Design of an arithmetical-logical floating-point unit

(The addition and multiplication)

Student: Moga Diana Marcela

Structure of computer Systems Project

Technical University of Cluj-Napoca

Contents

Introduction

1.1 Context

The objectives of this module are to discuss the need for floating point numbers, the standard representation used for floating point numbers and discuss how the various floating point arithmetic operations of addition and multiplication are carried out.

The main advantage of floating-point representation is that it can support more values than fixed-point and integer representations. Summation, Subtraction, multiplication and division are arithmetic functions in these calculations. In this floating-point unit, input must be provided in IEEE-754 format, which is 32 single precision floating point values. The application of This arithmetic unit is located in the math coprocessor.

1.2 Objectives

The unit will be designed in VHDL and included in a Xilinx Vivado Project. A testbench is also provided for simulation purposes. The focus of this paper is on planned to follow a Bottom to up apprach while designing this ALU. Firstly, we implemented simple logic blocks such as Comparator, Adder, Shifter, etc.

Bibliographic Research

2.1 What is ALU?

An **ALU** (Arithmetic Logic Unit) is a fundamental component of a computer's central processing unit (CPU). It performs arithmetic and logical operations, which are essential for computer processing. The ALU is responsible for executing operations like addition, subtraction, multiplication, division, and bitwise operations (AND, OR, XOR, NOT, etc.).

2.2 Methods to transform numbers from decimalism to binary.

To translate a number from decimal to binary, a straightforward method is employed. Take the example of decimal number 18. It can be represented as the sum of powers of 2, which is 16 + 2. In binary, 16 is represented as 10000, and 2 is represented as 00010. Combining there, 10010 is the result of the binary representation of 18. For negative numbers, a two’s complement approach is used.

2.3 Floating-point numbers and operations

Representation:

When you have to represent very small or very large numbers, a fixed point representation will not do. The accuracy will be lost. Therefore, you will have to look at floating-point representations, where the binary point is assumed to be floating. When you consider a decimal number 12.34 * 107, this can also be treated as 0.1234 * 109, where 0.1234 is the fixed-point mantissa. The other part represents the exponent value, and indicates that the actual position of the binary point is 9 positions to the right (left) of the indicated binary point in the fraction. Since the binary point can be moved to any position and the exponent value adjusted appropriately, it is called a floating-point representation.

The IEEE (Institute of Electrical and Electronics Engineers) has produced a standard for floating point arithmetic. This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are to be represented, as well as how arithmetic should be carried out on them. The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right.

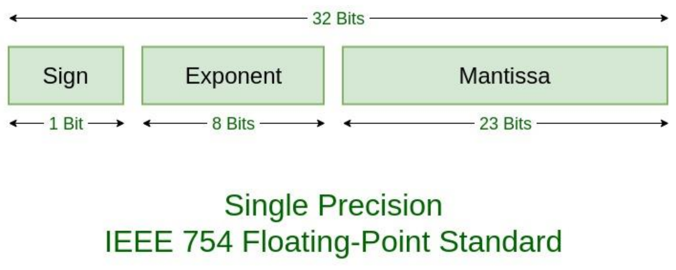


Figure 2.3 1 IEEE Standard 754 FLoating Point Numbers

2.4 Operations on floating point numbers

Floating Point Addition refers to the process of adding floating-point numbers by extracting exponents and fractions, aligning the numbers, adding the mantissas, normalizing the result, and rounding if necessary.

When **multiplying floating** point number, you multiply mantissa and then shift the result back down to the right size, and add the exponents together, followed by some adjustments to deal with overflows, not-a-number, rounding.

Analysis

3.1 Use-case diagram

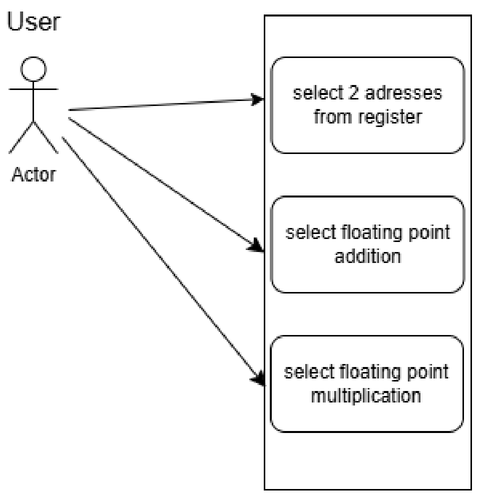

User
Actor
select 2 adresses from register
select floating point addition
select floating point multiplication

Figure 3.1 1 Use-case diagram

Use-case: select operation

Primary actor: user

Main scenario:

1. The user select 2 addresses from the register where are the numbers they want to perfom the operations
2. The user select the operation
3. The user clicks on button
4. The ALU performs the operation and displays the result

3.2 Adding floating point numbers

For the purpose of making the explanation easier, we will denote the 2 operands with X1, X2 and the result with S. Firstly: X1 and X2 can only be added if the exponents are the same i.e E1=E2. If this is not the case, we will see later in the algorithm that we will need some shifts. The steps are:

- Determine Absolute Values: First, ensure that the absolute value of X1 is greater than that of X2. If not, swap the two values so that |X1|>|X2|.

- Set Initial Exponent: The initial exponent (before any adjustments) should be taken as the larger exponent, specifically E3=E1, since X1 has the larger absolute value.

- Calculate Exponent Difference: Find the difference between the exponents: Exp_diff=E1−E2.
- Align Mantissa Exponents: Adjust the mantissa of X2 (denoted as M2) by left-shifting it according to the exponent difference calculated in the previous step. This adjustment will make the exponents of both X1 and X2 equal.
- Compute Mantissa Sum: Based on the sign bits of X1 (denoted as S1) and X2 (denoted as S2), calculate the result of the mantissas. If both signs are the same (S1=S2), add the mantissas.
- Normalize the Resulting Mantissa: If necessary, normalize the resultant mantissa (denoted as M3) to ensure it is in the form of 1.m3. Additionally, adjust the initial exponent E3 based on any changes made during the normalization of the mantissa.

We will take an example for more clarity:

A = 9.75

B = 0.5625

Then the numbers in binary are:

X1=0 10000010 0011100…

X2=0 01111110 0010000…

1) Abs (A) > Abs (B)? Yes.

