Basic Image Processing Using FPGA on DE10 Board

EC806 Project Report

Submitted by:
Nishchay Pallav 221EC233
Mohammad Omar Sulemani 221EC230
Md Atib Kaif 221EC129

Under the Guidance of:
Professor Pathipati Shrihari

March 30, 2025
Department of Electronics and Communication Engineering
National Institute of Technology Karnataka Surathkal—575025

Abstract

This study examines the implementation of fundamental image processing algorithms on the DE10 FPGA board using Verilog HDL. The primary focus of this project includes Invertion operation, brightness adjustment, and thresholding using suitable Verilog modules. The FPGA will be used to process the input images in hexadecimal format. We will use the VGA interface to display the processed images on the monitor.

Contents

Chapter 1

Introduction

Image processing is essential in numerous applications, including medical imaging, surveillance, autonomous driving, and industrial automation. Traditionally, such applications are carried out by general-purpose processors or GPUs, but these platforms are poorly suited to real-time performance in embedded systems.

Field-Programmable Gate Arrays (FPGAs) offer a compelling alternative with their inherent parallelism, agility, and reduced latency capabilities.FPGAs allow engineers to create their own processing pipelines at the hardware level, resulting in longer and more efficient operation than implementations in software.

The objective of this project is to design and implement simple image processing algorithms using the assistance of the DE10 FPGA board and Verilog HDL. Elementary operations such as image invertion and brightness control are performed through the design of custom hardware modules. Input image data is fed in hexadecimal format, and an application-specific VGA controller has been designed to display the processed output on a screen.

Apart from algorithmic implementation, this project is a learning experience in FPGA architecture, Verilog design, and peripheral interfacing. It demonstrates the practical limitations and advantages of hardware image processing, and the FPGA's ability to offer real-time visual feedback.

Chapter 2

Literature Review

Presently, advances in digital image processing have triggered hardware implementation research to address the real-time requirements of embedded systems. FPGAs, with their parallel processing and reconfigurable nature, provide an appropriate platform for low-latency and high-speed implementation of image processing algorithms.

In the paper "Development of Digital Image Processing Algorithms via FPGA Implementation" by Shamsiah Suhaili et al. (2024), the authors used fundamental image processing operations such as conversion of images to grayscale, brightness and contrast adjustment, thresholding, and flipping of images in an efficient manner by using Verilog HDL on an FPGA platform. They converted input images to hexadecimal, applied the images by processing them through their own Verilog modules, and verified the output by using MATLAB. This research demonstrated that images can be enhanced in real-time using FPGA technology in an efficient manner with regard to processing time and resources.

Building on this idea, our project aims to implement some of these methods—grayscale conversion, brightness adjustment, and thresholding—on the DE10 FPGA board. Although Suhaili et al. focused on showing output through files, our implementation has a VGA interface to provide real-time visual output on a monitor, which offers a more interactive presentation of fast image processing.

Chapter 3

Project Plan

This project follows a structured approach to design, implement, and validate basic image processing operations on an FPGA platform. The key phases include requirement analysis, module design, Verilog implementation, simulation, hardware integration, and testing.

1. Methodology Overview

The methodology involves implementing three image processing operations—Invertion operation, brightness adjustment, and thresholding—using Verilog HDL. The image data is provided in hexadecimal format, processed in real time on the FPGA, and displayed via a VGA output.

2. Block Diagram

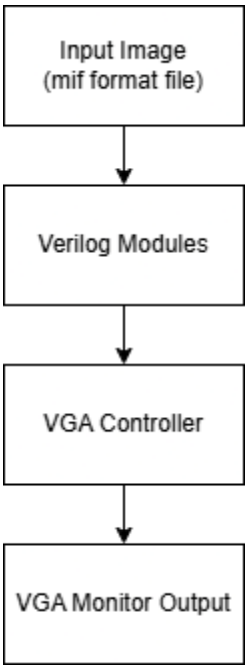

Figure: Functional block diagram of the image processing system.

