CSE 4224: Digital System Design Laboratory

4x4 4-bit Image Convolution Using 3x3 Kernel Hardware Implementation Using Verilog

By,

Afiat Khan Tahsin

Roll:1907047

Date of Submission: 29-01-2025

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna 9203, Bangladesh

4x4 4 bit Image Convolution Using 3x3 Kernel Hardware Implementation using Verilog

Contents

1 Introduction

Image processing is a crucial domain in digital systems, where convolution is widely used for feature extraction, edge detection, and filtering. Convolution operations apply a small matrix (kernel) to an image to produce a filtered output. This report presents the design and implementation of a 4x4 image convolution system using a 3x3 kernel. The system is implemented using digital logic and controlled via a finite state machine (FSM), ensuring efficient and structured execution.

2 Objectives

1. Develop a digital system to perform 4x4 image convolution using a 3x3 kernel.
2. Implement an FSM-based approach for structured control flow.
3. Utilize registers, ALU, memory units, and control logic for efficient processing.

3 System Design

The convolution system consists of the following main components:

- **Program Counter (PC)**: A 2-bit counter that selects the address counter using a demultiplexer (Demux) each time when T1 is high.
- **Address Counter**:Consists of four 4-bit counters, which increment when T2 is high. Each counter generate a sequence. For example: first counter generate 0-1-2-4-5-6-8-9-¿10 sequence. Which is the first 3x3 portion indices of the 4x4 image.
- **Demux and 4x16 Decoder**: The Demux is connected to a 4x16 decoder, which selects the rows of the ROM.
- **Kernel Counter**: Increments when T3 is high and, using a 4x16 decoder, selects the kernel rows in the kernel ROM.
- **ROM (4x16) Storage**: Stores the 4-bit image.
- **Kernel ROM (4x16)**: Stores the kernel values.
- **Multiplication Unit**: Performed by the ALU in the T4 state. The operation is controlled by s0, s1 bits.
- **Register A**: Stores the multiplication result in the T4 state.
- **Addition Unit**: Performed in the T5 state with ALU control pins S0, S1 set to 1,0. The addition result is stored in Register B.

- **Register B**: Enabled in the T5 state to store the addition result.
- **RAM Storage**: Addressed by a 2x4 decoder where each kernel sliding result is stored.
- **Control Unit**: Generates FSM states using four D flip-flops and a 4x16 decoder.
- **8-bit Register**: A sequential circuit consisting of 8 D flip-flops that store and maintain an 8-bit value based on clock pulses, useful for intermediate data storage.

4 Modules and Code Implementations

4.1 D Flip Flop

A fundamental memory storage element used in registers and sequential logic to maintain state information.

```
`timescale 1ps / 1ps

module D_Flip_Flop (
    input D,
    input CLK,
    input EN,
    input CLR,
    input L,
    output Q,
    output Q_bar
);

    // Internal signal declaration
    reg temp;

    // Asynchronous Reset and Sequential Logic
    always @(posedge CLK or posedge CLR) begin
        if (CLR) begin
            temp <= 1'b0; // Reset to 0 on CLR
        end else if (EN) begin
            if (L) begin
                temp <= D; // Load D when L is high
            end
        end
    end

    // Assign output signals
    assign Q = temp;
    assign Q_bar = ~temp;

endmodule
```

4.2 Program Counter

A 2-bit counter that sequences through instruction addresses, selecting the appropriate counter for data fetching.

```
`timescale 1ns / 1ps

module program_counter (
    input clk,
    input reset,
    output reg [1:0] Q
);

    wire [2:0] D;
    wire [2:0] Q_bar;

    assign D[1] = Q[1] ^ Q[0];
    assign D[0] = ~Q[0];

    D_Flip_Flop dff0 (
        .D(D[0]), .CLK(clk), .EN(1'b1), .CLR(reset), .L(1'b1), .Q(Q
            [0]), .Q_bar(Q_bar[0])
    );

    D_Flip_Flop dff1 (
        .D(D[1]), .CLK(clk), .EN(1'b1), .CLR(reset), .L(1'b1), .Q(Q
            [1]), .Q_bar(Q_bar[1])
    );
endmodule
```