2) Result of Initial exponent E3 = E1 = 10000010 = 130(10)

3) E1 - E2 = (10000010 - 01111110) => (130-126)=4

4) Shift the mantissa M2 by (E1-E2) so that the exponents are same for both numbers.
MX2=1. 0010000…

=00001. 001000…

=0. 0001001000…

5) Sign bits of both are equal? Yes. Add the mantissa's

1.001110000000…..(MX1)

0.000100100000…..(aligned mantissa of X2)

1.010010100000…..(mantissa of X3)

6) Normalization needed? No, (if Normalization was required for M3 then the initial exponent result E3=E1 should be adjusted accordingly)

7) Convert the result back into decimals which is 10.3125

3.3 Multiply floating point numbers

Floating point multiplication can be done by multiplying the significand of two floating point numbers and adding the exponents,then subtract the Bias from added exponent result (E1 + E2 – Bias).

The exponent is stored in a "biased" form. For single-precision floating-point numbers (32-bit), the bias is **127=**001111111. This means that the actual exponent (E) is offset by 127 when stored.

Sign is obtained by xor-ing the MSB of two numbers,then normalize the result.Rounding of result is done to fit in the available bits and if desired finally check the underflow/overflow occurrence.

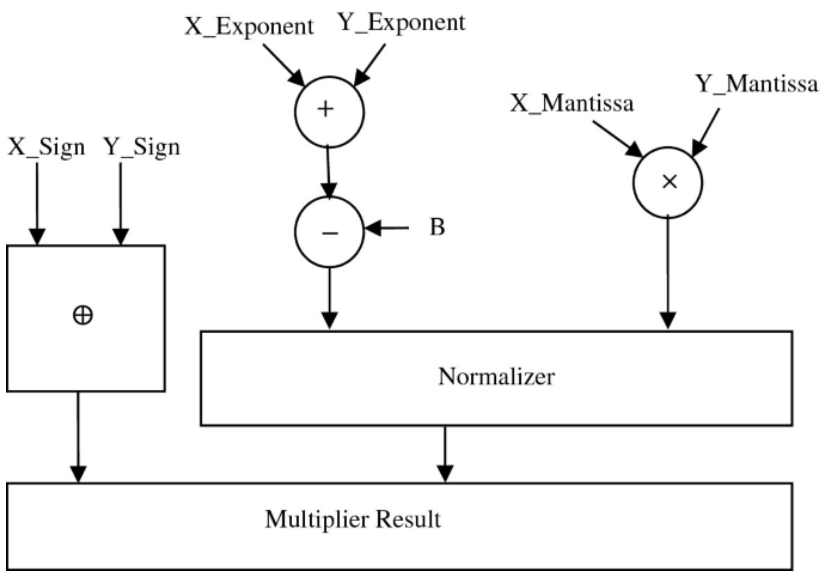


Figure 3.3 1 Floating Point Multiplier | SpringerLink

Design

4.1 Overview perspective

The Floating Point ALU would receive a control signal to select between addition and multiplication, forwarding the inputs to the respective operation.

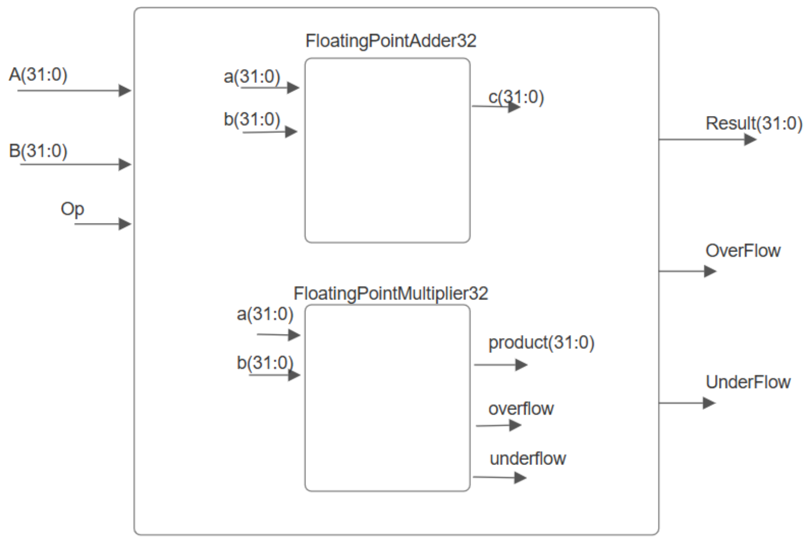


Figure 4 1 Block diagram of Floating Point ALU(addition and multiplication)

 Operation Select (op): The op signal controls which operation to perform. When op = '0', the ALU performs addition, and when op = '1', it performs multiplication.
 Adder and Multiplier Instantiations: The adder and multiplier components are instantiated using FloatingPointAdder32 and FloatingPointMultiplier32 designs.
 Output Selection Process: The process block chooses the output, overflow, and underflow signals based on the value of op. This block monitors op and the outputs of both components, updating the ALU's output signals accordingly.

4.2 Design of floating point adder

The implementation is structured in a modular fashion, allowing for clarity, reusability, and easier debugging. Each module addresses a specific functionality in the floating-point addition process.

The implementation is divided into the following components, each responsible for specific tasks in the addition process:

1. InputExtractor

The component uses bit slicing to retrieve the respective fields from the input numbers and concatenates "1" at the beginning of the mantissa to account for normalized values.

2. ExponentComparator

A simple comparison determines the larger and smaller exponent. The difference is calculated by converting the unsigned exponent values to integers.

3. MantissaAligner

The smaller mantissa is right-shifted based on the previously calculated exponent difference to align both mantissas.

4. SignChooser

The SignChooser component is responsible for determining the sign of the result in a floating-point addition or subtraction operation. It compares the signs of two input floating-point numbers and the aligned mantissas.

5. MantissaAdder

The MantissaAdder component performs a bit-by-bit addition of two 24-bit aligned mantissas, producing a 25-bit result. It handles carry propagation using a ripple carry adder, ensuring that the sum is correctly computed, including handling overflow from the most significant bit. The result is a 25-bit mantissa, which is necessary to ensure precision in floating-point arithmetic when adding two numbers.

