AMRITA VISHWA VIDYAPEETHAM

Course Code: 19ECE343

Course Name: FPGA Based System Design

Component: Mini Project

Date of Evaluation: 29/04/2025

Academic Year: 2024-2025 (Even Semester)

Batch Number: 11

Name of the Students: 1) Abburi Sai Keerthi
BL.EN.U4ECE22002

With Registration no. 2) B .A .V. N Hasini
BL.EN.U4ECE22010

3) Megha Elango
BL.EN.U4ECE22035

Semester, Branch and Section: 6th Semester, ECE-A

Batch: 2022-2026

Faculty In-charge: Ms. Sonali Agarwal

Aim:

To design Hardware-Efficient Accurate and Approximate Multipliers for FPGA systems using optimized Booth multiplication techniques, improving area, power, and performance, especially for error-tolerant applications like 5G wireless communication.

Tool Used:

Xilinx Vivado 2018.3, Verilog and VHDL, Vivado Simulator, Power Analyzer and Timing Analyzer

Theory:

Introduction

- Multipliers are a core part of FPGA-based systems like communications, DSP, and AI accelerators.
    - FPGAs combine LUTs and carrychains to build multipliers.
    - Traditional multipliers are area-consuming, power-hungry, and slow when directly mapped.
    - Approximate computing allows slight errors in applications where full accuracy is unnecessary (e.g., wireless systems, image processing).
    - The paper proposes:
    - An area-optimized accurate unsigned Booth multiplier (called Area-Mult).
    - A truncated approximate multiplier with probability-based error compensation (called Pro-AppT8)

Design of Accurate Multiplier:

Booth Encoding Principle:

Booth encoding is an efficient multiplication technique that reduces the number of partial products by encoding sequences of 1’s in the multiplier, helping to minimize the number of additions or subtractions. It is especially beneficial for signed binary multiplication, where it detects patterns like 01 or 10 to decide whether to add, subtract, or skip the multiplicand. In our design, we implement a simplified form of Booth encoding tailored for unsigned numbers, where subtraction is not needed. Instead, each bit of the multiplier (B) is individually checked. If B[i] is 1, it means the multiplicand (A) contributes to the result at position I ,So A is left-shifted by i bits and added to the total sum. If B[i] is 0, no operation is performed for that bit. This approach maintains accuracy while optimizing the multiplication process by skipping unnecessary additions, thus mimicking Booth's efficiency in a straightforward and hardware friendly way.

Example of Booth Encoding:
Multiplying A = 5 (0101) and B = 3 (0011)
Since B[0] = 1 → Add A shifted by 0 (value = 5)
Since B[1] = 1 → Add A shifted by 1 (value = 10)
All other bits are 0 → No contribution
Final Product = 5 + 10 = 15

Principle of Accurate Multiplier:

An accurate multiplier aims to compute the exact product of two binary numbers without approximation, ensuring full precision across all possible input combinations. This design uses partial product generation and accumulation to implement precise multiplication. By combining optimized techniques like Booth encoding and structured addition of partial products, the multiplier delivers accurate and reliable results essential for applications where numerical precision is critical.

Code Functionality:

The Verilog module area_mult_10x10 implements a 10x10-bit accurate multiplier using Booth-encoded partial products. Each bit of the multiplier B controls whether a shifted version of the multiplicand A is added to form a partial product (pp[0] to pp[9]). These 10 partial products are conditionally generated and aligned by shifting according to the bit position. They are then summed using a series of binary additions, starting with pp[0] + pp[1] and sequentially accumulating the rest, ensuring the final product P is correctly computed. The testbench applies several test cases and prints the output to validate the design's accuracy.

