CS 223 Digital Design

Project Report

14.12.2024

Alp Eren Köken
22202876
Section:2

1) General Information About Project:

In this project we aim to design and implement a Video Graphics Array (VGA) module on a Basys 3 FPGA board. To achieve that we have to implement VGA controller that manages timing, synchronization, and pixel output. In this project we firstly develop and display static patterns like checkboard, then we continue with more sophisticated dynamic interactive drawing canvas project. In this road we both use basys3 buttons and PS/2 mouse interface to interact with displayed objects.

The project is consist of three main stages:

1. **VGA Controller:** Build a working VGA controller that can scroll using directional buttons and render a test pattern.

2. **Drawing Canvas Application:** Build a completely interactive drawing canvas that allows users to assign colors and draw using the cursor.

3. **PS/2 Mouse Integration:** For simpler and more effective usage of operation, change the button-based cursor control to a PS/2 mouse interface.

2) Design and Implementation:

2.1) VGA Specification:

Here is the summary of VGA Specifications provided in project document:

Horizontal Timing:

Scanline Part	Pixels	Time (µs)
Visible Area	640	25.42
Front Porch	16	0.64
Sync Pulse	96	3.81
Back Porch	48	1.91
Whole Line	800	31.78

Vertical Timing:

Frame Part	Lines	Time (µs)
Visible Area	480	15,253.23
Front Porch	10	317.78
Sync Pulse	2	63.56
Back Porch	33	1,048.66
Whole Frame	525	16,683.22

The pixel clock runs at 25 MHz, synchronizing the transmission of RGB signals and HSYNC/VSYNC signals.

2.2) VGA Controller (Part 1):

In this part we focus on to display a checkboard pattern and scroll it by basys3 FPGA directional buttons. Our vga controller has a resolution standart 640x480 with a 60hz refresh rate. This module deals with issues like timing signal generation, dividing pixel clock and outputting RGB colour. This module uses Vga controller module in it.

Design:

VGA Controller Module:

Divides the 100 MHz frequency to create a 25 MHz pixel clock.keeps track of both vertical and horizontal counts for pixel locations.detects if the current pixel is inside the display area and outputs the synchronization signals (Hs and Vs).

Dividing Pixel Clock:

Normally basys3 works with 100MHz. However, we need 25MHz to be suitable for VGA requirements. We use 2 bit clock divider. In each rising clock edge we increase count and reset a fourth edge to get 100MHz/4=25MHz.

Timing Signal Generation:

We have used horizantal and vertical counters to scan all lines. We get HYSNC and VSYNC from this counters. At each clock tick horizontal counter increments with 1 and if pixel comes at thee end of the horizontal line we increment vertical counter with 1.

RGB Output (CheckBoard Pattern):

The RGB output produces a checkerboard appearance by cycling between black and white depending on the sum of the horizontal and vertical pixel indices.

2.3) Drawing Canvas (Part 2):

In this module we first display a black + shaped cursor on the white background which can be movable with directional buttons of basys3. Furthermore, we can paint the cursor’s point by center button of basys3 and select the colors via switches of basys3. Since our RAM is limited and gives RAM limit error, I have assumed one pixel as a real 32x32 pixel. By this change i have preserved the functionality of the code while increasing pixel size and decrase RAM usage. This module consists of these 3 parts: Color Selection, Drawing Logic, Simulation Waveforms. This module uses Vga controller module in it.

Design:

VGA Controller Module:

Divides the 100 MHz frequency to create a 25 MHz pixel clock.keeps track of both vertical and horizontal counts for pixel locations.detects if the current pixel is inside the display area and outputs the synchronization signals (Hs and Vs).

1. **Color Selection:**

   I have used 8 switches to select between 8 colors.On the other hand, i have stored colors as 3 bits for less RAM usage.

2. **Drawing Logic:**

   Initialization of the Canvas:

   Each element of the canvas, which represents a 20x15 grid, stores a 3-bit color value in a 1D array (memory). At first, every pixel is white (3'b110).

   Movement of the Cursor:

   Within the constraints of the grid, the basys3 directional buttons are used to update the cursor's location (cursor_x, cursor_y).

   Logic Drawing:

The current pixel and its surrounds are updated with the chosen color when the left center button of basys3 is pressed.

Drawing many neighboring pixels using the brush tool (isBrush) creates the illusion of a thicker brush.

3. **Simulation Waveforms:**

The HSYNC, VSYNC, and isVideo signals were analyzed in simulation to confirm correct timing and synchronization for the VGA output.

2.4) PS/2 Mouse Integration (Part 4):

In this module we just use a mouse to control our cursor and to paint background as a difference from part 2. This module uses Vga controller and ps2_mouse module in it. Furthermore ps2_mouse module has a submodule in it named ps2_validator.

Design:

The project integrates:

1. **VGA Controller (VGA_c):** Generates a 640x480 video signal and manages horizontal and vertical synchronization.

2. **PS/2 Mouse Controller (ps2_mouse, ps2_validator):** Decodes mouse inputs to retrieve position deltas and button states.

3. **Drawing Logic**: Updates a memory buffer to store pixel data, handles cursor movement, and draws on the canvas based on user input.

VGA Controller:

Divides the 100 MHz frequency to create a 25 MHz pixel clock.keeps track of both vertical and horizontal counts for pixel locations.detects if the current pixel is inside the display area and outputs the synchronization signals (Hs and Vs).

PS/2 Mouse Controller (ps2_mouse):

Assembles 11-bit words by capturing raw PS/2 data bits on clock edges.

Process goes with a submodule.

ps2_validator: Verifies the PS/2 packets' parity and start/stop bits. Validated packets are mapped to useable mouse signals by ps2_mouse_map.

Drawing Logic:

Initialization of the Canvas:

Each element of the canvas, which represents a 20x15 grid, stores a 3-bit color value in a 1D array (memory). At first, every pixel is white (3'b110).

Movement of the Cursor:

Within the constraints of the grid, the mouse's delta values are used to update the cursor's location (cursor_x, cursor_y).

Logic Drawing:

The current pixel and its surrounds are updated with the chosen color when the left mouse button is pressed.

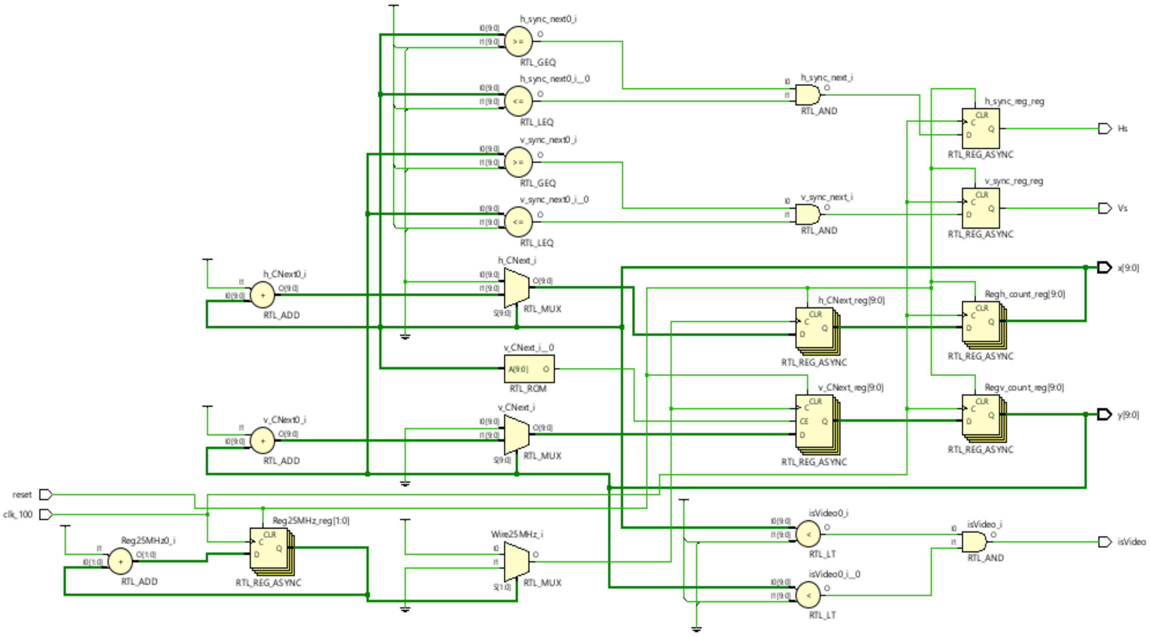
Drawing many neighboring pixels using the brush tool (isBrush) creates the illusion of a thicker brush.

