MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

LAB PROJECT REPORT

for

ECE 2244: VLSI Design Lab

IEEE-754 32-BIT FLOATING POINT MULTIPLIER

Submitted by

Lakshit Verma
Reg. 230959210
Roll No. 75

and

Pranav Saini
Reg. 230959212
Roll No. 76

Batch: A3

DEPT. OF ELECTRONICS AND COMMUNICATION (ECE)
MANIPAL INSTITUTE OF TECHNOLOGY, MANIPAL - 576104

April 2025

INDEX

OBJECTIVE

To design and implement an IEEE 754 compliant 32-bit floating point multiplier using Verilog and synthesize it, focusing on accuracy, power consumption, area, and timings.

CHAPTER 1: INTRODUCTION

Floating point arithmetic is widely used in various computing applications, including scientific computations, digital signal processing, and graphics processing. This project focuses on the design and implementation of an IEEE 754 compliant 32-bit floating point multiplier using VLSI design methodologies. The IEEE 754 standard defines a precise format for representing floating point numbers and ensures accuracy in floating point operations.

The proposed design consists of three main stages: exponent addition with bias adjustment, mantissa multiplication with normalization, and rounding to maintain precision. Special cases such as underflow, overflow, and handling of special numbers (NaN, INFINITY) are also considered. The implementation is carried out using Verilog HDL and synthesized using FPGA-based tools to evaluate the area, power, and timing performance.

The results demonstrate efficient multiplication with optimized resource utilization, making it suitable for high-performance computing applications. This mini-project provides insights into floating point arithmetic and its practical realization in VLSI circuits.

CHAPTER 2: PROJECT THEORY AND EXPLANATION

IEEE 754 Format

A 32-bit floating-point number consists of three parts:

```
S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
```

where:

1. **S (1 bit):** Sign bit (1 = positive, 0 = negative)
2. **E (8 bits):** Exponent (biased by 127)
3. **M (23 bits):** Mantisssa (fractional part, normalized)

Let's take a sample 32-bit binary representation:

```
0 10000010 11000000000000000000000
```

1. Sign bit (0) indicates that it is a positive number
2. Exponent (10000010 binary = 130 decimal) so actual exponent = 130 - 127 = 3
3. Mantissa (1.110 in binary = 1.75 in decimal, since implicit leading 1)

Thus, the value is:

$$1.75 \times 2^3 = 14.0$$

Structure of the module

The code is layed out as follows:

Internal Registers:

- `counter`: Controls execution stages (3-bit).
- `a_m`, `b_m`, `z_m`: Mantissas (24-bit).

- `a_e, b_e, z_e`: Exponents (10-bit).
- `a_s, b_s, z_s`: Signs (1-bit).
- `product`: Stores intermediate mantissa multiplication (50-bit).
- `guard_bit, round_bit, sticky`: Used for rounding.

The working of the code are as follows:

1. **Counter Update**: Increments on every clock cycle or resets if `rst` is high.
2. **Extract Components**: Extracts sign, exponent, and mantissa from inputs `a` and `b`.
3. **Handle Special Cases**:
    - Checks for `NaN` (Not a Number), Infinity, or Zero conditions.
    - Denormalizes subnormal numbers.
4. **Normalize Inputs**: Ensures mantissas are normalized.
5. **Compute Product**:
    - Computes sign (`z_s`), adds exponents (`z_e`), and multiplies mantissas (`product`).
6. **Extract Mantissa and Guard Bits**:
    - Extracts top 24 bits from `product` and sets guard, round, and sticky bits.
7. **Normalize & Round**:
    - Adjusts exponent and mantissa to maintain IEEE 754 compliance.
    - Rounds the result if necessary.
8. **Generate Output**:
    - Reconstructs final floating-point result in IEEE 754 format.
    - Handles overflow/underflow cases.