6. Normalizer

The Normalizer component ensures that the sum of two floating-point numbers, represented by their mantissas and exponents, is correctly normalized. It adjusts the mantissa and the exponent to conform to IEEE 754 standards, where the mantissa has a leading 1, and the exponent is adjusted to reflect any shifts.

7. OutputPacker

The outputs are concatenated to form the 32-bit result, which is output as a single vector.

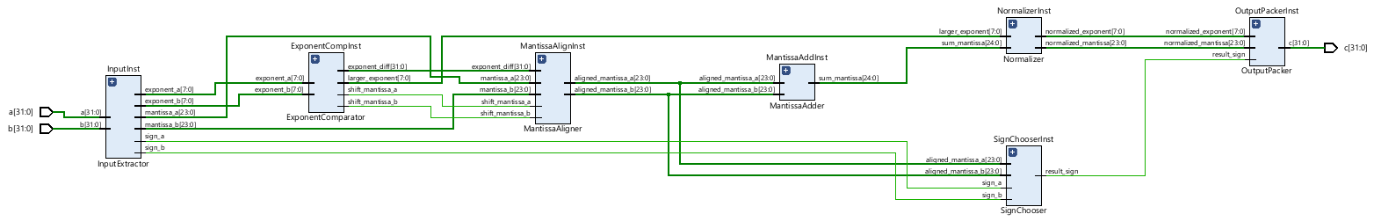


Figure 4.2 1 Block design of FloatingPointAdder32

4.3 Design of floating point multiplier

The implementation is modular, divided into the following components, each responsible for specific tasks in the multiplication process:

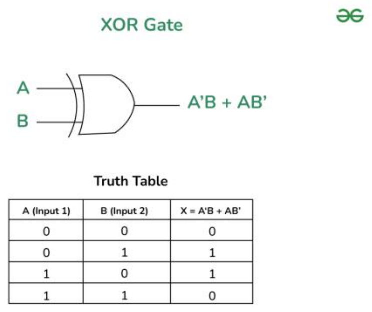
1. InputExtractor

The component uses bit slicing to retrieve the respective fields from the input numbers and concatenates "1" at the beginning of the mantissa to account for normalized values.

2. Sign Bit Calculation

Multiplying two number's result is a negative sign if one of the multiplied numbers is of a negative value.

By the aid of a truth table we find that this can be obtained by XORing the sign of two inputs.



A (Input 1)	B (Input 2)	X = A'B + AB'
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.3 1 XOR Gate- Truth Table

3. Unsigned Adder (for Exponent Addition)

The unsigned adder adds the exponents of two inputs, then subtracts a Bias of 127 to produce an intermediate exponent. This process employs an 8-bit ripple carry adder, which consists of cascaded full adders and a half adder. Each full adder has three inputs (A, B, Ci) and two outputs (S, C), with the carry-out (C) from each adder cascading to the next. The result is an 8-bit sum (S7 to S0) and a carry bit (C,7), forming a 9-bit output (S8 to S0). The Bias is subtracted using an array of ripple borrow subtractors. This approach is moderate in speed, allowing for a faster 24-bit significand multiplication.

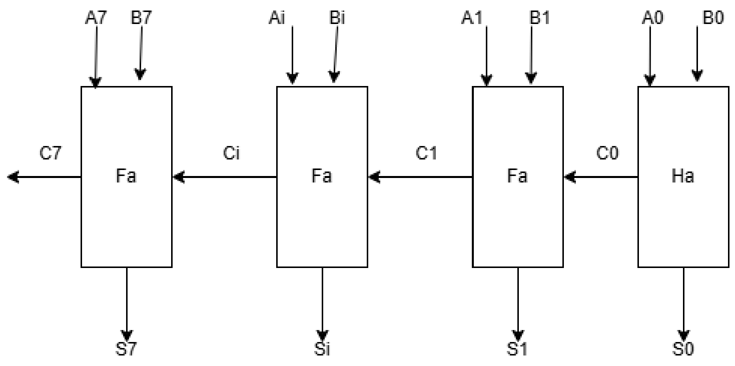


Figure 4.3 2 Unsigned Adder

4. Bias Subtraction

Subtract the bias constant (127 = 001111111) from unsigned exponent adder result for this, two zero subtractors (ZS) and seven one subtractors (OS) are used . S0…..S8 is the unsigned adder result (9 bit) .T=001111111 is the Bias constant. Bias subtractor result is R =S-T.

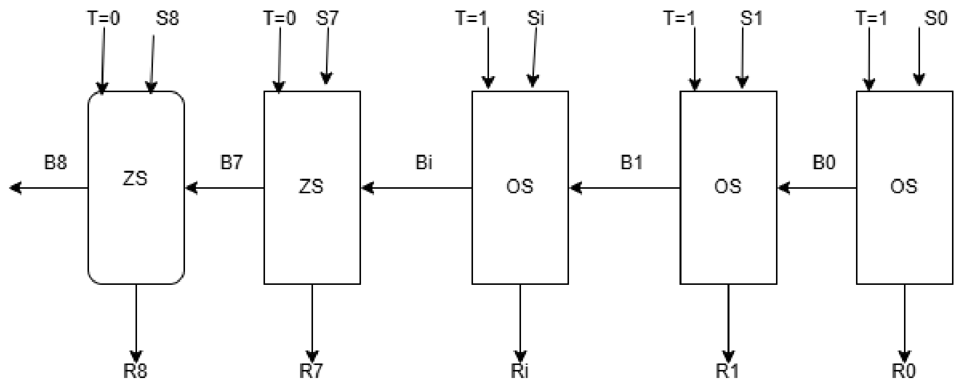


Figure 4.3 3 Bias Substractor

5. Mantissa Multiplier

The multiplication process is broken down into several key steps for efficiency:

- **Partial Product Generation:**
  The multiplication of two 24-bit mantissas (a and b) is decomposed into multiple smaller partial products. For each bit of mantissa `a`, it is multiplied with all bits of mantissa `b`. This generates a set of partial products. For example, multiplying bit 0 of `a` with each bit of `b` results in one partial product, then multiplying bit 1 of `a` with all bits of `b` forms another, and so on for all 24 bits of `a`.
- **Shifting:**
  After the partial products are generated, each one is shifted to align properly in the final result. This step is performed by the shifted versions of `b`, denoted as `bp1`, `bp2`, etc., which are used to generate the partial products.
- **Bitwise AND:**
  A bitwise AND operation is performed between corresponding bits of `a` and `b`. For example, `ab1 <= ap1 AND bp1` computes the product of the individual bits from `a` and `b`.
- **Summing Partial Products:**
  The results of the bitwise AND operations are then passed through a series of 24-bit full adders, instantiated as `Sumator24bits` components. These components sum the partial products, handle carry bits, and generate the final result.
- **Accumulate Results:**
  The summing process involves 23 different sums (from `sumator1` to `sumator23`), each handling a portion of the partial products and their carries, ultimately combining them into the final result.

6. Normalizer

The **intermediate product** from significand multiplication must be normalized to have a leading '1' in bit 46, just to the left of the decimal point. Since the inputs are already normalized, the leading '1' will be at either bit 46 or 47.

- If it's at bit 46, no shift is needed as the product is already normalized.
- If it's at bit 47, the product is shifted right, and the exponent is incremented by 1.

7.OverFlow/Underflow

Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. Underflow occurs if the intermediate exponent is less than zero, which cannot be compensated, or if it equals zero, which can be normalized by adding 1. In the event of overflow, the result is set to ±Infinity, with the sign determined by the inputs, and the overflow flag is raised. For underflow, the result is set to ±Zero (sign determined by inputs), and the underflow flag is raised. Denormalized numbers are treated as zero with the appropriate sign and an underflow flag raised.

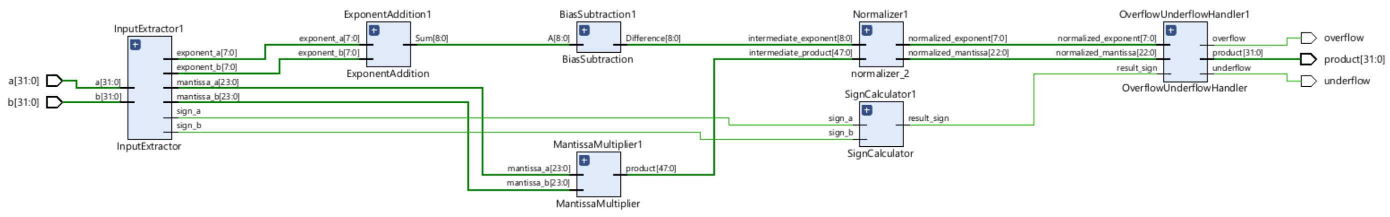


Figure 4.3 4 Block Design of FloatingPointMultiplier32

4.4 The registers

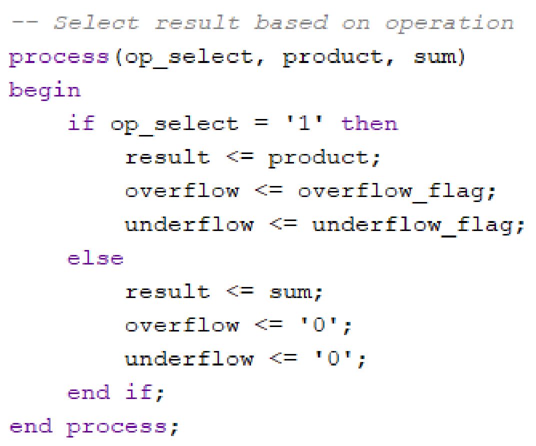
As for this component, it will behave like a ROM memory. We need to simulate the FPU and because the numbers are represented on 32 bits, to avoid writing multiple times when testing and writing simulation, this will be used to store values.

Implementation

When describing the implementation I will start from general to particular. Firstly, I will describe how the Top module works.

- Two ROM components provide the 32-bit operands (rom_data_a and rom_data_b) based on the addresses.
- The FloatingPointAdder32 and FloatingPointMultiplier32 components handle addition and multiplication operations, respectively.

- The output of either the adder or multiplier is selected based on op_select, with corresponding overflow and underflow flags set for multiplication results.



```
-- Select result based on operation
process(op_select, product, sum)
begin
    if op_select = '1' then
        result <= product;
        overflow <= overflow_flag;
        underflow <= underflow_flag;
    else
        result <= sum;
        overflow <= '0';
        underflow <= '0';
    end if;
end process;
```

Figure 5 1 Choosing the operation

The ROM registers(the project contain two for the two numbers that need to be chosen)

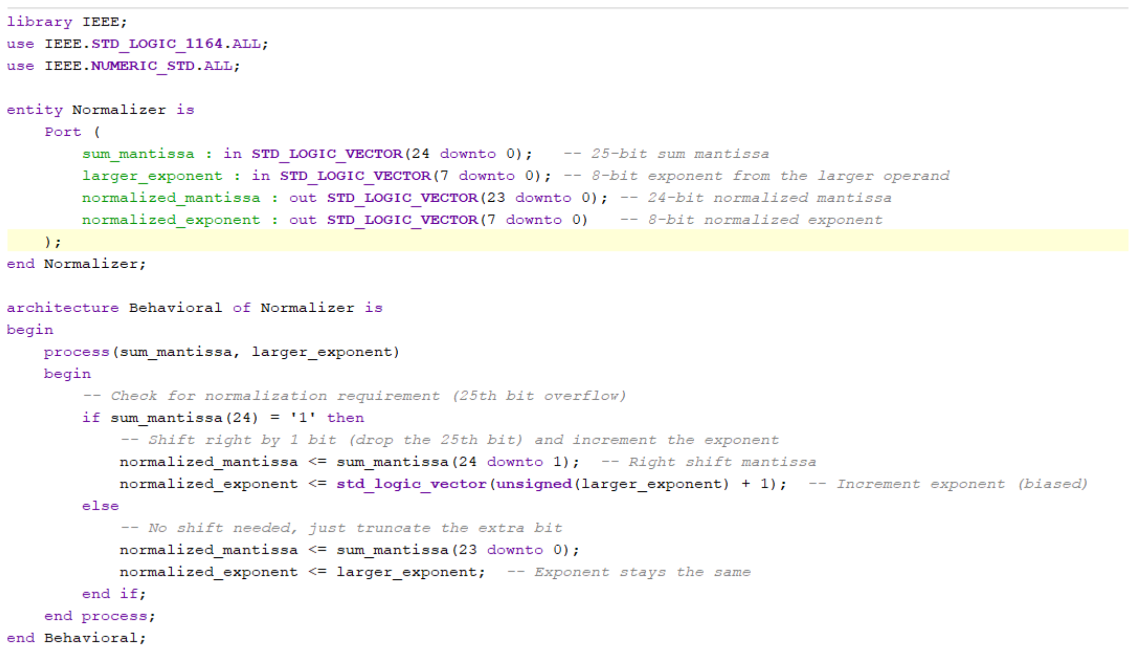
The ROM is initialized with specific 32-bit floating-point hexadecimal values (IEEE 754 format). Whenever the address input changes, the process block updates data_out with the 32-bit value from ROM_INIT at the specified address. The to_integer(unsigned(address)) function converts the address from STD_LOGIC_VECTOR to an integer index.

The FloatingPointAdder32 component decomposes the floating-point addition into specialized subcomponents that:

1. **Extract** and **parse** the fields of each operand.
2. **Align** the mantissas based on the exponent difference.
3. **Select the correct sign** for the result.
4. **Add** the mantissas.
5. **Normalize** the result to fit the IEEE-754 format.
6. **Package** the final result into a single 32-bit output.

Emphasis on Normalizer

The Normalizer is highlighted as the most important component because it ensures the floating-point result is in the correct format after addition. Without normalization, the result could exceed the representable range or lose precision. The Normalizer adjusts the mantissa and exponent, ensuring the result remains in normalized form, which is essential for IEEE-754 compliance.



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Normalizer is
    Port (
        sum_mantissa : in STD_LOGIC_VECTOR(24 downto 0);   -- 25-bit sum mantissa
        larger_exponent : in STD_LOGIC_VECTOR(7 downto 0); -- 8-bit exponent from the larger operand
        normalized_mantissa : out STD_LOGIC_VECTOR(23 downto 0); -- 24-bit normalized mantissa
        normalized_exponent : out STD_LOGIC_VECTOR(7 downto 0)   -- 8-bit normalized exponent
    );
end Normalizer;

architecture Behavioral of Normalizer is
begin
    process(sum_mantissa, larger_exponent)
    begin
        -- Check for normalization requirement (25th bit overflow)
        if sum_mantissa(24) = '1' then
            -- Shift right by 1 bit (drop the 25th bit) and increment the exponent
            normalized_mantissa <= sum_mantissa(24 downto 1);  -- Right shift mantissa
            normalized_exponent <= std_logic_vector(unsigned(larger_exponent) + 1);  -- Increment exponent (biased)
        else
            -- No shift needed, just truncate the extra bit
            normalized_mantissa <= sum_mantissa(23 downto 0);
            normalized_exponent <= larger_exponent;  -- Exponent stays the same
        end if;
    end process;
end Behavioral;
```

Figure 5 2 Normalizer Component

Normalization Check:

- The component checks if the 25th bit (bit 24) of sum_mantissa is set to '1'.
- If sum_mantissa(24) = '1', this indicates that an overflow occurred in the mantissa addition, and the result requires normalization.

    - The mantissa needs to be normalized by right-shifting it by 1 bit. This drops the overflow bit, fitting the result into the required 24 bits.
    - The exponent is incremented by 1 to account for this shift. This increment adjusts the result's scale, maintaining the correct value.

No Overflow:

- If sum_mantissa(24) = '0':
    - No shift is needed, so the component simply takes the lower 24 bits of sum_mantissa as the normalized_mantissa.
    - The exponent remains unchanged as larger_exponent.

The FloatingPointMultiplier32 component decomposes the floating-point multiplication into specialized subcomponents that:

1. InputExtractor:
   Extracts the sign, exponent, and mantissa from two 32-bit floating-point inputs (a, b).
2. SignCalculator:
   Computes the result sign by XORing the input signs (sign_a, sign_b).

3. ExponentAddition:
   Adds the exponents of a and b.
4. BiasSubtraction:
   Adjusts the exponent by subtracting the IEEE 754 bias.
5. Mantissa_Multiplier:
   Multiplies the mantissas (mantissa_a, mantissa_b).
6. Normalizer_2:
   Normalizes the mantissa and adjusts the exponent to ensure the result is in normalized form.
7. OverflowUnderflowHandler:
   Handles overflow and underflow conditions and produces the final result.

Logic of **Mantissa_Multiplier**:

- Inputs:
  - a: The 23-bit mantissa of the first floating-point number (a), which has already been extracted from the 32-bit input.
  - b: The 23-bit mantissa of the second floating-point number (b), also extracted.
- Output:
  - r: The 48-bit result of the mantissa multiplication. This is a larger bit-width (48 bits) than the input mantissas (23 bits) because the product of two 23-bit numbers can result in a value up to 46 bits wide. One extra bit is added to accommodate for any carry during the multiplication.

Intermediate Signals: There are multiple intermediate signals used throughout the design:

- ap1, bp1, ..., ap23, bp23: These signals represent shifted versions of a and b that are used in the partial product computations.
- ab1, ab2, ..., ab23: These signals represent bitwise AND operations between a and b for each term of the product.
- s1, s2, ..., s23: These signals are used to store the intermediate results of each addition performed by the Sumator24bit components.

 Bitwise AND Operation:

- For example, ap1 and bp1 are prepared by shifting b and a appropriately so that each bit of a is ANDed with each bit of b at different positions. This results in a set of partial products stored in ab1, ab2, ..., ab23.

 Shifting of Inputs:

- For each partial product, the bits of a and b are shifted before the AND operation. The shifts ensure that the intermediate products align properly for the subsequent addition.

 24-Bit Adders (Sumator24bit):

- The core of the design involves adding the partial products using a series of 24-bit adders. Each Sumator24bit instance computes the sum of two 24-bit values and includes an extra carry-out bit to propagate any carry between the summations.
- The first Sumator24bit (sumator1) adds the result of the AND operation ab1 with the shifted version of itself (from the previous iteration). The carry-out of each addition is used as input to the next adder.
- This process is repeated for each of the 24 bits, from sumator1 to sumator23.

 Result Accumulation:

- The results of the Sumator24bit operations are stored in intermediate signals s1, s2, ..., s23. Each sum at each stage adds a shifted version of the partial products to the cumulative result.
- The first bit of the sum from each Sumator24bit is stored in the result vector r. For example, r(1) stores the first bit of s1, r(2) stores the first bit of s2, and so on.

 Final Result:

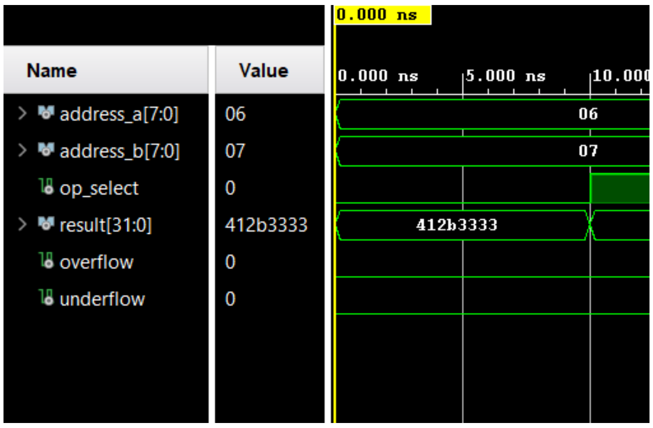
- The final output r(47 downto 23) is computed by taking the last sum, s23, which represents the fully accumulated result of the multiplication process. This is a 24-bit wide result, which is then assigned to the top 24 bits of the 48-bit result vector r.

Testing and Validation

The following examples were used to test the correct functionality of the algorithm using numbers in the floating-point representation:

- A=7.5= 01000000111100000000000000000000 = 40F00000 hex in ROM at adress 6
  B=3.2= 01000000010011001100110011001101= 404CCCCD hex in ROM at adress 7
  Operation= 0 (addition)
  Expected result= 10.7 = 01000001001010110011001100110011 = 412B3333hex

Actual Result:



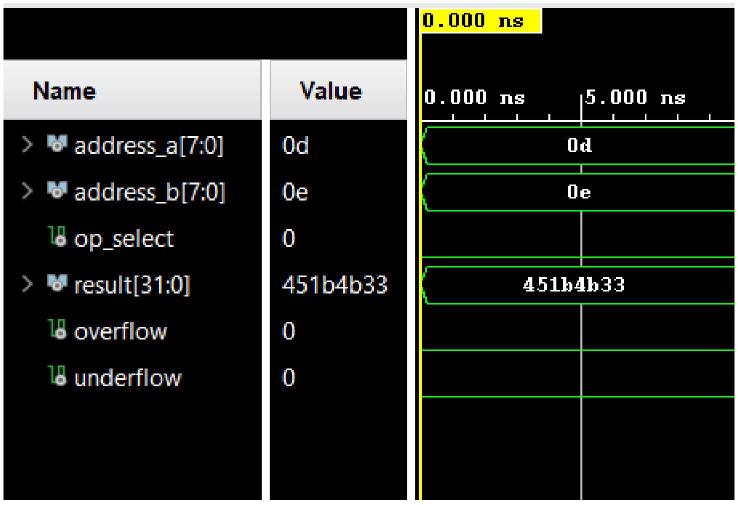
Name	Value
address_a[7:0]	06
address_b[7:0]	07
op_select	0
result[31:0]	412b3333
overflow	0
underflow	0

Figure 6 1 Simulation

Test passed.

- A=2034.5 = 01000100111111100101000000000000 = 44FE5000 hex in ROM at address 13
  B=450.2 = 01000011111000010001100110011010= 43E1199A hex in ROM at address 14
  Operation= 0 (addition)
  Expected result= 2484.7 = 01000101000110110100101100110011= 451B4B33 hex

Actual Result:



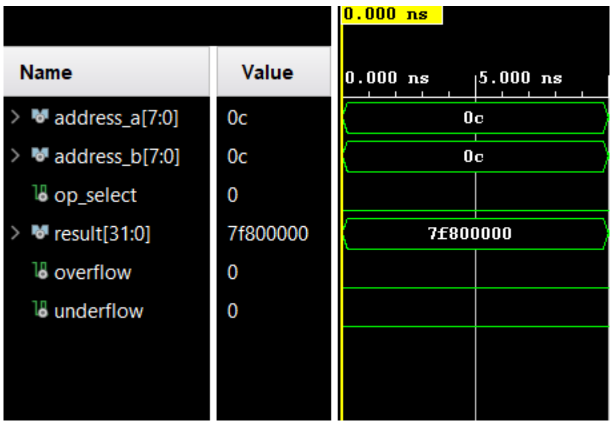
Name	Value
address_a[7:0]	0d
address_b[7:0]	0e
op_select	0
result[31:0]	451b4b33
overflow	0
underflow	0

Figure 6 2 Simulation

Test passed.

- A= +inf = 01111111100000000000000000000000= 7f800000 hex in ROM at address 12
  B= +inf = 01111111100000000000000000000000= 7f800000 hex in ROM at address 12
  Operation= 0 (addition)
  Expected result= 01111111100000000000000000000000= 7f800000 hex

Actual Result:



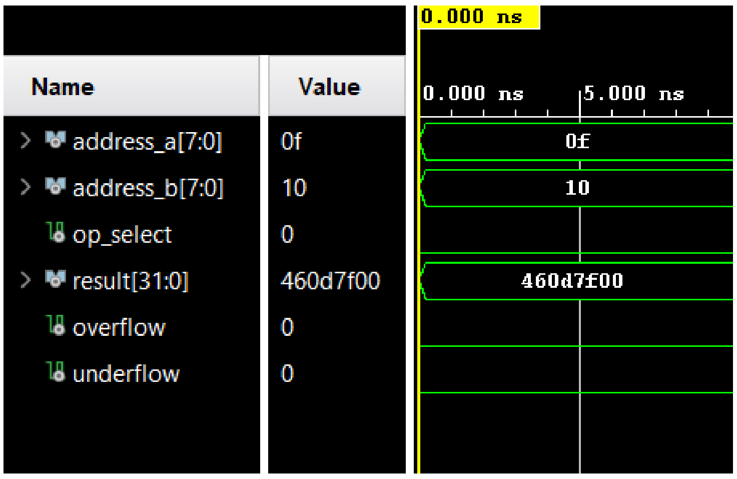
Name	Value
address_a[7:0]	0c
address_b[7:0]	0c
op_select	0
result[31:0]	7f800000
overflow	0
underflow	0

Figure 6 3 Simulation

Test passed.

- A= 5605.2= 01000101101011110010100110011010= 45AF299A hex in ROM at address 15
  B= 3450.55 = 01000101010101111010100011001101= 4557A8CD hex in ROM at address 16
  Operation= 0 (addition)
  Expected result= 01000110000011010111111100000000= 460D7F00 hex

Actual Result:



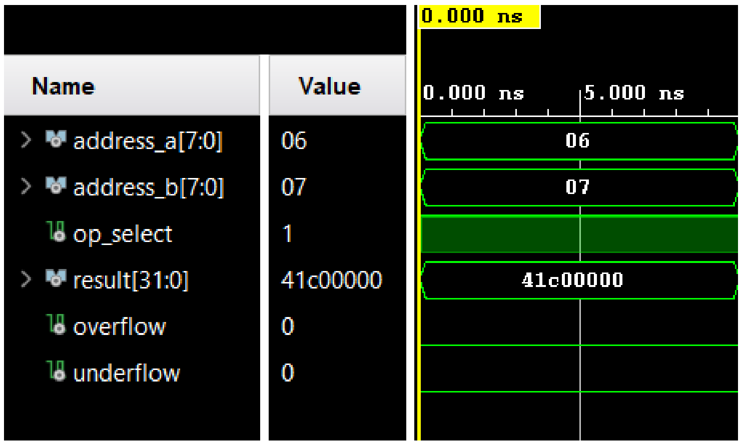
Name	Value
address_a[7:0]	0f
address_b[7:0]	10
op_select	0
result[31:0]	460d7f00
overflow	0
underflow	0

Figure 6 4 Simulation

Test passed.

- A= 7.5= 01000000111100000000000000000000= 40F00000 hex in ROM at address 6
  B= 3.2 = 01000000010011001100110011001101= 404CCCCD hex in ROM at address 7
  Operation= 1 (multiplication)
  Expected result= 01000001110000000000000000000000= 41C00000 hex
  Overflow=0
  Underflow=0

Actual Result:



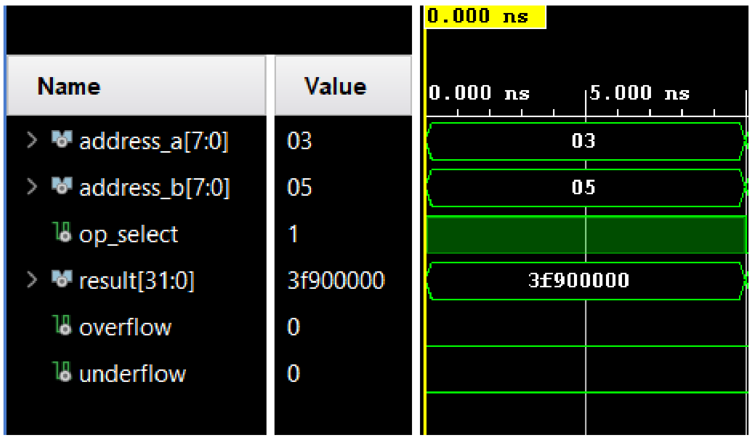
Name	Value
address_a[7:0]	06
address_b[7:0]	07
op_select	1
result[31:0]	41c00000
overflow	0
underflow	0

Figure 6 5 Simulation

Test passed.

- A= 0.5= 00111111000000000000000000000000= 3F000000 hex in ROM at address 3
  B= 2.25 = 01000000000100000000000000000000= 40100000 hex in ROM at address 5
  Operation= 1 (multiplication)
  Expected result= 00111111100100000000000000000000= 3F900000 hex
  Overflow=0
  Underflow=0

Actual Result:



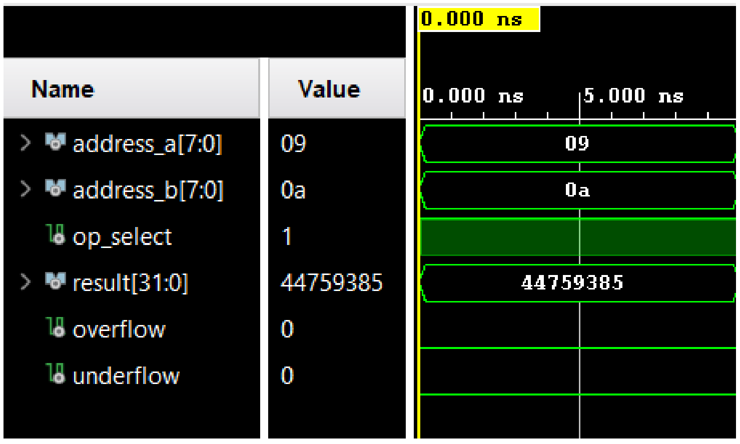
Name	Value
address_a[7:0]	03
address_b[7:0]	05
op_select	1
result[31:0]	3f900000
overflow	0
underflow	0

Figure 6 6 Simulation

Test passed.

- A= 41.5= 01000010001001100000000000000000= 42260000 hex in ROM at address 9
  B= 23.67 = 01000001101111010101110000101001= 41BD5C29 hex in ROM at address 10
  Operation= 1 (multiplication)
  Expected result= 01000100011101011001001110000101= 44759385 hex
  Overflow=0
  Underflow=0

Actual Result:



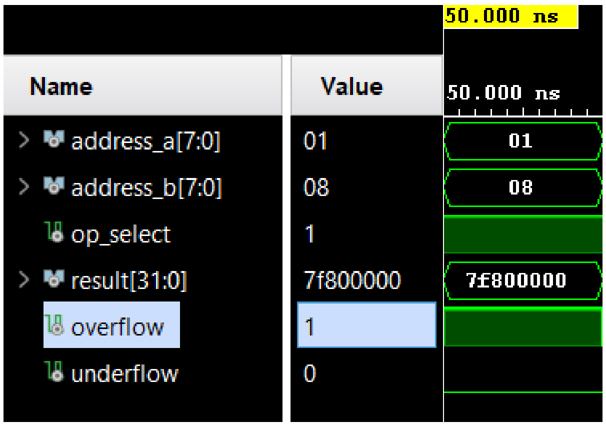
Name	Value
address_a[7:0]	09
address_b[7:0]	0a
op_select	1
result[31:0]	44759385
overflow	0
underflow	0

Figure 6 7 Simulation

Test passed.

- A= 2= 01000000000000000000000000000000= 40000000 hex in ROM at address 1
  B= +inf = 01111111100000000000000000000000= 41BD5C29 hex in ROM at address 8
  Operation= 1 (multiplication)
  Expected result= 01111111100000000000000000000000= 7f800000 hex
  Overflow=1
  Underflow=0

Actual Result:



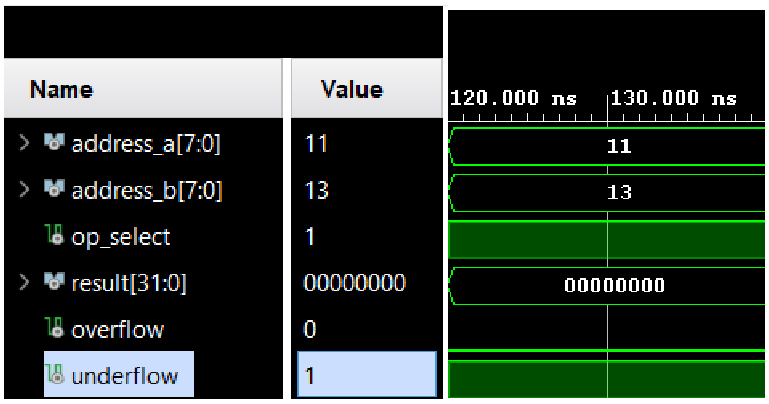
Name	Value
address_a[7:0]	01
address_b[7:0]	08
op_select	1
result[31:0]	7f800000
overflow	1
underflow	0

Figure 6 8 Simulation

Test passed.

- A= sub normal = 00000000000000000000000000000001= 00000001 hex in ROM at address 17
  B= sub normal 2 = 00000000000000000000000000000010= 0000002 hex in ROM at address 19
  Operation= 1 (multiplication)
  Expected result= 0000000000000000000000000000000= 00000000 hex
  Overflow=0
  Underflow=1

  Actual Result:



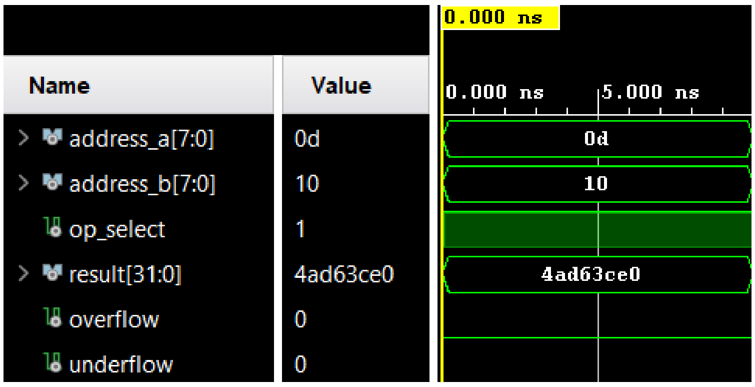
Name	Value
address_a[7:0]	11
address_b[7:0]	13
op_select	1
result[31:0]	00000000
overflow	0
underflow	1

Figure 6 9 Simulation

Test passed.

- A= 2034.5= 01000100111111100101000000000000= 42260000 hex in ROM at address 13
  B= 3450.55 = 01000101010101111010100011001101= 41BD5C29 hex in ROM at address 16
  Operation= 1 (multiplication)
  Expected result= 01001010110101100011110011100000= 4AD63CE0 hex
  Overflow=0
  Underflow=0

Actual Result:



```
0.000 ns
Name              Value       0.000 ns        5.000 ns
> address_a[7:0]  0d                  0d
> address_b[7:0]  10                  10
  op_select       1
> result[31:0]    4ad63ce0         4ad63ce0
  overflow        0
  underflow       0
```

Figure 6 10 Simulation

Test passed.

Conclusions

The goal of this project was to design, implement and test 2 operations on the standard IEEE format on 32 bits. The difference from the usual implementations of addition and multiplication is that this unit will perform them on floating point numbers with single precision.

Vivado, in particular, proved challenging for this project due to its numerous bugs and the lack of clarity in its error messages, especially when compared to tools like Visual Studio or IntelliJ. On the positive side, I gained valuable experience using debugging tools and breakpoints to identify and resolve issues.

Additionally, the online documentation for this subject was rather limited. It mostly outlines the general steps of the algorithm without providing many examples, especially for edge cases. However, by piecing together information from algorithms found online, YouTube tutorials, and various implementation ideas, I was able to develop a mostly functional FPU. I say "mostly" because, as with any application, testing can only confirm the presence of bugs, not their absence, and it was impractical to test every possible scenario.

Bibliography:

- [1] M.Bhavani, Design of Single Precision Floating Point Arithmetic Logic Unit: **https://www.ijera.com/papers/vol11no8/Ser-3/D1108031824.pdf**
- [2] Noor Alhuda Saad Adela**,** Design and Implementation of Single Precision Floating-point Arithmetic Logic Unit for RISC Processor on FPGA**,** **https://www.researchgate.net/publication/372170624_Design_and_Implementation_of_Single_Precision_Floating-point_Arithmetic_Logic_Unit_for_RISC_Processor_on_FPGA**
- [3] DR A. P. SHANTHI**,** Floating Point Arithmetic Unit, **https://www.cs.umd.edu/~meesh/411/CA-online/chapter/floating-point-arithmetic-unit/index.html**
- [4] Yuguo Liu, The Mechanism of The Arithmetic Logic Unit**,** **https://www.researchgate.net/publication/377737488_The_Mechanism_of_The_Arithmetic_Logic_Unit**
- [5] Geeks for geeks, IEEE-Standard-754-floating-point-numbers **https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/**
- [6] Remadevi R, Design and Simulation of Floating Point Multiplier Based on VHDL, 2013
- [7] remusbomba Floating-point-multiplier https://github.com/remusbompa/Floating-Point-multiplier/blob/master/sources_1/new/Modul4.vhd