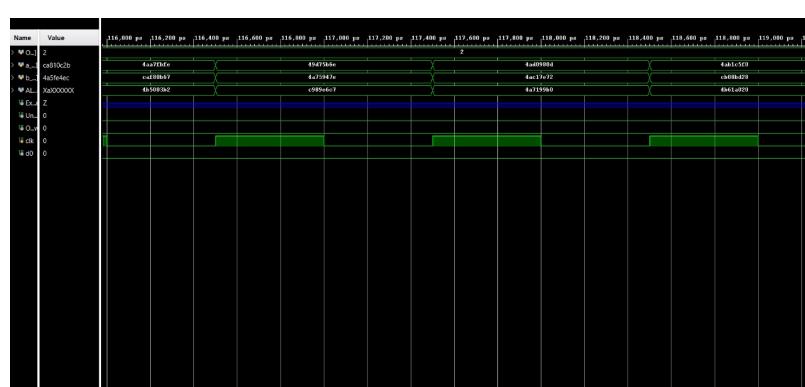
A. Accuracy:

We ran the test bench for 1000 cases for each operation. We achieved quite high accuracies on operations like Addition, Subtraction, Bitwise Logical AND, Bitwise Logical OR, Bitwise Logical NOT. However, due to the problem of Overflow and Underflow, our Multiplier and Divider Modules did not do so well. Earlier this accuracy was around 30-40% but we managed to handle quite a lot of exceptions using scaling and thus, the accuracy increased by more than two-fold.

Here are the results of the various tests conducted using the testbench files:

Operation Name (Code)	Total no. of tests failed	Total no. of tests passed	Total no. of tests failed	Accuracy (in %)
Addition (000)	1000	998	2	99.8%
Subtraction (001)	1000	997	3	99.7%
Multiplication (011)	1000	752	248	75.2%
Division (010)	1000	766	234	76.6%
Bitwise-AND (100)	1000	1000	0	100%
Bitwise-OR (101)	1000	1000	0	100%
Bitwise-NOT (110)	1000	1000	0	100%

B. Waveforms:

Operation Name (Code)	Waveform																						
Addition (000)	 <p>Timing diagram for the Addition operation (000). The diagram shows the state of various signals over time. The output Z is high whenever both A and B are high, indicating a carry or sum bit.</p> <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>• Op[3:0]</td> <td>3</td> </tr> <tr> <td>• A[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• B[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• C[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• ctrl[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• clk</td> <td>0</td> </tr> <tr> <td>• uf</td> <td>0</td> </tr> <tr> <td>• of</td> <td>0</td> </tr> <tr> <td>• ok</td> <td>1</td> </tr> <tr> <td>• do</td> <td>0</td> </tr> </tbody> </table>	Name	Value	• Op[3:0]	3	• A[3:0]	0x0000	• B[3:0]	0x0000	• C[3:0]	0x0000	• ctrl[3:0]	0x0000	• clk	0	• uf	0	• of	0	• ok	1	• do	0
Name	Value																						
• Op[3:0]	3																						
• A[3:0]	0x0000																						
• B[3:0]	0x0000																						
• C[3:0]	0x0000																						
• ctrl[3:0]	0x0000																						
• clk	0																						
• uf	0																						
• of	0																						
• ok	1																						
• do	0																						
Subtraction (001)	 <p>Timing diagram for the Subtraction operation (001). The diagram shows the state of various signals over time. The output Z is high whenever A is greater than B, indicating a borrow or result bit.</p> <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>• Op[3:0]</td> <td>2</td> </tr> <tr> <td>• A[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• B[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• C[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• ctrl[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• clk</td> <td>0</td> </tr> <tr> <td>• uf</td> <td>0</td> </tr> <tr> <td>• of</td> <td>0</td> </tr> <tr> <td>• ok</td> <td>0</td> </tr> <tr> <td>• do</td> <td>0</td> </tr> </tbody> </table>	Name	Value	• Op[3:0]	2	• A[3:0]	0x0000	• B[3:0]	0x0000	• C[3:0]	0x0000	• ctrl[3:0]	0x0000	• clk	0	• uf	0	• of	0	• ok	0	• do	0
Name	Value																						
• Op[3:0]	2																						
• A[3:0]	0x0000																						
• B[3:0]	0x0000																						
• C[3:0]	0x0000																						
• ctrl[3:0]	0x0000																						
• clk	0																						
• uf	0																						
• of	0																						
• ok	0																						
• do	0																						
Multiplication (011)	 <p>Timing diagram for the Multiplication operation (011). The diagram shows the state of various signals over time. The output Z is high whenever both A and B are non-zero, indicating a product bit.</p> <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>• Op[3:0]</td> <td>3</td> </tr> <tr> <td>• A[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• B[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• C[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• ctrl[3:0]</td> <td>0x0000</td> </tr> <tr> <td>• clk</td> <td>0</td> </tr> <tr> <td>• uf</td> <td>0</td> </tr> <tr> <td>• of</td> <td>0</td> </tr> <tr> <td>• ok</td> <td>1</td> </tr> <tr> <td>• do</td> <td>0</td> </tr> </tbody> </table>	Name	Value	• Op[3:0]	3	• A[3:0]	0x0000	• B[3:0]	0x0000	• C[3:0]	0x0000	• ctrl[3:0]	0x0000	• clk	0	• uf	0	• of	0	• ok	1	• do	0
Name	Value																						
• Op[3:0]	3																						
• A[3:0]	0x0000																						
• B[3:0]	0x0000																						
• C[3:0]	0x0000																						
• ctrl[3:0]	0x0000																						
• clk	0																						
• uf	0																						
• of	0																						
• ok	1																						
• do	0																						



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

- [1] Aarthy.M and Dave omkar.R," Asic implementation of 32 and 64-bit floating-point ALU using pipelining", International journal of computer applications, May 2014.
- [2] Kavita katole, Ashwin Shinde," Design & simulation of 32-bit floating-point ALU "International Journal of Advances in Science Engineering and Technology, April 2014.
- [3] Sukhmeet Kaur, Suman and Manpreet Singh Manna, "Implementation of Modified Booth Algorithm (Radix4) and its Comparison with Booth Algorithm (Radix-2) " Research India Publications,2013.
- [4] Geetanjali and Nishant Tripathi, "VHDL Implementation of 32-Bit Arithmetic Logic Unit (ALU)", International Journal of Computer Science and Communication Engineering, 2012.