32 BIT FLOATING POINT ADDITION AND SUBTRACTION

FPU ADDITION AND SUBTRACTION:

Multiplication is a fundamental arithmetic operation and multiplication of two numbers using the following architecture.

ARCHITECTURE:

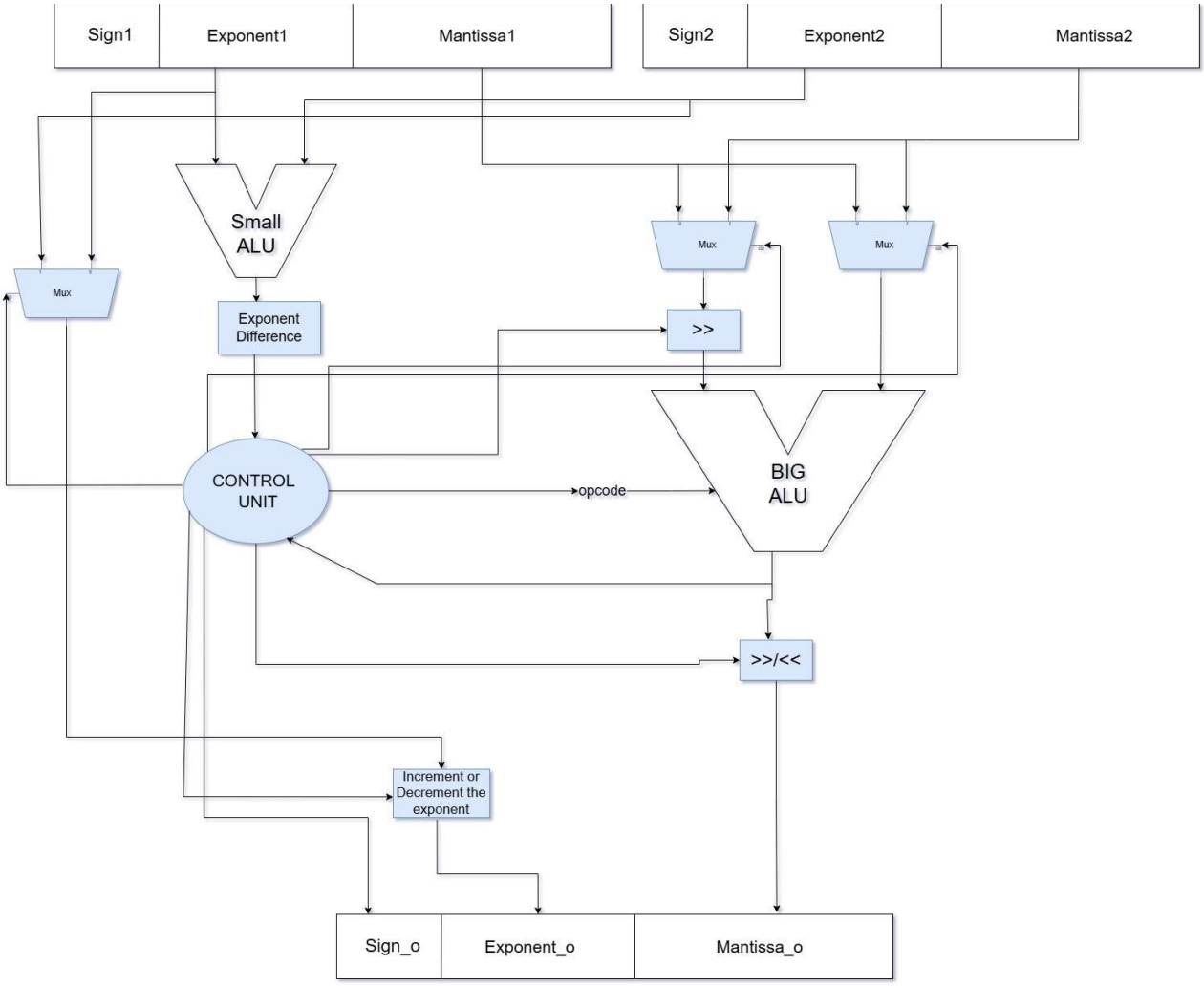


Figure : MULTIPLICATION OPERATION

Explanation of the flow of operation:

1. STAGE 1 : In which exponent difference is calculated using which bigger and smaller mantissas determined, also bigger exponent is determined.

2. STAGE 2 : Smaller exponent is right shifted by the amount of exponent difference so that both the numbers will have same exponent before performing addition or subtraction.

3. STAGE 3 : In which actual addition or subtraction performed on mantissa and passed to stage 4 and control unit to find position of 1st logic 1 checking from MSB.

4. STAGE 4 : Output of Stage 3 is passed for normalization using left shifting by the amount of position on mantissa and subtract position from bigger exponent to get normalized mantissa and normalized exponent.

CHALLENGES:

1. Initially designed was completely combination when registered input and output at 100MHz, observed slack of –106 ns, which is negative slack which will cause metastability.

2. After dividing design into stages and optimized timing using pipelining we observed slack of 2.645 ns, which is positive slack.

ELABORATED DESIGN:

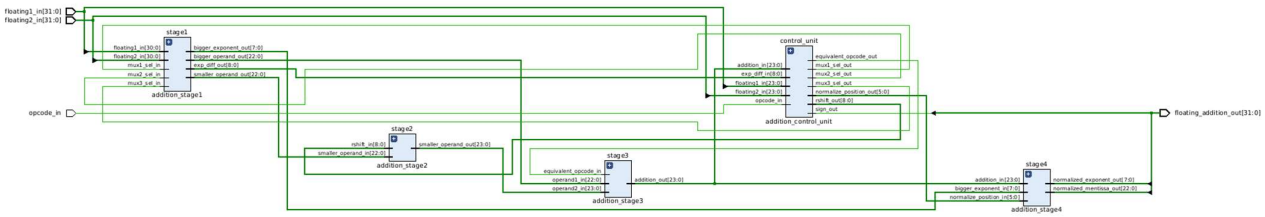
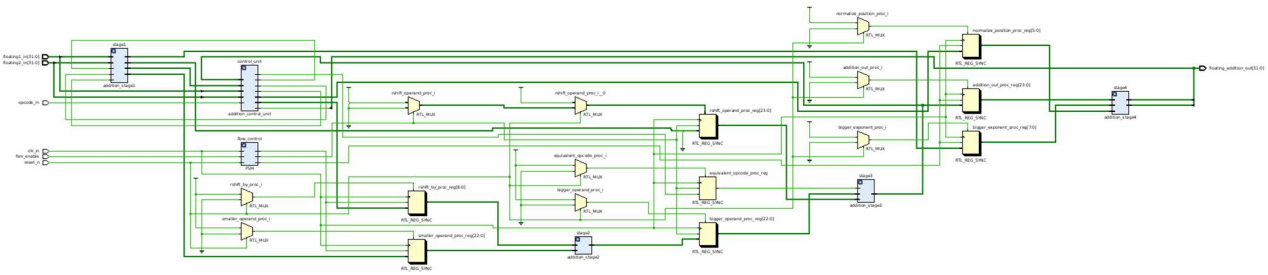


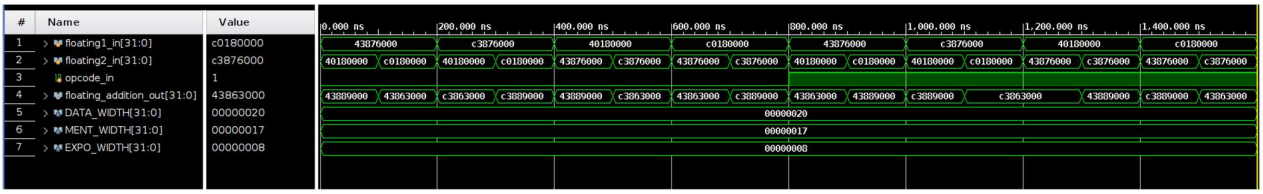
figure : Elaborated design of floating point addition and subtraction

ELABOARTED DESIGN:



SIMULATION RESULTS:

Timing Diagram

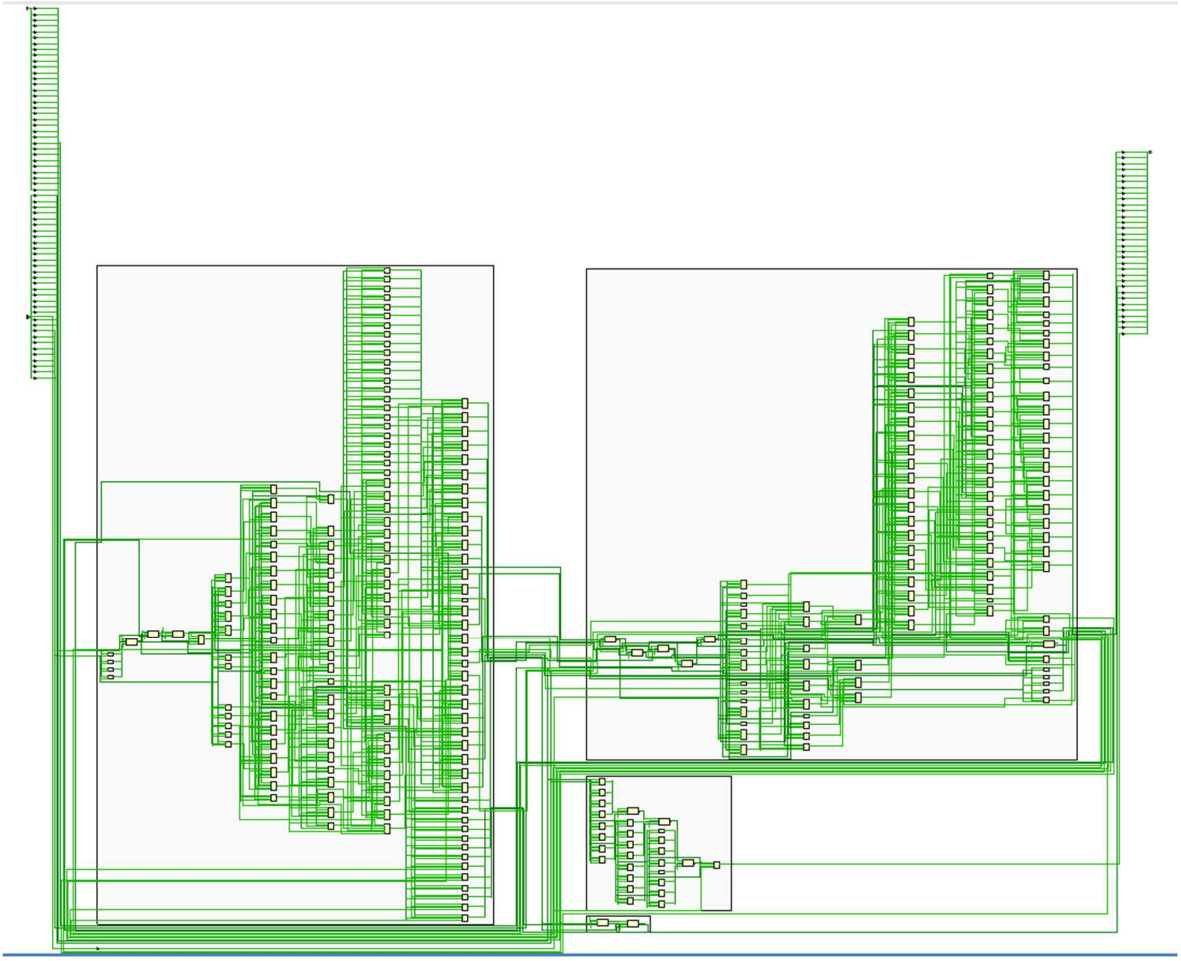


TCL Console Results

```
SIMULATION BEGINS...
CORNER_CASE I.1.1
opcode = 0 :: floating1 = 01000011100001110110000000000000 :: floating2 = 01000000000110000000000000000000 :: floating out = 01000011100010001001000000000000
CORNER_CASE I.1.2
opcode = 0 :: floating1 = 01000011100001110110000000000000 :: floating2 = 11000000000110000000000000000000 :: floating out = 01000011100001100011000000000000
CORNER_CASE I.1.3
opcode = 0 :: floating1 = 11000011100001110110000000000000 :: floating2 = 01000000000110000000000000000000 :: floating out = 11000011100001100011000000000000
CORNER_CASE I.1.4
opcode = 0 :: floating1 = 11000011100001110110000000000000 :: floating2 = 11000000000110000000000000000000 :: floating out = 11000011100010001001000000000000
CORNER_CASE I.2.1
opcode = 0 :: floating1 = 01000000000110000000000000000000 :: floating2 = 01000011100001110110000000000000 :: floating out = 01000011100010001001000000000000
CORNER_CASE I.2.2
opcode = 0 :: floating1 = 01000000000110000000000000000000 :: floating2 = 11000011100001110110000000000000 :: floating out = 11000011100001100011000000000000
CORNER_CASE I.2.3
opcode = 0 :: floating1 = 11000000000110000000000000000000 :: floating2 = 01000011100001110110000000000000 :: floating out = 01000011100001100011000000000000
CORNER_CASE I.2.4
opcode = 0 :: floating1 = 11000000000110000000000000000000 :: floating2 = 11000011100001110110000000000000 :: floating out = 11000011100010001001000000000000
CORNER_CASE II.1.1
opcode = 1 :: floating1 = 01000011100001110110000000000000 :: floating2 = 01000000000110000000000000000000 :: floating out = 01000011100001100011000000000000
CORNER_CASE II.1.2
opcode = 1 :: floating1 = 01000011100001110110000000000000 :: floating2 = 11000000000110000000000000000000 :: floating out = 01000011100010001001000000000000
CORNER_CASE II.1.3
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_floating_point_addition_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:07 ; elapsed = 00:00:12 . Memory (MB): peak = 8798.570 ; gain = 87.230 ; free physical = 4464 ; free virtual = 12336
run 600ns
opcode = 1 :: floating1 = 11000011100001110110000000000000 :: floating2 = 01000000000110000000000000000000 :: floating out = 11000011100010001001000000000000
CORNER_CASE II.1.4
opcode = 1 :: floating1 = 11000011100001110110000000000000 :: floating2 = 11000000000110000000000000000000 :: floating out = 11000011100001100011000000000000
CORNER_CASE II.2.1
opcode = 1 :: floating1 = 01000000000110000000000000000000 :: floating2 = 01000011100001110110000000000000 :: floating out = 11000011100001100011000000000000
CORNER_CASE II.2.2
opcode = 1 :: floating1 = 01000000000110000000000000000000 :: floating2 = 11000011100001110110000000000000 :: floating out = 01000011100010001001000000000000
CORNER_CASE II.2.3
opcode = 1 :: floating1 = 11000000000110000000000000000000 :: floating2 = 01000011100001110110000000000000 :: floating out = 11000011100010001001000000000000
CORNER_CASE II.2.4
opcode = 1 :: floating1 = 11000000000110000000000000000000 :: floating2 = 11000011100001110110000000000000 :: floating out = 01000011100001100011000000000000
SIMULATION ENDS...
```

figure : Output on TCL Console

<u>SYNTHESIS</u>:

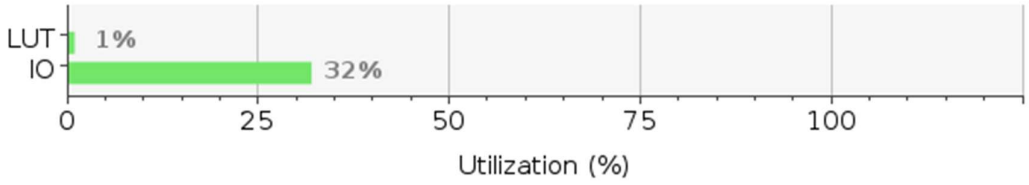


UTILIZATION OF BLOCKS FOR NON-PIPELINED:

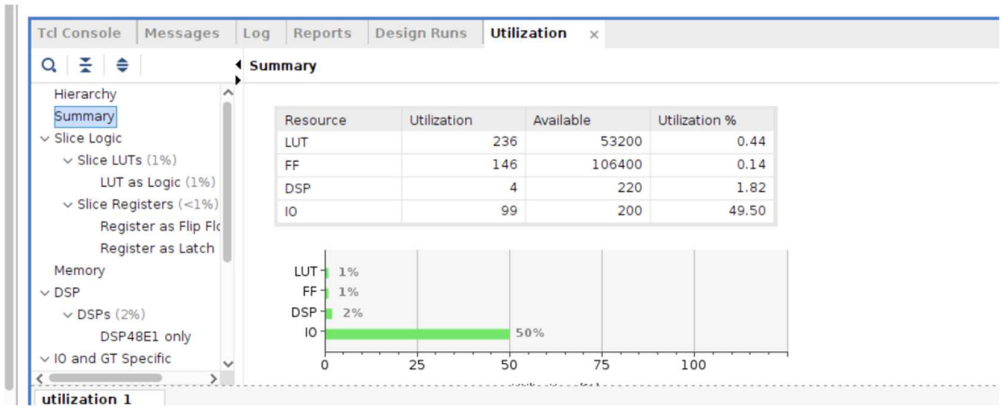
Name	Slice LUTs (41000)	Bonded IOB (300)
floating_point_addition	249	97
control_unit (addition_control_unit)	13	0
stage1 (addition_stage1)	138	0
stage3 (addition_stage3)	98	0
stage4 (addition_stage4)	0	0

Summary

Resource	Utilization	Available	Utilization %
LUT	249	41000	0.61
IO	97	300	32.33



WITH USAGE OF KARATSUBA MULTIPLIER:



UTILIZATION OF BLOCKS FOR PIPELINED:

Name	Slice LUTs (41000)	Slice Registers (82000)	Bonded IOB (300)	BUFGCTRL (32)
floating_point_addition	229	121	100	1
control_unit (addition_control_unit)	13	0	0	0
flow_control (FSM)	2	6	0	0
stage1 (addition_stage1)	36	0	0	0
stage3 (addition_stage3)	50	0	0	0
stage4 (addition_stage4)	7	0	0	0

MULTIPLICATION

FPU MULTIPLICATION:

Multiplication is a fundamental arithmetic operation and multiplication of two numbers using the following architecture.

ARCHITECTURE:

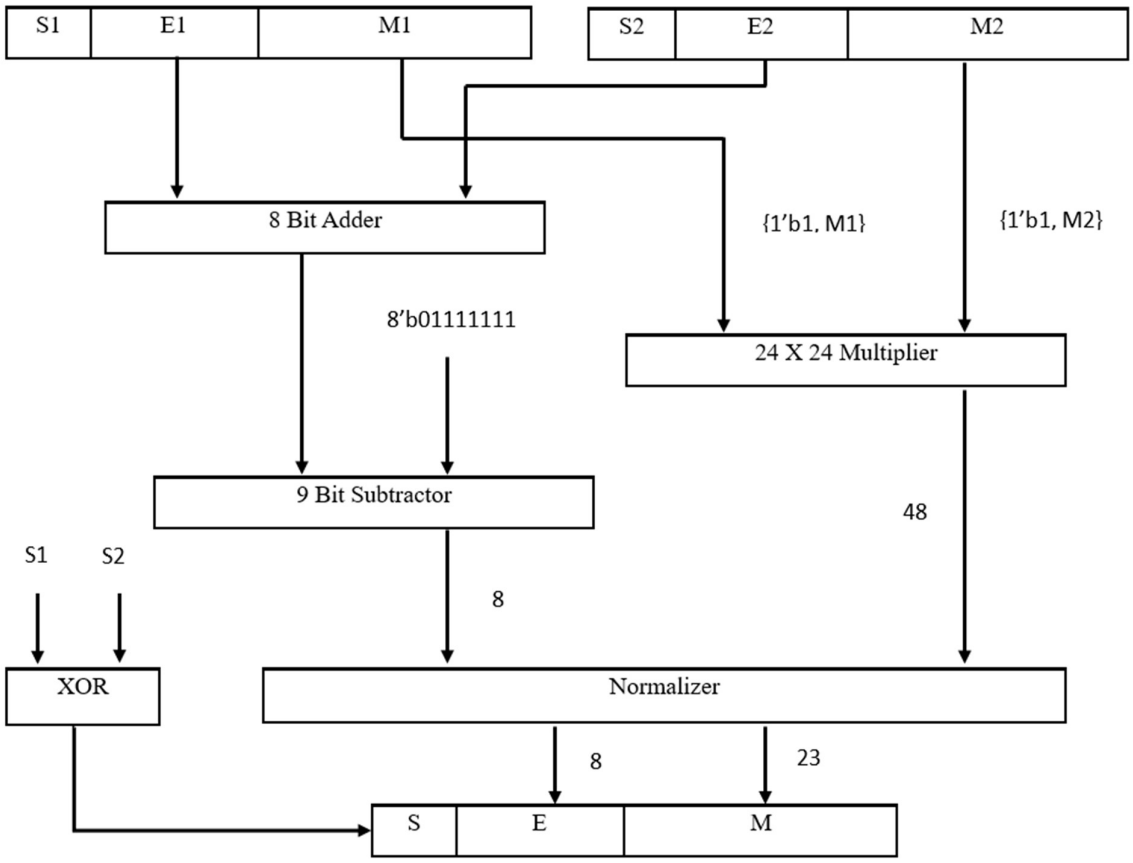


Figure : MULTIPLICATION OPERATION

Explanation of the flow of operation:

1. 32-bit input is taken from and swizzling the input into sign, exponent and mantissa.
2. 8-bit Exponents of two numbers were driven to the 8-bit adder, and the 9-bit result including carry is generated.
3. 23-bit Mantissas of two numbers were passed to the 24 x 24 multiplier, Karatsuba multiplier was used, where here our mantissas by default passed with the 1'b1 in the MSB because according to the IEEE 754 format, in formats they mentioned that 1 will be implicitly taken in the 24^{th} bit and that is encoded with the exponent later on.
4. In the multiplier block, output is 48-bit because in binary if **n** bit inputs are there then the resultant output will be **2n** bits.

5. Exponent result from the 8-bit adder is given to the subtractor, where to get the unbiased exponent, i.e., exponent result = $(exp1 + exp2)_{biased\_exponent}$ and now that result will be subtracted from the bias to get the true exponent. True exponent = exponent result – 8'b01111111(127 in decimal).

6. Normalization is required if the number is denormalized, the inputs exponent and 48-bit multiplied mantissa will be given to the normalizer,

7. In normalizer, based on the position of 1st occurrence of 1, the mantissa is left shifted, as well as the exponent is incremented or decremented by position magnitude.

8. From the normalizer, the upper side of the 48-bit mantissa i.e., 23-bit [47:25] will be taken and given as an mantissa out from normalizer.

9. Exponent is also given from the normalizer.

10. Sign is determined by the xor operation of two signs of two numbers.

    a. + x + = +;

    b. + x - = -;

    c. – x + = -;

    d. – x - = +;

    //Similar to the Xor Operation.

11. Final result is concatenated to 32-bit result.

FLOWCHART FOR MULTIPLICATION:

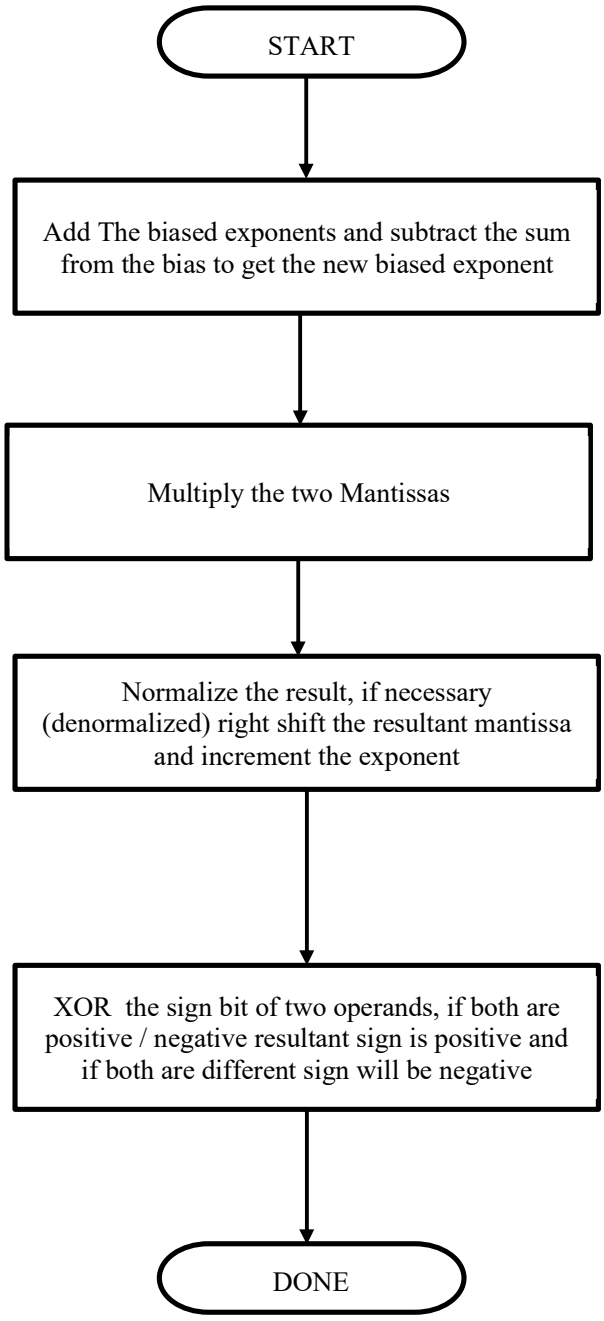


Figure FLOW CHART

CHALLENGES:

- After using the above architecture, purely combinational design has been done, during synthesis the timing violations are present with the design i.e., negative slack is more for the design.
- To avoid the above violation, Pipelined design was done for the above architecture.
- Three stage pipeline design was done.

a) In first stage, inputs are registered.

b) In second stage, inputs the normalizer are registered.

c) In third stage, outputs from the normalizer are registered.