4.3 Demultiplexer (Demux)

A 1x4 Demux that routes the program counter's output to one of the four address counters, facilitating instruction decoding.

```
`timescale 1ns / 1ps

module Demux_1x4 (
    input I,
    input [1:0] S,
    output [3:0] Y
);
    wire S1_n, S0_n;
    not (S1_n, S[1]);
    not (S0_n, S[0]);
    and (Y[0], I, S1_n, S0_n);
    and (Y[1], I, S1_n, S[0]);
    and (Y[2], I, S[1], S0_n);
    and (Y[3], I, S[1], S[0]);
endmodule
```

4.4 ROM

A 4x16 memory storing the 4-bit pixel values of the input image. The stored values are retrieved based on the address selection.

```
`timescale 1ns / 1ps

module ROM (
    input [3:0] address,
    output [3:0] data
);


    wire [15:0] row_select;
    wire [3:0] rom_data [15:0];
    reg [3:0] D [15:0];


    initial begin
        D[0]  = 4'b0001;
        D[1]  = 4'b0010;
        D[2]  = 4'b0011;
        D[3]  = 4'b0010;
        D[4]  = 4'b0001;
        D[5]  = 4'b0000;
        D[6]  = 4'b0001;
        D[7]  = 4'b0001;
        D[8]  = 4'b0011;
        D[9]  = 4'b0010;
        D[10] = 4'b0011;
        D[11] = 4'b0010;
        D[12] = 4'b0011;
        D[13] = 4'b0011;
        D[14] = 4'b0011;
        D[15] = 4'b0001;
    end


    decoder4to16 decoder (
        .address(address),
        .row_select(row_select)
    );


    generate
        genvar i, j;
        for (i = 0; i < 16; i = i + 1) begin : ROW
            for (j = 0; j < 4; j = j + 1) begin : BIT
                D_Flip_Flop dff (
                    .D(D[i][j]),
                    .CLK(row_select[i]),
                    .Q(rom_data[i][j]),
                    .Q_bar(),
                    .EN(1'b1),
                    .CLR(1'b0),
                    .L(1'b1)
                );
            end
```

```
        end
    endgenerate


    assign data = rom_data[address];

endmodule
```

4.5 Four-bit Adder

A combinational logic circuit that performs bitwise addition of two 4-bit binary numbers with an optional carry input.

```
`timescale 1ps / 1ps

module four_bit_adder (
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output [3:0] Sum,
    output Cout
);
    wire c1, c2, c3;
    full_adder FA0 (.a(A[0]), .b(B[0]), .cin(Cin), .s(Sum[0]), .
        cout(c1));
    full_adder FA1 (.a(A[1]), .b(B[1]), .cin(c1), .s(Sum[1]), .cout
        (c2));
    full_adder FA2 (.a(A[2]), .b(B[2]), .cin(c2), .s(Sum[2]), .cout
        (c3));
    full_adder FA3 (.a(A[3]), .b(B[3]), .cin(c3), .s(Sum[3]), .cout
        (Cout));
endmodule
```

