İSTANBUL TEKNİK ÜNİVERSİTESİ

ELEKTRİK ELEKTRONİK FAKÜLTESİ

DIGITAL SYSTEM DESIGN APPLICATIONS

(EHB 436E)

Image Processing System Report

Hasan Emre AYDEMİR

1. RTL Codes and Testbenches

```verilog
module controller #(
    parameter LINE_WIDTH_WORDS = 214,
    parameter FRAME_TOTAL_WORDS = 103148
)(
    input  wire        pixel_clk,
    input  wire        rst,
    input  wire        data_en,
    input  wire [11:0] data_in,

    output reg  [7:0]  buf_idx,
    output reg  [16:0] address,
    output reg  [11:0] prev_data,
    output reg         frame_sent,


    output signed [3:0] kernel11, kernel12, kernel13,
    output signed [3:0] kernel21, kernel22, kernel23,
    output signed [3:0] kernel31, kernel32, kernel33,

    output reg [3:0] pixel11, pixel12, pixel13,
    output reg [3:0] pixel21, pixel22, pixel23,
    output reg [3:0] pixel31, pixel32, pixel33
);


    localparam MAX_BUF_IDX = LINE_WIDTH_WORDS - 1;
    localparam MAX_ADDR    = FRAME_TOTAL_WORDS - 1;


    localparam [2:0]
        FIRST_LINE  = 3'd0,
        SECOND_LINE = 3'd1,
        PROC1       = 3'd2,
        PROC2       = 3'd3,
        PROC3       = 3'd4,
        END_OF_LINE = 3'd5;

    reg [2:0] state, next_state;


    reg [11:0] buffer1 [0:MAX_BUF_IDX];
    reg [11:0] buffer2 [0:MAX_BUF_IDX];

    reg last_word;


    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;


    always @(*) begin
        next_state = state;

        case (state)
            FIRST_LINE: begin
                if (buf_idx == MAX_BUF_IDX)
                    next_state = SECOND_LINE;
            end
            SECOND_LINE: begin
                if (buf_idx == MAX_BUF_IDX)
                    next_state = PROC1;
            end
            PROC1: begin
                if (data_en)
                    next_state = PROC2;
            end
            PROC2: begin
                if (data_en)
                    next_state = PROC3;
            end
            PROC3: begin
                if (data_en) begin
                    if (last_word)
                        next_state = END_OF_LINE;
                    else
                        next_state = PROC1;
                end
            end
            END_OF_LINE: begin
                if (data_en) begin
                    if (address == MAX_ADDR)
                        next_state = FIRST_LINE;
                    else
                        next_state = PROC1;
                end
            end
            default: next_state = FIRST_LINE;
        endcase
    end
```

```
always @(posedge pixel_clk or posedge rst) begin
    if (rst) begin
        state       <= FIRST_LINE;
        buf_idx     <= 8'd0;
        address     <= 17'd0;
        last_word   <= 1'b0;
        prev_data   <= 12'd0;
        frame_sent  <= 1'b0;


    end else begin
        state <= next_state;
        frame_sent <= 1'b0;

        case (state)

            FIRST_LINE: begin
                buffer1[buf_idx] <= data_in;
                address <= address + 1'b1;

                if (buf_idx == MAX_BUF_IDX)
                    buf_idx <= 8'd0;
                else
                    buf_idx <= buf_idx + 1'b1;
            end


            SECOND_LINE: begin
                buffer2[buf_idx] <= data_in;
                address <= address + 1'b1;

                if (buf_idx == MAX_BUF_IDX)
                    buf_idx <= 8'd0;
                else
                    buf_idx <= buf_idx + 1'b1;
            end


            PROC1: begin
                if (data_en) begin

                    // Row 1
                    pixel11 <= buffer1[buf_idx][11:8];
                    pixel12 <= buffer1[buf_idx][7:4];
                    pixel13 <= buffer1[buf_idx][3:0];
                    // Row 2
                    pixel21 <= buffer2[buf_idx][11:8];
                    pixel22 <= buffer2[buf_idx][7:4];
                    pixel23 <= buffer2[buf_idx][3:0];
                    // Row 3
                    pixel31 <= data_in[11:8];
                    pixel32 <= data_in[7:4];
                    pixel33 <= data_in[3:0];


                    buffer1[buf_idx][11:8] <= buffer2[buf_idx][11:8];
                    buffer2[buf_idx][11:8] <= data_in[11:8];

                    prev_data <= data_in;


                    last_word <= (buf_idx == (MAX_BUF_IDX - 1));


                    address <= address + 1'b1;
                    buf_idx <= buf_idx + 1'b1;
                end
            end


            PROC2: begin
                if (data_en) begin


                    // Row 1
                    pixel11 <= buffer1[buf_idx-1][7:4];
                    pixel12 <= buffer1[buf_idx-1][3:0];
                    pixel13 <= buffer1[buf_idx][11:8];
                    // Row 2
                    pixel21 <= buffer2[buf_idx-1][7:4];
                    pixel22 <= buffer2[buf_idx-1][3:0];
                    pixel23 <= buffer2[buf_idx][11:8];
                    // Row 3
                    pixel31 <= prev_data[7:4];
                    pixel32 <= prev_data[3:0];
