CS 223 Digital Design

Bilkent University

Fall 2024-25

Project Assignment

Video Graphics Array Driver and Drawing Canvas Design and Implementation

Nabeeha Khan

22301304

Section: 03

Date: 16th December 2024

1) PS/2 Mouse design and implementation:

Overview:

This project implements a hardware system to drive a VGA display and enable interaction via a PS/2 mouse. The system includes several key modules: VGA_Controller, MouseDisplay, MouseCtl, Ps2Interface, and blk_mem_gen_0. Together, these modules ensure graphical rendering and interactive control using a mouse.

VGA Specification:

The VGA interface operates at 640x480 resolution with a refresh rate of 60 Hz. It uses a pixel clock of 25 MHz and organizes the display in a structured manner, with 640 visible pixels out of 800 total horizontal pixels and 480 visible lines out of 525 total vertical lines.

Design and Implementation:

The VGA_Controller drives the VGA signals, while the MouseDisplay and blk_mem_gen_0 handle graphical rendering. Simultaneously, the MouseCtl and Ps2Interface modules process mouse input, enabling dynamic cursor control.

The VGA_Controller module is responsible for generating synchronization signals (hsync and vsync) and tracking the active pixel region. It uses horizontal and vertical counters to monitor the current pixel position and produces a video_enable signal that is active only during the visible area of 640x480 pixels. This module ensures the display timing aligns with the VGA standard.

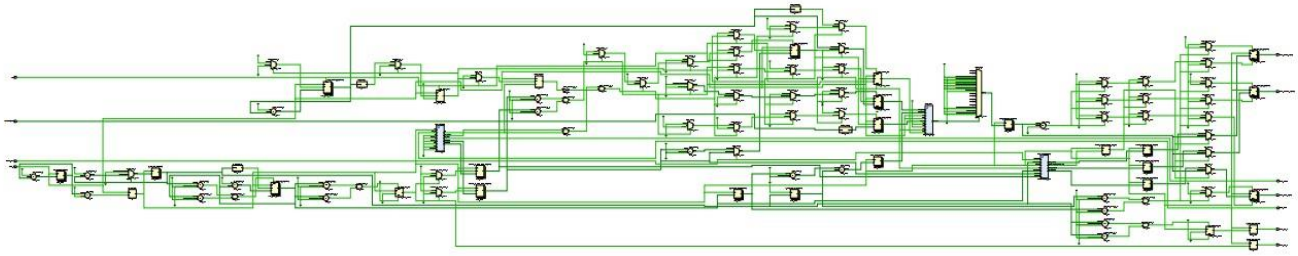
The MouseDisplay module combines mouse input with display logic to render the cursor on the VGA screen. It translates mouse movement data into x and y coordinates, ensuring that the cursor stays within screen boundaries. It interacts with the blk_mem_gen_0 module, which provides pixel color data for graphical rendering.

The MouseCtl module processes raw data packets from the PS/2 mouse. It parses the incoming data stream to extract movement deltas and button states, implementing a finite state machine (FSM) to synchronize and validate data. The processed signals, such as mouse_x, mouse_y, and mouse_btn, are then used by other modules for display and interaction.

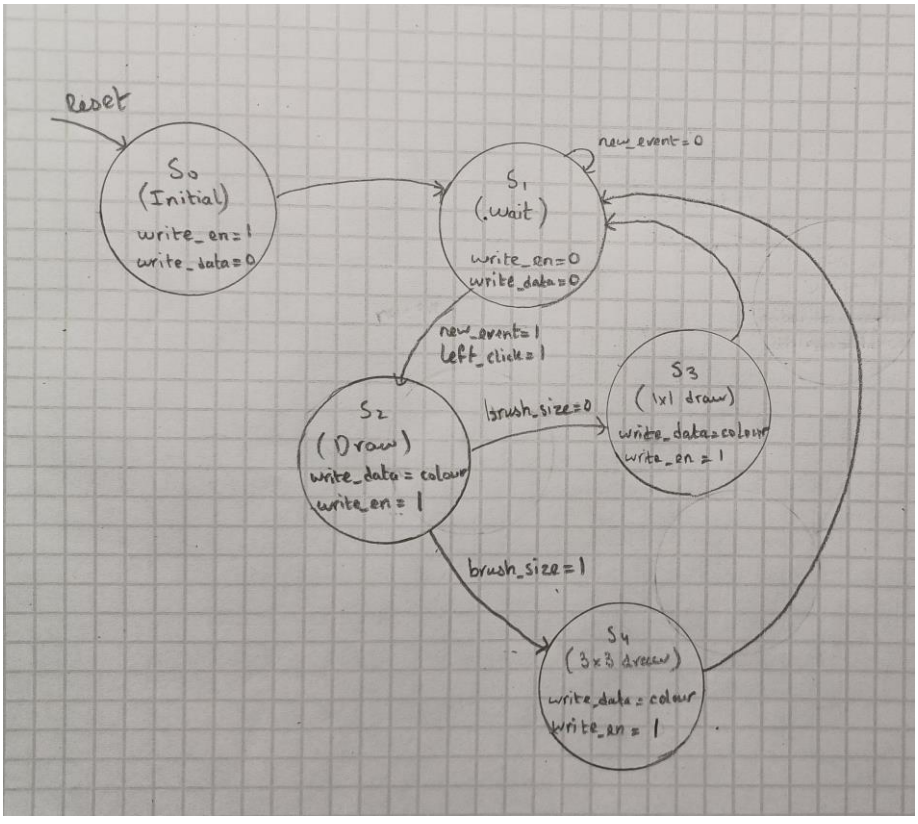
The Ps2Interface module manages communication with the PS/2 mouse. It captures the clock and data signals from the PS/2 interface and uses a state machine to synchronize data packets. Validated data is transferred to the MouseCtl module for further processing.

The blk_mem_gen_0 module acts as a frame buffer, storing pixel data for the visible area of the screen. It provides pixel color information to both the MouseDisplay and VGA_Controller modules, enabling seamless graphical output on the VGA display.

2) RTL Schematic:

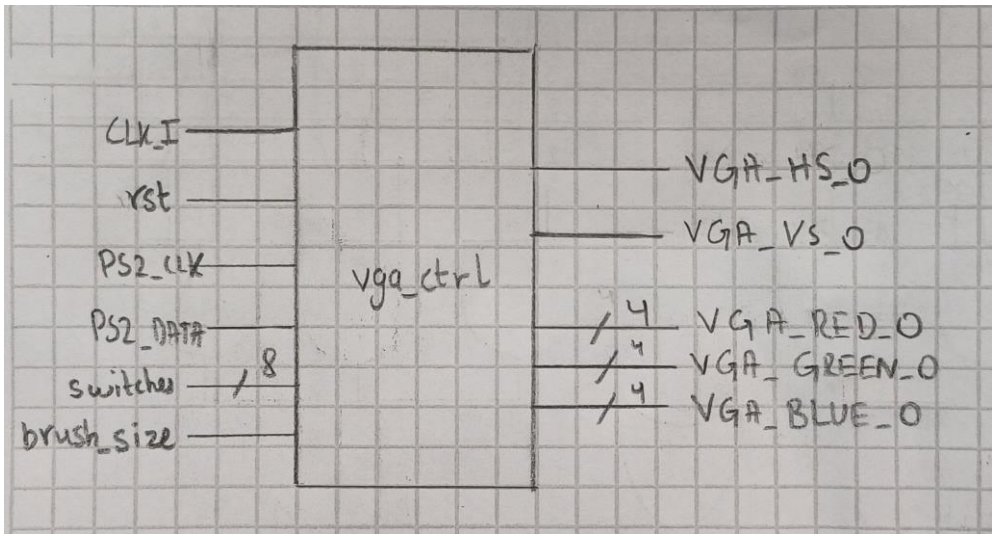


3) State Diagram:



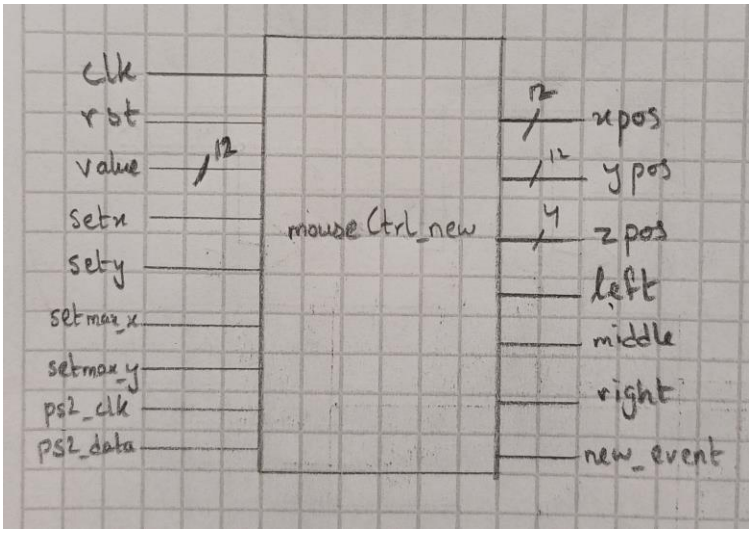
4) Block Diagram (PS/2 Mouse Interface):