3. Tools and Platforms

- **Hardware:** Intel DE10 FPGA Development Board
- **Software:**
  - Quartus Prime (for Verilog design and synthesis)
  - ModelSim (for simulation and verification)
  - Python (for converting images to hex format)
- **Peripheral:** VGA Monitor for output display

4. Implementation Steps

1. **Image Conversion**: Convert image to grayscale bitmap and generate corresponding hexadecimal data.
2. **Verilog Module Design**: Design individual Verilog modules for:
   - Invertion Operation
   - Brightness Control
   - Thresholding
3. **VGA Controller**: Develop a VGA interface to read processed image data and display it on a monitor.
4. **Simulation and Debugging**: Use ModelSim to verify the behavior of each module.
5. **Synthesis and Deployment**: Compile the project using Quartus Prime and deploy the bitstream to the DE10 board.
6. **Testing and Validation**: Connect VGA output and validate the image transformations in real time.

5. Timeline Summary

Week	Task
1	Literature review and requirement analysis
1	Image preprocessing and hex conversion
2	Design and simulation of invertion operation module
2	Design and integration of brightness and threshold modules
3	VGA display implementation and integration
3	Testing, debugging, and documentation

Chapter 4

Project Outcome

4.1 Image Processing Using Verilog

In this FPGA Verilog project, some simple processing operations are implemented in Verilog such as inversion, brightness control, and threshold operations. The image processing operation is selected by a `parameter.v` file and then, the processed image data are written to a bitmap image `output.bmp` for verification purposes.

The image reading Verilog code operates as a Verilog model of an image sensor or camera, which can be very helpful for functional verification in real-time FPGA image processing projects. The image writing part is also extremely useful for testing, especially when you want to view the output image in BMP format.

In this project, we added some simple image processing code into the reading part to serve as an example, but you can easily remove this to obtain raw image data.

Note: Verilog cannot read images directly. To read a `.bmp` image in Verilog, it must be converted from the bitmap format to a hexadecimal format. Below is a Python example code snippet to convert a bitmap image to a `.mem` file. The input image size is `768x512`, and the image `.mem` file includes R, G, B data of the bitmap image.

Python Code: bmp_to_mem.py

Below is the Python code that converts a BMP image to a memory file:

Listing 4.1: Python code to convert BMP to MEM

```
from PIL import Image

def bmp_to_mem(input_bmp, output_mem):
    # Open image and convert to RGB
    img = Image.open(input_bmp).convert("RGB")
    width, height = img.size
    print(f"Image size: {width}x{height}")

    pixels = list(img.getdata())

    # .mem file will contain pixel data in hex, each byte per
       line
    # Format: R G B R G B ...
    with open(output_mem, "w") as mem_file:
```

```
        for y in reversed(range(height)):  # BMP is bottom to top
            for x in range(width):
                r, g, b = pixels[y * width + x]
                mem_file.write(f"{r:02X}\n")
                mem_file.write(f"{g:02X}\n")
                mem_file.write(f"{b:02X}\n")

    print(f"Successfully written to {output_mem}")

# Example usage:
bmp_to_mem("gojo.bmp", "gojo.mem")
```

Verilog Code for Image Reading and Processing

To read the image hexadecimal data file, Verilog uses this command: `$readmemh` or `$readmemb` if the image data is in a binary text file. After reading the image `.mem` file, the RGB image data are saved into memory and read out for processing.