3) RTL Schematics and State Diagram

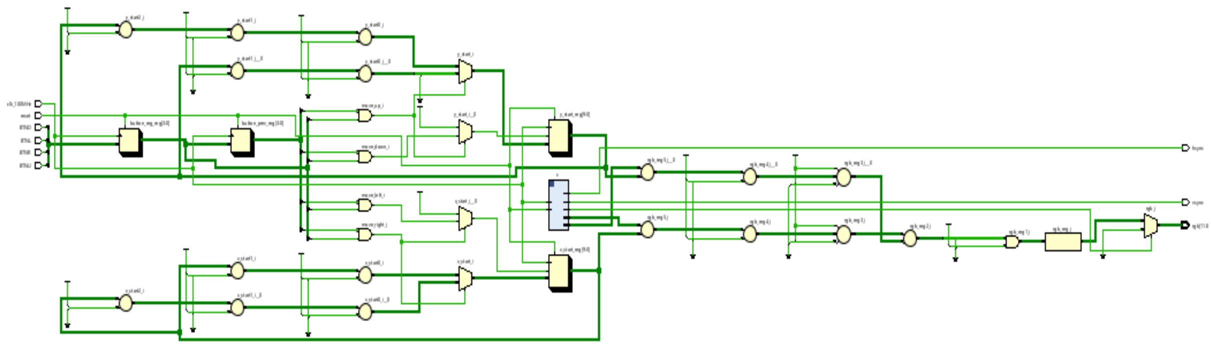
(VGA controller, drawing logic, cursor control, PS/2 mouse control):

RTL Schematics:

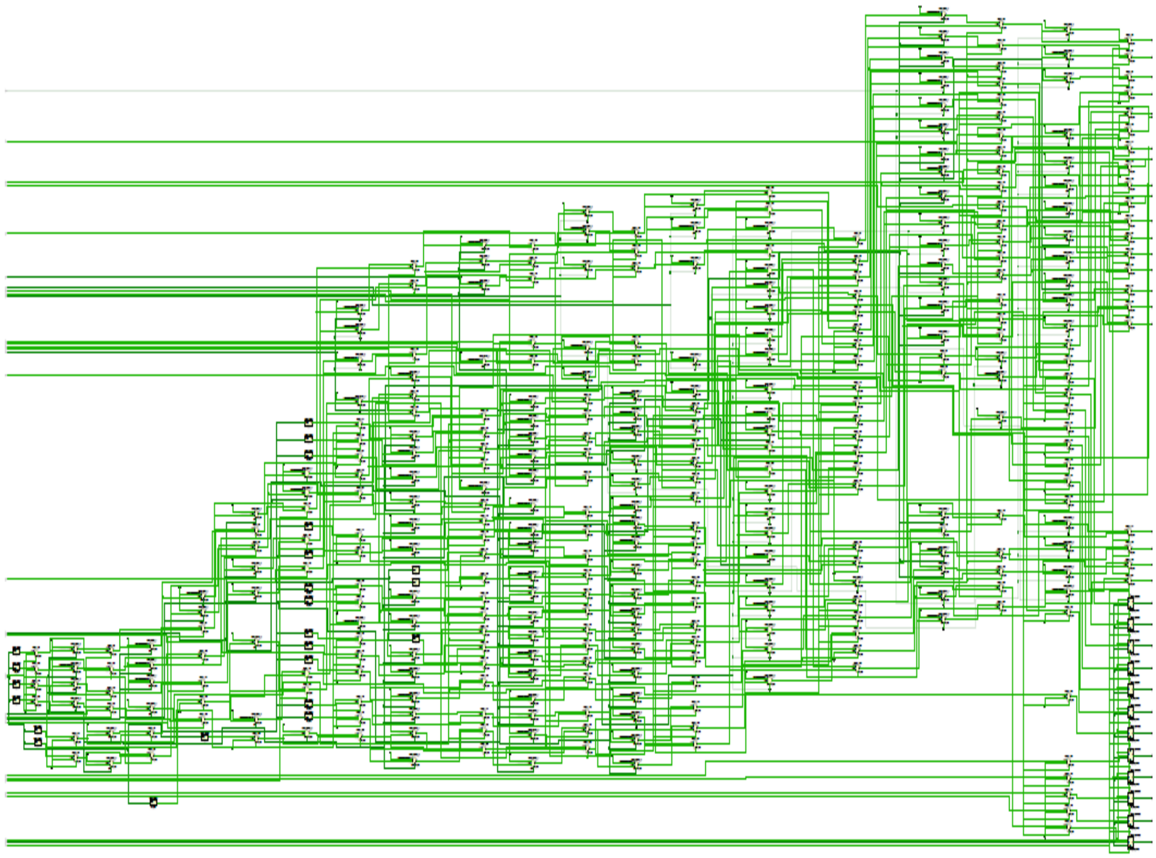
VGA_c:


h_sync_next0_i
RTL_GEQ
h_sync_next0_i__0
RTL_LEQ
v_sync_next0_i
RTL_GEQ
v_sync_next0_i__0
RTL_LEQ
h_sync_next_i
RTL_AND
v_sync_next_i
RTL_AND
h_sync_reg_reg
RTL_REG_ASYNC
v_sync_reg_reg
RTL_REG_ASYNC
Hs
Vs
x[9:0]
y[9:0]
h_CNext0_i
RTL_ADD
h_CNext_i
RTL_MUX
h_CNext_reg[9:0]
RTL_REG_ASYNC
Regh_count_reg[9:0]
RTL_REG_ASYNC
v_CNext_i__0
RTL_ROM
v_CNext0_i
RTL_ADD
v_CNext_i
RTL_MUX
v_CNext_reg[9:0]
RTL_REG_ASYNC
Regv_count_reg[9:0]
RTL_REG_ASYNC
reset
clk_100
Reg25MHz0_i
RTL_ADD
Reg25MHz_reg[1:0]
RTL_REG_ASYNC
Wire25MHz_i
RTL_MUX
isVideo0_i
RTL_LT
isVideo0_i__0
RTL_LT
isVideo_i
RTL_AND
isVideo