Top module:

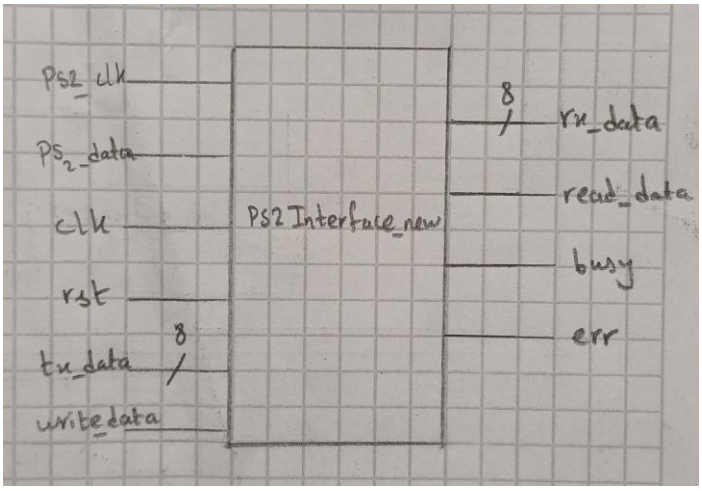


Other modules:

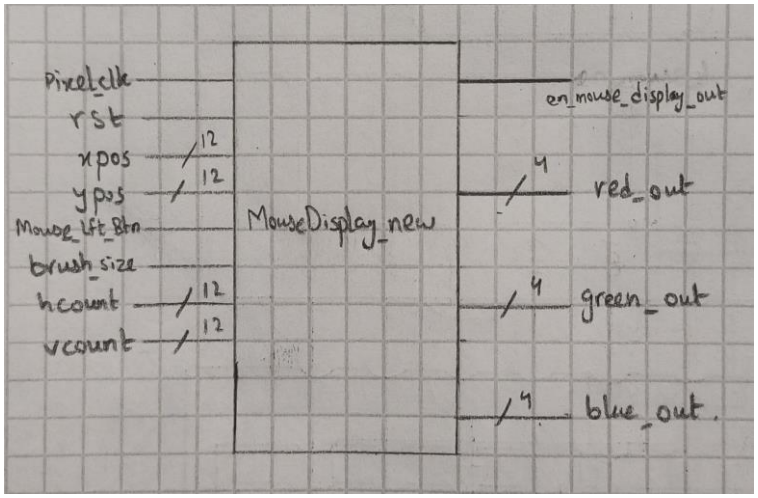
i) Mouse Controller



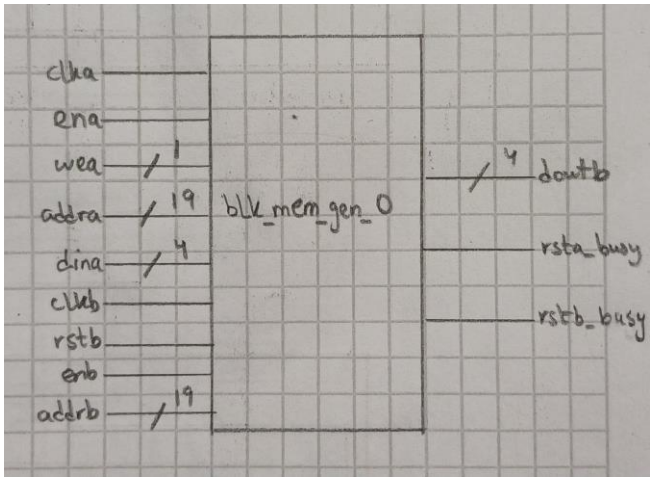
ii) PS/2 Interface



iii) Mouse display



iv) BRAM



5) Appendix:

```
//Top module implementing canvas with PS/2 mouse
module vga_ctrl(
        input wire CLK_I,
        input wire rst,
        output wire VGA_HS_O,
        output wire VGA_VS_O,
        output wire [3:0] VGA_RED_O,
        output wire [3:0] VGA_GREEN_O,
        output wire [3:0] VGA_BLUE_O,
        inout wire PS2_CLK,
        inout wire PS2_DATA,
        input wire [7:0] switches, // 8 switches for selecting brush color
        input wire brush_size );
localparam FRAME_WIDTH = 640; // Width of the active video
localparam FRAME_HEIGHT = 480; // Height of the active video
localparam H_FP = 16; // Horizontal front porch
localparam H_PW = 96; // Horizontal sync pulse width
localparam H_MAX = 800; // Total horizontal period
localparam V_FP = 10; // Vertical front porch
localparam V_PW = 2; // Vertical sync pulse width
localparam V_MAX = 525; // Total vertical period
localparam H_POL = 1'b0; // Horizontal sync polarity (negative for VGA)
localparam V_POL = 1'b0; // Vertical sync polarity (negative for VGA)

// Signals
wire pxl_clk;
reg active;
```

```
reg [11:0] h_cntr_reg = 12'd0;
reg [11:0] v_cntr_reg = 12'd0;

reg [11:0] h_cntr_reg_dly = 12'd0;
reg [11:0] v_cntr_reg_dly = 12'd0;

reg h_sync_reg = ~H_POL;
reg v_sync_reg = ~V_POL;
reg h_sync_reg_dly = ~H_POL;
reg v_sync_reg_dly = ~V_POL;

reg [3:0] vga_red;
reg [3:0] vga_green;
reg [3:0] vga_blue;
reg [3:0] vga_red_reg = 4'd0;
reg [3:0] vga_green_reg = 4'd0;
reg [3:0] vga_blue_reg = 4'd0;

// Mouse signals
wire [11:0] MOUSE_X_POS;
wire [11:0] MOUSE_Y_POS;
reg [11:0] MOUSE_X_POS_REG = 12'd0;
reg [11:0] MOUSE_Y_POS_REG = 12'd0;

wire [3:0] mouse_cursor_red;
wire [3:0] mouse_cursor_green;
wire [3:0] mouse_cursor_blue;
wire enable_mouse_display;

reg [3:0] mouse_cursor_red_dly = 4'd0;
reg [3:0] mouse_cursor_green_dly = 4'd0;
reg [3:0] mouse_cursor_blue_dly = 4'd0;
reg enable_mouse_display_dly;

// Background generation signals
reg [27:0] cntDyn = 28'd0;
integer intHcnt, intVcnt;
reg [3:0] bg_red_dly = 4'd0;
reg [3:0] bg_green_dly = 4'd0;
reg [3:0] bg_blue_dly = 4'd0;

reg [3:0] bg_red;
reg [3:0] bg_green;
reg [3:0] bg_blue;

// Intermediate signals for color computation
```

```
reg signed [31:0] tmp_red_int, tmp_green_int, tmp_blue_int;
reg [7:0] tmp8;

wire left,new_event;
reg [3:0] brush_red;
reg [3:0] brush_green;
reg [3:0] brush_blue;

 reg [18:0] read_addr;
reg [18:0] write_addr;
wire [3:0] read_data;
reg [3:0] write_data;
reg write_enable = 1'b0;


reg clk_div; reg [1:0] clk_counter = 2'b00; // Divide 100 MHz clock by 4 for 25 MHz

always @(posedge CLK_I) begin

        clk_counter <= clk_counter + 1;

                clk_div <= (clk_counter == 2'b11); // Toggle every 4 input clock cycles

end

assign pxl_clk = clk_div;


reg [3:0] brush_color;

always @(*) begin

        case (switches)

                8'b00000001: begin  brush_color= 4'b0001; end

                8'b00000010: begin brush_color= 4'b0010; end

                8'b00000100: begin brush_color= 4'b0011; end

                8'b00001000: begin  brush_color= 4'b0100; end

                8'b00010000: begin brush_color= 4'b0101; end

                8'b00100000: begin  brush_color= 4'b0110; end

                8'b01000000: begin  brush_color= 4'b0111; end

                8'b10000000: begin brush_color= 4'b1000; end
```

```
                default: begin  brush_color= 4'b0000; end
        endcase
end

wire [11:0] color_lut [15:0];
assign color_lut[4'b0000] = 12'h000; // Black assign
color_lut[4'b0001] = 12'hF00; // Red assign
color_lut[4'b0010] = 12'h0F0; // Green assign
color_lut[4'b0011] = 12'h00F; // Blue assign
color_lut[4'b0100] = 12'hFF0; // Yellow assign
color_lut[4'b0101] = 12'h0FF; // Cyan assign
color_lut[4'b0110] = 12'hF0F; // Magenta assign
color_lut[4'b0111] = 12'hFFF; // White
assign color_lut[4'b1000] = 12'h000; // Black

// Mouse Controller instance
MouseCtl_new #(
   .SYSCLK_FREQUENCY_HZ(108000000),
   .CHECK_PERIOD_MS(500),
   .TIMEOUT_PERIOD_MS(100)
) Inst_MouseCtl (
   .clk(pxl_clk),
   .rst(1'b0),
   .xpos(MOUSE_X_POS),
   .ypos(MOUSE_Y_POS),
   .zpos(), // open
   .left(left), // open
   .middle(), // open
   .right(), // open
   .new_event(new_event), // open
   .value(12'h000),
   .setx(1'b0),
   .sety(1'b0),
   .setmax_x(1'b0),
   .setmax_y(1'b0),
```

```
    .ps2_clk(PS2_CLK),
    .ps2_data(PS2_DATA)
);


always @(posedge pxl_clk) begin

        if (h_cntr_reg == H_MAX - 1) h_cntr_reg <= 12'd0;

        else h_cntr_reg <= h_cntr_reg + 1;

end

always @(posedge pxl_clk) begin

        if ((h_cntr_reg == H_MAX - 1) && (v_cntr_reg == V_MAX - 1))

                v_cntr_reg <= 12'd0;

        else if (h_cntr_reg == H_MAX - 1)

        v_cntr_reg <= v_cntr_reg + 1;

end

always @(posedge pxl_clk) begin

        if ((h_cntr_reg >= (H_FP + FRAME_WIDTH - 1)) && (h_cntr_reg < (H_FP +
FRAME_WIDTH + H_PW - 1)))

                h_sync_reg <= H_POL;

        else

                h_sync_reg <= ~H_POL;

end

always @(posedge pxl_clk) begin

        if ((v_cntr_reg >= (V_FP + FRAME_HEIGHT - 1)) && (v_cntr_reg < (V_FP +
FRAME_HEIGHT + V_PW - 1)))

                v_sync_reg <= V_POL;

        else

                v_sync_reg <= ~V_POL;

end
```

```
// Active region signal
always @(*) begin
   if (h_cntr_reg_dly < FRAME_WIDTH && v_cntr_reg_dly < FRAME_HEIGHT)
      active = 1'b1;
   else
      active = 1'b0;
end


always @(posedge pxl_clk) begin

        if (v_sync_reg == V_POL) begin

                MOUSE_X_POS_REG <= MOUSE_X_POS;

                MOUSE_Y_POS_REG <= MOUSE_Y_POS;

        end

end

// On a new mouse event and left button pressed, set the corresponding pixel with

//the selected color

reg prev_left;

 reg[3:0] prev_addr;

reg [18:0] addr_counter; // 19 bits to count up to 307199

reg init_done; // Flag to signal completion

always @ (posedge CLK_I) begin

        if (rst) begin

                addr_counter <= 0;

                write_enable <= 1'b1; // Enable write during initialization

                write_data <= 'b0; // Data to clear BRAM

                init_done <= 1'b0; // Reset completion flag

        end

        else if (!init_done) begin

                write_addr <= addr_counter;
```

```
                addr_counter <= addr_counter + 1;

   if (addr_counter == 307199) begin
       write_enable <= 1'b0; // Stop writing after clearing all entries
       init_done <= 1'b1;   // Set initialization done flag
    end
end

else begin

        if (new_event == 1'b1 && left == 1'b1 && !prev_left) begin

                if (MOUSE_X_POS_REG < FRAME_WIDTH && MOUSE_Y_POS_REG <
        FRAME_HEIGHT) begin

                // Compute linear address

                        if (brush_size) begin

                                for(int i=0; i<=9; i++) begin

                                        write_addr <= prev_addr;

                                        prev_addr<= (MOUSE_Y_POS_REG+i) *
                                FRAME_WIDTH + MOUSE_X_POS_REG+i;

                                        write_data <= brush_color; write_enable <= 1'b1;

                                end

                        end

                        else begin

                                write_addr <= prev_addr;

                                prev_addr<= MOUSE_Y_POS_REG * FRAME_WIDTH +
                                MOUSE_X_POS_REG;

                                write_data <= brush_color;