CHAPTER 3: VERILOG CODE AND TESTBENCH

The Verilog code for the multiplier is given below:

```
module fmultiplier(clk, rst, a, b, z);

input clk, rst;
input [31:0] a, b;
output reg [31:0] z;

reg [2:0] counter;

reg [23:0] a_m, b_m, z_m;
reg [9:0] a_e, b_e, z_e;
reg a_s, b_s, z_s;

reg [49:0] product;

reg guard_bit, round_bit, sticky;

always @(posedge clk or posedge rst) begin
    if (rst)
        counter <= 0;
    else
        counter <= counter + 1;
end


always @(counter) begin
    if(counter == 3'b001) begin
        a_m <= a[22:0];
            b_m <= b[22:0];
            a_e <= a[30:23] - 127;
            b_e <= b[30:23] - 127;
            a_s <= a[31];
            b_s <= b[31];
        end
end


always @(counter) begin
    if(counter == 3'b010) begin
        if ((a_e == 128 && a_m != 0) || (b_e == 128 && b_m != 0)) begin // NAN
                    z[31] <= 1;
                    z[30:23] <= 255;
                    z[22] <= 1;
                    z[21:0] <= 0;
                end
                else if (a_e == 128) begin // INF A
                    z[31] <= a_s ^ b_s;
                    z[30:23] <= 255;
```

```
                z[22:0] <= 0;
                if (($signed(b_e) == -127) && (b_m == 0)) begin // NAN if B=0
                      z[31] <= 1;
                      z[30:23] <= 255;
                      z[22] <= 1;
                      z[21:0] <= 0;
                end
            end
            else if (b_e == 128) begin //INF B
                z[31] <= a_s ^ b_s;
                z[30:23] <= 255;
                z[22:0] <= 0;
                if (($signed(a_e) == -127) && (a_m == 0)) begin //NAN if A=0
                      z[31] <= 1;
                      z[30:23] <= 255;
                      z[22] <= 1;
                      z[21:0] <= 0;
                end
            end
            else if (($signed(a_e) == -127) && (a_m == 0)) begin // 0 if A = 0
              z[31] <= a_s ^ b_s;
              z[30:23] <= 0;
                z[22:0] <= 0;
            end
            else if (($signed(b_e) == -127) && (b_m == 0)) begin // 0 if B = 0
                z[31] <= a_s ^ b_s;
                 z[30:23] <= 0;
                 z[22:0] <= 0;
            end
            else begin
                if ($signed(a_e) == -127) // denormalizing A
                      a_e <= -126;
                else
                      a_m[23] <= 1;

                  if ($signed(b_e) == -127) // denormalizing B
                      b_e <= -126;
                else
                      b_m[23] <= 1;
            end
        end
end


always @(counter) begin
    if(counter == 3'b011) begin
        if (~a_m[23]) begin // normalize A
                a_m <= a_m << 1;
               a_e <= a_e - 1;
            end
            if (~b_m[23]) begin // normalize B
                b_m <= b_m << 1;
               b_e <= b_e - 1;
            end
```

```
    end
end


always @(counter) begin
    if(counter == 3'b100) begin // get the signs xored and exponents added and
        z_s <= a_s ^ b_s;       // the intermediate mantissa multiplication
            z_e <= a_e + b_e + 1;
            product <= a_m * b_m * 4;
    end
end


always @(counter) begin
    if(counter == 3'b101) begin
        z_m <= product[49:26];
           guard_bit <= product[25];
              round_bit <= product[24];
              sticky <= (product[23:0] != 0);
    end
end

always @(counter) begin
    if(counter == 3'b110) begin
        if ($signed(z_e) < -126) begin
                z_e <= z_e + (-126 -$signed(z_e));
                z_m <= z_m >> (-126 -$signed(z_e));
                 guard_bit <= z_m[0];
               round_bit <= guard_bit;
               sticky <= sticky | round_bit;
            end
        else if (z_m[23] == 0) begin
                z_e <= z_e - 1;
                z_m <= z_m << 1;
                z_m[0] <= guard_bit;
                guard_bit <= round_bit;
                round_bit <= 0;
            end
            else if (guard_bit && (round_bit | sticky | z_m[0])) begin
                z_m <= z_m + 1;
                  if (z_m == 24'hffffff)
                        z_e <=z_e + 1;
            end
        end
end

always @(counter) begin
    if(counter == 3'b111) begin
        z[22:0] <= z_m[22:0];
            z[30:23] <= z_e[7:0] + 127;
            z[31] <= z_s;
            if ($signed(z_e) == -126 && z_m[23] == 0)
                  z[30:23] <= 0;
            if ($signed(z_e) > 127) begin // if overflow return INF
```

```
                z[22:0] <= 0;
                z[30:23] <= 255;
                z[31] <= z_s;
            end
    end
end

endmodule
```