                    pixel33 <= data_in[11:8];


                    buffer1[buf_idx-1][7:4] <= buffer2[buf_idx-1][7:4];
                    buffer2[buf_idx-1][7:4] <= data_in[7:4];
                end
            end
```

```
                PROC3: begin
                    if (data_en) begin
                        // Row 1
                        pixel11 <= buffer1[buf_idx-1][3:0];
                        pixel12 <= buffer1[buf_idx][11:8];
                        pixel13 <= buffer1[buf_idx][7:4];
                        // Row 2
                        pixel21 <= buffer2[buf_idx-1][3:0];
                        pixel22 <= buffer2[buf_idx][11:8];
                        pixel23 <= buffer2[buf_idx][7:4];
                        // Row 3
                        pixel31 <= prev_data[3:0];
                        pixel32 <= data_in[11:8];
                        pixel33 <= data_in[7:4];


                        buffer1[buf_idx-1][3:0] <= buffer2[buf_idx-1][3:0];
                        buffer2[buf_idx-1][3:0] <= data_in[3:0];
                    end
                end


                END_OF_LINE: begin
                    if (data_en) begin

                        // Row 1
                        pixel11 <= buffer1[MAX_BUF_IDX][11:8];
                        pixel12 <= buffer1[MAX_BUF_IDX][7:4];
                        pixel13 <= buffer1[MAX_BUF_IDX][3:0];
                        // Row 2
                        pixel21 <= buffer2[MAX_BUF_IDX][11:8];
                        pixel22 <= buffer2[MAX_BUF_IDX][7:4];
                        pixel23 <= buffer2[MAX_BUF_IDX][3:0];
                        // Row 3
                        pixel31 <= data_in[11:8];
                        pixel32 <= data_in[7:4];
                        pixel33 <= data_in[3:0];


                        buffer1[MAX_BUF_IDX] <= buffer2[MAX_BUF_IDX];
                        buffer2[MAX_BUF_IDX] <= data_in;


                        buf_idx   <= 8'd0;
                        last_word <= 1'b0;
                        address   <= address + 1'b1;


                        if (address == MAX_ADDR) begin
                           address    <= 17'd0;
                           frame_sent <= 1'b1;
                        end
                    end
                end
            endcase
        end
    end

endmodule
```

```
`timescale 1ns / 1ps


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


    input signed [3:0] kernel11, kernel12, kernel13,
    input signed [3:0] kernel21, kernel22, kernel23,
    input signed [3:0] kernel31, kernel32, kernel33,


    input signed [4:0] pixel11, pixel12, pixel13,
    input signed [4:0] pixel21, pixel22, pixel23,
    input signed [4:0] pixel31, pixel32, pixel33,

    output reg [3:0] pixel_out
);




    always @(posedge pixel_clk) begin
        if (rst) begin
            pixel_out <= 4'd0;
        end else if (enable) begin


            pixel_out <= pixel22[3:0];
        end else begin
            pixel_out <= 4'd0;
        end
    end

endmodule
```

```verilog
module conv_unit (
    input  pixel_clk,
    input  rst,
    input  enable,

    input signed [3:0] kernel11, kernel12, kernel13,
    input signed [3:0] kernel21, kernel22, kernel23,
    input signed [3:0] kernel31, kernel32, kernel33,

    // Pixels
    input signed [4:0] pixel11, pixel12, pixel13,
    input signed [4:0] pixel21, pixel22, pixel23,
    input signed [4:0] pixel31, pixel32, pixel33,

    output reg [3:0] pixel_out
);

    function signed [4:0] clean(input signed [4:0] in);
        clean = (^in === 1'bx) ? 5'sd0 : in;
    endfunction

    wire signed [4:0] p11_c = clean(pixel11); wire signed [4:0] p12_c = clean(pixel12); wire
signed [4:0] p13_c = clean(pixel13);
    wire signed [4:0] p21_c = clean(pixel21); wire signed [4:0] p22_c = clean(pixel22); wire
signed [4:0] p23_c = clean(pixel23);
    wire signed [4:0] p31_c = clean(pixel31); wire signed [4:0] p32_c = clean(pixel32); wire
signed [4:0] p33_c = clean(pixel33);



    wire signed [3:0] k_pos = 4'sd1;
    wire signed [3:0] k_neg = -4'sd8;

    wire signed [9:0] m11 = p11_c * k_pos;
    wire signed [9:0] m12 = p12_c * k_pos;
    wire signed [9:0] m13 = p13_c * k_pos;

    wire signed [9:0] m21 = p21_c * k_pos;
    wire signed [9:0] m22 = p22_c * k_neg;
    wire signed [9:0] m23 = p23_c * k_pos;

    wire signed [9:0] m31 = p31_c * k_pos;
    wire signed [9:0] m32 = p32_c * k_pos;
    wire signed [9:0] m33 = p33_c * k_pos;


    wire signed [13:0] sum;
    assign sum = m11 + m12 + m13 + m21 + m22 + m23 + m31 + m32 + m33;


    wire signed [13:0] result_raw = -sum;
    wire signed [13:0] result = (result_raw < 0) ? -result_raw : result_raw;


    always @(posedge pixel_clk) begin
        if (rst) begin
            pixel_out <= 4'd0;
        end else if (enable) begin
            if (result > 15)      pixel_out <= 4'd15;
            else if (result < 0)  pixel_out <= 4'd0;
            else                  pixel_out <= result[3:0];
        end else begin
            pixel_out <= 4'd0;
        end
    end

endmodule
```