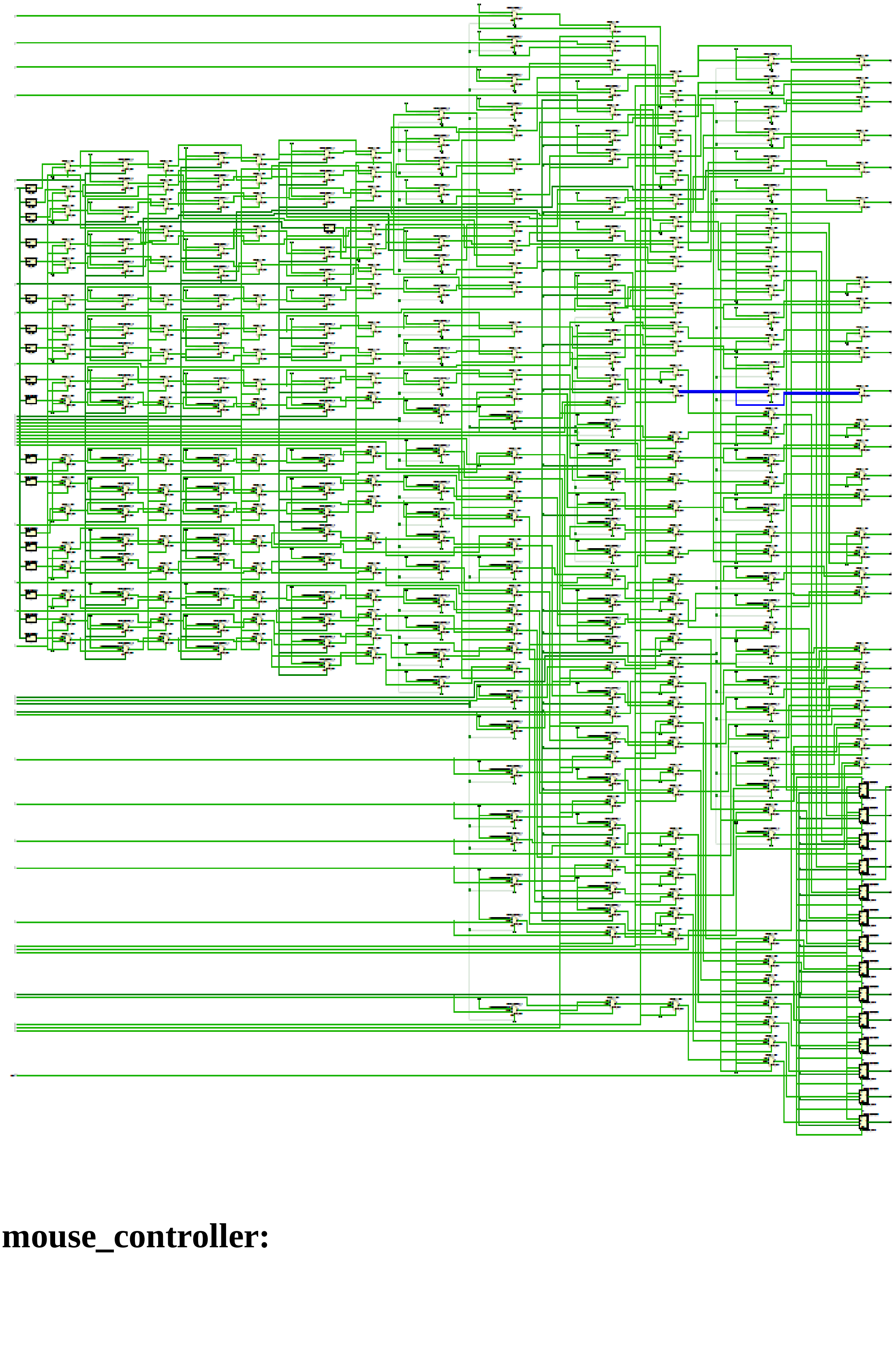
part1:



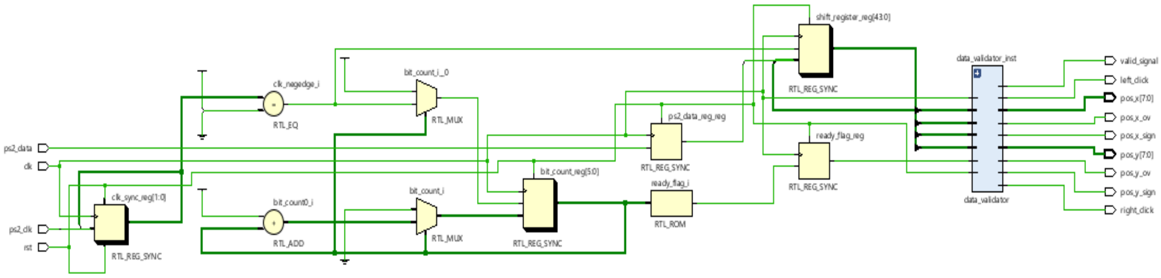
part2:



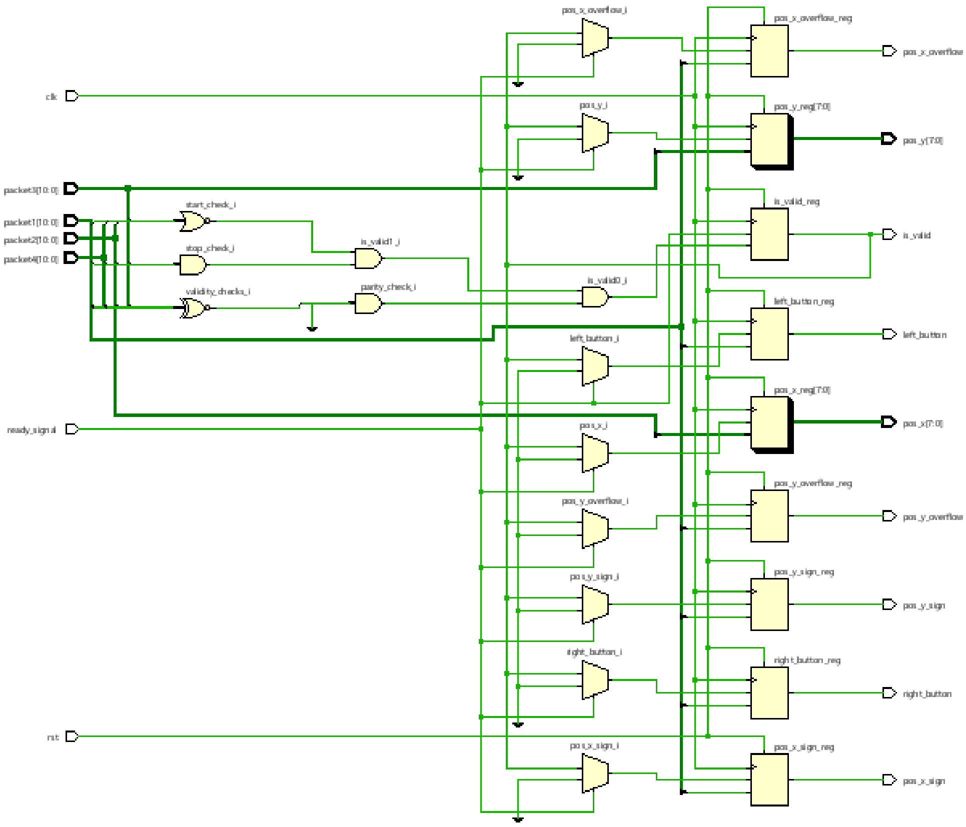
part3:



mouse_controller:

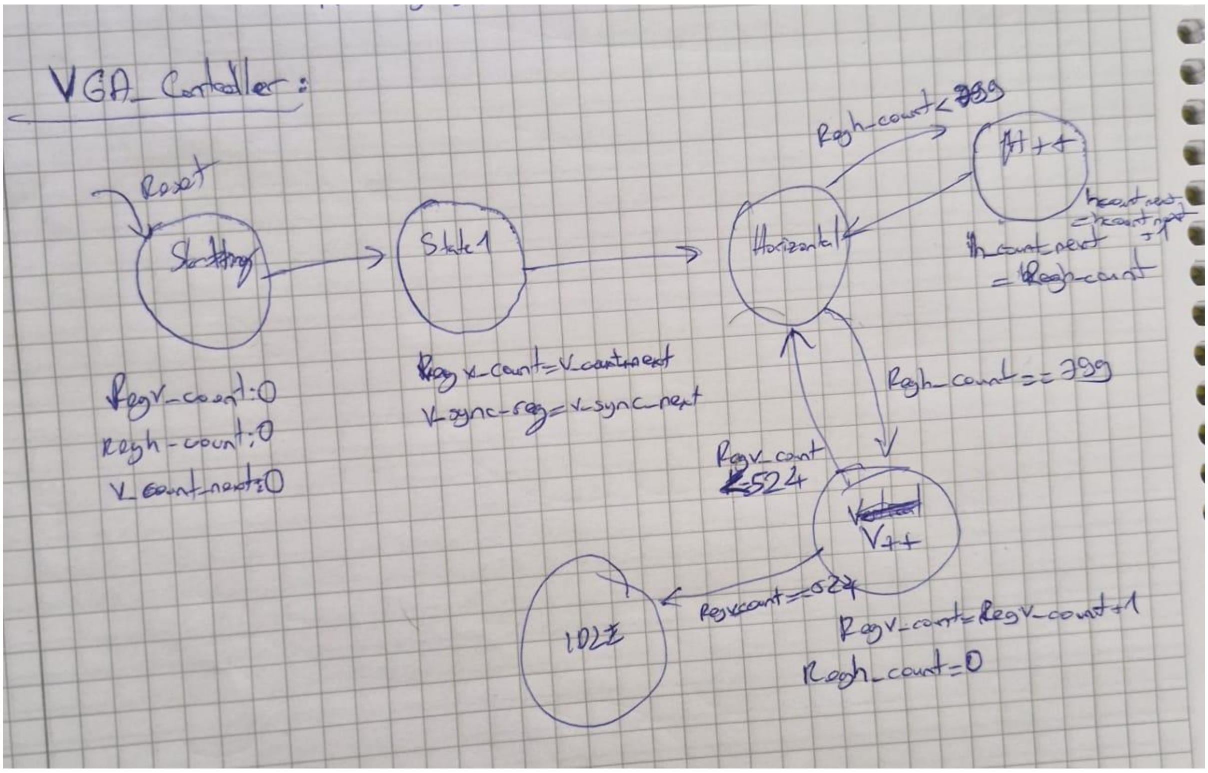


data_validator:

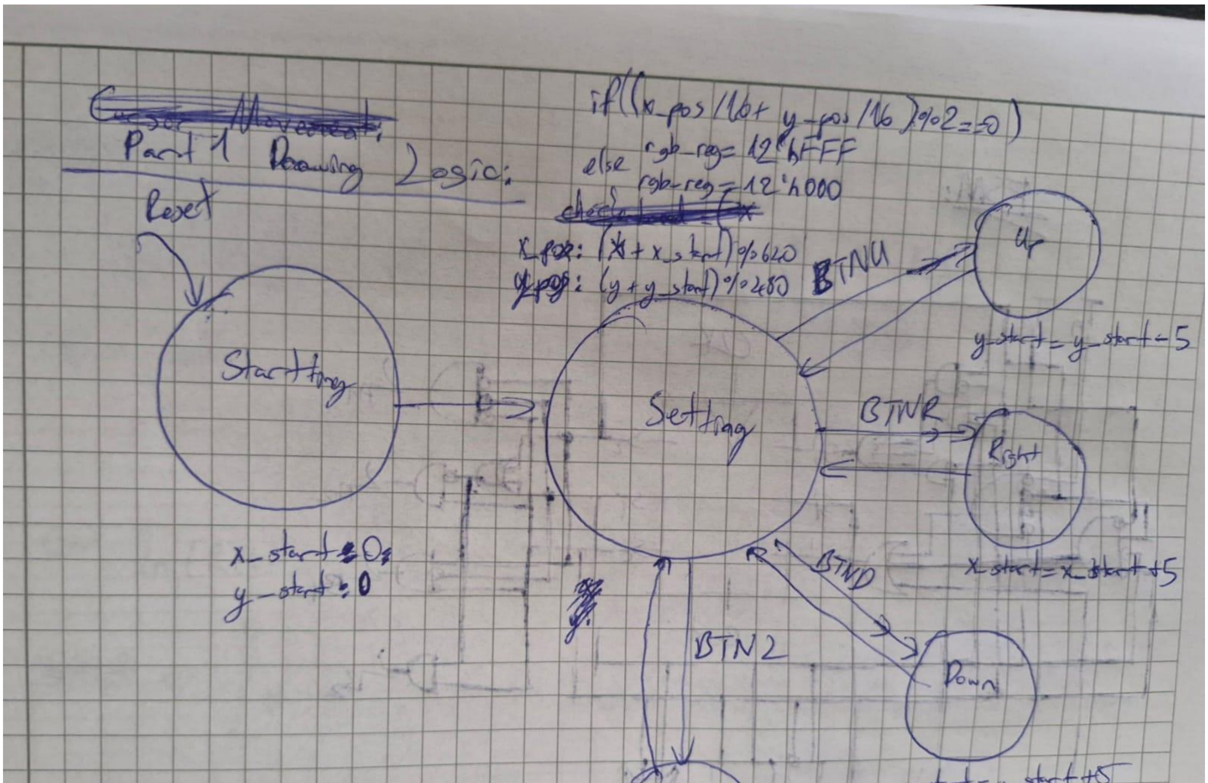


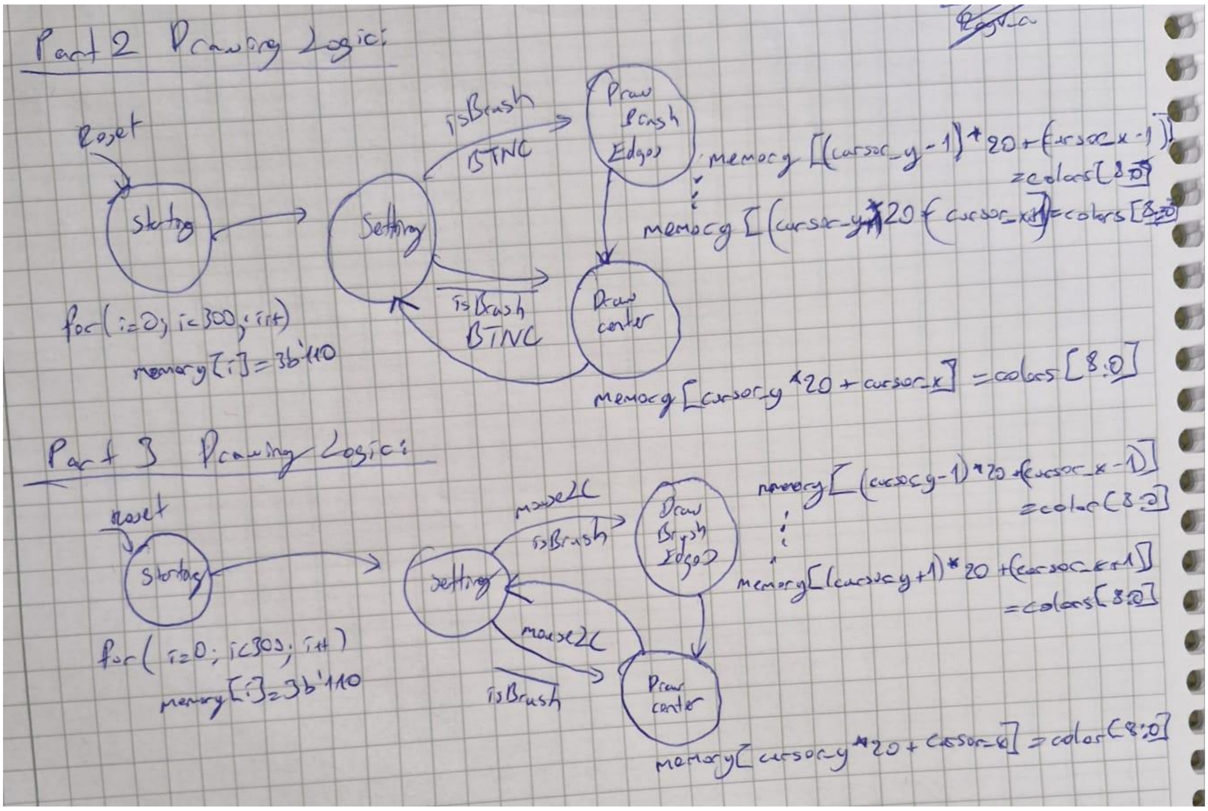
State Diagrams:

VGA Controller:

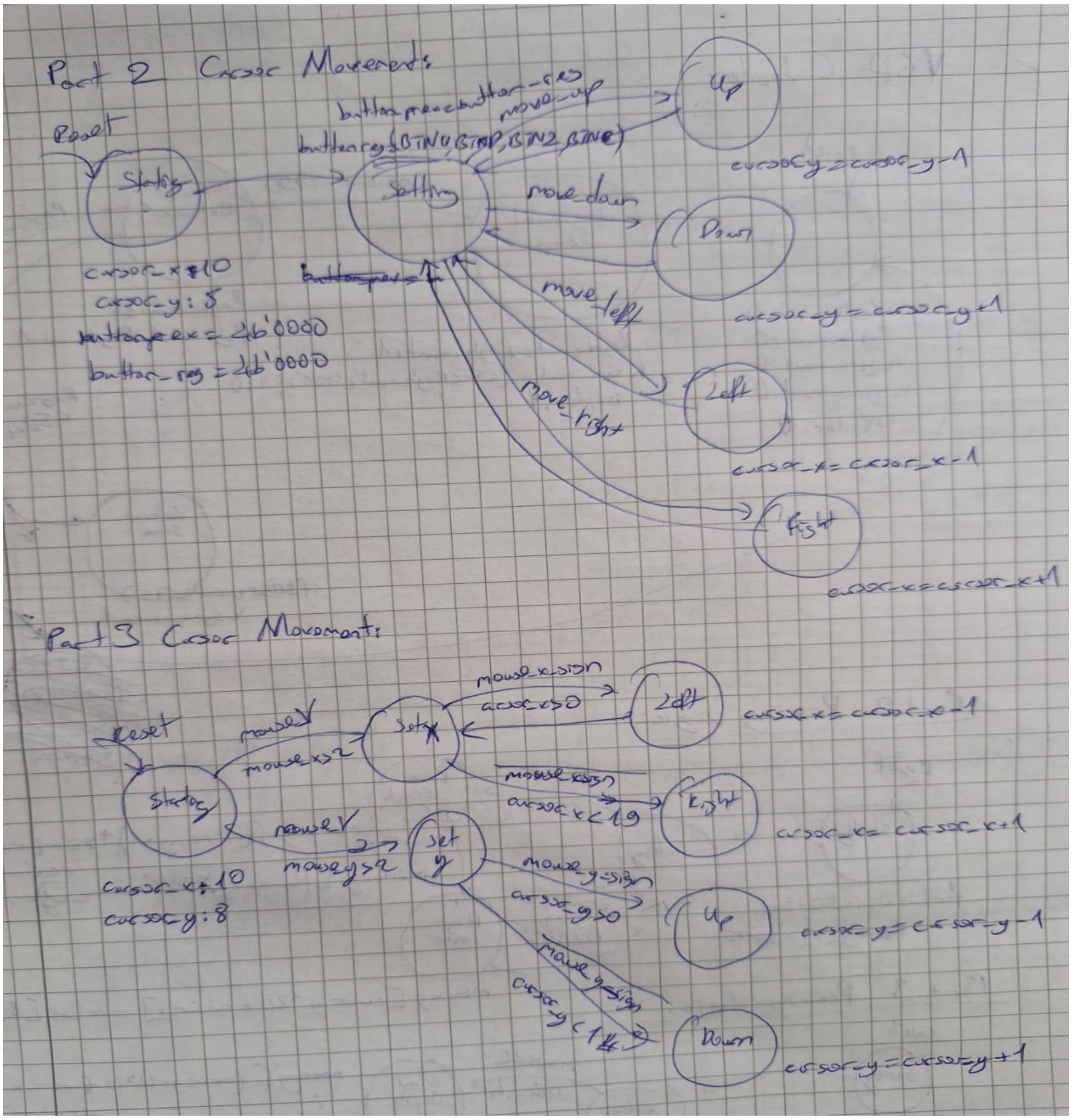


Drawing Logic:

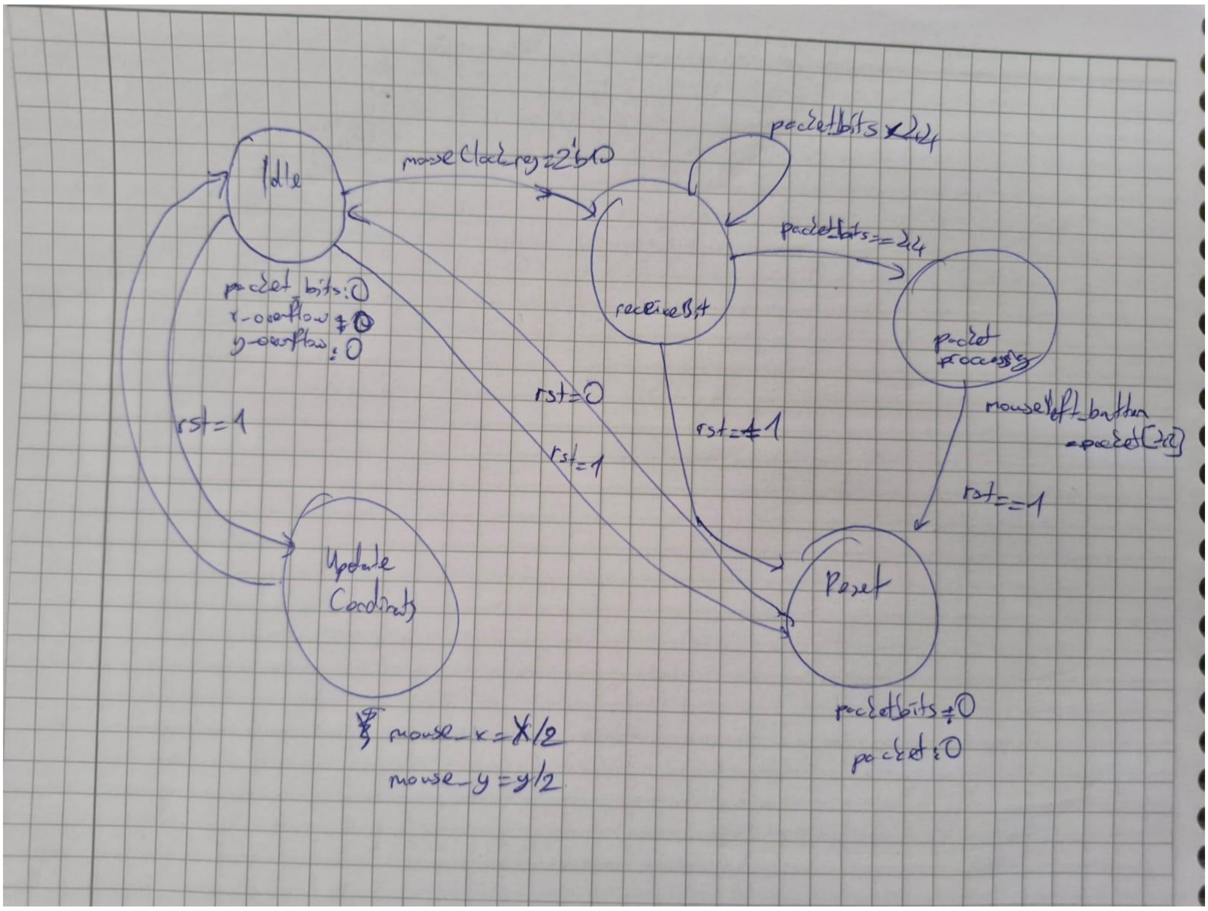




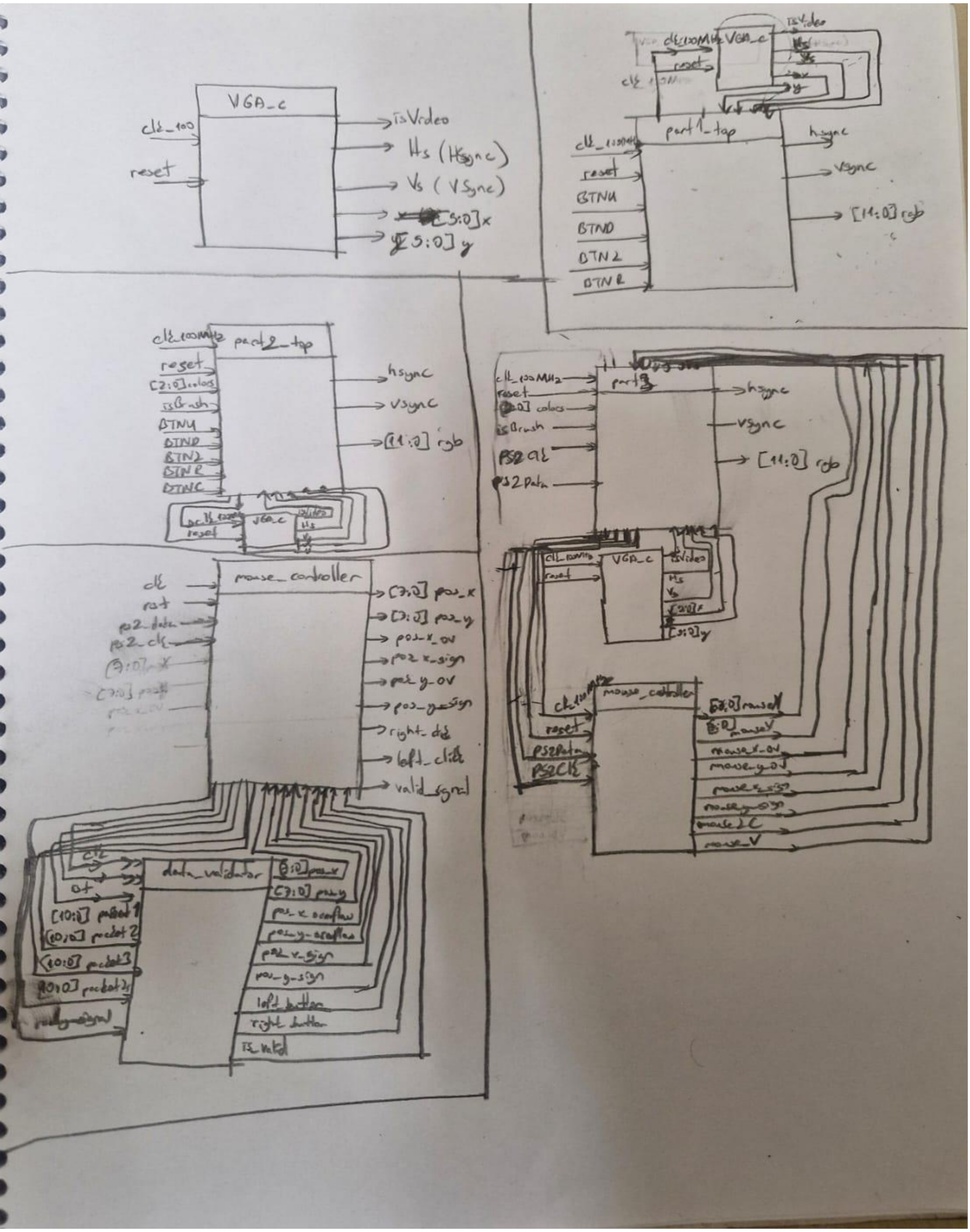
Cursor Movement:



PS/2 Mouse Control:



4) Block diagram of each module implemented:



VGA_c
clk_100
reset
isVideo
Hs (Hsync)
Vs (VSync)
[9:0] x
[9:0] y
part1_top
clk_100MHz
reset
BTNU
BTND
BTNL
BTNR
hsync
Vsync
[11:0] rgb
part2_top
clk_100MHz
reset
[7:0] colors
isBrush
BTNU
BTND
BTNL
BTNR
BTNC
hsync
Vsync
[11:0] rgb
part3
clk_100MHz
reset
colors
isBrush
PS2 clk
PS2 Data
hsync
Vsync
[11:0] rgb
mouse_controller
clk
rst
ps2_data
ps2_clk
[7:0] pos_x
[7:0] pos_y
pos_x_ov
pos_x_sign
pos_y_ov
pos_y_sign
right_click
left_click
valid_signal
data_validator
clk
rst
[10:0] packet 1
[10:0] packet 2
[10:0] packet 3
[10:0] packet 4
ready_signal
[7:0] pos_x
[7:0] pos_y
pos_x overflow
pos_y_overflow
pos_x_sign
pos_y_sign
left_button
right_button
is_valid
mouse_controller
clk_100MHz
reset
PS2Data
PS2Clk
mouseX
mouseY
mouse_x_ov
mouse_y_ov
mouse_x_sign
mouse_y_sign
mouseLC
mouse_V

5)References:

Vga signal 640 x 480 @ 60 hz industry standard timing — 60hz. http://www.tinyvga.com/vga-timing/ 640x480@60Hz. [Accessed 15-11-2024].

Javier Valcarce. Vga signal format timing and specifications. http://javiervalcarce.eu/html/vga-signal-format-timming-specs-en.html. [Accessed 15-11-2024]

CODES:

```
module VGA_c(

input clk_100,reset,
output isVideo,Hs,Vs,
output [9:0] x,y

);

reg  [1:0] Reg25MHz;       //25MHz from 100MHz
wire Wire25MHz;

always @(posedge clk_100 or posedge reset)
    if(reset)
      Reg25MHz<=0;
    else begin
      Reg25MHz <= Reg25MHz + 1;


     end


assign Wire25MHz = (Reg25MHz == 0)?1:0;

// divding clock, 1/4 of time

// Horizontal and Vertical registers for counting recording
reg [9:0] Regh_count, h_CNext;
reg [9:0] Regv_count, v_CNext;

// Outputs
```

```
reg v_sync_reg, h_sync_reg;
wire v_sync_next, h_sync_next;

always @(posedge clk_100 or posedge reset)
    if(reset) begin
        h_sync_reg<=0;
         v_sync_reg<=0;
        Regh_count<=0;
         Regv_count<=0;
    end


    else begin


        h_sync_reg<=h_sync_next;
         v_sync_reg<=v_sync_next;
        Regh_count<=h_CNext;
         Regv_count<=v_CNext;

    end

//------------------------------------------------------------


always @(posedge Wire25MHz or posedge reset)


    if(reset)
        h_CNext = 0;


    else


        if(Regh_count == 799)  begin


            h_CNext = 0;


        end else
```

```
            h_CNext = Regh_count + 1;

//-----------------------------------------------------------


always @(posedge Wire25MHz or posedge reset)


    if(reset)
        v_CNext = 0;
    else if((Regh_count == 799)&& (Regv_count == 524))
                v_CNext = 0;


            else if (((Regv_count<524) ||
(Regv_count>524))&&(Regh_count == 799)) begin
                v_CNext = Regv_count + 1;
                end

     assign h_sync_next=(Regh_count>=(656) && Regh_count<=(751));
     assign v_sync_next=(Regv_count>=(513) && Regv_count<=(515));
     assign isVideo=(Regh_count<640) && (Regv_count<480);

     assign Hs=h_sync_reg;
     assign Vs=v_sync_reg;


     assign x=Regh_count;
     assign y=Regv_count;


     assign pixelClk=Wire25MHz;
endmodule
module part1_top(
input clk_100MHz,reset,
input BTNU,BTND,BTNL,BTNR,
output hsync,vsync,
output [11:0] rgb


);
```

```
wire isVideo;

wire [9:0] x, y;
wire p_clk;

VGA_c c(clk_100MHz,reset,isVideo,hsync,vsync,x,y);

reg [9:0] x_start = 0;
reg [9:0] y_start = 0;
reg [9:0] x_pos, y_pos;

reg [3:0] button_reg, button_prev;
wire move_up, move_down, move_left, move_right;

reg [11:0] rgb_reg;

assign move_up    = ~button_prev[3] & button_reg[3];
assign move_down  = ~button_prev[2] & button_reg[2];
assign move_left  = ~button_prev[1] & button_reg[1];
assign move_right = ~button_prev[0] & button_reg[0];

always @(*) begin

    x_pos = (x + x_start) % 640;
    y_pos = (y + y_start) % 480;

    if ((x_pos/16 + y_pos/16)%2<0) || (x_pos/16 + y_pos/16)%2>0))
        rgb_reg = 12'h000;
    else
        rgb_reg = 12'hFFF;
end

//***************************************************************
always @(posedge clk_100MHz or posedge reset) begin
    if (reset) begin
        button_prev <= 4'b0000;
```

```
        button_reg <= 4'b0000;

         x_start <= 0;
        y_start <= 0;

    end else begin

        button_prev <= button_reg;
        button_reg <= {BTNU, BTND, BTNL, BTNR};
//Changes according to directions

        if (move_down)begin
            y_start <= (y_start+5) % 480;
     end

         if(move_left)begin
            x_start <= (x_start+635) % 640;
     end

        if (move_right)
            x_start <= (x_start + 5) % 640;

        if (move_up)
            y_start <= (y_start+475) % 480;

    end
end

assign rgb =(isVideo)?rgb_reg:12'h000;
```