                                write_enable <= 1'b1;

                        end

                end

                else begin

                        write_enable <= 1'b0;
```

```
                write_enable <= 1'b0;
                write_data <='b0;
                prev_addr <= 'b0;
            end
        end
        else begin
            write_enable <= 1'b0;
            write_enable <= 1'b0;
            write_data <='b0;
            prev_addr <= 'b0;
        end
        prev_left <= left;
        end
 end
reg [11:0] pixel_color_reg; // Stores the RGB color of the current pixel
always @(posedge pxl_clk) begin
// Read the RGB value from memory
        read_addr <= v_cntr_reg_dly * FRAME_WIDTH + h_cntr_reg_dly;
        pixel_color_reg <= color_lut[read_data];
end
always @(*) begin
        bg_red = 4'hF;
        bg_green = 4'hF;
        bg_blue = 4'hF;
end
blk_mem_gen_0 Inst_bram (
```

```
        .clka(CLK_I),
        .ena('b1), // Clock
        .wea(write_enable), // Write enable
        .addra(write_addr), // Write address
        .dina(write_data), // Write data
        .clkb(CLK_I),
        .rstb(rst),
         .enb('b1), // Read clock
        .addrb(read_addr), // Read address
        .doutb(read_data) // Read data
);

// MouseDisplay instance
MouseDisplay_new Inst_MouseDisplay (
    .pixel_clk(pxl_clk),
    .rst(rst),
    .xpos(MOUSE_X_POS_REG),
    .ypos(MOUSE_Y_POS_REG),
    .brush_size(brush_size),
    .hcount(h_cntr_reg),
    .vcount(v_cntr_reg),
    .enable_mouse_display_out(enable_mouse_display),
    .red_out(mouse_cursor_red),
    .green_out(mouse_cursor_green),
    .blue_out(mouse_cursor_blue)
);

// Pipeline and register outputs
always @(posedge pxl_clk) begin
    bg_red_dly   <= bg_red;
    bg_green_dly <= bg_green;
    bg_blue_dly  <= bg_blue;

    mouse_cursor_red_dly   <= mouse_cursor_red;
    mouse_cursor_green_dly <= mouse_cursor_green;
    mouse_cursor_blue_dly  <= mouse_cursor_blue;

    enable_mouse_display_dly <= enable_mouse_display;
```

```
   h_cntr_reg_dly <= h_cntr_reg;
   v_cntr_reg_dly <= v_cntr_reg;
end


always @(*) begin

        // Default to white background

                vga_red = 4'hF;

                vga_green = 4'hF;

                vga_blue = 4'hF;

        // If the pixel has a stored color in memory, use that color
        if (pixel_color_reg != 12'b0) begin

        // Check if the pixel is not black
                vga_red   = pixel_color_reg[11:8]; // Red component
                vga_green = pixel_color_reg[7:4];  // Green component
                vga_blue  = pixel_color_reg[3:0];  // Blue component
        end

        // Overlay mouse cursor if enabled
        if (enable_mouse_display_dly == 1'b1) begin
                vga_red   = mouse_cursor_red_dly;
                vga_green = mouse_cursor_green_dly;
                vga_blue  = mouse_cursor_blue_dly;
        end


end

// Turn off RGB outside active area by ANDing with 'active'
wire [3:0] vga_red_cmb   = (active) ? vga_red   : 4'd0;
wire [3:0] vga_green_cmb = (active) ? vga_green : 4'd0;
wire [3:0] vga_blue_cmb  = (active) ? vga_blue  : 4'd0;

always @(posedge pxl_clk) begin
   v_sync_reg_dly <= v_sync_reg;
   h_sync_reg_dly <= h_sync_reg;
   vga_red_reg    <= vga_red_cmb;
   vga_green_reg  <= vga_green_cmb;
   vga_blue_reg   <= vga_blue_cmb;
end

// Assign outputs
assign VGA_HS_O    = h_sync_reg_dly;
```

```
assign VGA_VS_O    = v_sync_reg_dly;
assign VGA_RED_O   = vga_red_reg;
assign VGA_GREEN_O = vga_green_reg;
assign VGA_BLUE_O  = vga_blue_reg;


endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Module for displaying cursor

module MouseDisplay_new (

        input logic pixel_clk,

        input logic rst,

        input logic [11:0] xpos,

        input logic [11:0] ypos,

        input logic MOUSE_LEFT_BUTTON,

        input logic brush_size,

        input logic [11:0] hcount,

        input logic [11:0] vcount,

        output logic enable_mouse_display_out,

        output logic [3:0] red_out,

        output logic [3:0] green_out,

        output logic [3:0] blue_out

        );

// Cursor ROM constants
typedef logic [1:0] pixel_t;
typedef pixel_t rom_t [0:255];


localparam pixel_t mouserom[0:255] = '{


"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
```

```
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"00","00","00","00","00","00","00","00","00","00","00","00","00","00","00","00",
"00","00","00","00","00","00","00","00","00","00","00","00","00","00","00","00",
"00","00","00","00","00","00","00","00","00","00","00","00","00","00","00","00",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11" };

localparam pixel_t mouserom2[0:255] =
'{ "11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11",
"11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","00","00","00","00","00","00","00","00","00","11","11","11","11",
"11","11","11","00","00","00","00","00","00","00","00","00","11","11","11","11",
"11","11","11","00","00","00","00","00","00","00","00","00","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","00","00","00","11","11","11","11","11","11","11",
"11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11",
"11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11",
"11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11" };

localparam logic [4:0] OFFSET = 5'd16; // Cursor width and height

// Internal signals
logic [3:0] xdiff, ydiff;
pixel_t mousepixel;
logic enable_mouse_display;

// Compute xdiff
always_comb begin
xdiff = hcount[3:0] - xpos[3:0];
end

// Compute ydiff
always_comb begin
    ydiff = vcount[3:0] - ypos[3:0];
```

```
end

// Read pixel from ROM
always_ff @(posedge pixel_clk) begin
if(brush_size)
   mousepixel <= mouserom[{ydiff, xdiff}];
else
   mousepixel <= mouserom2[{ydiff, xdiff}];
end

// Enable cursor display
always_ff @(posedge pixel_clk) begin
   if ((hcount >= xpos + 1 && hcount < (xpos + OFFSET - 1)) &&
      (vcount >= ypos && vcount < (ypos + OFFSET)) &&
      (mousepixel == 2'b00 || mousepixel == 2'b01)) begin
      enable_mouse_display <= 1;
   end else begin
      enable_mouse_display <= 0;
   end
end

assign enable_mouse_display_out = enable_mouse_display;

// Set output color channels
always_ff @(posedge pixel_clk) begin
   if (enable_mouse_display) begin
      case (mousepixel)
         2'b01: begin // White pixel
            red_out <= 4'b1111;
            green_out <= 4'b1111;
            blue_out <= 4'b1111;
         end
         2'b00: begin // Black pixel
            red_out <= 4'b0000;
            green_out <= 4'b0000;
            blue_out <= 4'b0000;
         end
      endcase

   end
end
endmodule

//////////////////////////////////////////////////////////////////////////////////

//Module to determine the position of mouse from PS/2 mouse interface

module MouseCtl_new #(
```

```
        parameter SYSCLK_FREQUENCY_HZ = 100000000,
        parameter CHECK_PERIOD_MS = 500,
        parameter TIMEOUT_PERIOD_MS = 100
)(
        input wire clk,
        input wire rst,
        output reg [11:0] xpos,
        output reg [11:0] ypos,
        output reg [3:0] zpos,
        output reg left,
        output reg middle,
        output reg right,
        output reg new_event,
        input wire [11:0] value,
        input wire setx,
        input wire sety,
        input wire setmax_x,
        input wire setmax_y,
        inout  ps2_clk,
        inout ps2_data
 );
// CONSTANTS
localparam [7:0] FA = 8'hFA; // ACK
localparam [7:0] FF = 8'hFF; // RESET
localparam [7:0] AA = 8'hAA; // BAT_OK
localparam [7:0] OO = 8'h00; // ID = 00h