And the Verilog code for the testbench is given below:

```
module fmultiplier_tb;

reg [31:0] a, b;
wire [31:0] z;
reg clk, rst;

fmultiplier multiplier(clk, rst, a, b, z);

initial begin
      $dumpfile("wave.vcd"); // dumps the waveforms
      $dumpvars(0, fmultiplier_tb);
end

initial begin
      clk <= 0;
      rst <= 1;
      repeat (17000) // clocks 17,000 times with a period of 5ns
            #5 clk <= ~clk;
end

initial #13 rst <= 0;

initial begin
      #3
      repeat (500) begin
            #80
            a = $random; // assigns random values to
            b = $random; // both a and b
      end
      #80 $finish;
end

endmodule
```

CHAPTER 4: RESULTS

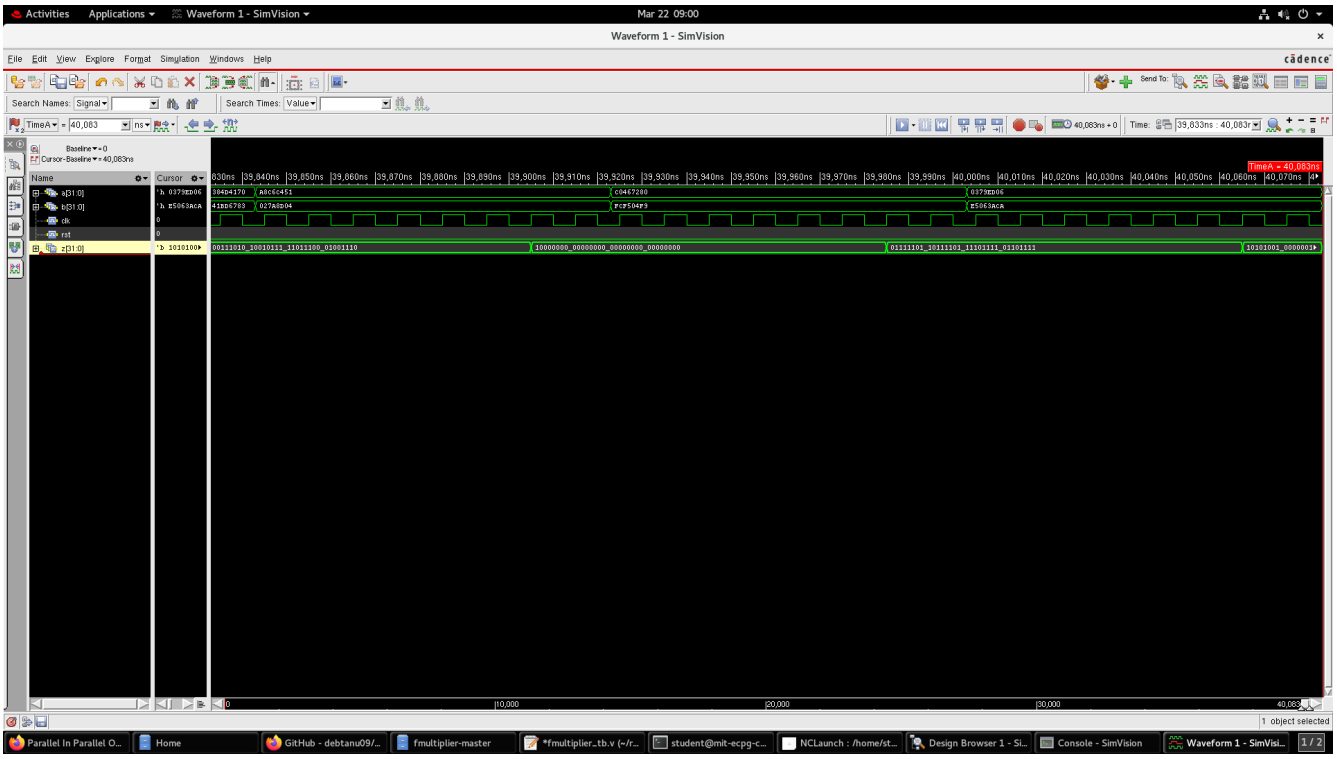

Fig. 1: Output Waveform in NCLAUNCH

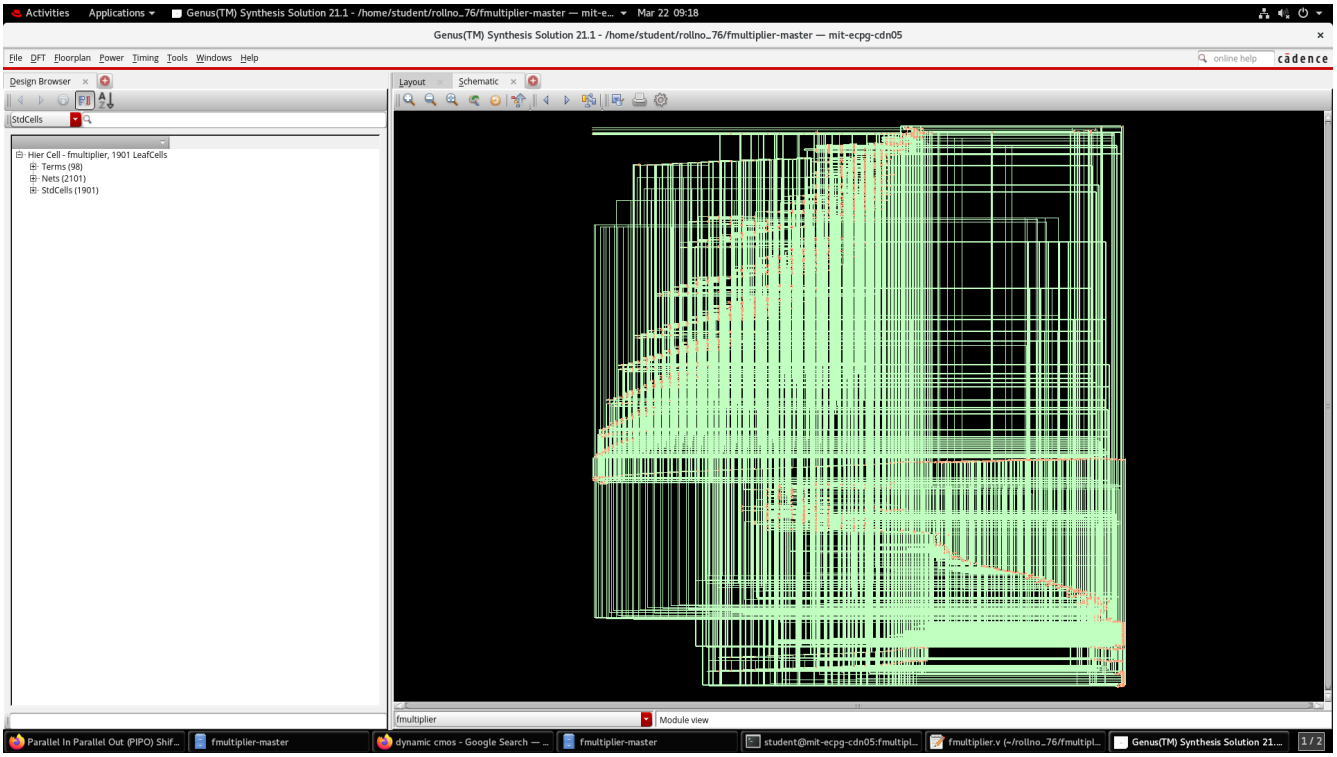

Fig. 2: Synthesized Circuit in Genus

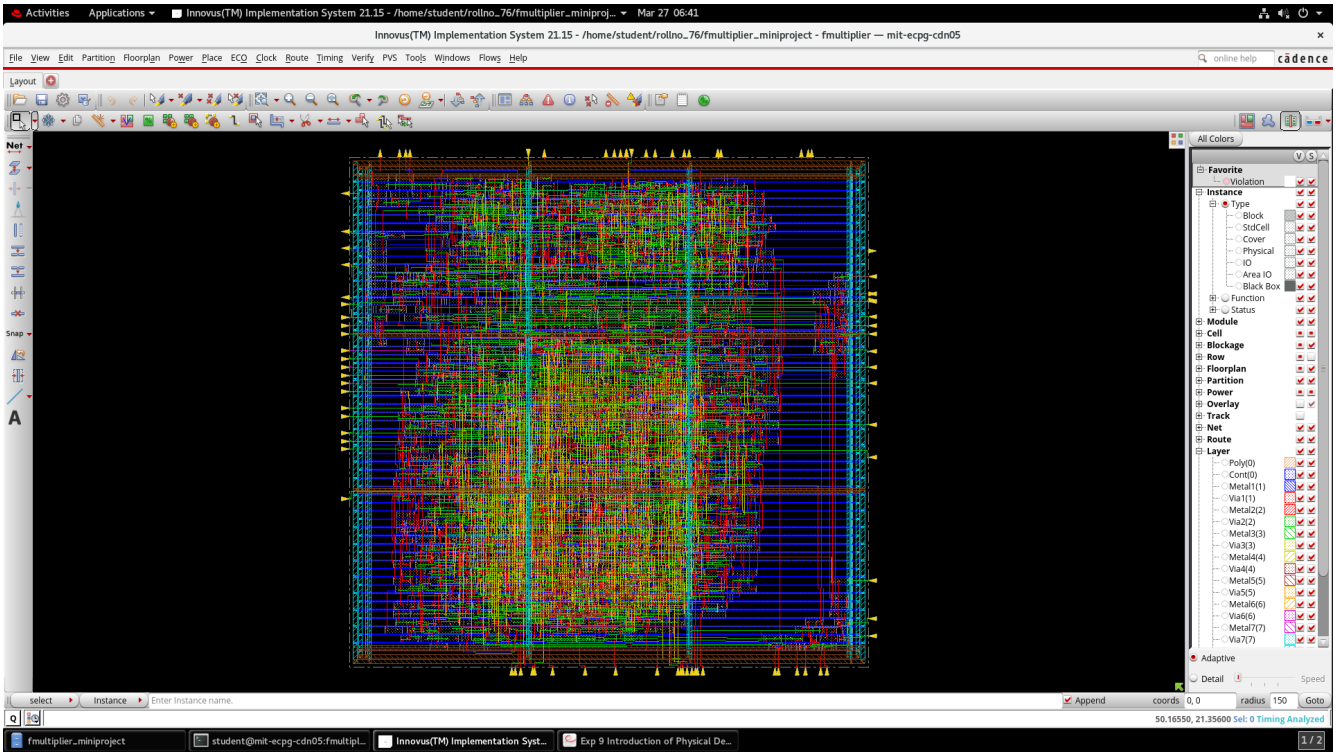

Fig. 3: Physical Design in Innovus

CONCLUSION

Through this project, we gained a deeper understanding of floating-point arithmetic. We explored the IEEE 754 standard, learning how floating-point numbers are represented and manipulated in hardware. Implementing normalization, rounding, and handling special cases gave us insight into practical challenges in floating-point multiplication. Additionally, we developed skills in Verilog coding, simulation, and synthesis, understanding how different design choices impact performance in terms of area, power, and timing. Having to debug and verify the design using a testbench also strengthened our problem-solving abilities, showing the importance of careful design validation in VLSI development.

REFERENCES

1. IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008.

2. Floating Point Numbers — Computerphile (YouTube): https://www.youtube.com/watch?v=PZRI1IfStY0

3. Online resources and research papers on IEEE 754 floating point arithmetic and Verilog design methodologies.

4. https://numeral-systems.com/ieee-754-multiply/