LAB 0
Systolic Array for Applying Matrix Multiplication

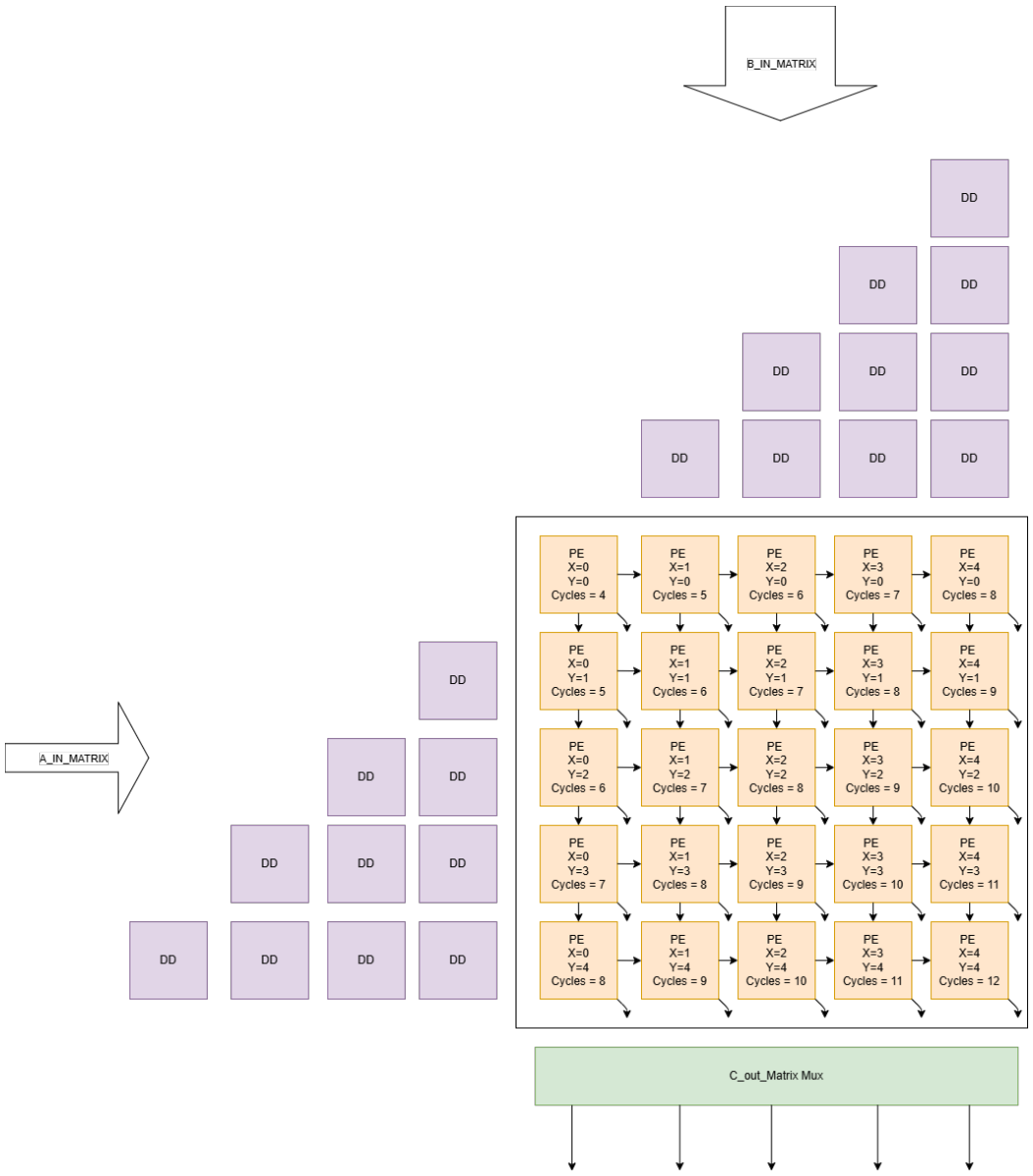
Submitted to Eng. Ahmed Abdelsalam
Made by: Ziad Alaa Anis Mohamed Kamal Kassem
Contact Info: Ziad.Kassem.eng@gmail.com
Linkedin **Github**
Project Repo

Systolic Array Project

Intro:

The systolic array is a hardware architecture optimized for matrix multiplication and similar linear algebra computations. It is composed of a two-dimensional grid of simple processing elements (PEs), where each PE performs basic operations such as multiply and accumulate. This structure allows for efficient and parallel data movement and computation, making it ideal for applications like digital signal processing (DSP) and machine learning.