localparam [7:0] READ_ID  = 8'hF2;
localparam [7:0] ENABLE_REPORTING= 8'hF4;
localparam [7:0] SET_RESOLUTION  = 8'hE8;
localparam [7:0] RESOLUTION = 8'h03; // 8 counts/mm
localparam [7:0] SET_SAMPLE_RATE = 8'hF3;
localparam [7:0] SAMPLE_RATE  = 8'h28; // 40 samples/s

localparam [11:0] DEFAULT_MAX_X = 12'h4FF; // 1279 decimal
localparam [11:0] DEFAULT_MAX_Y = 12'h3FF; // 1023 decimal
```

```
localparam integer CHECK_PERIOD_CLOCKS =
(CHECK_PERIOD_MS*1000000)/(1000000000/SYSCLK_FREQUENCY_HZ);
localparam integer TIMEOUT_PERIOD_CLOCKS =
(TIMEOUT_PERIOD_MS*1000000)/(1000000000/SYSCLK_FREQUENCY_HZ);

// SIGNALS
reg haswheel = 1'b0;

// Positions and increments
reg [11:0] x_pos = 12'd0;
reg [11:0] y_pos = 12'd0;

reg x_overflow = 1'b0;
reg y_overflow = 1'b0;

reg x_sign = 1'b0;
reg y_sign = 1'b0;

reg [7:0] x_inc = 8'd0;
reg [7:0] y_inc = 8'd0;

reg x_new = 1'b0;
reg y_new = 1'b0;

reg [11:0] x_max = DEFAULT_MAX_X;
reg [11:0] y_max = DEFAULT_MAX_Y;

reg left_down   = 1'b0;
reg middle_down = 1'b0;
reg right_down  = 1'b0;

// FSM states
localparam [5:0]
  RESET=0, RESET_WAIT_ACK=1, RESET_WAIT_BAT_COMPLETION=2,
RESET_WAIT_ID=3,
  RESET_SET_SAMPLE_RATE_200=4,
RESET_SET_SAMPLE_RATE_200_WAIT_ACK=5,
  RESET_SEND_SAMPLE_RATE_200=6,
RESET_SEND_SAMPLE_RATE_200_WAIT_ACK=7,
  RESET_SET_SAMPLE_RATE_100=8,
RESET_SET_SAMPLE_RATE_100_WAIT_ACK=9,
  RESET_SEND_SAMPLE_RATE_100=10,
RESET_SEND_SAMPLE_RATE_100_WAIT_ACK=11,
  RESET_SET_SAMPLE_RATE_80=12,
RESET_SET_SAMPLE_RATE_80_WAIT_ACK=13,
  RESET_SEND_SAMPLE_RATE_80=14,
RESET_SEND_SAMPLE_RATE_80_WAIT_ACK=15,
```

```
  RESET_READ_ID=16, RESET_READ_ID_WAIT_ACK=17,
RESET_READ_ID_WAIT_ID=18,
  RESET_SET_RESOLUTION=19, RESET_SET_RESOLUTION_WAIT_ACK=20,
  RESET_SEND_RESOLUTION=21, RESET_SEND_RESOLUTION_WAIT_ACK=22,
  RESET_SET_SAMPLE_RATE_40=23,
RESET_SET_SAMPLE_RATE_40_WAIT_ACK=24,
  RESET_SEND_SAMPLE_RATE_40=25,
RESET_SEND_SAMPLE_RATE_40_WAIT_ACK=26,
  RESET_ENABLE_REPORTING=27, RESET_ENABLE_REPORTING_WAIT_ACK=28,
  READ_BYTE_1=29, READ_BYTE_2=30, READ_BYTE_3=31, READ_BYTE_4=32,
  CHECK_READ_ID=33, CHECK_READ_ID_WAIT_ACK=34,
CHECK_READ_ID_WAIT_ID=35,
  MARK_NEW_EVENT=36;

reg [5:0] state = RESET;

// Ps2Interface_new signals
wire read_data;
wire err;
wire [7:0] rx_data;
reg  [7:0] tx_data;
reg  write_data;

// Periodic checking
reg [31:0] periodic_check_cnt = 0;
reg reset_periodic_check_cnt = 1'b0;
wire periodic_check_tick = (periodic_check_cnt == (CHECK_PERIOD_CLOCKS-1));

// Timeout checking
reg [31:0] timeout_cnt = 0;
reg reset_timeout_cnt = 1'b0;
wire timeout = (timeout_cnt == (TIMEOUT_PERIOD_CLOCKS - 1));

// Instantiate the Ps2Interface_new module
Ps2Interface_new ps2_inst (
   .ps2_clk(ps2_clk),
   .ps2_data(ps2_data),
   .clk(clk),
   .rst(rst),
   .tx_data(tx_data),
   .write_data(write_data),
   .rx_data(rx_data),
   .read_data(read_data),
   .busy(), // not used
   .err(err)
);

// PERIODIC CHECK COUNTER
```

```
always @(posedge clk) begin
   if (reset_periodic_check_cnt)
      periodic_check_cnt <= 0;
   else if (periodic_check_cnt == (CHECK_PERIOD_CLOCKS - 1))
      periodic_check_cnt <= 0;
   else
      periodic_check_cnt <= periodic_check_cnt + 1;
end

// TIMEOUT COUNTER
always @(posedge clk) begin
   if (reset_timeout_cnt)
      timeout_cnt <= 0;
   else if (timeout_cnt < (TIMEOUT_PERIOD_CLOCKS - 1))
      timeout_cnt <= timeout_cnt + 1;
   // else stay at max
end

// Set xpos, ypos outputs and buttons at clock edge
// Just replicate internal registers on outputs
always @(posedge clk) begin
   left   <= left_down;
   middle <= middle_down;
   right  <= right_down;
   xpos   <= x_pos;
   ypos   <= y_pos;
end

// zpos is updated within FSM states (when read_byte_4),
// new_event is updated in FSM as well.

// Position updating logic for X
always @(posedge clk) begin
   if (rst) begin
      x_pos <= 12'd0;
   end else if (setx) begin
      x_pos <= value;
   end else if (x_new) begin
      // compute increments
      reg [11:0] inc;
      reg [11:0] x_inter;
      if (x_sign) begin
         // negative movement
         if (x_overflow)
            inc = 12'b111000000000; // -256
         else
            inc = {4'b1111, x_inc}; // sign extend
         x_inter = x_pos + inc;
```

```
            // if negative overflow
            if (x_inter[11] == 1'b1)
                x_pos <= 12'd0;
            else
                x_pos <= x_inter;
        end else begin
            // positive movement
            if (x_overflow)
                inc = 12'b000100000000; // +256
            else
                inc = {4'b0000, x_inc}; // sign extend
            x_inter = x_pos + inc;
            // check upper bound
            if (x_inter > {1'b0,x_max})
                x_pos <= x_max;
            else
                x_pos <= x_inter;
        end
    end
end

// Position updating logic for Y
// Y axis is inverted
always @(posedge clk) begin
    if (rst) begin
        y_pos <= 12'd0;
    end else if (sety) begin
        y_pos <= value;
    end else if (y_new) begin
        reg [11:0] inc;
        reg [11:0] y_inter;
        if (y_sign) begin
            // negative movement (inverted axis)
            if (y_overflow)
                inc = 12'b111100000000; // -256
            else
                inc = {4'b1111, y_inc}; // sign extend
            y_inter = y_pos + inc;
            if (y_inter[11] == 1'b1)
                y_pos <= 12'd0;
            else
                y_pos <= y_inter;
        end else begin
            // positive movement
            if (y_overflow)
                inc = 12'b000100000000; // +256
            else
                inc = {4'b0000, y_inc}; // sign extend
```

```
            y_inter = y_pos + inc;
            if (y_inter > y_max)
               y_pos <= y_max;
            else
               y_pos <= y_inter;
         end
      end
   end

   // Set maximum x value
   always @(posedge clk) begin
      if (rst)
         x_max <= DEFAULT_MAX_X;
      else if (setmax_x)
         x_max <= value;
   end

   // Set maximum y value
   always @(posedge clk) begin
      if (rst)
         y_max <= DEFAULT_MAX_Y;
      else if (setmax_y)
         y_max <= value;
   end

   // MAIN FSM
   always @(posedge clk or posedge rst) begin
      if (rst) begin
         state <= RESET;
         haswheel <= 1'b0;
         x_overflow <= 1'b0;
         y_overflow <= 1'b0;
         x_sign <= 1'b0;
         y_sign <= 1'b0;
         x_inc <= 8'd0;
         y_inc <= 8'd0;
         x_new <= 1'b0;
         y_new <= 1'b0;
         new_event <= 1'b0;
         left_down <= 1'b0;
         middle_down <= 1'b0;
         right_down <= 1'b0;
         reset_periodic_check_cnt <= 1'b1;
         reset_timeout_cnt <= 1'b1;
         write_data <= 1'b0;
         zpos <= 4'd0;
      end else begin
         // Default assignments for each cycle
```

```
write_data <= 1'b0;
x_new <= 1'b0;
y_new <= 1'b0;

case (state)
   RESET: begin
      haswheel <= 1'b0;
      x_overflow <= 1'b0;
      y_overflow <= 1'b0;
      x_sign <= 1'b0;
      y_sign <= 1'b0;
      x_inc <= 8'd0;
      y_inc <= 8'd0;
      x_new <= 1'b0;
      y_new <= 1'b0;
      left_down <= 1'b0;
      middle_down <= 1'b0;
      right_down <= 1'b0;
      tx_data <= FF;
      write_data <= 1'b1;
      reset_periodic_check_cnt <= 1'b1;
      reset_timeout_cnt <= 1'b1;
      state <= RESET_WAIT_ACK;
   end

   RESET_WAIT_ACK: begin
      if (read_data) begin
         if (rx_data == FA)
            state <= RESET_WAIT_BAT_COMPLETION;
         else
            state <= RESET;
      end else if (err)
         state <= RESET;
   end

   RESET_WAIT_BAT_COMPLETION: begin
