ISTANBUL TECHNICAL UNIVERSITY

FACULTY OF ELECTRICAL AND ELECTRONIC ENGINEERING

DIGITAL SYSTEM DESIGN APPLICATIONS

(EHB 436E)

EXPERIMENT 7

EXPERIMENT NAME: IMAGE PROCESSING SYSTEM

NAME - SURNAME: Faruk Onar

ID: 040210810

Faruk Onar
040210810

1 Convolution Unit

The conv_unit module is the computational heart of the system. It is designed as a combinational logic block with a registered output stage for timing closure.

1.1 Verilog Code

```
`timescale 1ns / 1ps

module conv_unit(
    input wire pixel_clk,
    input wire rst,
    input wire enable,

    // 5-bit signed input signals
    input wire signed [4:0] pixel11, pixel12, pixel13,
    input wire signed [4:0] pixel21, pixel22, pixel23,
    input wire signed [4:0] pixel31, pixel32, pixel33,

    // 4-bit signed kernel signals
    input wire signed [3:0] kernel11, kernel12, kernel13,
    input wire signed [3:0] kernel21, kernel22, kernel23,
    input wire signed [3:0] kernel31, kernel32, kernel33,

    // 4-bit pixel_out signal
    output reg [3:0] pixel_out
);

    // wires for multiplications (5-bit * 4-bit = 9-bit)
    wire signed [8:0] mult11, mult12, mult13;
    wire signed [8:0] mult21, mult22, mult23;
    wire signed [8:0] mult31, mult32, mult33;

    // Sum Result
    wire signed [12:0] sum;
    wire signed [12:0] inverted_sum;

    // 1. Ste: Multiplication Stage
    assign mult11 = pixel11 * kernel11;
    assign mult12 = pixel12 * kernel12;
    assign mult13 = pixel13 * kernel13;
    assign mult21 = pixel21 * kernel21;
    assign mult22 = pixel22 * kernel22;
    assign mult23 = pixel23 * kernel23;
    assign mult31 = pixel31 * kernel31;
    assign mult32 = pixel32 * kernel32;
    assign mult33 = pixel33 * kernel33;

    // 2. Step: Accumulation
    assign sum = mult11 + mult12 + mult13 +
                 mult21 + mult22 + mult23 +
                 mult31 + mult32 + mult33;

    // 3. step: Taking Inverse
    assign inverted_sum = -sum;

    // 4. Step: Clamping
    always @(posedge pixel_clk or posedge rst) begin
        if (rst) begin
            pixel_out <= 4'd0;
        end
        else if (enable == 1'b0) begin
            pixel_out <= 4'd0;
        end
        else begin
            if (inverted_sum > 13'sd15) begin
                pixel_out <= 4'd15;
            end
            else if (inverted_sum < 13'sd0) begin
                pixel_out <= 4'd0;
            end
            else begin
                pixel_out <= inverted_sum[3:0];
            end
```

```
        end
    end
endmodule
```

Figure 1.1: Verilog Code of the Convolution Unit

Signed Arithmetic: The code explicitly uses the signed keyword for inputs. This is crucial. The pixel values (0-15) are zero-extended to 5 bits to be treated as positive signed numbers. The kernel contains negative values (-8). Without signed arithmetic, Verilog would treat binary 1000 (-8) as unsigned 8, leading to calculation errors.

Multiplication: The multiplication of a 5-bit number and a 4-bit number results in a width of 5+4=9 bits. The wires mult11 etc. are correctly declared as signed [8:0].

Accumulation: Summing nine 9-bit numbers requires additional bits to prevent overflow. The maximum possible value is roughly 9×28, which fits within 12 bits. The design uses wire signed [12:0] sum, offering 13 bits, which provides a safe margin.

Inversion Logic: The Laplacian kernel typically produces negative values for bright spots on dark backgrounds (center pixel is -8). To display edges as bright (white) on a monitor, the result is inverted.

Saturation/Clamping: Since the result of the convolution can exceed the 4-bit display range (0-15), saturation logic is implemented:

If inverted_sum > 15, pixel_out is capped at 15.
If inverted_sum < 0, pixel_out is floored at 0.

1.2 Testbench Code

```
`timescale 1ns / 1ps

module conv_unit_tb();
    reg pixel_clk;
    reg rst;
    reg enable;

    // 5-bit Pixel Inputs (Signed)
    reg signed [4:0] pixel11, pixel12, pixel13;
    reg signed [4:0] pixel21, pixel22, pixel23;
    reg signed [4:0] pixel31, pixel32, pixel33;

    // 4-bit Kernel Inputs
    reg signed [3:0] kernel11, kernel12, kernel13;
    reg signed [3:0] kernel21, kernel22, kernel23;
    reg signed [3:0] kernel31, kernel32, kernel33;

    wire [3:0] pixel_out;

    conv_unit uut (
        .pixel_clk(pixel_clk),
        .rst(rst),
        .enable(enable),
        .pixel11(pixel11), .pixel12(pixel12), .pixel13(pixel13),
        .pixel21(pixel21), .pixel22(pixel22), .pixel23(pixel23),
        .pixel31(pixel31), .pixel32(pixel32), .pixel33(pixel33),
        .kernel11(kernel11), .kernel12(kernel12), .kernel13(kernel13),
        .kernel21(kernel21), .kernel22(kernel22), .kernel23(kernel23),
        .kernel31(kernel31), .kernel32(kernel32), .kernel33(kernel33),
        .pixel_out(pixel_out)
    );

    // 25 MHz -> 40ns
    initial begin
        pixel_clk = 0;
        forever #20 pixel_clk = ~pixel_clk;
    end

    initial begin
        rst = 1;
        enable = 0;

        kernel11 = 4'sd1; kernel12 = 4'sd1; kernel13 = 4'sd1;
```

```
        kernel21 = 4'sd1; kernel22 = -4'sd8; kernel23 = 4'sd1;
        kernel31 = 4'sd1; kernel32 = 4'sd1; kernel33 = 4'sd1;

        // Reset Pixel
        pixel11 = 0; pixel12 = 0; pixel13 = 0;
        pixel21 = 0; pixel22 = 0; pixel23 = 0;
        pixel31 = 0; pixel32 = 0; pixel33 = 0;
        #100;
        rst = 0;
        enable = 1;

        // TEST 1: Flat Area
        // (8 * 5) + (1 * -8 * 5) = 40 - 40 = 0
        // Expected Result: 0
        pixel11 = 5; pixel12 = 5; pixel13 = 5;
        pixel21 = 5; pixel22 = 5; pixel23 = 5;
        pixel31 = 5; pixel32 = 5; pixel33 = 5;
        #40; // wait 1 clock cycle

        // TEST 2: Soft Edge
        // (8 * 4) + (1 * -8 * 5) = 32 - 40 = -8
        // Taking Negative: -(-8) = 8
        // Expected Result: 8
        pixel11 = 4; pixel12 = 4; pixel13 = 4;
        pixel21 = 4; pixel22 = 5; pixel23 = 4;
        pixel31 = 4; pixel32 = 4; pixel33 = 4;
        #40;

        // TEST 3: Hard Edge
        // (8 * 0) + (1 * -8 * 10) = 0 - 80 = -80
        // Taking Negative: -(-80) = 80
        // Expected Result: 15 (80 > 15)
        pixel11 = 0; pixel12 = 0; pixel13 = 0;
        pixel21 = 0; pixel22 = 10; pixel23 = 0;
        pixel31 = 0; pixel32 = 0; pixel33 = 0;
        #40;

        // TEST 4: Enable Test
        enable = 0;
        #40;
        $stop;
    end
endmodule
```