Design Diagram:



Data Flow:

-Matrix A elements are streamed into the array row-wise from the **left**, and
-Matrix B elements are streamed column-wise from the **top**.

Each PE:

- Receives one element from Matrix A and one from Matrix B.
- Multiplies the two input values.
- Adds the result to its internal accumulator register.
- Passes:
    - The A value to the right neighbor.
    - The B value to the bottom neighbor.

Computation Steps

At each clock cycle:

1. The first elements of A and B enter the top-left PE (position (0,0)).
2. The PE multiplies these values and stores the result.
3. On subsequent cycles, new A and B elements enter from the left and top edges respectively.
4. Each PE performs a multiply-accumulate operation and shifts data to its neighbors.
5. After (N_SIZE + X_id +Y_id) cycles, each PE will have computed the full value of its corresponding.

Each Processing Element (PE) in the systolic array includes an internal counter that plays a critical role in synchronization and control. The counter is responsible for tracking the computation cycles within the PE. Since matrix multiplication requires summing over (2 * N_SIZE) elements (i.e., performing (N_SIZE) multiply-accumulate operations per PE), the counter keeps track of how many valid data items have passed through the PE. This ensures that the accumulation is performed exactly (N_SIZE) times before producing the final result. The counter typically increments on every clock cycle when valid data is present. Once the counter reaches the value (N_SIZE + X_id +Y_id), it signals that the PE has completed its contribution to the output matrix. At this point, the PE can assert a `valid_out` signal, indicating that the result stored in its accumulator is ready to be read. Additionally, this counter-based logic helps in stalling or disabling unnecessary operations after the computation is done, optimizing power and performance in hardware implementations.

Benefits of the Systolic Array

- **Parallelism:** All PEs work concurrently, enabling high throughput.
- **Data Locality:** No need to store full matrices in memory.
- **Scalability:** Easily expandable for larger matrix sizes.
- **Efficiency:** Suitable for hardware implementations with regular, predictable data flow.