4.6 Multiplier

A 4x4-bit multiplier producing an 8-bit product, crucial for performing element-wise multiplication during convolution operations.

```
`timescale 1ns / 1ps

module multiplier_4x4 (
    input [3:0] A,
    input [3:0] B,
    output [7:0] Product
);
    wire [3:0] pp0, pp1, pp2, pp3;
    wire [4:0] sum1, sum2, sum3;
    wire c1, c2;
    assign pp0 = A & {4{B[0]}};
    assign pp1 = A & {4{B[1]}};
    assign pp2 = A & {4{B[2]}};
    assign pp3 = A & {4{B[3]}};
    four_bit_adder ADD1 (.A(pp1), .B({1'b0, pp0[3:1]}), .Cin(1'b0),
         .Sum(sum1[3:0]), .Cout(sum1[4]));
```

```
    four_bit_adder ADD2 (.A(pp2), .B({sum1[4], sum1[3:1]}), .Cin(1'
        b0), .Sum(sum2[3:0]), .Cout(sum2[4]));
    four_bit_adder ADD3 (.A(pp3), .B({sum2[4], sum2[3:1]}), .Cin(1'
        b0), .Sum(sum3[3:0]), .Cout(sum3[4]));
    assign Product = {sum3[4], sum3[3:0], sum2[0], sum1[0], pp0
        [0]};
endmodule
```

4.7 ALU

The Arithmetic Logic Unit (ALU) is responsible for both addition and multiplication, determined by a control selection signal.

```
`timescale 1ns / 1ps

module ALU (
    input [3:0] A,
    input [3:0] B,
    input [1:0] Sel,
    output [7:0] Result
);
    wire [7:0] Sum;
    wire [7:0] Product;
    wire Cout;
    wire [7:0] SelSum;
    wire [7:0] SelProduct;
    eight_bit_adder ADD (.A({4'b0000, A}), .B({4'b0000, B}), .Cin
        (1'b0), .Sum(Sum), .Cout(Cout));
    multiplier_4x4 MULT (.A(A), .B(B), .Product(Product));
    assign SelSum = {8{Sel[1]}} & Sum;
    assign SelProduct = {8{Sel[0] & ~Sel[1]}} & Product;
    assign Result = SelSum | SelProduct;
endmodule
```

4.8 Address Counter (first Sequence Register

A 4-bit sequence register that determines the cell of the ROM to select for ALU operations.

```
``timescale 1ps / 1ps

module counter_one (
    input CLK,
    input RESET,     // Reset input (maps to CLR in flip-flop)
    output [3:0] Q
        // 4-bit counter output
);



    wire [3:0] D;
    wire [3:0] Q_bar;


```

```
assign D[3]=  (Q[2] & Q[1]) | (Q[3] & ~Q[1]);
assign D[2] = (~Q[3] & ~Q[2] & Q[1]) | (Q[2] & ~Q[1]);
assign D[1] =  Q[0];
assign D[0] = (~Q[1] & ~Q[0]);

    // Instantiate D flip-flops
    D_Flip_Flop dff0 (
        .D(D[0]),
        .CLK(CLK),
        .EN(1'b1),    // Always enabled
        .CLR(RESET), // Reset signal
        .L(1'b1),     // Always load
        .Q(Q[0]),
        .Q_bar(Q_bar[0])
    );

    D_Flip_Flop dff1 (
        .D(D[1]),
        .CLK(CLK),
        .EN(1'b1),    // Always enabled
        .CLR(RESET), // Reset signal
        .L(1'b1),     // Always load
        .Q(Q[1]),
        .Q_bar(Q_bar[1])
    );

    D_Flip_Flop dff2 (
        .D(D[2]),
        .CLK(CLK),
        .EN(1'b1),    // Always enabled
        .CLR(RESET), // Reset signal
        .L(1'b1),     // Always load
        .Q(Q[2]),
        .Q_bar(Q_bar[2])
    );
          D_Flip_Flop dff3 (
        .D(D[3]),
        .CLK(CLK),
        .EN(1'b1),    // Always enabled
        .CLR(RESET), // Reset signal
        .L(1'b1),     // Always load
        .Q(Q[3]),
        .Q_bar(Q_bar[3])
    );




endmodule
```