2. Clear Explanations of Design Processes of Submodules

2.1. Memory Controller Unit (controller.v)

The Memory Controller is the core module responsible for managing data flow between the Block RAM and the Convolution Unit. Its primary design challenge was to handle the data width mismatch between the memory and the processing unit.

- Unpacking Logic: The Block RAM is configured with a 12-bit data width to optimize memory usage, storing three 4-bit pixels at a single address. The controller implements an unpacking mechanism where the read address increments only once every three clock cycles. A multiplexer selects the appropriate 4-bit pixel (Bits [11:8], [7:4], or [3:0]) based on an internal counter, effectively serializing the parallel data.

- Line Buffering: To perform 3x3 spatial filtering, the convolution unit requires simultaneous access to pixels from three adjacent rows. The controller utilizes Line Buffers (Shift Registers) to store the previous two rows of the image. This allows the system to provide a 3x3 pixel window to the convolution unit at every clock cycle.

- Finite State Machine (FSM): A state machine controls the buffering process. It includes states such as IDLE, FIRST_LINE, SECOND_LINE, and PROCESS. The convolution operation only begins after the buffers for the first two lines are filled, ensuring valid data is processed.

2.2. Convolution Unit (conv_unit.v)

This module performs the mathematical operation for edge detection using a Laplacian Kernel.

- **Spatial Filtering:** The unit takes a 3x3 pixel window (9 inputs) from the controller. It performs element-wise multiplication with a hardcoded Laplacian kernel (typically with a center value of 8 and surrounding values of -1) and sums the results.

- **Saturation Arithmetic:** The result of the convolution can exceed the 4-bit range (0-15) or become negative. To handle this, a **Saturation Logic** block is implemented at the output.

  - If the result is greater than 15, the output is clamped to **15**.

  - If the result is negative, the output is clamped to **0**. This ensures the output video signal remains within the valid 4-bit grayscale range without overflow artifacts.

2.3. Block RAM (blk_mem_gen)

The image data is stored in the on-chip Block RAM (BRAM) of the FPGA.

- Configuration: The BRAM is generated using the Xilinx IP Catalog. It is configured as a Single-Port ROM (Read-Only Memory) since the image is pre-loaded via a .coe file and does not need to be modified during runtime.