Below is the Verilog code for the image reading and processing part:

```
/*************************************************************/
/******  Module for reading and processing image     *********/
/*************************************************************/
`include "parameter.v"          // Include definition file
module image_read
#(
  parameter WIDTH       = 768,                  // Image width
        HEIGHT  = 512,            // Image height
        INFILE  = "gojo.mem", // image file
        START_UP_DELAY = 100, // Delay during start up time
        HSYNC_DELAY = 160,      // Delay between HSYNC pulses
        VALUE= 90,              // value for Brightness operation
        THRESHOLD= 200, // Threshold value for Threshold
           operation
        SIGN=1  // Sign value using for brightness operation
                        // SIGN = 0: Brightness subtraction
                        // SIGN = 1: Brightness addition
)
(
        input HCLK,             // clock

        input HRESETn,  // Reset (active low)
        output VSYNC,   // Vertical synchronous pulse
        // This signal is often a way to indicate that one entire
            image is transmitted.
        // Just create and is not used, will be used once a video
            or many images are transmitted.
        output reg HSYNC,// Horizontal synchronous pulse
        // An HSYNC indicates that one line of the image is
           transmitted.
```

```
        // Used to be a horizontal synchronous signals for
           writing bmp file.
    output reg [7:0]  DATA_R0,  // 8 bit Red data (even)
    output reg [7:0]  DATA_G0,  // 8 bit Green data (even)
    output reg [7:0]  DATA_B0,  // 8 bit Blue data (even)
    output reg [7:0]  DATA_R1,  // 8 bit Red  data (odd)
    output reg [7:0]  DATA_G1,  // 8 bit Green data (odd)
    output reg [7:0]  DATA_B1,  // 8 bit Blue data (odd)
        // Process and transmit 2 pixels in parallel to make the
           process faster, you can modify to transmit 1 pixels or
            more if needed
        output          ctrl_done               // Done flag
);
//------------------------------------------------
// Internal Signals
//------------------------------------------------

parameter sizeOfWidth = 8;              // data width
parameter sizeOfLengthReal = 1179648;   // image data : 1179648
   bytes: 512 * 768 *3
// local parameters for FSM
localparam              ST_IDLE         = 2'b00,
   // idle state
                        ST_VSYNC        = 2'b01,        // state
                           for creating vsync
                        ST_HSYNC        = 2'b10,        // state
                           for creating hsync
                        ST_DATA         = 2'b11;// state for data
                            processing
reg [1:0] cstate,                                       // current state
                  nstate;
                     // next state
reg start;   // start signal: trigger Finite state machine
              //beginning to operate
reg HRESETn_d;// delayed reset signal: use to create start signal
reg             ctrl_vsync_run; // control signal for vsync
   counter
reg [8:0]       ctrl_vsync_cnt; // counter for vsync
reg             ctrl_hsync_run; // control signal for hsync
   counter
reg [8:0]       ctrl_hsync_cnt; // counter  for hsync
reg             ctrl_data_run;  // control signal for data
   processing
reg [31 : 0]  in_memory  [0 : sizeOfLengthReal/4]; // memory
                            // to store  32-bit data image
reg [7 : 0]   total_memory [0 : sizeOfLengthReal-1];    // memory
                        //to store  8-bit data image
// temporary memory to save image data : size will be WIDTH*
   HEIGHT*3
integer temp_BMP   [0 : WIDTH*HEIGHT*3 - 1];
integer org_R  [0 : WIDTH*HEIGHT - 1]; // temporary storage for R
```

```
    //component
integer org_G  [0 : WIDTH*HEIGHT - 1];  // temporary storage for
   G //component
integer org_B  [0 : WIDTH*HEIGHT - 1];  // temporary storage for
   B //component
// counting variables
integer i, j;
// temporary signals for calculation: details in the paper.
integer tempR0,tempR1,tempG0,tempG1,tempB0,tempB1; // temporary
   //variables in contrast and brightness operation

integer value,value1,value2,value4;// temporary variables in
   invert //and threshold operation
reg [ 9:0] row; // row index of the image
reg [10:0] col; // column index of the image
reg [18:0] data_count;// data counting for entire pixels of the
   //image
//-------------------------------------------------//
// -------- Reading data from input file ----------//
//-------------------------------------------------//
initial begin
    $readmemh(INFILE,total_memory,0,sizeOfLengthReal-1); // read
       file from INFILE
end
// use 3 intermediate signals RGB to save image data
always@(start) begin
    if(start == 1'b1) begin
        for(i=0; i<WIDTH*HEIGHT*3 ; i=i+1) begin
            temp_BMP[i] = total_memory[i+0][7:0];
        end

        for(i=0; i<HEIGHT; i=i+1) begin
            for(j=0; j<WIDTH; j=j+1) begin
                org_R[WIDTH*i+j] = temp_BMP[WIDTH*3*(HEIGHT-i-1)
                   +3*j+0]; // save Red component
                org_G[WIDTH*i+j] = temp_BMP[WIDTH*3*(HEIGHT-i-1)