4.9 Control Unit

Implements an FSM using four D flip-flops and a 4x16 decoder to generate timing and control signals for managing the entire convolution process.

```
`timescale 1ns / 1ps

module control_unit (
    input clk,
    input reset,
    input E,
    output reg [15:0] T
);
    reg [3:0] D;
    wire [3:0] G;
    D_Flip_Flop dff0 (.D(D[0]), .CLK(clk), .EN(1'b1), .CLR(reset),
        .L(1'b1), .Q(G[0]), .Q_bar());
    D_Flip_Flop dff1 (.D(D[1]), .CLK(clk), .EN(1'b1), .CLR(reset),
        .L(1'b1), .Q(G[1]), .Q_bar());
    D_Flip_Flop dff2 (.D(D[2]), .CLK(clk), .EN(1'b1), .CLR(reset),
        .L(1'b1), .Q(G[2]), .Q_bar());
    D_Flip_Flop dff3 (.D(D[3]), .CLK(clk), .EN(1'b1), .CLR(reset),
        .L(1'b1), .Q(G[3]), .Q_bar());
    decoder4to16 decoder (.address(G), .row_select(T));
    always @(*) begin
        D[3] = (~G[3] & G[2] & G[1] & G[0] & E);
        D[2] = (~G[3] & ~G[2] & G[1] & G[0]) | (~G[3] & G[2] & ~G
            [1] & ~G[0]) | (~G[3] & G[2] & ~G[1] & G[0]) | (~G[3] &
             G[2] & G[1] & ~G[0]);
        D[1] = (~G[3] & ~G[2] & ~G[1] & G[0]) | (~G[3] & ~G[2] & G
            [1] & ~G[0]) | (~G[3] & G[2] & ~G[1] & G[0]) | (~G[3] &
             G[2] & G[1] & ~G[0]);
        D[0] = (~G[3] & ~G[2] & ~G[1] & ~G[0]) | (~G[3] & ~G[2] & G
            [1] & ~G[0]) | (~G[3] & G[2] & ~G[1] & ~G[0]) | (~G[3]
            & G[2] & G[1] & ~G[0])| (~G[3] & G[2] & G[1] & G[0] & ~
            E);
    end
endmodule
```

4.10 Kernel ROM

A 4x16 memory storing the kernel values used for convolution, selected through a kernel counter and a decoder.

```
`timescale 1ns / 1ps

module kernel (
    input [3:0] address,
    output [3:0] data
);

    // Internal signals
    wire [8:0] row_select;
    wire [3:0] rom_data [8:0];
    reg [3:0] D [8:0];

    // Initialize ROM with specific data
    initial begin
        D[0]  = 4'b0001;
        D[1]  = 4'b0010;
```

```
        D[2]  = 4'b0001;
        D[3]  = 4'b0000;
        D[4]  = 4'b0001;
        D[5]  = 4'b0010;
        D[6]  = 4'b0001;
        D[7]  = 4'b0000;
        D[8]  = 4'b0001;
    end


    decoder4to16 decoder (
        .address(address),
        .row_select(row_select)
    );


    generate
        genvar i, j;
        for (i = 0; i < 9; i = i + 1) begin : ROW
            for (j = 0; j < 4; j = j + 1) begin : BIT
                D_Flip_Flop dff (
                    .D(D[i][j]),
                    .CLK(row_select[i]),
                    .Q(rom_data[i][j]),
                    .Q_bar(),
                    .EN(1'b1),
                    .CLR(1'b0),
                    .L(1'b1)
                );
            end
        end
    endgenerate

    // Output Multiplexing
    assign data = rom_data[address];

endmodule
```