Design Code:

```
module systolic_array #(
    parameter N_SIZE = 5 ,
    parameter DATAWIDTH = 16
) (
    input logic clk,rst_n,valid_in, // Asynchronous Active low
    input wire signed [DATAWIDTH-1:0] matrix_a_in [0:N_SIZE-1], //Data Array  [COLOMN][DATA]
    input wire signed [DATAWIDTH-1:0] matrix_b_in [0:N_SIZE-1], //Data Array  [COLOMN][DATA]
    output logic valid_out,
    output logic signed [2*DATAWIDTH-1:0]matrix_c_out[0:N_SIZE-1]
     // Data Array [Each block in last row OUTPUT][DATA]
);

//Local Variables:
logic [DATAWIDTH-1:0] matrix_a_delayed [0:N_SIZE-1][0:N_SIZE-1]; // [ROW][STAGE][DATA]
logic [DATAWIDTH-1:0] matrix_b_delayed [0:N_SIZE-1][0:N_SIZE-1]; // [COLOMN][STAGE][DATA]

logic [DATAWIDTH-1:0] a_interconnects [0:N_SIZE-1][0:N_SIZE-1]; // a_in connections
[ROW][COLOMN][DATA]
logic [DATAWIDTH-1:0] b_interconnects [0:N_SIZE-1][0:N_SIZE-1]; // b_in connections
[COLOMN][ROW][DATA]

logic valid_out_array [N_SIZE-1:0][N_SIZE-1:0];

logic signed [2*DATAWIDTH-1:0] row_c_out [0:N_SIZE-1][0:N_SIZE-1];

//Generates iterators:
genvar row_h, col_h;
genvar row_v, col_v;
//Instantiating Horizontal Flip-Flops:
generate
    for (row_h = 0; row_h < N_SIZE; row_h++) begin : GEN_ROW_H
        for (col_h = 0; col_h <= row_h; col_h++) begin : GEN_COL_H_A
            if (col_h == 0) begin
                assign matrix_a_delayed[row_h][col_h] = matrix_a_in[row_h];
            end else begin
                D_ff #(.DATAWIDTH(DATAWIDTH)) dff_a (
                    .clk(clk),
                    .rst_n(rst_n),
                    .d_in(matrix_a_delayed[row_h][col_h-1]),
                    .d_out(matrix_a_delayed[row_h][col_h])
                );
            end
        end
    end
endgenerate


//Instantiating Vertical Flip-Flops:
```

```
generate
    for (col_v = 0; col_v<N_SIZE ; col_v++ ) begin : GEN_COL_v
        for (row_v =0; row_v<=col_v ;row_v++ ) begin : GEN_ROW_V_B
            if (row_v == 0) begin
                assign matrix_b_delayed[col_v][row_v] = matrix_b_in[col_v];
            end else begin
                D_ff #(.DATAWIDTH(DATAWIDTH)) dff_b (
                    .clk(clk),
                    .rst_n(rst_n),
                    .d_in(matrix_b_delayed[col_v][row_v-1]),
                    .d_out(matrix_b_delayed[col_v][row_v]));
            end
        end
    end
endgenerate

//Connecting interconnected to delayed matrix:
generate
    for (genvar row = 0 ; row<N_SIZE ;row++ ) begin :GEN_ROW_ASSIGN
        assign a_interconnects[row][0] = matrix_a_delayed[row][row];
    end
    for (genvar col = 0 ; col<N_SIZE ;col++ ) begin :GEN_COL_ASSIGN
        assign b_interconnects[col][0] = matrix_b_delayed[col][col];
    end
endgenerate

//PE Instantiation:
genvar row_PE, col_PE;
generate
    for (row_PE = 0; row_PE<N_SIZE ; row_PE++ ) begin :GEN_PE_R
        for (col_PE = 0; col_PE<N_SIZE ; col_PE++ ) begin :GEN_PE_C
                PE #(
                .N_SIZE(N_SIZE),
                .DATAWIDTH(DATAWIDTH),
                .x_id(row_PE),
                .y_id(col_PE)
                ) pe_inst (
                .clk(clk),
                .rst_n(rst_n),
                .valid_in(valid_in),
                .a_in(a_interconnects[row_PE][col_PE]),
                .b_in(b_interconnects[col_PE][row_PE]),
                .valid_out(valid_out_array[row_PE][col_PE]),
                .a_out_right(a_interconnects[row_PE][col_PE+1]),
                .b_out_down(b_interconnects[col_PE][row_PE+1]),
                .c_out(row_c_out[row_PE][col_PE]));
       end
    end
endgenerate
// valid_out logic: high when any row's last PE is valid
```

```
always_comb begin
    valid_out = 0;
    for (int row = 0; row < N_SIZE; row++) begin
        if (valid_out_array[row][N_SIZE-1]) begin
            valid_out = 1;
        end
    end
end

//OUTPUT MUX
always_comb begin
    matrix_c_out = '{default:0};
    for (int row = 0; row < N_SIZE; row++) begin
        if (valid_out_array[row][N_SIZE-1]) begin
            for (int col = 0; col < N_SIZE; col++) begin
                matrix_c_out[col] = row_c_out[row][col];
            end
        end
    end
end

endmodule
```

```
module PE #(
    parameter N_SIZE = 5,
    parameter DATAWIDTH = 16,
    parameter x_id = 0 ,
    parameter y_id = 0
) (
    input logic clk,rst_n,valid_in,
    input logic signed [DATAWIDTH-1:0] a_in,b_in,
    output logic valid_out,
    output logic signed [(DATAWIDTH)-1:0]b_out_down,
    output logic signed [(DATAWIDTH)-1:0]a_out_right,
    output logic signed [(2*DATAWIDTH)-1:0]c_out
);

logic [N_SIZE:0] internal_counter;
logic [(2*DATAWIDTH)-1:0] sum;
logic valid_in_flag_holder;
    always @(posedge clk , negedge rst_n)begin
       if (!rst_n) begin
            internal_counter <= 0;
            sum <= 0;
            valid_in_flag_holder <=0;
            c_out<=0;
       end
       else begin
        valid_out<=0;
        if (valid_in || valid_in_flag_holder) begin
            valid_in_flag_holder<=1;
            if (internal_counter < N_SIZE + x_id + y_id) begin
                internal_counter <= internal_counter + 1;
                if (internal_counter >= x_id +y_id) begin
                    if (internal_counter == N_SIZE + x_id + y_id - 1) begin
                        valid_out <= 1;
                        valid_in_flag_holder <= 0;    internal_counter <= 0;
                        c_out <= sum + (a_in*b_in);
                        sum <= 0;
                    end
                    else begin
                        sum <= sum + (a_in*b_in);
                    end
                    a_out_right <= a_in;   b_out_down <= b_in;
                end
            end
            else begin

            end
        end
       end
    end
endmodule
```

```
module D_ff #(
    parameter DATAWIDTH = 16
) (
    input logic clk,rst_n,
    input logic [DATAWIDTH-1:0] d_in,
    output logic [DATAWIDTH-1:0] d_out
);

always @(posedge clk , negedge rst_n) begin
    if(!rst_n) d_out <= 0;
    else d_out <= d_in;
end

endmodule
```