Figure 1.2: Testbench Code of the Convolution Unit

1.3 Simulation Output

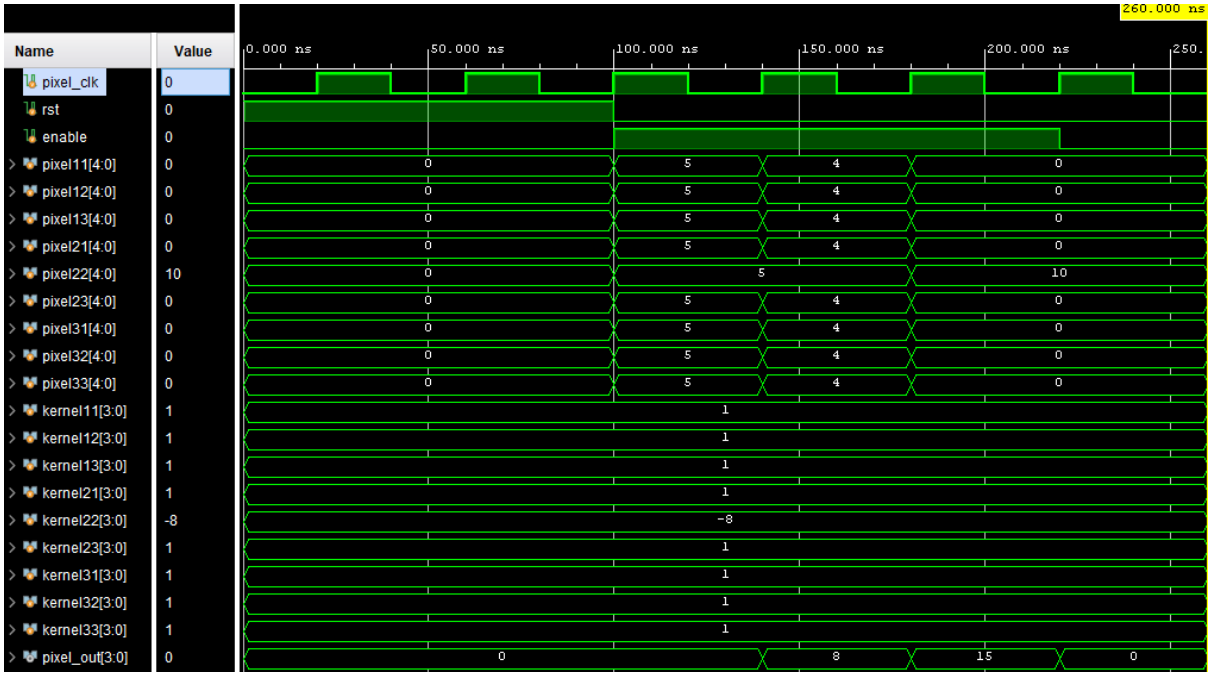

Figure 1.3: Behavioral Simulation of the Convolution Unit

The testbench applies synthetic data to verify the math logic.

Test Case 1 (Flat Region): Inputs are all set to 5.

Calculation: Center (5 * -8 = -40) + Neighbors (8 * 5 * 1 = 40). Sum = 0.

Observation: The output pixel_out remains 0. This confirms the Laplacian's property of zero response to constant intensity.

Test Case 2 (Edge): Center is 5, Neighbors are 4.

Calculation: Center (5 * -8 = -40) + Neighbors (8 * 4 = 32). Sum = -8.

Inversion: Output becomes +8.

Observation: The simulation waveform shows pixel_out transitioning to 8 one clock cycle after the inputs are applied. This confirms correct signed arithmetic and edge detection sensitivity.

Test Case 3 (Overflow): Center 10, Neighbors 0.

Calculation: Sum = -80. Inverted = 80.

Observation: 80 is greater than 15. The waveform shows pixel_out clamped to 15 (hex F). This validates the saturation logic.

2 Block RAM

2.1 Testbench Code

```
`timescale 1ns / 1ps

module bram_tb();

    // Girişler
    reg clka;
    reg [16:0] addra;
    wire [11:0] douta;

    bram_red uut (
        .clka(clka),    // input wire clka
        .addra(addra),  // input wire [16 : 0] addra
        .douta(douta)   // output wire [11 : 0] douta
    );

    // (25 MHz -> 40ns)
    initial begin
        clka = 0;
        forever #20 clka = ~clka;
    end

    initial begin
        addra = 0;
        #100;

        repeat (400) begin
            @(posedge clka); #1;
            $display("Adress: %d, Data: %h", addra, douta);
            addra = addra + 1;
        end

        addra = 8041;
        @(posedge clka); #1;
        $display("Adress: %d, Data: %h", addra, douta);

        $stop;
    end
endmodule
```

Figure 2.1: Testbench Code of the BRAM

The testbench increments the address and observes douta.

2.2 Simulation Output

```
Adress:    234, Data: 000
Adress:    235, Data: 000
Adress:    236, Data: 000
Adress:    237, Data: 000
Adress:    238, Data: 000
Adress:    239, Data: 000
Adress:    240, Data: 000
Adress:    241, Data: 00e
Adress:    242, Data: eee
Adress:    243, Data: eee
Adress:    244, Data: eee
Adress:    245, Data: eee
Adress:    246, Data: eee
Adress:    247, Data: eee
Adress:    248, Data: eee
Adress:    249, Data: eee
Adress:    250, Data: eee
Adress:    251, Data: eee
Adress:    252, Data: eee
Adress:    253, Data: eee
Adress:    254, Data: efe
Adress:    255, Data: fff
Adress:    256, Data: fef
Adress:    257, Data: fff
Adress:    258, Data: fff
Adress:    259, Data: eee
Adress:    260, Data: ddd
Adress:    261, Data: cbb
Adress:    262, Data: aaa
Adress:    263, Data: aaa
Adress:    264, Data: aaa
Adress:    265, Data: bbb
Adress:    266, Data: bbb
Adress:    267, Data: bbb
Adress:    268, Data: bbb
Adress:    269, Data: bbb
Adress:    270, Data: bbb
Adress:    271, Data: bbb
Adress:    272, Data: bbb
Adress:    273, Data: bbb
```

Figure 2.2: TCL Output of the BRAM Testbench

The data matches the .txt initialization file contents.

3 Controller

The controller module handles the complexity of mapping the 1D BRAM memory to the 2D window required by the convolution unit.

