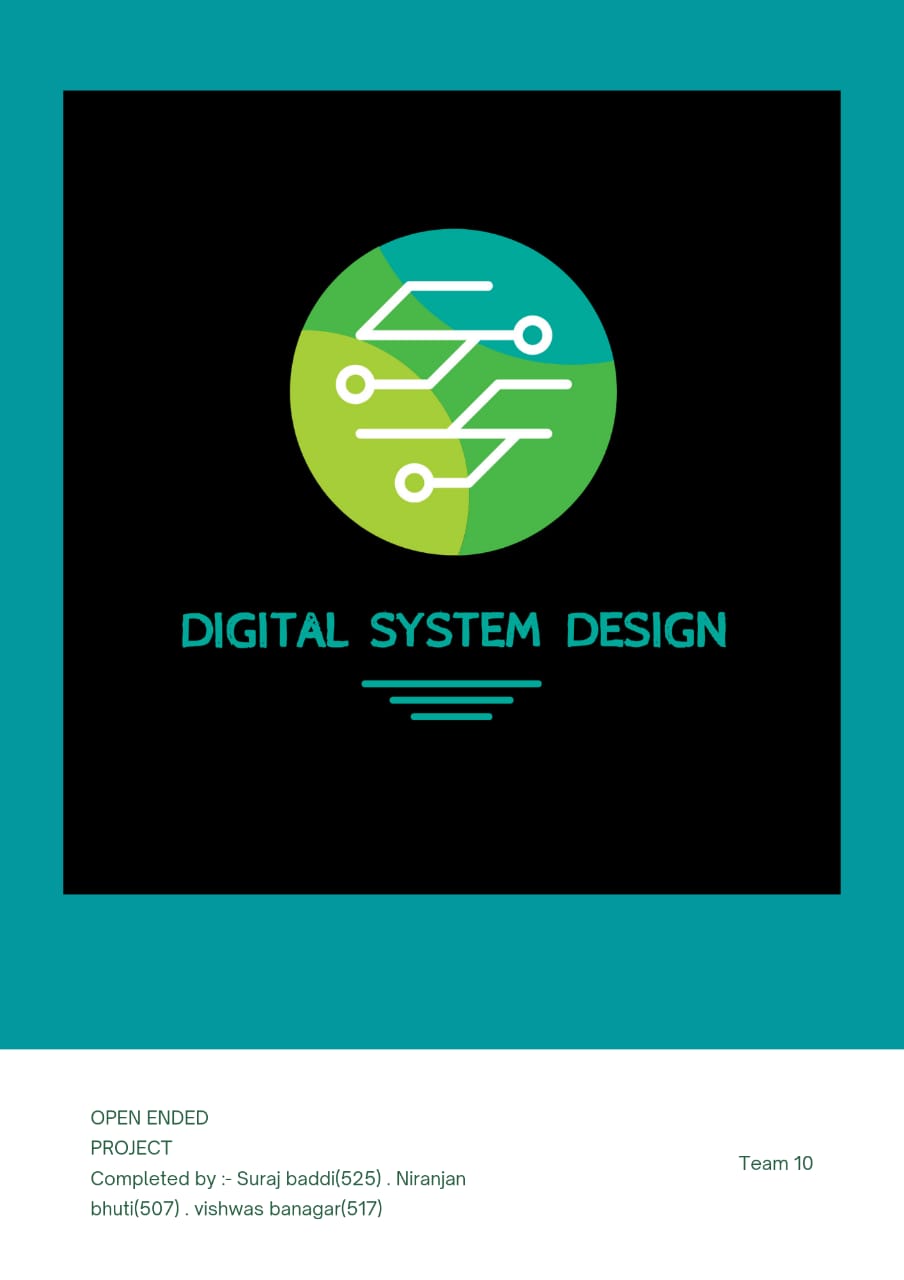
**DSD OPEN ENDED**

****

IV SEMESTER

‘E’ DIVISION

**Title of the Project: 32 bit floating point multiplier**

Team members:

|  |  |  |
| --- | --- | --- |
| **NAME** | **Roll no.** | **USN** |
| SURAJ BADDI | 525 | 01FE19BEC268 |
| VISHWAS BANAGAR | 517 | 01FE19BEC258 |
| NIRANJAN BHUTI | 507 | 01FE19BEC248 |

**PROBLEM STATEMENT:** Towrite a Verilog code and simulate 32 bit floating point multiplier

**EXPLAINING PROBLEM STATEMENT WITH INTRODUCTION:**

Explanation of problem statement

The IEEE-745 standard presents the floating format

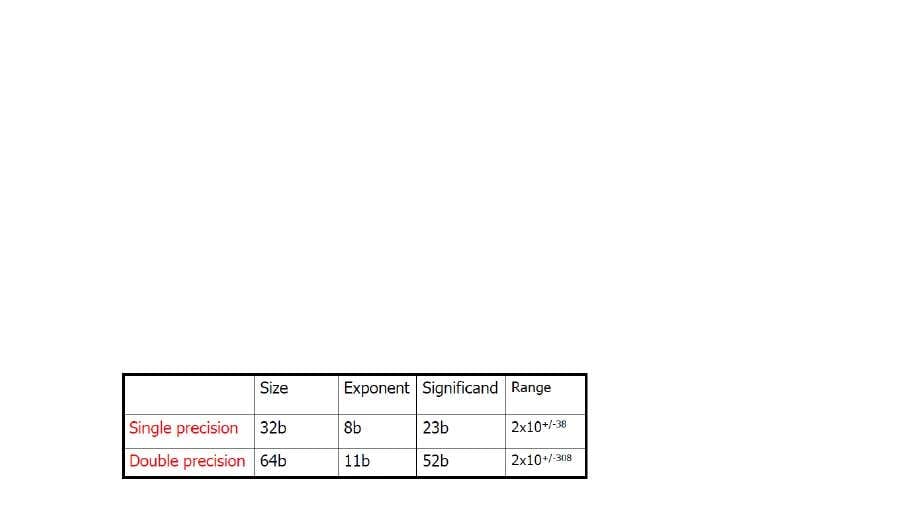
It consist of

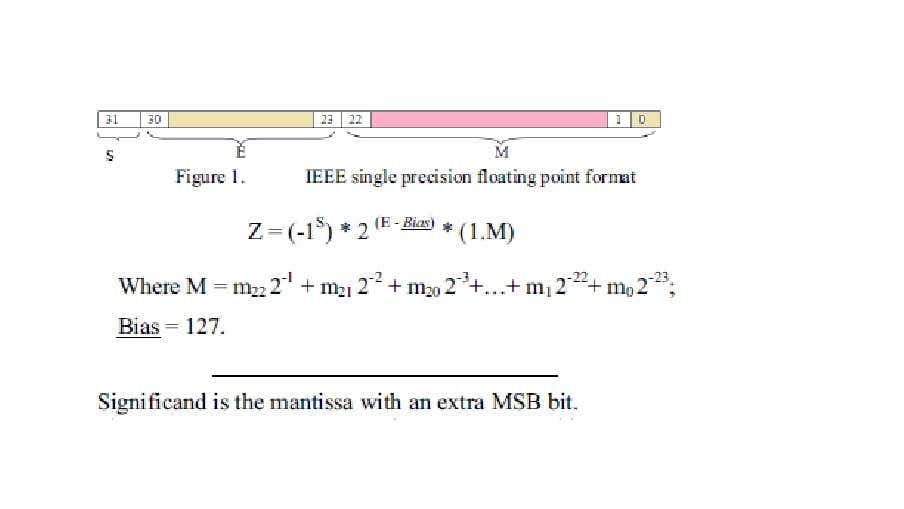
* -1 sign bit(S)
* -8-bit exponent( E)
* -23-bit mantissa(M)

An extra bit is added to the fraction to form what is called the significand

If the exponent is greater than 0 and smaller than 255, and there is 1 in MSB of the significand

Then a number is said to be a normalized number



****

* Let’s suppose a multiplication of 2 floating point numbers A & B, where A= -18.0

and B=9.5

* Binary representation of the operands:

A=-10010.000

B=+1001.1

* Normalized representation of the operands:

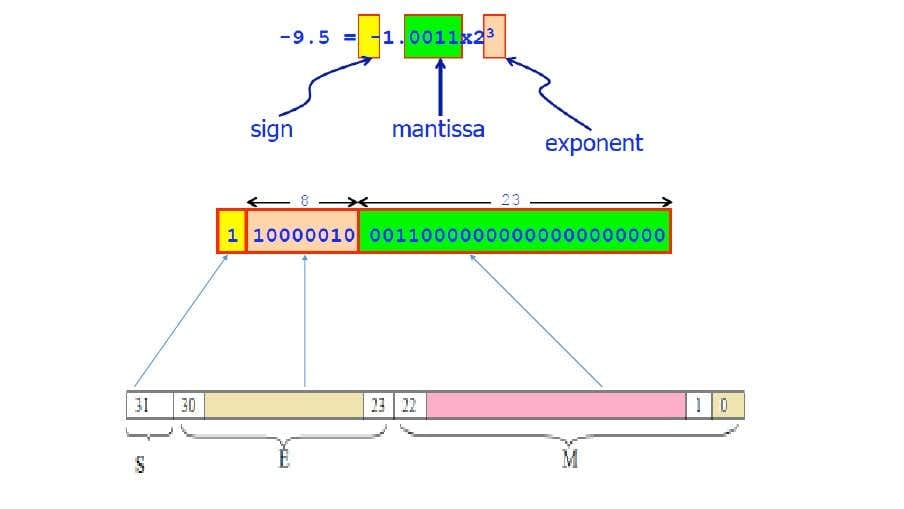
A=-1.001x2^4

B=+1.0011x2^4

* IEEE representation of the operands

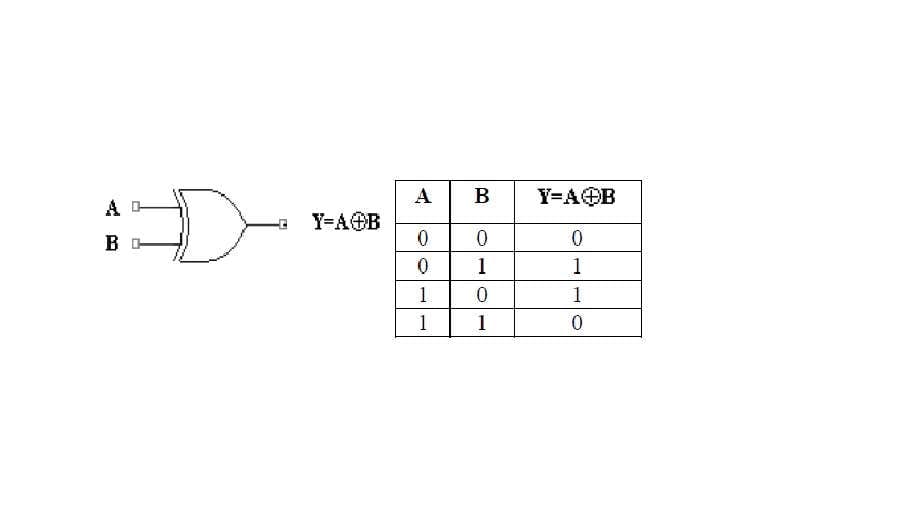
A= 11000001100100000000000000000000

B= 0100000100011000000000000000000



**DESIGN: Design of Floating Point Multiplier**

Sign bit calculation:



-multiplying Result is a negative sign if one of the multiplied number of a negative value

**DESIGN/ALGORITHM:**

FLOATING POINT MULTIPLICATION:

Multiplying two numbers in floating point format is done by

1. Adding the exponent of the two numbers then subtracting the bias from their result.

2. Multiplying the significant of the two numbers.

3. Calculating the sign by XOR ing the sign of the two numbers. In order to represent the multiplication result as a normalized number their should be in the MSB of the result.

FLOATING POINT MULTIPLICATION ALGORITHM:

As stated in the introduction, normalized floating point numbers have the form of

Z-(-1S) 2 (E-Bias) (LM).

To multiply two floating point numbers the following is done:

1. Multiplying the significand; i.e. (1.M1\*1.M2).

2. Placing the decimal point in the result.

3. Adding the exponents; i.e. (E1 + E2 - Bias).

4. Obtaining the sign; i.e. sl xor s2.

5. Normalizing the result: i.e. obtaining 1 at the MSB of the results significand.

6. Rounding the result to fit in the available bits.

7. Checking the underflow and overflow occurrence.

Consider a floating point representation similar to the IEEE 754 single precision floating point format, but with a reduced number of mantissa bits (only 4) while still retaining the hidden

1 bit for normalized numbers:

A-0100001000100-40,

B-1100000011110 -7.5.

To multiply A and B

1. Multiply significand:

1.0100

x 1.1110

00000

10100

10100

10100

10100

1001011000

2. Place the decimal point: 10.01011000

3. Add exponents:

10000100

+10000001

100000101

The exponent representing the two numbers is already shified/biased by the bias value (127) and

is not the true exponent; i.e. EA EA-true bias and EB EB-true + bias And EA+EB EA-true + EB-true + 2 bias

So we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

100000101

**PROCEDURE:**

* Obtaining the sign of the product
* Adding the exponents and subtracting the bias (=127)
* Multiplying the Mantissa with MSB 1 concatenated
* Placing the binary point in the result and checking for normalization requirement
* Normalising the Mantissa and incrementing exponent if needed
* Rounding off the Mantissa to 23 bits
* Checking for underflow/overflow

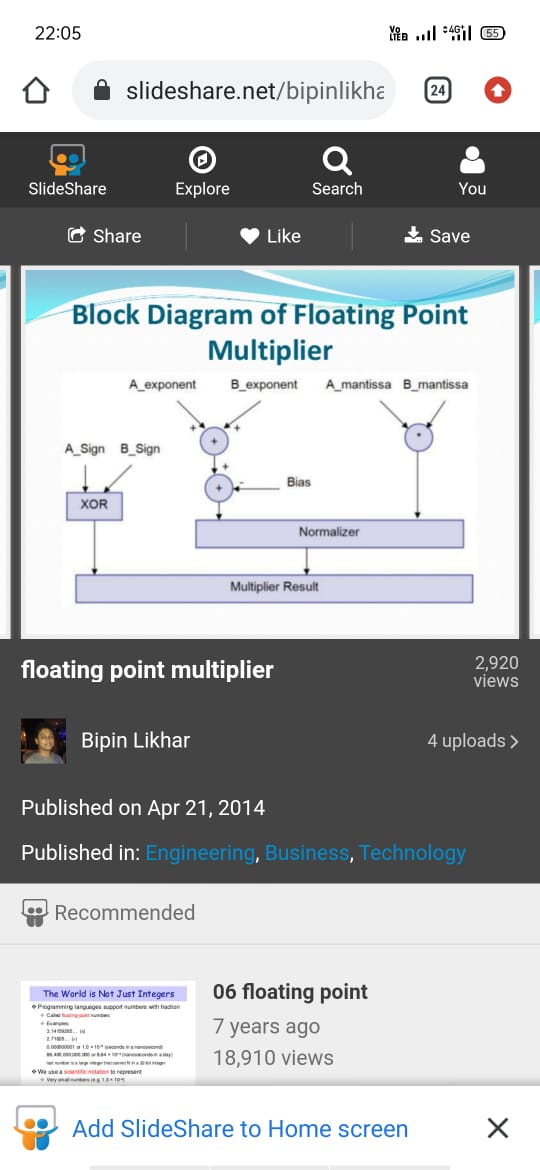
**APPLICATIONS :**

Floating point multiplication is a crucial operation in high power computing applications such as image processing, signal processing etc. And also multiplication is the most time and power consuming operation.