                   +3*j+1];// save Green component
                org_B[WIDTH*i+j] = temp_BMP[WIDTH*3*(HEIGHT-i-1)
                   +3*j+2];// save Blue component
            end
        end
    end
end
//-------------------------------------------------//
// ---Begin to read image file once reset was high ---//
// ---by creating a starting pulse (start)------------//
//-------------------------------------------------//
always@(posedge HCLK, negedge HRESETn)
begin
    if(!HRESETn) begin
        start <= 0;
```

```
                HRESETn_d <= 0;
    end
    else begin
        HRESETn_d <= HRESETn;
                if(HRESETn == 1'b1 && HRESETn_d == 1'b0)

                        start <= 1'b1;
                else
                        start <= 1'b0;
    end
end

//-------------------------------------------------------------//
// Finite state machine for reading RGB888 data from memory and
   creating hsync and vsync pulses --//
//-------------------------------------------------------------//
always@(posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
        cstate <= ST_IDLE;
    end
    else begin
        cstate <= nstate; // update next state
    end
end
//-----------------------------------------//
//--------- State Transition --------------//
//-----------------------------------------//
// IDLE . VSYNC . HSYNC . DATA
always @(*) begin
        case(cstate)
                ST_IDLE: begin
                        if(start)
                                nstate = ST_VSYNC;
                        else
                                nstate = ST_IDLE;
                end
                ST_VSYNC: begin
                        if(ctrl_vsync_cnt == START_UP_DELAY)
                                nstate = ST_HSYNC;
                        else
                                nstate = ST_VSYNC;
                end
                ST_HSYNC: begin
                        if(ctrl_hsync_cnt == HSYNC_DELAY)
                                nstate = ST_DATA;
                        else
                                nstate = ST_HSYNC;
                end
                ST_DATA: begin
                        if(ctrl_done)
```

```
                                nstate = ST_IDLE;
                        else begin
                                if(col == WIDTH - 2)
                                        nstate = ST_HSYNC;
                                else
                                        nstate = ST_DATA;
                        end
                end
        endcase
end
// ------------------------------------ //
//  counting for time period of vsync, hsync, data processing  //
// ------------------------------------ //
always @(*) begin
        ctrl_vsync_run = 0;
        ctrl_hsync_run = 0;
        ctrl_data_run  = 0;
        case(cstate)
                ST_VSYNC:       begin ctrl_vsync_run = 1; end
                   // trigger counting for vsync
                ST_HSYNC:       begin ctrl_hsync_run = 1; end
                   // trigger counting for hsync
                ST_DATA:        begin ctrl_data_run  = 1; end
                   // trigger counting for data processing
        endcase
end
// counters for vsync, hsync
always@(posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
        ctrl_vsync_cnt <= 0;
                ctrl_hsync_cnt <= 0;
    end
    else begin
        if(ctrl_vsync_run)
                        ctrl_vsync_cnt <= ctrl_vsync_cnt + 1; //
                           counting for vsync
                else
                        ctrl_vsync_cnt <= 0;

        if(ctrl_hsync_run)
                        ctrl_hsync_cnt <= ctrl_hsync_cnt + 1;
                           // counting for hsync
                else
                        ctrl_hsync_cnt <= 0;
    end
end
// counting column and row index  for reading memory
always@(posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
```

```
        row <= 0;
                col <= 0;
    end
        else begin
                if(ctrl_data_run) begin
                        if(col == WIDTH - 2) begin
                                row <= row + 1;
                        end
                        if(col == WIDTH - 2)
                                col <= 0;
                        else
                                col <= col + 2; // reading 2
                                   pixels in parallel
                end
        end
end
//-------------------------------------------------//
//-----------------Data counting---------- ---------//
//-------------------------------------------------//
always@(posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
        data_count <= 0;
    end
    else begin
        if(ctrl_data_run)
                        data_count <= data_count + 1;
    end
end
assign VSYNC = ctrl_vsync_run;
assign ctrl_done = (data_count == 196607)? 1'b1: 1'b0; // done
   flag
//-------------------------------------------------//
//-------------  Image processing   ---------------//
//-------------------------------------------------//
always @(*) begin

        HSYNC   = 1'b0;
        DATA_R0 = 0;
        DATA_G0 = 0;
        DATA_B0 = 0;
        DATA_R1 = 0;
        DATA_G1 = 0;
        DATA_B1 = 0;
        if(ctrl_data_run) begin