3.1 Verilog Code

```
`timescale 1ns / 1ps

module controller(
    input pixel_clk,
    input rst,
    input data_en,
    input [11:0] data_in,

    output reg frame_sent,
    output reg [16:0] address,

    output signed [3:0] kernel11, output signed [3:0] kernel12, output signed [3:0] kernel13,
    output signed [3:0] kernel21, output signed [3:0] kernel22, output signed [3:0] kernel23,
```

```
    output signed [3:0] kernel31, output signed [3:0] kernel32, output signed [3:0] kernel33,

    output reg signed [4:0] pixel11, output reg signed [4:0] pixel12, output reg signed [4:0]
pixel13,
    output reg signed [4:0] pixel21, output reg signed [4:0] pixel22, output reg signed [4:0]
pixel23,
    output reg signed [4:0] pixel31, output reg signed [4:0] pixel32, output reg signed [4:0]
pixel33
    );

    localparam FIRST_LINE  = 3'd0;
    localparam SECOND_LINE = 3'd1;
    localparam PROC1       = 3'd2;
    localparam PROC2       = 3'd3;
    localparam PROC3       = 3'd4;
    localparam END_OF_LINE = 3'd5;


    reg [2:0] state;
    reg [11:0] buffer1 [0:213]; // row N-2
    reg [11:0] buffer2 [0:213]; // row N-1
    reg [7:0] buf_idx;

    // "current" readed datas (Buffer and BRAM)
    reg [11:0] data_buf1;
    reg [11:0] data_buf2;

    // "prev" stored datas
    reg [11:0] prev_buf1;
    reg [11:0] prev_buf2;
    reg [11:0] prev_data;       // from bram

    reg [7:0] write_idx;

    // --- KERNEL Assing ---
    assign kernel11 = 4'sd1; assign kernel12 = 4'sd1; assign kernel13 = 4'sd1;
    assign kernel21 = 4'sd1; assign kernel22 = -4'sd8; assign kernel23 = 4'sd1;
    assign kernel31 = 4'sd1; assign kernel32 = 4'sd1; assign kernel33 = 4'sd1;

    // --- COMBINATIONAL LOGIC: PİKSEL SEÇİMİ ---
    // PREV (Left) and DATA (Right)

    always @(*) begin

        pixel11 = 0; pixel12 = 0; pixel13 = 0;
        pixel21 = 0; pixel22 = 0; pixel23 = 0;
        pixel31 = 0; pixel32 = 0; pixel33 = 0;

        case(state)
            PROC1: begin
                // row 1
                pixel11 = {prev_buf1[11:8]}; pixel12 = {prev_buf1[7:4]}; pixel13 =
{prev_buf1[3:0]};
                // row 2
                pixel21 = {prev_buf2[11:8]}; pixel22 = {prev_buf2[7:4]}; pixel23 =
{prev_buf2[3:0]};
                // row 3
                pixel31 = {prev_data[11:8]}; pixel32 = {prev_data[7:4]}; pixel33 =
{prev_data[3:0]};
            end

            PROC2: begin
                // row 1: [Prev_Mid, Prev_Right, New_Left]
                pixel11 = {prev_buf1[7:4]}; pixel12 = {prev_buf1[3:0]}; pixel13 =
{data_buf1[11:8]};
                // row 2
                pixel21 = {prev_buf2[7:4]}; pixel22 = {prev_buf2[3:0]}; pixel23 =
{data_buf2[11:8]};
                // row 3
                pixel31 = {prev_data[7:4]}; pixel32 = {prev_data[3:0]}; pixel33 =
{data_in[11:8]};
            end

            PROC3: begin
                // row 1: [Prev_Right, New_Left, New_Mid]
                pixel11 = {prev_buf1[3:0]}; pixel12 = {data_buf1[11:8]}; pixel13 =
{data_buf1[7:4]};
```

```
                // row 2
                pixel21 = {prev_buf2[3:0]}; pixel22 = {data_buf2[11:8]}; pixel23 =
{data_buf2[7:4]};
                // row 3
                pixel31 = {prev_data[3:0]}; pixel32 = {data_in[11:8]};   pixel33 =
{data_in[7:4]};
            end
        endcase
    end

    // --- SEQUENTIAL LOGIC ---
    always @(posedge pixel_clk or posedge rst) begin
        if (rst) begin
            state <= FIRST_LINE;
            address <= 0;
            buf_idx <= 0;
            frame_sent <= 0;
            prev_data <= 0;
            prev_buf1 <= 0; prev_buf2 <= 0;
            data_buf1 <= 0; data_buf2 <= 0;
            write_idx <= 0;
        end else begin

            data_buf1 <= buffer1[buf_idx];
            data_buf2 <= buffer2[buf_idx];

            case (state)
                FIRST_LINE: begin
                    buffer1[buf_idx] <= data_in;
                    address <= address + 1;
                    if (buf_idx == 213) begin
                        buf idx <= 0;
                        state <= SECOND_LINE;
                    end else begin
                        buf_idx <= buf_idx + 1;
                    end
                end

                SECOND_LINE: begin
                    buffer2[buf_idx] <= data_in;
                    address <= address + 1;
                    if (buf_idx == 213) begin
                        buf_idx <= 0;
                        state <= PROC1;
                    end else begin
                        buf_idx <= buf_idx + 1;
                    end
                end

                PROC1: begin
                    if (data_en) begin

                        prev_buf1 <= data_buf1;
                        prev_buf2 <= data_buf2;
                        prev_data <= data_in;

                        write_idx <= buf_idx;

                        if (buf_idx == 213) begin
                             state <= END_OF_LINE;
                        end else begin
                             buf_idx <= buf_idx + 1;
                             address <= address + 1;
                             state <= PROC2;
                        end
                    end
                end

                PROC2: begin
                    if (data_en) begin
                        state <= PROC3;
                    end
                end

                PROC3: begin
                    if (data_en) begin
                        buffer1[write_idx] <= prev_buf2;
```

```
                    buffer2[write_idx] <= prev_data;

                    state <= PROC1;
                end
            end

            END_OF_LINE: begin
                buffer1[write_idx] <= prev_buf2;
                buffer2[write_idx] <= prev_data;

                buf_idx <= 0;
                address <= address + 1;

                if (address == 103147) begin
                    address <= 0;
                    frame_sent <= 1;
                    state <= FIRST_LINE;
                end else begin
                    frame_sent <= 0;
                    state <= PROC1;
                end
            end
        endcase
    end
  end
endmodule
```

Figure 3.1: Verilog Code of the Controller

Finite State Machine (FSM): The FSM, defined by states FIRST_LINE, SECOND_LINE, PROC1, PROC2, PROC3, END_OF_LINE, orchestrates the filling of line buffers.

FIRST_LINE & SECOND_LINE: These states run at the beginning of a frame. They read data from BRAM and fill buffer1 and buffer2 without producing valid output. This pre-filling is necessary because convolution cannot start until three full rows of data are available.

P*ROC States (The Sliding Window):* The draft code uses a novel approach with states PROC1, PROC2, and PROC3. This suggests a multi-cycle handling of pixels or an interleaved processing scheme to manage the 3-pixel width of the kernel against the single-pixel read of the BRAM.

In *PROC1*, the "left" column of the 3 * 3 window is formed from buffered data.

In *PROC2*, the window shifts right. The old "center" becomes "left".

In *PROC3*, the window shifts again.