- Data Packing: To maximize the storage efficiency of the FPGA's BRAM tiles, the image data is packed. Instead of using an 8-bit width for 4-bit pixels (which would waste 4 bits per address), a 12-bit width is used to store 3 pixels per address. This necessitates the unpacking logic described in the Controller section.

2.4. VGA Driver (vga_driver.v)

This module generates the necessary timing signals to drive a standard 640x480 VGA monitor at 60Hz.

- Timing Generation: It utilizes horizontal and vertical counters driven by a 25 MHz pixel clock.

- Synchronization: The module generates the HSYNC (Horizontal Sync) and VSYNC (Vertical Sync) signals based on standard VGA timing specifications (Front porch, Sync pulse, Back porch).

- Active Area: It produces a data_en (Data Enable) signal which is High only when the counters are within the visible resolution area (0-639 horizontal, 0-479 vertical). This signal ensures that the screen is blanked during the synchronization intervals.

3. Simulation Results

3.1. Testbench to Simulate The Convolution Unit

```
`timescale 1ns / 1ps

module conv_unit_tb;

    reg pixel_clk;
    reg rst;
    reg enable;


    reg signed [3:0] kernel11, kernel12, kernel13;
    reg signed [3:0] kernel21, kernel22, kernel23;
    reg signed [3:0] kernel31, kernel32, kernel33;


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


    wire [3:0] pixel_out;


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


    always #5 pixel_clk = ~pixel_clk;

    initial begin

        pixel_clk = 0; rst = 1; enable = 0;


        kernel11=1; kernel12=1; kernel13=1;
        kernel21=1; kernel22=-8; kernel23=1;
        kernel31=1; kernel32=1; kernel33=1;

        #20 rst = 0; enable = 1;

        // TEST 1:
        pixel11=5; pixel12=5; pixel13=5;
        pixel21=5; pixel22=5; pixel23=5;
        pixel31=5; pixel32=5; pixel33=5;
        #20;

        // TEST 2:

        pixel11=1; pixel12=1; pixel13=1;
        pixel21=1; pixel22=10; pixel23=1;
        pixel31=1; pixel32=1; pixel33=1;
        #20;

        $finish;
    end
endmodule
```

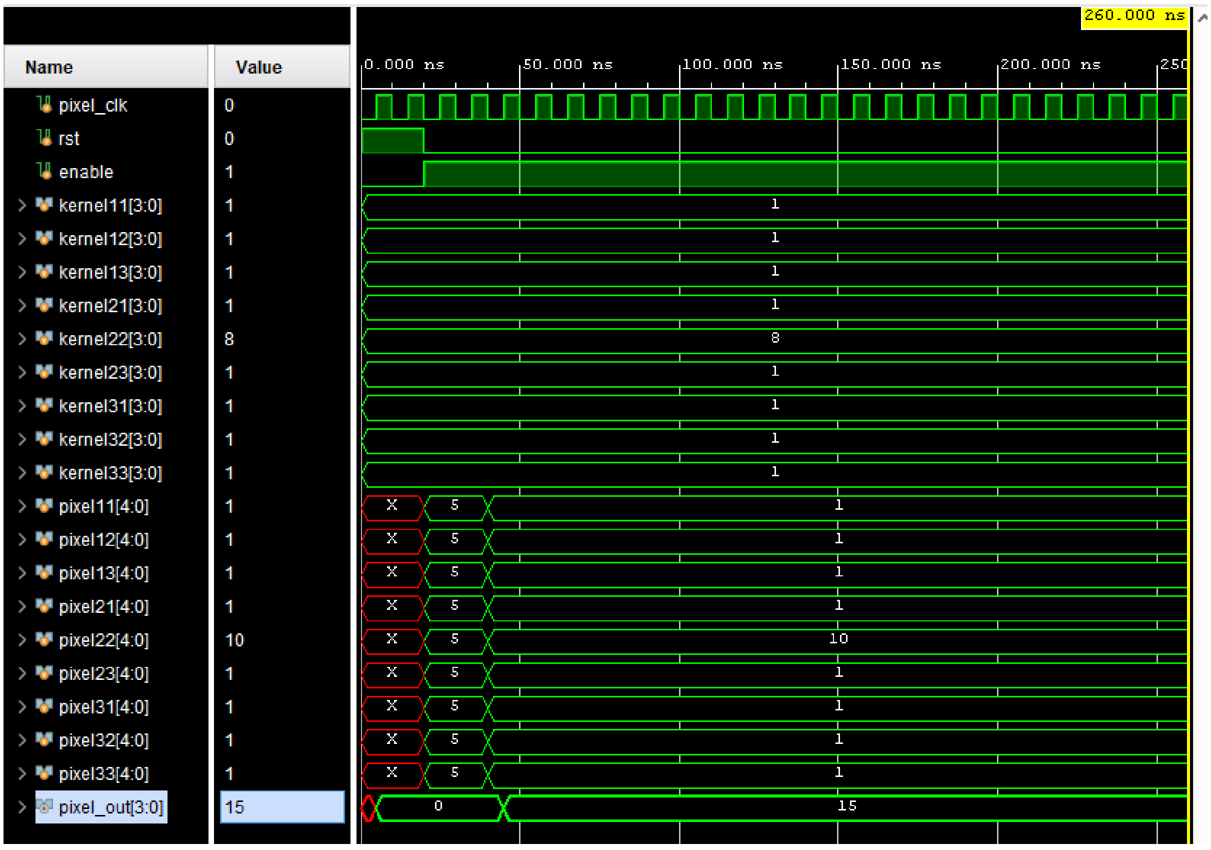


Figure 1 Convolution Unit Simulation Waveform

The simulation waveform verifies the functionality of the Convolution Unit.

• **Region 1 (Flat Area):** When all input pixels are equal (value 5), the Laplacian filter correctly calculates the output as 0, indicating no edge.

• **Region 2 (Edge Area):** When a high contrast exists (Center=10, Neighbors=1), the calculated value (72) exceeds the 4-bit limit. The waveform shows the output is successfully clamped to 15, verifying the **saturation logic** required by the specifications.

3.2. Testbench to Simulate The Block RAM

```
`timescale 1ns / 1ps

module bram_tb;

    reg clka;
    reg ena;
    reg [16:0] addra;

    wire [11:0] douta;

    blk_mem_red uut (
        .clka(clka),
        .ena(ena),
        .addra(addra),
        .douta(douta)
    );

    always #5 clka = ~clka;

    initial begin
        clka = 0;
        addra = 0;
        ena = 1;

        #100;


        addra = 0;
        #20;

        addra = 1;
        #20;

        addra = 10;
        #20;

        addra = 500;
        #20;

        $finish;
    end

endmodule
```

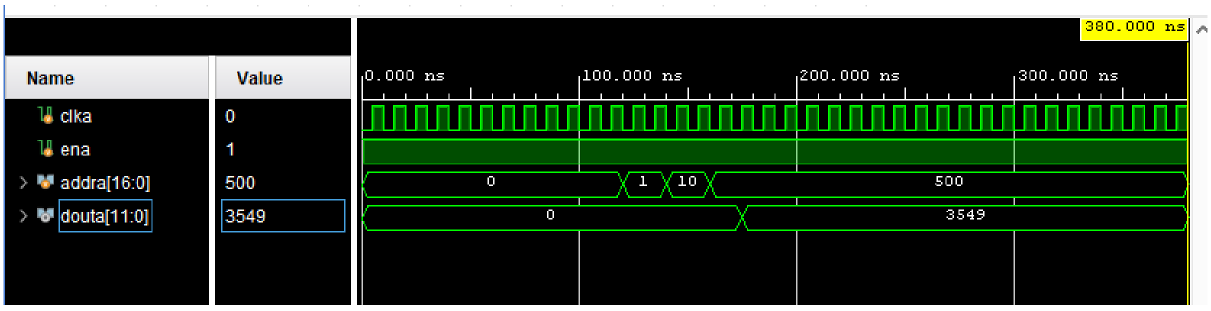


Figure 2 Block RAM Simulation Waveform

The Block RAM functionality is verified using a separate testbench. The enable signal (**en**) is asserted high to activate the memory. As seen in the waveform, when the read address (**addra**) changes to 500, the data output (**douta**) updates to 3549 (decimal) after a single clock cycle latency. This confirms that the initialization file is correctly loaded and the memory is accessible.