                HSYNC   = 1'b1;
                `ifdef BRIGHTNESS_OPERATION
                /*************************************/

                /*              BRIGHTNESS ADDITION OPERATION */
```

```
            /*************************************/
            if(SIGN == 1) begin
            // R0
            tempR0 = org_R[WIDTH * row + col    ] + VALUE;
            if (tempR0 > 255)
                    DATA_R0 = 255;
            else
                    DATA_R0 = org_R[WIDTH * row + col    ] +
                       VALUE;
            // R1
            tempR1 = org_R[WIDTH * row + col+1    ] + VALUE;
            if (tempR1 > 255)
                    DATA_R1 = 255;
            else
                    DATA_R1 = org_R[WIDTH * row + col+1    ] +
                        VALUE;
            // G0
            tempG0 = org_G[WIDTH * row + col    ] + VALUE;
            if (tempG0 > 255)
                    DATA_G0 = 255;
            else
                    DATA_G0 = org_G[WIDTH * row + col    ] +
                       VALUE;
            tempG1 = org_G[WIDTH * row + col+1    ] + VALUE;
            if (tempG1 > 255)
                    DATA_G1 = 255;
            else
                    DATA_G1 = org_G[WIDTH * row + col+1    ] +
                        VALUE;
            // B
            tempB0 = org_B[WIDTH * row + col    ] + VALUE;
            if (tempB0 > 255)
                    DATA_B0 = 255;
            else
                    DATA_B0 = org_B[WIDTH * row + col    ] +
                       VALUE;
            tempB1 = org_B[WIDTH * row + col+1    ] + VALUE;
            if (tempB1 > 255)
                    DATA_B1 = 255;
            else
                    DATA_B1 = org_B[WIDTH * row + col+1    ] +
                        VALUE;
    end
    else begin
    /*************************************/
    /*      BRIGHTNESS SUBTRACTION OPERATION */
    /*************************************/
            // R0
            tempR0 = org_R[WIDTH * row + col    ] - VALUE;
            if (tempR0 < 0)
                    DATA_R0 = 0;
```

```
                else
                        DATA_R0 = org_R[WIDTH * row + col    ] -
                           VALUE;
                // R1
                tempR1 = org_R[WIDTH * row + col+1    ] - VALUE;
                if (tempR1 < 0)
                        DATA_R1 = 0;
                else
                        DATA_R1 = org_R[WIDTH * row + col+1    ] -
                            VALUE;
                // G0
                tempG0 = org_G[WIDTH * row + col    ] - VALUE;
                if (tempG0 < 0)
                        DATA_G0 = 0;
                else
                        DATA_G0 = org_G[WIDTH * row + col    ] -
                           VALUE;
                tempG1 = org_G[WIDTH * row + col+1    ] - VALUE;
                if (tempG1 < 0)
                        DATA_G1 = 0;
                else
                        DATA_G1 = org_G[WIDTH * row + col+1    ] -
                            VALUE;
                // B
                tempB0 = org_B[WIDTH * row + col    ] - VALUE;
                if (tempB0 < 0)
                        DATA_B0 = 0;
                else
                        DATA_B0 = org_B[WIDTH * row + col    ] -
                           VALUE;
                tempB1 = org_B[WIDTH * row + col+1    ] - VALUE;
                if (tempB1 < 0)
                        DATA_B1 = 0;
                else
                        DATA_B1 = org_B[WIDTH * row + col+1    ] -
                            VALUE;
        end
                `endif

                /*************************************/

                /*              INVERT_OPERATION
                             */
                /*************************************/
                `ifdef INVERT_OPERATION
                        value2 = (org_B[WIDTH * row + col  ] +
                           org_R[WIDTH * row + col  ] +org_G[
                           WIDTH * row + col  ])/3;
                        DATA_R0=255-value2;
                        DATA_G0=255-value2;
                        DATA_B0=255-value2;
```

```
                        value4 = (org_B[WIDTH * row + col+1  ] +
                           org_R[WIDTH * row + col+1  ] +org_G[
                           WIDTH * row + col+1  ])/3;
                        DATA_R1=255-value4;
                        DATA_G1=255-value4;
                        DATA_B1=255-value4;
                `endif
                /*************************************/

                /********THRESHOLD OPERATION  *********/
                /*************************************/
                 /*************************************/
  `ifdef THRESHOLD_OPERATION
  value = (org_R[WIDTH * row + col   ]+org_G[WIDTH * row + col
     ]+org_B[WIDTH * row + col   ])/3;
  if(value > THRESHOLD) begin
   DATA_R0=255;
   DATA_G0=255;
   DATA_B0=255;
  end
  else begin
   DATA_R0=0;
   DATA_G0=0;
   DATA_B0=0;
  end
  value1 = (org_R[WIDTH * row + col+1   ]+org_G[WIDTH * row + col
     +1   ]+org_B[WIDTH * row + col+1   ])/3;
  if(value1 > THRESHOLD) begin
   DATA_R1=255;
   DATA_G1=255;
   DATA_B1=255;
  end
  else begin
   DATA_R1=0;
   DATA_G1=0;
   DATA_B1=0;
  end
  `endif

        end
end

endmodule
```

The image processing operation is selected in the following "parameter.v" file. To change the processing operation, just switch the comment line.

```
/*********************************/
/************* Definition file **********/
/*************************************/

`define INPUTFILENAME            "gojo.mem" // Input file name
```

```
`define OUTPUTFILENAME            "output.bmp"            // Output
    file name

// Choose the operation of code by delete // in the beginning of
   the selected line

//`define BRIGHTNESS_OPERATION //uncomment this line for
   brightneess operation
//`define INVERT_OPERATION //uncomment this line for invertion
   operation
`define THRESHOLD_OPERATION
```

The "parameter.v" file is also to define paths and names of the input and output file. After processing the image, it is needed to write the processed data to an output image for verifications.

The following Verilog code is to write the processed image data to a bitmap image for verification:

```
/*****************************************************/
/*******     Module for writing .bmp image      ***/
/*****************************************************/
module image_write
#(parameter WIDTH       = 768,  // Image width
                        HEIGHT  = 512,  // Image height
                        INFILE  = "output.bmp", // Output image
                        BMP_HEADER_NUM = 54     // Header for bmp
                            image
)
(
        input HCLK,     // Clock
        input HRESETn,// Reset active low
        input hsync,// Hsync pulse

    input [7:0]  DATA_WRITE_R0,// Red 8-bit data (odd)
    input [7:0]  DATA_WRITE_G0, // Green 8-bit data (odd)
    input [7:0]  DATA_WRITE_B0, // Blue 8-bit data (odd)
    input [7:0]  DATA_WRITE_R1, // Red 8-bit data (even)
    input [7:0]  DATA_WRITE_G1,// Green 8-bit data (even)
    input [7:0]  DATA_WRITE_B1,// Blue 8-bit data (even)
        output  reg      Write_Done
);
integer BMP_header [0 : BMP_HEADER_NUM - 1];// BMP header
reg [7:0] out_BMP  [0 : WIDTH*HEIGHT*3 - 1];// Temporary memory
   for image
reg [18:0] data_count;  // Counting data
wire done;                                  // done flag
// counting variables
integer i;
integer k, l, m;
integer fd;
//------------------------------------------------------------//
```

```
//-------Header data for bmp image--------------------------//
//----------------------------------------------------------//
// Windows BMP files begin with a 54-byte header:
// Check the website to see the value of this header: http://www.
   fastgraph.com/help/bmp_header_format.html
initial begin
        BMP_header[ 0] = 66;BMP_header[28] =24;
        BMP_header[ 1] = 77;BMP_header[29] = 0;
        BMP_header[ 2] = 54;BMP_header[30] = 0;
        BMP_header[ 3] =  0;BMP_header[31] = 0;
        BMP_header[ 4] = 18;BMP_header[32] = 0;
        BMP_header[ 5] =  0;BMP_header[33] = 0;
        BMP_header[ 6] =  0;BMP_header[34] = 0;
        BMP_header[ 7] =  0;BMP_header[35] = 0;
        BMP_header[ 8] =  0;BMP_header[36] = 0;
        BMP_header[ 9] =  0;BMP_header[37] = 0;
        BMP_header[10] = 54;BMP_header[38] = 0;
        BMP_header[11] =  0;BMP_header[39] = 0;
        BMP_header[12] =  0;BMP_header[40] = 0;
        BMP_header[13] =  0;BMP_header[41] = 0;
        BMP_header[14] = 40;BMP_header[42] = 0;
        BMP_header[15] =  0;BMP_header[43] = 0;