Verilog Code:

```
`timescale 1ns / 1ps

module area_mult_10x10(

    input  wire [9:0] A,

    input  wire [9:0] B,

    output wire [19:0] P

);

    wire [19:0] pp[9:0];

    wire [19:0] sum1, sum2, sum3, sum4;

    wire carry1, carry2, carry3, carry4;
```

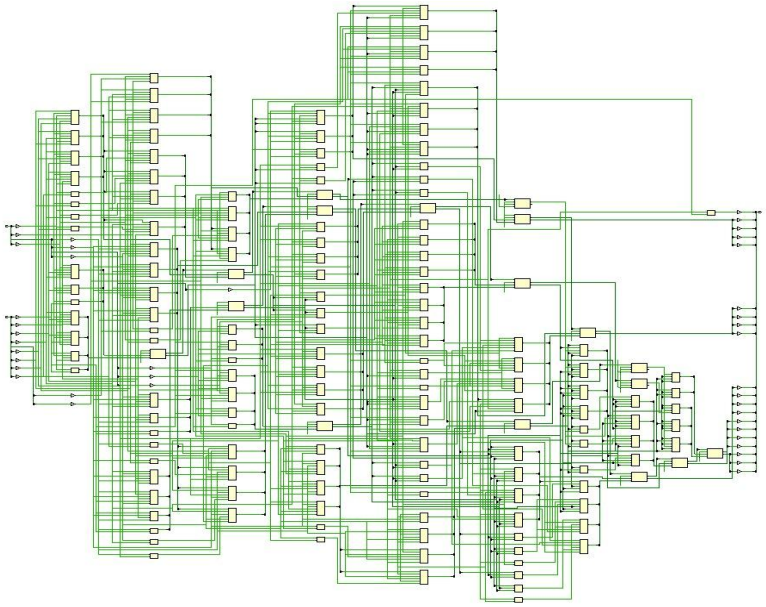
```
    assign pp[0] = (B[0]) ? {10'b0, A} : 20'b0;
    assign pp[1] = (B[1]) ? {9'b0, A, 1'b0} : 20'b0;
    assign pp[2] = (B[2]) ? {8'b0, A, 2'b00} : 20'b0;
    assign pp[3] = (B[3]) ? {7'b0, A, 3'b000} : 20'b0;
    assign pp[4] = (B[4]) ? {6'b0, A, 4'b0000} : 20'b0;
    assign pp[5] = (B[5]) ? {5'b0, A, 5'b00000} : 20'b0;
    assign pp[6] = (B[6]) ? {4'b0, A, 6'b000000} : 20'b0;
    assign pp[7] = (B[7]) ? {3'b0, A, 7'b0000000} : 20'b0;
    assign pp[8] = (B[8]) ? {2'b0, A, 8'b00000000} : 20'b0;
    assign pp[9] = (B[9]) ? {1'b0, A, 9'b000000000} : 20'b0;
    assign {carry1, sum1} = pp[0] + pp[1];
    assign {carry2, sum2} = sum1 + pp[2];
    assign {carry3, sum3} = sum2 + pp[3];
    assign {carry4, sum4} = sum3 + pp[4];
    assign P = sum4 + pp[5] + pp[6] + pp[7] + pp[8] + pp[9];
endmodule
```

Testbench:

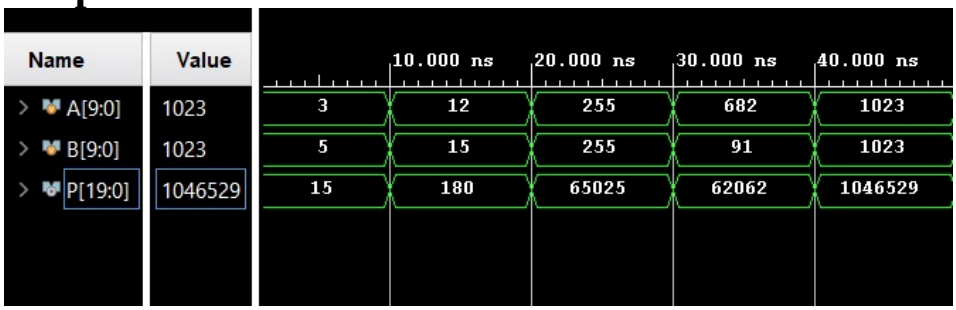
```
 `timescale 1ns / 1ps
module tb_area_mult_10x10;
    reg [9:0] A;
    reg [9:0] B;
    wire [19:0] P;
    area_mult_10x10 uut (
        .A(A),
```

```
        .B(B),
        .P(P)
    );
    Initial
      begin
       $monitor("Time = %0t | A = %d, B = %d, P = %d", $time, A, B, P);
      End
    initial
       begin
       A = 10'd3; B = 10'd5;
       #10;
       A = 10'd12; B = 10'd15;
       #10;
       A = 10'd255; B = 10'd255;
       #10;
       A = 10'd682; B = 10'd91;
       #10;
       A = 10'd1023; B = 10'd1023;
       #10;
       $finish;
     end
endmodule
```

Block Diagram of Accurate Multiplier:



Output:

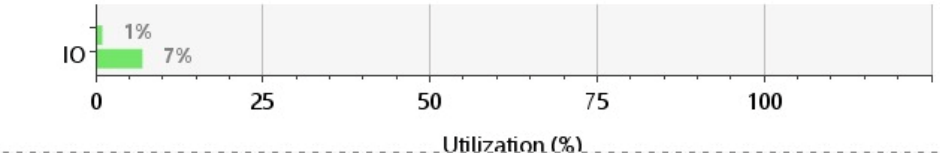



```
# run 1000ns
Time = 0 | A =    3, B =    5, P =      15
Time = 10000 | A =   12, B =   15, P =     180
Time = 20000 | A =  255, B =  255, P =   65025
Time = 30000 | A =  682, B =   91, P =   62062
Time = 40000 | A = 1023, B = 1023, P = 1046529
```

Synthesis:

Area

Resource	Utilization	Available	Utilization %
LUT	114	303600	0.04
IO	40	600	6.67



Timing

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	20	Total Number of Endpoints:	20	Total Number of Endpoints:	NA

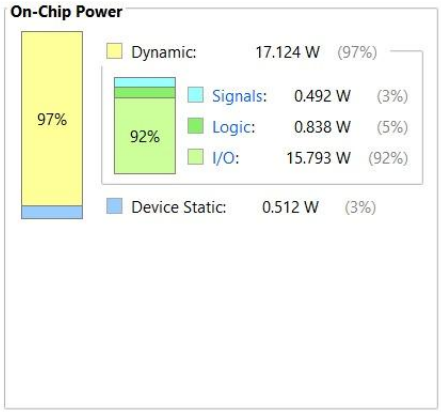
There are no user specified timing constraints.

Power

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**17.635 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**49.7°C**
Thermal Margin:	35.3°C (24.0 W)
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



Principle of Approximate Multiplier:

Approximate multipliers are designed to trade off a small amount of computational accuracy in exchange for significant improvements in area, power, and delay, making them ideal for error-tolerant applications such as image processing, machine learning, and signal processing. The key idea is to simplify hardware operations by reducing the number of partial products and ignoring the least significant bits (LSBs), which have minimal impact on the final result. This reduces the complexity of addition and enables faster and more energy-efficient designs.

Code Functionality:

This design implements a 10x10 Pro-AppT8 approximate multiplier that performs basic bitwise AND operations between the multiplicand (A) and each bit of the multiplier (B) to generate partial products. These partial products are then shifted and accumulated to form a full product estimate. To reduce complexity, the least significant 8 bits of the final sum are truncated, which simplifies hardware but introduces error. To compensate for the loss, a probabilistic correction is applied by checking the top 4 partial products in the highest bit column (pp[0][9] to pp[3][9]) and adding a scaled compensation value. This combination of LSB truncation and probabilistic compensation helps balance accuracy with performance.