4.11 8-bit Register

A set of eight D flip-flops that store an 8-bit value, used for intermediate result storage and sequential data manipulation.

```
`timescale 1ns / 1ps

module register_8bit (
    input [7:0] in,
    input i_en,
    input clr,
    input clk,
    output [7:0] out
);
    D_Flip_Flop dff0 (.D(in[0]), .CLK(clk), .EN(~i_en), .CLR(clr),
        .L(1'b1), .Q(out[0]));
    D_Flip_Flop dff1 (.D(in[1]), .CLK(clk), .EN(~i_en), .CLR(clr),
        .L(1'b1), .Q(out[1]));
```

```
    D_Flip_Flop dff2 (.D(in[2]), .CLK(clk), .EN(~i_en), .CLR(clr),
        .L(1'b1), .Q(out[2]));
    D_Flip_Flop dff3 (.D(in[3]), .CLK(clk), .EN(~i_en), .CLR(clr),
        .L(1'b1), .Q(out[3]));
    D_Flip_Flop dff4 (.D(in[4]), .CLK(clk), .EN(~i_en), .CLR(clr),
        .L(1'b1), .Q(out[4]));
    D_Flip_Flop dff5 (.D(in[5]), .CLK(clk), .EN(~i_en), .CLR(clr),
        .L(1'b1), .Q(out[5]));
    D_Flip_Flop dff6 (.D(in[6]), .CLK(clk), .EN(~i_en), .CLR(clr),
        .L(1'b1), .Q(out[6]));
    D_Flip_Flop dff7 (.D(in[7]), .CLK(clk), .EN(~i_en), .CLR(clr),
        .L(1'b1), .Q(out[7]));
endmodule
```

4.12 RAM

A 2x4 memory storing the convolution results after computation, enabling retrieval for further processing or display.

```
`timescale 1ns / 1ps

module RAM (
    input [7:0] data_in,     // 8-bit data input
    input [1:0] address,     // 2-bit address input (for 4 addresses
        )
    input write_enable,      // Write enable signal
    input clk,               // Clock signal
    input clr,               // Clear signal for reset
    output [7:0] data_out    // 8-bit data output
);

    // Internal signals
    wire [3:0] row_select;       // Decoder output for row selection
    reg [7:0] data_out_reg;      // Output register for data_out

    // Memory implementation using 32 D flip-flops (4x8 matrix)
    wire [7:0] memory [3:0];     // 4x8 memory array

    // Address decoder: 2-to-4 line decoder
    decoder2to4 decoder (
        .address(address),
        .row_select(row_select)
    );

    // Instantiate D flip-flops for each bit in the 4x8 memory
    genvar i, j;
    generate
        for (i = 0; i < 4; i = i + 1) begin : ROW
            for (j = 0; j < 8; j = j + 1) begin : BIT
                D_Flip_Flop dff (
                    .D(data_in[j]),               // Data input for
                        the flip-flop
                    .CLK(clk),                    // Clock signal
                    .EN(write_enable & row_select[i]), // Enable
                        writing on selected row
                    .CLR(clr),                    // Clear signal
```

```
                    .L(1'b1),                       // Always load
                        when EN is high
                    .Q(memory[i][j]),               // Data stored in
                         the flip-flop
                    .Q_bar()                        // Not used
                );
            end
        end
    endgenerate

    // Read operation: Assign the selected row to the output
    always @(*) begin
        case (address)
            2'b00: data_out_reg = memory[0];
            2'b01: data_out_reg = memory[1];
            2'b10: data_out_reg = memory[2];
            2'b11: data_out_reg = memory[3];
            default: data_out_reg = 8'b00000000;
        endcase
    end

    // Connect the output register to the data_out port
    assign data_out = data_out_reg;

endmodule
```

5 Inputs and Outputs

- **Inputs**: Image pixel values (4x4 matrix), kernel values (3x3 matrix).

```
initial begin
    D[0]  = 4'b0001;
    D[1]  = 4'b0010;
    D[2]  = 4'b0011;
    D[3]  = 4'b0010;
    D[4]  = 4'b0001;
    D[5]  = 4'b0000;
    D[6]  = 4'b0001;
    D[7]  = 4'b0001;
    D[8]  = 4'b0011;
    D[9]  = 4'b0010;
    D[10] = 4'b0011;
    D[11] = 4'b0010;
    D[12] = 4'b0011;
    D[13] = 4'b0011;
    D[14] = 4'b0011;
    D[15] = 4'b0001;
end
```

Figure 1: Image

```
// Initialize ROM with specific data
initial begin
    D[0]  = 4'b0001;
    D[1]  = 4'b0010;
    D[2]  = 4'b0001;
    D[3]  = 4'b0000;
    D[4]  = 4'b0001;
    D[5]  = 4'b0010;
    D[6]  = 4'b0001;
    D[7]  = 4'b0000;
    D[8]  = 4'b0001;
end
```

Figure 2: Kernel

- **Outputs**: Convolution result matrix stored in RAM.