Verification Code:

```
module systolic_array_tb;
    localparam N_SIZE = 5;
    localparam DATAWIDTH = 16;

    logic clk, rst_n, valid_in;
    logic signed [DATAWIDTH-1:0] matrix_a_in [N_SIZE-1:0];
    logic signed [DATAWIDTH-1:0] matrix_b_in [N_SIZE-1:0];
    logic valid_out;
    logic signed [2*DATAWIDTH-1:0] matrix_c_out [N_SIZE-1:0];

    // Instantiate DUT
    systolic_array #(
        .N_SIZE(N_SIZE),
        .DATAWIDTH(DATAWIDTH)
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .valid_in(valid_in),
        .matrix_a_in(matrix_a_in),
        .matrix_b_in(matrix_b_in),
        .valid_out(valid_out),
        .matrix_c_out(matrix_c_out)
    );

    // Clock generation
    initial begin
        clk=0;
        forever begin
            #1 clk=~clk;
        end
    end

    // Test vectors
    initial begin
        rst_n = 0;
        valid_in = 0;
        matrix_a_in = '{default:0};
        matrix_b_in = '{default:0};
        repeat(1) @(negedge clk);
        rst_n = 1;
```

```
        // Test Case 1 5*5
        matrix_a_in = '{1, 2, 3, 4, 5};
        matrix_b_in = '{25, 24, 23, 22, 21};
        valid_in = 1;
        repeat(1) @(negedge clk);

        matrix_a_in = '{6, 7, 8, 9, 10};
        matrix_b_in = '{20, 19, 18, 17, 16};
        repeat(1) @(negedge clk);

        matrix_a_in = '{11, 12, 13, 14, 15};
        matrix_b_in = '{15, 14, 13, 12, 11};
        repeat(1) @(negedge clk);

        matrix_a_in = '{16, 17, 18, 19, 20};
        matrix_b_in = '{10, 9, 8, 7, 6};
        repeat(1) @(negedge clk);

        matrix_a_in = '{21, 22, 23, 24, 25};
        matrix_b_in = '{5, 4, 3, 2, 1};
        repeat(1) @(negedge clk);

        valid_in = 0;
        repeat(13) @(negedge clk);
        $display("Test 1 should be done");
        $stop;
        //================================================================
        // Test Case 2 5*5
        matrix_a_in = '{5, 4, 3, 2, 1};
        matrix_b_in = '{1, 1, 1, 1, 1};
        valid_in = 1;
        repeat(1) @(negedge clk);

        matrix_a_in = '{10, 9, 8, 7, 6};
        matrix_b_in = '{2, 2, 2, 2, 2};
        repeat(1) @(negedge clk);

        matrix_a_in = '{15, 14, 13, 12, 11};
        matrix_b_in = '{3, 3, 3, 3, 3};
        repeat(1) @(negedge clk);

        matrix_a_in = '{20, 19, 18, 17, 16};
        matrix_b_in = '{4, 4, 4, 4, 4};
        repeat(1) @(negedge clk);

        matrix_a_in = '{25, 24, 23, 22, 21};
        matrix_b_in = '{5, 5, 5, 5, 5};
        repeat(1) @(negedge clk);

        valid_in = 0;
```

```
        repeat(13) @(negedge clk);
        $display("Test 2 should be done");
        $stop;
        //================================================================
        // Test Case 3 5*5
        matrix_a_in = '{1, 3, 5, 7, 9};
        matrix_b_in = '{5, 4, 3, 2, 1};
        valid_in = 1;
        repeat(1) @(negedge clk);

        matrix_a_in = '{2, 4, 6, 8, 10};
        matrix_b_in = '{10, 9, 8, 7, 6};
        repeat(1) @(negedge clk);

        matrix_a_in = '{11, 13, 15, 17, 19};
        matrix_b_in = '{15, 14, 13, 12, 11};
        repeat(1) @(negedge clk);

        matrix_a_in = '{12, 14, 16, 18, 20};
        matrix_b_in = '{20, 19, 18, 17, 16};
        repeat(1) @(negedge clk);

        matrix_a_in = '{21, 22, 23, 24, 25};
        matrix_b_in = '{25, 24, 23, 22, 21};
        repeat(1) @(negedge clk);

        valid_in = 0;
        repeat(13) @(negedge clk);
        $display("Test 3 should be done");
        $stop;
        //================================================================
```

```
// Test Case 4 5*5
        matrix_a_in = '{1, 0, 1, 0, 1};
        matrix_b_in = '{1, 2, 3, 4, 5};
        valid_in = 1;
        repeat(1) @(negedge clk);

        matrix_a_in = '{0, 1, 0, 1, 0};
        matrix_b_in = '{5, 4, 3, 2, 1};
        repeat(1) @(negedge clk);

        matrix_a_in = '{1, 1, 1, 1, 1};
        matrix_b_in = '{1, 2, 3, 4, 5};
        repeat(1) @(negedge clk);

        matrix_a_in = '{2, 2, 2, 2, 2};
        matrix_b_in = '{5, 4, 3, 2, 1};
        repeat(1) @(negedge clk);

        matrix_a_in = '{3, 3, 3, 3, 3};
        matrix_b_in = '{1, 2, 3, 4, 5};
        repeat(1) @(negedge clk);

        valid_in = 0;
        repeat(13) @(negedge clk);
        $display("Test 4 should be done");
        $stop;
```