PIPELINED ARCHITECTURE:

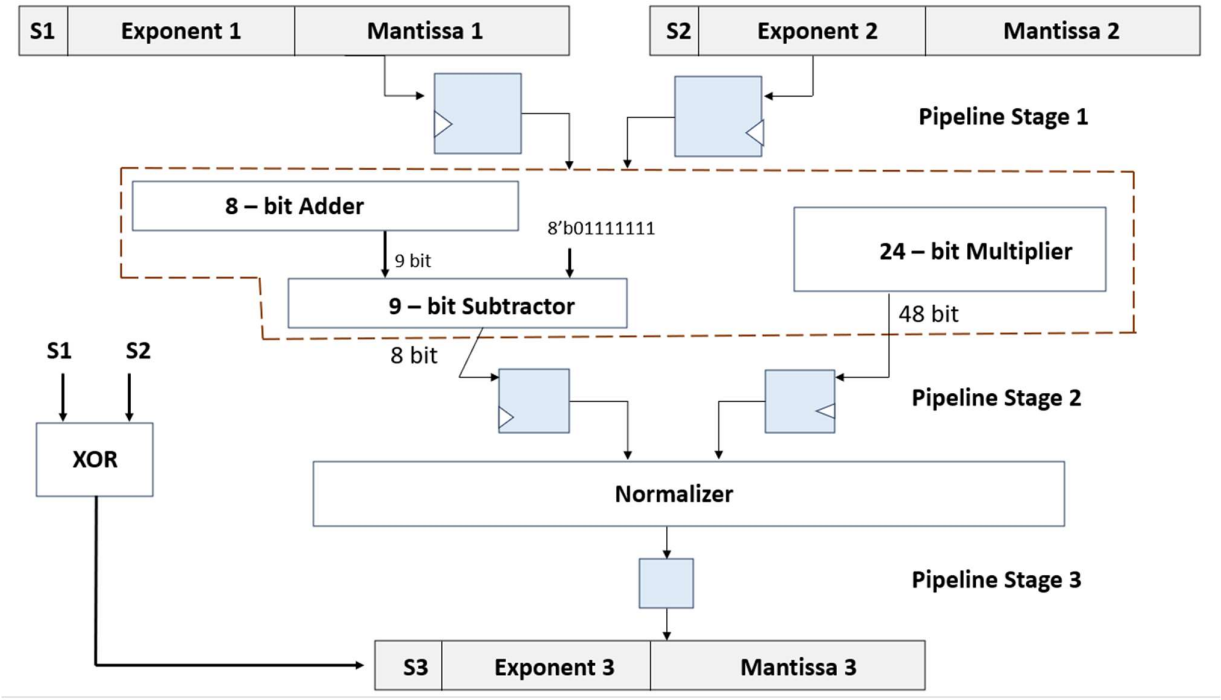


Figure Pipelined Architecture

ELABORATED DESIGN:

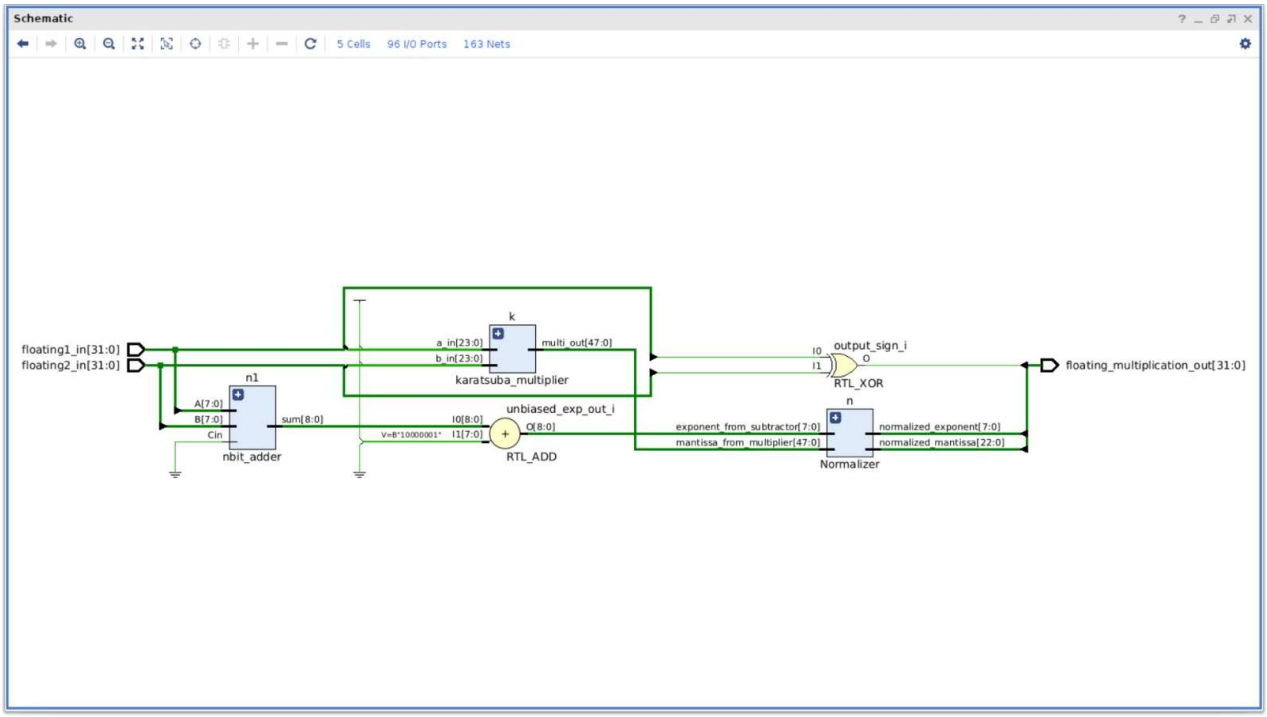


Figure Elaborated Design Non Pipelined

PIPELINED ELABORATED DESIGN:

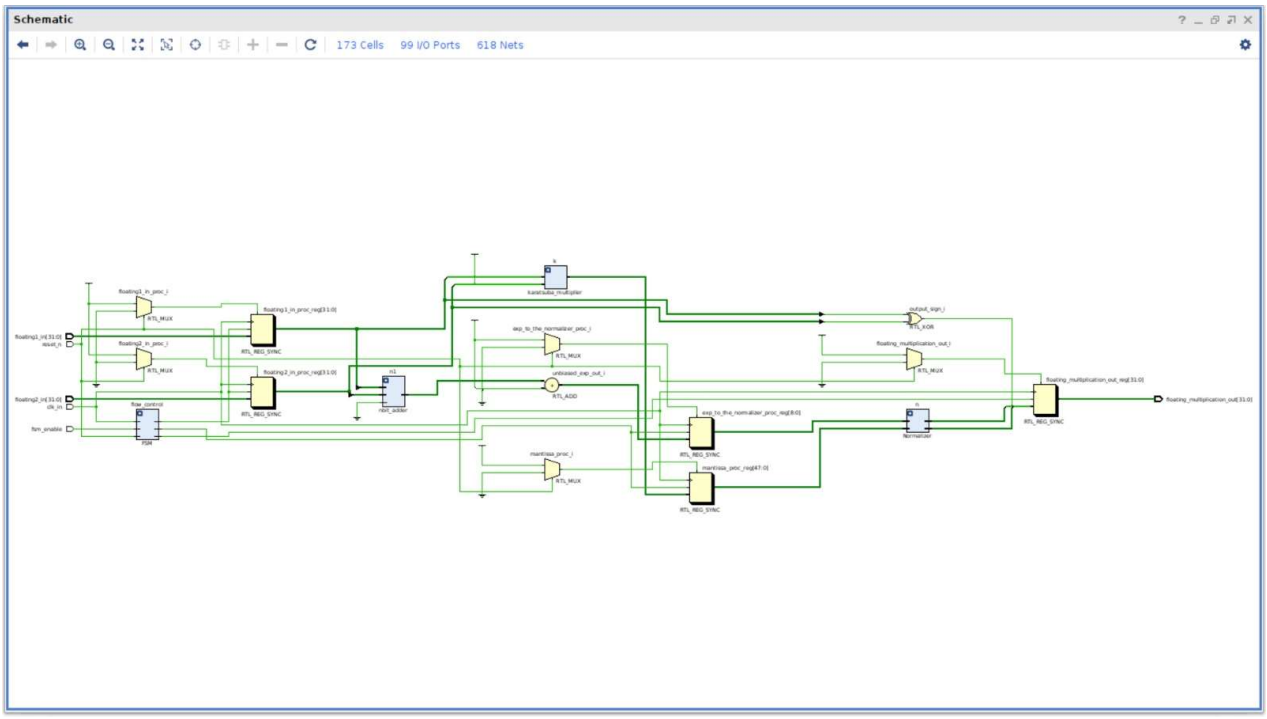


Figure Elaborated Design Pipelined

SIMULATION RESULTS:

Generated test cases for the design and validated through IEEE 754 format calculator.

Console Results

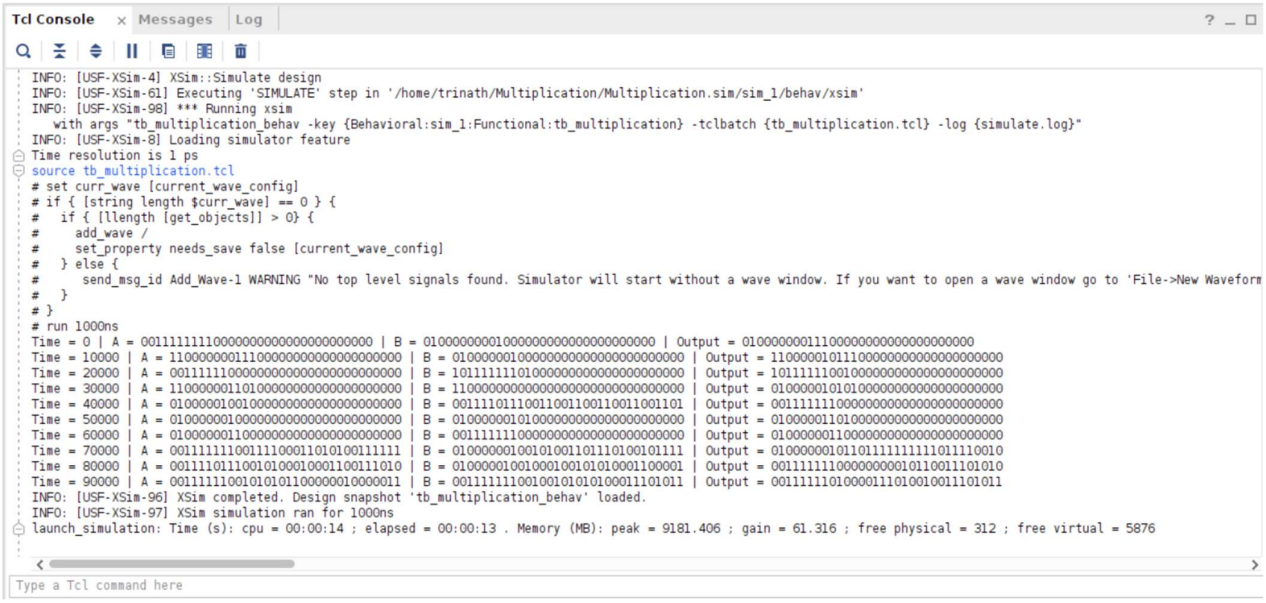


Figure Outputs from the console

Simulation Graph

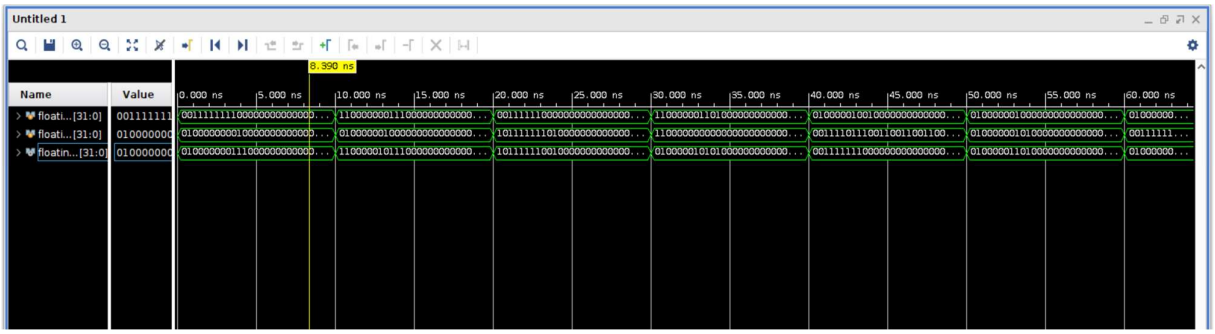
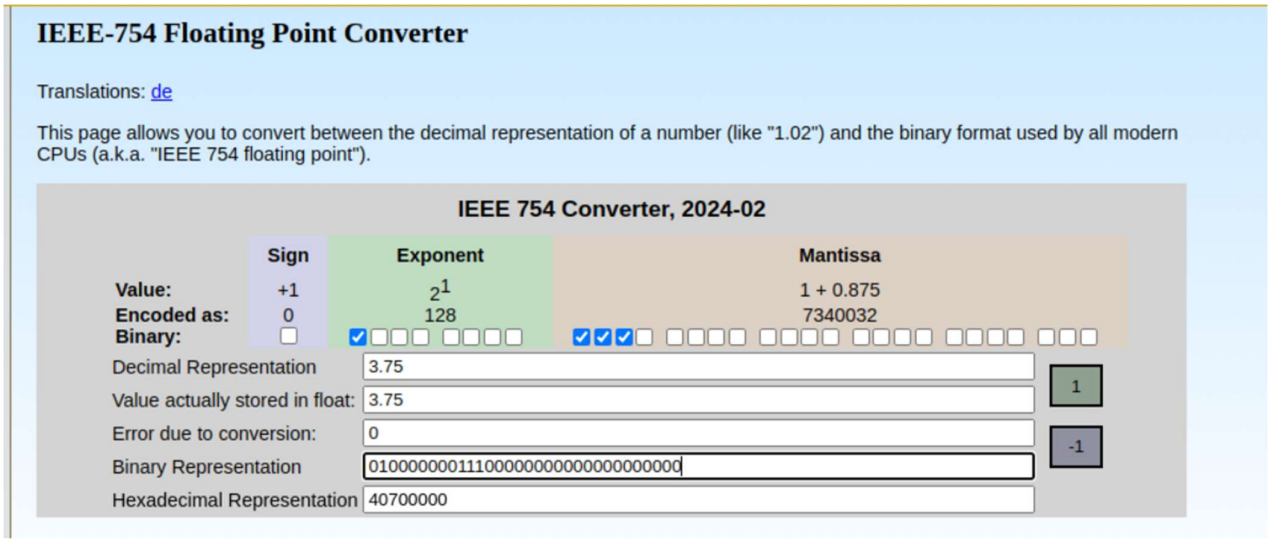


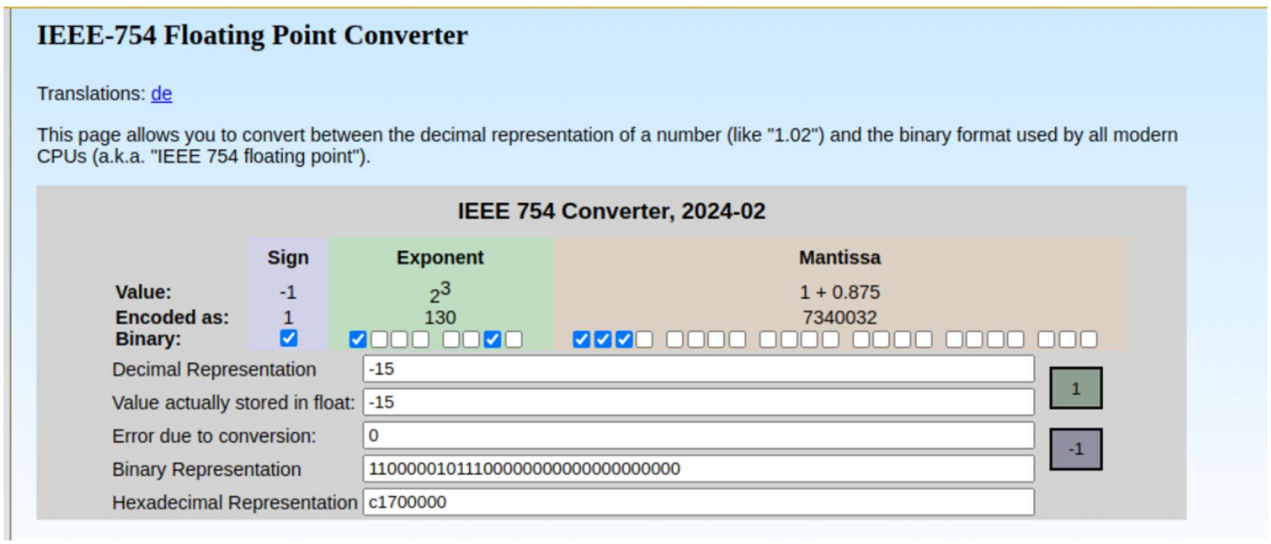
Figure Simulation Graph

Test Cases:

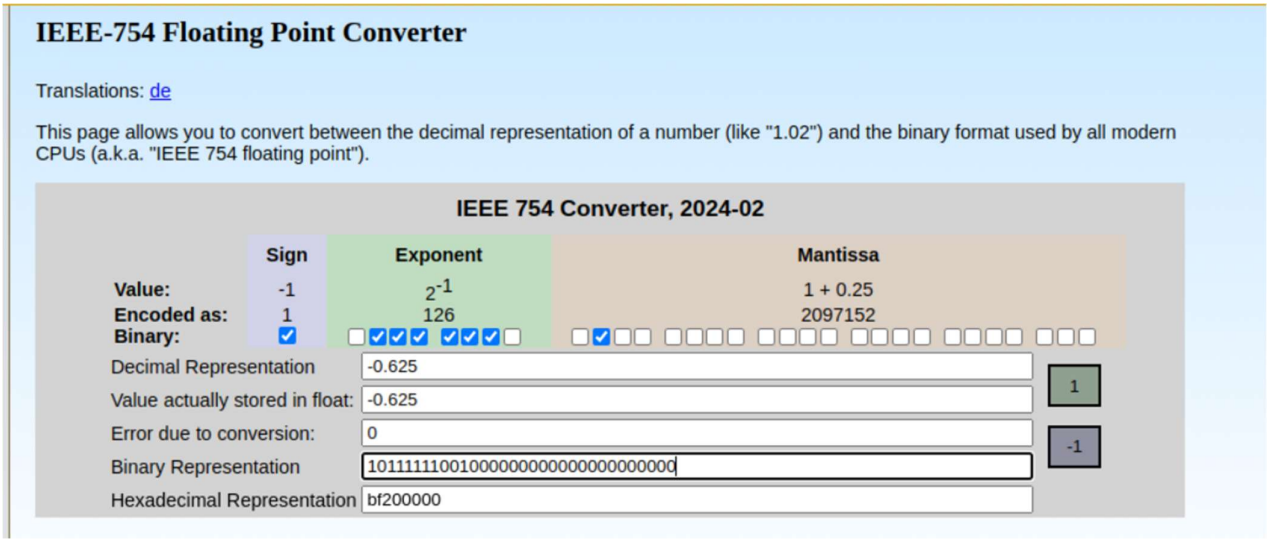
1. 2.5 * 1.5 = 3.75 (01000000011100000000000000000000)



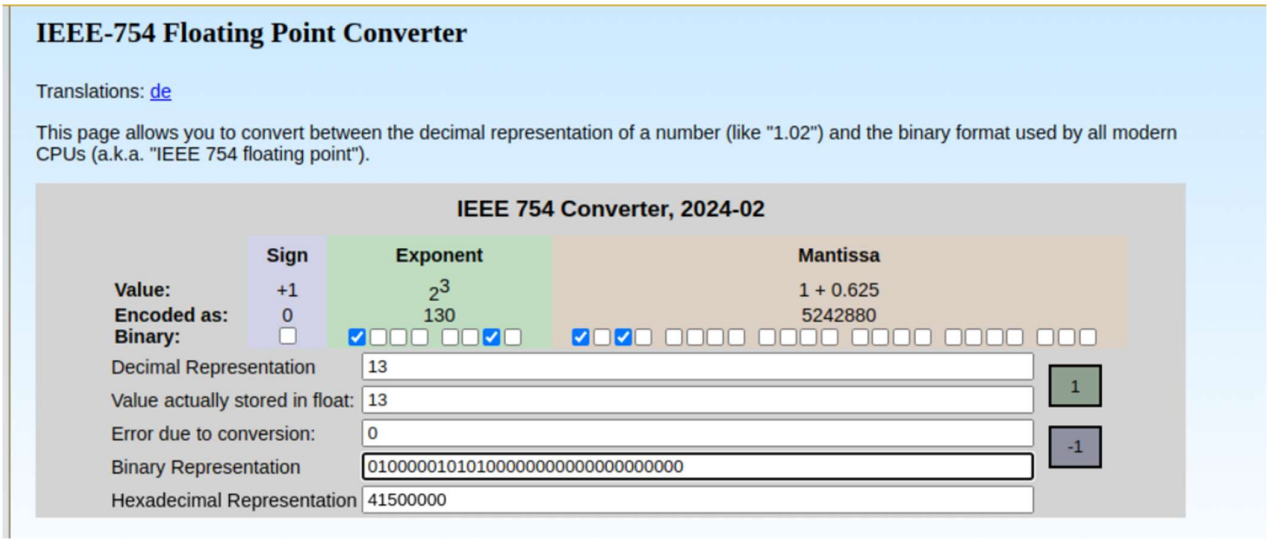
2. (-3.75) * 4 = -15 (11000001011100000000000000000000)



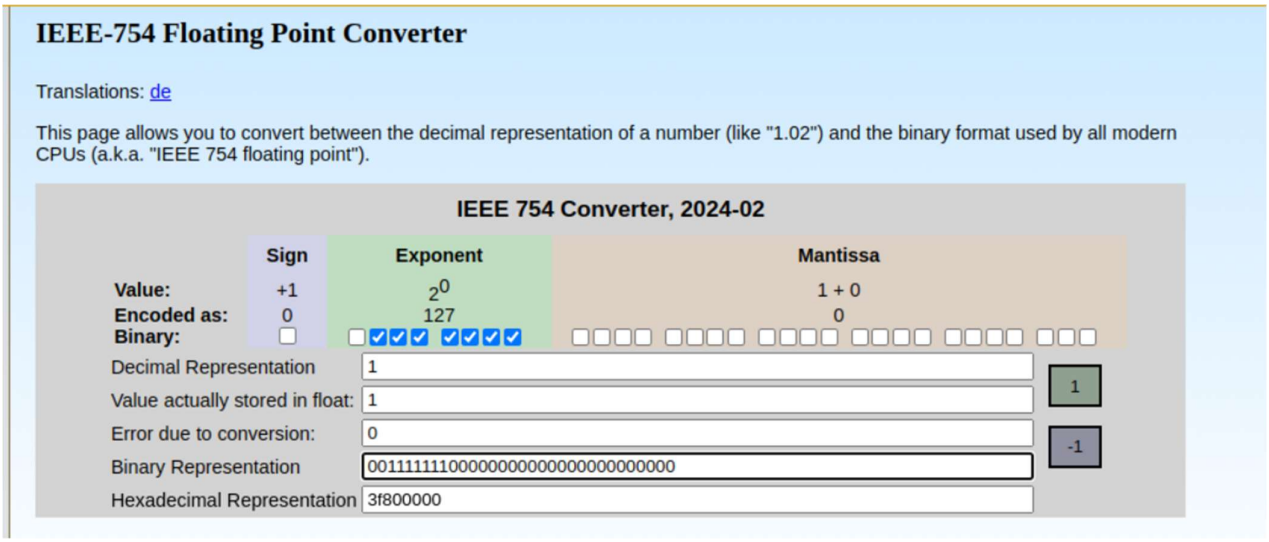
3. 0.5 * (-1.25) = -0.625 (10111111001000000000000000000000)



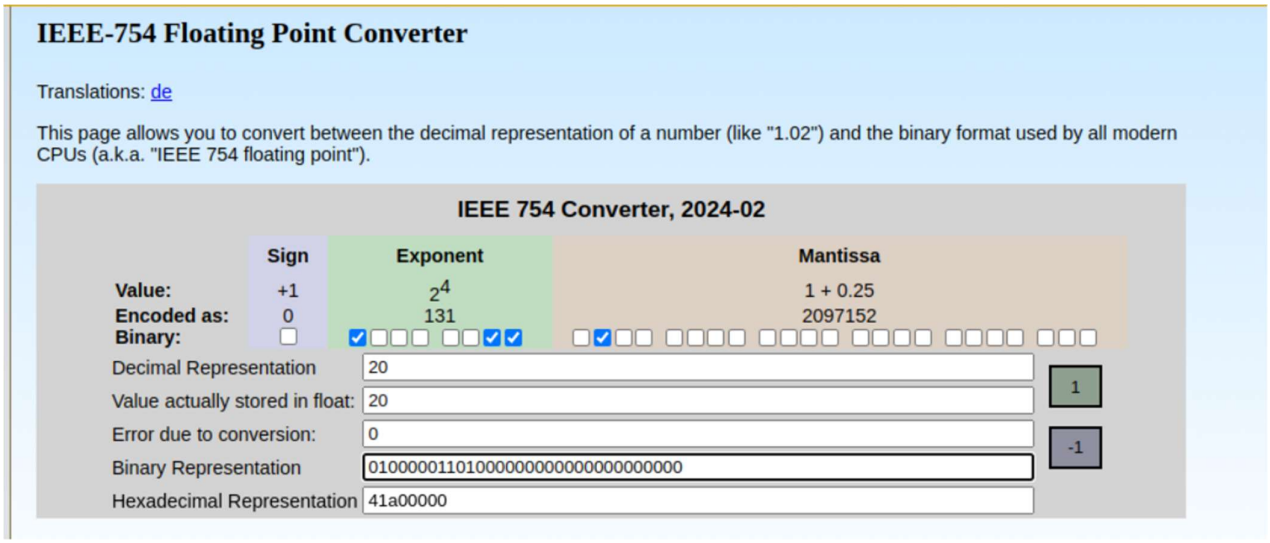
4. (-6.5) * (-2.0) = 13



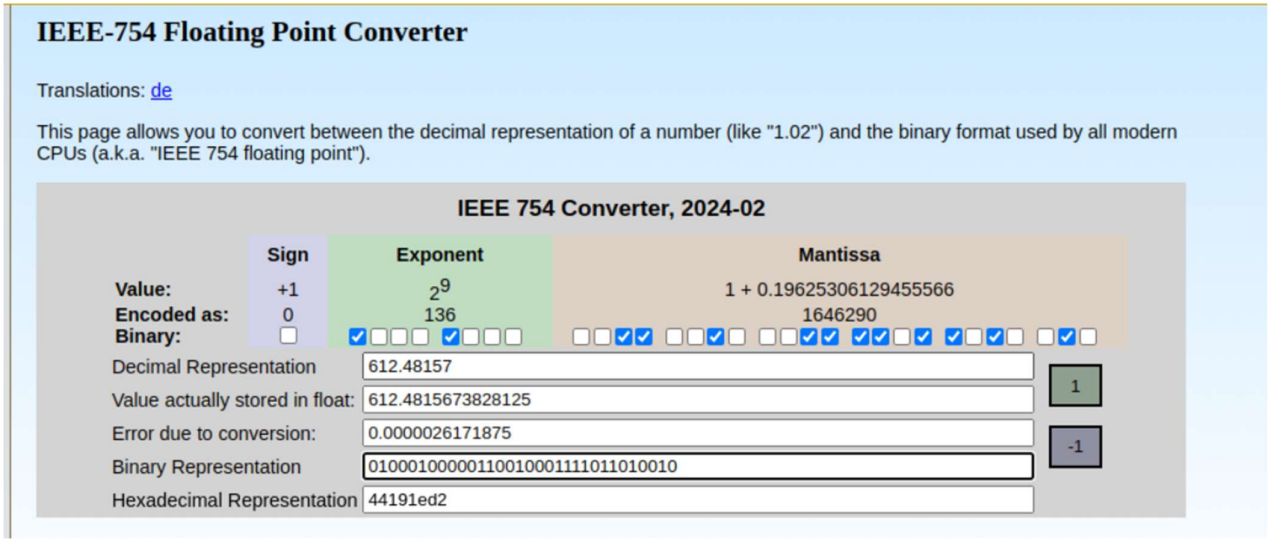
5. 10.0 * 0.1 = 1 (00111111100000000000000000000000)



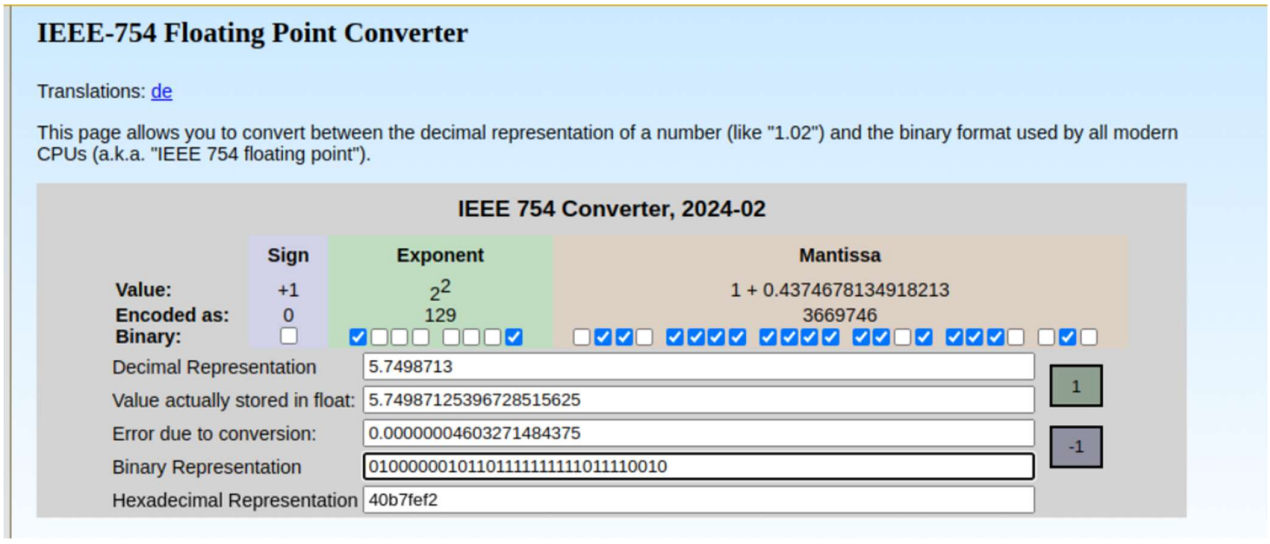
6. 4 * 5 = 20 (01000001101000000000000000000000)



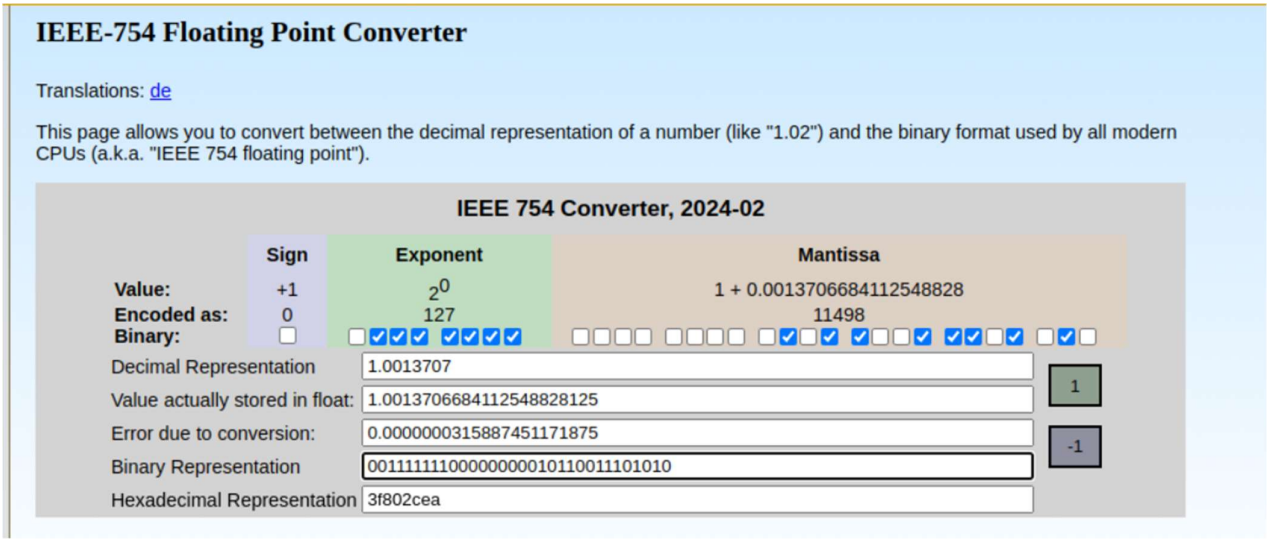
7. 6.7865 * 90.25 = 612.48157 (01000100000110010001111011010010)



8. 1.236 * 4.652 = 5.74987 (01000000101101111111111011110010)



9. 0.0987 * 10.1456 = 1.0013707 (00111111100000000010110011101010)



IEEE-754 Floating Point Converter

Translations: de

This page allows you to convert between the decimal representation of a number (like "1.02") and the binary format used by all modern CPUs (a.k.a. "IEEE 754 floating point").

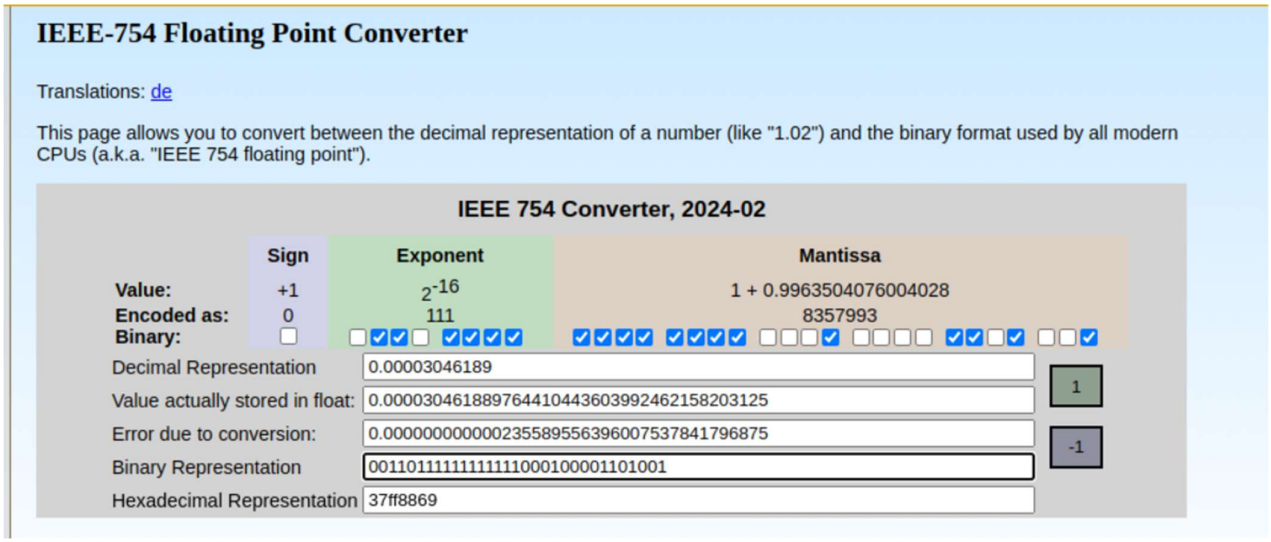
IEEE 754 Converter, 2024-02

	Sign	Exponent	Mantissa
Value:	+1	2^{0}	1 + 0.0013706684112548828
Encoded as:	0	127	11498
Binary:	☐	☑☑☑☑☑☑☑☐	☐☐☐☐☐☐☐☐☑☐☑☐☑☐☐☑☑☑☑☐☑☐

Decimal Representation	1.0013707
Value actually stored in float:	1.001370668411254828125
Error due to conversion:	0.000000031588745117187 5
Binary Representation	00111111100000000010110011101010
Hexadecimal Representation	3f802cea

1 -1

10. 0.667 * 0.00004567 = 0.00003046189 (00110111111111111100010000110100 1)



IEEE-754 Floating Point Converter

Translations: de

This page allows you to convert between the decimal representation of a number (like "1.02") and the binary format used by all modern CPUs (a.k.a. "IEEE 754 floating point").

IEEE 754 Converter, 2024-02

	Sign	Exponent	Mantissa
Value:	+1	2^{-16}	1 + 0.9963504076004028
Encoded as:	0	111	8357993
Binary:	☐	☐☑☑☐☑☑☑☑	☑☑☑☑☑☑☑☑☐☐☐☑☐☐☐☐☑☑☐☑☐☐☑

Decimal Representation	0.00003046189
Value actually stored in float:	0.000030461889764410443603992462158203125
Error due to conversion:	0.000000000000235589556396007537841796875
Binary Representation	00110111111111111100010000110100 1
Hexadecimal Representation	37ff8869

1 -1

SYNTHESIS:

UTILIZATION OF BLOCKS FOR NON-PIPELINED:

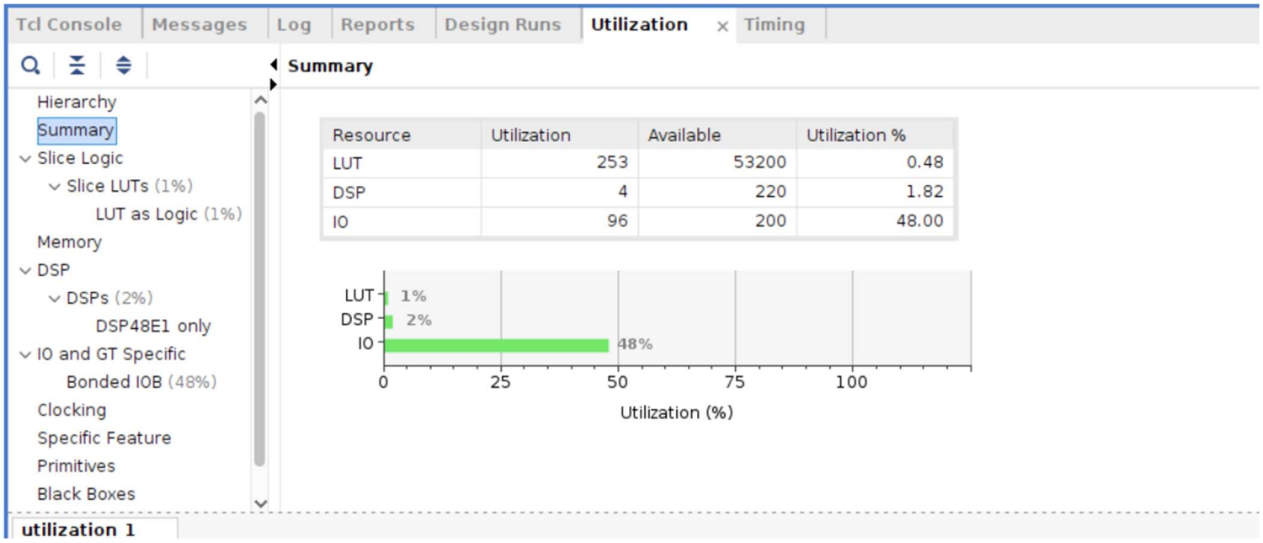


Figure non-pipelined design utilization summary

UTILIZATION OF BLOCKS FOR PIPELINED DESIGN WITH KARATSUBA MULTIPLIER:

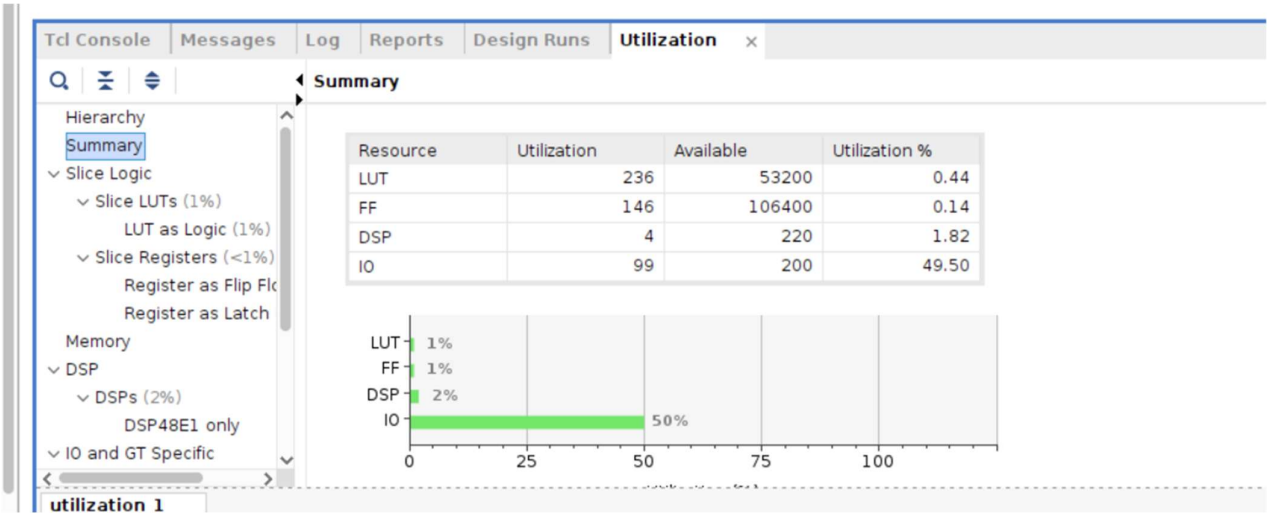


Figure pipelined design utilization summary with karatsubamultiplier

TIMING REPORT WITH USAGE OF KARATSUBA MULTIPLIER:

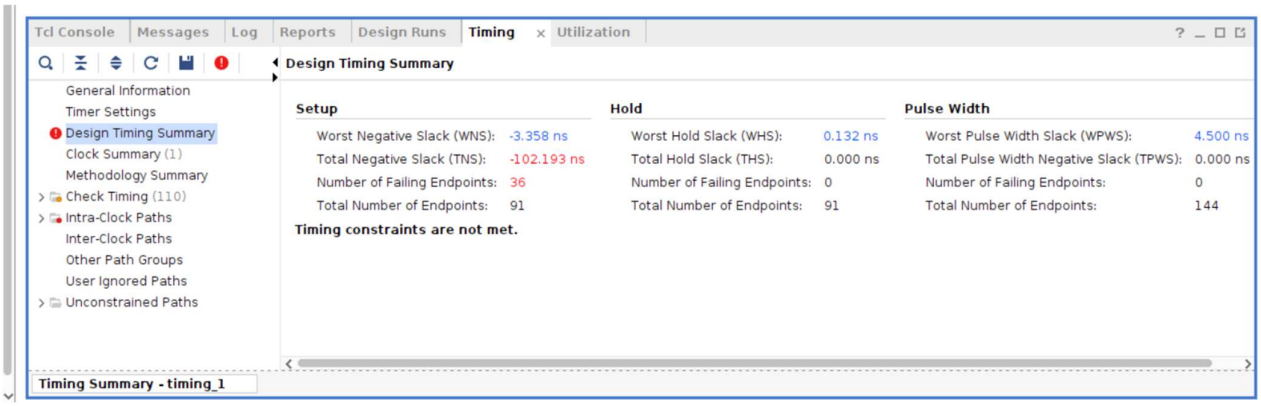


Figure Timing summary with usage pipelined and Karatsuba multiplier

UTILIZATION OF BLOCKS WITHOUT KARATSUBA MULTIPLIER:

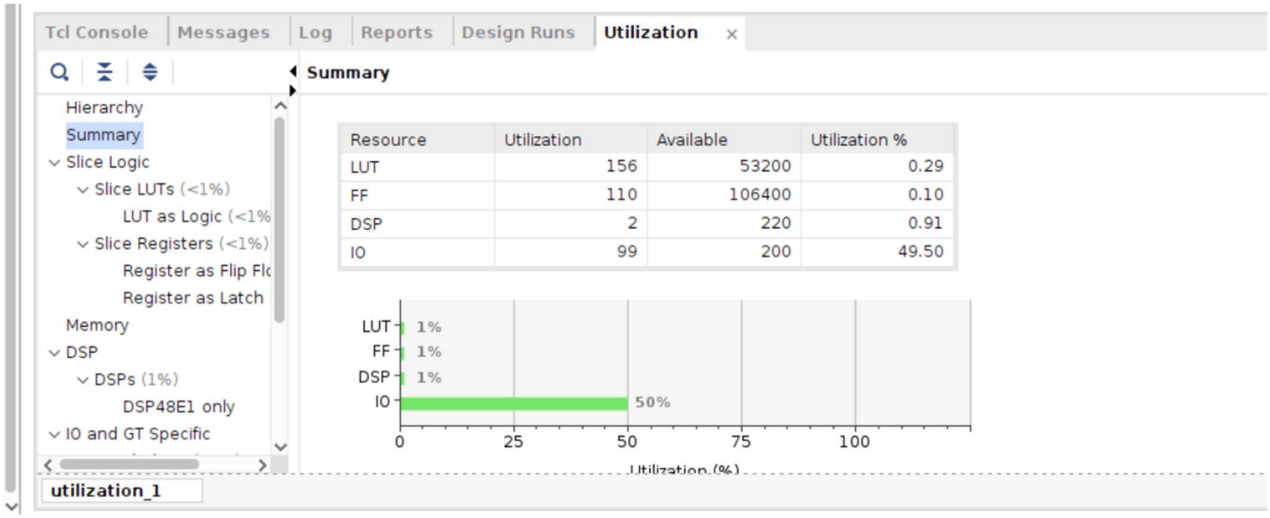


Figure pipelined design utilization summary without Karatsuba multiplier

TIMING REPORT WITHOUT USAGE OF KARATSUBA MULTIPLIER:

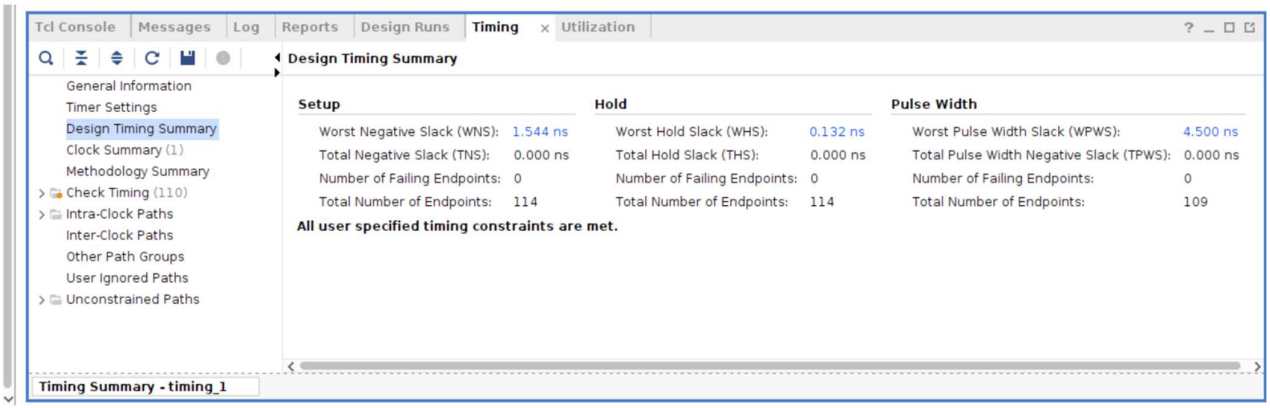
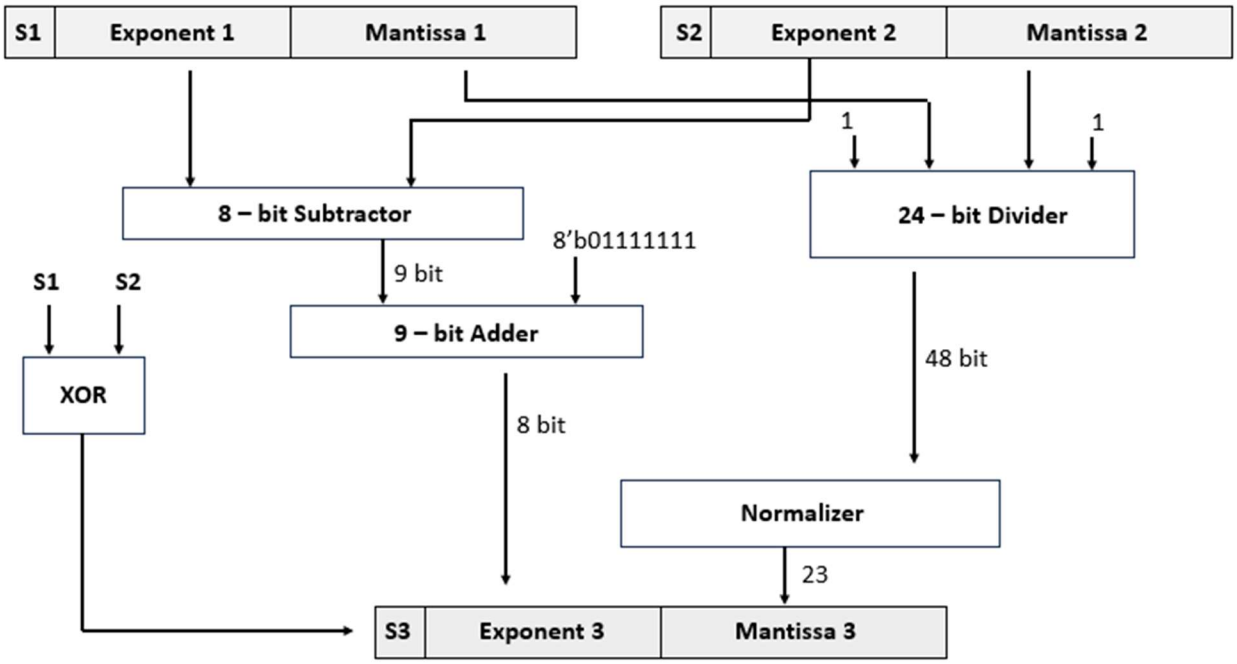


Figure pipelined design timing summary without Karatsuba multiplier

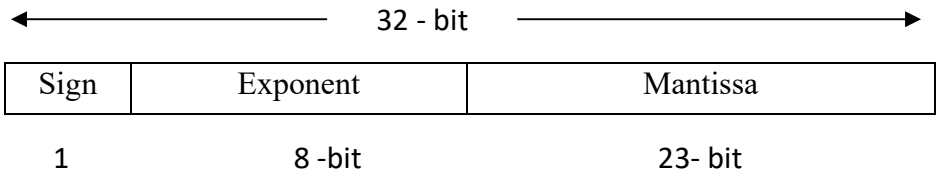
DIVISION

Architecture :



The above diagram represents the architecture diagram of 32- bit single precision floating point numbers.

Input format :



S1, S2: These are the sign bits of the two input floating-point numbers.

Exponent 1, Exponent 2: The exponents of the two floating-point numbers (8 bits each).

Mantissa 1, Mantissa 2: The mantissas (or significands) of the two numbers (23 bits each, without an implicit leading 1).

Architecture Components :

1. **XOR - Gate**
    - The signs of the input numbers (S1, S2) are XORed to compute the sign of the result (S3). This is because if anyone of the numerator or denominator is negative the resultant sign is negative(1).

$$\text{Sign} = S1 \text{ xor } S2$$

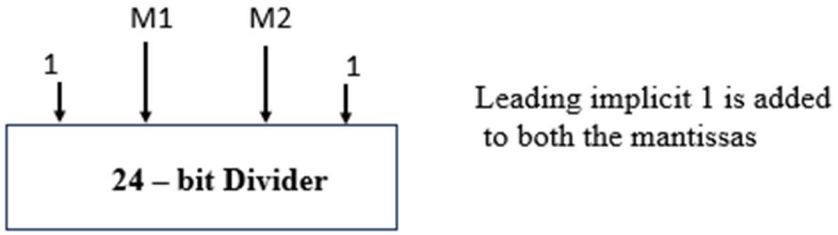
2. **Exponent Difference (8-bit Subtractor)**
    - The exponents of the two input numbers are subtracted using an 8-bit subtractor:

$$\text{Exponent Difference} = \text{Exponent 1} - \text{Exponent 2}$$

3. **Bias Adjustment (9-bit Adder)**
    - The IEEE 754 standard uses a bias value of 127 for single-precision floating-point numbers. To adjust for this, a bias value of 8′b011111111 (127 in binary) is added back to the result of the exponent subtraction using a 9-bit adder.

4. **Mantissa Division (24-bit Divider)**
    - The mantissas (with an implicit leading 1) are fed into a 24-bit divider to compute the division.

$$\text{Mantissa Result} = \text{Mantissa 1} / \text{Mantissa 2}$$



    - The output of the division is a 48-bit result, which includes the quotient and any additional precision.

5. **Normalization**
    - The division result may need normalization to ensure it adheres to the IEEE 754 format, which requires the mantissa to have values after a leading 1.
    - The Normalizer adjusts the mantissa and modifies the exponent accordingly to maintain the correct representation.

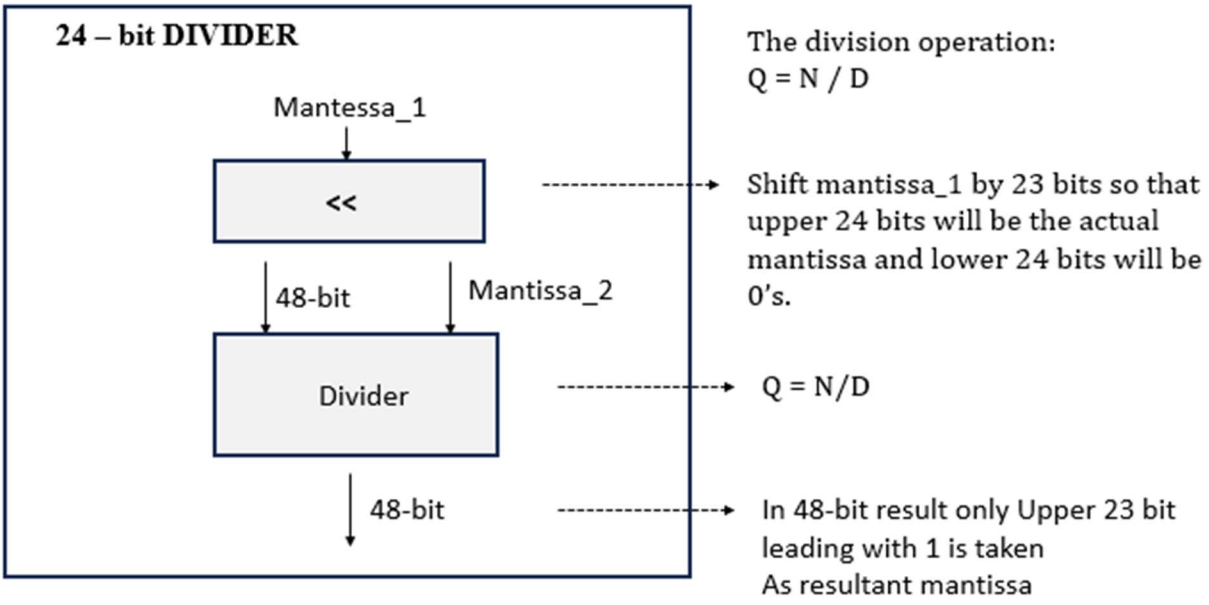
Output

- **S3**: The sign of the result, determined from the XOR of the input signs.
- **Exponent 3**: The final exponent, calculated as
  Exponent 3=Normalized Exponent Difference
- **Mantissa 3**: The normalized mantissa, extracted from the 48-bit division result.

