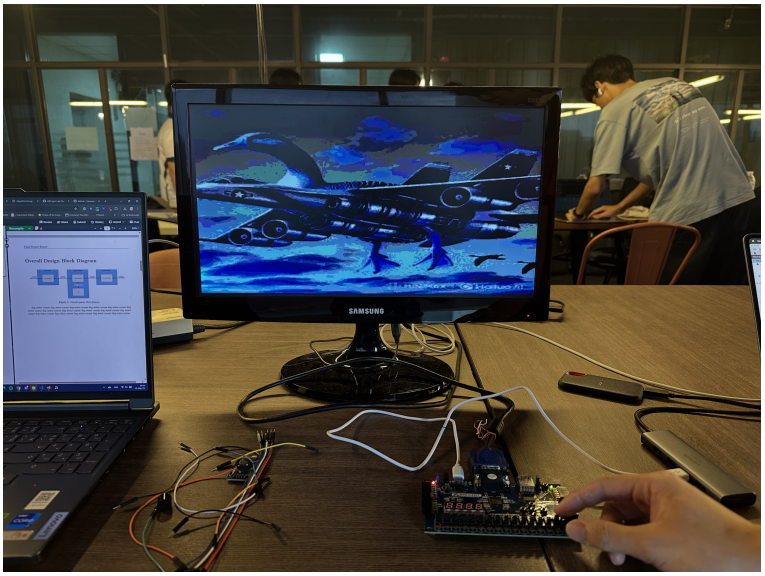
Hardware Synthesis Laboratory I

Final Project Report

FPGA Image Displayer via VGA
Using Verilog on Basys 3



Group 23:

Stewie Griffin

Pon Kittinaraporn	6630237121
Pakornkiat Opaprakasit	6630180021
Chayutphong Soisri	6630054621
Thitiwut Ariyaprayoon	6631326521

Department of Computer Engineering
Faculty of Engineering, Chulalongkorn University

Contents

Overall Design Block Diagram

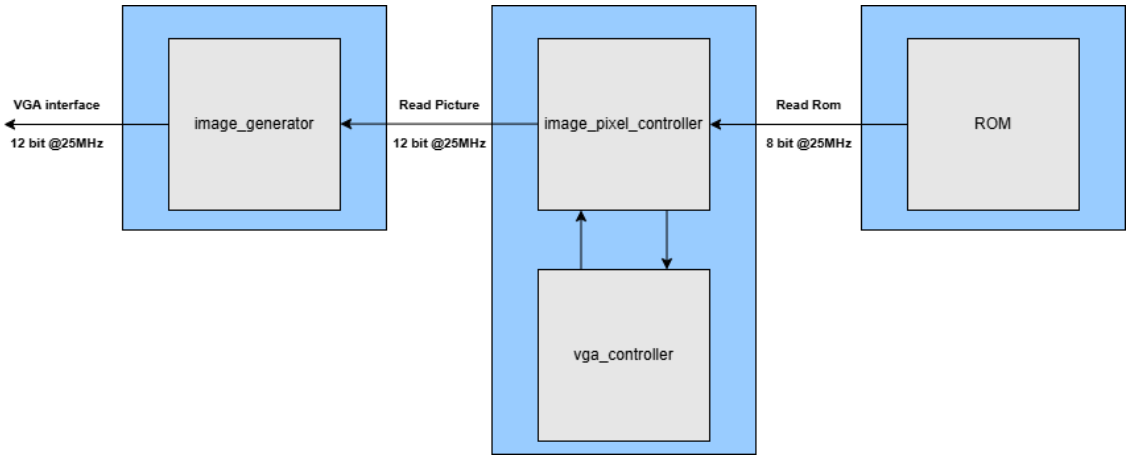


Figure 1: *Overall system block diagram*

Figure 1 shows the overall system block diagram.

Design Decision

2.1 Module Architecture

The system is divided into three modules: `image_generator`, `image_pixel_controller`, and `vga_controller`. This modular approach improves clarity, debugging, and scalability. Each module has a defined role—generation, coordination, and display—allowing isolated testing and easier future upgrades.

2.2 Communication Protocols

We used native signal interfacing instead of protocols like AXI, as all modules share the same clock and communicate directly. This reduces latency and resource overhead, which suits our tightly-coupled design.

2.3 Clock and Data Rate

A shared clock domain was chosen to avoid synchronization issues. The VGA controller operates at 25 MHz to match 640×480 @60Hz standards. Other modules operate within this timing without critical delays.

2.4 Resource Utilization

Using Single Port ROM for storing image data, but logic is efficiently distributed. The current design minimizes LUT and FF usage, focusing on simplicity.

Implementation Detail

3.1 VGA Controller

```
module vga_controller(
    input wire clk,
    input wire reset,
    output wire hsync,
    output wire vsync,
    output wire video_on,
    output wire [9:0] curr_x,
    output wire [9:0] curr_y
);

    parameter HD = 640;
    parameter HF = 16;
    parameter HS = 96;
    parameter HB = 48;
    parameter HMAX = HD + HF + HS + HB - 1;
    parameter VD = 480;
    parameter VF = 10;
    parameter VS = 2;
    parameter VB = 33;
    parameter VMAX = VD + VF + VS + VB -1;

    reg [9:0] h_count = 0;
    reg [9:0] v_count = 0;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            h_count <= 0;
            v_count <= 0;
        end else begin
            if (h_count == HMAX) begin
                h_count <= 0;
                if (v_count == VMAX) v_count <= 0;
                else v_count <= v_count + 1;
            end
            else h_count <= h_count + 1;
        end
    end

    assign hsync = (h_count >= (HD + HF) &&
                    h_count <  (HD + HF + HS)) ? 0 : 1;
    assign vsync = (v_count >= (VD + VF) &&
                    v_count <  (VD + VF + VS)) ? 0 : 1;
    assign video_on = (h_count < HD && v_count < VD);
    assign curr_x = h_count;
    assign curr_y = v_count;

endmodule
```

Figure 1: *vga_controller.v*

Module: vga_controller

Description: Generates timing signals for VGA display at 60 Hz using a 25 MHz clock.

3.2 Image Pixel Controller

```
module image_pixel_controller (
    input sw,
    input wire clk,
    input wire reset,
    input wire [9:0] curr_x,
    input wire [9:0] curr_y,
    input wire video_on,
    output wire [7:0] pixel_rgb
);

    wire [7:0] rom_data;
    reg [17:0] rom_addr;

    blk_mem_gen_0 (
        .addra(rom_addr),
        .clka(clk),
        .douta(rom_data),
        .rsta(reset)
    );

    always @(*) begin
        if (video_on && curr_x < 640 && curr_y < 480) begin
            // Scale 640x480 down to 320x240
            rom_addr = (curr_y >> 1) * 320 + (curr_x >> 1);
        end else begin
            rom_addr = 0;
        end
        rom_addr = (sw) ? rom_addr + 76800: rom_addr;
    end

    assign pixel_rgb = rom_data;

endmodule
```

Figure 2: *image_pixel_controller.v*

Module: image_pixel_controller

Description: Scale image to 320 x 420 and switch image by jumping to another address

3.3 Image Generator

```
module image_generator (
    input [1:0] sw,
    input clk,
    input reset,
    output wire [11:0] pixel_rgb,
    output wire hsync,
    output wire vsync
);

    wire clk_25MHz;

    clk_25MHz (
        .clk_in1(clk),
        .reset(reset),
        .clk_out1(clk_25MHz)
    );

    wire video_on;
    wire [9:0] curr_x;
    wire [9:0] curr_y;

    vga_controller (
        .clk(clk_25MHz),
        .reset(reset),
        .hsync(hsync),
        .vsync(vsync),
        .video_on(video_on),
        .curr_x(curr_x),
        .curr_y(curr_y)
    );

    wire [11:0] w_pixel_rgb;

    image_pixel_controller (
        .sw(sw[1]),
        .clk(clk_25MHz),
        .reset(reset),
        .curr_x(curr_x),
        .curr_y(curr_y),
        .video_on(video_on),
        .pixel_rgb(w_pixel_rgb)
    );

    assign pixel_rgb[11:9] = (sw[0]) ? 4097 - w_pixel_rgb[7:5] : w_pixel_rgb[7:5];
    assign pixel_rgb[7:5] = (sw[0]) ? 4097 - w_pixel_rgb[4:2] : w_pixel_rgb[4:2];
    assign pixel_rgb[3:2] = (sw[0]) ? 4097 - w_pixel_rgb[1:0] : w_pixel_rgb[1:0];

endmodule
```

Figure 3: *image_generator.v*

Module: image_generator

Description: This is the top module use to connect the wires and all the modules together. It contains Clock Wizrad module, vga_controller module and image_pixel_controller module.

3.4 Constraints

```
set_property PACKAGE_PIN W5      [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
    create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
        clk]

set_property PACKAGE_PIN V17     [get_ports {sw[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
set_property PACKAGE_PIN V16     [get_ports {sw[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]

set_property PACKAGE_PIN G19     [get_ports {pixel_rgb[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[0]}]
set_property PACKAGE_PIN H19     [get_ports {pixel_rgb[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[1]}]
set_property PACKAGE_PIN J19     [get_ports {pixel_rgb[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[2]}]
set_property PACKAGE_PIN N19     [get_ports {pixel_rgb[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[3]}]
set_property PACKAGE_PIN J17     [get_ports {pixel_rgb[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[4]}]
set_property PACKAGE_PIN H17     [get_ports {pixel_rgb[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[5]}]
set_property PACKAGE_PIN G17     [get_ports {pixel_rgb[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[6]}]
set_property PACKAGE_PIN D17     [get_ports {pixel_rgb[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[7]}]
set_property PACKAGE_PIN N18     [get_ports {pixel_rgb[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[8]}]
set_property PACKAGE_PIN L18     [get_ports {pixel_rgb[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[9]}]
set_property PACKAGE_PIN K18     [get_ports {pixel_rgb[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[10]}]
set_property PACKAGE_PIN J18     [get_ports {pixel_rgb[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pixel_rgb[11]}]
set_property PACKAGE_PIN P19     [get_ports hsync]
set_property IOSTANDARD LVCMOS33 [get_ports hsync]
set_property PACKAGE_PIN R19     [get_ports vsync]
set_property IOSTANDARD LVCMOS33 [get_ports vsync]

set_property PACKAGE_PIN U18     [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports reset]
```

Figure 4: *image_constr.xdc*

Module: image_constr.xdc (not a module)

Description: Mapping the logical design (Verilog Code) to Physical Hardware Pins on the Basys 3 Board

Challenges Faced

- Reading from and writing data to the SD card
- Limited learning resources
- Displaying images on the screen from the SD card
- Understanding communication protocols (e.g., SPI)
- Debugging with macOS machines
- Lack of proper documentation or example code
- Time-consuming trial and error process
- Lengthy synthesis and implementation process in Vivado

The main challenges of this project stemmed from our lack of experience, documentation, and code examples for working with VGA and SD card interfaces. Since we had no prior experience with reading from or writing to an SD card, working with it proved to be extremely challenging for us—as it likely is for many others as well.

Additionally, a significant portion of the project time was spent waiting for Vivado to complete tasks such as compilation, synthesis, or bitstream generation, all of which heavily depend on the performance of the machine used. Apart from software debugging, hardware debugging on macOS was also difficult. Since we used Docker to run Vivado on macOS, the built-in probe tool could not directly connect to the FPGA board via USB, adding another layer of complexity to the process.