Verilog Code:

```
`timescale 1ns / 1ps

module pro_app_multiplier #(

parameter TRUNC_BITS = 8

)(

    input  [9:0] A,

    input  [9:0] B,

    output [19:0] P

);

    wire [9:0] pp [0:9];

    reg [19:0] compensation;

    integer i;

    integer j;

    genvar m;

    generate

        for (m = 0; m < 10; m = m + 1) begin: gen_pp_rows

            assign pp[m] = A & {10{B[m]}};

        end

    endgenerate

    reg [19:0] approx_sum;

    always @(*) begin

        approx_sum = 0;

        for (i = 0; i < 10; i = i + 1) begin

            approx_sum = approx_sum + (pp[i] << i);
```

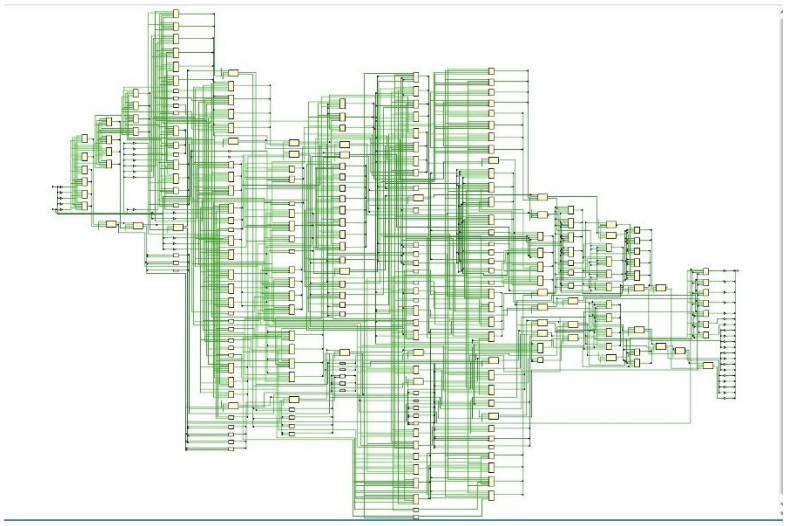
```
        end
    end
    wire [19:0] truncated_sum = {approx_sum[19:TRUNC_BITS],
{TRUNC_BITS{1'b0}}};
    always @(*) begin
        compensation = 0;
        for (j = 0; j < 4; j = j + 1) begin
            if (pp[j][9]) begin
                compensation = compensation + (1 << 9);
            end
        end
        compensation = (compensation * 256) / 384;
    end
    assign P = truncated_sum + compensation;
endmodule
```

Testbench:

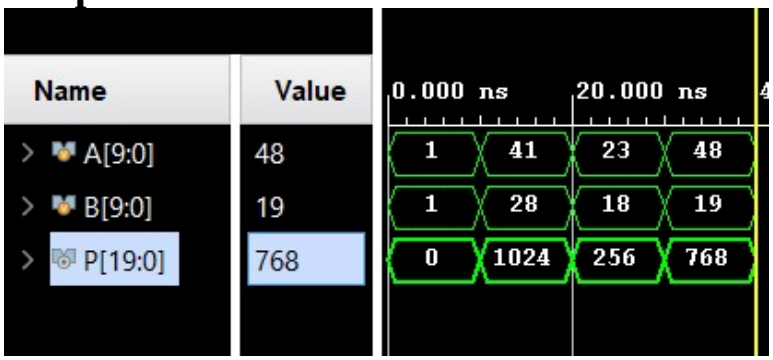
```
`timescale 1ns / 1ps
module tb_pro_app_multiplier;
    reg [9:0] A;
    reg [9:0] B;
    wire [19:0] P;
    pro_app_multiplier #(
        .TRUNC_BITS(8)
    ) uut (
        .A(A),
```

```
        .B(B),
        .P(P)
    );
    initial begin
        $monitor("Time = %0t | A = %d, B = %d, P = %d", $time, A, B, P);
    end
    initial
        begin
        A = 10'd1; B = 10'd1;
        #10;
        A = 10'd41; B = 10'd28;
        #10;
        A = 10'd23; B = 10'd18;
        #10;
        A = 10'd48; B = 10'd19;
        #10;
        $finish;
    end
endmodule
```

Block Diagram for Approximate Multiplier:



Output:

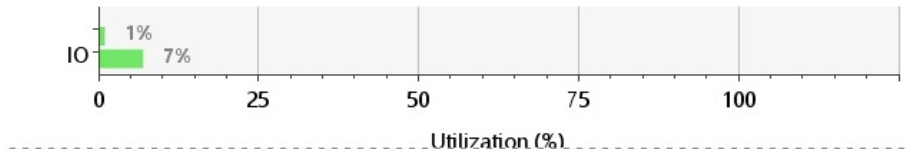



```
# run 1000ns
Time = 0 | A =    1, B =    1, P =       0
Time = 10000 | A =   41, B =   28, P =    1024
Time = 20000 | A =   23, B =   18, P =     256
Time = 30000 | A =   48, B =   19, P =     768
```

Synthesis:

Area

Resource	Utilization	Available	Utilization %
LUT	114	303600	0.04
IO	40	600	6.67



Timing

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	20	Total Number of Endpoints:	20	Total Number of Endpoints:	NA

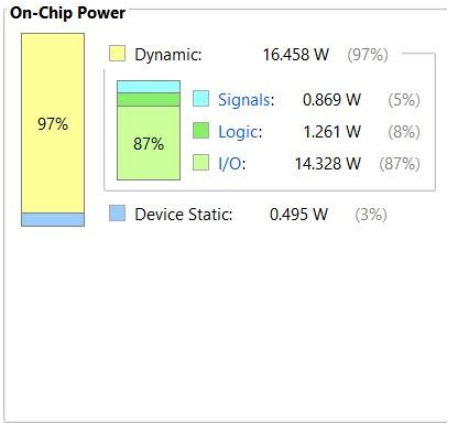
There are no user specified timing constraints.

Power

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**16.953 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**48.7°C**
Thermal Margin:	36.3°C (24.6 W)
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



Analysis of Accurate and Approximate Multiplier:

Synthesis	Accurate	Approximate
Area	Slightly larger (more full-width adders)	Slightly smaller (reduced logic complexity)
Power	Higher (92%)	Lower (87%)
Timing	Slightly higher (due to full-width additions)	Slightly better (fewer logic stages

Advantages and Disadvantages

- **Advantages:**
  - Huge area reduction: Area-Mult saves up to 45.5% LUTs compared to Vivado default synthesized multipliers.
  - Power and delay improvements: especially in approximate versions.
  - Low error rate even after truncation: Worst BER degradation in 5G wireless communication is only 8×10^{-4}, which is negligible.
  - No extra resources needed for compensation in Pro-AppT8.
  - Highly FPGA-optimized design (uses LUT6_2 merging, optimized carrychains).

- **Disadvantages:**
  - Approximate results: Pro-AppT8 is not suitable for high-precision or mission-critical applications.
  - Booth-based design complexity: Slightly more complex to design than basic array multipliers.
  - Specific to unsigned multipliers (this design doesn't handle signed multiplication directly).

Challenges:

- Trade-off Between Accuracy and Efficiency
- Compensation Logic Design
- Synthesis Result Variability

Conclusion:

This project compared an accurate and an approximate 10×10 multiplier. The accurate multiplier provided full precision but consumed more power (92%), while the approximate version, using LSB truncation and compensation, reduced power to 87% with minimal impact on timing and area. The approximate design is suitable for low-power applications where small errors are acceptable.

Reference:

[1]. H. Wang, K. Chen, C. Yan, B. Wu and W. Liu, "Hardware-Efficient Accurate and Approximate FPGA Multipliers for Error-Tolerant Applications," 2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS), Tempe, AZ, USA, 2023, pp. 977-981, doi: 10.1109/MWSCAS57524.2023.10406016. keywords: {Wireless communication;5G mobile communication; Bit error rate;Receivers;Hardware;Table lookup;Field programmable gate arrays;Field-programmable gate arrays(FPGAs);booth multiplication;approximate computing;wireless communication},