      if (read_data) begin
         if (rx_data == AA)
            state <= RESET_WAIT_ID;
         else
            state <= RESET;
      end else if (err)
         state <= RESET;
   end

   RESET_WAIT_ID: begin
      if (read_data) begin
         if (rx_data == OO)
```

```
            state <= RESET_SET_SAMPLE_RATE_200;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_SET_SAMPLE_RATE_200: begin
    tx_data <= SET_SAMPLE_RATE;
    write_data <= 1'b1;
    state <= RESET_SET_SAMPLE_RATE_200_WAIT_ACK;
end

RESET_SET_SAMPLE_RATE_200_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
            state <= RESET_SEND_SAMPLE_RATE_200;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_SEND_SAMPLE_RATE_200: begin
    tx_data <= 8'hC8; //200 decimal
    write_data <= 1'b1;
    state <= RESET_SEND_SAMPLE_RATE_200_WAIT_ACK;
end

RESET_SEND_SAMPLE_RATE_200_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
            state <= RESET_SET_SAMPLE_RATE_100;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_SET_SAMPLE_RATE_100: begin
    tx_data <= SET_SAMPLE_RATE;
    write_data <= 1'b1;
    state <= RESET_SET_SAMPLE_RATE_100_WAIT_ACK;
end

RESET_SET_SAMPLE_RATE_100_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
```

```
            state <= RESET_SEND_SAMPLE_RATE_100;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_SEND_SAMPLE_RATE_100: begin
    tx_data <= 8'h64; //100 decimal
    write_data <= 1'b1;
    state <= RESET_SEND_SAMPLE_RATE_100_WAIT_ACK;
end

RESET_SEND_SAMPLE_RATE_100_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
            state <= RESET_SET_SAMPLE_RATE_80;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_SET_SAMPLE_RATE_80: begin
    tx_data <= SET_SAMPLE_RATE;
    write_data <= 1'b1;
    state <= RESET_SET_SAMPLE_RATE_80_WAIT_ACK;
end

RESET_SET_SAMPLE_RATE_80_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
            state <= RESET_SEND_SAMPLE_RATE_80;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_SEND_SAMPLE_RATE_80: begin
    tx_data <= 8'h50; //80 decimal
    write_data <= 1'b1;
    state <= RESET_SEND_SAMPLE_RATE_80_WAIT_ACK;
end

RESET_SEND_SAMPLE_RATE_80_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
```

```
            state <= RESET_READ_ID;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_READ_ID: begin
    tx_data <= READ_ID;
    write_data <= 1'b1;
    state <= RESET_READ_ID_WAIT_ACK;
end

RESET_READ_ID_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
            state <= RESET_READ_ID_WAIT_ID;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_READ_ID_WAIT_ID: begin
    if (read_data) begin
        if (rx_data == 8'h00) begin
            haswheel <= 1'b0;
            state <= RESET_SET_RESOLUTION;
        end else if (rx_data == 8'h03) begin
            haswheel <= 1'b1;
            state <= RESET_SET_RESOLUTION;
        end else
            state <= RESET;
    end else if (err)
        state <= RESET;
end

RESET_SET_RESOLUTION: begin
    tx_data <= SET_RESOLUTION;
    write_data <= 1'b1;
    state <= RESET_SET_RESOLUTION_WAIT_ACK;
end

RESET_SET_RESOLUTION_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
            state <= RESET_SEND_RESOLUTION;
        else
```

```
            state <= RESET;
        end else if (err)
            state <= RESET;
    end

    RESET_SEND_RESOLUTION: begin
        tx_data <= RESOLUTION;
        write_data <= 1'b1;
        state <= RESET_SEND_RESOLUTION_WAIT_ACK;
    end

    RESET_SEND_RESOLUTION_WAIT_ACK: begin
        if (read_data) begin
            if (rx_data == FA)
                state <= RESET_SET_SAMPLE_RATE_40;
            else
                state <= RESET;
        end else if (err)
            state <= RESET;
    end

    RESET_SET_SAMPLE_RATE_40: begin
        tx_data <= SET_SAMPLE_RATE;
        write_data <= 1'b1;
        state <= RESET_SET_SAMPLE_RATE_40_WAIT_ACK;
    end

    RESET_SET_SAMPLE_RATE_40_WAIT_ACK: begin
        if (read_data) begin
            if (rx_data == FA)
                state <= RESET_SEND_SAMPLE_RATE_40;
            else
                state <= RESET;
        end else if (err)
            state <= RESET;
    end

    RESET_SEND_SAMPLE_RATE_40: begin
        tx_data <= SAMPLE_RATE;
        write_data <= 1'b1;
        state <= RESET_SEND_SAMPLE_RATE_40_WAIT_ACK;
    end

    RESET_SEND_SAMPLE_RATE_40_WAIT_ACK: begin
        if (read_data) begin
            if (rx_data == FA)
                state <= RESET_ENABLE_REPORTING;
            else
```

```
            state <= RESET;
        end else if (err)
            state <= RESET;
    end

    RESET_ENABLE_REPORTING: begin
        tx_data <= ENABLE_REPORTING;
        write_data <= 1'b1;
        state <= RESET_ENABLE_REPORTING_WAIT_ACK;
    end

    RESET_ENABLE_REPORTING_WAIT_ACK: begin
        if (read_data) begin
            if (rx_data == FA)
                state <= READ_BYTE_1;
            else
                state <= RESET;
        end else if (err)
            state <= RESET;
    end

    READ_BYTE_1: begin
        reset_periodic_check_cnt <= 1'b0;
        new_event <= 1'b0;
        zpos <= 4'd0;
        if (read_data) begin
            left_down <= rx_data[0];
            middle_down <= rx_data[2];
            right_down <= rx_data[1];
            x_sign <= rx_data[4];
            y_sign <= ~rx_data[5];
            x_overflow <= rx_data[6];
            y_overflow <= rx_data[7];
            state <= READ_BYTE_2;
        end else if (periodic_check_tick) begin
            state <= CHECK_READ_ID;
        end else begin
            // stay in READ_BYTE_1
        end
    end

    READ_BYTE_2: begin
        if (read_data) begin
            x_inc <= rx_data;
            x_new <= 1'b1;
            state <= READ_BYTE_3;
        end else if (periodic_check_tick) begin
            state <= CHECK_READ_ID;
```

```
    end else if (err) begin
        state <= RESET;
    end
end

READ_BYTE_3: begin
    if (read_data) begin
        if (rx_data != 8'h00) begin
            y_inc <= (~rx_data) + 8'h01;
            y_new <= 1'b1;
        end
        if (haswheel)
            state <= READ_BYTE_4;
        else
            state <= MARK_NEW_EVENT;
    end else if (periodic_check_tick) begin
        state <= CHECK_READ_ID;
    end else if (err) begin
        state <= RESET;
    end
end

READ_BYTE_4: begin
    if (read_data) begin
        zpos <= rx_data[3:0];
        state <= MARK_NEW_EVENT;
    end else if (periodic_check_tick) begin
        state <= CHECK_READ_ID;
    end else if (err) begin
        state <= RESET;
    end
end

CHECK_READ_ID: begin
    reset_timeout_cnt <= 1'b0;
    tx_data <= READ_ID;
    write_data <= 1'b1;
    state <= CHECK_READ_ID_WAIT_ACK;
end

CHECK_READ_ID_WAIT_ACK: begin
    if (read_data) begin
        if (rx_data == FA)
            state <= CHECK_READ_ID_WAIT_ID;
        else
            state <= RESET;
    end else if (err)
        state <= RESET;
```

```
            else if (timeout)
                state <= RESET;
        end

        CHECK_READ_ID_WAIT_ID: begin
            if (read_data) begin
                if ((rx_data == 8'h00) || (rx_data == 8'h03)) begin
                    reset_timeout_cnt <= 1'b1;
                    state <= READ_BYTE_1;
                end else
                    state <= RESET;
            end else if (err)
                state <= RESET;
            else if (timeout)
                state <= RESET;
        end

        MARK_NEW_EVENT: begin
            new_event <= 1'b1;
            state <= READ_BYTE_1;
        end

        default: state <= RESET;
      endcase
   end
end


endmodule
```