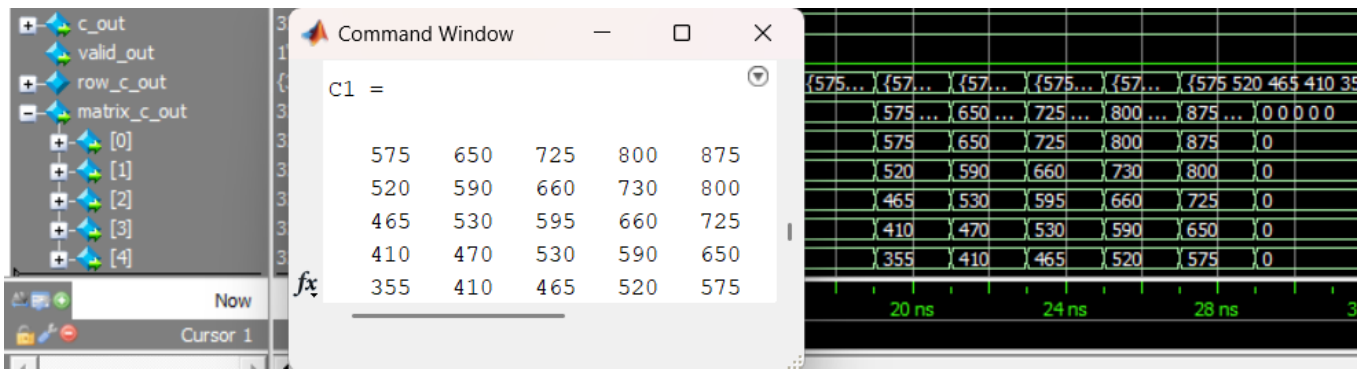
```
//================================================================
        // Test Case 5 5*5
        matrix_a_in = '{1, 1, 1, 1, 1};
        matrix_b_in = '{1, 2, 3, 4, 5};
        valid_in = 1;
        repeat(1) @(negedge clk);

        matrix_a_in = '{2, 2, 2, 2, 2};
        matrix_b_in = '{6, 7, 8, 9, 10};
        repeat(1) @(negedge clk);

        matrix_a_in = '{3, 3, 3, 3, 3};
        matrix_b_in = '{11, 12, 13, 14, 15};
        repeat(1) @(negedge clk);

        matrix_a_in = '{4, 4, 4, 4, 4};
        matrix_b_in = '{16, 17, 18, 19, 20};
        repeat(1) @(negedge clk);

        matrix_a_in = '{5, 5, 5, 5, 5};
        matrix_b_in = '{21, 22, 23, 24, 25};
        repeat(1) @(negedge clk);

        valid_in = 0;
        repeat(13) @(negedge clk);
        $display("Test 5 should be done");
        $stop;
        end


endmodule
```

Simulation Output (Questasim):