        BMP_header[16] =  0;BMP_header[44] = 0;
        BMP_header[17] =  0;BMP_header[45] = 0;
        BMP_header[18] =  0;BMP_header[46] = 0;
        BMP_header[19] =  3;BMP_header[47] = 0;
        BMP_header[20] =  0;BMP_header[48] = 0;
        BMP_header[21] =  0;BMP_header[49] = 0;
        BMP_header[22] =  0;BMP_header[50] = 0;
        BMP_header[23] =  2;BMP_header[51] = 0;
        BMP_header[24] =  0;BMP_header[52] = 0;
        BMP_header[25] =  0;BMP_header[53] = 0;
        BMP_header[26] =  1;
        BMP_header[27] =  0;
end
// row and column counting for temporary memory of image
always@(posedge HCLK, negedge HRESETn) begin
    if(!HRESETn) begin
        l <= 0;
        m <= 0;
    end else begin
        if(hsync) begin
            if(m == WIDTH/2-1) begin
                m <= 0;
                l <= l + 1; // count to obtain row index of the
                   out_BMP temporary memory to save image data
            end else begin
                m <= m + 1; // count to obtain column index of
                   the out_BMP temporary memory to save image
                   data
            end
```

```
        end
    end
end
// Writing RGB888 even and odd data to the temp memory
always@(posedge HCLK, negedge HRESETn) begin
    if(!HRESETn) begin
        for(k=0;k<WIDTH*HEIGHT*3;k=k+1) begin
            out_BMP[k] <= 0;
        end
    end else begin
        if(hsync) begin
            out_BMP[WIDTH*3*(HEIGHT-1-1)+6*m+2] <= DATA_WRITE_R0;
            out_BMP[WIDTH*3*(HEIGHT-1-1)+6*m+1] <= DATA_WRITE_G0;
            out_BMP[WIDTH*3*(HEIGHT-1-1)+6*m  ] <= DATA_WRITE_B0;
            out_BMP[WIDTH*3*(HEIGHT-1-1)+6*m+5] <= DATA_WRITE_R1;
            out_BMP[WIDTH*3*(HEIGHT-1-1)+6*m+4] <= DATA_WRITE_G1;
            out_BMP[WIDTH*3*(HEIGHT-1-1)+6*m+3] <= DATA_WRITE_B1;
        end
    end
end
// data counting
always@(posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
        data_count <= 0;
    end
    else begin
        if(hsync)
                        data_count <= data_count + 1; // pixels
                           counting for create done flag
    end
end
assign done = (data_count == 196607)? 1'b1: 1'b0; // done flag
   once all pixels were processed
always@(posedge HCLK, negedge HRESETn)
begin
    if(~HRESETn) begin
        Write_Done <= 0;
    end
    else begin
                Write_Done <= done;
    end
end
//------------------------------------------------------------//
//--------------Write .bmp file          ----------------------//
//------------------------------------------------------------//
initial begin
    fd = $fopen(INFILE, "wb+");
end
always@(Write_Done) begin // once the processing was done, bmp
   image will be created
```

```
    if(Write_Done == 1'b1) begin
        for(i=0; i<BMP_HEADER_NUM; i=i+1) begin
            $fwrite(fd, "%c", BMP_header[i][7:0]); // write the
               header
        end

        for(i=0; i<WIDTH*HEIGHT*3; i=i+6) begin
                // write R0B0G0 and R1B1G1 (6 bytes) in a loop
            $fwrite(fd, "%c", out_BMP[i  ][7:0]);
            $fwrite(fd, "%c", out_BMP[i+1][7:0]);
            $fwrite(fd, "%c", out_BMP[i+2][7:0]);
            $fwrite(fd, "%c", out_BMP[i+3][7:0]);
            $fwrite(fd, "%c", out_BMP[i+4][7:0]);
            $fwrite(fd, "%c", out_BMP[i+5][7:0]);
        end
    end
end
endmodule
```