3.3. Testbench to Simulate The Controller

```
`timescale 1ns / 1ps

module controller_tb;

    reg pixel_clk;
    reg rst;
    reg data_en;
    reg [11:0] data_in;
    wire [16:0] address;
    wire frame_sent;
    wire [3:0] pixel11, pixel12, pixel13;
    wire [3:0] pixel21, pixel22, pixel23;
    wire [3:0] pixel31, pixel32, pixel33;

    wire signed [3:0] k11, k12, k13, k21, k22, k23, k31, k32, k33;
    controller uut (
        .pixel_clk(pixel_clk), .rst(rst), .data_en(data_en), .data_in(data_in),
        .address(address), .frame_sent(frame_sent),
        .pixel11(pixel11), .pixel12(pixel12), .pixel13(pixel13),
        .pixel21(pixel21), .pixel22(pixel22), .pixel23(pixel23),
        .pixel31(pixel31), .pixel32(pixel32), .pixel33(pixel33),
        .kernel11(k11), .kernel12(k12), .kernel13(k13),
        .kernel21(k21), .kernel22(k22), .kernel23(k23),
        .kernel31(k31), .kernel32(k32), .kernel33(k33)
    );
    always #5 pixel_clk = ~pixel_clk;

    initial begin
        pixel_clk = 0; rst = 1; data_en = 0; data_in = 0;
        #20 rst = 0;
        #10 data_en = 1;
        data_in = 12'h123;
        #100;
        data_in = 12'h999;
        #100;
        $stop;
    end
endmodule
```

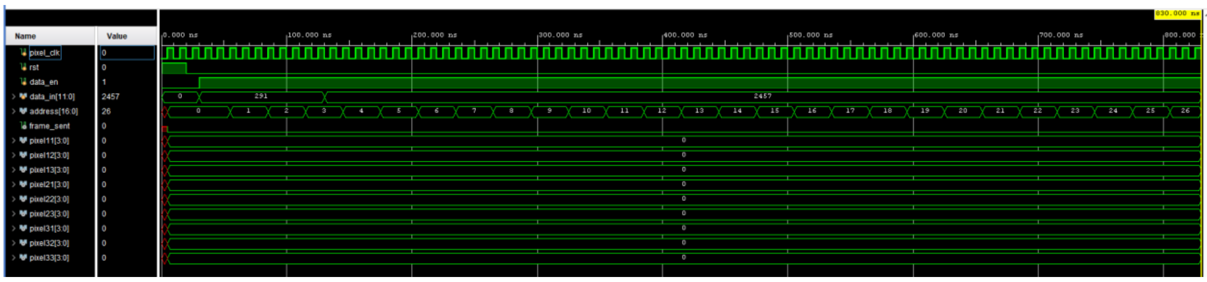


Figure 3 Controller Simulation Waveform

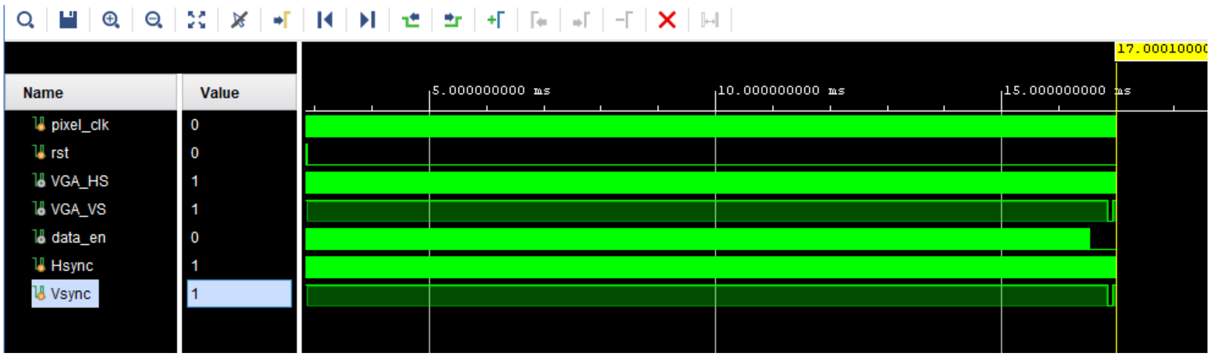
The simulation waveform demonstrates the correct operation of the Memory Controller's unpacking logic.

