Logic Design Laboratory term project

SUBJECT: Logic Design Laboratory

Professor : Il Yong Jeon

Team Number : 3

NAME(STUDENT NUMBER): TaeHyoung Kim (2023312453), SeoYeung Byun(2024315016), MinHyuk Jeon(2019312305), MinJun Kim(2022316130)

Affiliation : Sungkyunkwan University, School of Mechanical Engineering

Date : 2025.07.12

I. Introduction

1. Objective

The purpose of this project is to understand the computational principles and processes of 2D Convolution, and to implement them in hardware using Verilog HDL. First, a Single Processing Element (Single PE) is designed, and 3×3 and 2×2 Systolic Array structures are configured based on this to perform efficient matrix multiplication operations. The computational results of each array are sequentially output through a 7-Segment Display, and the results can be confirmed through an actual FPGA board. In addition, the effectiveness of hardware parallel processing is analyzed by comparing the speed difference between the Single PE and Systolic Array structures when performing the same operation. The entire system is configured in a Structural Modeling manner to clearly understand the operating principles and interactions of each module. Through this process, the goal is to deepen the understanding of Verilog-based digital design capabilities and hardware operation optimization.

2. 2D convolution

2D convolution is the main operation of DNN. It uses a 4x4 input matrix and a 3x3 filter matrix to generate a 2x2 output matrix. Each output matrix value is calculated as follows.

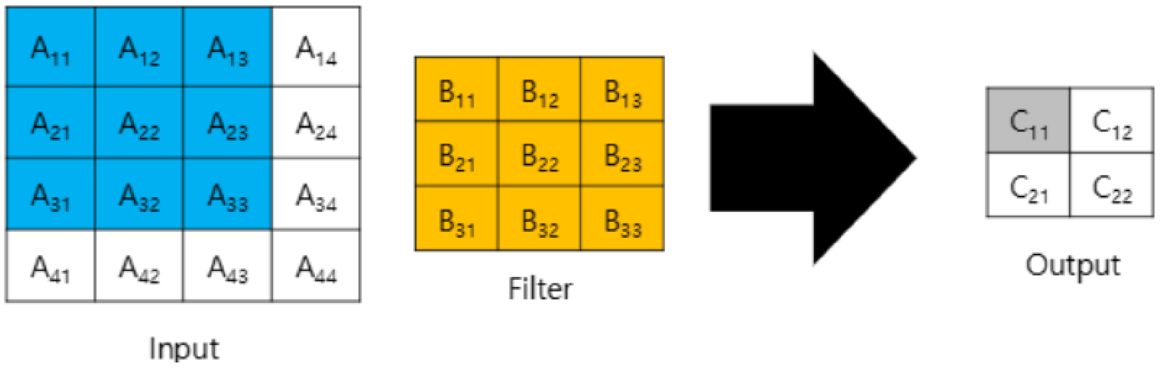


Fig 1. Process of 2D Convolution

$$C_{11} = A_{11} * B_{33} + A_{12} * B_{32} + A_{13} * B_{31} * A_{21} * B_{23} + A_{22} * B_{22} + A_{23} * B_{21} + A_{21} * B_{13} + A_{32} * B_{12} + A_{33} * B_{11}$$

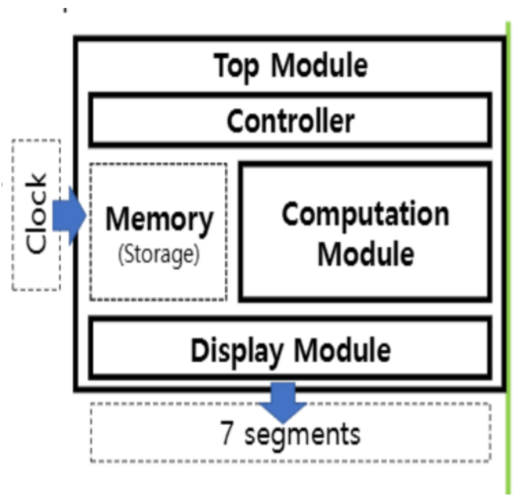
$$C_{12} = A_{12} * B_{33} + A_{13} * B_{32} + A_{14} * B_{31} + A_{22} * B_{23} + A_{23} * B_{22} + A_{24} * B_{21} + A_{32} * B_{13} + A_{33} * B_{12} + A_{34} * B_{11}$$

$$C_{21} = A_{21} * B_{33} + A_{22} * B_{32} + A_{23} * B_{31} + A_{31} * B_{23} + A_{32} * B_{22} + A_{33} * B_{21} + A_{41} * B_{13} + A_{42} * B_{12} + A_{43} * B_{11}$$

$$C_{22} = A_{22} * B_{33} + A_{23} * B_{32} + A_{24} * B_{31} + A_{32} * B_{23} + A_{33} * B_{22} + A_{34} * B_{21} + A_{42} * B_{13} + A_{43} * B_{12} + A_{44} * B_{11}$$

3. Plan for implementation and simulation

The outline of the calculator is as follows.



First, the controller controls the operation order of each module, and each module must be connected to the top module and work organically. The memory module stores the input and connects these values to the calculation module. The calculation module has a single PE, a 3x3 systolic array, and a 2x2 systolic array, which are used to perform the convolution operation of the input matrix and the filter matrix. The result calculated in this way is confirmed to our eyes through the display module on the 7-segment.

Fig 2. Structure of the Whole System

II. Module Description

1. Controller

1.1 Controller Theoretical Approach

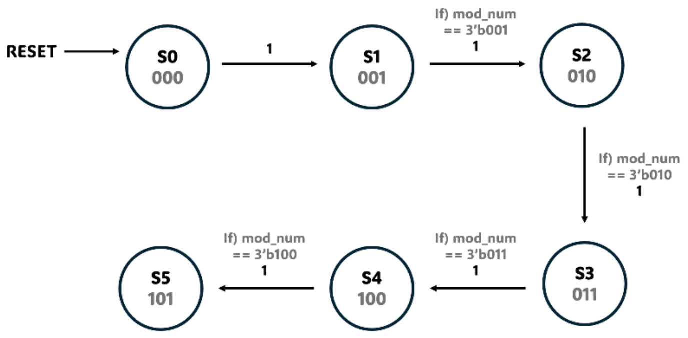
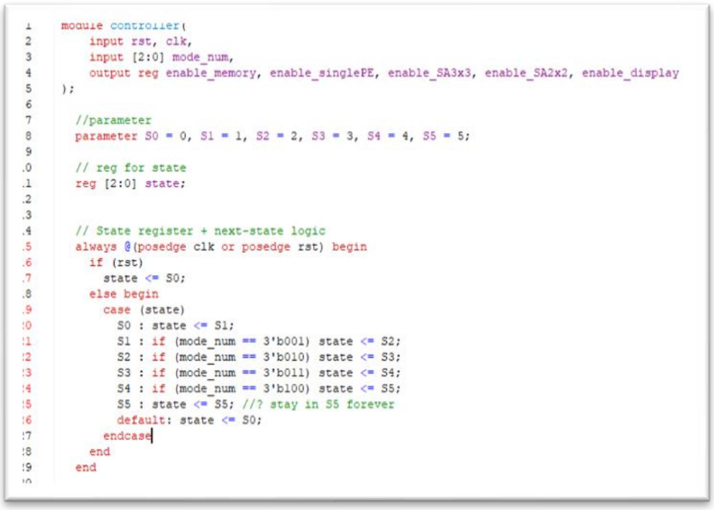


Fig 3. Moore Machine State Diagram

The controller is implemented using a Moore machine to manage the execution flow of the system. Compared to a Mealy machine, the Moore machine provides more predictable output transitions because outputs depend solely on the current state. The state diagram consists of six states (S0–S5), each corresponding to a specific module operation. **S0**: Initialization, **S1**: Memory operation, **S2**: Single PE operation, **S3**: 3x3 Systolic Array operation, **S4**: 2x2 Systolic Array operation, **S5**: Display operation

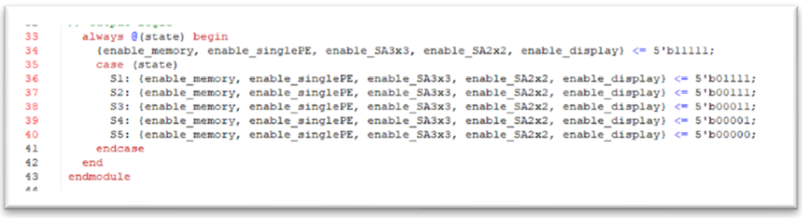
1.2 Controller Implementation Code



```
module controller(
    input rst, clk,
    input [2:0] mode_num,
    output reg enable_memory, enable_singlePE, enable_SA3x3, enable_SA2x2, enable_display
);

  //parameter
  parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4, S5 = 5;

  // reg for state
  reg [2:0] state;


  // State register + next-state logic
  always @(posedge clk or posedge rst) begin
    if (rst)
      state <= S0;
    else begin
      case (state)
        S0 : state <= S1;
        S1 : if (mode_num == 3'b001) state <= S2;
        S2 : if (mode_num == 3'b010) state <= S3;
        S3 : if (mode_num == 3'b011) state <= S4;
        S4 : if (mode_num == 3'b100) state <= S5;
        S5 : state <= S5; //? stay in S5 forever
        default: state <= S0;
      endcase
    end
  end
```

The controller transitions sequentially from S0 to S5. Each module asserts its **done signal** upon completion, which triggers the transition to the next state in the **Top module** via the **MODE_NUM** signal. Once S5 is reached, the controller remains in the display state without restarting the sequence.

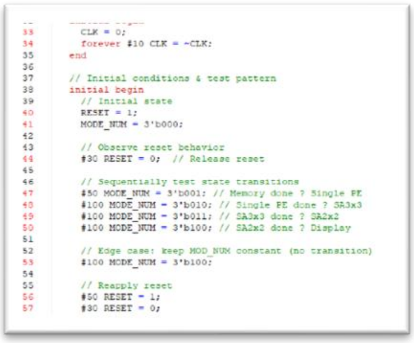
Fig 4. Controller Implementation Code



```
  always @(state) begin
    {enable_memory, enable_singlePE, enable_SA3x3, enable_SA2x2, enable_display} <= 5'b11111;
    case (state)
      S1: {enable_memory, enable_singlePE, enable_SA3x3, enable_SA2x2, enable_display} <= 5'b01111;
      S2: {enable_memory, enable_singlePE, enable_SA3x3, enable_SA2x2, enable_display} <= 5'b00111;
      S3: {enable_memory, enable_singlePE, enable_SA3x3, enable_SA2x2, enable_display} <= 5'b00011;
      S4: {enable_memory, enable_singlePE, enable_SA3x3, enable_SA2x2, enable_display} <= 5'b00001;
      S5: {enable_memory, enable_singlePE, enable_SA3x3, enable_SA2x2, enable_display} <= 5'b00000;
    endcase
  end
endmodule
```

Enable signals determine the on/off state of each module. In this design, **enable = 0** turns a module **on**, and **enable = 1** turns it **off** because these signals are directly connected to each module's reset input. As a result, modules from earlier states stay active while transitioning to subsequent states.

1.3 Controller testbench and simulation



```
    CLK = 0;
    forever #10 CLK = ~CLK:
  end

  // Initial conditions & test pattern
  initial begin
    // Initial state
    RESET = 1;
    MODE_NUM = 3'b000;

    // Observe reset behavior
    #30 RESET = 0;  // Release reset

    // Sequentially test state transitions
    #50 MODE_NUM = 3'b001; // Memory done ? Single PE
    #100 MODE_NUM = 3'b010; // Single PE done ? SA3x3
    #100 MODE_NUM = 3'b011; // SA3x3 done ? SA2x2
    #100 MODE_NUM = 3'b100; // SA2x2 done ? Display

    // Edge case: keep MOD_NUM constant (no transition)
    #100 MODE_NUM = 3'b100;

    // Reapply reset
    #50 RESET = 1;
    #30 RESET = 0;
```

The controller was tested by incrementally increasing MODE_NUM every 100ns to verify state transitions. Simulations confirmed that state transitions occur on the next positive clock edge after MODE_NUM changes, and previously activated modules remain enabled as intended. Waveform analysis validated the correct timing and activation of enable signals.

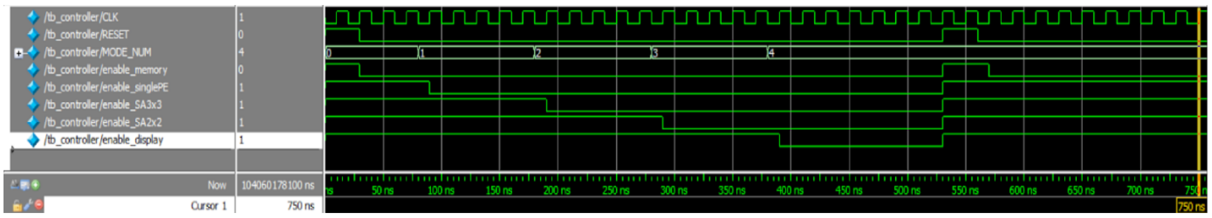


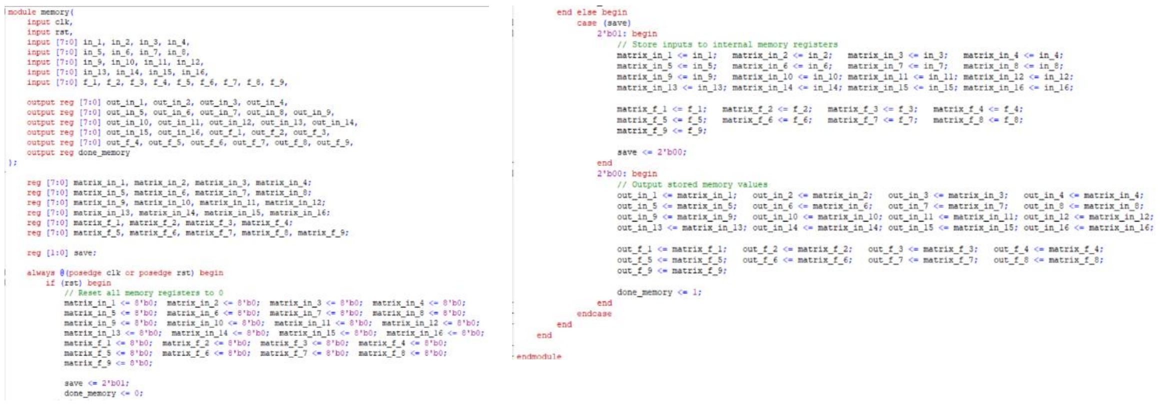
Fig 5. Controller testbench and simulation

2. Memory Module

2.1 Memory Theoretical Approach

Memory is a module that simply receives 16 values of the input matrix and 9 values of the filter matrix, temporarily stores them, and then outputs them. Memory operates for at least two positive clock edges. In the first clock, the input values are stored in the storage inside the memory module. In the next clock, the values inside the memory module are output as 25 outputs.

2.2 Memory Module

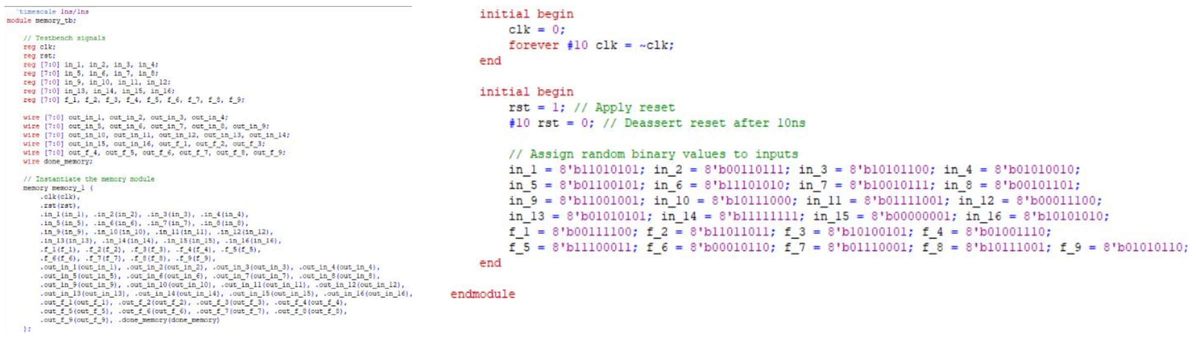


```verilog
module memory(
    input clk,
    input rst,
    input [7:0] in_1, in_2, in_3, in_4,
    input [7:0] in_5, in_6, in_7, in_8,
    input [7:0] in_9, in_10, in_11, in_12,
    input [7:0] in_13, in_14, in_15, in_16,
    input [7:0] f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9,

    output reg [7:0] out_in_1, out_in_2, out_in_3, out_in_4,
    output reg [7:0] out_in_5, out_in_6, out_in_7, out_in_8, out_in_9,
    output reg [7:0] out_in_10, out_in_11, out_in_12, out_in_13, out_in_14,
    output reg [7:0] out_in_15, out_in_16, out_f_1, out_f_2, out_f_3,
    output reg [7:0] out_f_4, out_f_5, out_f_6, out_f_7, out_f_8, out_f_9,
    output reg done_memory
);

    reg [7:0] matrix_in_1, matrix_in_2, matrix_in_3, matrix_in_4;
    reg [7:0] matrix_in_5, matrix_in_6, matrix_in_7, matrix_in_8;
    reg [7:0] matrix_in_9, matrix_in_10, matrix_in_11, matrix_in_12;
    reg [7:0] matrix_in_13, matrix_in_14, matrix_in_15, matrix_in_16;
    reg [7:0] matrix_f_1, matrix_f_2, matrix_f_3, matrix_f_4;
    reg [7:0] matrix_f_5, matrix_f_6, matrix_f_7, matrix_f_8, matrix_f_9;

    reg [1:0] save;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // Reset all memory registers to 0
            matrix_in_1 <= 8'b0;  matrix_in_2 <= 8'b0;  matrix_in_3 <= 8'b0;  matrix_in_4 <= 8'b0;
            matrix_in_5 <= 8'b0;  matrix_in_6 <= 8'b0;  matrix_in_7 <= 8'b0;  matrix_in_8 <= 8'b0;
            matrix_in_9 <= 8'b0;  matrix_in_10 <= 8'b0;  matrix_in_11 <= 8'b0;  matrix_in_12 <= 8'b0;
            matrix_in_13 <= 8'b0;  matrix_in_14 <= 8'b0;  matrix_in_15 <= 8'b0;  matrix_in_16 <= 8'b0;
            matrix_f_1 <= 8'b0;  matrix_f_2 <= 8'b0;  matrix_f_3 <= 8'b0;  matrix_f_4 <= 8'b0;
            matrix_f_5 <= 8'b0;  matrix_f_6 <= 8'b0;  matrix_f_7 <= 8'b0;  matrix_f_8 <= 8'b0;
            matrix_f_9 <= 8'b0;

            save <= 2'b01;
            done_memory <= 0;
        end else begin
            case (save)
                2'b01: begin
                    // Store inputs to internal memory registers
                    matrix_in_1 <= in_1;   matrix_in_2 <= in_2;   matrix_in_3 <= in_3;   matrix_in_4 <= in_4;
                    matrix_in_5 <= in_5;   matrix_in_6 <= in_6;   matrix_in_7 <= in_7;   matrix_in_8 <= in_8;
                    matrix_in_9 <= in_9;   matrix_in_10 <= in_10; matrix_in_11 <= in_11; matrix_in_12 <= in_12;
                    matrix_in_13 <= in_13; matrix_in_14 <= in_14; matrix_in_15 <= in_15; matrix_in_16 <= in_16;

                    matrix_f_1 <= f_1;   matrix_f_2 <= f_2;   matrix_f_3 <= f_3;   matrix_f_4 <= f_4;
                    matrix_f_5 <= f_5;   matrix_f_6 <= f_6;   matrix_f_7 <= f_7;   matrix_f_8 <= f_8;
                    matrix_f_9 <= f_9;

                    save <= 2'b00;
                end
                2'b00: begin
                    // Output stored memory values
                    out_in_1 <= matrix_in_1;   out_in_2 <= matrix_in_2;   out_in_3 <= matrix_in_3;   out_in_4 <= matrix_in_4;
                    out_in_5 <= matrix_in_5;   out_in_6 <= matrix_in_6;   out_in_7 <= matrix_in_7;   out_in_8 <= matrix_in_8;
                    out_in_9 <= matrix_in_9;   out_in_10 <= matrix_in_10; out_in_11 <= matrix_in_11; out_in_12 <= matrix_in_12;
                    out_in_13 <= matrix_in_13; out_in_14 <= matrix_in_14; out_in_15 <= matrix_in_15; out_in_16 <= matrix_in_16;

                    out_f_1 <= matrix_f_1;   out_f_2 <= matrix_f_2;   out_f_3 <= matrix_f_3;   out_f_4 <= matrix_f_4;
                    out_f_5 <= matrix_f_5;   out_f_6 <= matrix_f_6;   out_f_7 <= matrix_f_7;   out_f_8 <= matrix_f_8;
                    out_f_9 <= matrix_f_9;

                    done_memory <= 1;
                end
            endcase
        end
    end

endmodule
```