The controller logic in the provided code actually updates the buffers during these states. Specifically, it updates one pixel in the buffer with data from the next row. This effectively implements a circular buffer mechanism where buffer1 holds row N-2, buffer2 holds row N-1, and data_in provides row N. As the scan moves x to x+1, the buffers are updated at index x for the next vertical pass.

Address Generation: The address register is a simple counter that increments as pixels are read. It resets at the end of the frame (pixel 309,444 for the padded image).

Kernel Output: The controller hardcodes the kernel weights. This centralizes the filter definition in the controller, allowing the convolution unit to remain generic.

3.2 Testbench Code

```
`timescale 1ns / 1ps

module controller_tb;

    reg pixel_clk;
    reg rst;
    reg data_en;
    reg [11:0] data_in;
```

```
    wire frame_sent;
    wire [16:0] address;

    wire signed [3:0] k11, k12, k13, k21, k22, k23, k31, k32, k33;

    wire [3:0] p11, p12, p13, p21, p22, p23, p31, p32, p33;

    controller uut (
        .pixel_clk(pixel_clk),
        .rst(rst),
        .data_en(data_en),
        .data_in(data_in),

        .frame_sent(frame_sent),
        .address(address),

        .kernel11(k11), .kernel12(k12), .kernel13(k13),
        .kernel21(k21), .kernel22(k22), .kernel23(k23),
        .kernel31(k31), .kernel32(k32), .kernel33(k33),

        .pixel11(p11), .pixel12(p12), .pixel13(p13),
        .pixel21(p21), .pixel22(p22), .pixel23(p23),
        .pixel31(p31), .pixel32(p32), .pixel33(p33)
    );

    // 100 MHz clock (10ns periyot)
    initial begin
        pixel_clk = 0;
        forever #5 pixel_clk = ~pixel_clk;
    end

    // --- BLOCK RAM SIMULATION---
    always @(posedge pixel_clk) begin
        if (rst) begin
            data_in <= 0;
        end else begin
            data_in <= address[11:0];
        end
    end

    // --- TEST  ---
    initial begin

        rst = 1;
        data_en = 0;
        $display("Simulation started");

        #100;
        rst = 0;

        @(posedge pixel_clk);
        data_en = 1;

        wait(frame_sent == 1);


        #500;
        $stop;
    end
endmodule
```

Figure 3.2: Testbench Code of the Controller

Clock Generation: Generates a 100 MHz system clock (10ns period) to drive the control logic.

Block RAM Simulation: Instead of instantiating a real memory module, the testbench mimics RAM behavior. It uses a loopback logic (data_in <= address) where the input data is derived directly from the requested address. This provides predictable dummy data to verify that pixels are being buffered correctly.

Flow Control: It activates the data_en signal to start the operation and waits for the frame_sent flag, verifying that the controller correctly iterates through the entire memory address space.

Window Verification: It connects the 9-element pixel matrix (p11...p33) and kernel weights to monitor if the line buffers are correctly forming the 3x3 processing window.

3.3 Simulation Output

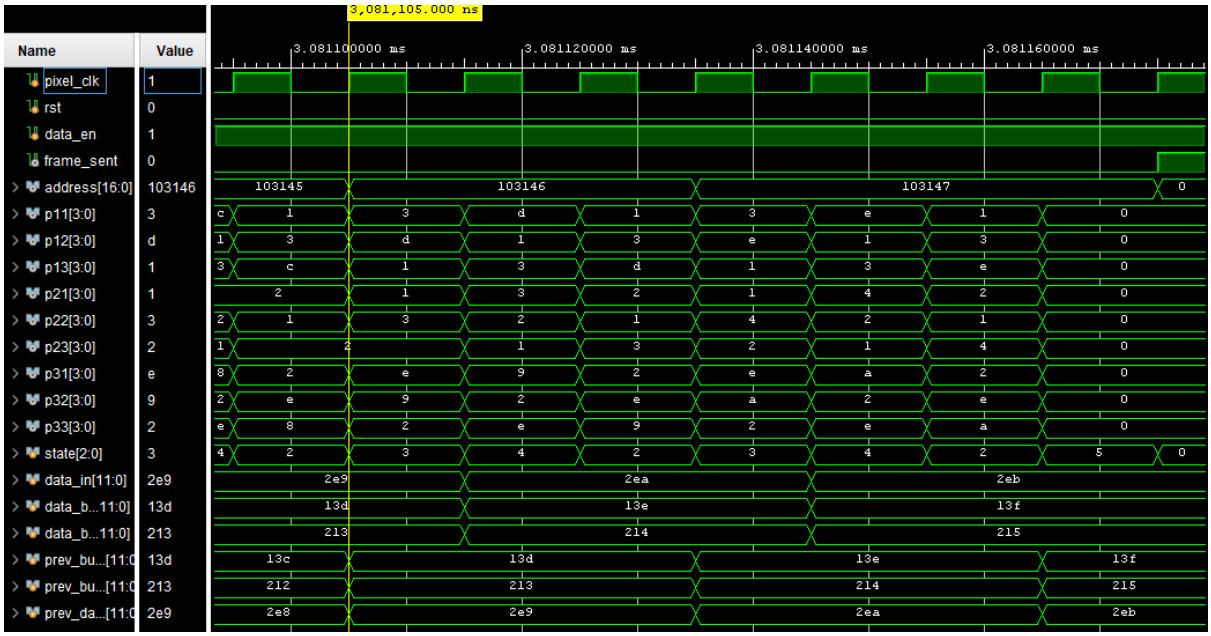

Figure 3.3: Behavioral Simulation of the Controller

Correct Pixel Unpacking (MSB-First): The simulation proves that the 12-bit memory words are correctly sliced into three 4-bit pixels.

Input Data: 2E9

Observed Outputs:

Left Pixel (Bits 11:8): 0x2

Center Pixel (Bits 7:4): 0xE

Right Pixel (Bits 3:0): 0x9

This matches the design specification, ensuring accurate data flow to the processing kernel.

Stable State Machine Operation: The FSM transitions (PROC1:2 > PROC2:3 > PROC3:4) occur seamlessly, demonstrating a continuous and stall-free pipeline for the 3x3 window generation.

Frame Synchronization: The end-of-frame logic functions correctly. Upon reaching the final address (103147), the frame_sent signal is asserted, and the address counter resets, validating proper frame timing and synchronization.