6 Implementation

The implementation follows a structured sequence controlled by the FSM:

6.1 Flow Chart

The system follows these steps:

- The system starts with program counter increment.
- Address counter and kernel counter update accordingly.
- Multiplication and addition operations are performed.
- A control input (E) determines whether to store the result or continue processing.

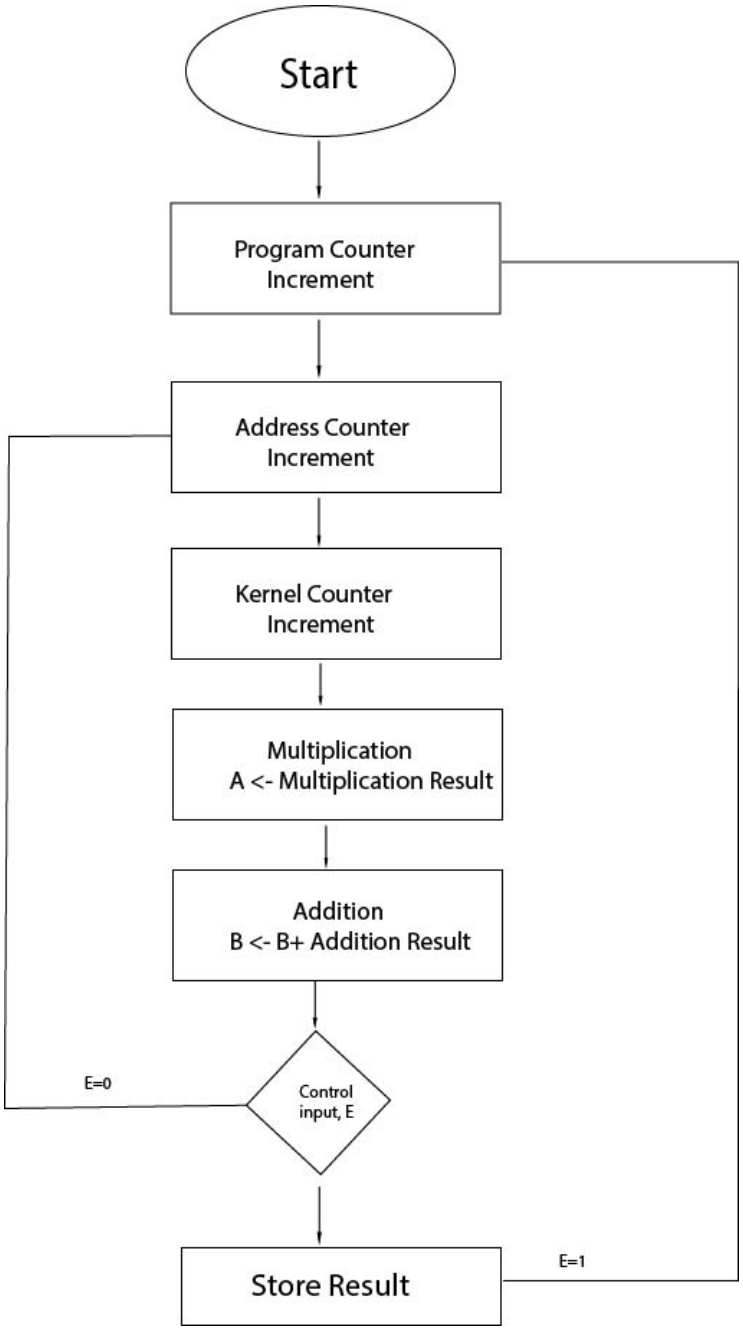

Figure 3: Flow Chart of the System

6.2 State Diagram

The FSM transitions through states from T0 to T8. The control input (E) dictates whether the system loops back or stores the final result.