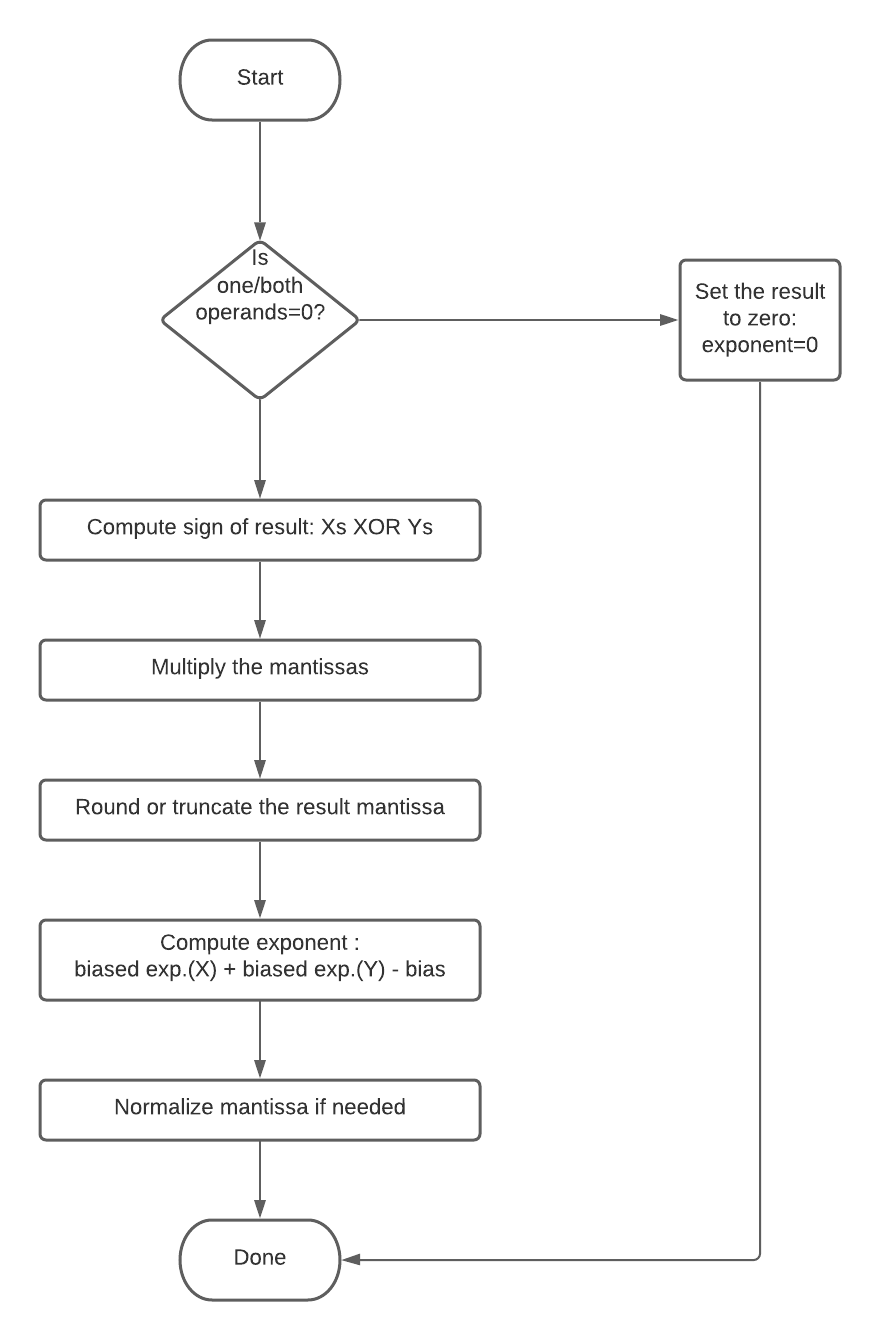
floating-point multiplication is commonly

used in 2D and 3D applications, which mostly use single precision floating-point operands

**BLOCK DIAGRAM:**

****

**FLOW CHART:**

****

**CODE:**

/\* Input and outputs are in single-precision IEEE 754 FP format:

Sign - 1 bit, position 32

Exponent - 8 bits, position 31-24

Mantissa - 23 bits, position 22-1

Not implemented: special cases like inf, NaN

\*/

// indices of components of IEEE 754 FP

`define SIGN 31

`define EXP 30:23

`define M 22:0

`define P 24 // number of bits for mantissa (including

`define G 23 // guard bit index

`define R 22 // round bit index

`define S 21:0 // sticky bits range

`define BIAS 127

module fp\_mul(input wire clk,

input wire[31:0] a,

input wire[31:0] b,

output wire[31:0] y,output underflow,output overflow);

reg [22:0] m;

reg [7:0] e;

reg s;

reg [`P\*2-1:0] product;

reg G;

reg R;

reg S;

reg normalized;

reg state;

reg next\_state = 0;

parameter STEP\_1 = 1'b0, STEP\_2 = 1'b1;

always @(posedge clk) begin

state <= next\_state;

end

always @(state) begin

case(state)

STEP\_1: begin

// mantissa is product of a and b's mantissas,

// with a 1 added as the MSB to each

product = {1'b1, a[`M]} \* {1'b1, b[`M]};

// get sticky bits by ORing together all bits right of R

S = |product[`S];

// if the MSB of the resulting product is 0

// normalize by shifting right

normalized = product[47];

if(!normalized) product = product << 1;

next\_state = STEP\_2;

end

STEP\_2: begin

// if either mantissa is 0, result is 0

if(!a[`M] | !b[`M]) begin

s = 0; e = 0; m = 0;

end else begin

// sign is xor of signs

s = a[`SIGN] ^ b[`SIGN];

// mantissa is upper 22-bits of product w/ nearest-even rounding

m = product[46:24] + (product[`G] & (product[`R] | S));

// exponent is sum of a and b's exponents, minus the bias

// if the mantissa was shifted, increment the exponent to balance it

e = a[`EXP] + b[`EXP] - `BIAS + normalized;

end

next\_state = STEP\_1;

end

endcase

end

//if sum of exponent is greater than 255 then overflow condition takes place

assign underflow=(e[8]&e[7]&!0)?1'b1:1'b0;

//if sum of exponent is less than 127 then underflow condition takes place

assign overflow=(e[8]&!e[7]&!0);

// output is concatenation of sign, exponent, and mantissa

assign y = {s, e, m};

endmodule

**SIMULATION RESULTS:**

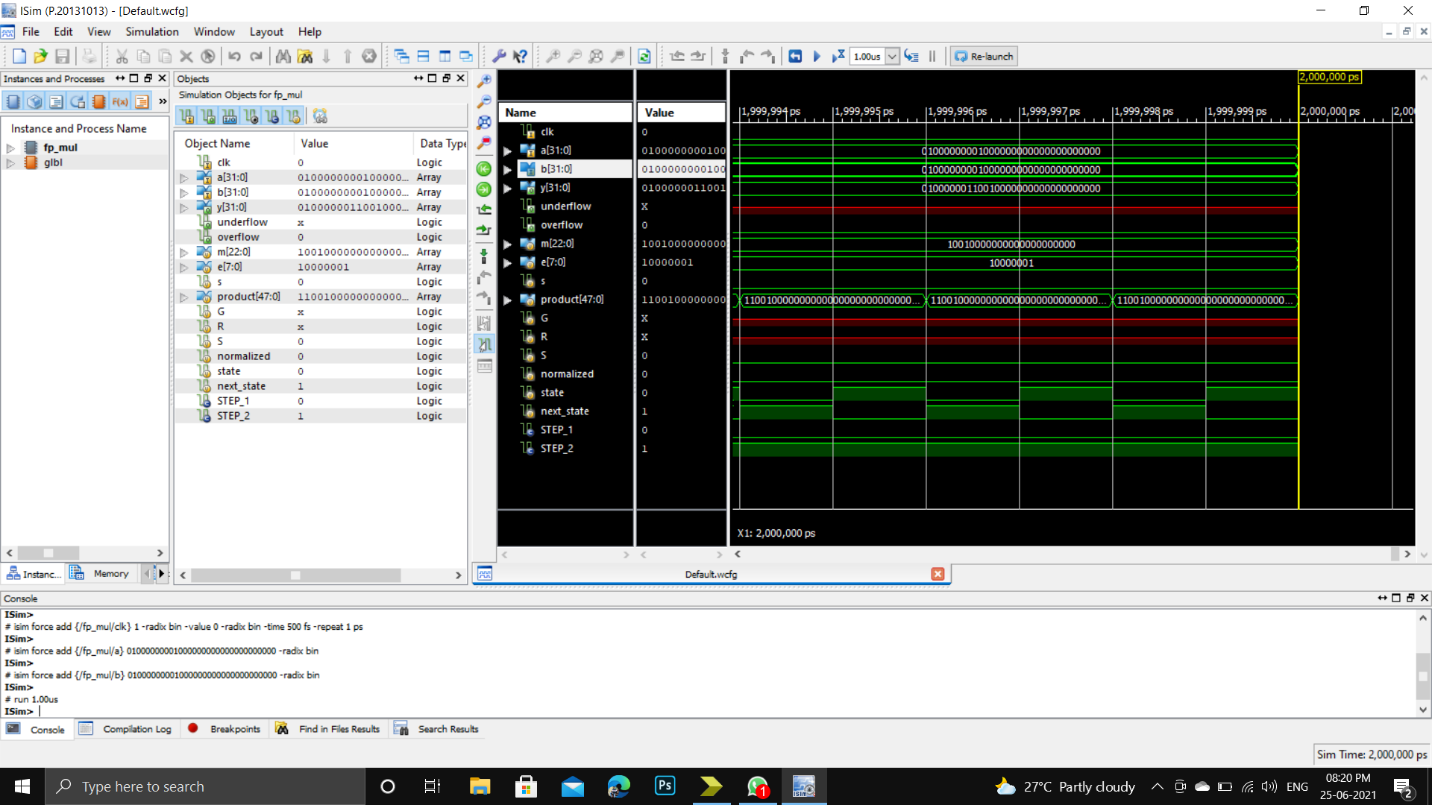
**Example 1 :**

Input:

a = 01000000001000000000000000000000 b = 01000000001000000000000000000000

Output:

y = 01000000110010000000000000000000

****

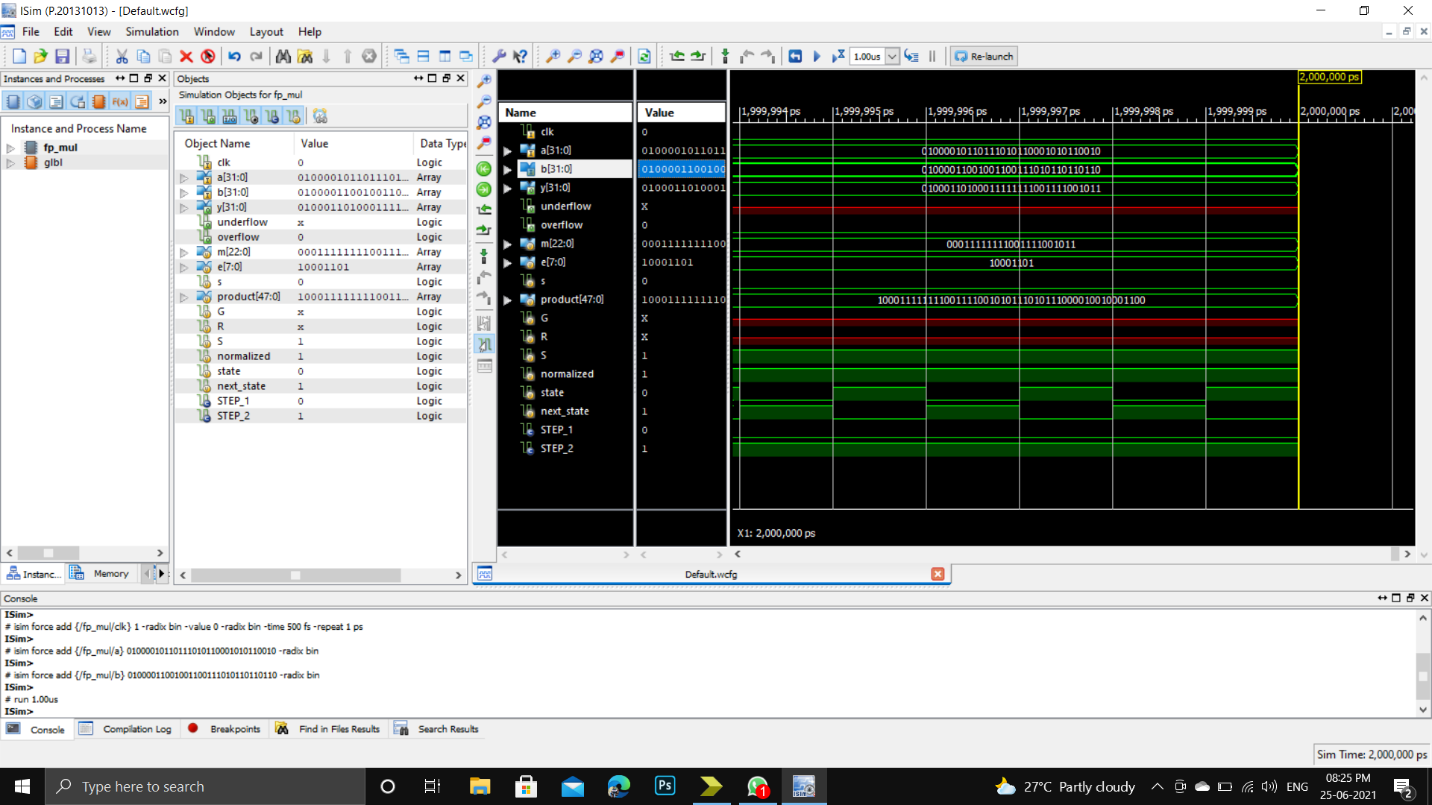
**Example 2:**

Input:

a = 01000011001001100111010110110110 b = 01000011001001100111010110110110

Output:

Y = 01000110100011111111001111001011

****

**CONCLUSION:**

This Seminar presents format, design and simulation of a 32-bit floating point multiplier that supports the IEEE 754-2008 binary