//

```
//Module to handle communication with PS/2 interface
module Ps2Interface_new(
        inout ps2_clk,
        inout ps2_data,
        input clk,
        input rst,
        input [7:0] tx_data,
        input write_data,
        output reg [7:0] rx_data,
        output reg read_data,
        output busy,
```

```
        output reg err
);
// CONSTANTS
localparam [13:0] DELAY_100US = 14'b10011100010000; // 10000 decimal = 100us
localparam [10:0] DELAY_20US     = 11'b11111010000;    // 2000 decimal = 20us
localparam [6:0]  DELAY_63CLK    = 7'b1111111;         // 63 decimal
localparam [3:0]  DEBOUNCE_DELAY = 4'b1111;
localparam [3:0]  NUMBITS        = 4'd11;              // 11 bits per frame
localparam        PARITY_BIT     = 9;                  // parity bit index in frame

// SIGNALS
reg [13:0] delay_100us_count = 14'd0;
reg [10:0] delay_20us_count  = 11'd0;
reg [6:0]  delay_63clk_count = 7'd0;

reg delay_100us_done, delay_20us_done, delay_63clk_done;
reg delay_100us_counter_enable = 1'b0;
reg delay_20us_counter_enable  = 1'b0;
reg delay_63clk_counter_enable = 1'b0;

// debounced and synchronized ps2 signals
reg ps2_clk_clean = 1'b1, ps2_data_clean = 1'b1;
reg [3:0] clk_count = 4'd0, data_count = 4'd0;
reg clk_inter = 1'b1, data_inter = 1'b1;
reg ps2_clk_s = 1'b1, ps2_data_s = 1'b1;

reg ps2_clk_h = 1'b1, ps2_data_h = 1'b1;

// FSM states
typedef enum logic [4:0] {
    IDLE,
    RX_CLK_H,
    RX_CLK_L,
    RX_DOWN_EDGE,
    RX_ERROR_PARITY,
    RX_DATA_READY,
    TX_FORCE_CLK_L,
    TX_BRING_DATA_DOWN,
    TX_RELEASE_CLK,
    TX_FIRST_WAIT_DOWN_EDGE,
    TX_CLK_L,
    TX_WAIT_UP_EDGE,
    TX_CLK_H,
    TX_WAIT_UP_EDGE_BEFORE_ACK,
    TX_WAIT_ACK,
    TX_RECEIVED_ACK,
    TX_ERROR_NO_ACK
```

```
} fsm_state_t;

fsm_state_t state = IDLE;

reg [10:0] frame = 11'd0;
reg [3:0]  bit_count = 4'd0;
reg reset_bit_count = 1'b0;
reg shift_frame = 1'b0;
reg load_tx_data = 1'b0;
reg load_rx_data = 1'b0;

// Compute odd parity bits directly:
// For odd parity: parity bit = ~^data
// ^ is reduction XOR. If data has an even number of 1s, ^data=0, parity=1.
// If data has an odd number of 1s, ^data=1, parity=0.
wire rx_parity = ~^frame[8:1];
wire tx_parity = ~^tx_data;

// Assign busy signal
assign busy = (state == IDLE) ? 1'b0 : 1'b1;

// TRISTATE BUFFERS
assign ps2_clk  = ps2_clk_h  ? 1'bZ : 1'b0;
assign ps2_data = ps2_data_h ? 1'bZ : 1'b0;

// DEBOUNCE AND SYNCHRONIZE PS2_CLK AND PS2_DATA
always @(posedge clk or posedge rst) begin
   if (rst) begin
      ps2_clk_clean <= 1'b1;
      ps2_data_clean <= 1'b1;
      clk_count <= 4'd0;
      data_count <= 4'd0;
      clk_inter <= 1'b1;
      data_inter <= 1'b1;
   end else begin
      // Debounce ps2_clk
      if (ps2_clk != clk_inter) begin
         clk_inter <= ps2_clk;
         clk_count <= 4'd0;
      end else if (clk_count == DEBOUNCE_DELAY) begin
         ps2_clk_clean <= clk_inter;
      end else begin
         clk_count <= clk_count + 1'b1;
      end

      // Debounce ps2_data
      if (ps2_data != data_inter) begin
         data_inter <= ps2_data;
```

```
            data_count <= 4'd0;
        end else if (data_count == DEBOUNCE_DELAY) begin
            ps2_data_clean <= data_inter;
        end else begin
            data_count <= data_count + 1'b1;
        end
    end
end

// Synchronize signals
always @(posedge clk or posedge rst) begin
    if (rst) begin
        ps2_clk_s <= 1'b1;
        ps2_data_s <= 1'b1;
    end else begin
        ps2_clk_s <= ps2_clk_clean;
        ps2_data_s <= ps2_data_clean;
    end
end

// MAIN FSM
always @(posedge clk or posedge rst) begin
    if (rst) begin
        state <= IDLE;
        ps2_clk_h <= 1'b1;
        ps2_data_h <= 1'b1;
        load_tx_data <= 1'b0;
        load_rx_data <= 1'b0;
        read_data <= 1'b0;
        err <= 1'b0;
    end else begin
        // Default assignments every clock
        ps2_clk_h <= 1'b1;
        ps2_data_h <= 1'b1;
        load_tx_data <= 1'b0;
        load_rx_data <= 1'b0;
        read_data <= 1'b0;
        err <= 1'b0;

        case (state)
            IDLE: begin
                if (ps2_clk_s == 1'b0) begin
                    state <= RX_DOWN_EDGE;
                end else if (write_data == 1'b1) begin
                    state <= TX_FORCE_CLK_L;
                end else begin
                    state <= IDLE;
                end
```

```
end

RX_CLK_H: begin
   if (bit_count == NUMBITS) begin
      if (rx_parity != frame[PARITY_BIT]) begin
         state <= RX_ERROR_PARITY;
      end else begin
         load_rx_data <= 1'b1;
         state <= RX_DATA_READY;
      end
   end else if (ps2_clk_s == 1'b0) begin
      state <= RX_DOWN_EDGE;
   end else begin
      state <= RX_CLK_H;
   end
end

RX_DOWN_EDGE: begin
   state <= RX_CLK_L;
end

RX_CLK_L: begin
   if (ps2_clk_s == 1'b1) begin
      state <= RX_CLK_H;
   end else begin
      state <= RX_CLK_L;
   end
end

RX_ERROR_PARITY: begin
   err <= 1'b1;
   state <= IDLE;
end

RX_DATA_READY: begin
   read_data <= 1'b1;
   state <= IDLE;
end

TX_FORCE_CLK_L: begin
   load_tx_data <= 1'b1;
   ps2_clk_h <= 1'b0;
   if (delay_100us_done == 1'b1) begin
      state <= TX_BRING_DATA_DOWN;
   end else begin
      state <= TX_FORCE_CLK_L;
   end
end
```

```
TX_BRING_DATA_DOWN: begin
   ps2_clk_h <= 1'b0;
   ps2_data_h <= 1'b0; // start bit
   if (delay_20us_done == 1'b1) begin
      state <= TX_RELEASE_CLK;
   end else begin
      state <= TX_BRING_DATA_DOWN;
   end
end

TX_RELEASE_CLK: begin
   ps2_clk_h <= 1'b1;
   ps2_data_h <= 1'b0;
   state <= TX_FIRST_WAIT_DOWN_EDGE;
end

TX_FIRST_WAIT_DOWN_EDGE: begin
   ps2_data_h <= 1'b0;
   if (delay_63clk_done == 1'b1) begin
      if (ps2_clk_s == 1'b0) begin
         state <= TX_CLK_L;
      end else begin
         state <= TX_FIRST_WAIT_DOWN_EDGE;
      end
   end else begin
      state <= TX_FIRST_WAIT_DOWN_EDGE;
   end
end

TX_CLK_L: begin
   ps2_data_h <= frame[0];
   state <= TX_WAIT_UP_EDGE;
end

TX_WAIT_UP_EDGE: begin
   ps2_data_h <= frame[0];
   if (bit_count == NUMBITS-1) begin
      ps2_data_h <= 1'b1;
      state <= TX_WAIT_UP_EDGE_BEFORE_ACK;
   end else if (ps2_clk_s == 1'b1) begin
      state <= TX_CLK_H;
   end else begin
      state <= TX_WAIT_UP_EDGE;
   end
end

TX_CLK_H: begin
```

```
    ps2_data_h <= frame[0];
    if (ps2_clk_s == 1'b0) begin
        state <= TX_CLK_L;
    end else begin
        state <= TX_CLK_H;
    end
end

TX_WAIT_UP_EDGE_BEFORE_ACK: begin
    ps2_data_h <= 1'b1;
    if (ps2_clk_s == 1'b1) begin
        state <= TX_WAIT_ACK;
    end else begin
        state <= TX_WAIT_UP_EDGE_BEFORE_ACK;
    end
end

TX_WAIT_ACK: begin
    if (ps2_clk_s == 1'b0) begin
        if (ps2_data_s == 1'b0) begin
            state <= TX_RECEIVED_ACK;
        end else begin
            state <= TX_ERROR_NO_ACK;
        end
    end else begin
        state <= TX_WAIT_ACK;
    end
end

TX_RECEIVED_ACK: begin
    if (ps2_clk_s == 1'b1 && ps2_data_s == 1'b1) begin
        state <= IDLE;
    end else begin
        state <= TX_RECEIVED_ACK;
    end
end

TX_ERROR_NO_ACK: begin
    if (ps2_clk_s == 1'b1 && ps2_data_s == 1'b1) begin
        err <= 1'b1;
        state <= IDLE;
    end else begin
        state <= TX_ERROR_NO_ACK;
    end
end

default: begin
    err <= 1'b1;
```

```
            state <= IDLE;
        end
      endcase
   end
end

// COUNTERS FOR DELAYS
always @(*) begin
   delay_100us_counter_enable = (state == TX_FORCE_CLK_L);
   delay_20us_counter_enable  = (state == TX_BRING_DATA_DOWN);
   delay_63clk_counter_enable = (state == TX_FIRST_WAIT_DOWN_EDGE);
end

// 100us counter
always @(posedge clk or posedge rst) begin
   if (rst) begin
      delay_100us_count <= 14'd0;
      delay_100us_done <= 1'b0;
   end else if (delay_100us_counter_enable) begin
      if (delay_100us_count == DELAY_100US) begin
         delay_100us_done <= 1'b1;
      end else begin
         delay_100us_count <= delay_100us_count + 1'b1;
         delay_100us_done <= 1'b0;
      end
   end else begin
      delay_100us_count <= 14'd0;
      delay_100us_done <= 1'b0;
   end
end

// 20us counter
always @(posedge clk or posedge rst) begin
   if (rst) begin
      delay_20us_count <= 11'd0;
      delay_20us_done <= 1'b0;
   end else if (delay_20us_counter_enable) begin
      if (delay_20us_count == DELAY_20US) begin
         delay_20us_done <= 1'b1;
      end else begin
         delay_20us_count <= delay_20us_count + 1'b1;
         delay_20us_done <= 1'b0;
      end
   end else begin
      delay_20us_count <= 11'd0;
      delay_20us_done <= 1'b0;
   end
end
```

```
// 63clk counter
always @(posedge clk or posedge rst) begin
   if (rst) begin
      delay_63clk_count <= 7'd0;
      delay_63clk_done <= 1'b0;
   end else if (delay_63clk_counter_enable) begin
      if (delay_63clk_count == DELAY_63CLK) begin
         delay_63clk_done <= 1'b1;
      end else begin
         delay_63clk_count <= delay_63clk_count + 1'b1;
         delay_63clk_done <= 1'b0;
      end
   end else begin
      delay_63clk_count <= 7'd0;
      delay_63clk_done <= 1'b0;
   end
end

// BIT COUNTER AND FRAME SHIFT
always @(*) begin
   reset_bit_count = (state == IDLE);
   shift_frame = (state == RX_DOWN_EDGE || state == TX_CLK_L);
end

always @(posedge clk or posedge rst) begin
   if (rst) begin
      bit_count <= 4'd0;
   end else if (reset_bit_count) begin
      bit_count <= 4'd0;
   end else if (shift_frame) begin
      bit_count <= bit_count + 1'b1;
   end
end

// load_tx_data and shift
always @(posedge clk or posedge rst) begin
   if (rst) begin
      frame <= 11'd0;
   end else if (load_tx_data) begin
      // frame[8:1] = tx_data, frame[0] = start bit=0, frame[10]=stop=1, frame[9]=parity
      frame[8:1] <= tx_data;
      frame[0]    <= 1'b0;       // start bit
      frame[10]   <= 1'b1;       // stop bit
      frame[9]    <= tx_parity;  // parity bit
   end else if (shift_frame) begin
      // shift right by 1 bit
      frame[9:0]   <= frame[10:1];
```

```
      frame[10]    <= ps2_data_s;
   end
end

// load_rx_data
always @(posedge clk or posedge rst) begin
   if (rst) begin
      rx_data <= 8'd0;
   end else if (load_rx_data) begin
      rx_data <= frame[8:1];
   end
end


endmodule
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
//Bram module (From IP catalog)
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
LIBRARY blk_mem_gen_v8_4_8;
USE blk_mem_gen_v8_4_8.blk_mem_gen_v8_4_8;
ENTITY blk_mem_gen_0 IS
        PORT (
                clka : IN STD_LOGIC;
                ena : IN STD_LOGIC;
                wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
                addra : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
                dina : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                clkb : IN STD_LOGIC;
                rstb : IN STD_LOGIC;
                enb : IN STD_LOGIC;
                addrb : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
                doutb : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
                rsta_busy : OUT STD_LOGIC;
                rstb_busy : OUT STD_LOGIC
```

```
    );
END blk_mem_gen_0;

ARCHITECTURE blk_mem_gen_0_arch OF blk_mem_gen_0 IS

ATTRIBUTE DowngradeIPIdentifiedWarnings : STRING;

ATTRIBUTE DowngradeIPIdentifiedWarnings OF blk_mem_gen_0_arch: ARCHITECTURE
IS "yes";

COMPONENT blk_mem_gen_v8_4_8 IS
    GENERIC (
        C_FAMILY: STRING;
        C_XDEVICEFAMILY: STRING;
        C_ELABORATION_DIR: STRING;
        C_INTERFACE_TYPE: INTEGER;
        C_AXI_TYPE: INTEGER;
        C_AXI_SLAVE_TYPE: INTEGER;
        C_USE_BRAM_BLOCK: INTEGER;
        C_ENABLE_32BIT_ADDRESS: INTEGER;
        C_CTRL_ECC_ALGO: STRING;
        C_HAS_AXI_ID: INTEGER;
        C_AXI_ID_WIDTH: INTEGER;
        C_MEM_TYPE: INTEGER;
        C_BYTE_SIZE: INTEGER;
        C_ALGORITHM: INTEGER;
        C_PRIM_TYPE : INTEGER; C_LOAD_INIT_FILE : INTEGER;
    C_INIT_FILE_NAME : STRING; C_INIT_FILE : STRING;
    C_USE_DEFAULT_DATA : INTEGER; C_DEFAULT_DATA : STRING;
    C_HAS_RSTA : INTEGER; C_RST_PRIORITY_A : STRING; C_RSTRAM_A :
    INTEGER; C_INITA_VAL : STRING; C_HAS_ENA : INTEGER; C_HAS_REGCEA :
    INTEGER; C_USE_BYTE_WEA : INTEGER; C_WEA_WIDTH : INTEGER;
    C_WRITE_MODE_A : STRING; C_WRITE_WIDTH_A : INTEGER;
    C_READ_WIDTH_A : INTEGER; C_WRITE_DEPTH_A : INTEGER;
    C_READ_DEPTH_A : INTEGER; C_ADDRA_WIDTH : INTEGER; C_HAS_RSTB :
    INTEGER; C_RST_PRIORITY_B : STRING; C_RSTRAM_B : INTEGER;
    C_INITB_VAL : STRING; C_HAS_ENB : INTEGER; C_HAS_REGCEB : INTEGER;
    C_USE_BYTE_WEB : INTEGER; C_WEB_WIDTH : INTEGER;
    C_WRITE_MODE_B : STRING; C_WRITE_WIDTH_B : INTEGER;
    C_READ_WIDTH_B : INTEGER; C_WRITE_DEPTH_B : INTEGER;
```