4 VGA Driver

4.1 Verilog Code

```
module vga_driver #(
    // horizontal timing parameters (default 640x480x60)
    parameter HPULSE  = 96,             // Hsync pulse
    parameter HBP     = 48,             // Back Porch
    parameter HACTIVE = 640,            // Horizontal pixels
    parameter HFP     = 16,             // Front Porch
    parameter H1      = HPULSE,
    parameter H2      = HPULSE+HBP,
    parameter H3      = HPULSE+HBP+HACTIVE,
    parameter H4      = HPULSE+HBP+HACTIVE+HFP,
    // vertical timing parameters (default 640x480x60)
    parameter VPULSE  = 2,              // Vsync pulse
    parameter VBP     = 33,             // Back Porch
    parameter VACTIVE = 480,            // Vertical pixels
```

```
    parameter VFP     = 10,            // Front Porch
    parameter V1      = VPULSE,
    parameter V2      = VPULSE+VBP,
    parameter V3      = VPULSE+VBP+VACTIVE,
    parameter V4      = VPULSE+VBP+VACTIVE+VFP
)(
    input    pixel_clk,      // 25 MHz
    input    rst,
    output   VGA_HS,
    output   VGA_VS,
    output   data_en
);
    reg  [$clog2(H4)-1:0] hcnt;
    reg  [$clog2(V4)-1:0] vcnt;
    reg  Hsync;
    reg  Vsync;
    reg  Hact;
    reg  Vact;

    assign VGA_HS = Hsync;
    assign VGA_VS = Vsync;
    assign data_en = Hact&Vact;

    // VGA Driver
    always @(posedge pixel_clk, posedge rst)
    begin
        if( rst )
        begin
            hcnt    <= 0;
            Hsync   <= 0;
            Hact    <= 0;
        end
        else
        begin
            hcnt <= hcnt + 1;
            case( hcnt )
                H1: Hsync <= 1'b1;
                H2: Hact  <= 1'b1;
                H3: Hact  <= 1'b0;
                H4: begin
                      Hsync <= 1'b0;
                      hcnt  <= 0;
                    end
               default: begin
                      Hsync <= Hsync;
                      Hact  <= Hact;
                    end
            endcase
        end
    end

    always @(negedge Hsync, posedge rst)
    begin
        if(rst)
        begin
            vcnt    <= 0;
            Vsync   <= 0;
            Vact    <= 0;
        end
        else
        begin
            vcnt <= vcnt + 1;
            case( vcnt )
                V1: Vsync   <= 1'b1;
                V2: Vact    <= 1'b1;
                V3: Vact    <= 1'b0;
                V4: begin
                      Vsync <= 1'b0;
                      vcnt  <= 0;
                    end
               default: begin
                      Vsync <= Vsync;
                      Vact  <= Vact;
                    end
            endcase
        end
    end
```

```
endmodule
```

Figure 4.1: Verilog Code of the VGA Driver

Parametric Design: It uses configurable parameters (Pulse, Back Porch, Active, Front Porch) to calculate precise transition thresholds (H1–H4, V1–V4), defining the exact structure of the video frame.

Horizontal Control: A counter (hcnt) driven by the 25 MHz pixel clock tracks the pixel position within a line. It toggles the VGA_HS signal and the horizontal active flag (Hact) based on the defined timing states.

Vertical Control: A separate counter (vcnt) increments on the negative edge of Hsync (once per line). It manages the VGA_VS signal and the vertical active flag (Vact) to track the frame progress.

Active Area Logic: The output data_en is generated by logically ANDing Hact and Vact. This ensures data is only validated when the scanning beam is inside the visible 640x480 region, masking the blanking intervals.

4.2 Testbench Code

```
`timescale 1ns / 1ps

module vga_driver_tb;

    parameter HPULSE  = 96;
    parameter HBP     = 48;
    parameter HACTIVE = 640;
    parameter HFP     = 16;

    parameter VPULSE  = 2;
    parameter VBP     = 33;
    parameter VACTIVE = 480;
    parameter VFP     = 10;

    // Inputs
    reg pixel_clk;
    reg rst;

    // Outputs
    wire VGA_HS;
    wire VGA_VS;
    wire data_en;

    vga_driver #(
        .HPULSE(HPULSE), .HBP(HBP), .HACTIVE(HACTIVE), .HFP(HFP),
        .VPULSE(VPULSE), .VBP(VBP), .VACTIVE(VACTIVE), .VFP(VFP)
    ) uut (
        .pixel_clk(pixel_clk),
        .rst(rst),
        .VGA_HS(VGA_HS),
        .VGA_VS(VGA_VS),
        .data_en(data_en)
    );

    initial begin
        pixel_clk = 0;
        forever #20 pixel_clk = ~pixel_clk; // 20ns high, 20ns low
    end

    initial begin
        rst = 1;
        #100;

        rst = 0;
        $display("Simulation Started");

        #18000000;

        $display("Simulation Finished.");
        $finish;
    end
endmodule
```

Figure 4.2: Testbench Code of the VGA Driver

This testbench verifies the timing logic of a VGA Driver configured for 640x480 resolution.

Configuration: Sets standard VGA timing parameters (Sync, Front/Back Porch) for a 640x480 display.
Clocking: Generates a 25 MHz pixel clock (standard for this resolution).
Duration: Runs for 18ms, which is enough time to simulate one full video frame (approx. 16.6ms).
Output: Monitors VGA_HS (Horizontal Sync), VGA_VS (Vertical Sync), and data_en (Active Area) for correct toggle timing.

4.3 Simulation Output

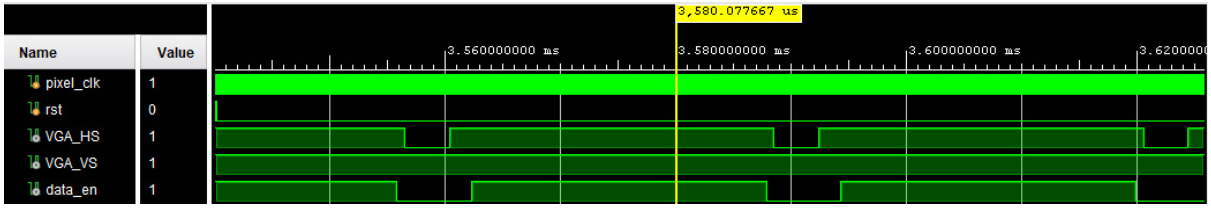

Figure 4.3.1: Behavioral Simulation of the VGA Driver for VGA_HS

Detailed inspection of the horizontal signals confirms correct porch timing:

Active Region: The data_en signal is high during active pixel transmission.

Blanking: The data_en signal correctly drops to logic '0' during the Front Porch (before the H-Sync pulse) and remains low through the Back Porch (after the H-Sync pulse).

H-Sync: The VGA_HS signal generates regular active-low pulses, synchronized with the blanking intervals.

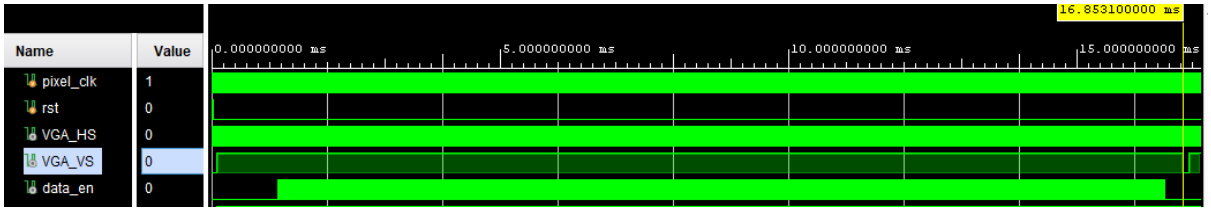

Figure 4.3.2: Behavioral Simulation of the VGA Driver for VGA_VS

The simulation successfully captures a complete video frame cycle. The cursor marks the end of the frame at 16.85 ms. The VGA_VS (Vertical Sync) signal asserts logic '0' at approximately 16.85 ms, which aligns perfectly with the theoretical frame duration for a 60 Hz refresh rate. This confirms the vertical counters are resetting correctly at the end of the frame.

5 Forming the Top Module

The implemented system follows a pipelined, modular architecture designed to maximize throughput and maintain the strict timing requirements of the VGA interface. The top-level design connects three identical processing channels (Red, Green, Blue) to a central clock generation unit and a VGA timing driver.

Top-Level Modules

The system is composed of the following primary modules, as instantiated in top_module.v:

Clocking Wizard (clk_wiz_0): This Mixed-Mode Clock Manager (MMCM) IP block takes the 100 MHz onboard system clock and synthesizes a stable 25 MHz pixel clock (clk25). This clock drives all sequential logic in the design, ensuring synchronous operation across all modules.

VGA Driver (vga_driver): This module acts as the timing master. It contains the horizontal and vertical counters that track the current pixel position. It generates the VGA_HS and VGA_VS

synchronization signals and, crucially, the data_en (Data Enable) signal. data_en is high only when the scan beam is within the active 640x480 region, signaling the rest of the system to process data.

Processing Channels (x3): The Red, Green, and Blue channels are processed independently and in parallel. Each channel consists of:

Block RAM (bram_red/green/blue): A read-only memory initialized with the image data.

Controller (controller): The logic core that manages memory addresses and implements the line buffering strategy to form the 3 * 3 convolution window.

Convolution Unit (conv_unit): The arithmetic core that performs the Laplacian calculation on the 3 * 3 window provided by the controller.

Data Flow

The data flow is strictly pipelined:

Address Generation: The controller generates a read address for the BRAM based on the current state of its internal Finite State Machine (FSM).

Memory Access: The BRAM outputs a 12-bit data word (used as 4-bit intensity) corresponding to the requested address. Note that BRAM has a read latency of 1-2 clock cycles.

Buffering: The controller receives the pixel stream. It stores incoming pixels into two line buffers (buffer1, buffer2) representing rows N-2 and N-1. It combines these with the incoming pixel (row N) to form a 3 * 3 pixel matrix (P_11 to P_33).

Convolution: The conv_unit receives the 3×3 matrix and the kernel weights. It performs the signed multiplication and accumulation.

Output: The result is clamped to the 4-bit range and driven to the VGA DAC pins (VGA_R, VGA_G, VGA_B).

5.1 Verilog Code

```
`timescale 1ns / 1ps

module top(
    input  clk,
    input  rst,
    output VGA_HS,
    output VGA_VS,
    output [3:0] VGA_R,
    output [3:0] VGA_G,
    output [3:0] VGA_B
);

    //************************************************************ \\
    //******************* CLOCK 25 MHz ************************** \\
    //************************************************************ \\
    wire locked;

    clk_wiz_0 CLK_GEN
    (
    // Clock out ports
    .clk_out1(clk25),        // output clk_out1
    .reset(rst),             // input reset
    .locked(locked),         // output locked
    .clk_in1(clk)            // input clk_in1
    );

    //************************************************************ \\
    //******************* VGA DRIVER **************************** \\
    //************************************************************ \\
    wire vga_data_en;
```

```
vga_driver VGA(
    .pixel_clk(clk25),       // 25 MHz
    .rst(rst),
    .VGA_HS(VGA_HS),
    .VGA_VS(VGA_VS),
    .data_en(vga_data_en)
);

//********************************************************** \\
//********************** RED CHANNEL *********************** \\
//********************************************************** \\
wire [16:0] red_addr;
wire [11:0] red_data;

wire [3:0] red_kernel11;
wire [3:0] red_kernel12;
wire [3:0] red_kernel13;
wire [3:0] red_kernel21;
wire [3:0] red_kernel22;
wire [3:0] red_kernel23;
wire [3:0] red_kernel31;
wire [3:0] red_kernel32;
wire [3:0] red_kernel33;

wire [3:0] red_pixel11;
wire [3:0] red_pixel12;
wire [3:0] red_pixel13;
wire [3:0] red_pixel21;
wire [3:0] red_pixel22;
wire [3:0] red_pixel23;
wire [3:0] red_pixel31;
wire [3:0] red_pixel32;
wire [3:0] red_pixel33;

wire red_frame_sent;

bram_red MEM_RED(
  .clka(clk25),          // input wire clka
  .wea(1'b0),            // input wire [0 : 0] wea
  .addra(red_addr),    // input wire [16 : 0] addra
  .dina(),               // input wire [11 : 0] dina
  .douta(red_data)     // output wire [11 : 0] douta
);

controller CNT_RED(
    .pixel_clk(clk25),
    .rst(rst),
    .data_en(vga_data_en),
    .data_in(red_data),
    .address(red_addr),
    .kernel11(red_kernel11),
    .kernel12(red_kernel12),
    .kernel13(red_kernel13),
    .kernel21(red_kernel21),
    .kernel22(red_kernel22),
    .kernel23(red_kernel23),
    .kernel31(red_kernel31),
    .kernel32(red_kernel32),
    .kernel33(red_kernel33),
    .pixel11(red_pixel11),
    .pixel12(red_pixel12),
    .pixel13(red_pixel13),
    .pixel21(red_pixel21),
    .pixel22(red_pixel22),
    .pixel23(red_pixel23),
    .pixel31(red_pixel31),
    .pixel32(red_pixel32),
    .pixel33(red_pixel33),
    .frame_sent(red_frame_sent)  // indicate one frame has been sent
);

conv_unit CONV_RED(
    .pixel_clk(clk25),
    .rst(rst),
    .enable(vga_data_en),
    .kernel11(red_kernel11),
    .kernel12(red_kernel12),
```

```
    .kernel13(red_kernel13),
    .kernel21(red_kernel21),
    .kernel22(red_kernel22),
    .kernel23(red_kernel23),
    .kernel31(red_kernel31),
    .kernel32(red_kernel32),
    .kernel33(red_kernel33),
    .pixel11( {1'b0, red_pixel11} ),
    .pixel12( {1'b0, red_pixel12} ),
    .pixel13( {1'b0, red_pixel13} ),
    .pixel21( {1'b0, red_pixel21} ),
    .pixel22( {1'b0, red_pixel22} ),
    .pixel23( {1'b0, red_pixel23} ),
    .pixel31( {1'b0, red_pixel31} ),
    .pixel32( {1'b0, red_pixel32} ),
    .pixel33( {1'b0, red_pixel33} ),
    .pixel_out(VGA_R)
);


//************************************************************ \\
//******************** GREEN CHANNEL  ********************** \\
//************************************************************ \\
wire [16:0] green_addr;
wire [11:0] green_data;

wire [3:0] green_kernel11;
wire [3:0] green_kernel12;
wire [3:0] green_kernel13;
wire [3:0] green_kernel21;
wire [3:0] green_kernel22;
wire [3:0] green_kernel23;
wire [3:0] green_kernel31;
wire [3:0] green_kernel32;
wire [3:0] green_kernel33;

wire [3:0] green_pixel11;
wire [3:0] green_pixel12;
wire [3:0] green_pixel13;
wire [3:0] green_pixel21;
wire [3:0] green_pixel22;
wire [3:0] green_pixel23;
wire [3:0] green_pixel31;
wire [3:0] green_pixel32;
wire [3:0] green_pixel33;

wire green_frame_sent;

bram_green MEM_GREEN(
  .clka(clk25),          // input wire clka
  .wea(1'b0),            // input wire [0 : 0] wea
  .addra(green_addr),    // input wire [16 : 0] addra
  .dina(),               // input wire [11 : 0] dina
  .douta(green_data)     // output wire [11 : 0] douta
);

controller CNT_GREEN(
    .pixel_clk(clk25),
    .rst(rst),
    .data_en(vga_data_en),
    .data_in(green_data),
    .address(green_addr),
    .kernel11(green_kernel11),
    .kernel12(green_kernel12),
    .kernel13(green_kernel13),
    .kernel21(green_kernel21),
    .kernel22(green_kernel22),
    .kernel23(green_kernel23),
    .kernel31(green_kernel31),
    .kernel32(green_kernel32),
    .kernel33(green_kernel33),
    .pixel11(green_pixel11),
    .pixel12(green_pixel12),
    .pixel13(green_pixel13),
    .pixel21(green_pixel21),
    .pixel22(green_pixel22),
    .pixel23(green_pixel23),
```

```
    .pixel31(green_pixel31),
    .pixel32(green_pixel32),
    .pixel33(green_pixel33),
    .frame_sent(green_frame_sent)  // indicate one frame has been sent
);

conv_unit CONV_GREEN(
    .pixel_clk(clk25),
    .rst(rst),
    .enable(vga_data_en),
    .kernel11(green_kernel11),
    .kernel12(green_kernel12),
    .kernel13(green_kernel13),
    .kernel21(green_kernel21),
    .kernel22(green_kernel22),
    .kernel23(green_kernel23),
    .kernel31(green_kernel31),
    .kernel32(green_kernel32),
    .kernel33(green_kernel33),
    .pixel11( {1'b0, green_pixel11} ),
    .pixel12( {1'b0, green_pixel12} ),
    .pixel13( {1'b0, green_pixel13} ),
    .pixel21( {1'b0, green_pixel21} ),
    .pixel22( {1'b0, green_pixel22} ),
    .pixel23( {1'b0, green_pixel23} ),
    .pixel31( {1'b0, green_pixel31} ),
    .pixel32( {1'b0, green_pixel32} ),
    .pixel33( {1'b0, green_pixel33} ),
    .pixel_out(VGA_G)
);

//************************************************************ \\
//******************** BLUE CHANNEL  ************************ \\
//************************************************************ \\
wire [16:0] blue_addr;
wire [11:0] blue_data;

wire [3:0] blue_kernel11;
wire [3:0] blue_kernel12;
wire [3:0] blue_kernel13;
wire [3:0] blue_kernel21;
wire [3:0] blue_kernel22;
wire [3:0] blue_kernel23;
wire [3:0] blue_kernel31;
wire [3:0] blue_kernel32;
wire [3:0] blue_kernel33;

wire [3:0] blue_pixel11;
wire [3:0] blue_pixel12;
wire [3:0] blue_pixel13;
wire [3:0] blue_pixel21;
wire [3:0] blue_pixel22;
wire [3:0] blue_pixel23;
wire [3:0] blue_pixel31;
wire [3:0] blue_pixel32;
wire [3:0] blue_pixel33;

wire blue_frame_sent;

bram_blue MEM_BLUE (
  .clka(clk25),        // input wire clka
  .wea(1'b0),          // input wire [0 : 0] wea
  .addra(blue_addr),   // input wire [16 : 0] addra
  .dina(),             // input wire [11 : 0] dina
  .douta(blue_data)    // output wire [11 : 0] douta
);

controller CNT_BLUE(
    .pixel_clk(clk25),
    .rst(rst),
    .data_en(vga_data_en),
    .data_in(blue_data),
    .address(blue_addr),
    .kernel11(blue_kernel11),
    .kernel12(blue_kernel12),
    .kernel13(blue_kernel13),
    .kernel21(blue_kernel21),
```

```
        .kernel22(blue_kernel22),
        .kernel23(blue_kernel23),
        .kernel31(blue_kernel31),
        .kernel32(blue_kernel32),
        .kernel33(blue_kernel33),
        .pixel11(blue_pixel11),
        .pixel12(blue_pixel12),
        .pixel13(blue_pixel13),
        .pixel21(blue_pixel21),
        .pixel22(blue_pixel22),
        .pixel23(blue_pixel23),
        .pixel31(blue_pixel31),
        .pixel32(blue_pixel32),
        .pixel33(blue_pixel33),
        .frame_sent(blue_frame_sent)   // indicate one frame has been sent
    );

    conv_unit CONV_BLUE(
        .pixel_clk(clk25),
        .rst(rst),
        .enable(vga_data_en),
        .kernel11(blue_kernel11),
        .kernel12(blue_kernel12),
        .kernel13(blue_kernel13),
        .kernel21(blue_kernel21),
        .kernel22(blue_kernel22),
        .kernel23(blue_kernel23),
        .kernel31(blue_kernel31),
        .kernel32(blue_kernel32),
        .kernel33(blue_kernel33),
        .pixel11( {1'b0, blue_pixel11} ),
        .pixel12( {1'b0, blue_pixel12} ),
        .pixel13( {1'b0, blue_pixel13} ),
        .pixel21( {1'b0, blue_pixel21} ),
        .pixel22( {1'b0, blue_pixel22} ),
        .pixel23( {1'b0, blue_pixel23} ),
        .pixel31( {1'b0, blue_pixel31} ),
        .pixel32( {1'b0, blue_pixel32} ),
        .pixel33( {1'b0, blue_pixel33} ),
        .pixel_out(VGA_B)
    );

endmodule
```

Figure 5.1: Verilog Code of the Top Module

5.2 Testbench Code

```
`timescale 1ns / 1ps

module top_tb();

    reg  clk   = 0;
    reg  clk25 = 0;
    reg  rst   = 0;
    wire VGA_HS;
    wire VGA_VS;
    wire [3:0] VGA_R;
    wire [3:0] VGA_G;
    wire [3:0] VGA_B;

    top TOP_SYSTEM(
        .clk(clk),
        .rst(rst),
        .VGA_HS(VGA_HS),
        .VGA_VS(VGA_VS),
        .VGA_R(VGA_R),
        .VGA_G(VGA_G),
        .VGA_B(VGA_B)
    );

    always #5  clk   = ~clk;
    always #20 clk25 = ~clk25;


    parameter REFRESH = 416667;     // 1/60 second  refresh rate
    integer i   = 0;
```

```
    integer cnt = 0;

    integer fdr, fdg, fdb;

    initial
    begin
        fdr = $fopen("red.txt",   "w");
        fdg = $fopen("green.txt", "w");
        fdb = $fopen("blue.txt",  "w");

        repeat (200) @(posedge clk25);
        rst = 1;
        repeat (200) @(posedge clk25);
        rst = 0;

        // Write the output pixel values
        for( i = 0; i < REFRESH; i=i+1)
        begin
            @(posedge clk25);

            if(TOP_SYSTEM.vga_data_en)
            begin
                $fwrite(fdr, "%h", VGA_R);
                $fwrite(fdg, "%h", VGA_G);
                $fwrite(fdb, "%h", VGA_B);
                cnt = cnt + 1;
                if(cnt == 640)
                begin
                    $fwrite(fdr, "\n");
                    $fwrite(fdg, "\n");
                    $fwrite(fdb, "\n");
                    cnt = 0;
                end
            end
        end
        $fclose(fdr);
        $fclose(fdg);
        $fclose(fdb);
    end

endmodule
```

Figure 5.2: Testbench Code of the Top Module

This testbench (top_tb) simulates the entire top-level digital system and captures the resulting VGA video output into text files for image verification.