Fig 6. Memory Module Code

The memory module receives input matrix values (in_X) and filter values (f_X), storing them in matrix_in_X and matrix_f_X, respectively. The stored values are then output through out_in_X and out_f_X. The **save** signal determines whether to store or output data, while **done_memory** indicates whether the output process is complete. Initially, save is set to 1 and done_memory to 0. The always block synchronizes operations with clk and rst. When rst is 1, all values are reset to 0. Otherwise, a case statement handles storing values when save is 1 and outputting values when save is 0. After the output is complete, done_memory is set to 1. In this way, the memory module is implemented in two distinct phases: storing and outputting.

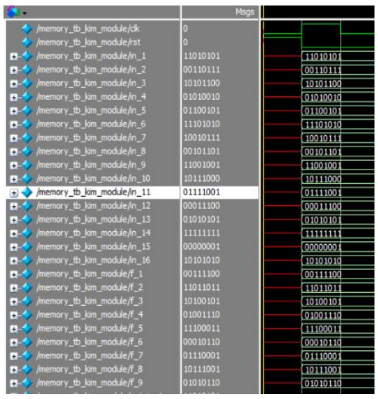
2.3 Memory Module Testbench and Simulation



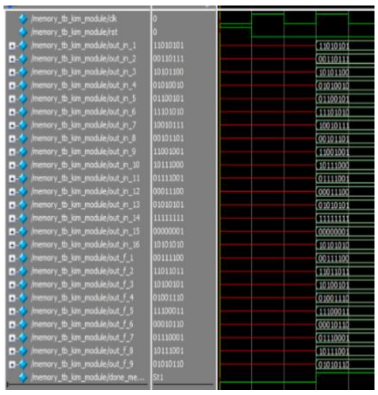
```verilog
 `timescale 1ns/1ns
module memory_tb;

    // Testbench signals
    reg clk;
    reg rst;
    reg [7:0] in_1, in_2, in_3, in_4;
    reg [7:0] in_5, in_6, in_7, in_8;
    reg [7:0] in_9, in_10, in_11, in_12;
    reg [7:0] in_13, in_14, in_15, in_16;
    reg [7:0] f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9;

    wire [7:0] out_in_1, out_in_2, out_in_3, out_in_4;
    wire [7:0] out_in_5, out_in_6, out_in_7, out_in_8, out_in_9;
    wire [7:0] out_in_10, out_in_11, out_in_12, out_in_13, out_in_14;
    wire [7:0] out_in_15, out_in_16, out_f_1, out_f_2, out_f_3;
    wire [7:0] out_f_4, out_f_5, out_f_6, out_f_7, out_f_8, out_f_9;
    wire done_memory;

    // Instantiate the memory module
    memory memory_1 (
        .clk(clk),
        .rst(rst),
        .in_1(in_1), .in_2(in_2), .in_3(in_3), .in_4(in_4),
        .in_5(in_5), .in_6(in_6), .in_7(in_7), .in_8(in_8),
        .in_9(in_9), .in_10(in_10), .in_11(in_11), .in_12(in_12),
        .in_13(in_13), .in_14(in_14), .in_15(in_15), .in_16(in_16),
        .f_1(f_1), .f_2(f_2), .f_3(f_3), .f_4(f_4), .f_5(f_5),
        .f_6(f_6), .f_7(f_7), .f_8(f_8), .f_9(f_9),
        .out_in_1(out_in_1), .out_in_2(out_in_2), .out_in_3(out_in_3), .out_in_4(out_in_4),
        .out_in_5(out_in_5), .out_in_6(out_in_6), .out_in_7(out_in_7), .out_in_8(out_in_8),
        .out_in_9(out_in_9), .out_in_10(out_in_10), .out_in_11(out_in_11), .out_in_12(out_in_12),
        .out_in_13(out_in_13), .out_in_14(out_in_14), .out_in_15(out_in_15), .out_in_16(out_in_16),
        .out_f_1(out_f_1), .out_f_2(out_f_2), .out_f_3(out_f_3), .out_f_4(out_f_4),
        .out_f_5(out_f_5), .out_f_6(out_f_6), .out_f_7(out_f_7), .out_f_8(out_f_8),
        .out_f_9(out_f_9), .done_memory(done_memory)
    );

    initial begin
        clk = 0;
        forever #10 clk = ~clk;
    end

    initial begin
        rst = 1; // Apply reset
        #10 rst = 0; // Deassert reset after 10ns

        // Assign random binary values to inputs
        in_1 = 8'b11010101; in_2 = 8'b00110111; in_3 = 8'b10101100; in_4 = 8'b01010010;
        in_5 = 8'b01100101; in_6 = 8'b11101010; in_7 = 8'b10010111; in_8 = 8'b00101101;
        in_9 = 8'b11001001; in_10 = 8'b10111000; in_11 = 8'b01111001; in_12 = 8'b00011100;
        in_13 = 8'b01010101; in_14 = 8'b11111111; in_15 = 8'b00000001; in_16 = 8'b10101010;
        f_1 = 8'b00111100; f_2 = 8'b11011011; f_3 = 8'b10100101; f_4 = 8'b01001110;
        f_5 = 8'b11100011; f_6 = 8'b00010110; f_7 = 8'b01110001; f_8 = 8'b10111001; f_9 = 8'b01010110;
    end

endmodule
```

Fig 7. Memory Module Testbench Code

In Fig 7, the variables to be input to the memory and the variables to be output were specified, and the values of the input matrix and the values of the filter matrix were randomly set and input.



```
                                        Msgs
/memory_tb_kim_module/clk              0
/memory_tb_kim_module/rst              0
/memory_tb_kim_module/in_1             11010101
/memory_tb_kim_module/in_2             00110111
/memory_tb_kim_module/in_3             10101100
/memory_tb_kim_module/in_4             01010010
/memory_tb_kim_module/in_5             01100101
/memory_tb_kim_module/in_6             11101010
/memory_tb_kim_module/in_7             10010111
/memory_tb_kim_module/in_8             00101101
/memory_tb_kim_module/in_9             11001001
/memory_tb_kim_module/in_10            10111000
/memory_tb_kim_module/in_11            01111001
/memory_tb_kim_module/in_12            00011100
/memory_tb_kim_module/in_13            01010101
/memory_tb_kim_module/in_14            11111111
/memory_tb_kim_module/in_15            00000001
/memory_tb_kim_module/in_16            10101010
/memory_tb_kim_module/f_1              00111100
/memory_tb_kim_module/f_2              11011011
/memory_tb_kim_module/f_3              10100101
/memory_tb_kim_module/f_4              01001110
/memory_tb_kim_module/f_5              11100011
/memory_tb_kim_module/f_6              00010110
/memory_tb_kim_module/f_7              01110001
/memory_tb_kim_module/f_8              10111001
/memory_tb_kim_module/f_9              01010110
```



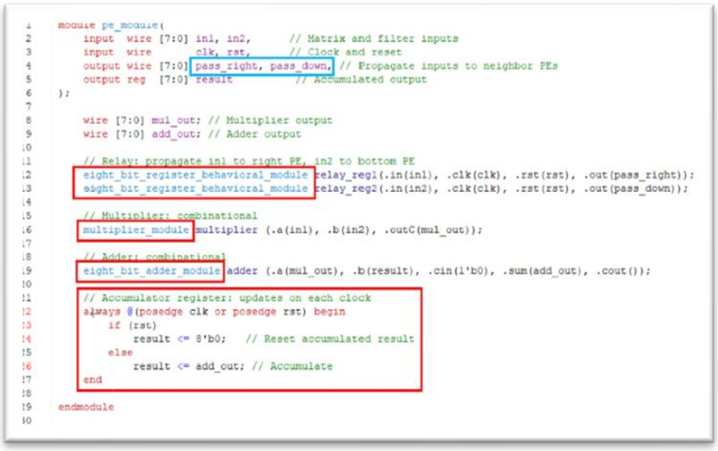
```
/memory_tb_kim_module/clk              0
/memory_tb_kim_module/rst              0
/memory_tb_kim_module/out_in_1         11010101
/memory_tb_kim_module/out_in_2         00110111
/memory_tb_kim_module/out_in_3         10101100
/memory_tb_kim_module/out_in_4         01010010
/memory_tb_kim_module/out_in_5         01100101
/memory_tb_kim_module/out_in_6         11101010
/memory_tb_kim_module/out_in_7         10010111
/memory_tb_kim_module/out_in_8         00101101
/memory_tb_kim_module/out_in_9         11001001
/memory_tb_kim_module/out_in_10        10111000
/memory_tb_kim_module/out_in_11        01111001
/memory_tb_kim_module/out_in_12        00011100
/memory_tb_kim_module/out_in_13        01010101
/memory_tb_kim_module/out_in_14        11111111
/memory_tb_kim_module/out_in_15        00000001
/memory_tb_kim_module/out_in_16        10101010
/memory_tb_kim_module/out_f_1          00111100
/memory_tb_kim_module/out_f_2          11011011
/memory_tb_kim_module/out_f_3          10100101
/memory_tb_kim_module/out_f_4          01001110
/memory_tb_kim_module/out_f_5          11100011
/memory_tb_kim_module/out_f_6          00010110
/memory_tb_kim_module/out_f_7          01110001
/memory_tb_kim_module/out_f_8          10111001
/memory_tb_kim_module/out_f_9          01010110
/memory_tb_kim_module/done_me...       St1
```

As a result of running the testbench, if we see Fig 8, input value is stored on the first clock and output is generated on the second clock. And at the same time as the output is generated, the done_memory value becomes 1, indicating that the output is complete.

Fig 8. Memory Module Simulation Results

3. PE Module

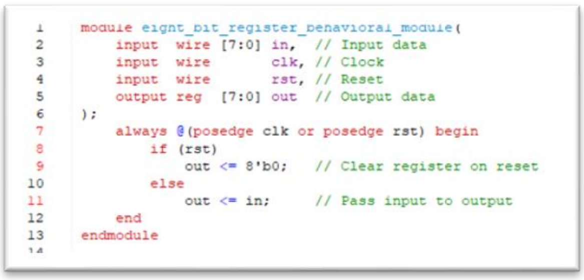
3.1 PE module Theoretical Approach



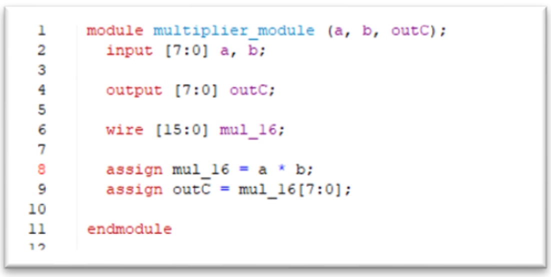
```verilog
module pe_module(
    input  wire [7:0] in1, in2,       // Matrix and filter inputs
    input  wire       clk, rst,       // Clock and reset
    output wire [7:0] pass_right, pass_down, // Propagate inputs to neighbor PEs
    output reg  [7:0] result          // Accumulated output
);

    wire [7:0] mul_out; // Multiplier output
    wire [7:0] add_out; // Adder output

    // Relay: propagate in1 to right PE, in2 to bottom PE
    eight_bit_register_behavioral_module relay_reg1(.in(in1), .clk(clk), .rst(rst), .out(pass_right));
    eight_bit_register_behavioral_module relay_reg2(.in(in2), .clk(clk), .rst(rst), .out(pass_down));

    // Multiplier: combinational
    multiplier_module multiplier (.a(in1), .b(in2), .outC(mul_out));

    // Adder: combinational
    eight_bit_adder_module adder (.a(mul_out), .b(result), .cin(1'b0), .sum(add_out), .cout());

    // Accumulator register: updates on each clock
    always @(posedge clk or posedge rst) begin
        if (rst)
            result <= 8'b0;   // Reset accumulated result
        else
            result <= add_out; // Accumulate
    end

endmodule
```

The PE (Processing Element) module is the fundamental computation block for the systolic array. It accepts two inputs (input and filter), performs multiplication and addition, accumulates the result, and passes data to neighboring PEs (relay function).

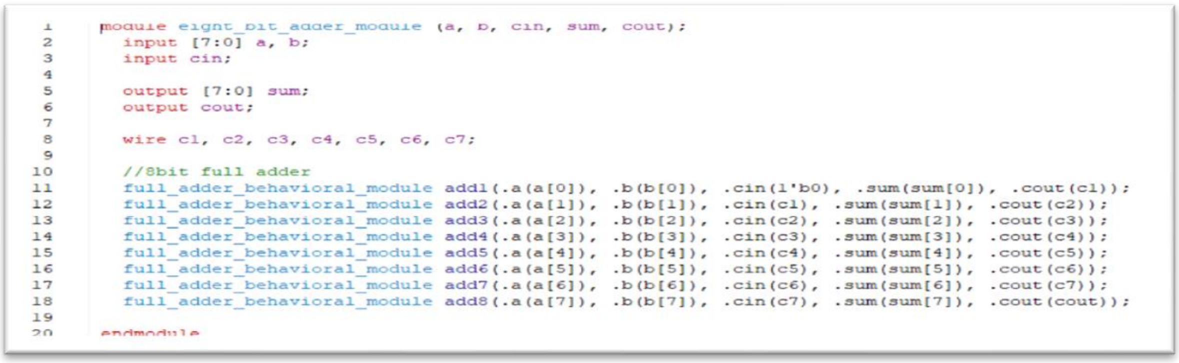
3.2 Detailed Design of PE Module Subcomponents



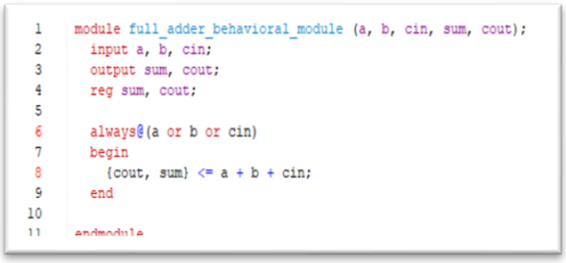
```verilog
module eight_bit_register_behavioral_module(
    input  wire       in,   // Input data
    input  wire       clk,  // Clock
    input  wire       rst,  // Reset
    output reg  [7:0] out   // Output data
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            out <= 8'b0;    // Clear register on reset
        else
            out <= in;      // Pass input to output
    end
endmodule
```



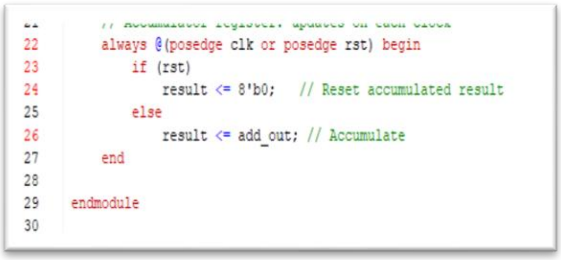
```verilog
module multiplier_module (a, b, outC);
  input [7:0] a, b;

  output [7:0] outC;

  wire [15:0] mul_16;

  assign mul_16 = a * b;
  assign outC = mul_16[7:0];

endmodule
```



```
module eight_bit_adder_module (a, b, cin, sum, cout);
  input [7:0] a, b;
  input cin;

  output [7:0] sum;
  output cout;

  wire c1, c2, c3, c4, c5, c6, c7;

  //8bit full adder
  full_adder_behavioral_module add1(.a(a[0]), .b(b[0]), .cin(1'b0), .sum(sum[0]), .cout(c1));
  full_adder_behavioral_module add2(.a(a[1]), .b(b[1]), .cin(c1), .sum(sum[1]), .cout(c2));
  full_adder_behavioral_module add3(.a(a[2]), .b(b[2]), .cin(c2), .sum(sum[2]), .cout(c3));
  full_adder_behavioral_module add4(.a(a[3]), .b(b[3]), .cin(c3), .sum(sum[3]), .cout(c4));
  full_adder_behavioral_module add5(.a(a[4]), .b(b[4]), .cin(c4), .sum(sum[4]), .cout(c5));
  full_adder_behavioral_module add6(.a(a[5]), .b(b[5]), .cin(c5), .sum(sum[5]), .cout(c6));
  full_adder_behavioral_module add7(.a(a[6]), .b(b[6]), .cin(c6), .sum(sum[6]), .cout(c7));
  full_adder_behavioral_module add8(.a(a[7]), .b(b[7]), .cin(c7), .sum(sum[7]), .cout(cout));

endmodule
```



```
module full_adder_behavioral_module (a, b, cin, sum, cout);
  input a, b, cin;
  output sum, cout;
  reg sum, cout;

  always@(a or b or cin)
  begin
    {cout, sum} <= a + b + cin;
  end

endmodule
```

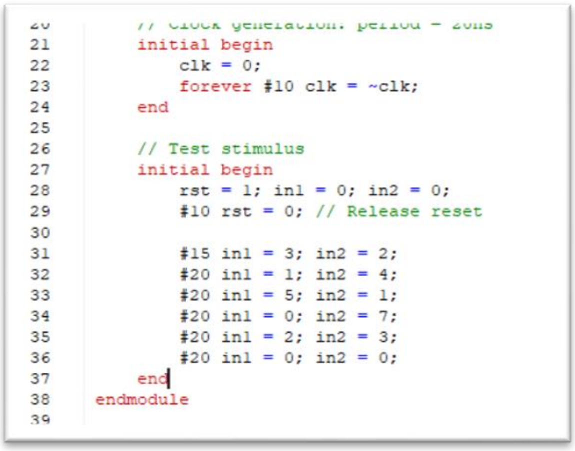


```
    // Accumulator register: updates on each clock
    always @(posedge clk or posedge rst) begin
        if (rst)
            result <= 8'b0;   // Reset accumulated result
        else
            result <= add_out; // Accumulate
    end

endmodule
```

Fig 9. PE Module Subcomponents : (a) Relay Register (b) Multiplier (c) 8bit Adder (d) Full Adder (e) Accumulator

The module comprises four key components. **(a) Register** : The register stores incoming data synchronized to the clock and propagates it rightward and downward using pass_right and pass_down. This enables pipelined dataflow across the systolic array. **(b) Multiplier** : The multiplier computes an 8-bit product of the two inputs. To avoid overflow, the higher bits beyond the 8-bit width are discarded. **(c),(d) Adder** : The adder sums the multiplier output and the accumulator value using a cascade of eight 1-bit full adder modules. **(e) Accumulator** : The accumulator stores the current sum and updates its value on every rising clock edge.

3.3 Testbench & Simulation of PE module



```
    // Clock generation: period = 20ns
    initial begin
        clk = 0;
        forever #10 clk = ~clk;
    end

    // Test stimulus
    initial begin
        rst = 1; in1 = 0; in2 = 0;
        #10 rst = 0; // Release reset

        #15 in1 = 3; in2 = 2;
        #20 in1 = 1; in2 = 4;
        #20 in1 = 5; in2 = 1;
        #20 in1 = 0; in2 = 7;
        #20 in1 = 2; in2 = 3;
        #20 in1 = 0; in2 = 0;
    end
endmodule
```

The testbench periodically provided random inputs to verify multiplication, addition, and relay functions. Simulation results showed that:

Relay outputs (pass_right, pass_down) updated correctly on the next positive clock edge.

Result outputs reflected correct accumulated values in sync with the clock.

This validated that the PE module performs as expected.

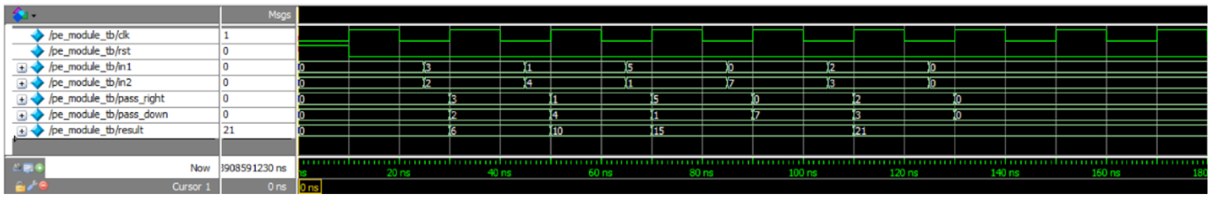
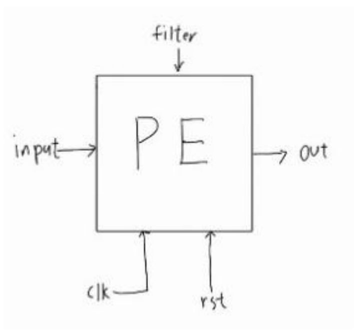


Fig 10. Testbench & Simulation of PE module

4. Single PE Module

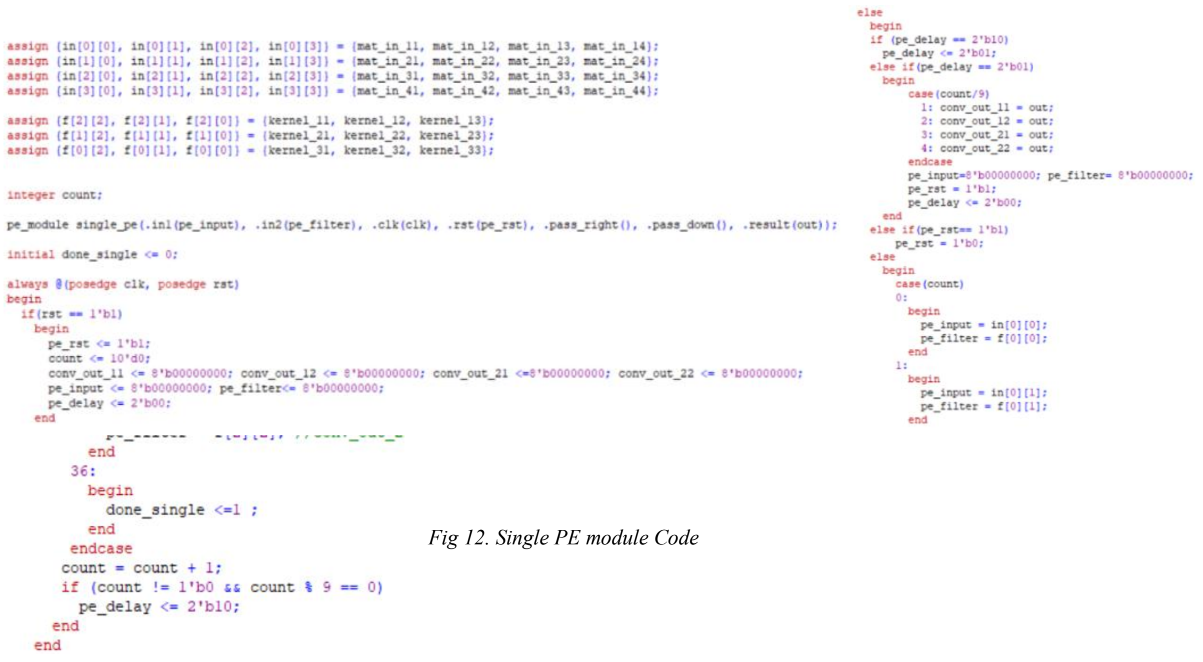
4.1 Single PE theoretical design



In the case of single PE, the previously designed PE module was used for structural modeling. IN and FILTER are input, and the two inputs are multiplied and the product of newly entered inputs is accumulated before initialization. Four output values are produced (c11, c12, c21, c22). In the PE module, **pass-right** and **pass_down** were excluded because they do not need to be output. Those will be used in the systolic array computation later.

Fig 11. Outline of Single PE Computation

4.2 Single PE module



```
assign {in[0][0], in[0][1], in[0][2], in[0][3]} = {mat_in_11, mat_in_12, mat_in_13, mat_in_14};
assign {in[1][0], in[1][1], in[1][2], in[1][3]} = {mat_in_21, mat_in_22, mat_in_23, mat_in_24};
assign {in[2][0], in[2][1], in[2][2], in[2][3]} = {mat_in_31, mat_in_32, mat_in_33, mat_in_34};
assign {in[3][0], in[3][1], in[3][2], in[3][3]} = {mat_in_41, mat_in_42, mat_in_43, mat_in_44};

assign {f[2][2], f[2][1], f[2][0]} = {kernel_11, kernel_12, kernel_13};
assign {f[1][2], f[1][1], f[1][0]} = {kernel_21, kernel_22, kernel_23};
assign {f[0][2], f[0][1], f[0][0]} = {kernel_31, kernel_32, kernel_33};


integer count;

pe_module single_pe(.in1(pe_input), .in2(pe_filter), .clk(clk), .rst(pe_rst), .pass_right(), .pass_down(), .result(out));

initial done_single <= 0;

always @(posedge clk, posedge rst)
begin
  if(rst == 1'b1)
   begin
     pe_rst <= 1'b1;
     count <= 10'd0;
     conv_out_11 <= 8'b00000000; conv_out_12 <= 8'b00000000; conv_out_21 <=8'b00000000; conv_out_22 <= 8'b00000000;
     pe_input <= 8'b00000000; pe_filter<= 8'b00000000;
     pe_delay <= 2'b00;
   end
```

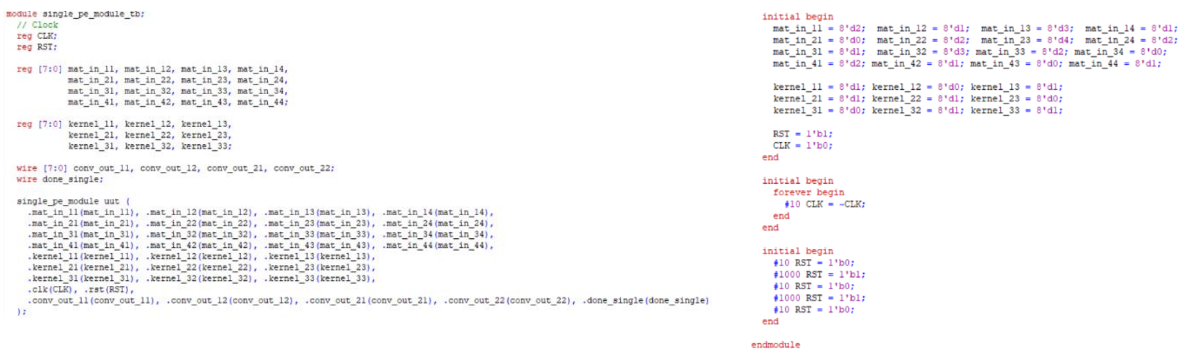
```
        end
      36:
        begin
          done_single <=1 ;
        end
      endcase
    count = count + 1;
    if (count != 1'b0 && count % 9 == 0)
      pe_delay <= 2'b10;
   end
 end
```

```
else
  begin
  if (pe_delay == 2'b10)
    pe_delay <= 2'b01;
  else if(pe_delay == 2'b01)
    begin
        case(count/9)
          1: conv_out_11 = out;
          2: conv_out_12 = out;
          3: conv_out_21 = out;
          4: conv_out_22 = out;
        endcase
        pe_input=8'b00000000; pe_filter= 8'b00000000;
        pe_rst = 1'b1;
        pe_delay <= 2'b00;
    end
  else if(pe_rst== 1'b1)
      pe_rst = 1'b0;
  else
    begin
      case(count)
      0:
        begin
          pe_input = in[0][0];
          pe_filter = f[0][0];
        end
      1:
        begin
          pe_input = in[0][1];
          pe_filter = f[0][1];
        end
```

Fig 12. Single PE module Code

The report includes only the core logic of the code specifically the input/output assignments to the matrix, reset behavior, result storage, and the start/end of the case statement (full details are in the code zip). The input to the module includes the input and filter arrays. Since convolution requires the filter to be rotated 180 degrees, the filter is entered in reverse order (f[2][2] to f[0][0]), ensuring proper alignment. The pe module is instantiated, and done_single is initially set to 0, indicating that the computation hasn't finished. If rst is 1, all values are reset. Otherwise, operations proceed. A 2-bit register **pe_delay** tracks the processing state. When pe_delay is 2, it's simply decremented. When pe_delay is 1, the output is saved, pe_rst is set to 1, and pe_delay is again decremented. In the case where pe_delay is 0, the count increases with each PE operation. Every 9 counts, pe_delay is set to 2 to process the result. This repeats 4 times (for c11, c12, c21, c22), and when count reaches 36, done_single is set to 1 to signal completion.

4.3 Single PE Testbench and simulation

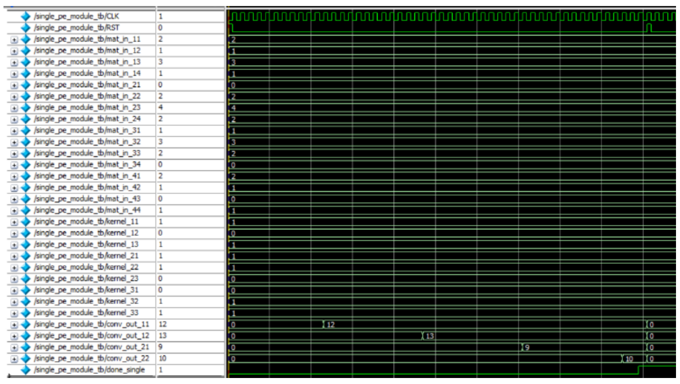


```verilog
module single_pe_module_tb;
  // Clock
  reg CLK;
  reg RST;

  reg [7:0] mat_in_11, mat_in_12, mat_in_13, mat_in_14,
            mat_in_21, mat_in_22, mat_in_23, mat_in_24,
            mat_in_31, mat_in_32, mat_in_33, mat_in_34,
            mat_in_41, mat_in_42, mat_in_43, mat_in_44;

  reg [7:0] kernel_11, kernel_12, kernel_13,
            kernel_21, kernel_22, kernel_23,
            kernel_31, kernel_32, kernel_33;

  wire [7:0] conv_out_11, conv_out_12, conv_out_21, conv_out_22;
  wire done_single;

  single_pe_module uut (
    .mat_in_11(mat_in_11), .mat_in_12(mat_in_12), .mat_in_13(mat_in_13), .mat_in_14(mat_in_14),
    .mat_in_21(mat_in_21), .mat_in_22(mat_in_22), .mat_in_23(mat_in_23), .mat_in_24(mat_in_24),
    .mat_in_31(mat_in_31), .mat_in_32(mat_in_32), .mat_in_33(mat_in_33), .mat_in_34(mat_in_34),
    .mat_in_41(mat_in_41), .mat_in_42(mat_in_42), .mat_in_43(mat_in_43), .mat_in_44(mat_in_44),
    .kernel_11(kernel_11), .kernel_12(kernel_12), .kernel_13(kernel_13),
    .kernel_21(kernel_21), .kernel_22(kernel_22), .kernel_23(kernel_23),
    .kernel_31(kernel_31), .kernel_32(kernel_32), .kernel_33(kernel_33),
    .clk(CLK), .rst(RST),
    .conv_out_11(conv_out_11), .conv_out_12(conv_out_12), .conv_out_21(conv_out_21), .conv_out_22(conv_out_22), .done_single(done_single)
  );
```

```verilog
  initial begin
    mat_in_11 = 8'd2;  mat_in_12 = 8'd1;  mat_in_13 = 8'd3;  mat_in_14 = 8'd1;
    mat_in_21 = 8'd0;  mat_in_22 = 8'd2;  mat_in_23 = 8'd4;  mat_in_24 = 8'd2;
    mat_in_31 = 8'd1;  mat_in_32 = 8'd3; mat_in_33 = 8'd2; mat_in_34 = 8'd0;
    mat_in_41 = 8'd2; mat_in_42 = 8'd1; mat_in_43 = 8'd0; mat_in_44 = 8'd1;

    kernel_11 = 8'd1; kernel_12 = 8'd0; kernel_13 = 8'd1;
    kernel_21 = 8'd1; kernel_22 = 8'd1; kernel_23 = 8'd0;
    kernel_31 = 8'd0; kernel_32 = 8'd1; kernel_33 = 8'd1;

    RST = 1'b1;
    CLK = 1'b0;
  end

  initial begin
    forever begin
      #10 CLK = ~CLK;
    end
  end

  initial begin
    #10 RST = 1'b0;
    #1000 RST = 1'b1;
    #10 RST = 1'b0;
    #1000 RST = 1'b1;
    #10 RST = 1'b0;
  end

endmodule
```

Fig 13. Single PE testbench

I entered random input values and filter values and made a single pe run.



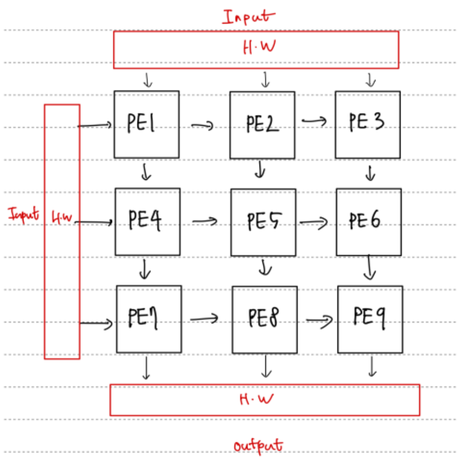
```
/single_pe_module_tb/CLK            1
/single_pe_module_tb/RST            0
/single_pe_module_tb/mat_in_11      2
/single_pe_module_tb/mat_in_12      1
/single_pe_module_tb/mat_in_13      3
/single_pe_module_tb/mat_in_14      1
/single_pe_module_tb/mat_in_21      0
/single_pe_module_tb/mat_in_22      2
/single_pe_module_tb/mat_in_23      4
/single_pe_module_tb/mat_in_24      2
/single_pe_module_tb/mat_in_31      1
/single_pe_module_tb/mat_in_32      3
/single_pe_module_tb/mat_in_33      2
/single_pe_module_tb/mat_in_34      0
/single_pe_module_tb/mat_in_41      2
/single_pe_module_tb/mat_in_42      1
/single_pe_module_tb/mat_in_43      0
/single_pe_module_tb/mat_in_44      1
/single_pe_module_tb/kernel_11      1
/single_pe_module_tb/kernel_12      0
/single_pe_module_tb/kernel_13      1
/single_pe_module_tb/kernel_21      1
/single_pe_module_tb/kernel_22      1
/single_pe_module_tb/kernel_23      0
/single_pe_module_tb/kernel_31      0
/single_pe_module_tb/kernel_32      1
/single_pe_module_tb/kernel_33      1
/single_pe_module_tb/conv_out_11    12
/single_pe_module_tb/conv_out_12    13
/single_pe_module_tb/conv_out_21    9
/single_pe_module_tb/conv_out_22    10
/single_pe_module_tb/done_single    1
```

Looking at the simulation results, we can see that c11~c22 values are output in order **every 12 clocks**(c11=12, c12=13, c21=9, c22=10). And when c22 is output, the value of done_single also becomes 1. Finally, when rst=1, all values are initialized to 0. This shows that the code is working normally.

Fig 14. Single PE simulation results

5. 3x3 Systolic Array Module

5.1 3x3 Systolic Array Theoretical Design



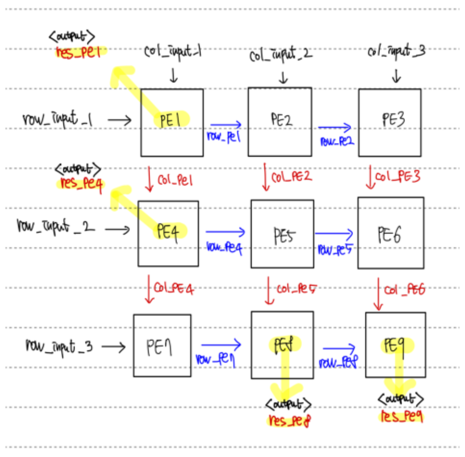
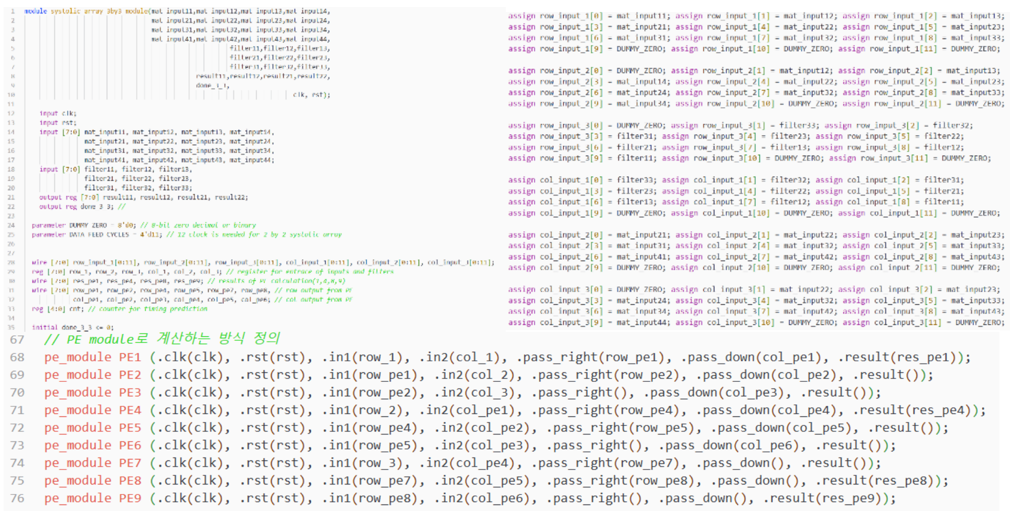


Fig 15. Structural diagram of 3x3 systolic array

The 3×3 Systolic Array consists of nine PE (Processing Element) modules, each of which performs partial computations based on the input values. The inputs are **a 4×4 matrix and a 3×3 filter**, and a convolution operation is performed to produce a final **2×2 output matrix**.

As shown in Fig. 15, input values are fed into the array from both the row and column directions. Each PE receives inputs through wires such as row_pe1 and col_pe1, performs multiplication and accumulation operations, and passes the input values unchanged to the next PE. The accumulated result within each PE is ultimately output through the resultXY output reg.

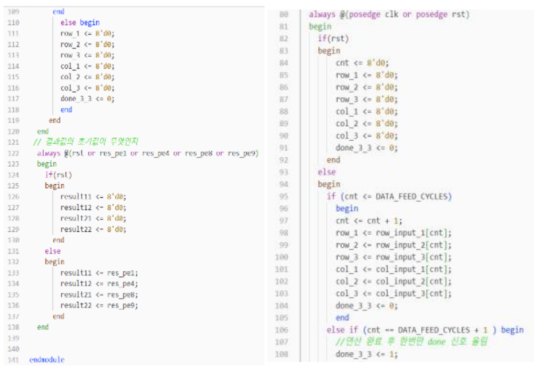
5.2 3x3 Systolic Array Code Design & Simulation



```
module systolic_array_3by3_module(mat_input11,mat_input12,mat_input13,mat_input14,
                                  mat_input21,mat_input22,mat_input23,mat_input24,
                                  mat_input31,mat_input32,mat_input33,mat_input34,
                                  mat_input41,mat_input42,mat_input43,mat_input44,
                                                filter11,filter12,filter13,
                                                filter21,filter22,filter23,
                                                filter31,filter32,filter33,
                                          result11,result12,result21,result22,
                                          done_3_3,
                                                                    clk, rst);

    input clk;
    input rst;
    input [7:0] mat_input11, mat_input12, mat_input13, mat_input14,
                mat_input21, mat_input22, mat_input23, mat_input24,
                mat_input31, mat_input32, mat_input33, mat_input34,
                mat_input41, mat_input42, mat_input43, mat_input44;
    input [7:0] filter11, filter12, filter13,
                filter21, filter22, filter23,
                filter31, filter32, filter33;
    output reg [7:0] result11, result12, result21, result22;
    output reg done_3_3; //

  parameter DUMMY_ZERO = 8'd0; // 8-bit zero decimal or binary
  parameter DATA_FEED_CYCLES = 4'd11; // 12 clock is needed for 2 by 2 systolic array


  wire [7:0] row_input_1[0:11], row_input_2[0:11], row_input_3[0:11], col_input_1[0:11], col_input_2[0:11], col_input_3[0:11];
  reg [7:0] row_1, row_2, row_3, col_1, col_2, col_3; // register for entrace of inputs and filters
  wire [7:0] res_pe1, res_pe4, res_pe8, res_pe9; // results of PE calculation(1,4,8,9)
  wire [7:0] row_pe1, row_pe2, row_pe4, row_pe5, row_pe7, row_pe8, // row output from PE
             col_pe1, col_pe2, col_pe3, col_pe4, col_pe5, col_pe6; // col output from PE
  reg [4:0] cnt; // counter for timing prediction

  initial done_3_3 <= 0;
```

```
assign row_input_1[0] = mat_input11; assign row_input_1[1] = mat_input12; assign row_input_1[2] = mat_input13;
assign row_input_1[3] = mat_input21; assign row_input_1[4] = mat_input22; assign row_input_1[5] = mat_input23;
assign row_input_1[6] = mat_input31; assign row_input_1[7] = mat_input32; assign row_input_1[8] = mat_input33;
assign row_input_1[9] = DUMMY_ZERO; assign row_input_1[10] = DUMMY_ZERO; assign row_input_1[11] = DUMMY_ZERO;

assign row_input_2[0] = DUMMY_ZERO; assign row_input_2[1] = mat_input12; assign row_input_2[2] = mat_input13;
assign row_input_2[3] = mat_input14; assign row_input_2[4] = mat_input22; assign row_input_2[5] = mat_input23;
assign row_input_2[6] = mat_input24; assign row_input_2[7] = mat_input32; assign row_input_2[8] = mat_input33;
assign row_input_2[9] = mat_input34; assign row_input_2[10] = DUMMY_ZERO; assign row_input_2[11] = DUMMY_ZERO;

assign row_input_3[0] = DUMMY_ZERO; assign row_input_3[1] = filter33; assign row_input_3[2] = filter32;
assign row_input_3[3] = filter31; assign row_input_3[4] = filter23; assign row_input_3[5] = filter22;
assign row_input_3[6] = filter21; assign row_input_3[7] = filter13; assign row_input_3[8] = filter12;
assign row_input_3[9] = filter11; assign row_input_3[10] = DUMMY_ZERO; assign row_input_3[11] = DUMMY_ZERO;

assign col_input_1[0] = filter33; assign col_input_1[1] = filter32; assign col_input_1[2] = filter31;
assign col_input_1[3] = filter23; assign col_input_1[4] = filter22; assign col_input_1[5] = filter21;
assign col_input_1[6] = filter13; assign col_input_1[7] = filter12; assign col_input_1[8] = filter11;
assign col_input_1[9] = DUMMY_ZERO; assign col_input_1[10] = DUMMY_ZERO; assign col_input_1[11] = DUMMY_ZERO;

assign col_input_2[0] = mat_input21; assign col_input_2[1] = mat_input22; assign col_input_2[2] = mat_input23;
assign col_input_2[3] = mat_input31; assign col_input_2[4] = mat_input32; assign col_input_2[5] = mat_input33;
assign col_input_2[6] = mat_input41; assign col_input_2[7] = mat_input42; assign col_input_2[8] = mat_input43;
assign col_input_2[9] = DUMMY_ZERO; assign col_input_2[10] = DUMMY_ZERO; assign col_input_2[11] = DUMMY_ZERO;

assign col_input_3[0] = DUMMY_ZERO; assign col_input_3[1] = mat_input22; assign col_input_3[2] = mat_input23;
assign col_input_3[3] = mat_input24; assign col_input_3[4] = mat_input32; assign col_input_3[5] = mat_input33;
assign col_input_3[6] = mat_input34; assign col_input_3[7] = mat_input42; assign col_input_3[8] = mat_input43;
assign col_input_3[9] = mat_input44; assign col_input_3[10] = DUMMY_ZERO; assign col_input_3[11] = DUMMY_ZERO;
```

```
 // PE module로 계산하는 방식 정의
 pe_module PE1 (.clk(clk), .rst(rst), .in1(row_1), .in2(col_1), .pass_right(row_pe1), .pass_down(col_pe1), .result(res_pe1));
 pe_module PE2 (.clk(clk), .rst(rst), .in1(row_pe1), .in2(col_2), .pass_right(row_pe2), .pass_down(col_pe2), .result());
 pe_module PE3 (.clk(clk), .rst(rst), .in1(row_pe2), .in2(col_3), .pass_right(), .pass_down(col_pe3), .result());
 pe_module PE4 (.clk(clk), .rst(rst), .in1(row_2), .in2(col_pe1), .pass_right(row_pe4), .pass_down(col_pe4), .result(res_pe4));
 pe_module PE5 (.clk(clk), .rst(rst), .in1(row_pe4), .in2(col_pe2), .pass_right(row_pe5), .pass_down(col_pe5), .result());
 pe_module PE6 (.clk(clk), .rst(rst), .in1(row_pe5), .in2(col_pe3), .pass_right(), .pass_down(col_pe6), .result());
 pe_module PE7 (.clk(clk), .rst(rst), .in1(row_3), .in2(col_pe4), .pass_right(row_pe7), .pass_down(), .result());
 pe_module PE8 (.clk(clk), .rst(rst), .in1(row_pe7), .in2(col_pe5), .pass_right(row_pe8), .pass_down(), .result(res_pe8));
 pe_module PE9 (.clk(clk), .rst(rst), .in1(row_pe8), .in2(col_pe6), .pass_right(), .pass_down(), .result(res_pe9));
```



```verilog
always @(posedge clk or posedge rst)
begin
  if(rst)
  begin
    cnt <= 8'd0;
    row_1 <= 8'd0;
    row_2 <= 8'd0;
    row_3 <= 8'd0;
    col_1 <= 8'd0;
    col_2 <= 8'd0;
    col_3 <= 8'd0;
    done_3_3 <= 0;
   end
  else
  begin
   if (cnt <= DATA_FEED_CYCLES)
    begin
    cnt <= cnt + 1;
    row_1 <= row_input_1[cnt];
    row_2 <= row_input_2[cnt];
    row_3 <= row_input_3[cnt];
    col_1 <= col_input_1[cnt];
    col_2 <= col_input_2[cnt];
    col_3 <= col_input_3[cnt];
    done_3_3 <= 0;
    end
   else if (cnt == DATA_FEED_CYCLES + 1 ) begin
    //연산 완료 후 한번만 done 신호 올림
    done_3_3 <= 1;
      end
        else begin
        row_1 <= 8'd0;
        row_2 <= 8'd0;
        row_3 <= 8'd0;
        col_1 <= 8'd0;
        col_2 <= 8'd0;
        col_3 <= 8'd0;
        done_3_3 <= 0;
        end
     end
   end
  // 결과값의 초기값이 무엇인지
   always @(rst or res_pe1 or res_pe4 or res_pe8 or res_pe9)
   begin
     if(rst)
     begin
        result11 <= 8'd0;
        result12 <= 8'd0;
        result21 <= 8'd0;
        result22 <= 8'd0;
      end
     else
     begin
        result11 <= res_pe1;
        result12 <= res_pe4;
        result21 <= res_pe8;
        result22 <= res_pe9;
      end
   end


endmodule
```

Fig 16. 3x3 Systolic Array Code (A) Port declaration and Input data mapping (B) Instantiation of PE module (C) Result Assignment

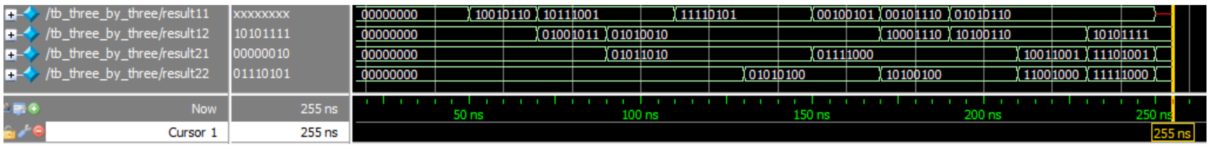
Table 1. Summary of major Variable

Variable Name	Type	Width	Description
mat_inputXY	input	[7:0]	Each element of the 4x4 input matrix
filterXY	input	[7:0]	Each element of the 3x3 input matrix
row_[1-3]	reg	[7:0]	Current input values for each row
col_[1-3]	reg	[7:0]	Current input values for each column
res_peX	wire	[7:0]	Output accumulated result each PE module
resultXY	output reg	[7:0]	Final convolution result (2x2)
done_3_3	output reg	1 bit	Signal indicating computation complete (1 = done)

Table 1 provides a brief explanation of key variables used in the code along with their roles. The key variables and their roles are summarized in the table above. In Fig. 16, the input and output ports are declared, including the done_3_3 signal that indicates the completion of computation.

The PE modules are arranged in a 3×3 grid following a row-major order and are labeled PE1 through PE9. Among them, **PE1, PE4, PE8, and PE9 generate accumulated results**, which are mapped to **result11, result12, result21, and result22,** respectively. The intermediate values passed between PEs are named row_pe and col_pe, with each wire clearly indicating the originating PE for traceability. Every PE receives new input data on each clock cycle, performs multiplication and accumulation, and updates its internal accumulated value. The final result is output through res_pe once the data flow through the systolic array is complete.

Additionally, the module is designed to reset all internal registers and input buffers to zero when a reset signal is received during operation. To feed data into the PEs, a clock counter cnt is used, pushing input data for a total of **12 cycles** (DATA_FEED_CYCLES = 11). The done_3_3 signal is asserted on the 13th cycle (cnt == 12) to indicate that computation is complete. Whenever a clock pulse occurs, the output registers are updated to reflect the latest values from the PEs.


/tb_three_by_three/result11 xxxxxxxx 00000000 10010110 10111001 11110101 00100101 00101110 01010110
/tb_three_by_three/result12 10101111 00000000 01001011 01010010 10001110 10100110 10101111
/tb_three_by_three/result21 00000010 00000000 01011010 01111000 10011001 11101001
/tb_three_by_three/result22 01110101 00000000 01010100 10100100 11001000 11111000
Now 255 ns
Cursor 1 255 ns
50 ns
100 ns
150 ns
200 ns
250 ns
255 ns
Fig 17. Effort to minimize clock cycles

In the case of Fig. 17, PE8 and PE9 failed to complete their accumulation operations, resulting in incomplete outputs. In particular, within the 3×3 systolic array structure, input data propagates sequentially, and it became clear through this test that an **additional two zeros** at the end of the input sequence are necessary for the data to reach the third-row PEs. This experiment highlighted the timing requirement for the input to travel through the array and reach the lower-row PEs. Based on this insight, the value of DATA_FEED_CYCLES was set to 11, allowing all computations to complete in parallel and ensuring the correctness of the final outputs. All variables were named to clearly reflect their roles, with an emphasis on code readability and maintainability.

Table 2. (a) Input Feeding Schedule by Clock Cycle (b) Accumulation Results and Computation Timing Summary

Clock	row_input_1	row_input_2	row_input_3	col_input_1	col_input_2	col_input_3
1	mat_input11	0	0	filter33	mat_input21	0
2	mat_input12	mat_input12	filter33	filter32	mat_input22	mat_input22
3	mat_input13	mat_input13	filter32	filter31	mat_input23	mat_input23
4	mat_input21	mat_input14	filter31	filter23	mat_input31	mat_input24
5	mat_input22	mat_input22	filter23	filter22	mat_input32	mat_input32
6	mat_input23	mat_input23	filter22	filter21	mat_input33	mat_input33
7	mat_input31	mat_input24	filter21	filter13	mat_input41	mat_input34
8	mat_input32	mat_input32	filter13	filter12	mat_input42	mat_input42
9	mat_input33	mat_input33	filter12	filter11	mat_input43	mat_input43
10	0	mat_input34	filter11	0	0	mat_input44
11	0	0	0	0	0	0
12	0	0	0	0	0	0

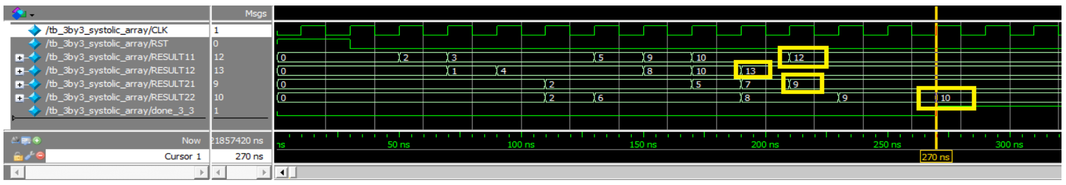
PE No.	Output Signal	Computed at Clock	Computation Formula (Accumulation Sequence)
PE1	result11	Clock 9	mat_input11 × filter33 + mat_input12 × filter32 + mat_input13 × filter31 + mat_input21 × filter23 + mat_input22 × filter22 + mat_input23 × filter21 + mat_input31 × filter13 + mat_input32 × filter12 + mat_input33 × filter11
PE4	result12	Clock 10	0 + mat_input12 × filter33 + mat_input13 × filter32 + mat_input14 × filter31 + mat_input22 × filter23 + mat_input23 × filter22 + mat_input24 × filter21 + mat_input32 × filter13 + mat_input33 × filter12 + mat_input34 × filter11
PE8	result21	Clock 11	0 + 0 + filter33 × mat_input21 + filter32 × mat_input22 + filter31 × mat_input23 + filter23 × mat_input31 + filter22 × mat_input32 + filter21 × mat_input33 + filter13 × mat_input41 + filter12 × mat_input42 + filter11 × mat_input43
PE9	result22	Clock 12	0 + 0 + filter33 × mat_input22 + filter32 × mat_input23 + filter31 × mat_input24 + filter23 × mat_input32 + filter22 × mat_input33 + filter21 × mat_input34 + filter13 × mat_input42 + filter12 × mat_input43 + filter11 × mat_input44

The table above summarizes the accumulation results and computation timing. It indicates the clock cycle at which each PE module receives its inputs, completes the computation, and assigns the result.

5.3 3x3 Systolic Array Testbench

4x4 Input Matrix				3x3 filter Matrix			result	
2	1	3	1	1	0	1	result11 = 12	result12 = 13
0	2	4	2	1	1	0	result21 = 9	result22 =10
1	3	2	0	0	1	1		
2	1	0	1					

Fig 18. Input Matrices and result of 3x3 systolic array



	Msgs
/tb_3by3_systolic_array/CLK	1
/tb_3by3_systolic_array/RST	0
/tb_3by3_systolic_array/RESULT11	12
/tb_3by3_systolic_array/RESULT12	13
/tb_3by3_systolic_array/RESULT21	9
/tb_3by3_systolic_array/RESULT22	10
/tb_3by3_systolic_array/done_3_3	1
Now	21857420 ns
Cursor 1	270 ns

When the input values are given as a 4×4 matrix and a 3×3 filter, as shown in the table above, the output results are observed as follows: result11 = 12, result12 = 13, result21 = 9, and result22 = 10. Rst is deasserted at 30 ns, which marks the start of the computation. After 12 clock cycles, the final result is produced, and the done_2_2 signal transitions to 1, indicating that the computation has completed. The total computation time is 240 ns. Since these output values match the expected results from a 2D convolution between the input matrix and filter, we can conclude that the model has been implemented correctly.

6. 2x2 Systolic Array Module

6.1 2x2 Systolic Array Theoretical Design

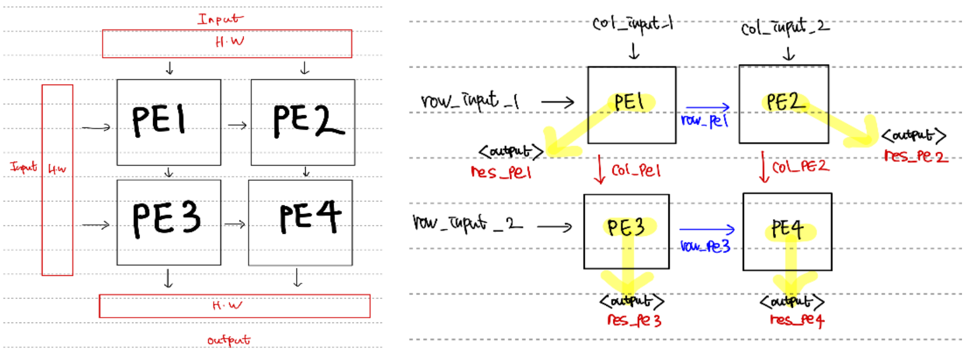
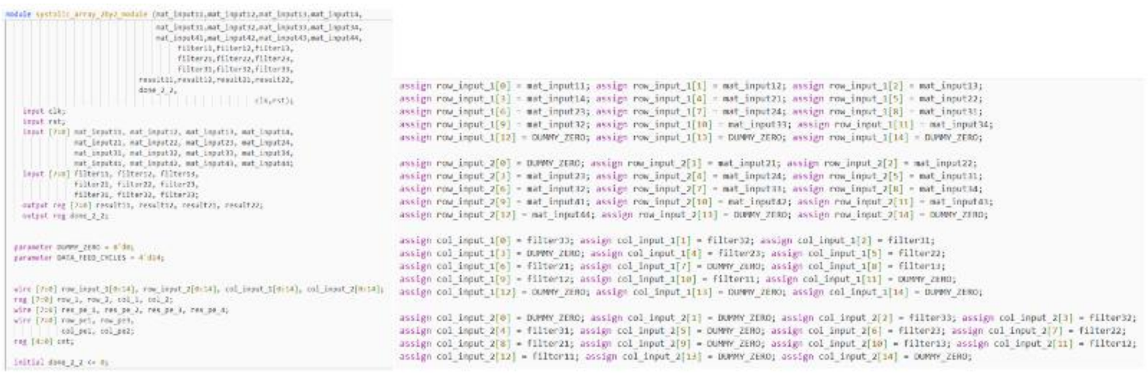


Fig 19. Structural diagram of 2x2 systolic array

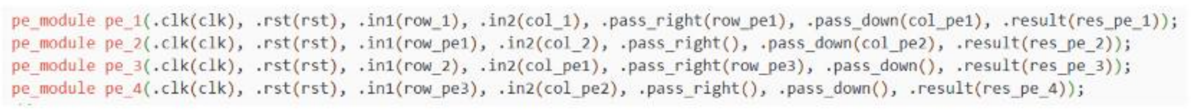
In the case of the 2×2 systolic array, the primary difference from the 3×3 structure lies in the use of **only four PE modules**. While the computation method is fundamentally the same as that of the 3×3 systolic array, the key distinction is that in the 3×3 array, only a subset of PEs produces output values, whereas in the 2×2 design, all PE modules generate res_pe values by carefully adjusting input timing. This input timing control is achieved using the DUMMY_ZERO variable, which allows each PE to compute an independent convolution result. As a result, the design maximizes both **parallelism** and **hardware utilization** across the array.

6.2 2x2 Systolic Array Code design & Simulation

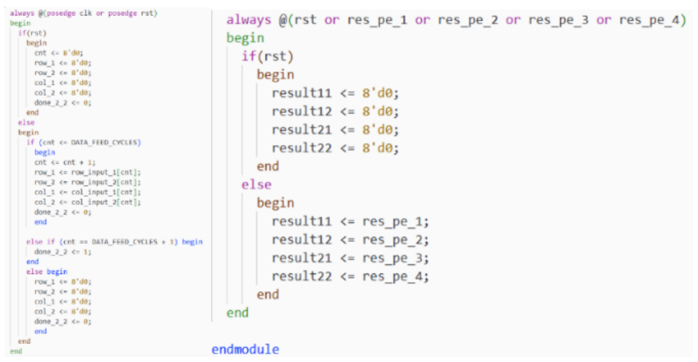


```
module systolic_array_2by2_module (mat_input11,mat_input12,mat_input13,mat_input14,
                                   mat_input31,mat_input32,mat_input33,mat_input34,
                                   mat_input41,mat_input42,mat_input43,mat_input44,
                                        filter11,filter12,filter13,
                                        filter21,filter22,filter23,
                                        filter31,filter32,filter33,
                                   result11,result12,result21,result22,
                                   done_2_2,
                                                          clk,rst);
  input clk;
  input rst;
  input [7:0] mat_input11, mat_input12, mat_input13, mat_input14,
              mat_input21, mat_input22, mat_input23, mat_input24,
              mat_input31, mat_input32, mat_input33, mat_input34,
              mat_input41, mat_input42, mat_input43, mat_input44;
  input [7:0] filter11, filter12, filter13,
              filter21, filter22, filter23,
              filter31, filter32, filter33;
  output reg [7:0] result11, result12, result21, result22;
  output reg done_2_2;


parameter DUMMY_ZERO = 8'd0;
parameter DATA_FEED_CYCLES = 4'd14;


wire [7:0] row_input_1[0:14], row_input_2[0:14], col_input_1[0:14], col_input_2[0:14];
reg [7:0] row_1, row_2, col_1, col_2;
wire [7:0] res_pe_1, res_pe_2, res_pe_3, res_pe_4;
wire [7:0] row_pe1, row_pe3,
           col_pe1, col_pe2;
reg [4:0] cnt;

initial done_2_2 <= 0;
```

```
assign row_input_1[0] = mat_input11; assign row_input_1[1] = mat_input12; assign row_input_1[2] = mat_input13;
assign row_input_1[3] = mat_input14; assign row_input_1[4] = mat_input21; assign row_input_1[5] = mat_input22;
assign row_input_1[6] = mat_input23; assign row_input_1[7] = mat_input24; assign row_input_1[8] = mat_input31;
assign row_input_1[9] = mat_input32; assign row_input_1[10] = mat_input33; assign row_input_1[11] = mat_input34;
assign row_input_1[12] = DUMMY_ZERO; assign row_input_1[13] = DUMMY_ZERO; assign row_input_1[14] = DUMMY_ZERO;

assign row_input_2[0] = DUMMY_ZERO; assign row_input_2[1] = mat_input21; assign row_input_2[2] = mat_input22;
assign row_input_2[3] = mat_input23; assign row_input_2[4] = mat_input24; assign row_input_2[5] = mat_input31;
assign row_input_2[6] = mat_input32; assign row_input_2[7] = mat_input33; assign row_input_2[8] = mat_input34;
assign row_input_2[9] = mat_input41; assign row_input_2[10] = mat_input42; assign row_input_2[11] = mat_input43;
assign row_input_2[12] = mat_input44; assign row_input_2[13] = DUMMY_ZERO; assign row_input_2[14] = DUMMY_ZERO;

assign col_input_1[0] = filter33; assign col_input_1[1] = filter32; assign col_input_1[2] = filter31;
assign col_input_1[3] = DUMMY_ZERO; assign col_input_1[4] = filter23; assign col_input_1[5] = filter22;
assign col_input_1[6] = filter21; assign col_input_1[7] = DUMMY_ZERO; assign col_input_1[8] = filter13;
assign col_input_1[9] = filter12; assign col_input_1[10] = filter11; assign col_input_1[11] = DUMMY_ZERO;
assign col_input_1[12] = DUMMY_ZERO; assign col_input_1[13] = DUMMY_ZERO; assign col_input_1[14] = DUMMY_ZERO;

assign col_input_2[0] = DUMMY_ZERO; assign col_input_2[1] = DUMMY_ZERO; assign col_input_2[2] = filter33; assign col_input_2[3] = filter32;
assign col_input_2[4] = filter31; assign col_input_2[5] = DUMMY_ZERO; assign col_input_2[6] = filter23; assign col_input_2[7] = filter22;
assign col_input_2[8] = filter21; assign col_input_2[9] = DUMMY_ZERO; assign col_input_2[10] = filter13; assign col_input_2[11] = filter12;
assign col_input_2[12] = filter11; assign col_input_2[13] = DUMMY_ZERO; assign col_input_2[14] = DUMMY_ZERO;
```



```
pe_module pe_1(.clk(clk), .rst(rst), .in1(row_1), .in2(col_1), .pass_right(row_pe1), .pass_down(col_pe1), .result(res_pe_1));
pe_module pe_2(.clk(clk), .rst(rst), .in1(row_pe1), .in2(col_2), .pass_right(), .pass_down(col_pe2), .result(res_pe_2));
pe_module pe_3(.clk(clk), .rst(rst), .in1(row_2), .in2(col_pe1), .pass_right(row_pe3), .pass_down(), .result(res_pe_3));
pe_module pe_4(.clk(clk), .rst(rst), .in1(row_pe3), .in2(col_pe2), .pass_right(), .pass_down(), .result(res_pe_4));
```

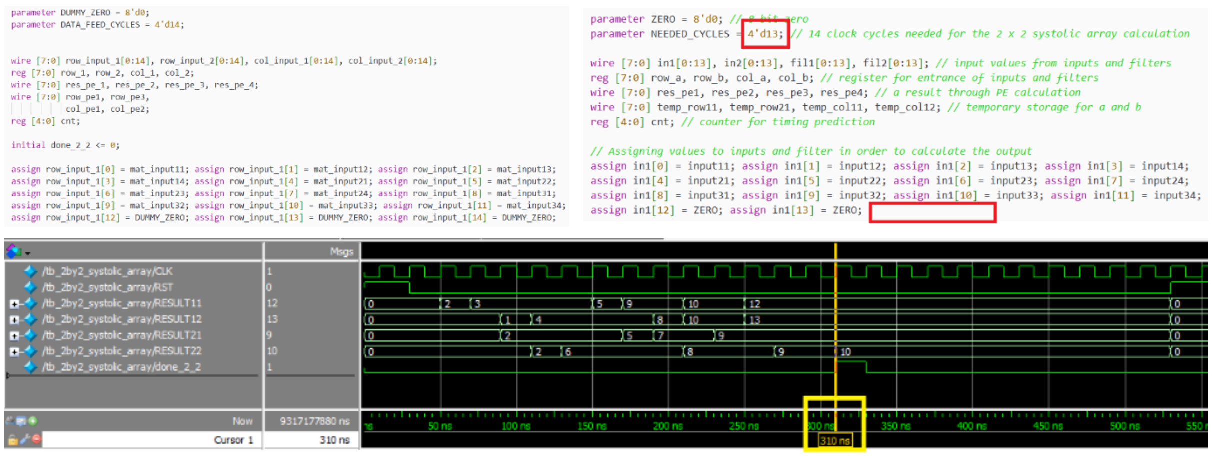


```
always @(posedge clk or posedge rst)
begin
  if(rst)
    begin
      cnt <= 8'd0;
      row_1 <= 8'd0;
      row_2 <= 8'd0;
      col_1 <= 8'd0;
      col_2 <= 8'd0;
      done_2_2 <= 0;
    end
  else
  begin
    if (cnt <= DATA_FEED_CYCLES)
      begin
      cnt <= cnt + 1;
      row_1 <= row_input_1[cnt];
      row_2 <= row_input_2[cnt];
      col_1 <= col_input_1[cnt];
      col_2 <= col_input_2[cnt];
      done_2_2 <= 0;
      end

    else if (cnt == DATA_FEED_CYCLES + 1) begin
      done_2_2 <= 1;
    end
    else begin
      row_1 <= 8'd0;
      row_2 <= 8'd0;
      col_1 <= 8'd0;
      col_2 <= 8'd0;
      done_2_2 <= 0;
      end
  end
end
```

```
always @(rst or res_pe_1 or res_pe_2 or res_pe_3 or res_pe_4)
begin
  if(rst)
    begin
      result11 <= 8'd0;
      result12 <= 8'd0;
      result21 <= 8'd0;
      result22 <= 8'd0;
    end
  else
    begin
      result11 <= res_pe_1;
      result12 <= res_pe_2;
      result21 <= res_pe_3;
      result22 <= res_pe_4;
    end
end

endmodule
```

Fig 20. 2x2 systolic Array Code

The systolic_array_2by2_module implements a 2×2 systolic array using four PE modules to perform a convolution between a 4×4 input matrix and a 3×3 filter. The module is designed to feed input values over **15 clock cycles**, with each of the row_input and col_input arrays containing 15 values. The DUMMY_ZERO parameter is used to control **input timing**, ensuring that each PE receives its data at the correct cycle to begin computation and generate a result. This systolic array uses a dataflow structure where the input matrix and filter values flow diagonally through the array. **Row_input_1 and row_input_2 are inputs injected from the top** in the row direction, while **col_input_1 and col_input_2 are filter values injected from the left** in the column direction. Zeros are inserted into the inputs to ensure that each PE performs multiplication and accumulation at the correct timing.



```
parameter DUMMY_ZERO = 8'd0;
parameter DATA_FEED_CYCLES = 4'd14;


wire [7:0] row_input_1[0:14], row_input_2[0:14], col_input_1[0:14], col_input_2[0:14];
reg [7:0] row_1, row_2, col_1, col_2;
wire [7:0] res_pe_1, res_pe_2, res_pe_3, res_pe_4;
wire [7:0] row_pe1, row_pe3,
           col_pe1, col_pe2;
reg [4:0] cnt;

initial done_2_2 <= 0;

assign row_input_1[0] = mat_input11; assign row_input_1[1] = mat_input12; assign row_input_1[2] = mat_input13;
assign row_input_1[3] = mat_input14; assign row_input_1[4] = mat_input21; assign row_input_1[5] = mat_input22;
assign row_input_1[6] = mat_input23; assign row_input_1[7] = mat_input24; assign row_input_1[8] = mat_input31;
assign row_input_1[9] = mat_input32; assign row_input_1[10] = mat_input33; assign row_input_1[11] = mat_input34;
assign row_input_1[12] = DUMMY_ZERO; assign row_input_1[13] = DUMMY_ZERO; assign row_input_1[14] = DUMMY_ZERO;
```

```
parameter ZERO = 8'd0; // 8 bit zero
parameter NEEDED_CYCLES = 4'd13; // 14 clock cycles needed for the 2 x 2 systolic array calculation

wire [7:0] in1[0:13], in2[0:13], fil1[0:13], fil2[0:13]; // input values from inputs and filters
reg [7:0] row_a, row_b, col_a, col_b; // register for entrance of inputs and filters
wire [7:0] res_pe1, res_pe2, res_pe3, res_pe4; // a result through PE calculation
wire [7:0] temp_row11, temp_row21, temp_col11, temp_col12; // temporary storage for a and b
reg [4:0] cnt; // counter for timing prediction

// Assigning values to inputs and filter in order to calculate the output
assign in1[0] = input11; assign in1[1] = input12; assign in1[2] = input13; assign in1[3] = input14;
assign in1[4] = input21; assign in1[5] = input22; assign in1[6] = input23; assign in1[7] = input24;
assign in1[8] = input31; assign in1[9] = input32; assign in1[10] = input33; assign in1[11] = input34;
assign in1[12] = ZERO; assign in1[13] = ZERO;
```

Msgs
/tb_2by2_systolic_array/CLK 1
/tb_2by2_systolic_array/RST 0
/tb_2by2_systolic_array/RESULT11 12
/tb_2by2_systolic_array/RESULT12 13
/tb_2by2_systolic_array/RESULT21 9
/tb_2by2_systolic_array/RESULT22 10
/tb_2by2_systolic_array/done_2_2 1
Now 9317177880 ns
Cursor 1 310 ns
310 ns

Fig 21. Effort to minimize clock cycles

In the case of Fig 21, an attempt was made to minimize the number of clock cycles by optimizing the input arrangement and reducing the amount of zero padding. Although one of the final DUMMY_ZEROs was removed during testing, it was observed that the total number of **required clock cycles remained unchanged**. Therefore, for consistency with the 3×3 structure and ease of code maintenance, two zero paddings were retained at the end of the input sequence.

Each PE module computes and accumulates its result internally and outputs it via its res_pe port. The final results are assigned to the outputs result11, result12, result21, and result22. Unlike in a 3×3 systolic array where only some PEs produce output, this 2×2 design is structured to produce results in all four PEs, maximizing both **parallelism and hardware utilization.**

Table 3. Input Feeding Schedule by Clock Cycle and Accumulation Results and Computation Timing Summary

Cycle	row_input_1	col_input_1	row_input_2	col_input_2
1	mat_input11	filter33	0	0
2	mat_input12	filter32	mat_input21	0
3	mat_input13	filter31	mat_input22	filter33
4	mat_input14	0	mat_input23	filter32
5	mat_input21	filter23	mat_input24	filter31
6	mat_input22	filter22	mat_input31	0
7	mat_input23	filter21	mat_input32	filter23
8	mat_input24	0	mat_input33	filter22
9	mat_input31	filter13	mat_input34	filter21
10	mat_input32	filter12	mat_input41	0
11	mat_input33	filter11	mat_input42	filter13
12	mat_input34	0	mat_input43	filter12
13	0	0	mat_input44	filter11
14	0	0	0	0
15	0	0	0	0

PE No.	Output Signal	Computed at Clock	Computation Formula (Accumulation Sequence)
PE1	result11	Clock 11	mat_input11 × filter33 + mat_input12 × filter32 + mat_input13 × filter31 + mat_input21 × filter23 + mat_input22 × filter22 + mat_input23 × filter21 + mat_input31 × filter13 + mat_input32 × filter12 + mat_input33 × filter11
PE2	result12	Clock 1	0 + mat_input12 × filter33 + mat_input13 × filter32 + mat_input14 × filter31 + mat_input22 × filter23 + mat_input23 × filter22 + mat_input24 × filter21 + mat_input32 × filter13 + mat_input33 × filter12 + mat_input34 × filter11
PE3	result21	Clock 10	0 + 0 + mat_input21 × filter33 + mat_input22 × filter32 + mat_input23 × filter31 + mat_input31 × filter23 + mat_input32 × filter22 + mat_input33 × filter21 + mat_input41 × filter13 + mat_input42 × filter12 + mat_input43 × filter11
PE4	result22	Clock 14	0 + 0 + mat_input22 × filter33 + mat_input23 × filter32 + mat_input24 × filter31 + mat_input32 × filter23 + mat_input33 × filter22 + mat_input34 × filter21 + mat_input42 × filter13 + mat_input43 × filter12 + mat_input44 × filter11

6.3 2x2 Systolic Array Testbench and Simulation

Table 4. Input Matrices and Convolution Results

4x4 Input Matrix				3x3 filter Matrix			result	
2	1	3	1	1	0	1	result11 = 12	result12 = 13
0	2	4	2	1	1	0	result21 = 9	result22 =10
1	3	2	0	0	1	1		
2	1	0	1					

```
begin
      #10 RST = 1'b1;
      #20 RST = 1'b0;
          INPUT11 = 8'd2;  INPUT12 = 8'd1;  INPUT13 = 8'd3;  INPUT14 = 8'd1;
          INPUT21 = 8'd0;  INPUT22 = 8'd2;  INPUT23 = 8'd4;  INPUT24 = 8'd2;
          INPUT31 = 8'd1;  INPUT32 = 8'd3;  INPUT33 = 8'd2;  INPUT34 = 8'd0;
          INPUT41 = 8'd2;  INPUT42 = 8'd1;  INPUT43 = 8'd0;  INPUT44 = 8'd1;

          FILTER11 = 8'd1;  FILTER12 = 8'd0;  FILTER13 = 8'd1;
          FILTER21 = 8'd1;  FILTER22 = 8'd1;  FILTER23 = 8'd0;
          FILTER31 = 8'd0;  FILTER32 = 8'd1;  FILTER33 = 8'd1;

      #500 RST= 1'b1;

end
```

When the input values are given as a 4×4 matrix and a 3×3 filter, as shown in the table above, the output results are observed as follows: result11 = 12, result12 = 13, result21 = 9, and result22 = 10, which are **identical to those produced by the 3×3 systolic array.**

The computation begins when the reset signal (rst) is deasserted (set to 0) at 30 ns. After 14 clock cycles, the final output is generated, and the done_2_2 signal transitions to 1, indicating that the computation has completed.

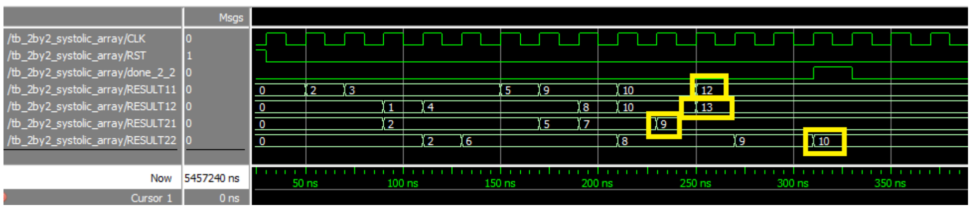
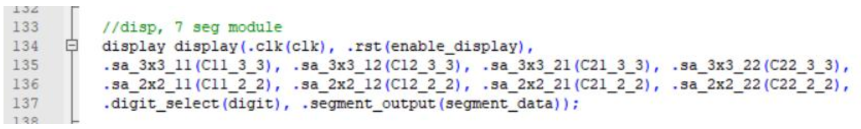

Msgs
/tb_2by2_systolic_array/CLK 0
/tb_2by2_systolic_array/RST 1
/tb_2by2_systolic_array/done_2_2 0
/tb_2by2_systolic_array/RESULT11 0
/tb_2by2_systolic_array/RESULT12 0
/tb_2by2_systolic_array/RESULT21 0
/tb_2by2_systolic_array/RESULT22 0
0 2 3 5 9 10 12
0 1 4 8 10 13
0 2 5 7 9
0 2 6 8 9 10
Now 5457240 ns
Cursor 1 0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
350 ns

Fig 22. waveform of 2x2 systolic array

The total computation time is 280 ns. Since the output matches the expected results from a 2D convolution between the input matrix and the filter, it can be concluded that the model functions correctly. When the input values are given as a 4×4 matrix and a 3×3 filter, as shown in the table above, the output results are observed as follows: result11 = 12, result12 = 13, result21 = 9, and result22 = 10, which are identical to those produced by the 3×3 systolic array. The computation begins when the reset signal (rst) is deasserted (set to 0) at 30 ns. After 14 clock cycles, the final output is generated, and the done_2_2 signal transitions to 1, indicating that the computation has completed. The total computation time is 280 ns. Since the output matches the expected results from a 2D convolution between the input matrix and the filter, it can be concluded that the model functions correctly.

7. Display, 7-Segment Module

7.1 Display, 7-Segment Module Theoretical Design

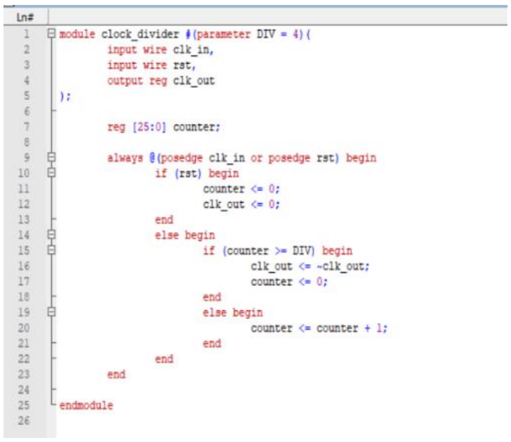


```
//disp, 7 seg module
display display(.clk(clk), .rst(enable_display),
.sa_3x3_11(C11_3_3), .sa_3x3_12(C12_3_3), .sa_3x3_21(C21_3_3), .sa_3x3_22(C22_3_3),
.sa_2x2_11(C11_2_2), .sa_2x2_12(C12_2_2), .sa_2x2_21(C21_2_2), .sa_2x2_22(C22_2_2),
.digit_select(digit), .segment_output(segment_data));
```

Figure 23. Display Module Block Inside the Top Module

The display system consists of three modules: **clock divider**, **seven_segment**, and **display**. It converts eight 8-bit outputs from the systolic arrays into 3-digit decimal numbers and sequentially presents them on a 7-segment display. The **display** module outputs each value one at a time at fixed intervals, ensuring that results appear in order without overlapping. It uses a **clock divider** and index control logic to manage the overall flow of output, helping to visually distinguish different stages of system operation. The **seven_segment** module addresses the hardware limitation of the 7-segment display, which can show only one digit at a time, by using a multiplexing approach. It decomposes each value into individual digits and rapidly cycles through them, creating the visual effect of a full 3-digit number being displayed simultaneously. Together, they form the final stage of the Top module, enabling clear visualization of computation results.

7.2 Clock Divider Module



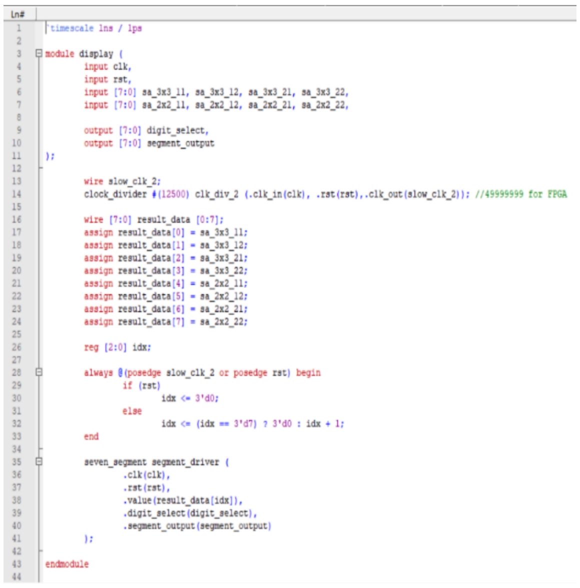
```
module clock_divider #(parameter DIV = 4)(
        input wire clk_in,
        input wire rst,
        output reg clk_out
);

        reg [25:0] counter;

        always @(posedge clk_in or posedge rst) begin
                if (rst) begin
                        counter <= 0;
                        clk_out <= 0;
                end
                else begin
                        if (counter >= DIV) begin
                                clk_out <= ~clk_out;
                                counter <= 0;
                        end
                        else begin
                                counter <= counter + 1;
                        end
                end
        end

endmodule
```

Fig 23. Clock Divider Module

The Clock Divider is a key module responsible for timing control. It converts a high-frequency input clock (clk_in) into a **slower output clock** (clk_out). Internally, it uses a 26-bit counter that increments on every rising edge of clk_in. When the counter reaches a predefined DIV value, clk_out toggles and the counter resets. The DIV value determines how many input clock cycles are required for one toggle of the output clock, thereby controlling its period.

This module is used in both the Display Module and the 7-Segment Module, each with different DIV settings. In the Display Module, the **DIV value is set to 49,999,999** to display each output value at 1-second intervals. In the 7-Segment Module, **the DIV value is set to 99,999** to rapidly cycle through digit positions. This enables smooth and flicker-free display of 3-digit numbers using the persistence of vision effect.

7.3 Display Module



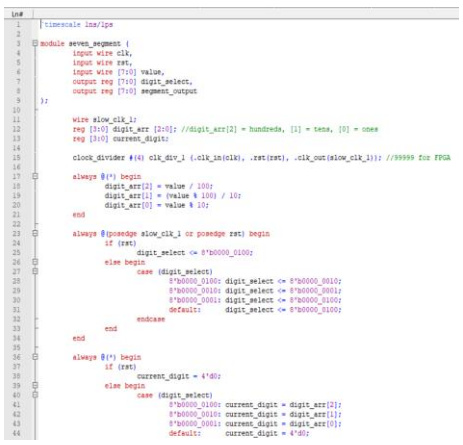
```verilog
`timescale 1ns / 1ps

module display (
        input clk,
        input rst,
        input [7:0] sa_3x3_11, sa_3x3_12, sa_3x3_21, sa_3x3_22,
        input [7:0] sa_2x2_11, sa_2x2_12, sa_2x2_21, sa_2x2_22,

        output [7:0] digit_select,
        output [7:0] segment_output
);

        wire slow_clk_2;
        clock_divider #(12500) clk_div_2 (.clk_in(clk), .rst(rst),.clk_out(slow_clk_2)); //49999999 for FPGA

        wire [7:0] result_data [0:7];
        assign result_data[0] = sa_3x3_11;
        assign result_data[1] = sa_3x3_12;
        assign result_data[2] = sa_3x3_21;
        assign result_data[3] = sa_3x3_22;
        assign result_data[4] = sa_2x2_11;
        assign result_data[5] = sa_2x2_12;
        assign result_data[6] = sa_2x2_21;
        assign result_data[7] = sa_2x2_22;

        reg [2:0] idx;

        always @(posedge slow_clk_2 or posedge rst) begin
                if (rst)
                        idx <= 3'd0;
                else
                        idx <= (idx == 3'd7) ? 3'd0 : idx + 1;
        end

        seven_segment segment_driver (
                .clk(clk),
                .rst(rst),
                .value(result_data[idx]),
                .digit_select(digit_select),
                .segment_output(segment_output)
        );

endmodule
```

The display module is responsible for sequentially presenting eight 8-bit output values—four from the 3×3 Systolic Array and four from the 2×2 Systolic Array—on a 7-segment display. Each value is shown for approximately one second, and to achieve this behavior, the module is composed of a clock divider, index control logic, and a submodule called seven_segment. The input clock (clk) is first divided into a slower clock (**slow_clk_2**) using the clock_divider. This slower clock determines the interval between output transitions. On the actual FPGA board, the DIV value is set to 49,999,999 to produce a 1-second interval; in simulation, a smaller DIV value is used for easier waveform observation.

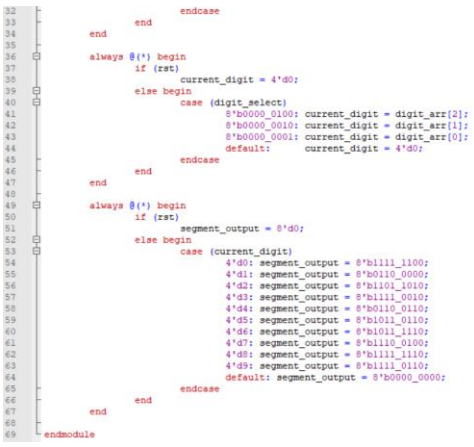
Fig 24. Display module

The eight input values are stored in an internal array (result_data), and a 3-bit index register (idx) increments from 0 to 7 on every rising edge of slow_clk_2. This index selects one value at a time, which is then passed to the seven_segment module for display. The seven_segment module handles digit decomposition and multiplexed output.

7.4 7-Segment module



```verilog
`timescale 1ns/1ps

module seven_segment (
        input wire clk,
        input wire rst,
        input wire [7:0] value,
        output reg [7:0] digit_select,
        output reg [7:0] segment_output
);

        wire slow_clk_1;
        reg [3:0] digit_arr [2:0]; //digit_arr[2] = hundreds, [1] = tens, [0] = ones
        reg [3:0] current_digit;

        clock_divider #(4) clk_div_1 (.clk_in(clk), .rst(rst), .clk_out(slow_clk_1)); //99999 for FPGA

        always @(*) begin
                digit_arr[2] = value / 100;
                digit_arr[1] = (value % 100) / 10;
                digit_arr[0] = value % 10;
        end

        always @(posedge slow_clk_1 or posedge rst) begin
                if (rst)
                        digit_select <= 8'b0000_0100;
                else begin
                        case (digit_select)
                                8'b0000_0100: digit_select <= 8'b0000_0010;
                                8'b0000_0010: digit_select <= 8'b0000_0001;
                                8'b0000_0001: digit_select <= 8'b0000_0100;
                                default:      digit_select <= 8'b0000_0100;
                        endcase
                end
        end

        always @(*) begin
                if (rst)
                        current_digit = 4'd0;
                else begin
                        case (digit_select)
                                8'b0000_0100: current_digit = digit_arr[2];
                                8'b0000_0010: current_digit = digit_arr[1];
                                8'b0000_0001: current_digit = digit_arr[0];
                                default:      current_digit = 4'd0;
```

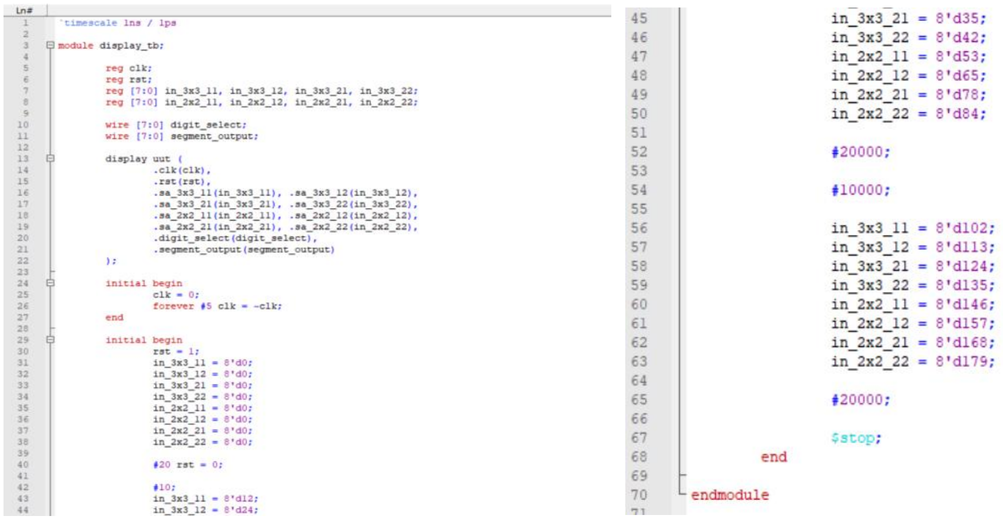


```verilog
                        endcase
                end
        end

        always @(*) begin
                if (rst)
                        current_digit = 4'd0;
                else begin
                        case (digit_select)
                                8'b0000_0100: current_digit = digit_arr[2];
                                8'b0000_0010: current_digit = digit_arr[1];
                                8'b0000_0001: current_digit = digit_arr[0];
                                default:      current_digit = 4'd0;
                        endcase
                end
        end

        always @(*) begin
                if (rst)
                        segment_output = 8'd0;
                else begin
                        case (current_digit)
                                4'd0: segment_output = 8'b1111_1100;
                                4'd1: segment_output = 8'b0110_0000;
                                4'd2: segment_output = 8'b1101_1010;
                                4'd3: segment_output = 8'b1111_0010;
                                4'd4: segment_output = 8'b0110_0110;
                                4'd5: segment_output = 8'b1011_0110;
                                4'd6: segment_output = 8'b1011_1110;
                                4'd7: segment_output = 8'b1110_0100;
                                4'd8: segment_output = 8'b1111_1110;
                                4'd9: segment_output = 8'b1111_0110;
                                default: segment_output = 8'b0000_0000;
                        endcase
                end
        end

endmodule
```

Figure 25. 7-Segment Module

The seven_segment module converts an 8-bit input value into a 3-digit number and displays it smoothly on a 7-segment display. It consists of a clock divider, digit decomposition logic, and segment output logic. The input value is split into **hundreds, tens, and unit digits**, which are stored internally. These digits are displayed one at a time in rapid succession using a slower clock (**slow_clk_1**). The digit_select signal cycles through each digit, and the corresponding value is converted into a 7-segment pattern. Through this process, the display appears to show all three digits simultaneously due to the persistence of vision effect. On the actual FPGA board, the DIV value is set to 99999 to achieve a flicker-free display, while in simulation, it is reduced to 4 for easier waveform observation. This module plays a key role in presenting computational results in a clear and readable format.

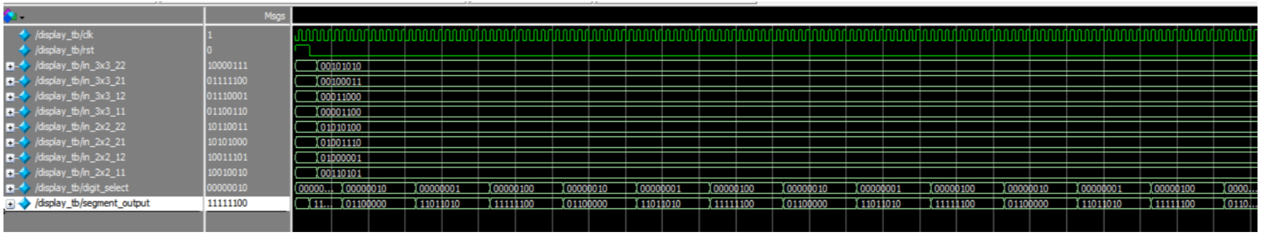
7.5 Display Module Testbench



```verilog
`timescale 1ns / 1ps

module display_tb;

        reg clk;
        reg rst;
        reg [7:0] in_3x3_11, in_3x3_12, in_3x3_21, in_3x3_22;
        reg [7:0] in_2x2_11, in_2x2_12, in_2x2_21, in_2x2_22;

        wire [7:0] digit_select;
        wire [7:0] segment_output;

        display uut (
                .clk(clk),
                .rst(rst),
                .sa_3x3_11(in_3x3_11), .sa_3x3_12(in_3x3_12),
                .sa_3x3_21(in_3x3_21), .sa_3x3_22(in_3x3_22),
                .sa_2x2_11(in_2x2_11), .sa_2x2_12(in_2x2_12),
                .sa_2x2_21(in_2x2_21), .sa_2x2_22(in_2x2_22),
                .digit_select(digit_select),
                .segment_output(segment_output)
        );

        initial begin
                clk = 0;
                forever #5 clk = ~clk;
        end

        initial begin
                rst = 1;
                in_3x3_11 = 8'd0;
                in_3x3_12 = 8'd0;
                in_3x3_21 = 8'd0;
                in_3x3_22 = 8'd0;
                in_2x2_11 = 8'd0;
                in_2x2_12 = 8'd0;
                in_2x2_21 = 8'd0;
                in_2x2_22 = 8'd0;

                #20 rst = 0;

                #10;
                in_3x3_11 = 8'd12;
                in_3x3_12 = 8'd24;
                in_3x3_21 = 8'd35;
                in_3x3_22 = 8'd42;
                in_2x2_11 = 8'd53;
                in_2x2_12 = 8'd65;
                in_2x2_21 = 8'd78;
                in_2x2_22 = 8'd84;

                #20000;

                #10000;

                in_3x3_11 = 8'd102;
                in_3x3_12 = 8'd113;
                in_3x3_21 = 8'd124;
                in_3x3_22 = 8'd135;
                in_2x2_11 = 8'd146;
                in_2x2_12 = 8'd157;
                in_2x2_21 = 8'd168;
                in_2x2_22 = 8'd179;

                #20000;

                $stop;
        end

endmodule
```

Figure 26. Testbench of Display Module

The display_tb testbench verifies whether the outputs from the 3×3 and 2×2 Systolic Arrays are correctly displayed through the Display Module. Eight 8-bit input values are provided, and the test confirms that they appear sequentially on the 7-segment display. At the start, all inputs are reset to zero, and a 10 ns clock is generated. Two sets of input values are applied at different times to observe output changes. The clock divider is set to DIV = 12500 to speed up output transitions and make waveform analysis easier.

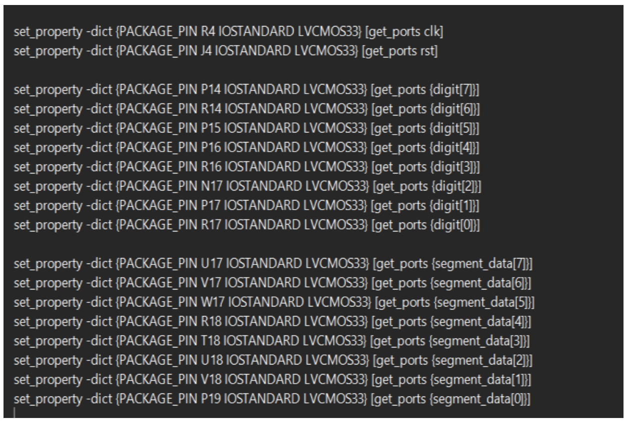


```
                              Msgs
/display_tb/clk               1
/display_tb/rst               0
/display_tb/in_3x3_22         10000111    00101010
/display_tb/in_3x3_21         01111100    00100011
/display_tb/in_3x3_12         01110001    00011000
/display_tb/in_3x3_11         01100110    00001100
/display_tb/in_2x2_22         10110011    01010100
/display_tb/in_2x2_21         10101000    01001110
/display_tb/in_2x2_12         10011101    01000001
/display_tb/in_2x2_11         10010010    00110101
/display_tb/digit_select      00000010    00000... 00000010 00000001 00000100 00000010 00000001 00000100 00000010 00000001 00000100 00000010 00000001 00000100 0000...
/display_tb/segment_output    11111100    11... 01100000 11011010 11111100 01100000 11011010 11111100 01100000 11011010 11111100 01100000 11011010 11111100 0110...
```

Figure 27. Display Module Wave Simulaiton

The simulation results confirmed that both segment_output and digit_select signals functioned as expected. All eight input values were displayed in order, and digit_select rapidly cycled through each digit position, activating the corresponding 7-segment display. As a result, segment_output changed periodically, successfully creating the visual effect of a 3-digit number being displayed **simultaneously**. By adjusting the clock cycles and parameter values, each output remained visible for a sufficient duration, and the final value was also correctly maintained. These results verify that the module operates as intended and can provide accurate visual output when implemented on an FPGA board.

7.6 XDC Code (Vivado)



```
set_property -dict {PACKAGE_PIN R4 IOSTANDARD LVCMOS33} [get_ports clk]
set_property -dict {PACKAGE_PIN J4 IOSTANDARD LVCMOS33} [get_ports rst]

set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {digit[7]}]
set_property -dict {PACKAGE_PIN R14 IOSTANDARD LVCMOS33} [get_ports {digit[6]}]
set_property -dict {PACKAGE_PIN P15 IOSTANDARD LVCMOS33} [get_ports {digit[5]}]
set_property -dict {PACKAGE_PIN P16 IOSTANDARD LVCMOS33} [get_ports {digit[4]}]
set_property -dict {PACKAGE_PIN R16 IOSTANDARD LVCMOS33} [get_ports {digit[3]}]
set_property -dict {PACKAGE_PIN N17 IOSTANDARD LVCMOS33} [get_ports {digit[2]}]
set_property -dict {PACKAGE_PIN P17 IOSTANDARD LVCMOS33} [get_ports {digit[1]}]
set_property -dict {PACKAGE_PIN R17 IOSTANDARD LVCMOS33} [get_ports {digit[0]}]

set_property -dict {PACKAGE_PIN U17 IOSTANDARD LVCMOS33} [get_ports {segment_data[7]}]
set_property -dict {PACKAGE_PIN V17 IOSTANDARD LVCMOS33} [get_ports {segment_data[6]}]
set_property -dict {PACKAGE_PIN W17 IOSTANDARD LVCMOS33} [get_ports {segment_data[5]}]
set_property -dict {PACKAGE_PIN R18 IOSTANDARD LVCMOS33} [get_ports {segment_data[4]}]
set_property -dict {PACKAGE_PIN T18 IOSTANDARD LVCMOS33} [get_ports {segment_data[3]}]
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports {segment_data[2]}]
set_property -dict {PACKAGE_PIN V18 IOSTANDARD LVCMOS33} [get_ports {segment_data[1]}]
set_property -dict {PACKAGE_PIN P19 IOSTANDARD LVCMOS33} [get_ports {segment_data[0]}]
```

Figure 28. XDC Pin Configuration

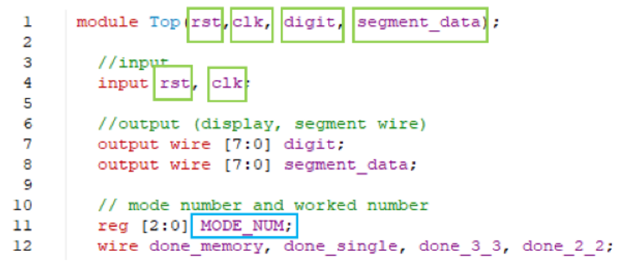
This XDC file maps the Top module's ports to the FPGA board's I/O pins in Vivado. clk and rst are connected to pins R4 and J4, handling the system clock and reset. digit[7:0] selects each digit of the 7-segment display via pins P14 to R17, while segment_data[7:0] controls the display segments through pins U17 to P19. This mapping ensures that the Top module's output is correctly displayed on the FPGA board.

8. Top Module

8.1. Top Module Theoretical Approach

The Top module orchestrates all subsystems: controller, memory, PE, systolic arrays, and display. It monitors **done signals** from each module to increment **MODE_NUM** and transitions through each computation stage.

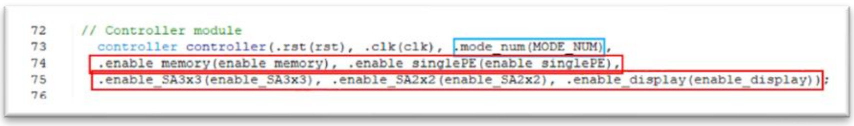
8.2 Top Module Implementation Code



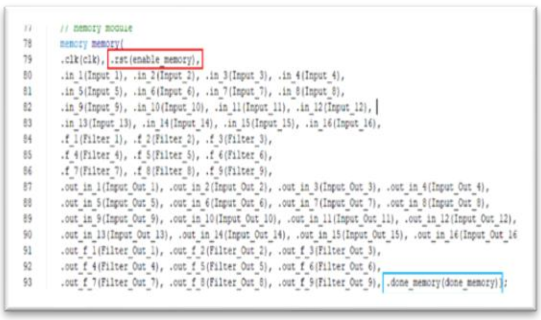
```
module Top(rst,clk, digit, segment_data);

  //input
  input rst, clk;

  //output (display, segment wire)
  output wire [7:0] digit;
  output wire [7:0] segment_data;

  // mode number and worked number
  reg [2:0] MODE_NUM;
  wire done_memory, done_single, done_3_3, done_2_2;
```

Fig 29. Define Top module

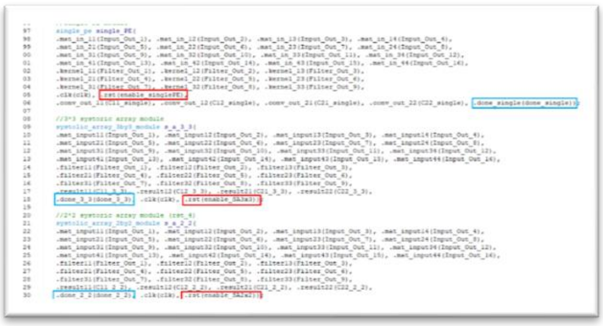
The Top module initializes 16 input data values and 9 filter data. These propagate through memory and computational modules before being displayed.



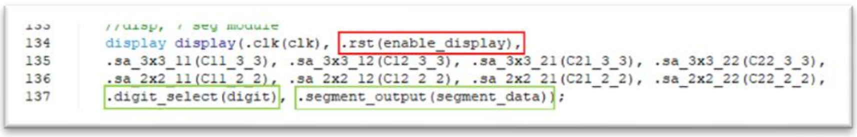
```
// Controller module
controller controller(.rst(rst), .clk(clk), .mode_num(MODE_NUM),
.enable_memory(enable_memory), .enable_singlePE(enable_singlePE),
.enable_SA3x3(enable_SA3x3), .enable_SA2x2(enable_SA2x2), .enable_display(enable_display));
```



```
// memory module
memory memory(
.clk(clk), .rst(enable_memory),
.in_1(Input_1), .in_2(Input_2), .in_3(Input_3), .in_4(Input_4),
.in_5(Input_5), .in_6(Input_6), .in_7(Input_7), .in_8(Input_8),
.in_9(Input_9), .in_10(Input_10), .in_11(Input_11), .in_12(Input_12),
.in_13(Input_13), .in_14(Input_14), .in_15(Input_15), .in_16(Input_16),
.f_1(Filter_1), .f_2(Filter_2), .f_3(Filter_3),
.f_4(Filter_4), .f_5(Filter_5), .f_6(Filter_6),
.f_7(Filter_7), .f_8(Filter_8), .f_9(Filter_9),
.out_in_1(Input_Out_1), .out_in_2(Input_Out_2), .out_in_3(Input_Out_3), .out_in_4(Input_Out_4),
.out_in_5(Input_Out_5), .out_in_6(Input_Out_6), .out_in_7(Input_Out_7), .out_in_8(Input_Out_8),
.out_in_9(Input_Out_9), .out_in_10(Input_Out_10), .out_in_11(Input_Out_11), .out_in_12(Input_Out_12),
.out_in_13(Input_Out_13), .out_in_14(Input_Out_14), .out_in_15(Input_Out_15), .out_in_16(Input_Out_16
.out_f_1(Filter_Out_1), .out_f_2(Filter_Out_2), .out_f_3(Filter_Out_3),
.out_f_4(Filter_Out_4), .out_f_5(Filter_Out_5), .out_f_6(Filter_Out_6),
.out_f_7(Filter_Out_7), .out_f_8(Filter_Out_8), .out_f_9(Filter_Out_9), .done_memory(done_memory));
```

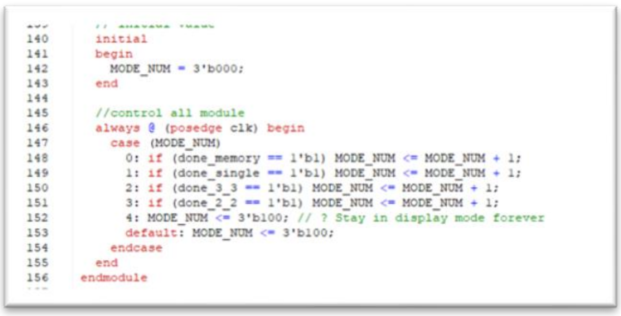


```
single_pe single_PE(
.mat_in_11(Input_Out_1), .mat_in_12(Input_Out_2), .mat_in_13(Input_Out_3), .mat_in_14(Input_Out_4),
.mat_in_21(Input_Out_5), .mat_in_22(Input_Out_6), .mat_in_23(Input_Out_7), .mat_in_24(Input_Out_8),
.mat_in_31(Input_Out_9), .mat_in_32(Input_Out_10), .mat_in_33(Input_Out_11), .mat_in_34(Input_Out_12),
.mat_in_41(Input_Out_13), .mat_in_42(Input_Out_14), .mat_in_43(Input_Out_15), .mat_in_44(Input_Out_16),
.kernel_11(Filter_Out_1), .kernel_12(Filter_Out_2), .kernel_13(Filter_Out_3),
.kernel_21(Filter_Out_4), .kernel_22(Filter_Out_5), .kernel_23(Filter_Out_6),
.kernel_31(Filter_Out_7), .kernel_32(Filter_Out_8), .kernel_33(Filter_Out_9),
.clk(clk), .rst(enable_singlePE),
.conv_out_11(C11_single), .conv_out_12(C12_single), .conv_out_21(C21_single), .conv_out_22(C22_single), .done_single(done_single));

//3*3 systolic array module
systolic_array_3by3_module s_a_3_3(
.mat_input11(Input_Out_1), .mat_input12(Input_Out_2), .mat_input13(Input_Out_3), .mat_input14(Input_Out_4),
.mat_input21(Input_Out_5), .mat_input22(Input_Out_6), .mat_input23(Input_Out_7), .mat_input24(Input_Out_8),
.mat_input31(Input_Out_9), .mat_input32(Input_Out_10), .mat_input33(Input_Out_11), .mat_input34(Input_Out_12),
.mat_input41(Input_Out_13), .mat_input42(Input_Out_14), .mat_input43(Input_Out_15), .mat_input44(Input_Out_16),
.filter11(Filter_Out_1), .filter12(Filter_Out_2), .filter13(Filter_Out_3),
.filter21(Filter_Out_4), .filter22(Filter_Out_5), .filter23(Filter_Out_6),
.filter31(Filter_Out_7), .filter32(Filter_Out_8), .filter33(Filter_Out_9),
.result11(C11_3_3), .result12(C12_3_3), .result21(C21_3_3), .result22(C22_3_3),
.done_3_3(done_3_3), .clk(clk), .rst(enable_SA3x3));

//2*2 systolic array module (ret_4)
systolic_array_2by2_module s_a_2_2(
.mat_input11(Input_Out_1), .mat_input12(Input_Out_2), .mat_input13(Input_Out_3), .mat_input14(Input_Out_4),
.mat_input21(Input_Out_5), .mat_input22(Input_Out_6), .mat_input23(Input_Out_7), .mat_input24(Input_Out_8),
.mat_input31(Input_Out_9), .mat_input32(Input_Out_10), .mat_input33(Input_Out_11), .mat_input34(Input_Out_12),
.mat_input41(Input_Out_13), .mat_input42(Input_Out_14), .mat_input43(Input_Out_15), .mat_input44(Input_Out_16),
.filter11(Filter_Out_1), .filter12(Filter_Out_2), .filter13(Filter_Out_3),
.filter21(Filter_Out_4), .filter22(Filter_Out_5), .filter23(Filter_Out_6),
.filter31(Filter_Out_7), .filter32(Filter_Out_8), .filter33(Filter_Out_9),
.result11(C11_2_2), .result12(C12_2_2), .result21(C21_2_2), .result22(C22_2_2),
.done_2_2(done_2_2), .clk(clk), .rst(enable_SA2x2));
```



```
//disp, 7 seg module
display display(.clk(clk), .rst(enable_display),
.sa_3x3_11(C11_3_3), .sa_3x3_12(C12_3_3), .sa_3x3_21(C21_3_3), .sa_3x3_22(C22_3_3),
.sa_2x2_11(C11_2_2), .sa_2x2_12(C12_2_2), .sa_2x2_21(C21_2_2), .sa_2x2_22(C22_2_2),
.digit_select(digit), .segment_output(segment_data));
```

Fig 30. Connections with all the Subsystems : (a) Controller Module (b) Memory Module (c) Computational Module (d) Display Module

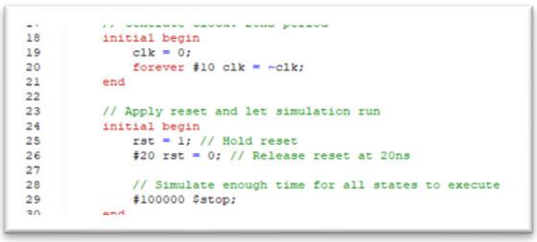


```
initial
begin
  MODE_NUM = 3'b000;
end

//control all module
always @ (posedge clk) begin
  case (MODE_NUM)
    0: if (done_memory == 1'b1) MODE_NUM <= MODE_NUM + 1;
    1: if (done_single == 1'b1) MODE_NUM <= MODE_NUM + 1;
    2: if (done_3_3 == 1'b1) MODE_NUM <= MODE_NUM + 1;
    3: if (done_2_2 == 1'b1) MODE_NUM <= MODE_NUM + 1;
    4: MODE_NUM <= 3'b100; // ? Stay in display mode forever
    default: MODE_NUM <= 3'b100;
  endcase
end
endmodule
```

Each module is enabled via **enable_xx** signals, and transitions are coordinated using **done_xx** flags. Once all computations are complete, the display state (S5) remains active to showcase the results.

Fig 31. Top Module: Mode Number Update Mechanism

8.3 Top module testbench and simulation

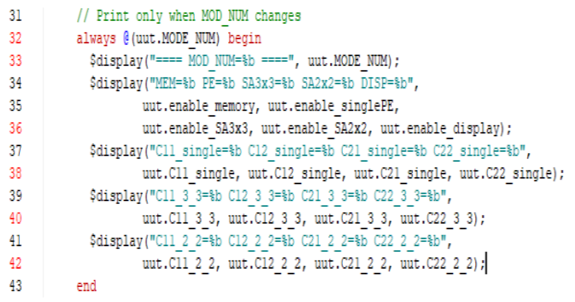


```
initial begin
    clk = 0;
    forever #10 clk = ~clk;
end

// Apply reset and let simulation run
initial begin
    rst = 1; // Hold reset
    #20 rst = 0; // Release reset at 20ns

    // Simulate enough time for all states to execute
    #100000 $stop;
```

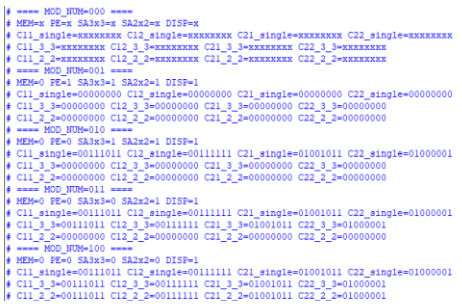
The Top module has a straightforward structure, with rst and clk as inputs and digit and segment_data as outputs. Accordingly, its testbench was designed with a simple configuration: the clock was set to a 20ns period, and rst was deasserted after 20ns.

Fig 32. Top Module Testbench

However, digit and segment_data alone do not provide insight into the system's internal operation. They do not convey critical information such as the state transition flow or the computation results from the Single PE, 3x3 Systolic Array, and 2x2 Systolic Array modules. To address this, additional monitoring logic was implemented to print key variables to the **console and waveform** during simulation. Specifically, the enable signals for each module and the computed results (C11, C12, C21, C22) were displayed.



```verilog
    // Print only when MOD_NUM changes
    always @(uut.MODE_NUM) begin
      $display("==== MOD_NUM=%b ====", uut.MODE_NUM);
      $display("MEM=%b PE=%b SA3x3=%b SA2x2=%b DISP=%b",
              uut.enable_memory, uut.enable_singlePE,
              uut.enable_SA3x3, uut.enable_SA2x2, uut.enable_display);
      $display("C11_single=%b C12_single=%b C21_single=%b C22_single=%b",
              uut.C11_single, uut.C12_single, uut.C21_single, uut.C22_single);
      $display("C11_3_3=%b C12_3_3=%b C21_3_3=%b C22_3_3=%b",
              uut.C11_3_3, uut.C12_3_3, uut.C21_3_3, uut.C22_3_3);
      $display("C11_2_2=%b C12_2_2=%b C21_2_2=%b C22_2_2=%b",
              uut.C11_2_2, uut.C12_2_2, uut.C21_2_2, uut.C22_2_2);
    end
```



```
# ==== MOD_NUM=000 ====
# MEM=x PE=x SA3x3=x SA2x2=x DISP=x
# C11_single=xxxxxxxx C12_single=xxxxxxxx C21_single=xxxxxxxx C22_single=xxxxxxxx
# C11_3_3=xxxxxxxx C12_3_3=xxxxxxxx C21_3_3=xxxxxxxx C22_3_3=xxxxxxxx
# C11_2_2=xxxxxxxx C12_2_2=xxxxxxxx C21_2_2=xxxxxxxx C22_2_2=xxxxxxxx
# ==== MOD_NUM=001 ====
# MEM=0 PE=1 SA3x3=1 SA2x2=1 DISP=1
# C11_single=00000000 C12_single=00000000 C21_single=00000000 C22_single=00000000
# C11_3_3=00000000 C12_3_3=00000000 C21_3_3=00000000 C22_3_3=00000000
# C11_2_2=00000000 C12_2_2=00000000 C21_2_2=00000000 C22_2_2=00000000
# ==== MOD_NUM=010 ====
# MEM=0 PE=0 SA3x3=1 SA2x2=1 DISP=1
# C11_single=00111011 C12_single=00111111 C21_single=01001011 C22_single=01000001
# C11_3_3=00000000 C12_3_3=00000000 C21_3_3=00000000 C22_3_3=00000000
# C11_2_2=00000000 C12_2_2=00000000 C21_2_2=00000000 C22_2_2=00000000
# ==== MOD_NUM=011 ====
# MEM=0 PE=0 SA3x3=0 SA2x2=1 DISP=1
# C11_single=00111011 C12_single=00111111 C21_single=01001011 C22_single=01000001
# C11_3_3=00111011 C12_3_3=00111111 C21_3_3=01001011 C22_3_3=01000001
# C11_2_2=00000000 C12_2_2=00000000 C21_2_2=00000000 C22_2_2=00000000
# ==== MOD_NUM=100 ====
# MEM=0 PE=0 SA3x3=0 SA2x2=0 DISP=1
# C11_single=00111011 C12_single=00111111 C21_single=01001011 C22_single=01000001
# C11_3_3=00111011 C12_3_3=00111111 C21_3_3=01001011 C22_3_3=01000001
# C11_2_2=00111011 C12_2_2=00111111 C21_2_2=01001011 C22_2_2=01000001
```

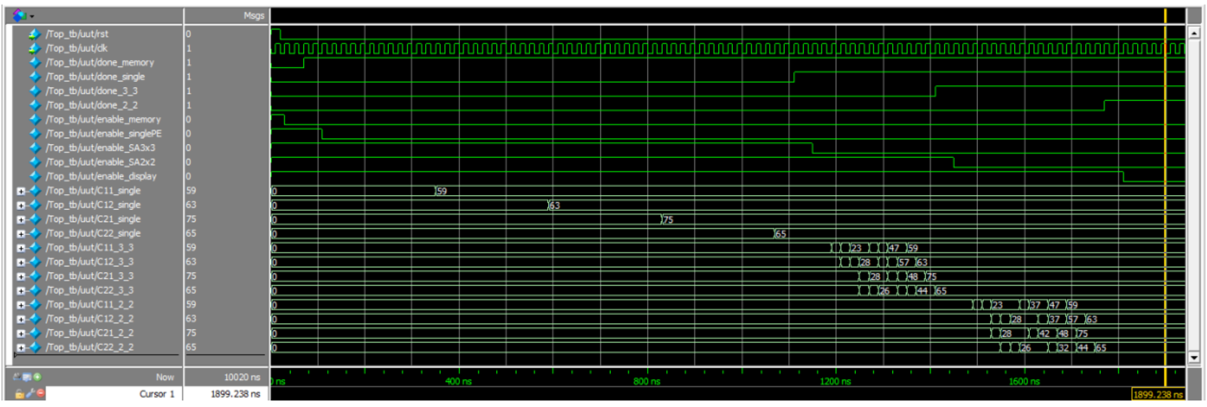


Fig 33. Top Module Simulation (a) Testbench Code to display results in console (b) Results displayed in Console (c) Results displayed in Waveform

This approach made it possible to verify that all modules were properly connected and activated in sequence, confirming the Top module's role in coordinating the overall system operation. The **console output Fig. 33-b** clearly shows how state transitions (MOD_NUM changes) control module enable signals (enable_memory, enable_singlePE, etc.) and how each computation result appears as expected. In addition**, the waveform view Fig. 33-c** provides a timeline of these events, showing precise timings of done_memory, enable_singlePE, and the sequential appearance of computation results. For example, after **done_memory** transitions **from 0 (incomplete) to 1 (complete)**, **enable_singlePE** changes from **1 (off) to 0 (on)** shortly thereafter, activating state S2. Following this transition, the **computation results (C11_single through C22_single) are sequentially produced and visualized**.

This approach allowed verification that all modules were properly connected and executed sequentially, confirming the Top module's role as the orchestrator of the overall system.

III. Result

1. Pre-Demonstration FPGA Test Results

Prior to the official demonstration, the system was tested on the FPGA board to ensure proper functionality under actual operating conditions. Input matrices and filter kernels were manually applied, and the results of the 2D convolution operation were observed to verify that the module performed as intended.

Test Case 1

0	6	8	7
5	7	5	5
4	3	4	6
0	6	0	7

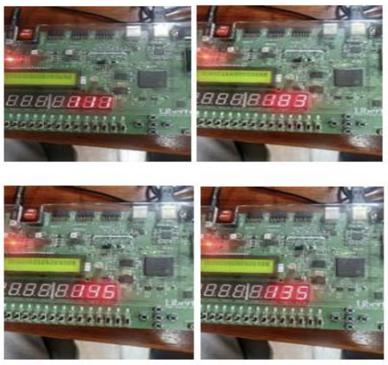
Input matrix

1	5	1
0	1	5
4	4	9

Filter matrix

111	183
146	135

Output matrix



Result Test Case 2

Fig 34. Test Case 1

Test Case 2

1	4	7	5
0	8	6	3
5	10	4	2
3	6	9	7

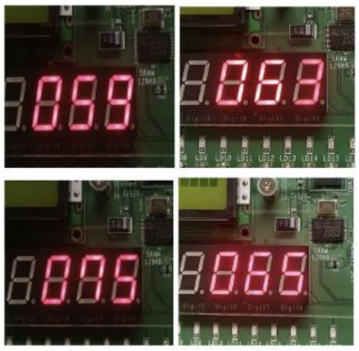
Input matrix

3	0	2
1	1	0
2	2	1

Filter matrix

59	63
75	65

Output matrix



Result

Fig 35. Test Case 2

2. Demonstration Result

2	10	9	6
0	3	2	1
1	8	10	9
2	10	6	1

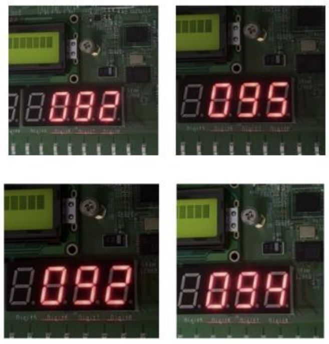
Input matrix

1	4	0
2	2	3
2	1	1

Filter matrix

82	95
92	94

Output matrix

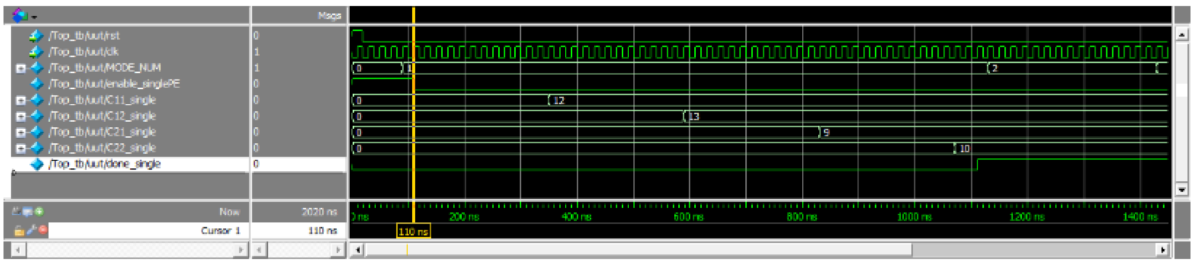


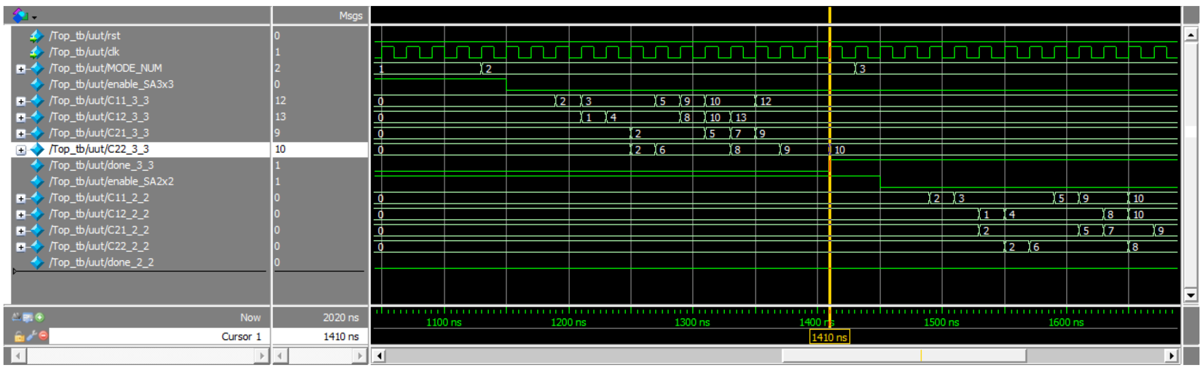
Result

Fig 36. Demonstration Result

The demonstration confirmed that the system functioned correctly on the FPGA board. All convolution results were accurately computed, and the output values — **82, 95, 92, and 94** — were displayed sequentially on the 7-segment display at 1-second intervals. Throughout the process, no functional errors occurred, and both the processing logic and display module operated as intended.

3. Performance Comparison (Speedup of 3x3 / 2x2 systolic array over a single PE)





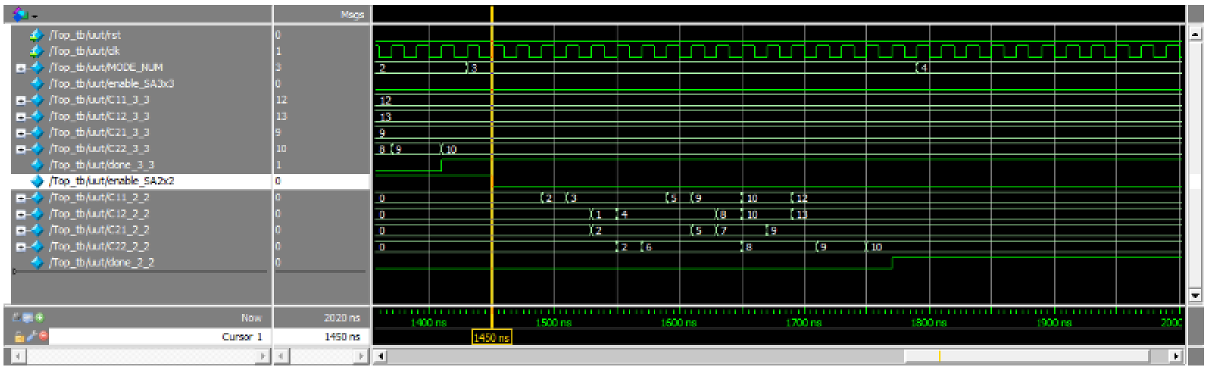
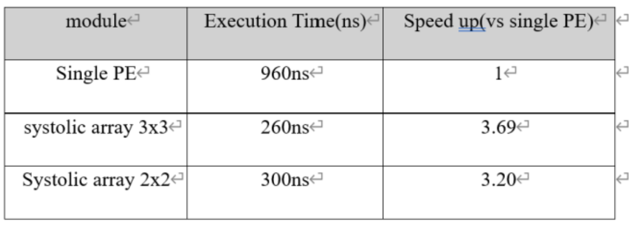


Fig 37. Output of the Top Module : (A)Single PE (B) 3x3 systolic array (C) 2x2 systolic array

The images above show the execution speeds of each module when run from the top module. The measurement is based on the moment when the reset signal input goes low, marking the start of computation, and ends when the final accumulated output value is produced.



module	Execution Time(ns)	Speed up(vs single PE)
Single PE	960ns	1
systolic array 3x3	260ns	3.69
Systolic array 2x2	300ns	3.20

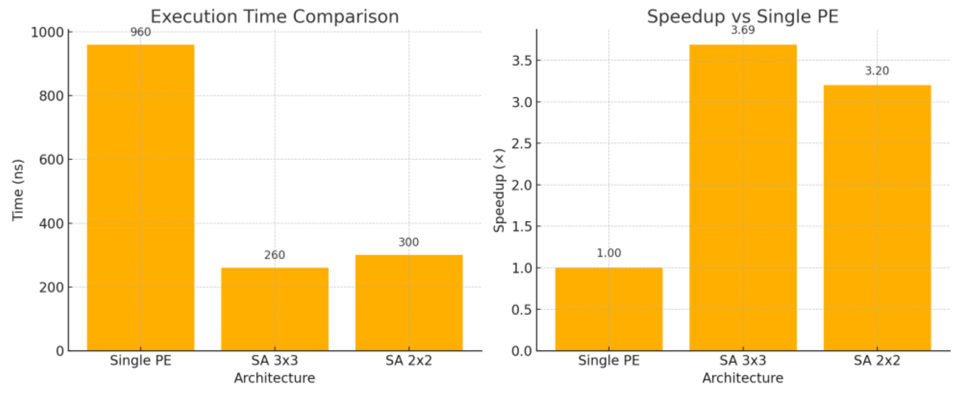
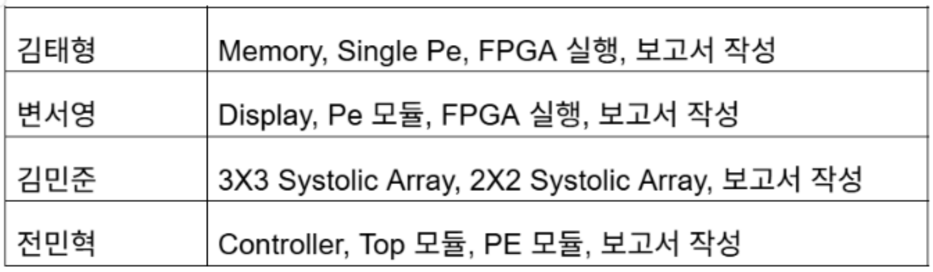


Fig 38. Execution Time of Each Computational Modules and Speed up comparing to Single PE (a) Table (b) Graph

The execution times were measured **as 960 ns for the Single PE, 300 ns for the 2×2 array, and 260 ns for the 3×3 array.** These results demonstrate that utilizing more PEs enables faster computation. This is because, in parallel processing, performing more operations simultaneously leads to quicker completion. Through these results, it can be confirmed that the systolic array is a viable structure for use as a hardware accelerator, and that the systolic array was successfully implemented in this project.

4. Contribution

This project involved the implementation of a 2D convolution calculator, with each team member responsible for designing and testing specific modules. Each member independently handled code development, testbench creation, and report writing for their assigned part. Based on this collaboration, the system was successfully integrated and demonstrated.



김태형	Memory, Single Pe, FPGA 실행, 보고서 작성
변서영	Display, Pe 모듈, FPGA 실행, 보고서 작성
김민준	3X3 Systolic Array, 2X2 Systolic Array, 보고서 작성
전민혁	Controller, Top 모듈, PE 모듈, 보고서 작성