The header data for the bitmap image is very important and it is published here. If there is no header data, the written image could not be correctly displayed. In Verilog HDL, the `$fwrite` command is used to write data to a file.

Next, is a test bench Verilog code to verify the image processing operations.

```
`timescale 1ns/1ps
/*****************************************************/
/************** Testbench for simulation ***************/
/*****************************************************/

`include "parameter.v"  // include definition file

module tb_simulation;

//-----------------------------------------------
// Internal Signals
//-----------------------------------------------

reg HCLK, HRESETn;
wire          vsync;
wire          hsync;
wire [ 7 : 0] data_R0;
wire [ 7 : 0] data_G0;
wire [ 7 : 0] data_B0;
wire [ 7 : 0] data_R1;
wire [ 7 : 0] data_G1;
wire [ 7 : 0] data_B1;
wire enc_done;

//-----------------------------------------------
// Components
//-----------------------------------------------
```

```
image_read
#(.INFILE(`INPUTFILENAME))
        u_image_read
(
    .HCLK                       (HCLK    ),
    .HRESETn                  (HRESETn ),
    .VSYNC                      (vsync   ),
    .HSYNC                      (hsync   ),
    .DATA_R0                  (data_R0 ),
    .DATA_G0                  (data_G0 ),
    .DATA_B0                  (data_B0 ),
    .DATA_R1                  (data_R1 ),
    .DATA_G1                  (data_G1 ),
    .DATA_B1                  (data_B1 ),
        .ctrl_done                                  (enc_done)
);

image_write
#(.INFILE(`OUTPUTFILENAME))
        u_image_write
(
        .HCLK(HCLK),
        .HRESETn(HRESETn),
        .hsync(hsync),
   .DATA_WRITE_R0(data_R0),
   .DATA_WRITE_G0(data_G0),
   .DATA_WRITE_B0(data_B0),
   .DATA_WRITE_R1(data_R1),
   .DATA_WRITE_G1(data_G1),
   .DATA_WRITE_B1(data_B1),
        .Write_Done()
);

//------------------------------------------------
// Test Vectors
//------------------------------------------------
initial begin
    HCLK = 0;
    forever #10 HCLK = ~HCLK;
end

initial begin
    HRESETn     = 0;
    #25 HRESETn = 1;
end


endmodule
```

Figure 4.1: Input Original image.

Run the simulation for 6ms, close the simulation and open the output image for checking the result. Followings are the output images which are processed by the selected operations in parameter.v:

Figure 4.2: Output bitmap image after Brightness reduction.

Figure 4.3: Output bitmap image after Increasing Brightness .

Figure 4.4: Output bitmap image after Invertion Operation .

Figure 4.5: Output bitmap image after Thresholding Operation .

Chapter 5

Results and Conclusion

Present screenshots, waveforms, or VGA outputs. Discuss performance, verification, and accuracy.