Design and Implementation of Convolution Accelerator based on FPGA

Team 3 | Logical Circuit Final Project

1. Introduction

1.1 Objective

Through the team project, we aim to understand Convolutional Neural Networks (CNN) used in Deep Neural Networks (DNN) and to implement these computations in Verilog using three different ways: a single Processing Element (PE), a 2x2 systolic array, and a 3x3 systolic array. The main objective is to get a output the resulting 2x2 matrix on an FPGA by providing a 4x4 input matrix and a 3x3 filter as inputs. Additionally, by using testbenches, we are able to measure and compare the processing speed differences between the single PE implementation and the 2x2 systolic array, as well as between the single PE and the 3x3 systolic array.

1.2 System Overview

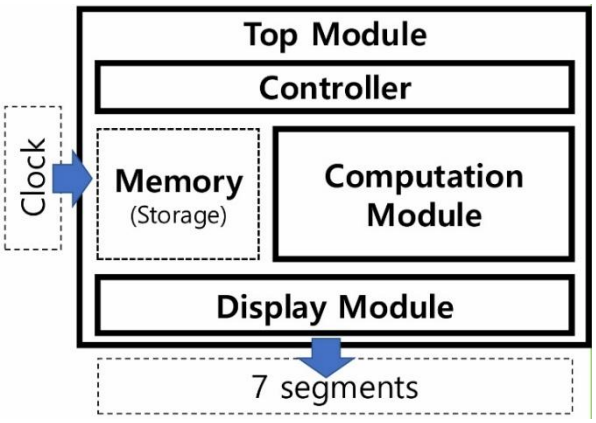

Figure 1. System Design

This system consists of a total of 7 modules, as shown in **Figure 1**: a **Controller**, **Memory**, **Processing Element (PE)**, **2x2 Systolic Array**, **3x3 Systolic Array**, **Display Module**, and a **Top Module** that connects and activates the operation of all these modules.

1.3 Division of Roles

Contribution / Name	Module + Testbench	Report
김민석 (2021312188)	Display	Display, Conclusion
김예훈 (2020310654)	PE, TOP, Single	PE, TOP, Comparison, Conclusion
손민석 (2020314410)	Controller, Memory	Controller, Memory
이승수 (2022315987)	2X2, 3X3	Single, 2X2, 3X3

Table 1. Contribution of each member

Each member also implemented the testbench code for their assigned module.

2. Controller

2-1. Theory

The controller module is designed to initialize and manage the operation of the memory, computation, and display modules within the system. To control these modules, I define a set of input and output ports as follows.

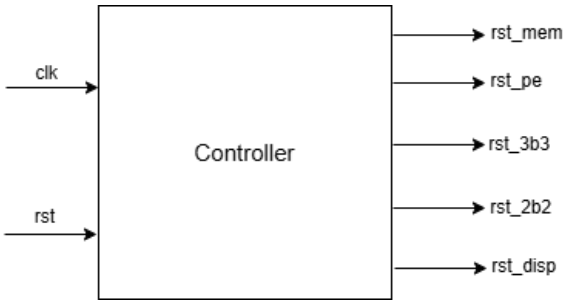

Figure 2. Controller Module

port	Function
clk	clock
rst	reset for controller
rst_mem	Reset signal for memory module
rst_pe	Reset signal for PE module
rst_3b3	Reset signal for 3x3 systolic array module
rst_2b2	Reset signal for 2x2 systolic array module
rst_disp	Reset signal for display module

Table 2. Port Declaration of Controller Module

Since the modules controlled must be activated sequentially, a Moore state machine where outputs (control signals) depend solely on the current state was determined to be the most appropriate implementation for the controller design. The FSM consists of the six states

The value of the states is represented below.

S0 = 0(3'b000), S1 = 1(3'b001), S2 = 2(3'b010), S3 = 3(3'b011), S4 = 4(3'b100), S5 = 5(3'b101)

The output signals of the controller module (rst_mem, rst_pe, rst_3b3, rst_2b2, and rst_disp) are determined solely based on the current state of the FSM. In other words, the input signal clk and rst affect only the state transitions, while the outputs depend exclusively on the present state.

And the controller operates on the **posedge** of the clock, with the input reset (rst) triggered asynchronously on the **negedge**, while all output reset signals (rst_mem, rst_pe, rst_3b3, rst_2b2, rst_disp) are updated synchronously; a value of 1 disables the corresponding module, while a value of 0 enables it based on the FSM state.

To manage precise state durations, the controller employs a **counter-based transition mechanism**. Each FSM state is associated with a specific time constant (TIME_Sx), which defines how long the controller should remain in that state, measured in clock cycles.

A 32-bit counter increments on every positive edge of the clock (posedge clk). When the counter reaches or exceeds the predefined threshold for the current state, the FSM transitions to the next state and the counter is reset to zero.

TIME_S0 = 200, TIME_S1 = 200, TIME_S2 = 4000, TIME_S3 = 2000, TIME_S4 = 2000(clock cycle)

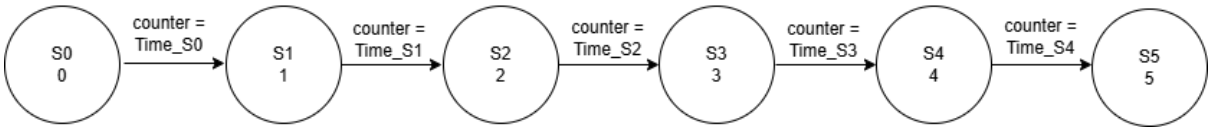

Figure 3. state diagram

State	rst_mem	rst_pe	rst_3b3	rst_2b2	rst_disp	Activated modules	Description
S0	1	1	1	1	0	None	All modules are held in reset.
S1	0	1	1	1	0	Memory	Memory module is activated.
S2	0	0	1	1	0	Memory,PE	PE module is activated for computation.
S3	0	0	0	1	0	Memory,PE 3x3SA	3x3 SA (systolic array) module is activated.
S4	0	0	0	0	0	Memory,PE 3x3SA,2x2SA	All computation modules are activated.
S5	0	0	0	0	1	All	Display module is finally activated.

Table 3. Reset Declaration according to State

2-1-2 Implementation & Testbench plan

In the testbench, the system is first initialized with clk set to 0 and rst set to 1. The clock signal then begins toggling every 5 ns, creating a 100 MHz clock. After 5 ns, the reset signal is deactivated (rst = 0), allowing the controller's FSM to begin its operation. From the moment the reset is deactivated, the internal counter starts incrementing with each clock cycle. Once the counter reaches the specified threshold (TIME_Sx) for the current state, the FSM transitions to the next state. This transition occurs sequentially through all defined states, from S0 to S5.

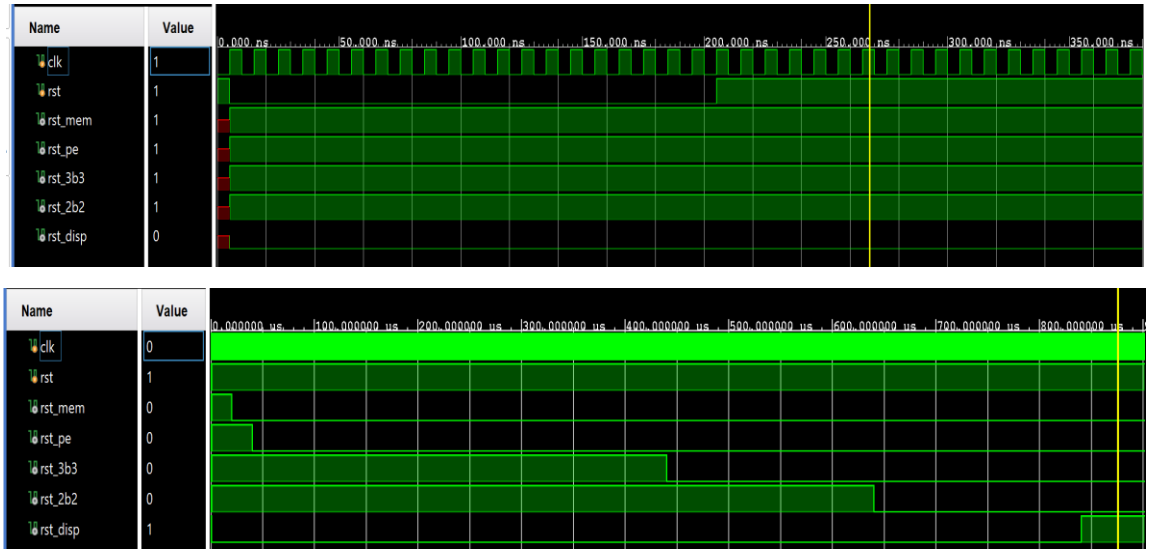

Figure 4. Simulation Result of Controller module

3. Memory

3-1. Theory

The memory module is designed to store and provide a fixed set of input and filter data to processing elements (PE,3x3SA, 2x2SA). It serves as a read-only memory during operation. It contains input

matrix (16 fixed 8-bit values), filter (9 fixed 8-bit values). Each data value is assigned to a corresponding 8-bit register, implemented using **eight_bit_register** modules. These registers are initialized with constant values and controlled clk and reset (rst) signals. When reset is 0, the register values become valid and are made available through dedicated output ports (input_data0 ~ input_data15 and filter_data0 ~ filter_data8).

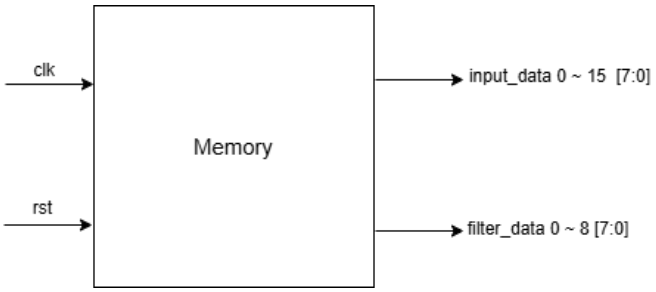

Figure 5. Memory module

The memory module is implemented using structural modeling. It is composed of 25 **eight_bit_register** modules, each of which consists of 8 **one_bit_register** instantiations. The **one_bit_register** is a register based on a **D flip-flop**. The figure is too large to display in full, so only a part of it is shown.

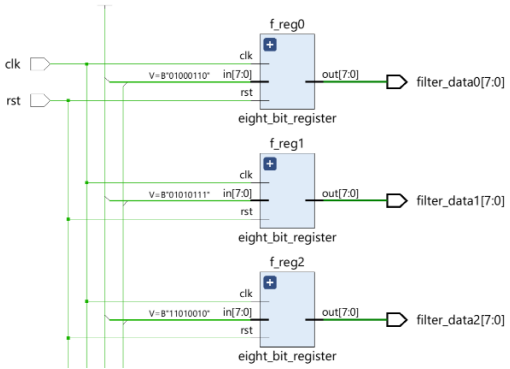

Fig 6. Schematic of Memory module

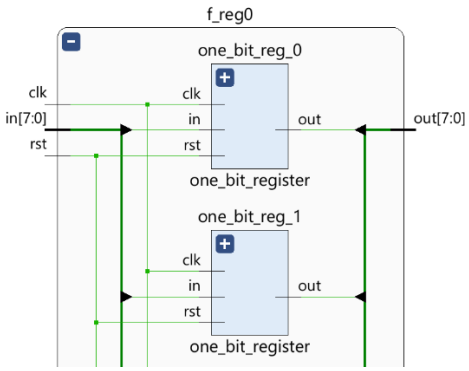

Fig.7 Schematic of eight_bit_register module

3-1-2 Implementation & Testbench plan

In the testbench, **clk** is initially set to 0 and **rst** is set to 1. The clock toggles every 5 ns and **rst** is set to 0 after 10 ns, allowing the register modules to operate. As a result, their predefined constant values will be output.

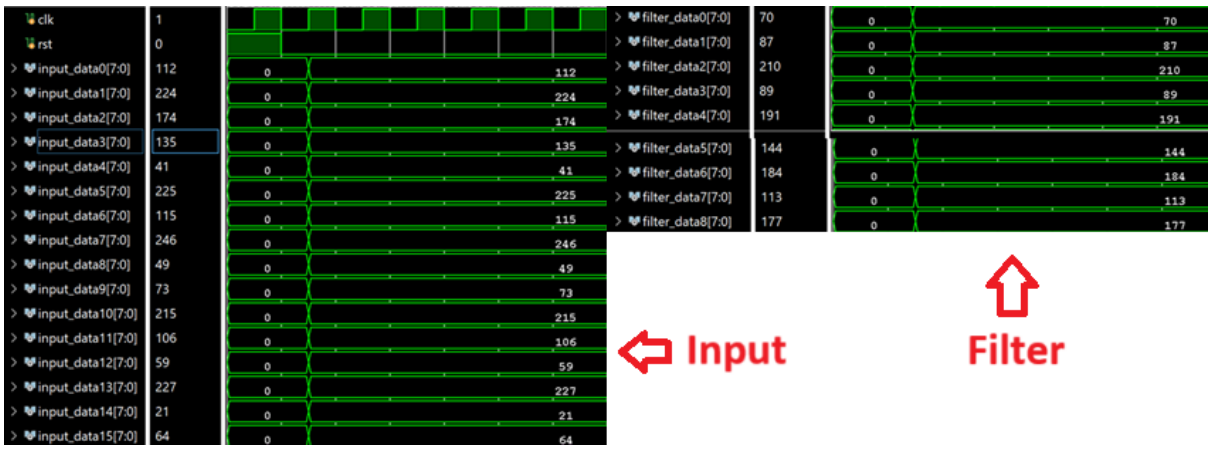

Figure 8. Simulation Result of Memory module

4. Computation

4-1. PE Module

4-1-1. Theory

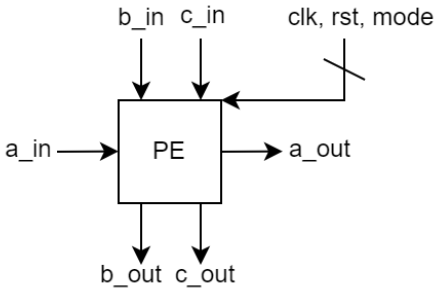

Figure 9. PE Module

PE module is a basic operational unit that iterates predefined operation of certain inputs. There are various types of PE, but the classical PE is defined as basic operational unit with MAC (Multiplication & Accumulation) operator. We will use this MAC operator based PE to design multi-dimensional systolic array. Figure 9 shows the I/O port of PE module, and function of each port is defined as follows.

Port Name	Function
clk	clock
rst	reset
mode	mode selector (0: acc to c_out | 1: c_in to c_out)
a_in	operand to be multiplied with b_in
b_in	operand to be multiplied with a_in
c_in	input port for receiving accumulated value of upper module
a_out	output port for propagating a_in (including 1 clock delay)
b_out	output port for propagating b_in (including 1 clock delay)
c_out	output port for propagating accumulated value or c_in

Table 4. I/O Port of PE Module

To understand the full connection relationships of each port, we first need to theoretically specify how to design MAC operator into logic circuit. MAC operator consists of two operation, multiplication and accumulation (or addition). However, there is no direct method to calculate multiplication in logical basis. Instead, we calculate multiplication by separating equation into addition operator. For example, 62×12 can be calculated as follows.

$$62 \times 2 \times \mathbf{10^0} = 124 \qquad 62 \times 1 \times \mathbf{10^1} = 620$$

$$62 \times 12 = 124 + 620 = 744$$

In this equation, 10^n can be interpreted as shift operator. For example, the value is shift to the left one times when n is 1 (62 -> 620). This concept can be applied to binary format as well. Let's modify this decimal number into binary, and apply the same method with decimal.

$$62 = 00111110 \qquad 12 = 00001100$$

$$00111101 \times 0 \times 2^0 = 00000000 \ (<<0)*0 \qquad 00111101 \times 0 \times 2^1 = 00000000 \ (<<1)*0$$
$$00111101 \times 1 \times 2^2 = 11111000 \ (<<2)*1 \qquad 00111101 \times 1 \times 2^3 = 11110000 \ (<<3)*1$$
$$00111101 \times 0 \times 2^4 = 00000000 \ (<<4)*0 \qquad 00111101 \times 0 \times 2^5 = 00000000 \ (<<5)*0$$
$$00111101 \times 0 \times 2^6 = 00000000 \ (<<6)*0 \qquad 00111101 \times 0 \times 2^7 = 00000000 \ (<<7)*0$$

$$\therefore 00000000 + \cdots + 11111000 + 11110000 + \cdots + 00000000 = 11101000 = 232 \ \text{(8 Bit)}$$

Note that each segment is same as nth shifted version of first operand when nth index bit of second operand is one, and otherwise is zero. Therefore, we have to select two values according to the values of second operand. This can be realized by using 2:1 Multiplexer. By connecting the nth index into selection port, we can select between zero and nth shifted first operand values according to the second operand values. These 8 outputs are connected to 8 Bit Full Adder in a cascaded form to get a total sum of each segment. The following is a schematic diagram that summarizes the upper descriptions.

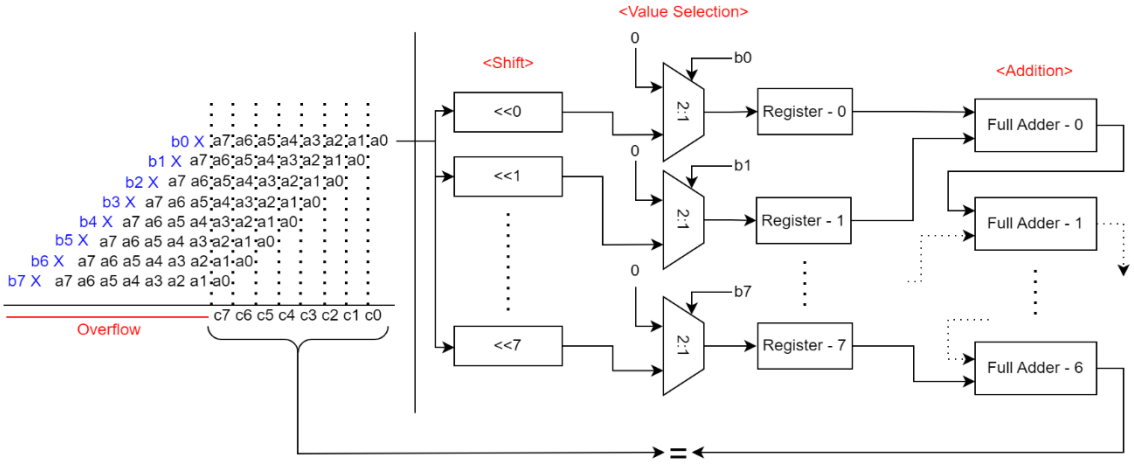

Figure 10. Circuital Expression of Multiplication

This is the full concept of multiplication operation in binary system. Because PE requires MAC operation, we need to design additional system that continuously accumulates the result of multiplication. This system can be designed by using 8 Bit register and Full Adder. By connecting two modules into feedback loops and injecting the multiplication result into Full Adder, accumulated value is continuously updated into the register with its clock edge. By printing out the register value, PE module can be utilized as MAC operator. The following shows the full schematic diagram of PE module that includes MAC operator which we described in words previously.

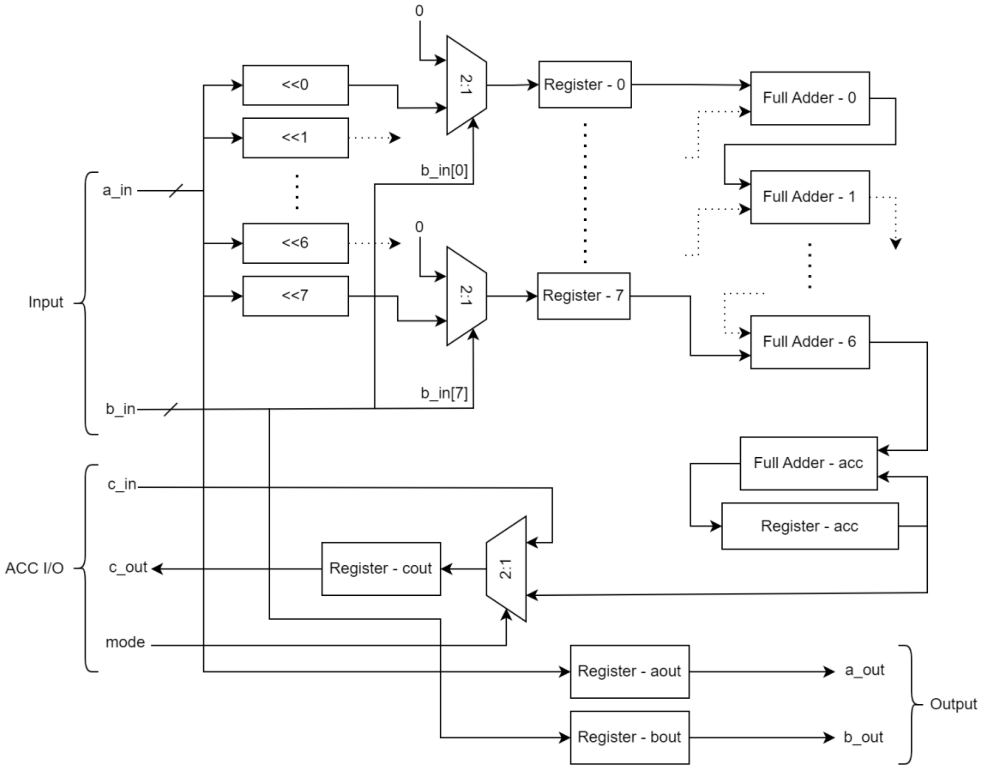

Figure 11. Internal Structure of PE Module

In Figure 11, **a_out** and **b_out** are outputs for another PE Module, and its values are one clock delay of **a_in** and **b_in** respectively. We can also see that 8 Bit register is added between 2:1 Mux and Full Adder. The function of this register is to synchronize with the clock and remove unexpected operations like Hazard.

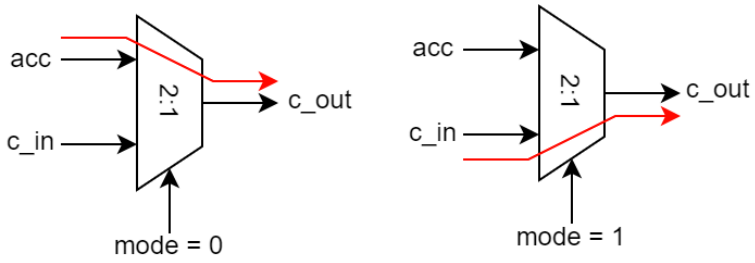

Figure 12. PE Operation Mode

Note that accumulated value (**acc**) is not directly printed out to the external port. Instead, it is connected with 2:1 Mux. In this assignment, there is a restriction that accumulated value should be printed out downward. In a multi-dimensional systolic system, all the values should be extracted from a downward direction. However, there is no appropriate method to extract the accumulated value of center PE when restriction is applied. Therefore, we make a output tunnel for extracting all the accumulated values in each PE. This tunnel is activated when mode port is set to 1. As soon as the tunnel is activated, the accumulated values is shifted downward with its clock edge. By saving those values into external registers, we can access all the accumulated values without violating the restriction. This strategy will be specifically utilized in the next multi-dimensional systolic section.

4-1-2. Implementation & Testbench

Because most part of PE module is described in gate-level (or structural model) and its structure is same as Figure 11, we briefly explain about the code. The following shows the part of module instantiations for calculating multiplication.

```
// Shift Operation
assign shift_0 = {a_in[7], a_in[6], a_in[5], a_in[4], a_in[3], a_in[2], a_in[1], a_in[0]};
assign shift_1 = {a_in[6], a_in[5], a_in[4], a_in[3], a_in[2], a_in[1], a_in[0], 1'b0};
assign shift_2 = {a_in[5], a_in[4], a_in[3], a_in[2], a_in[1], a_in[0], 1'b0, 1'b0};
assign shift_3 = {a_in[4], a_in[3], a_in[2], a_in[1], a_in[0], 1'b0, 1'b0, 1'b0};
assign shift_4 = {a_in[3], a_in[2], a_in[1], a_in[0], 1'b0, 1'b0, 1'b0, 1'b0};
assign shift_5 = {a_in[2], a_in[1], a_in[0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0};
assign shift_6 = {a_in[1], a_in[0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0};
assign shift_7 = {a_in[0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0};

// Value Selector
eight_bit_2_1_mux mux_0 (.a(8'd0), .b(shift_0), .s(b_in[0]), .out(operand_val_0_pre));
eight_bit_2_1_mux mux_1 (.a(8'd0), .b(shift_1), .s(b_in[1]), .out(operand_val_1_pre));
eight_bit_2_1_mux mux_2 (.a(8'd0), .b(shift_2), .s(b_in[2]), .out(operand_val_2_pre));
eight_bit_2_1_mux mux_3 (.a(8'd0), .b(shift_3), .s(b_in[3]), .out(operand_val_3_pre));
eight_bit_2_1_mux mux_4 (.a(8'd0), .b(shift_4), .s(b_in[4]), .out(operand_val_4_pre));
eight_bit_2_1_mux mux_5 (.a(8'd0), .b(shift_5), .s(b_in[5]), .out(operand_val_5_pre));
eight_bit_2_1_mux mux_6 (.a(8'd0), .b(shift_6), .s(b_in[6]), .out(operand_val_6_pre));
eight_bit_2_1_mux mux_7 (.a(8'd0), .b(shift_7), .s(b_in[7]), .out(operand_val_7_pre));

// Multiplication Process
eight_bit_full_adder fa0 (.a(operand_val_1_post), .b(operand_val_0_post), .cin(1'b0), .sum(sum_0), .cout());
eight_bit_full_adder fa1 (.a(operand_val_2_post), .b(sum_0), .cin(1'b0), .sum(sum_1), .cout());
eight_bit_full_adder fa2 (.a(operand_val_3_post), .b(sum_1), .cin(1'b0), .sum(sum_2), .cout());
eight_bit_full_adder fa3 (.a(operand_val_4_post), .b(sum_2), .cin(1'b0), .sum(sum_3), .cout());
eight_bit_full_adder fa4 (.a(operand_val_5_post), .b(sum_3), .cin(1'b0), .sum(sum_4), .cout());
eight_bit_full_adder fa5 (.a(operand_val_6_post), .b(sum_4), .cin(1'b0), .sum(sum_5), .cout());
eight_bit_full_adder fa6 (.a(operand_val_7_post), .b(sum_5), .cin(1'b0), .sum(sum_6), .cout());
```

Figure 13. Instantiation for Multiplication (Part)

Note that shift operation is realized by using wire allocation. Although bit shift operator is already provided in Verilog syntax, we refrain to use primitive operator like ">>" to design a system on a low

level basis. After the shift operation, the shifted value is connected to 2:1 multiplexer, and the multiplexer prints out either shifted value or 0 according to the selection signal. The printed signals from 8 multiplexers are added using 8 Bit Full Adder, and total sum is accumulated in the register whose output port is **acc**. The full code of PE module can be viewed in assignment file or GitHub repository which will be published after submission. In PE module, 8 Bit 2:1 Mux, 8 Bit Register and 8 Bit Full Adder are instantiated, and these modules are designed based on basic gates like AND. We will verify the normal operation of this code by injecting the following sequences into the PE module.

clk	1~2	3	4	5	6	7~11	12~16	17~
rst	1	0	0	0	0	0	0	0
mode	0	0	0	0	0	0	1	0
a_in	0	0	117	18	149	0	0	0
b_in	0	0	55	44	117	0	0	0
c_in	4	4	4	4	4	4	4	4

Table 5. Testbench Sequence for PE Module

The following shows the simulation results based on the upper testbench sequence.

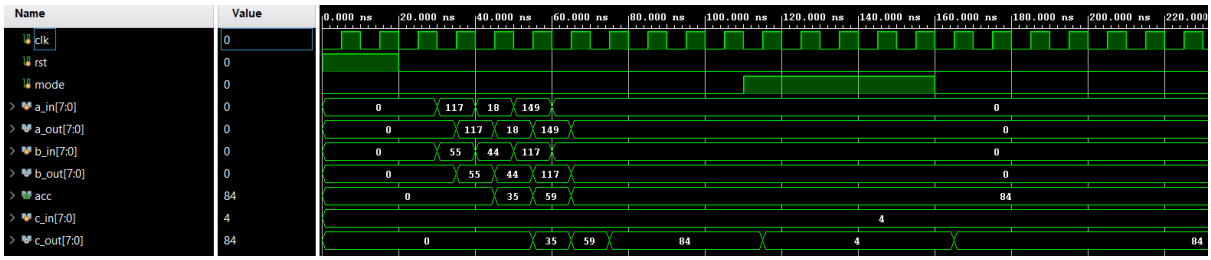

Figure 14. Simulation Result of PE module

In Figure 14, we can check that **a_out** and **b_out** are one clock delay of **a_in** and **b_in**. Note that **acc** is not originally included in I/O port. This port is included after executing the simulation just for checking the accumulated value. The following shows the result of accumulated value according to clock edge.

clock	**multiplication**	**accumulation**
4	$(117 \times 55)_{overflow} = 35$	$0 + 35 = 35$
5	$(18 \times 44)_{overflow} = 24$	$35 + 24 = 59$
6	$(149 \times 117)_{overflow} = 25$	$59 + 25 = 84$

Table 6. MAC Operation Result

Comparing with the Figure 14, we can check that MAC operation is normally operated. After finishing the operation, we need to extract accumulated values including upper module. The following shows the result of **c_out** according to variation of **mode**.

mode	**acc**	**c_in**	**c_out**
0	84	4	84
1	84	4	4
0	84	4	84

Table 7. Mode Operation Result

In Table 7, we can check that upper value (**c_in**) is propagated into **c_out** when mode is set to 1. After changing the mode to 0, **c_out** is returned to the original acc value. This results prove that PE module can be used to propagate upper value to downward when PEs are connected in multi-dimensional system.

We diagnosed the MAC operation results and mode operation of PE module, and the simulation results are compatible with our expectations. Therefore, we can conclude that PE module is well designed in respect of its functionality.

4-2. 3 × 3 Systolic

4-2-1. Theoretical design

A systolic array is a type of computational architecture that consists of processing elements (PEs) in a two-dimensional grid. Unlike general-purpose processors, systolic arrays are designed for high-throughput parallel computation, which makes them especially well-suited for tasks like matrix multiplication, convolution, and signal processing. The data flows through the array in sync with a clock signal. It is similar to the pulsing of a heart, which is where the name "systolic" comes from. This structure enables efficient pipelining at the hardware level while reducing the need for frequent memory access. I/O Ports of 3×3 Systolic Array and function of each port is defined as follows.

Port name	Function
clk_in	clock
rst	reset
i00, i01, i02, i03, i10, i11, i12, i13, i20, i21, i22, i23, i30, i31, i32, i33	input matrix
f00, f01, f02, f10, f11, f12, f20, f21, f22	filter matrix
o00, o01, o10, o11	output matrix

Table 8. I/O Ports of 3×3 Systolic Array

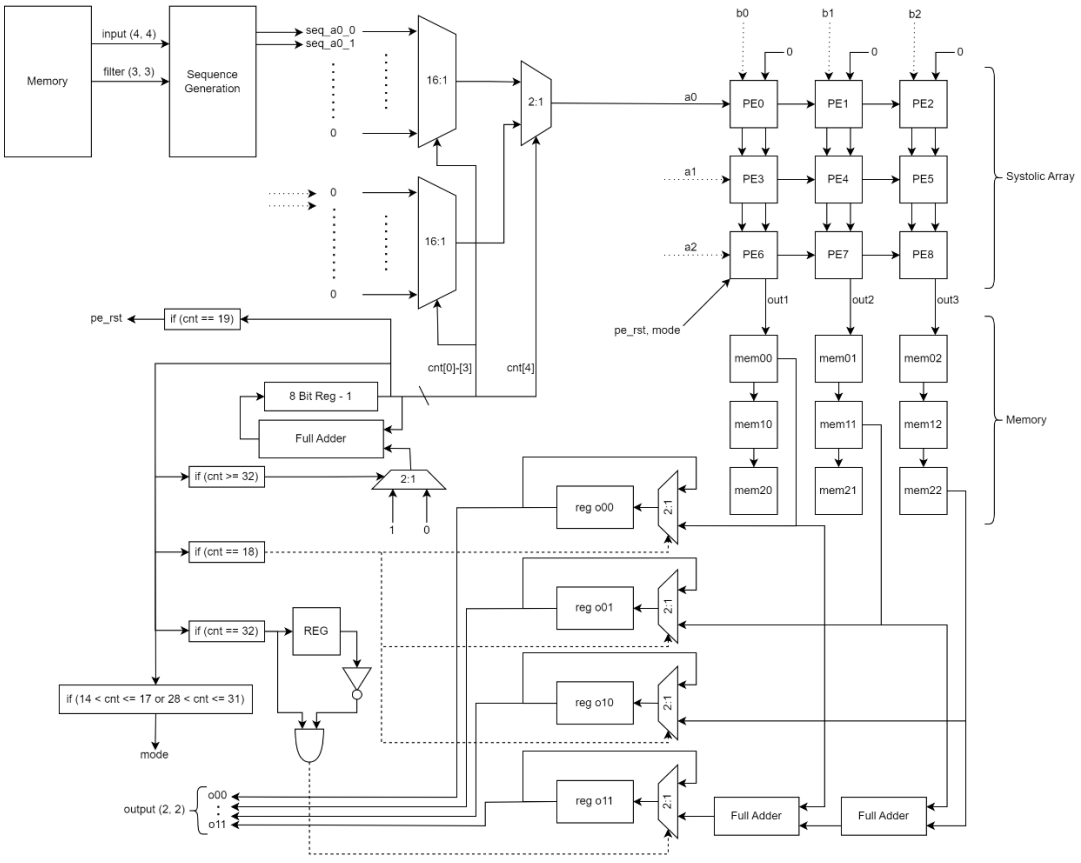

Figure 15. Structure of 3x3 systolic array module

At first, we implemented the systolic array module without considering explicit directional separation. Elements from the input matrix and the filter matrix were mixed and injected into the array from arbitrary directions, and the architecture was designed that each PE directly produced its output without passing the result down to the bottom PE. This structure was successfully implemented, and we verified correct operation by inputting random values and observing the expected outputs.

However, we later decided to completely redesign the structure so that the elements of the input matrix and filter matrix enter the array from distinct directions, and the results of the MAC operations are propagated through the PEs down to the bottommost layer.

To meet the two conditions mentioned above and to implement the systolic array using a gate-level, structural modeling approach as much as possible, we decided to build a 3x3 systolic array based on the structure shown in Figure 15.

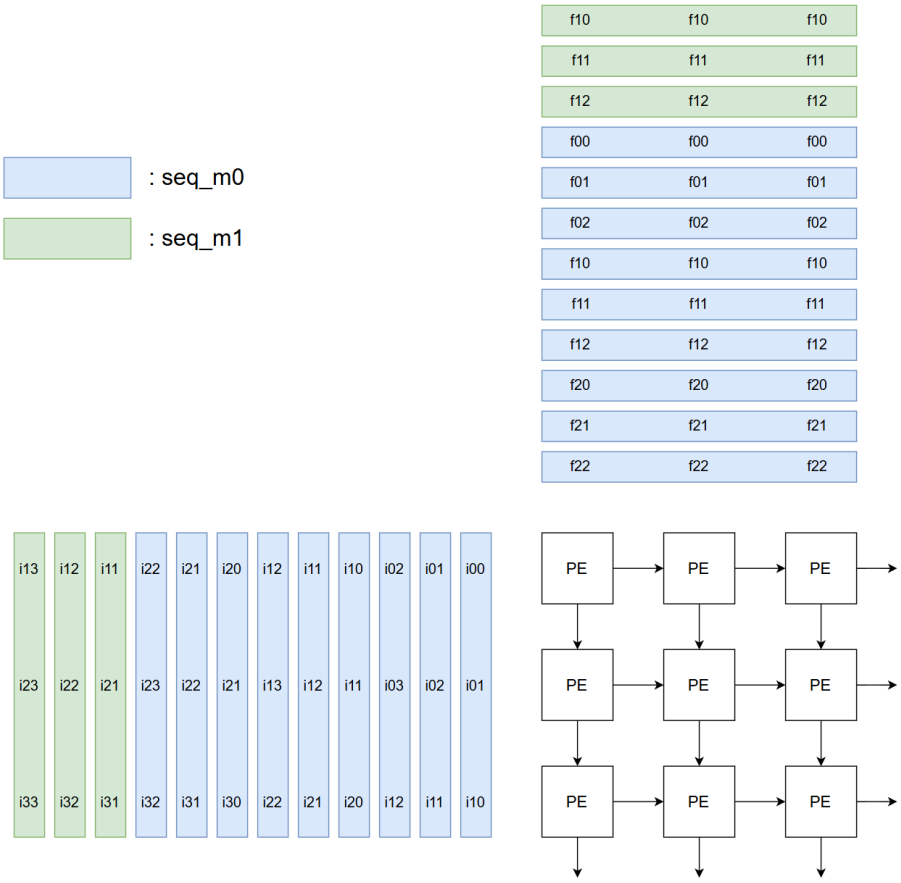

Figure 16. Input sequence of 3×3 systolic array

The main challenges in designing the 3x3 systolic array was that the array provides outputs at only three positions along the bottom row, while we needed to obtain four distinct output values. To resolve this mismatch, we decided to divide the input into two separate sequences: the first input sequence computes o00, o01, and o10, while the second input sequence is used to obtain o11.

First sequence consumes total of 8 clock cycles and propagates in sync with the clock. Once the first input sequence is complete, the mode of each PE changes, and the results of the MAC operations begin propagating downward through the array, one PE per clock cycle, toward the memory modules connected to the bottommost PEs. As a result of this computation and propagation, the values corresponding to o00, o01, and o10 are stored in mem00, mem11, and mem22, respectively.

One significant difficulty we faced was capturing the values from mem00, mem11, and mem22 precisely at the clock cycle when propagation completes. This timing had to be determined precisely

because the second input sequence begins immediately afterward. Without correctly identifying the exact clock cycle at which the computation and propagation from the first input sequence are completed, it would be impossible to obtain accurate results.

This was solved by connecting the outputs of each memory module to a 2:1 mux and configuring the mux to select the current memory output only at the specific clock cycle corresponding to the completion of data propagation.

To minimize the overall computation time, we optimized the second input sequence by distributing the MAC operations required to compute o11 across three PEs. Since o11 is the only output produced from this sequence, each PE performs a portion of the computation, and the intermediate results are then summed using an adder to produce the final value of o11.

4-2-2. Implementation & Test bench

The input matrix and filter matrix used to test implemented 3x3 systolic array are as follows.

$$\text{Input matrix: } \begin{pmatrix} 8 & 3 & 9 & 1 \\ 7 & 7 & 2 & 8 \\ 5 & 6 & 3 & 1 \\ 4 & 9 & 2 & 6 \end{pmatrix}$$

$$\text{Filter: } \begin{pmatrix} 1 & 5 & 8 \\ 6 & 0 & 7 \\ 3 & 1 & 2 \end{pmatrix}$$

$$\text{Convolution result : } \begin{pmatrix} 180 & 179 \\ 159 & 176 \end{pmatrix}$$

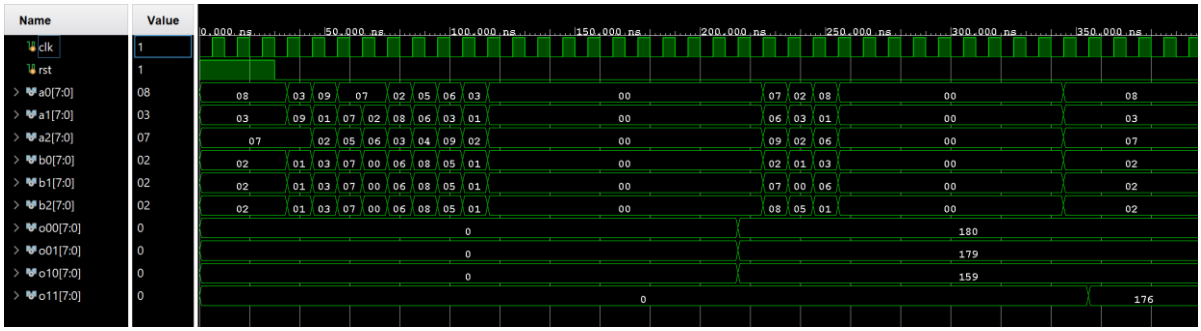

Figure 17. Simulation result of 3×3 systolic array

The elements of the input matrix and filter start entering the systolic array, after rst becomes 0. Following the first input sequence, the computation and propagation processes produced the values 180, 179, and 159 for o00, o01, and o10, respectively. After the second input sequence was completed, we observed that the value of o11, which was computed and propagated through the array, correctly resulted in 176.

4-3. 2 × 2 Systolic

4-3-1. Theoretical design

The I/O ports of the 2×2 systolic array and the function of each port are identical to those of the 3×3 systolic array. The 2×2 systolic array was implemented using a structure similar to that of the 3×3 array. The figure below shows the architecture of the 2×2 systolic array we designed.

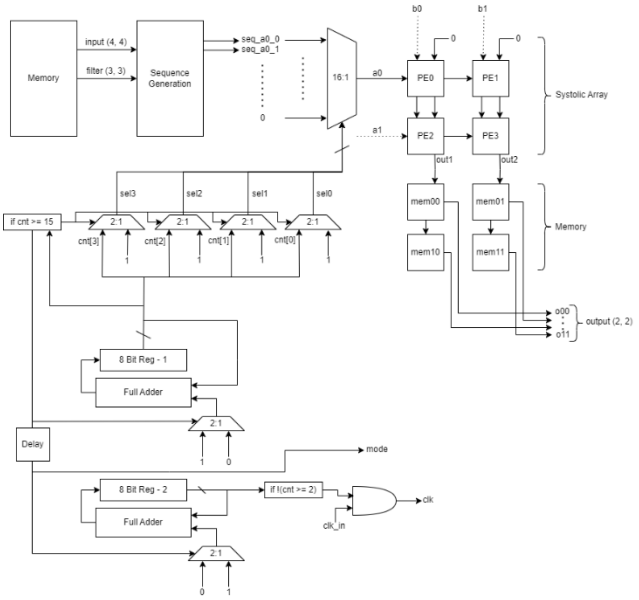

Figure 18. Structure of 3x3 systolic array module

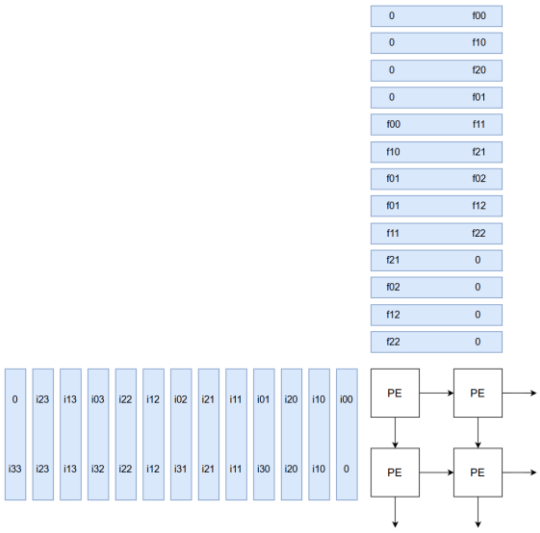

Figure 19. Input sequence of 2×2 systolic array

A key feature of the 2×2 systolic array that we implemented is that the entire convolution operation is completed with a single input sequence. The elements of the input matrix and filter matrix enter the array from different directions and shift one PE per clk cycle. Each PE receives the inputs and performs MAC operations accordingly.

Once the input sequence is completed, the mode of each PE changes, and the results of the MAC operations are propagated downward. These values reach the memory modules connected to the bottom of the systolic array, where the final results o00, o01, o10, and o11 are stored.

4-3-2. Implementation & Test bench

The test pattern for the 2×2 systolic array is based on the same input matrix and filter matrix as those used for the 3×3 systolic array. The convolution results for a00, a01, a10, and a11 were verified to be accurate, producing the expected values of 180, 179, 159, and 176, respectively.

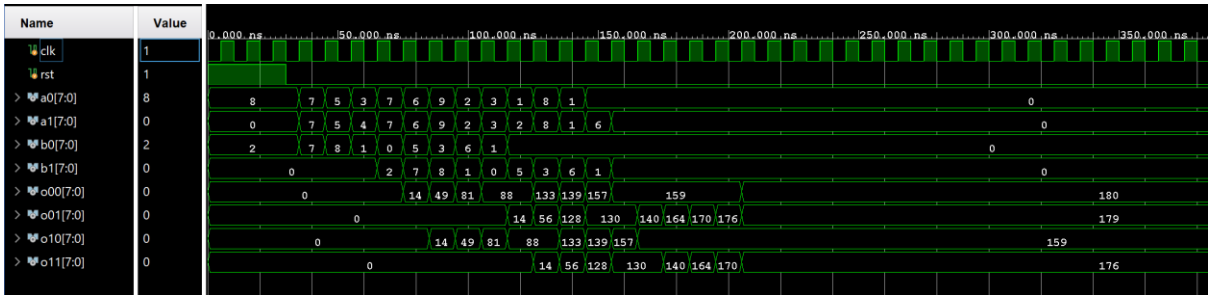

Figure 20. Simulation result of 2×2 systolic array

4-4. A Single PE

4-4-1. Theoretical design

Unlike a systolic array, a single-PE convolution calculator performs convolution between the input matrix and the filter using only one PE so it processes each element sequentially rather than computing in parallel. The I/O ports of a single PE and the function of each port are identical to those of the 2×2 and 3×3 systolic array. The following is the structure of the single-PE convolution calculator we designed.

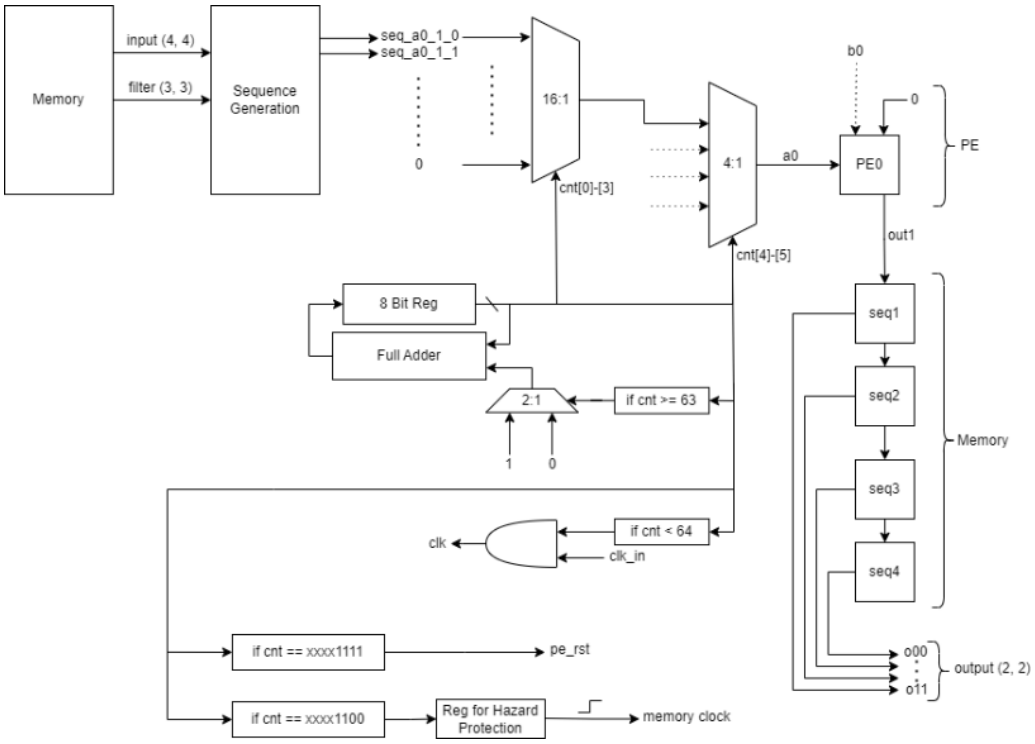

Figure 21. Structure of a single PE convolution calculator

4-4-2. Implementation & Test bench

The test pattern for single PE is same as the test pattern for 2×2 and the 3×3 systolic array. The convolution results for a00, a01, a10, and a11 were 180, 179, 159, and 176 respectively. This result produced by a single PE is consistent with the results obtained from 2×2 and 3×3 systolic array implementations.

As will be discussed in detail later, we can observe that the convolution results of a single PE are produced significantly more slowly compared to the parallel computation of the systolic array.

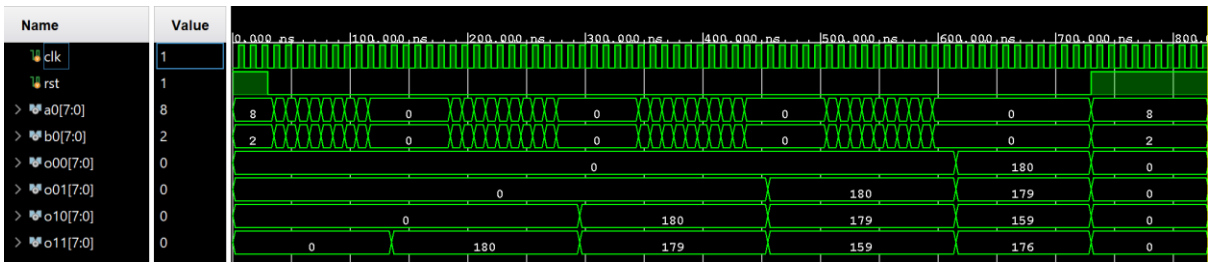

Figure 22. Simulation result of a single PE convolution calculator

5. Display

5-1. Theoretical approach

The display module functions to visualize the outputs from the computation module on a 7-segment display. It displays four output values from the 3x3 systolic array and four from the 2x2 systolic array, each shown with a reasonable time gap to ensure readability and proper timing. Since the display module can also be used for debugging purposes to verify output values, it must be highly reliable and ensure error-free operation. The following table lists the input and output ports of the display module.

Port Name	Function
input port	
clk	clock
resetn	reset (activated on the falling edge)
c9_11	The (1,1) output value of the 3x3 systolic array (8 bits)
C9_12	The (1,2) output value of the 3x3 systolic array (8 bits)
c9_21	The (2,1) output value of the 3x3 systolic array (8 bits)
c9_22	The (2,2) output value of the 3x3 systolic array (8 bits)
c4_11	The (1,1) output value of the 2x2 systolic array (8 bits)
c4_12	The (1,2) output value of the 2x2 systolic array (8 bits)
c4_21	The (2,1) output value of the 2x2 systolic array (8 bits)
c4_22	The (2,2) output value of the 2x2 systolic array (8 bits)
output port	
digit	selected digit position (3 bits)
seg_data	the data of the 7-segment (8 bits)

Table 9. The input and output ports of the display module

The overall data flow of the display module is as follows. First, the data received from the systolic arrays are in 8-bit binary format. These binary values are converted into decimal format using a BCD converter. The resulting decimal digits are then passed to a decoder, which translates them into a format suitable for 7-segment display. Finally, the decoded 7-segment data are displayed according to the current state, and the corresponding values appear on the 7-segment display. The diagram below illustrates this data flow.

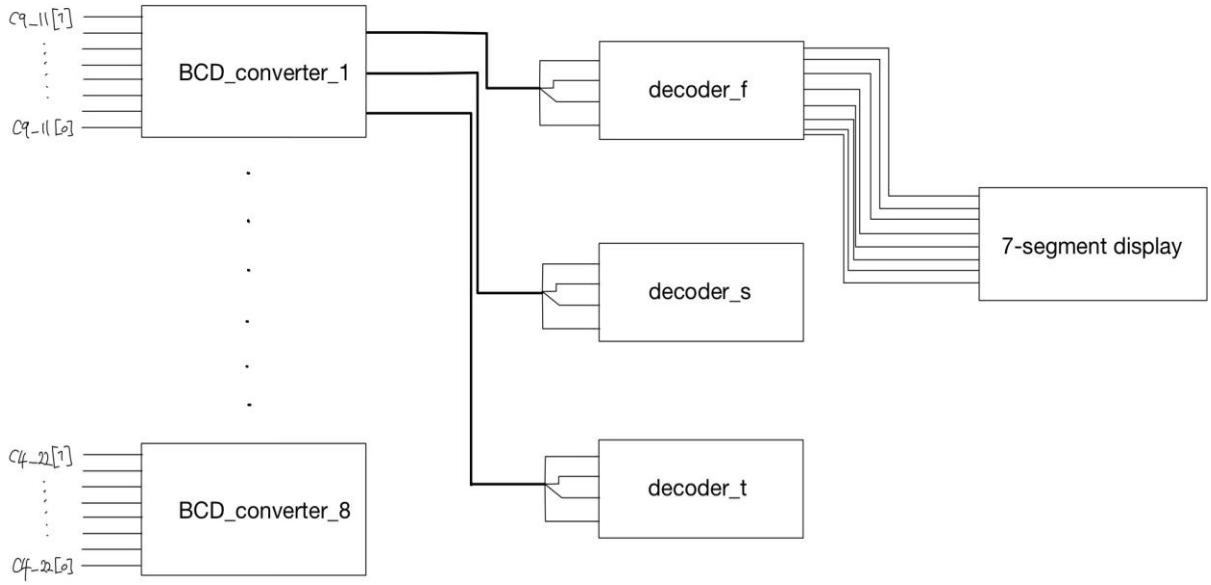

Figure 23. The brief diagram of whole display module

5-2. Bi-to-BCD converter

The implementation of a BCD converter requires an understanding of the Double dabble algorithm, which is a method for converting binary numbers to their decimal representation. The Double dabble algorithm is a simple method for converting binary numbers to BCD by shifting the binary value bit by bit. During each shift, if any BCD digit becomes 5 or greater, 3 is added to that digit. As an example, the binary number representing 146 will be converted to its decimal (BCD) form using the Double dabble algorithm below.

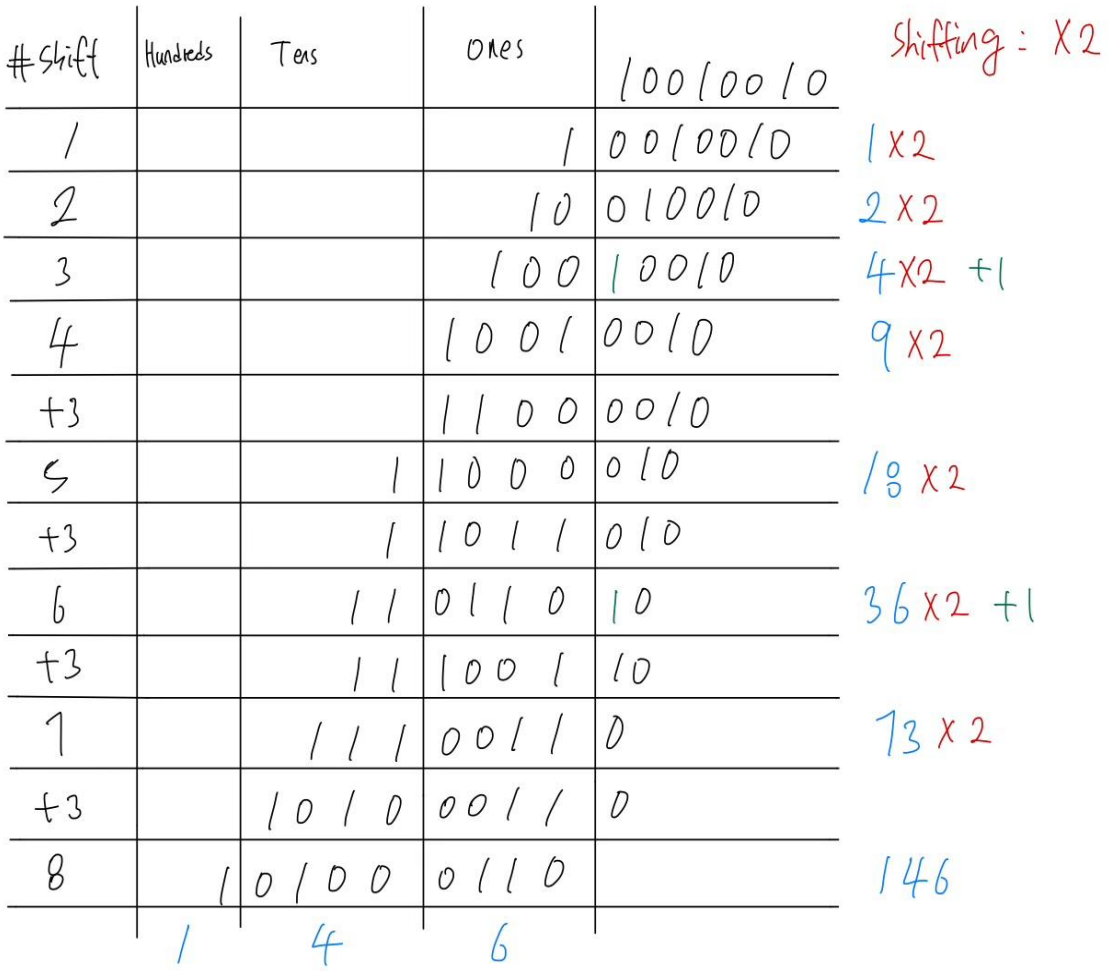

Figure 24. Diagram converting the 8-bit binary number 146 to BCD with Double dabble algorithm

The reason for adding 3 when a digit is 5 or greater is that, up to 4, doubling the value through shifting does not cause a carry to the next decimal place. However, starting from 5, doubling the value would result in a carry, so adding 3 ensures that the digit remains valid in BCD representation. The number of shifts is the Double dabble algorithm must match the number of bits in the input. In this case, since the input is an 8-bit binary number, eight shifts are required. After the final (eight) shift, no further shifting occurs, meaning that no carry-over to the next digit can happen. Therefore, the addition of 3 is not necessary after the last shift.

To convert an 8-bit binary number to BCD using the Double dabble algorithm, a module named add_3_if_ge5 was implemented. This module adds 3 to the input value if it is greater than or equal to 5. By properly connecting these modules in the correct order, it is possible to convert a binary number to BCD without performing actual bit shifting. The diagram below shows the completed circuit for the BCD converter.

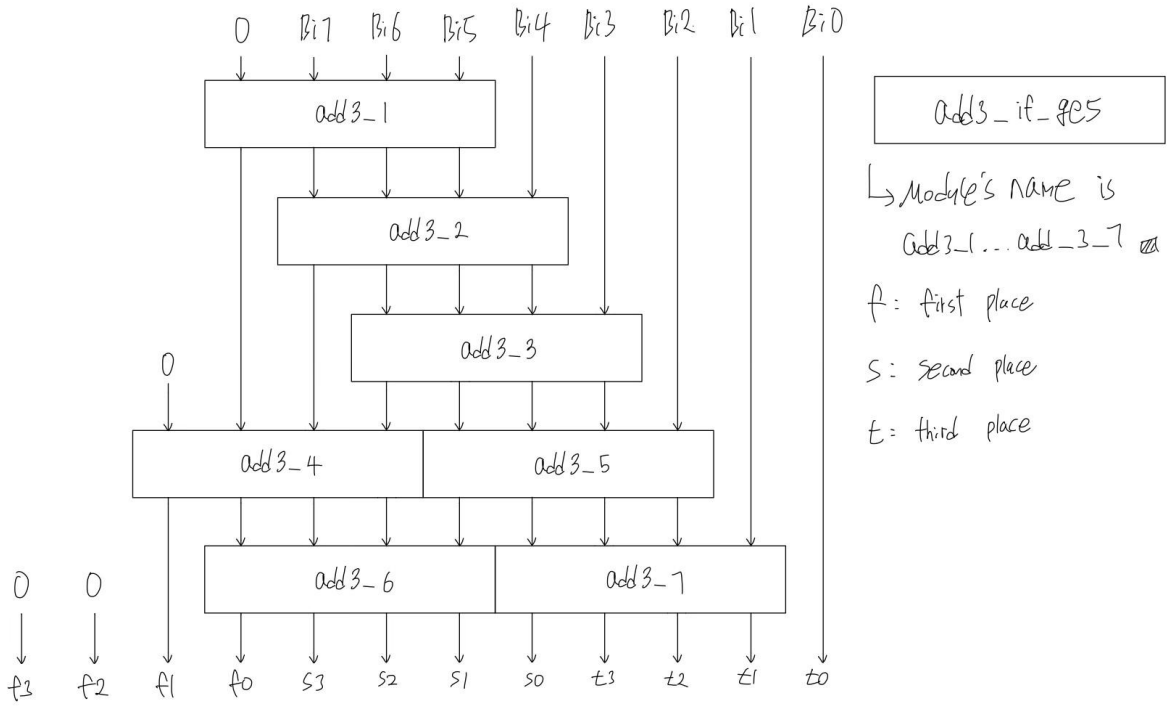

Figure 25. Structural modeled circuit diagram of the BCD converter

The add3_if_ge5 module included in the diagram should be implemented using an adder and a multiplexer. The multiplexer should be configured with two 4-bit inputs: 0000 and 0011, and its selector must be designed to detect whether the input value is greater than or equal to 5. The diagram below shows the internal architecture or the add3_if_ge5 module.

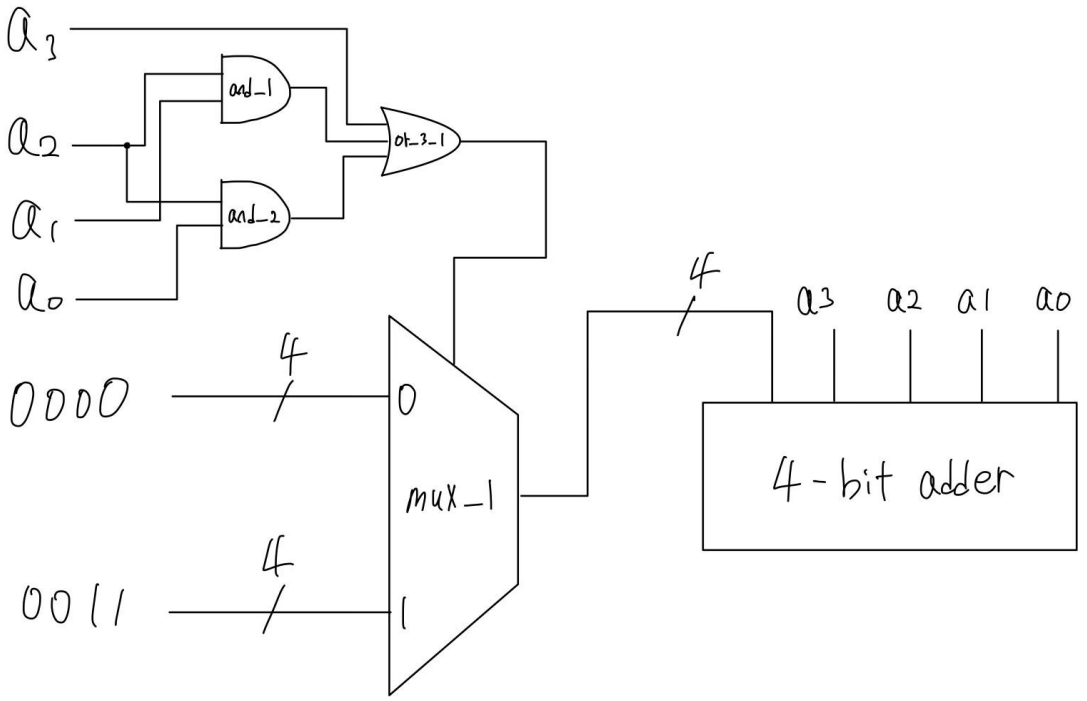

Figure 26. The circuit diagram of the add3_if_ge5 module

The selector logic for the multiplexer can be easily derived using a Karnaugh map. The following Karnaugh map shows the case where the output is 1 only when the input value is greater than or equal to 5.

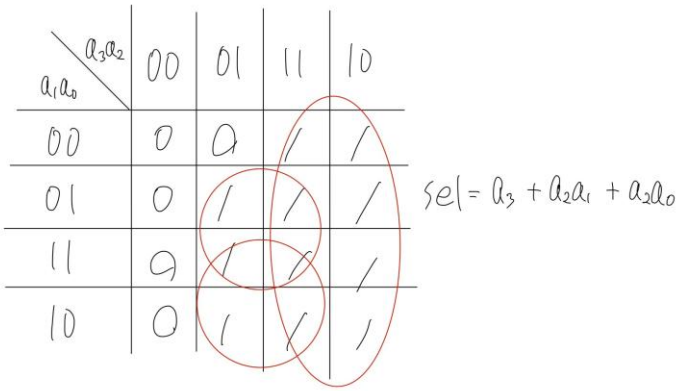

Figure 27. The Karnaugh map of the selector logic

Finally, the 4-bit adder was implemented using one 1-bit half adder, two 1-bit full adders, and one custom adder that only takes a carry input. The following diagram shows the structural design of custom 4-bit adder and the Karnaugh map for the carry-input only adder(Adder_Cin).

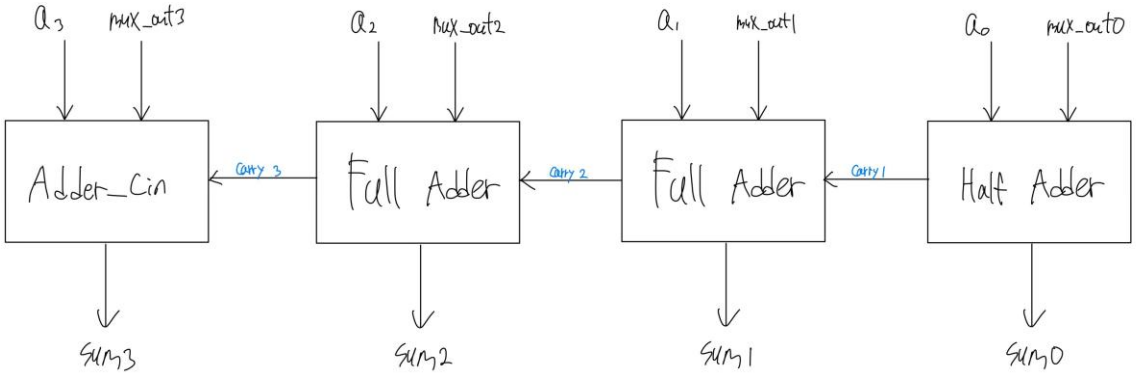

Figure 28. The circuit diagram of the custom 4-bit adder

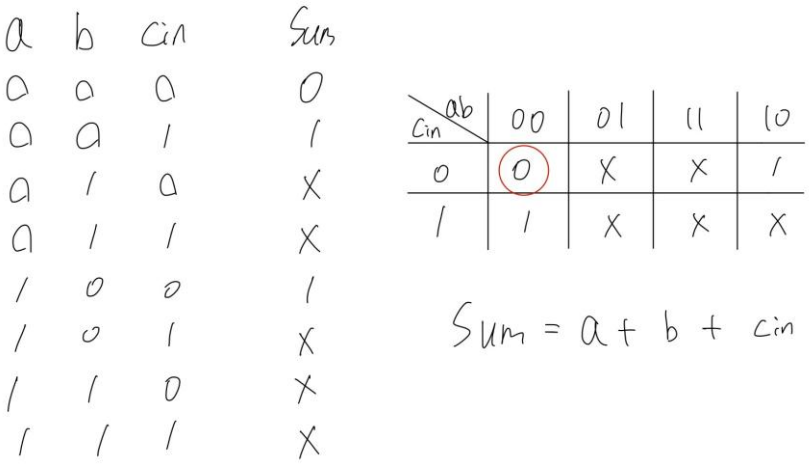

Figure 29. The Karnaugh map of the Adder_Cin module

Unlike a typical 1-bit adder, the Adder_Cin module only needs to add either 0000 or 0011. Therefore, all cases where b is 1 can be treated as don't care conditions. Additionally, the case where both a and Cin are 1 does not occur due to the structure of the Double dabble algorithm, and thus it can also be treated as a don't care condition. As a result, the circuit can be simplified significantly.

5-3. Decoder

The docder module converts a 4-bit BCD value into seg_data to be displayed on a 7-segment display. The possible inputs range from 0000(representing decimal 0) to 100 (representing decimal 9). To determine the correct output patterns for each segment, eight Karnaugh maps were constructed: one for each segment of the display. The following are the Karnaugh maps constructed for each segment.

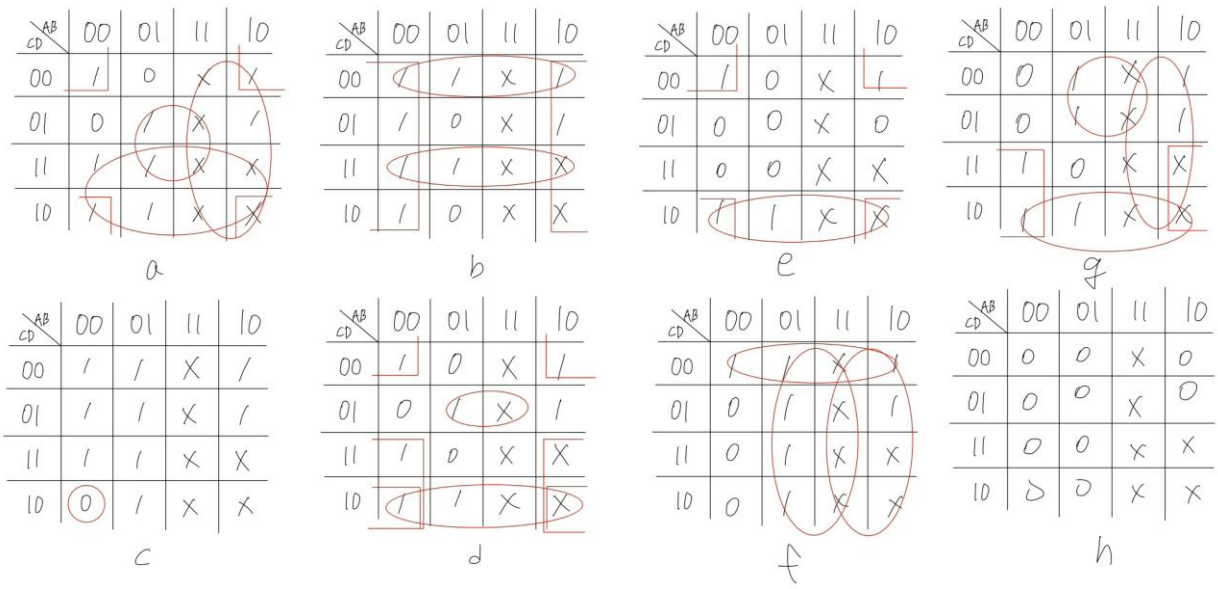

Figure 30. The Karnaugh map of the decoder

5-4. clock_divider

The clock_divider module converts the default 100MHz clock provided by the FPGA kit into a desired frequency by adjusting parameter variables. In the display module, two instances of the clock_divider are used: one to generate a 1Hz clock for the FSM, and another to generate a 1000Hz clock to make multiple digits of the 7-segment display appear to be displayed at the same time.

5-5. FSM

The FSM in the display module consists of a total of 9 states: an initial state entered upon receiving a reset signal, and a sequence of states that pass the outputs from the systolic array to the decoder module in order. All states return to the initial state when the resetn signal is asserted. For the remaining states, transitions occur with each rising edge of the clk_1hz signal. In the final state, S08, the FSM remains in that state even when additional clk_1hz pulses are received. The following diagram shows the state transition of the FSM.

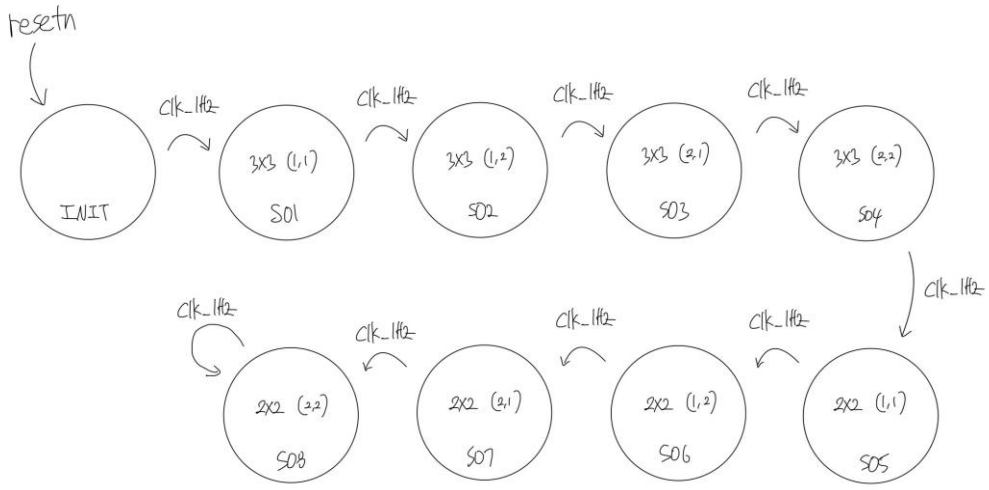

Figure 31. The state diagram of the FSM

5-6 Sending seg_data at appropriate timing

An integer i was defined to cyclically iterate through 0, 1, 2, 3, 1, 2, 3… to control digit placement. When the resetn signal is asserted, i is reset to 0. The design ensures that 3-digit natural number outputs are displayed on the appropriate digit positions. When i is 1, the hundreds digit is displayed; when i is 2, the tens digit is shown; and when i is 3, the units digit is sent to the 7-segment display.

5-7 Testbench

In the testbench, the periods of clk_1hz and clk_1000hz were adjusted to improve visibility. The eight input values from the systolic array were assumed to be 1, 2, 4 … , 64, and 128 for simulation purposes.

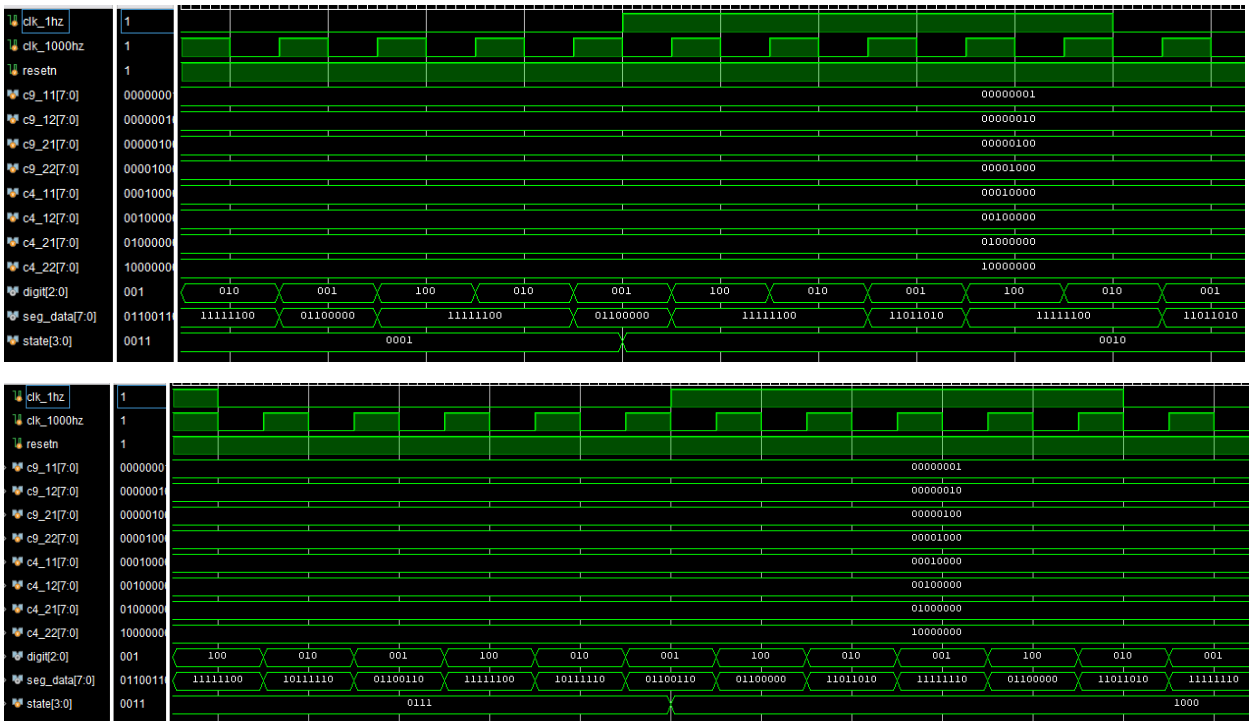

Figure 32. Testbench simulation at the moment of transition from S01 state to S02 state, and from S07 state to S08 state

As shown in the following testbench simulation, the seg_data outputs are updated according to each state. In state S01, the values 0, 0, 1 are sequentially sent as seg_data; in S02, the values 0, 0, 2 are sent. In S07, the values 0, 6, 4 are transmitted, and in S08, the values 1, 2, 8 are output. Each value is displayed at the correct timing according to the FSM transitions.

6. TOP

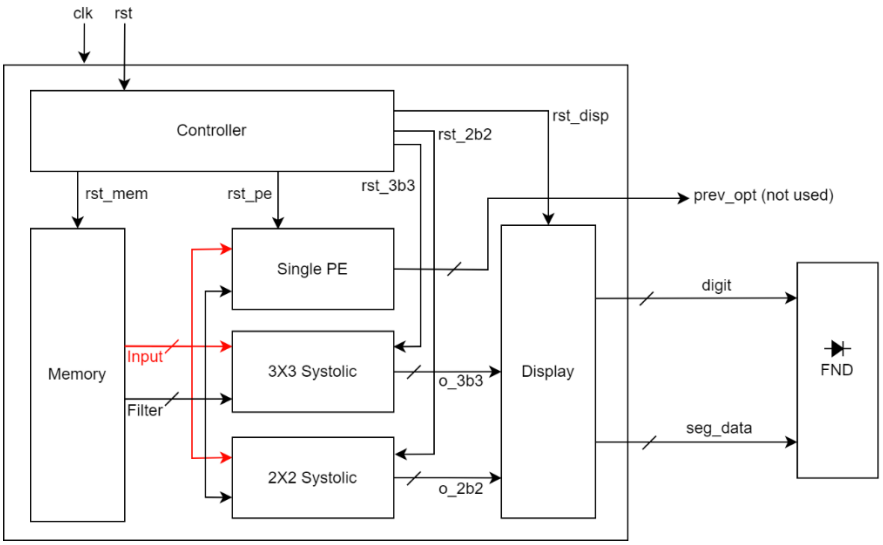

Figure 33. Internal Structure of Top Module

TOP is a packaged module that integrates all the modules which we introduced so far. Figure 33 shows the internal structure of TOP module, and the operation flow is as follows. When **clk** is set as 100 MHz and **rst** is activated, controller module is set to S0 which is its initial state. This state also sets **rst_mem**, **rst_pe**, **rst_3b3**, **rst_2b2**, **rst_disp** to reset value (0 or 1) which depends on destination module. Note that reset of display module is zero (or negedge) and others are one (or posedge). After the initialization, controller state is changed to S1. In S1 state, memory module prints out the input and filter values which are injected in advance. For checking behavioral simulation afterward, suppose memory values are set as follows.

$$I = \begin{bmatrix} 145 & 37 & 19 & 86 \\ 190 & 227 & 126 & 234 \\ 232 & 117 & 92 & 232 \\ 128 & 105 & 153 & 204 \end{bmatrix} \quad F = \begin{bmatrix} 18 & 116 & 231 \\ 139 & 13 & 188 \\ 51 & 132 & 142 \end{bmatrix} \quad O = \begin{bmatrix} 152 & 47 \\ 165 & 123 \end{bmatrix}$$

This value will be considered after introducing all the operation flow of TOP module. These values are transmitted in each computation module, and computation modules wait until reset is deactivated. When state is changed to S2, Single PE module is activated to calculate convolution of input and filter matrix. Controller module waits until Single PE module finishes the calculation. Note that output of Single PE module is not used in this system. Vivado automatically removes unused ports when generating a bitstream, and this can cause malfunctions like timing violation. To solve this problem, we arbitrarily allocate outputs of Single PE to external ports (**prev_opt** in Figure 33) which are not used. By connecting Single PE to physical ports, we can prevent Vivado to optimize the Single PE module. Actually, there are lots of issues related with optimization of Vivado when integrating the modules into one package, and we almost solve those problems by using *(* keep = "true" *)* and *(* dont_touch = "true" *)* options which prohibit optimization. The following shows the utilization of these options in TOP module.

```
(* keep = "true" *) wire [7:0]
o00_2b2, o01_2b2,
o10_2b2, o11_2b2;

// This code is included for preventing optimization(auto remove) of PE module which is not used
wire [7:0] prev_opt;
assign prev_opt = o00_pe + o01_pe + o10_pe + o11_pe;

// Controller
(* dont_touch = "true" *) controller control (.clk(clk), .rst(rst),
                        .rst_mem(rst_mem), .rst_pe(rst_pe),
                        .rst_3b3(rst_3b3), .rst_2b2(rst_2b2),
                        .rst_disp(rst_disp));
```

Figure 34. Verilog Code for TOP Module (Part)

(dont_touch = "true" *)* option is applied to all the instantiated modules in TOP. By doing so, each module can be conserved as its original form which is already diagnosed in separate development phase. This will be specifically covered in later conclusion section. Let's move on to the next state. In S3 state, 3 By 3 Systolic module initiates the convolution of input and filter matrix. After finishing the calculation, controller changes the state to S4. Like in S3 state, 2 By 2 Systolic module executes the same actions. After all the calculations are completed, state is changed to S5 which is related with activation of display module. The display module receives 8 inputs which is originated from 2 By 2 and 3 By 3 Modules, and generates signals to control FND in desired format. These signals are wired with **digit** and **seg_data**, and the following shows the generated signal in behavioral simulation.

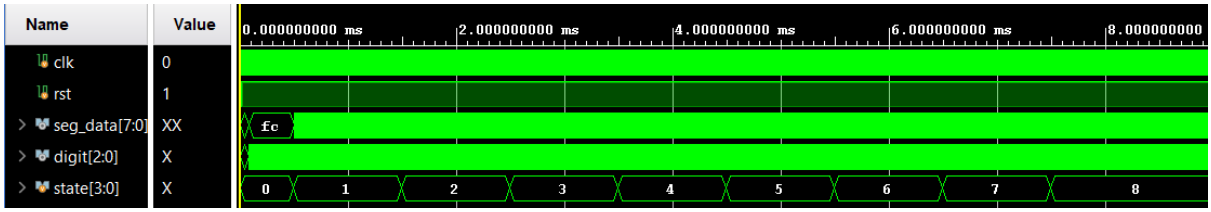

Figure 35. Simulation Result

Because frequencies in display module is too small to cover in simulation, we arbitrarily change the frequency division when executing a simulation. The following shows the **digit** and **seg_data** in each state of display module.

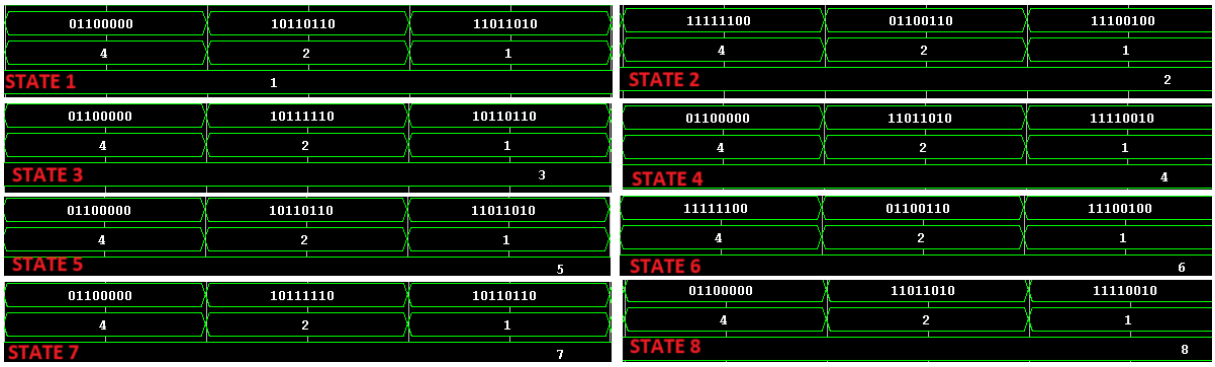

Figure 36. Output Values in Each State

To verify the normal operation of TOP module, we need to check whether output results are same as 152, 47, 165, 123 which are the convolution results of input and filter matrix. Because binary numbers in Figure 36 are for FND operation, we need to convert them to decimal numbers to compare with desired values. The conversion table is specified in Display module section, and we can convert the binary values to its original decimal values as follows.

State	3rd	2nd	1st	Desired Value
1	01100000 = 1	10110110 = 5	11011010 = 2	152
2	11111100 = 0	01100110 = 4	11100100 = 7	47
3	01100000 = 1	10111110 = 6	10110110 = 5	165

4	01100000 = 1	11011010 = 2	11110010 = 3	123
5	01100000 = 1	10110110 = 5	11011010 = 2	152
6	11111100 = 0	01100110 = 4	11100100 = 7	47
7	01100000 = 1	10111110 = 6	10110110 = 5	165
8	01100000 = 1	11011010 = 2	11110010 = 3	123

Table 10. Decoding Results

We can check that output values are same as desired values. Therefore, we can conclude that TOP module is operated as we intended in simulation phase. In the next section, we generate bitstream and program the codes in real FPGA board. This will be the final verification of the TOP module.

7. FPGA Implementation

$$I = \begin{bmatrix} 145 & 37 & 19 & 86 \\ 190 & 227 & 126 & 234 \\ 232 & 117 & 92 & 232 \\ 128 & 105 & 153 & 204 \end{bmatrix} \quad F = \begin{bmatrix} 18 & 116 & 231 \\ 139 & 13 & 188 \\ 51 & 132 & 142 \end{bmatrix} \quad O = \begin{bmatrix} 152 & 47 \\ 165 & 123 \end{bmatrix}$$

The following are the results obtained from operating the FPGA kit after loading the above input matrix and filter matrix into the memory module.

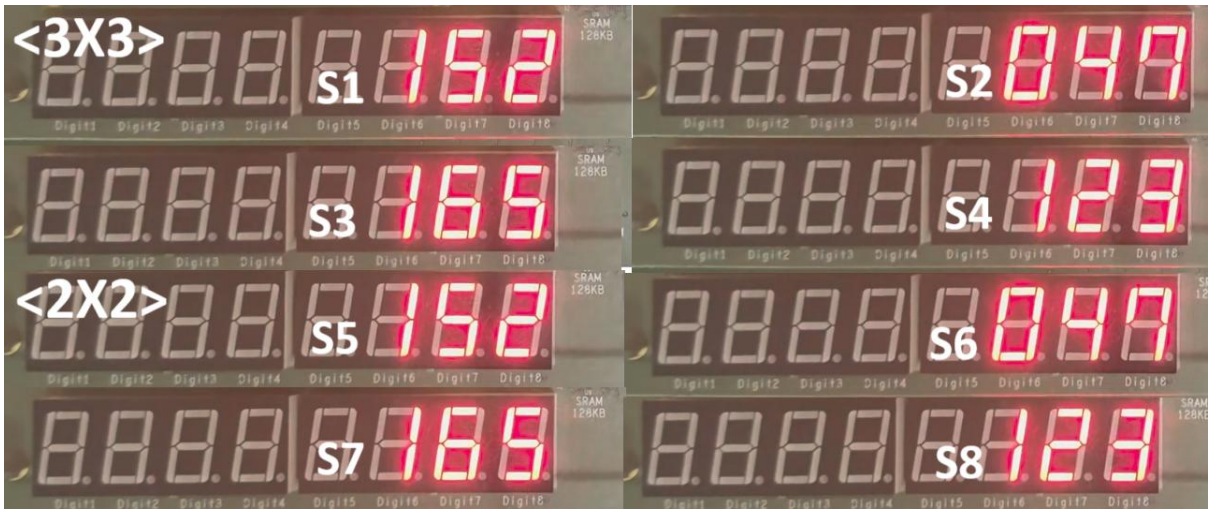

Figure 37. FPGA Implementation Results

In Figure 37, We can check that output results are same as TOP module simulation results in previous section. Therefore, we can conclude that FPGA implementation is compatible with our expectation.

8. Comparison

To compare a performance between Single PE and Systolic Array (including 2 By 2 and 3 By 3), we made a testbench (**tb_comparison.v**) that activates three modules at the same time. In the testbench, clock is generated with 100 [MHz] and reset is set to 1 at initialization. After 20 [ns], reset value is configured as 0, and three modules calculate the convolution of input and filter matrix simultaneously. By measuring the end point of calculation and reset time, we can measure the time interval of calculation in each module. The following shows the behavioral simulation results of Single PE, 2 By 2 Systolic and 3 By 3 Systolic modules based on the **tb_comparison.v**.

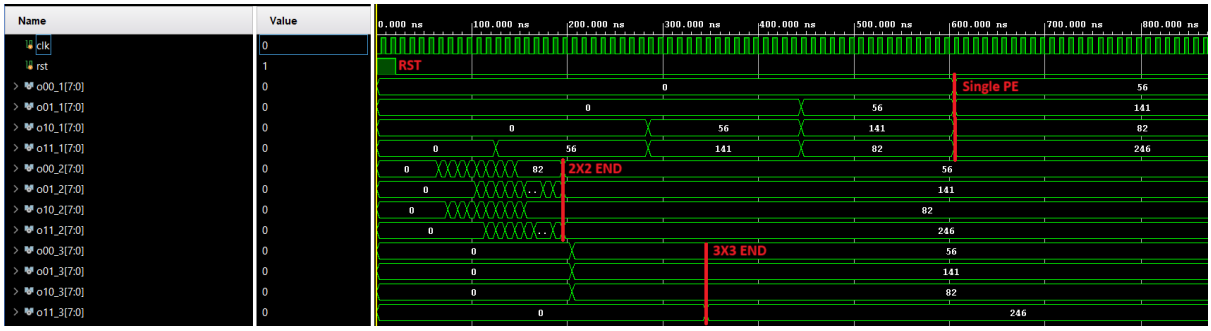

Figure 38. Simulation Result

In Figure 38, we can see that 2 By 2 Systolic module finishes the calculation first and Single PE finishes last. Let's measure the calculation time in each module using the cursor. The following shows the time difference between start time and end time of calculation in each module.

Module	End Time [ns]	Reset Time [ns]	Diff [ns]	x [Single PE]
Single PE	605	20	585	1
2X2	195	20	175	**3.3429**
3X3	345	20	325	**1.8**

Table 11. Calculation Time

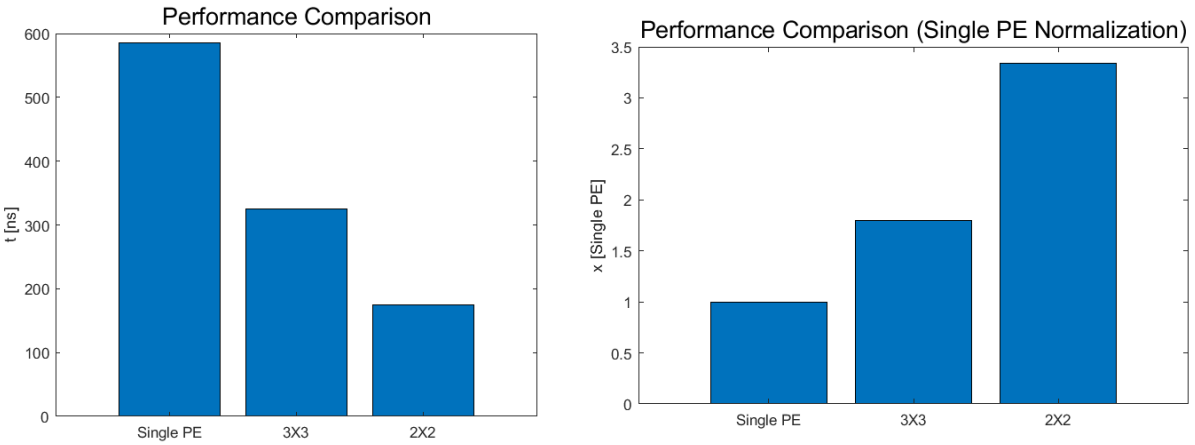

Figure 39. Performance Comparison between Single PE and Systolic
(L: Time Basis, R: Single PE Normalization)

As we can see in the figure, 2 By 2 Systolic has dramatic performance enhancement in calculation time compared with Single PE (3.3429 times enhancement). On the other hand, 3 By 3 Systolic has lower enhancement compared with 2 By 2 (1.8 times enhancement). It seems like this result is contradictory to our intuition. We usually tend to think that more processors perform better. However, it is always not the case. We will deeply cover this problem in next discussion section.

9. Discussion & Conclusion

9.1 Balancing between Computation Loads and Processors

In previous section, we checked that 2 By 2 Systolic has higher performance increment than 3 By 3 Systolic module. We usually think that 3 By 3 Systolic would be fast because it has lots of computation power. However, we need to observe the utilization rate of PE, not the number of PE. In 2X2 Systolic, all the PEs are utilized to calculate each element of output matrix. That means, accumulated values of four PEs correspond with each output (**o00**, **o01**, **o10**, **o11**). Therefore, utilization rate of PE is 100% in 2X2. On the other hand, 3X3 has 9 PEs and only 3 PEs are used to calculate the convolution. Actually,

we previously designed the 3X3 Systolic that utilizes 4 PEs at the same time. However, because of the restriction that input and filter values should be injected in certain way (input values to the right side, filter values to downward), we deprecated that 3X3 Systolic module (version 1) and made a new module (version 2) which uses 3 PEs only. Because 3 PEs are utilized, we need to rotate the calculation two times to derive the 4 output values. This rotation process requires additional system like PE reset. Furthermore, 3X3 system requires more clock to calculate, because the number of register in dataflow path from input to output increases compared with 2X2 Systolic. Because of these reasons, 3X3 has lower performance enhancement than 2X2. To fully utilize 9 PEs in 3X3 Systolic module, we need to use that module when 9 outputs are required. In this report, the size of output matrix is limited to 2 by 2. If we calculate something that requires 9 outputs, 3X3 Systolic will perform better. This is the main design problem that we have to consider. In engineering perspective, finding a optimal point between computation loads (size of input and filter matrix) and processors (PE) is important in respect of resource efficiency and cost. By using appropriate number of PEs in certain calculation, we can maximize the efficiency and lower the design costs. Considering that principle in mind, we can conclude that 2X2 Systolic module is appropriate design for calculating convolution of 4 by 4 input and 3 by 3 filter matrix regarding costs and efficiency.

9.2 Optimization Problem

Vivado automatically optimizes the structure of the code when implementation or bit stream generation is executed. This optimization process removes unused ports and integrates the similar circuits. By doing so, optimal results like low usage of power can be achieved. However, optimization process sometimes distorts the functionality of the original module, and we struggled to solve the problem related to this optimization process. There was a continuous issue regarding first calculation result of 2X2 Systolic module. We thought that this problem is occurred because of timing distortion which is caused by optimization. The specific explanation is as follows. In controller module, flow control using FSM is executed by waiting certain time which is a little bit longer than calculation time of each module. For example, **B** module should be operated after **A** module completes the operation. To achieve this sequence control, controller module waits until **A** module finishes the operation. By measuring the time interval of calculation, we can control the operation flow like **B** after **A**. We already measured the operation time of each module by using post-implementation simulation, and this measured values were utilized in controller module. Therefore, we can expect that each module is operated in predefined order. However, if this calculation time interval is changed because of the optimization process, the results can be distorted. In that in mind, we suggested some hypotheses that can solve the timing problem in optimization process.

<1> The output of Single PE module is included, but it is not used. Single PE module can be removed by optimization process, and it would cause severe timing issue in controller.

<2> When integrating the modules into one package, optimization strategy can be changed compared with single module. In certain optimization situation, different modules sometimes share same registers and circuits to reduce total resources. If timing is not optimized well, this would cause problem.

<3> If time interval between modules is too long or short, Vivado considers them as unused and remove (or optimize) them. This would cause severe problem in operation of the system.

We solved the first problem by allocating output port of Single PE to physical port. By doing so, we can prevent Vivado to optimize Single PE module. Second problem is also solved by utilizing *(* keep = "true" *)* and *(* dont_touch = "true" *)* options in TOP module. By amending two problems, we almost solved the problem that first part of 2X2 Systolic module prints wrong values. When testing the various input and filter values, 8 of 10 were successful. The remaining 2 cases still have same problem that first calculation result of 2X2 Systolic module prints wrong value. This problem was solved by changing the timing interval between each modules in controller module. We thought that this problem is related to third hypothesis. In current situation, strategy for solving this problem is to find the time

interval that does not cause optimization in certain input and filter matrix values. To fully remove this weird problem, we think that we should directly set input and filter matrix values by using input ports. In this code, input and filter matrix values are already saved in memory module. Because input is predefined, Vivado optimizes the circuits by utilizing that fixed inputs. This would cause various optimization strategy according to input and filter values. Various optimization can cause difference between calculation interval and predefined interval which is configured in controller module. If we use external input ports which are not predefined values, Vivado would not optimize according to input and filter values. Although we can't verify the hypothesis because of the limitations of input ports, we strongly suggest that this would solve irregular optimization problem.

9.3 Limitation & Improvement of Display Module

A limitation of the display module is that the clock_divider and FSM parts were not implemented at the gate level. However, both components can potentially be implemented using gate level design. For the clock_divider, it is possible to construct a large scale counter capable of reducing a 100MHz clock down to 1Hz. Such a large counter can be realized using a series of flip flops. Since $\log_2(100{,}000{,}000) \approx 26.58$, a counter composed of 27 flip flops would be sufficient to achieve this division, enabling a gate level implementation. Similarly, the FSM used in the display module, which consists of 9 states including the initial state, can also be implemented as a saturation counter ranging from 0 to 8 using just 3 flip flops. If this display module were to be further developed, exploring these possibilities and enhancing the design to include full gate level implementation would be a meaningful improvement.