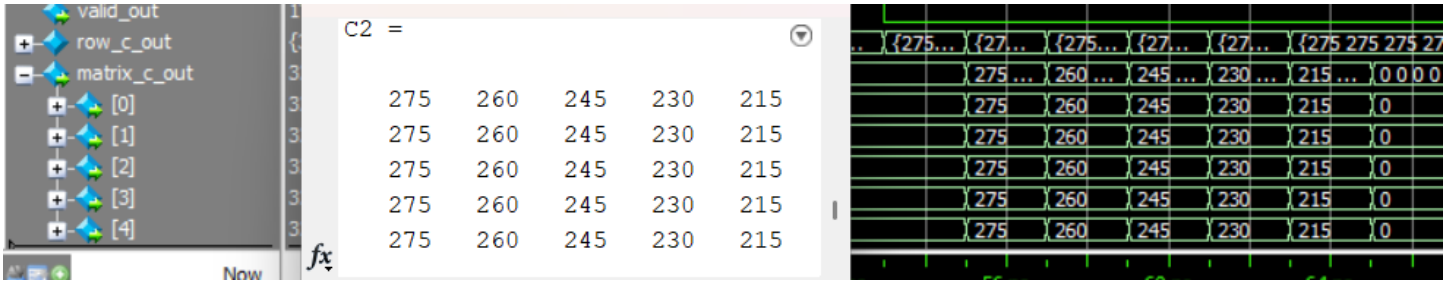
I used Matlab as a golden model to compare output validation & correctness:

Each snippet Is integrated with Matlab output of same test case:

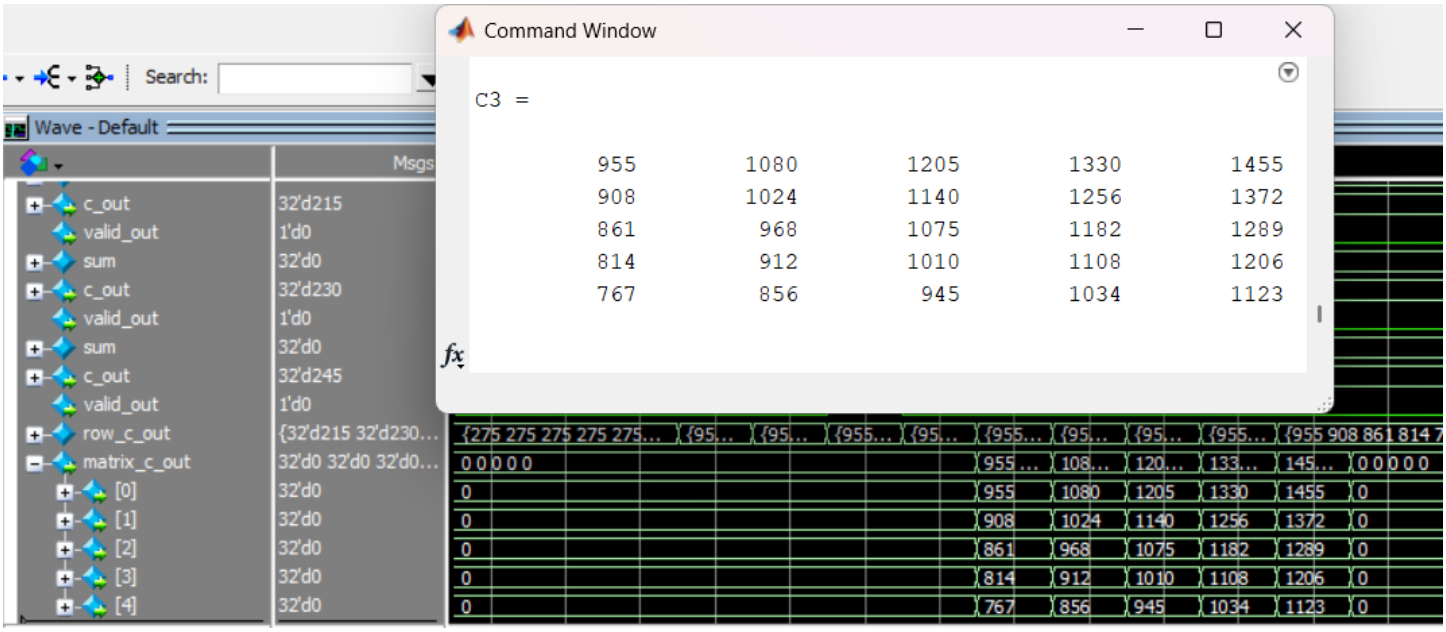
Case 1:



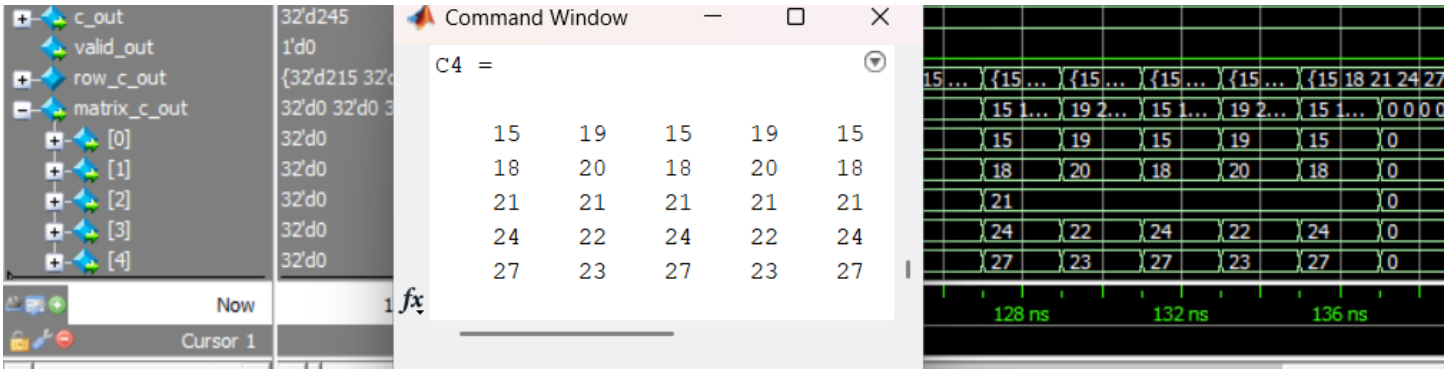
Case 2:



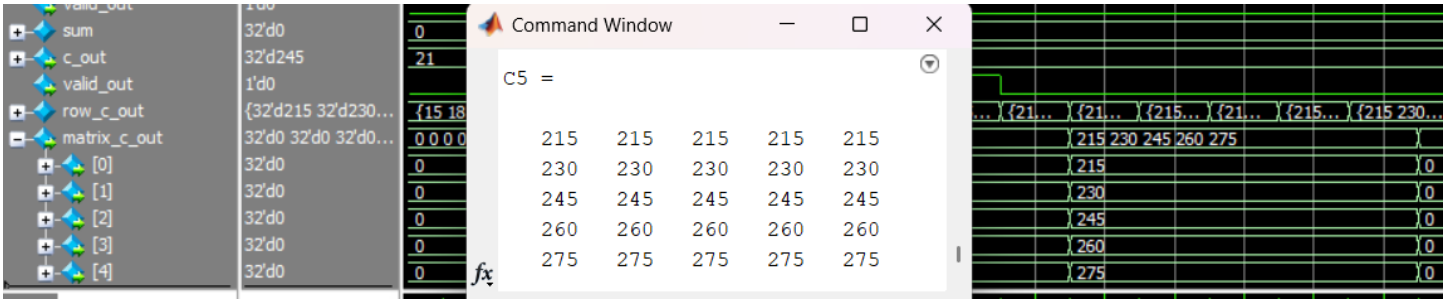
Case 3:



Case 4:



Case 5:



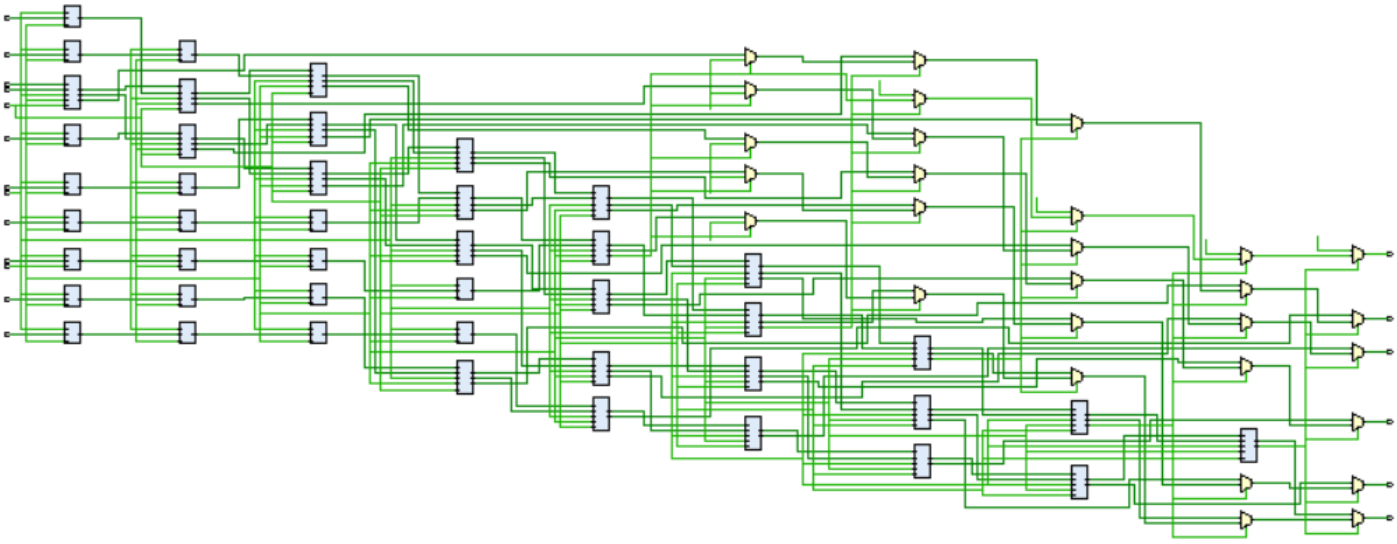
Vivado Output:

FPGA USED: **SYNTHESIZED DESIGN** - xc7a35ticpg236-1L

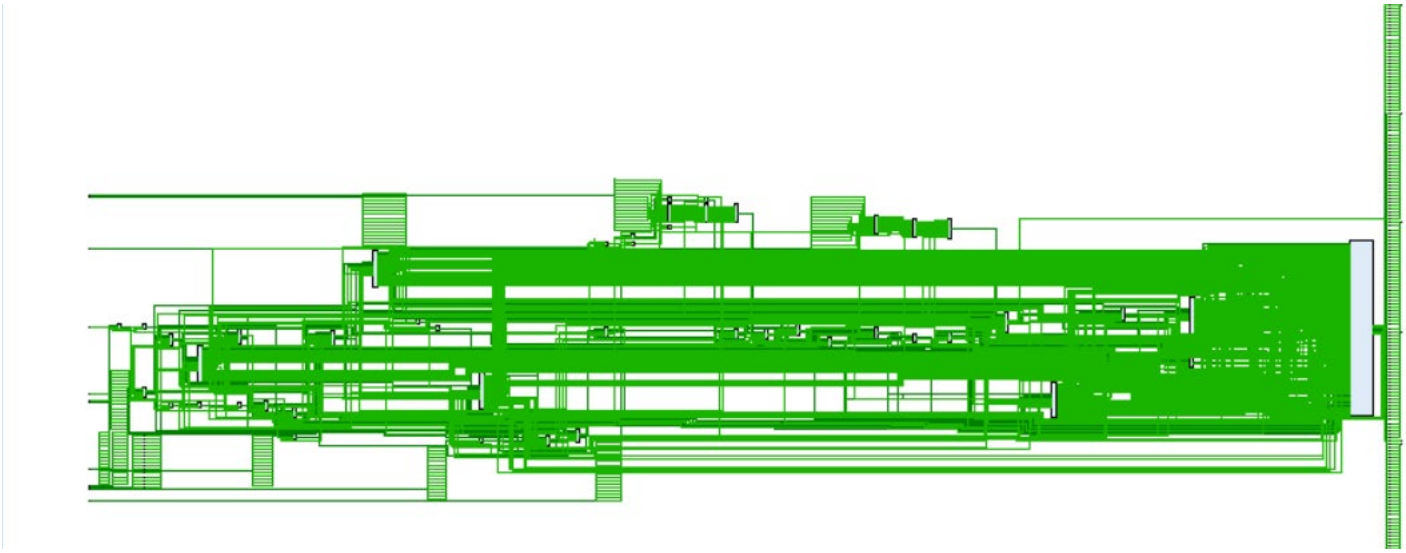
Constraint File for Timing Analysis:

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]
```

Elaborated RLT Schematic:



Synthesis :



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.625 ns	Worst Hold Slack (WHS):	0.144 ns	Worst Pulse Width Slack (WPWS):	4.020 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5133	Total Number of Endpoints:	5133	Total Number of Endpoints:	2492

All user specified timing constraints are met.

Power Report:

Summary

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:	**0.184 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.9°C**
Thermal Margin:	74.1°C (14.7 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity