HO CHI MINH UNIVERSITY OF TECHNOLOGY

ELECTRICAL ELECTRONIC ENGINEERING

Electronic Department

----------o0o----------

LOGIC DESIGN / LOGIC SYNTHESIS

32-bit Floating Point Unit (FPU)

INSTRUCTORS: **PhD. Tran Hoang Linh**

Nguyen Tuan Hung

STUDENT **:** **Ha Gia Huy _ 2051117**

Tran Minh Tri _ 2051022

HO CHI MINH CITY, MAY, 2024

CATALOG

I. Introduction

1.1. IEEE 754 Floating-Point Data Formats

The IEEE 754 Standard defines mainly two floating-point formats, float and double, which we commonly use in software development. Suppose that in a C program we declare three variables: k, s, and x, as shown below:

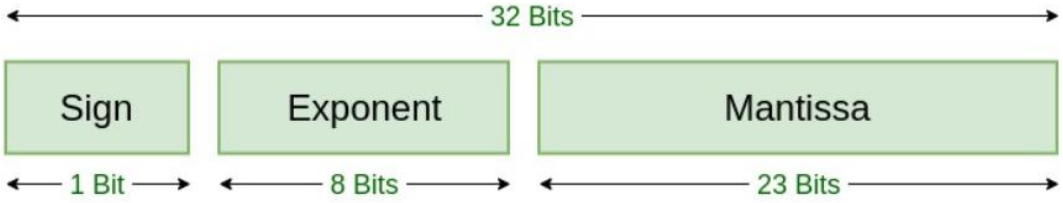
```
int k = -1;
float s = -1.75;
double x = 1.75;
```

Then a compiler will convert them to the following binary patterns: (Ignore the underbars that divide a binary pattern into three parts for easy reading.)

Variable	Binary pattern	Bits
k:	11111111111111111111111111111111	32
s:	1_01111111_11000000000000000000000	32
x:	0_01111111111_1100000000000000000000000000000000000000000000000000	64

1.2. Overview of sign, exponent, mantissa

The float s = -1.75 is converted to a 32-bit pattern in the IEEE 754 single-precision floating-point format. Referring to Figure, an IEEE 754 single-precision floating-point number consists of three parts: s, e, and f.



Single Precision
IEEE 754 Floating-Point Standard

1.3. Special values

The 1-bit s (sign) indicates the sign of the number: s = 1 for a negative number and s = 0 for a nonnegative number. The 8-bit e (exponent) is the biased exponent with the bias of 127. The 23-bit f (fraction) is the significand (mantissa). The value (V) of a single-precision floating-point number is defined by IEEE as follows:

- Normalized : If $0 < e < 255$, then $V = (-1)^s \times 1.f \times 2^{e-127}$;
- +0, −0 : If $e = 0$ and $f = 0$, then $V = (-1)^s \times 0$;
- Denormalized : If $e = 0$ and $f \neq 0$, then $V = (-1)^s \times 0.f \times 2^{-126}$;
- +∞, −∞ : If $e = 255$ and $f = 0$, then $V = (-1)^s \infty$;
- NaN (not a number): If $e = 255$ and $f \neq 0$, then V = NaN.

1.4. Normalization and denormalization of floating point values

For a normalized number, the 1 of 1. f is not stored in the 32-bit data; it is called a hidden bit. For a denormalized number, the hidden bit is a 0. Some examples of the single-precision floating-point numbers are shown below.

- Normalized numbers ($V = (-1)^s \times 1.f \times 2^{e-127}$):

  1 01111110 10000000000000000000000 = −1.1 × 2126−127 = −0.75

  0 00000001 00000000000000000000000 = +1.0 × 21−127 = +2−126

  0 11111110 00000000000000000000000 = +1.0 × 2254−127 = +2127

  0 11111110 11111111111111111111111 = +(2 − 2−23) × 2254−127

- Zero, infinity, and NaN:

  0 00000000 00000000000000000000000 = +0

  1 00000000 00000000000000000000000 = −0

  0 11111111 00000000000000000000000 = +∞

  1 11111111 00000000000000000000000 = −∞

  0 11111111 10000000000000000000000 = NaN

  1 11111111 00000100001100000000000 = NaN

- Denormalized numbers ($V = (-1)^s \times 0.f \times 2^{-126}$):

  0 00000000 10000000000000000000000 = $+0.1 \times 2^{-126} = +2^{-127}$

  0 00000000 00000000000000000000001 = $+2-23 \times 2^{-126} = +2^{-149}$

The following steps show how to convert a real number in decimal format to the IEEE 754 single-precision floating-point format.

$V = -6.25_{10} = -(6_{10} + 0.25_{10}) = -(110_2 + 0.01_2) = -110.01_2 = -1.1001_2 \times 2^2$

$= -1.1001_2 \times 2^{129-127} = (-1)^s \times 1.f \times 2e^{-127}$

II. DESIGN STRATEGY

2.1. Implementation General of Top Module

Interface

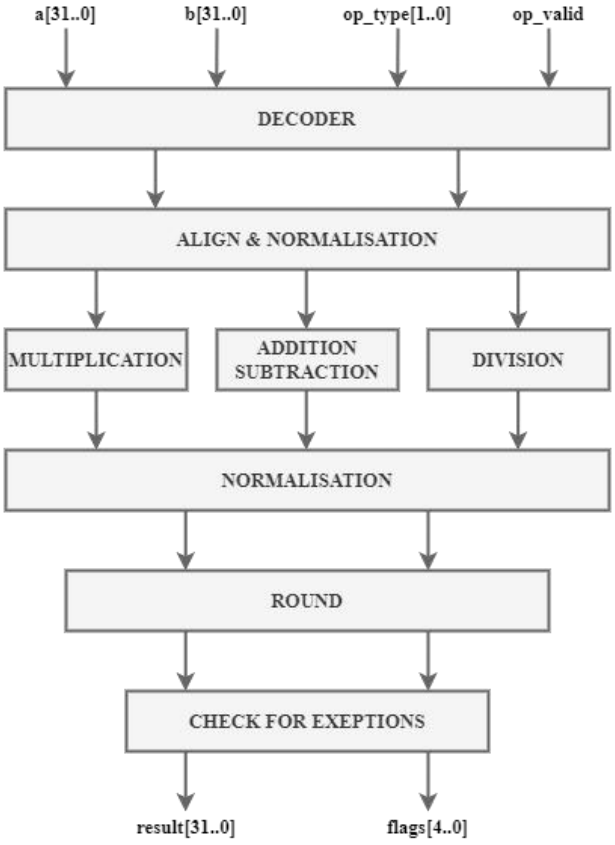
- Two 32-bit input ports for operands
- One 32-bit output port for result

Input signals to specify:

- Operation (+, −, *, /)
- Output signals for exception flags

Implementation Guidelines

- Decode inputs into IEEE 754 format
- Align and normalize mantissas
- Perform arithmetic on mantissas
- Normalize result mantissa
- Round mantissa to nearest value by:
  + Truncating LSB if 0
  + Adding 1 and truncating if LSB is 1
- Check for exceptions
- Encode result into IEEE 754 format



Base on the above implementation guidelines, we have the overall flowchart as below. In this lab, we follow this flowchart to construct, verify and test our FPU. However, this is just in general. For details, we would like to discuss in the following session with each modules.

2.2. Implementation Details of Hierarchy Modules

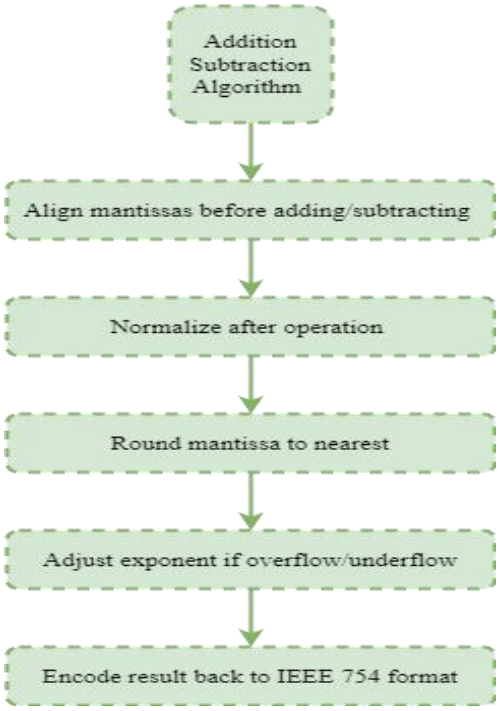
2.2.1. Floating-Point Adder (FADD) Design

❖ **Floating-Point Addition Algorithm**

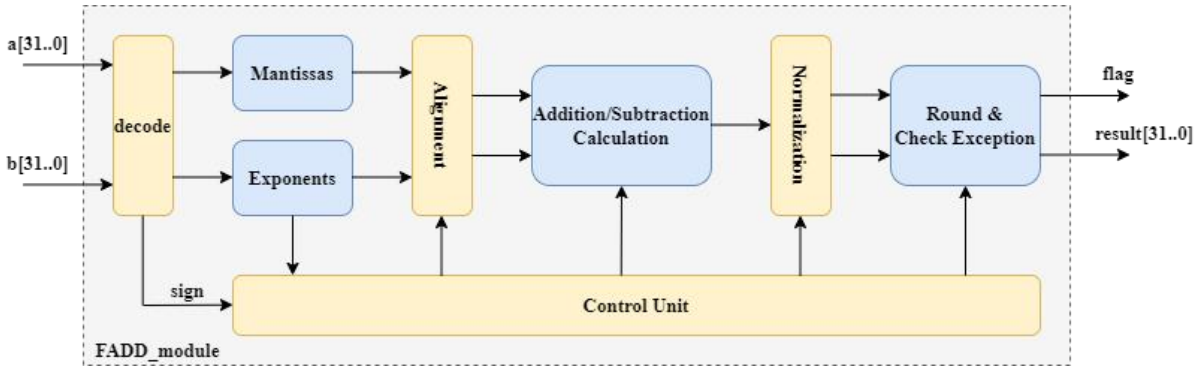
The algorithm for floating point addition is explained by the flow chart & diagram. While adding the two floating point numbers, two cases may arise.

Case I: when both the numbers are of the same sign i.e. when both the numbers are either +ve or –ve. In this case, MSB of both the numbers are either 1 or 0.

Case II: when both the numbers are of different sign i.e. when one number is +ve and other number is –ve. In this case, MSB of one number is 1, the other is 0.



❖ **Floating-Point Addition Architecture**

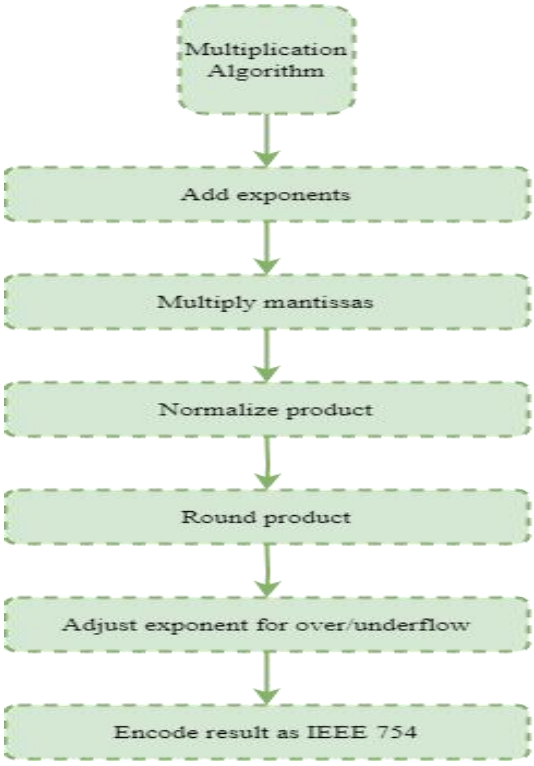


Decode the IEEE 754 formatted inputs into sign, exponent, and mantissa fields. Shift the mantissas to align them based on the exponent difference. The mantissa with the smaller exponent needs to be shifted right by the difference in exponents. For addition, add the aligned mantissas. For subtraction, subtract the smaller mantissa from the larger mantissa. Normalize the result mantissa by left shifting until the MSB is 1. Adjust the exponent accordingly. Round the mantissa to the nearest value by adding 1 to the LSB if it is 1. Truncate the mantissa if LSB is 0. Check for overflow or underflow based on the exponent range. Also check for inexact rounding. Encode the sign, updated exponent, and rounded mantissa into the IEEE 754 output format.

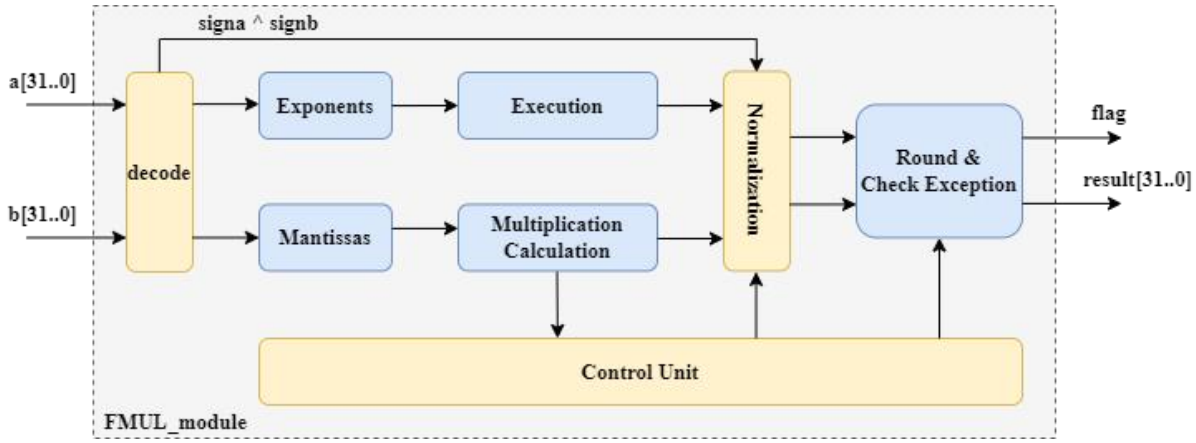
2.2.2. Floating-Point Multiplier (FMUL) Design

❖ **Floating-Point Multiplication Algorithm**

The result of multiplication is a negative sign, if one of the multiplied numbers is a negative value, by XORing sign of two inputs. Exponent addition is done through unsigned adder for adding the exponent of the first input to the exponent of the second input and after that subtract the Bias (127) from the addition result. Significand multiplication is done for multiplying the unsigned significand and placing the decimal point in multiplication product. The unsigned significand multiplication done on 24 bits.



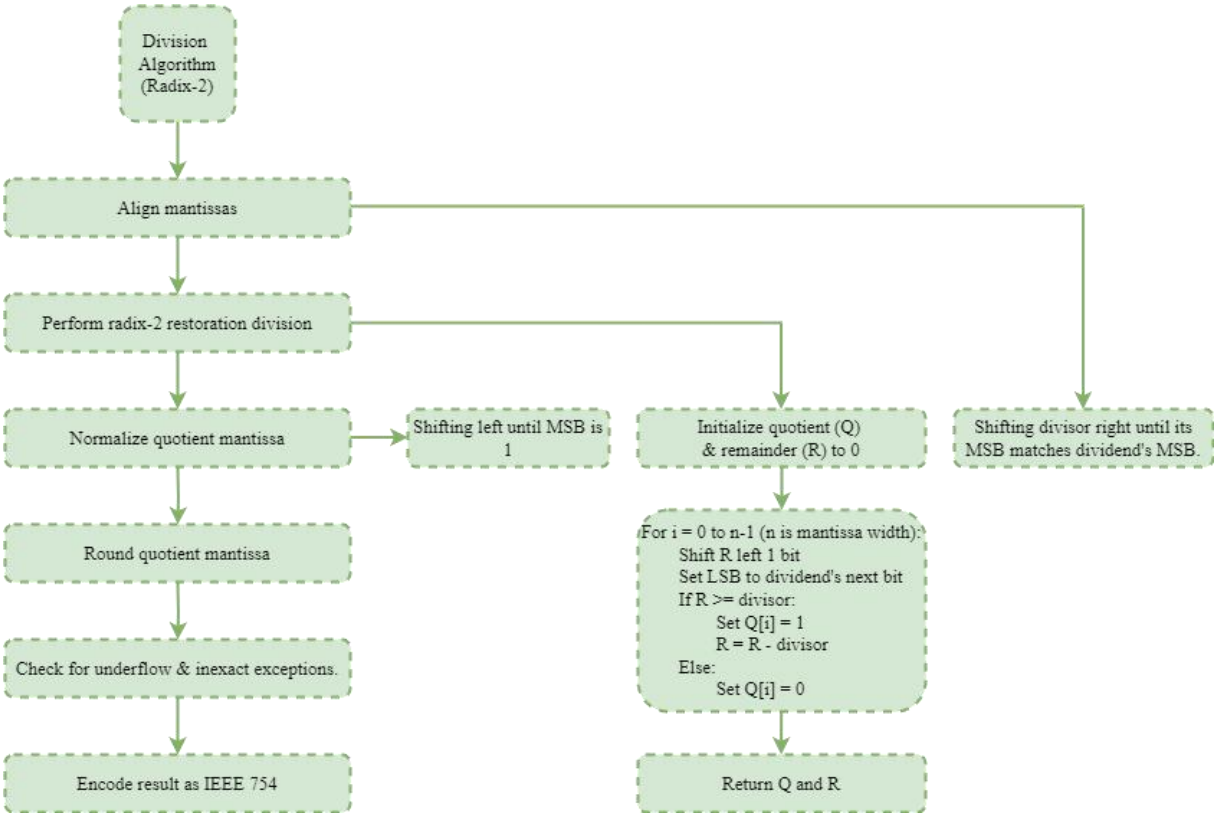
❖ **Floating-Point Multiplication Architecture**



Decode the IEEE 754 formatted inputs into sign, exponent, and mantissa. Multiply the mantissas to generate a product. Add the exponents of the two inputs to get the exponent for the result. Normalize the product mantissa by left shifting until the MSB is 1. Adjust the exponent accordingly. Round the mantissa to the nearest value by adding 1 to the LSB if it is 1. Truncate if LSB is 0. Check for overflow or underflow and set exception flags. Encode the sign, exponent, and rounded mantissa into the 16-bit IEEE 754 output format.

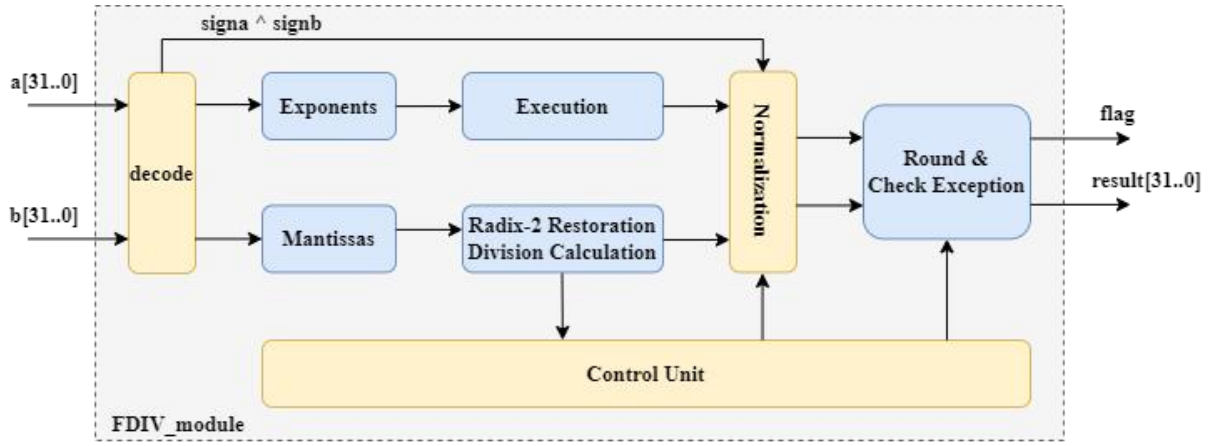
2.2.3. Floating-Point Divider (FDIV) Design

❖ **Floating-Point Division Algorithm**



The radix-2 restoration algorithm performs bitwise division by subtracting the divisor & shifting until quotient is computed. This avoids doing full hardware division.

❖ **Floating-Point Division Architecture**

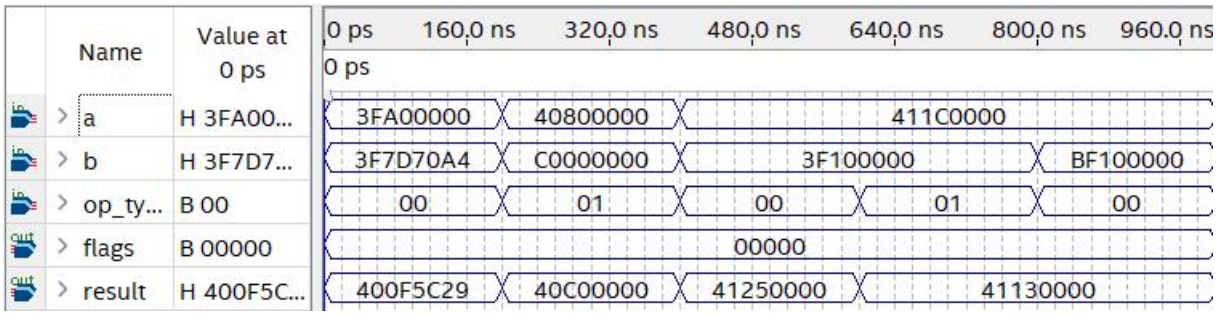


Decode inputs into sign, exponent, mantissa. Align mantissas by shifting divisor right until its MSB matches dividend's MSB. Perform radix-2 restoration division. Normalize quotient by shifting left until MSB is 1. Round quotient mantissa to nearest value. Check for underflow and inexact exceptions. Encode sign, exponent, mantissa into IEEE 754 format.

III. SIMULATION ON QUARTUS

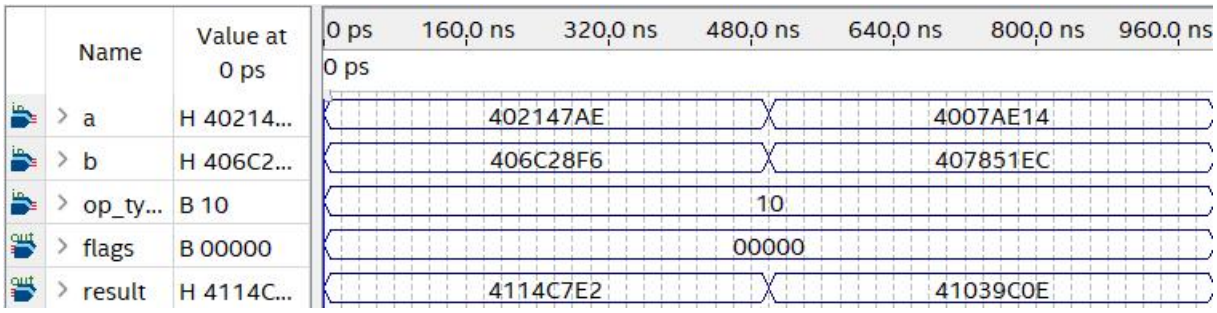
❖ Floating Point Addition & Floating Point Subtraction

Test Case	Operations	Expected Results
a + b	a = 1.25 => 0x3FA00000 b = 0.99 => 0x3F7D70A4 op = (+) => 2'b00	result = 2.24 => 0x400F5C29
a − (−b)	a = 4 => 0x40800000 b = −2 => 0xC0000000 op = (−) => 2'b01	result = 6 => 0x40C00000
a + b	a = 9.75 => 0x411C0000 b = 0.5625 => 0x3F100000 op = (+) => 2'b00	result = 10.3125 => 0x41250000
a − b	a = 9.75 => 0x411C0000 b = 0.5625 => 0x3F100000 op = (−) => 2'b01	result = 9.1875 => 0x41130000
a + (−b)	a = 9.75 => 0x411C0000 b = −0.5625 => 0xBF100000 op = (+) => 2'b00	result = 9.1875 => 0x41130000



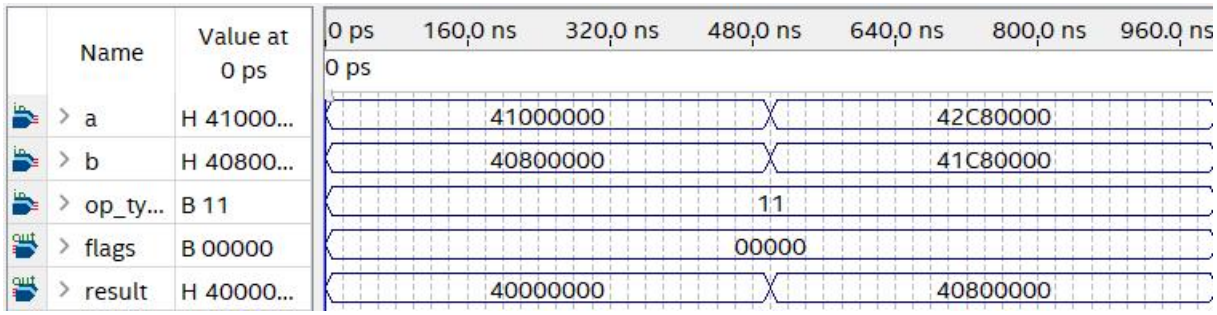
❖ Floating Point Multiplication

Operations (a *b)	Expected Results	Actual Results
a = 2.52 => 0x402147AE b = 3.69 => 0x406C28F6 op = (*) => 2'b10	result = 9.2988 => 0x4114C7E3	result = 9.2987995 => 0x4114C7E2
a = 2.12 => 0x4007AE14 b = 3.88 => 0x407851EC op = (*) => 2'b10	result = 8.2256 => 0x41039C0F	result = 8.225599 => 0x41039C0E



❖ Floating Point Division

Test Case	Operations	Expected Results
a / b	a = 8 => 0x41000000 b = 4 => 0x40800000 op = (/) => 2'b11	result = 2 => 0x40000000
a / b	a = 100 => 0x42C80000 b = 25 => 0x41C80000 op = (/) => 2'b11	result = 4 => 0x40800000



IV. CONCLUSION

Design and implement a 32-bit floating point unit (FPU) compliant with IEEE 754 standard. It perform basic arithmetic operations on 2-bit floating point operands, rounding results to the nearest value. We study IEEE 754 floating point format, Overview of sign, exponent, mantissa, special values (infinity, NaN, denormals), normalization and denormalization of floating point values, handling underflow and overflow conditions. However, we can not cover all test cases. We must improve and develop our algorithm an architecture for the furture project such as embedded FPU in our RISC-V cpu for floating point instructions.

V. REFERENCES

1. Book: Computer Principles and Design in Verilog HDL
2. Wikipedia: Division algorithm
3. GeeksforGeeks: IEEE Standard 754 Floating Point Numbers
4. Paper: Design and Simulation of 32-Bit Floating Point Arithmetic Logic Unit using VerilogHDL
5. Paper Single Precision Floating Point Unit (FPU) Based on IEEE 754 Standard Using Verilog