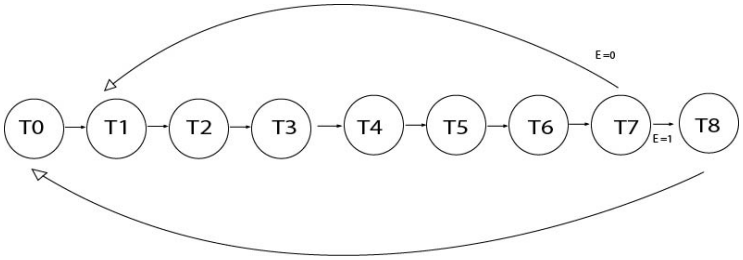

Figure 4: State Diagram of the System

6.3 Block Diagram

Demonstrates data flow between components, including program counter, demultiplexer, address counter, ROM, ALU, control unit, and memory.

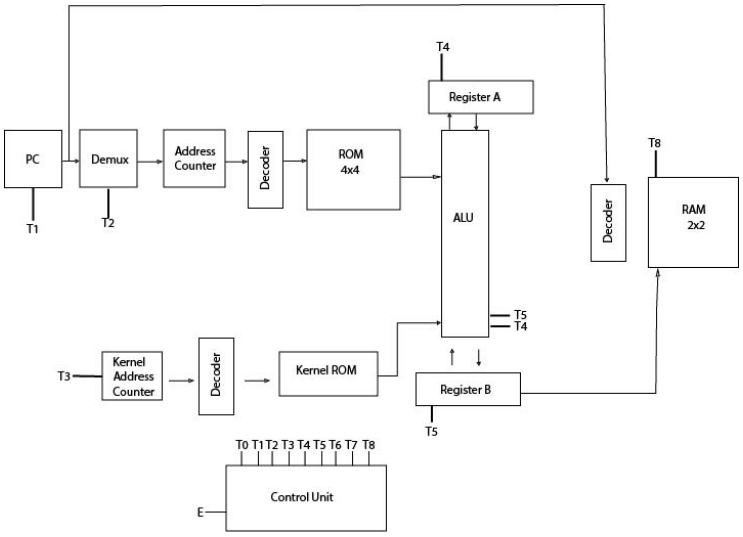

Figure 5: Block Diagram of the System

7 Results

- The designed system successfully performs 4x4 image convolution using a 3x3 kernel.

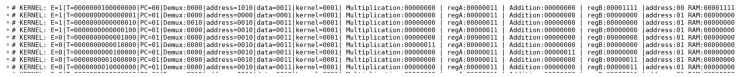

Figure 6: Result showing a portion where T8 state transit to T0 state storing the result in RAM.

- State transitions ensure proper control of data flow and computation.
- The output is stored in RAM after the completion of the convolution process.

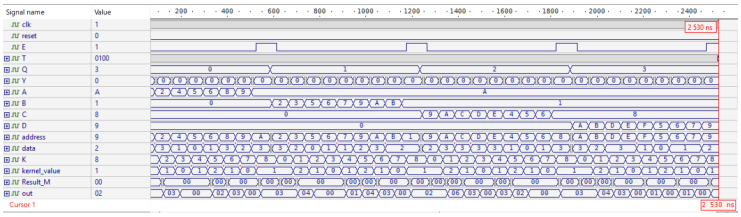

Figure 7: Output Waveform of the System

8 Conclusion

The digital convolution system was successfully developed using an FSM-based approach. Key components, including the ALU, control unit, and memory handling, were implemented effectively. Future improvements include optimizing memory access for larger image sizes and integrating advanced convolution techniques.

9 References

1. Digital System Design Course Materials.
2. Relevant Image Processing Literature.