System Initialization: Generates the system clock (clk) and pixel clock (clk25), and applies a reset sequence to initialize the design.

Frame Simulation: Runs the simulation for 416,667 clock cycles, which corresponds to approximately 16.6 ms (one full video frame at 60Hz).

Data Logging: Monitors the internal vga_data_en signal. When active, it writes the Red, Green, and Blue pixel values into three separate text files (red.txt, green.txt, blue.txt).

Image Formatting: It counts valid pixels and inserts a newline character (\n) every 640 pixels. This formats the text files as a 2D matrix, allowing the output to be easily reconstructed into an image for visual inspection.

5.3 Simulation Output

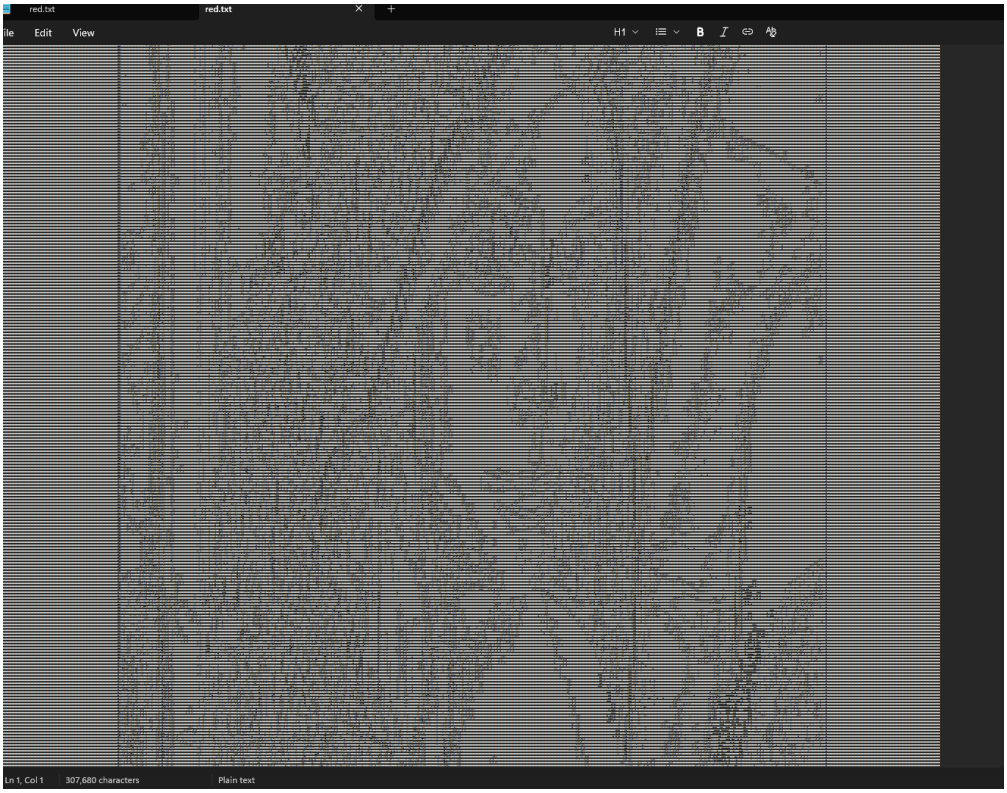

Figure 5.3.1: red.txt Output from the Top Module

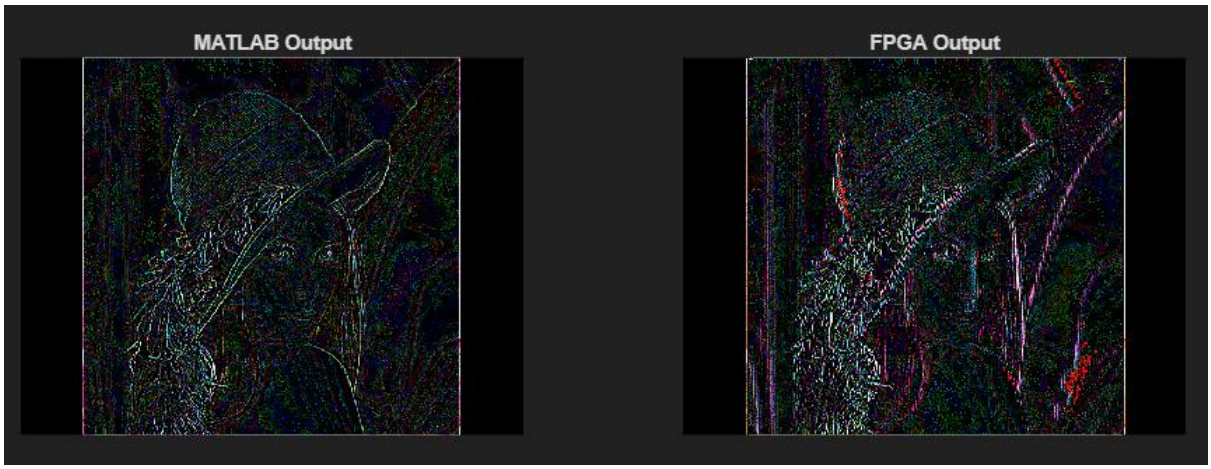

Figure 5.3.2: Visual comparison between the MATLAB and the FPGA Output

5.4 Utilization Report

Resource	Utilization	Available	Utilization %
LUT	6891	63400	10.87
FF	15861	126800	12.51
BRAM	106.50	135	78.89
IO	16	210	7.62
MMCM	1	6	16.67

Figure 5.4: Utilization Report

LUTs: The logic consumption is very low. The convolution unit, despite having 9 multiplications, uses few resources because the multiplications are by constants (1 and -8). The synthesizer optimizes multiplication by 8

into a simple bit-shift operation (left shift by 3), eliminating the need for complex multiplier logic or DSP slices. The primary LUT usage comes from the adder trees and the FSM control logic.

BRAM: This is the dominant resource. Storing a 640x480 image (even at 4-bit depth) requires significant memory. The design uses independent BRAMs for R, G, and B channels to allow parallel access. This trades off memory density for throughput performance.

Timing: The design easily met the 25 MHz timing constraint. The critical path is likely within the adder tree of the convolution unit, but at 40ns period, there is substantial positive slack (>30ns), indicating the design could potentially run much faster (e.g., for 720p or 1080p resolutions) with modified clocking.