endmodule module part2_top(

```
input clk_100MHz,reset,
input BTNU,BTND,BTNL,BTNR,BTNC,
input [7:0] color_switches, // 8 switches for 8 colors
input isBrush,
output hsync,
```

```
output vsync,
output [11:0] rgb


);

wire isVideo;
reg [11:0] rgb_reg;
wire [9:0] x, y;

reg [4:0] cursor_x = 10;
reg [4:0] cursor_y = 8;

reg [3:0] button_reg, button_prev;
wire move_up, move_down, move_left, move_right;
wire w_25MHz;
reg  [1:0] r_25MHz;
    always @(posedge clk_100MHz or posedge reset)
        if(reset)
          r_25MHz <= 0;
        else
          r_25MHz <= r_25MHz + 1;



assign w_25MHz = (r_25MHz == 0) ? 1 : 0;

assign move_up    = ~button_prev[3] & button_reg[3];
assign move_down  = ~button_prev[2] & button_reg[2];
assign move_left  = ~button_prev[1] & button_reg[1];
assign move_right = ~button_prev[0] & button_reg[0];

(* ram_style = "block" *) reg [2:0] memory [0:299];


VGA_c c(clk_100MHz,reset,isVideo,hsync,vsync,x,y);

integer i;

always @(posedge w_25MHz or posedge reset) begin
   if (reset) begin
        cursor_x <= 10;
        cursor_y <= 8;
        button_reg <= 4'b0000;
        button_prev <= 4'b0000;
```

```
        for (i = 0; i < 300; i = i + 1) begin
             memory[i] <= 3'b110;
             end

    end else begin
        button_prev <= button_reg;
        button_reg <= {BTNU, BTND, BTNL, BTNR};

        if (move_up && cursor_y > 0)
            cursor_y <= cursor_y - 1;
        if (move_down && cursor_y < 14)
            cursor_y <= cursor_y + 1;
        if (move_left && cursor_x > 0)
            cursor_x <= cursor_x - 1;
        if (move_right && cursor_x < 19)
            cursor_x <= cursor_x + 1;

        if (BTNC) begin
            if (isBrush) begin
            if (cursor_x > 0 && cursor_y > 0)
                memory[(cursor_y - 1) * 20 + (cursor_x - 1)] <=
get_selected_color(color_switches);
            if (cursor_x > 0)
                memory[cursor_y * 20 + (cursor_x - 1)] <=
get_selected_color(color_switches);
            if (cursor_x > 0 && cursor_y < 14)
                memory[(cursor_y + 1) * 20 + (cursor_x - 1)] <=
get_selected_color(color_switches);
            if (cursor_y > 0)
                memory[(cursor_y - 1) * 20 + cursor_x] <=
get_selected_color(color_switches);
            if (cursor_y < 14)
                memory[(cursor_y + 1) * 20 + cursor_x] <=
get_selected_color(color_switches);
            if (cursor_x < 19 && cursor_y > 0)
                memory[(cursor_y - 1) * 20 + (cursor_x + 1)] <=
get_selected_color(color_switches);
            if (cursor_x < 19)
                memory[cursor_y * 20 + (cursor_x + 1)] <=
get_selected_color(color_switches);
```

```
            if (cursor_x < 19 && cursor_y < 14)
                memory[(cursor_y + 1) * 20 + (cursor_x + 1)] <=
get_selected_color(color_switches);

            end
            if (cursor_x>0 && cursor_y>0 && cursor_x < 19 &&
cursor_y < 14)
            memory[cursor_y * 20 + cursor_x] <=
get_selected_color(color_switches);
        end
    end
end

reg [2:0] color;
reg [4:0] x_part;
reg [4:0] y_part;
```

always @(*) begin

```
   x_part = x >> 5;
    y_part = y >> 5;

    color = memory[y_part * 20 + x_part];

    case (color)
        3'b000: rgb_reg = 12'hF00; // Red
        3'b001: rgb_reg = 12'h0F0; // Green
        3'b010: rgb_reg = 12'h00F; // Blue
        3'b011: rgb_reg = 12'hFF0; // Yellow
        3'b100: rgb_reg = 12'hF0F; // Magenta
        3'b101: rgb_reg = 12'h0FF; // Cyan
        3'b110: rgb_reg = 12'hFFF; // White
        3'b111: rgb_reg = 12'h000; // Black
        default: rgb_reg = 12'hFFF; // Default white

    endcase

    if ((x_part == cursor_x && (y_part >= cursor_y - 2 && y_part <=
cursor_y + 2)) || (y_part == cursor_y && (x_part >= cursor_x - 2 &&
x_part <= cursor_x + 2)))
```

```
        rgb_reg = 12'h000;
end
assign rgb = (isVideo) ? rgb_reg : 12'b0;

// Function to determine selected color based on switches
function [2:0] get_selected_color;
    input [7:0] switches;
    begin
        case (switches)
            8'b00000001: get_selected_color = 3'b000; // Red
            8'b00000010: get_selected_color = 3'b001; // Green
            8'b00000100: get_selected_color = 3'b010; // Blue
            8'b00001000: get_selected_color = 3'b011; // Yellow
            8'b00010000: get_selected_color = 3'b100; // Magenta
            8'b00100000: get_selected_color = 3'b101; // Cyan
            8'b01000000: get_selected_color = 3'b110; // White
            8'b10000000: get_selected_color = 3'b111; // Black
            default:     get_selected_color = 3'b110; // Default
White
        endcase
    end
endfunction
```