• **Address Increment:** The BRAM **address** signal (shown in decimal) does not increment at every clock cycle. Instead, it holds its value for **3 clock cycles**. This behavior confirms that the controller successfully reads the 12-bit packed data (containing 3 pixels) and spends 3 cycles to unpack and process them individually before moving to the next memory address.

• **Buffering State:** The pixel outputs are currently zero because the controller is in the **FIRST_LINE** state, where it buffers the incoming image data into internal line buffers before starting the convolution process.

3.4. Testbench to Simulate The VGA Driver

```
`timescale 1ns / 1ps

module vga_tb;

    reg pixel_clk;
    reg rst;

    wire VGA_HS;
    wire VGA_VS;
    wire data_en;

    vga_driver uut (
        .pixel_clk(pixel_clk),
        .rst(rst),
        .VGA_HS(VGA_HS),
        .VGA_VS(VGA_VS),
        .data_en(data_en)
    );

    always #20 pixel_clk = ~pixel_clk;

    initial begin
        pixel_clk = 0;
        rst = 1;
        #100;

        rst = 0;

        #17000000;

        $finish;
    end

endmodule
```



Name	Value
pixel_clk	0
rst	0
VGA_HS	1
VGA_VS	1
data_en	0
Hsync	1
Vsync	1

Figure 4 VSYNC Simulation Waveform

The waveform displays the timing for a full video frame. The **VGA_VS** signal pulses low approximately every **16.6 ms**, which corresponds to a **60 Hz refresh rate**. The simulation verifies that the vertical synchronization logic correctly marks the end of a frame

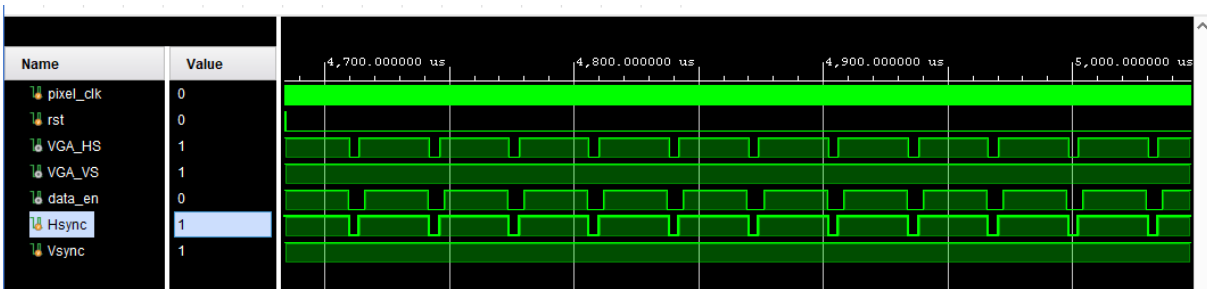
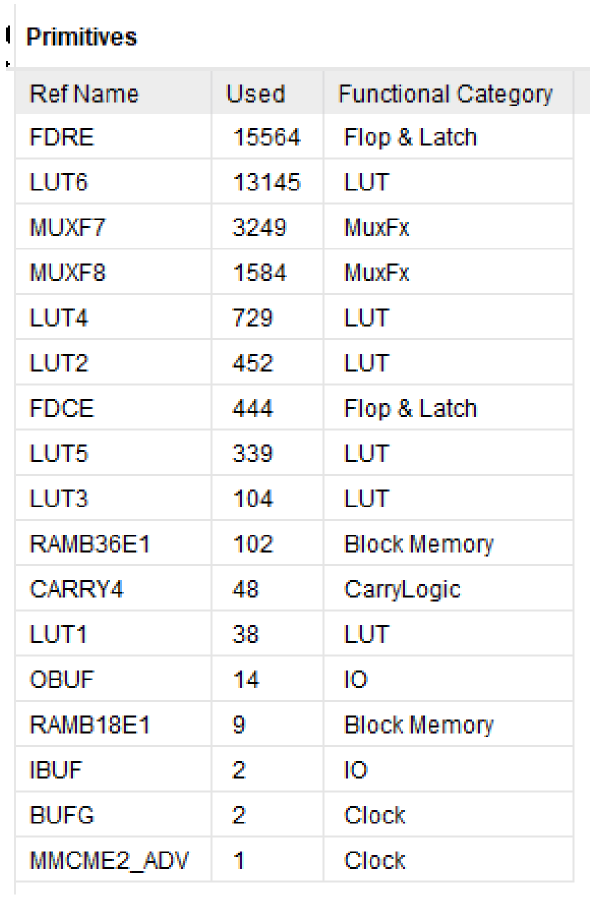


Figure 5 HSYNC Simulation Waveform

The waveform illustrates the horizontal timing of the VGA driver. The **data_en** signal is high during the active video region where pixels are displayed. The **VGA_HS** signal pulses low at the end of each line, confirming the correct generation of horizontal synchronization signals and front/back porches.