Design Implementation :

An 8 – bit subtractor is used for calculating the exponent difference for faster computation. Then the 9-bit result from subtractor is send adder to add bias value (127) to the exponent.

The partitioned Mantissa is fed into a 24-bit divider to perform division operation. The divider consist of a 23 – bit binary shifter so 48-bit mantissa_1 (numerator) is divided with mantissa_2 (denominator) get 48 – bit result.



In IEEE 754 single-precision floating-point format, the mantissa is a normalized 23-bit fraction (with an implicit leading 1). The complete mantissa is represented as a value between 1.0 and 2.0 in binary, where the leading 1 is implicit (not stored).

When you perform the division of two normalized floating-point numbers, the result of the division between two mantissas will also be a normalized value. This result can be a value that has more than 23 bits of precision (because of division), which is why the shift by 23 bits in mant_float_num1 is needed to align it with the precision of the other mantissa (mant_float_num2).

However, as you correctly mentioned, shifting a 24-bit number by 23 bits causes the upper 23 bits to be discarded, leaving only the least significant bits. This does result in precision loss.

Shifting 24 bits to the left by 23 places essentially means you're multiplying the number by 2^{23}, but this is done in a way where the left most bits (which contain higher order bits) are discarded.

Example :

mant_float_num1 = 1.101010... (24 bits)

mant_float_num1 << 23 --> 101010...000 (24-bits shifted left)

The shifted value is now a 47-bit number, but only the lower 24 bits of the result are kept after the division, effectively truncating the higher-order bits. The division result is stored in mant_temp, which is 48 bits wide to store any possible overflow from the division.

After the division, you extract the lower 23 bits of the mant_temp because these represent the most accurate part of the division result. The upper bits are discarded to ensure that the result fits within the single-precision floating-point format.

What happens if you consider upper 23-bit for result?

The upper 23 bits of the result instead of the lower 23 bits would lead to:

- Loss of Precision in the Result:
- Incorrect Normalization or Scale:
- Potential for Inaccurate Results:

The reason lower 23 bits are selected in your code is because:

- They preserve the finer fractional details of the result, which are more significant in floating-point arithmetic.

- When you discard the least significant bits (upper 23 bits), you risk losing important fractional information, which could lead to inaccuracies.
- By keeping the lower 23 bits, you are effectively keeping the more accurate part of the result and ensuring the number stays correctly normalized.

After division, the mantissa may need to be normalized. If the most significant bit of mant_result is 0, it means that the mantissa is smaller than 1, so it must be shifted left by 1 bit to restore it to the normalized form (where the leading bit is 1).

When the mantissa is shifted left, the exponent is decremented by 1 (exp_result = exp_result - 1).

The final result is assembled by concatenating the sign, exponent, and mantissa:

result_out = {sign, exp_result, mant_result[22:0]}.

The sign is stored in the most significant bit, followed by the 8-bit exponent, and the 23-bit mantissa.

CODE EXPLANATION :

```
module division_fpu (
        input [31:0] float_num1,                // IEEE 754 Single Precision Floating Point
        input [31:0] float_num2,
        output reg [31:0] result_out       );
```

The Division takes two 32-bit inputs (which are IEEE 754 single-precision floating-point numbers) and produces a 32-bit output that will also be a single-precision floating-point result of the division.

```
reg sign;
reg [7:0] exp_float_num1, exp_float_num2, exp_result;
reg [23:0] mant_float_num1, mant_float_num2, mant_result;
reg [47:0] mant_temp;
```

Register Declarations

- Sign : A register that holds the sign bit of the result.
- exp_float_num1, exp_float_num2, exp_result : Registers to store the exponents of float_num1, float_num2, and the result, respectively.
- mant_float_num1, mant_float_num2, mant_result : Registers to store the mantissas (significand) of the numbers involved.
- mant_temp : A temporary register used to store the result of mantissa multiplication/division (since the result can be large, 48 bits are used).

```
always @(*)
begin
if (float_num2 == 32'b0) begin
        result_out = 32'b0;              // Handle divide by zero (could be Inf or NaN)
end
else if (float_num1 == 32'b0) begin
        result_out = 32'b0;    // 0 divided by anything is 0 valid = 1;
end
```

Division by Zero Check:

- Division by Zero (float_num2 == 32'b0):
    - If the second input (float_num2) is zero, it will result in an invalid operation. In that case, the result_out is set to zero, and valid is set to 1 (indicating the result is available).
- Zero Dividend (float_num1 == 32'b0):
    - If the first input (float_num1) is zero, the result of dividing zero by anything is zero. So, the result is set to zero and valid is set to 1.

```
else begin                                          // Extract sign, exponent, and mantissa
        sign = float_num1[31] ^ float_num2[31];
        exp_float_num1 = float_num1[30:23];
        exp_float_num2 = float_num2[30:23];
         mant_float_num1 = {1'b1, float_num1[22:0]};       // Implicit leading 1
         mant_float_num2 = {1'b1, float_num2[22:0]};
```

Extraction of resultant_sign, exponent and mantissa is done in this step.

```
exp_result = exp_float_num1 - exp_float_num2 + 127;
```

Exponents are passed to 8- bit subtractor and result is passed to adder as shown in the architecture where exponent difference is added with bias(which is 127 or 8'b01111_1111 for 32-bit single precision format)

```
// Divide mantissas
 mant_temp = (mant_float_num1 << 23) / mant_float_num2;
mant_result = mant_temp[23:0];
```

Division operation on Mantissa is performed and result of the mantissa division is stored in mant_result, which is a 24-bit value.

```
// Normalize if necessary
 if (mant_result[23] == 0) begin
        mant_result = mant_result << 1;
        exp_result = exp_result - 1;
end
```

After division, the mantissa may need to be normalized. If the most significant bit of mant_result is 0.

```
// Assemble final result
        result_out = {sign, exp_result, mant_result[22:0]};
    end
end
 endmodule
```

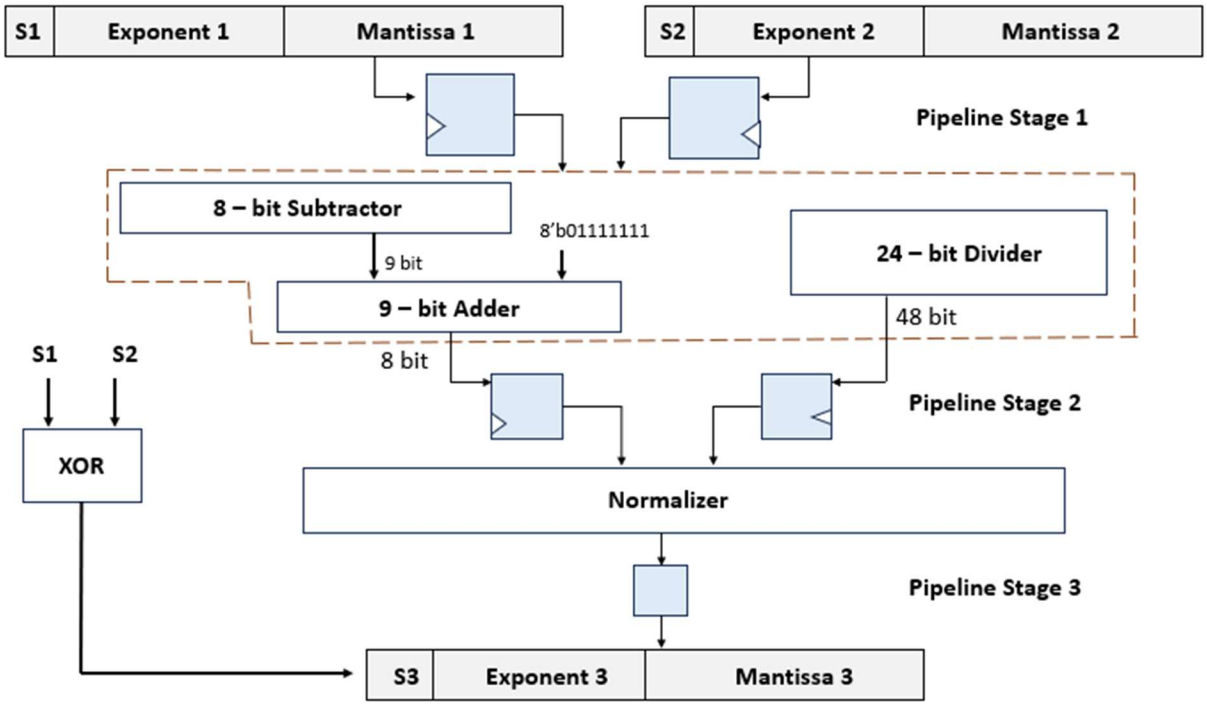
The final result is assembled by concatenating the sign, exponent, and mantissa.

The sign is stored in the most significant bit, followed by the 8-bit exponent, and the 23-bit mantissa.

Pipelining Design :

The timing behavior of design is not as intended when the design is fully combinational. It produces a negative slack, which effect the functionality of the design. So, the design is pipelined by adding registers at 3 stages which dramatically reduced the maximum negative slack zero.

The below diagram shows the pipelined architecture of design

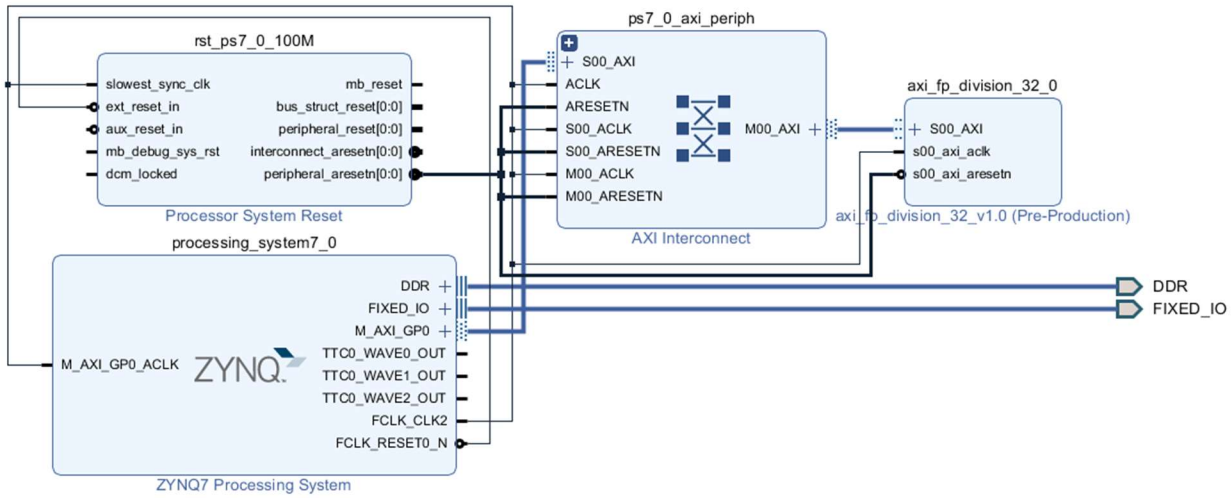


Each pipeline stage has a separate enable signal, which is asserted using a finite state machine.

The pipelined the architecture now it is implemented with AXI IP so pipelined design to communicate with Zed board through UART.

AXI IP Integration:

- The pipelined FPU is encapsulated into an AXI-compliant IP core (axi_fp_division_32_0), which enables it to communicate effectively with the Zynq Processing System (PS) over the AXI interface.
- This approach simplifies the connection between the custom hardware and the PS, leveraging standard AXI interfaces for configuration, data transfer, and status monitoring.



System-Level Integration:

- The Zynq Processing System (represented as processing_system7_0) is configured with the necessary peripherals and memory interfaces (DDR and Fixed IO). The system clock and reset signals are managed by the Processor System Reset block (rst_ps7_0_100M).
- The AXI Interconnect (ps7_0_axi_periph) facilitates communication between the Zynq PS and the custom AXI IP core. It ensures proper arbitration and data transfer between master and slave components on the AXI bus.

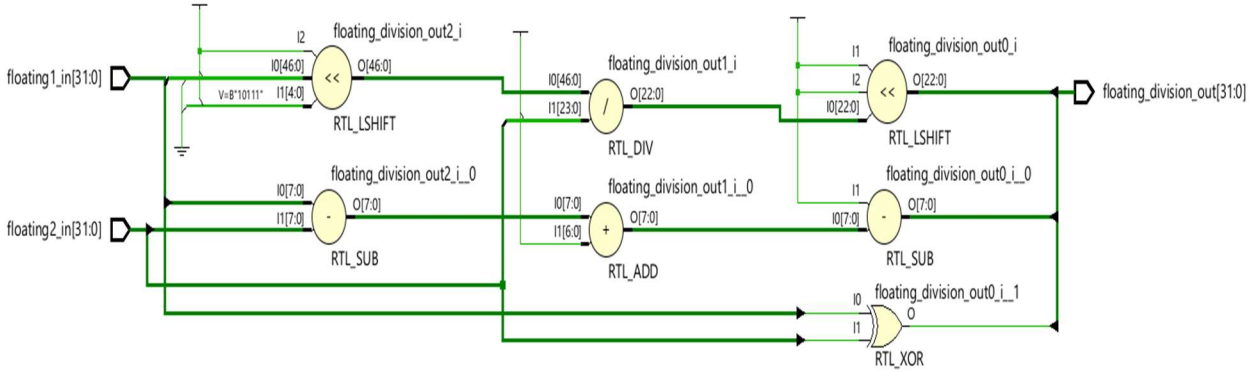
Communication with ZedBoard:

- For external communication, only the UART interface is configured, allowing the ZedBoard to exchange data with the system's CPU. This decision simplifies the hardware interface, reducing pin usage and focusing on serial communication for debugging and data interaction.