endmodule

```
module part3( input clk_100MHz, reset,

 input [7:0] color_switches, // 8 switches for 8 colors input
isBrush,

 input PS2Clk, PS2Data, // PS/2 mouse interface output hsync, output
vsync, output [11:0] rgb

);

wire isVideo;
reg [11:0] rgb_reg;
wire [9:0] x, y;

reg [4:0] cursor_x = 10;
reg [4:0] cursor_y = 8;
```

```
wire [7:0] mouse_x, mouse_y;
wire mouse_x_sign, mouse_y_sign;
wire mouse_x_ov, mouse_y_ov;
wire mouse_left_click, mouse_valid;

wire w_25MHz;
reg  [1:0] r_25MHz;
always @(posedge clk_100MHz or posedge reset)
    if (reset)
        r_25MHz <= 0;
    else
        r_25MHz <= r_25MHz + 1;

assign w_25MHz = (r_25MHz == 0) ? 1 : 0;

(* ram_style = "block" *) reg [2:0] memory [0:299];

// Instantiate the VGA controller
VGA_c c(clk_100MHz, reset, isVideo, hsync, vsync, x, y);

// Instantiate the PS/2 mouse controller
ps2_mouse mouse_controller(
    .i_clk(clk_100MHz),
    .i_reset(reset),
    .i_PS2Data(PS2Data),
    .i_PS2Clk(PS2Clk),
    .o_x(mouse_x),
    .o_x_ov(mouse_x_ov),
    .o_x_sign(mouse_x_sign),
    .o_y(mouse_y),
    .o_y_ov(mouse_y_ov),
    .o_y_sign(mouse_y_sign),
    .o_r_click(),
    .o_l_click(mouse_left_click),
    .o_valid(mouse_valid)
);

integer i;

always @(posedge clk_100MHz or posedge reset) begin
    if (reset) begin
```

```
        cursor_x <= 10;
        cursor_y <= 8;

        for (i = 0; i < 300; i = i + 1) begin
            memory[i] <= 3'b110; // Initialize memory with white
        end
    end else if (mouse_valid) begin
        // Update cursor position with bounds check
        if (mouse_x > 2) begin
            if (mouse_x_sign && cursor_x > 0)
                cursor_x <= cursor_x - 1;
            else if (!mouse_x_sign && cursor_x < 19)
                cursor_x <= cursor_x + 1;
        end

        if (mouse_y > 2) begin
            if (!mouse_y_sign && cursor_y > 0)
                cursor_y <= cursor_y - 1;
            else if (mouse_y_sign && cursor_y < 14)
                cursor_y <= cursor_y + 1;
        end

        // Drawing logic
        if (mouse_left_click) begin
            if (isBrush) begin
                if (cursor_x > 0 && cursor_y > 0)
                    memory[(cursor_y - 1) * 20 + (cursor_x - 1)] <=
get_selected_color(color_switches);
                if (cursor_x > 0)
                    memory[cursor_y * 20 + (cursor_x - 1)] <=
get_selected_color(color_switches);
                if (cursor_x > 0 && cursor_y < 14)
                    memory[(cursor_y + 1) * 20 + (cursor_x - 1)] <=
get_selected_color(color_switches);
                if (cursor_y > 0)
                    memory[(cursor_y - 1) * 20 + cursor_x] <=
get_selected_color(color_switches);
                if (cursor_y < 14)
                    memory[(cursor_y + 1) * 20 + cursor_x] <=
get_selected_color(color_switches);
                if (cursor_x < 19 && cursor_y > 0)
                    memory[(cursor_y - 1) * 20 + (cursor_x + 1)] <=
```

```
get_selected_color(color_switches);
                if (cursor_x < 19)
                    memory[cursor_y * 20 + (cursor_x + 1)] <=
get_selected_color(color_switches);
                if (cursor_x < 19 && cursor_y < 14)
                    memory[(cursor_y + 1) * 20 + (cursor_x + 1)] <=
get_selected_color(color_switches);
            end
            if (cursor_x >= 0 && cursor_y >= 0 && cursor_x <= 19 &&
cursor_y <= 14)
                memory[cursor_y * 20 + cursor_x] <=
get_selected_color(color_switches);
        end
    end
end

reg [2:0] color;
reg [4:0] x_part;
reg [4:0] y_part;

always @(*) begin
    x_part = (x >> 5) < 20 ? (x >> 5) : 19;  // Bound check for
x_part
    y_part = (y >> 5) < 15 ? (y >> 5) : 14;  // Bound check for
y_part

    color = memory[y_part * 20 + x_part];

    case (color)
        3'b000: rgb_reg = 12'hF00; // Red
        3'b001: rgb_reg = 12'h0F0; // Green
        3'b010: rgb_reg = 12'h00F; // Blue
        3'b011: rgb_reg = 12'hFF0; // Yellow
        3'b100: rgb_reg = 12'hF0F; // Magenta
        3'b101: rgb_reg = 12'h0FF; // Cyan
        3'b110: rgb_reg = 12'hFFF; // White
        3'b111: rgb_reg = 12'h000; // Black
        default: rgb_reg = 12'hFFF; // Default white
    endcase

    if ((x_part == cursor_x && (y_part >= cursor_y - 1 && y_part <=
cursor_y + 1)) ||
```

```
        (y_part == cursor_y && (x_part >= cursor_x - 1 && x_part <=
cursor_x + 1)))
        rgb_reg = 12'h000; // Cursor color
end

assign rgb = (isVideo) ? rgb_reg : 12'b0;

// Function to determine selected color based on switches
function [2:0] get_selected_color;
    input [7:0] switches;
    begin
        case (switches)
            8'b00000001: get_selected_color = 3'b000; // Red
            8'b00000010: get_selected_color = 3'b001; // Green
            8'b00000100: get_selected_color = 3'b010; // Blue
            8'b00001000: get_selected_color = 3'b011; // Yellow
            8'b00010000: get_selected_color = 3'b100; // Magenta
            8'b00100000: get_selected_color = 3'b101; // Cyan
            8'b01000000: get_selected_color = 3'b110; // White
            8'b10000000: get_selected_color = 3'b111; // Black
            default:     get_selected_color = 3'b110; // Default
White
        endcase
    end
endfunction


endmodule
```

module mouse_controller(input wire clk, input wire rst, input wire ps2_data, input wire ps2_clk, output wire [7:0] pos_x, output wire pos_x_ov, output wire pos_x_sign, output wire [7:0] pos_y, output wire pos_y_ov, output wire pos_y_sign, output wire right_click, output wire left_click, output wire valid_signal);

```
reg [43:0] shift_register;
reg [5:0] bit_count;
reg [1:0] clk_sync;
reg ready_flag;
reg ps2_data_reg;
wire clk_negedge;
```

```
wire [10:0] data_packet1 = shift_register[33 +: 11];
wire [10:0] data_packet2 = shift_register[22 +: 11];
wire [10:0] data_packet3 = shift_register[11 +: 11];
wire [10:0] data_packet4 = shift_register[0 +: 11];

// Negative edge detection for ps2_clk
assign clk_negedge = (clk_sync == 2'b10);

always @(posedge clk) begin
   if (rst) begin
        shift_register <= 44'b0;
        bit_count <= 6'b0;
        clk_sync <= 2'b1;
        ready_flag <= 1'b0;
        ps2_data_reg <= 1'b0;
    end else begin
        clk_sync <= {clk_sync[0], ps2_clk};
        ps2_data_reg <= ps2_data;

        if (clk_negedge) begin
            shift_register <= {shift_register[42:0], ps2_data_reg};
            bit_count <= bit_count + 6'b1;
        end

        if (bit_count == 6'd44) begin
            bit_count <= 6'b0;
            ready_flag <= 1'b1;
        end else begin
            ready_flag <= 1'b0;
        end
    end
end

data_validator data_validator_inst(
    .clk(clk),
    .rst(rst),
    .packet1(data_packet1),
    .packet2(data_packet2),
    .packet3(data_packet3),
    .packet4(data_packet4),
    .ready_signal(ready_flag),
    .pos_x(pos_x),
```

```
    .pos_y(pos_y),
    .pos_x_overflow(pos_x_ov),
    .pos_y_overflow(pos_y_ov),
    .pos_x_sign(pos_x_sign),
    .pos_y_sign(pos_y_sign),
    .left_button(left_click),
    .right_button(right_click),
    .is_valid(valid_signal)
);
```

endmodule

module data_validator(input wire clk, input wire rst, input wire [10:0] packet1, input wire [10:0] packet2, input wire [10:0] packet3, input wire [10:0] packet4, input wire ready_signal, output reg [7:0] pos_x, output reg [7:0] pos_y, output reg pos_x_overflow, output reg pos_y_overflow, output reg pos_x_sign, output reg pos_y_sign, output reg left_button, output reg right_button, output reg is_valid);

```
wire [7:0] signal1, signal2, signal3, signal4;
wire parity_check, start_check, stop_check;
wire [3:0] parity_bits, start_bits, stop_bits;
wire [3:0] validity_checks;

// Extract signals from packets
assign {start_bits[0], signal1, parity_bits[0], stop_bits[0]} =
packet1;
assign {start_bits[1], signal2, parity_bits[1], stop_bits[1]} =
packet2;
assign {start_bits[2], signal3, parity_bits[2], stop_bits[2]} =
packet3;
assign {start_bits[3], signal4, parity_bits[3], stop_bits[3]} =
packet4;

// Perform checks
assign validity_checks = ~^{signal1, parity_bits[0], signal2,
parity_bits[1], signal3, parity_bits[2], signal4, parity_bits[3]};
assign parity_check = &validity_checks;
assign start_check = ~|start_bits;
assign stop_check = &stop_bits;

always @(posedge clk or posedge rst) begin
    if (rst) begin
```

```
        pos_x <= 8'b0;
        pos_y <= 8'b0;
        pos_x_overflow <= 1'b0;
        pos_y_overflow <= 1'b0;
        pos_x_sign <= 1'b0;
        pos_y_sign <= 1'b0;
        left_button <= 1'b0;
        right_button <= 1'b0;
        is_valid <= 1'b0;
    end else if (ready_signal) begin
        is_valid <= start_check && stop_check && parity_check;

        if (is_valid) begin
            pos_x <= signal2;
            pos_y <= signal3;
            {pos_x_overflow, pos_y_overflow} <= signal1[1:0];
            {pos_x_sign, pos_y_sign} <= signal1[3:2];
            {left_button, right_button} <= signal1[7:6];
        end
    end
end
```

endmodule