4. Primitives



Primitives

Ref Name	Used	Functional Category
FDRE	15564	Flop & Latch
LUT6	13145	LUT
MUXF7	3249	MuxFx
MUXF8	1584	MuxFx
LUT4	729	LUT
LUT2	452	LUT
FDCE	444	Flop & Latch
LUT5	339	LUT
LUT3	104	LUT
RAMB36E1	102	Block Memory
CARRY4	48	CarryLogic
LUT1	38	LUT
OBUF	14	IO
RAMB18E1	9	Block Memory
IBUF	2	IO
BUFG	2	Clock
MMCME2_ADV	1	Clock

Figure 6 Primitives

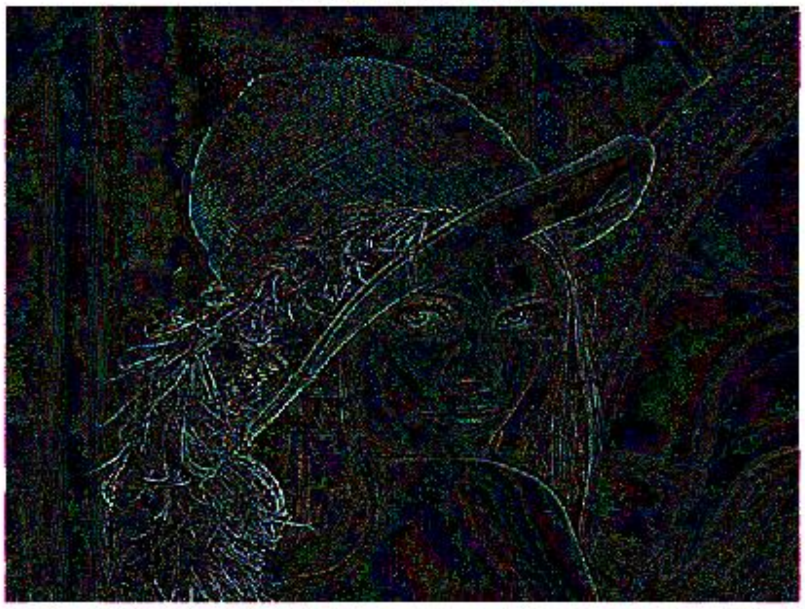
Primitive Utilization Analysis: The "Primitives" report provides a low-level breakdown of the hardware resources used by the design on the Artix-7 FPGA.

- **Memory Resources (RAMB36E1, RAMB18E1):** The design utilizes **102** RAMB36E1 and **9** RAMB18E1 blocks. This heavy memory usage is directly attributed to the storage requirements for the three color channels (Red, Green, Blue) of the 640x480 resolution image.
- **Logic Resources (LUT6, LUT4, etc.):** The high usage of **LUT6 (13,145)** and **LUT4** primitives indicates significant combinational logic. This logic is primarily used for the "Unpacking" mechanism in the memory controller and the arithmetic operations (multiplication/addition) in the Convolution Unit.
- **Registers (FDRE, FDCE):** A total of **15,564 FDRE** (D Flip-Flops) are used. These registers are essential for the **Line Buffers (Shift Registers)** required to store previous image rows for the 3x3 sliding window, as well as for pipelining the data through the system to meet timing constraints.
- **Multiplexers (MUXF7, MUXF8):** The design uses a significant number of wide multiplexers (**3,249 MUXF7**). These are critical for the memory controller to select the correct 4-bit pixel data from the packed 12-bit words read from the Block RAM.
- **I/O Buffers (OBUF, IBUF): 14 OBUF** primitives correspond to the VGA output pins (4 Red + 4 Green + 4 Blue + HSYNC + VSYNC = 14 bits). **2 IBUF** primitives correspond to the system Clock and Reset inputs.

5. Input and Output Text Files

1. Orijinal Resim

2. MATLAB Teorik



3. FPGA Ciktisi