Data Flow:

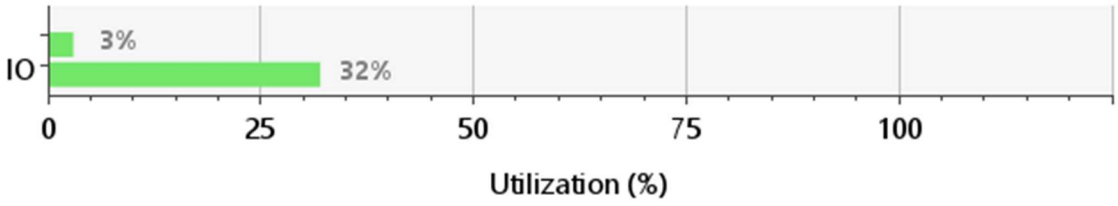
- The Zynq Processing System generates data and control signals, which are transmitted to the custom FPU through the AXI interface.
- The FPU processes the input data based on its pipelined architecture and returns the computed results via the same AXI pathway.
- The processed data is then sent to the external system via UART.

ELOBARATED DESIGN:

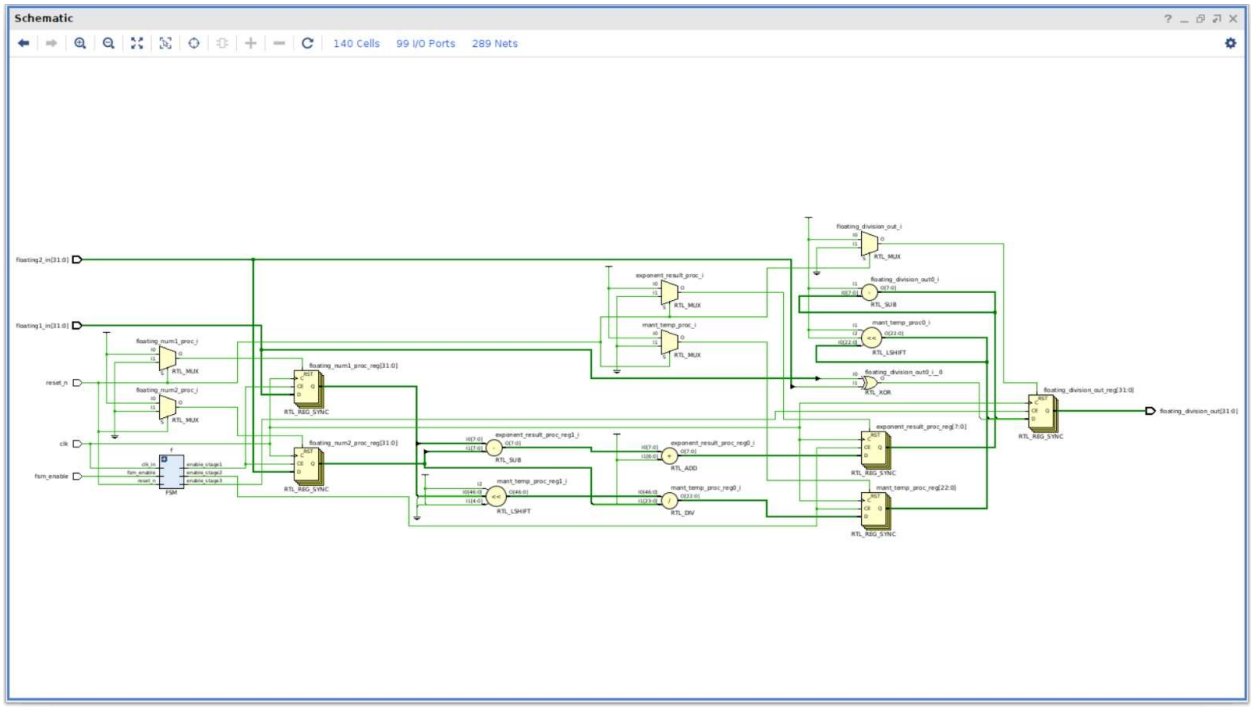


UTILIZATION SUMMARY:

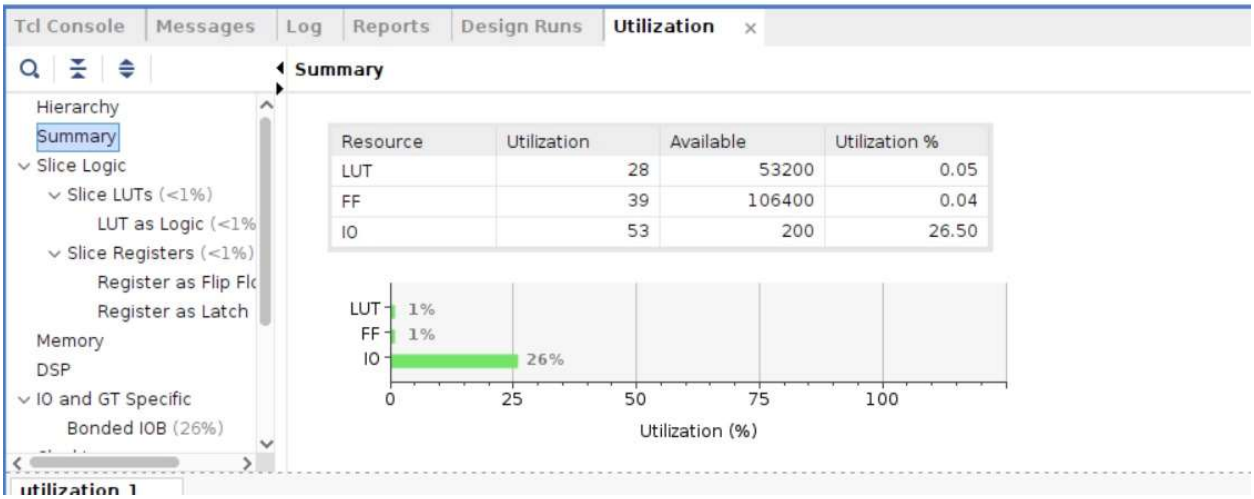
Resource	Utilization	Available	Utilization %
LUT	1181	41000	2.88
IO	96	300	32.00



ELOBARATED DESIGN(PIPELINED):



UTILIZATION SUMMARY(PIPELINED):



RESULTS :

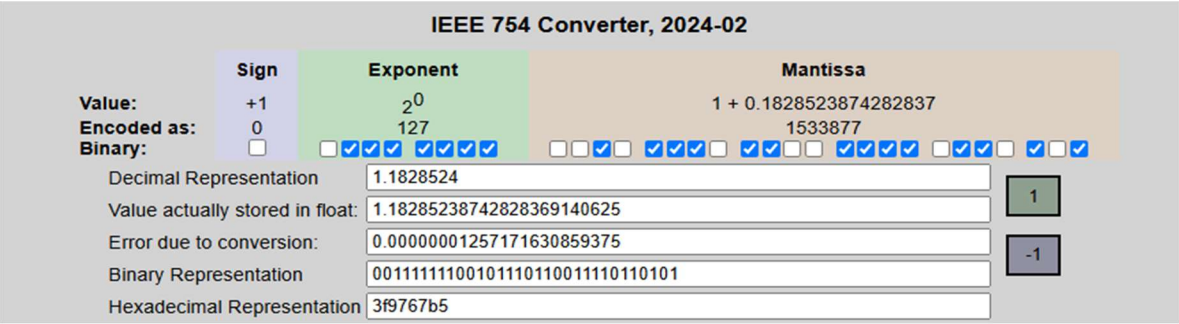
```
Vivado Simulator 2019.1
Time resolution is 1 ps
Test 1: 01000000010010001111010111000010 / 01000000001010011110010011110111 = 00111111100101110110011110110101
Test 2: 01000001010010001111010111000011 / 00111111100000000000000000000000 = 01000001010010001111010111000011
Test 3:  00111111000011011101001011110010 / 01000001000001011001100110011010 = 00111101100001111111000010011000
Test 4:  01000011011111110010111101110110 / 11000010011010010000110001100100 = 11000000100011000010100010011000
Test 5:  00111111100000000000000000000000 / 10111110011000101101011000100100 = 11000001010010000011101001101101
Test 6:  01000001010010001111010111000011 / 10111111100000000000000000000000 = 11000001010010001111010111000011
Test 7:  01000000111101111101011100001010 / 00000000000000000000000000000000 = 00000000011101111101011100001010
Test 8:  01000011100000000000000000000000 / 01000001100000000000000000000000 = 01000001100000000000000000000000
```

Test Case 1 :

float_num1 = 32'b0_1000_0000_10010001111010111000010; // **3.14**

float_num2 = 32'b0_1000_0000_01010011110010011110111; // **2.6546**

Actual Result : 1.1828524

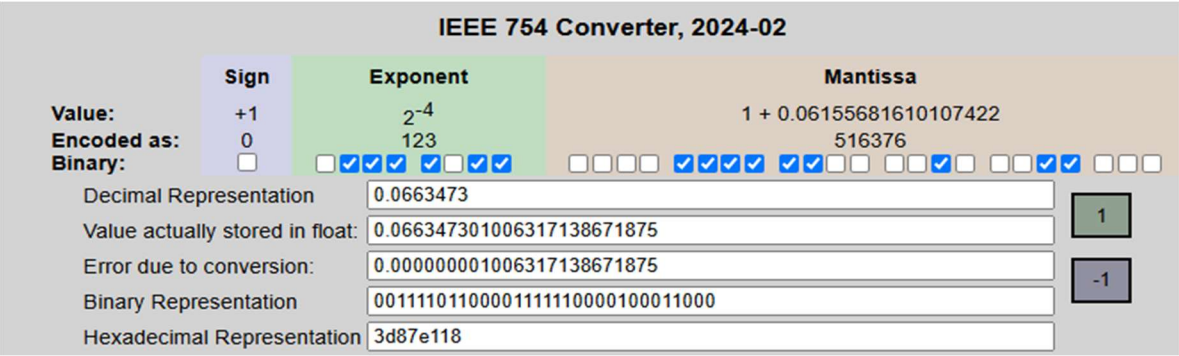


Test Case 2 :

float_num1 = 32'b00111111000011011101001011110010; // **0.554**

float_num2 = 32'b00111111000101100100111000100111; // **8.35**

Actual Result : 0.066347

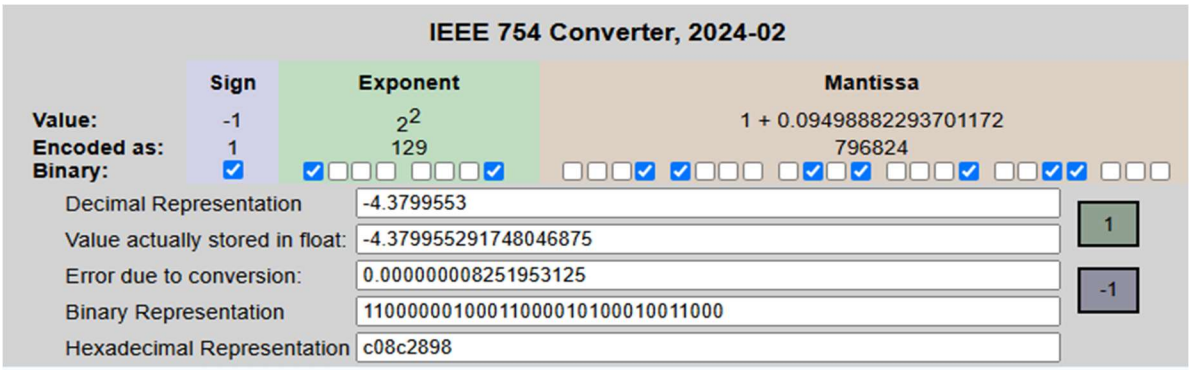


Test Case 3 :

float_num1 = 32'b01000011011111110010111101110110; // **255.1854**

float_num2 = 32'b11000010011010010000110001100100; // **-58.2621**

Actual Result : 4.3799

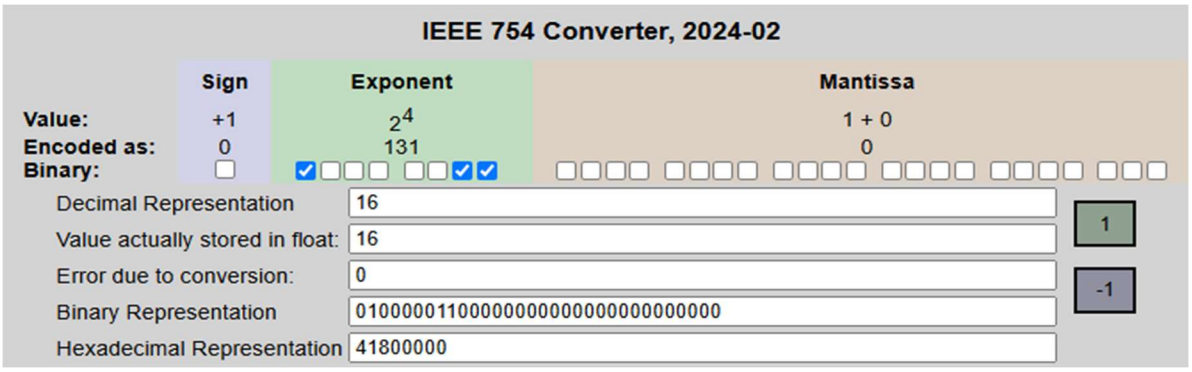


Test Case 4 :

float_num1 = 32'b0_10000111_00000000000000000000000; // **256**

float_num2 = 32'b0_10000011_00000000000000000000000; // **16**

Actual Result : 16



Test Case 5 :

float_num1 = 32'b01000000111101111101011100001010; // **7.745**

float_num2 = 32'b00000000000000000000000000000000; // **0.0**

Actual Result : Infinity