C_READ_DEPTH_B : INTEGER; C_ADDRB_WIDTH : INTEGER;
C_HAS_MEM_OUTPUT_REGS_A : INTEGER; C_HAS_MEM_OUTPUT_REGS_B :
INTEGER; C_HAS_MUX_OUTPUT_REGS_A : INTEGER;
C_HAS_MUX_OUTPUT_REGS_B : INTEGER; C_MUX_PIPELINE_STAGES :
INTEGER; C_HAS_SOFTECC_INPUT_REGS_A : INTEGER;
C_HAS_SOFTECC_OUTPUT_REGS_B : INTEGER; C_USE_SOFTECC : INTEGER;
C_USE_ECC : INTEGER; C_EN_ECC_PIPE : INTEGER; C_READ_LATENCY_A :
INTEGER; C_READ_LATENCY_B : INTEGER; C_HAS_INJECTERR : INTEGER;
C_SIM_COLLISION_CHECK : STRING; C_COMMON_CLK : INTEGER;
C_DISABLE_WARN_BHV_COLL : INTEGER; C_EN_SLEEP_PIN : INTEGER;
C_USE_URAM : INTEGER; C_EN_RDADDRA_CHG : INTEGER;
C_EN_RDADDRB_CHG : INTEGER; C_EN_DEEPSLEEP_PIN : INTEGER;
C_EN_SHUTDOWN_PIN : INTEGER; C_EN_SAFETY_CKT : INTEGER;
C_DISABLE_WARN_BHV_RANGE : INTEGER; C_COUNT_36K_BRAM :
STRING; C_COUNT_18K_BRAM : STRING; C_EST_POWER_SUMMARY :
STRING); PORT (clka : IN STD_LOGIC; rsta : IN STD_LOGIC; ena : IN
STD_LOGIC; regcea : IN STD_LOGIC; wea : IN STD_LOGIC_VECTOR(0 DOWNTO
0); addra : IN STD_LOGIC_VECTOR(18 DOWNTO 0); dina : IN
STD_LOGIC_VECTOR(3 DOWNTO 0); douta : OUT STD_LOGIC_VECTOR(3
DOWNTO 0); clkb : IN STD_LOGIC; rstb : IN STD_LOGIC; enb : IN STD_LOGIC;
regceb : IN STD_LOGIC; web : IN STD_LOGIC_VECTOR(0 DOWNTO 0); addrb : IN
STD_LOGIC_VECTOR(18 DOWNTO 0); dinb : IN STD_LOGIC_VECTOR(3
DOWNTO 0); doutb : OUT STD_LOGIC_VECTOR(3 DOWNTO 0); injectsbiterr : IN
STD_LOGIC; injectdbiterr : IN STD_LOGIC; eccpipece : IN STD_LOGIC; sbiterr :
OUT STD_LOGIC; dbiterr : OUT STD_LOGIC; rdaddrecc : OUT
STD_LOGIC_VECTOR(18 DOWNTO 0); sleep : IN STD_LOGIC; deepsleep : IN
STD_LOGIC; shutdown : IN STD_LOGIC; rsta_busy : OUT STD_LOGIC; rstb_busy :
OUT STD_LOGIC; s_aclk : IN STD_LOGIC; s_aresetn : IN STD_LOGIC; s_axi_awid :
IN STD_LOGIC_VECTOR(3 DOWNTO 0); s_axi_awaddr : IN
STD_LOGIC_VECTOR(31 DOWNTO 0); s_axi_awlen : IN STD_LOGIC_VECTOR(7
DOWNTO 0); s_axi_awsize : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
s_axi_awburst : IN STD_LOGIC_VECTOR(1 DOWNTO 0); s_axi_awvalid : IN
STD_LOGIC; s_axi_awready : OUT STD_LOGIC; s_axi_wdata : IN
STD_LOGIC_VECTOR(3 DOWNTO 0); s_axi_wstrb : IN STD_LOGIC_VECTOR(0
DOWNTO 0); s_axi_wlast : IN STD_LOGIC; s_axi_wvalid : IN STD_LOGIC;
s_axi_wready : OUT STD_LOGIC; s_axi_bid : OUT STD_LOGIC_VECTOR(3
DOWNTO 0); s_axi_bresp : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
s_axi_bvalid : OUT STD_LOGIC; s_axi_bready : IN STD_LOGIC; s_axi_arid : IN
STD_LOGIC_VECTOR(3 DOWNTO 0); s_axi_araddr : IN STD_LOGIC_VECTOR(31
DOWNTO 0); s_axi_arlen : IN STD_LOGIC_VECTOR(7 DOWNTO 0); s_axi_arsize :
IN STD_LOGIC_VECTOR(2 DOWNTO 0); s_axi_arburst : IN
STD_LOGIC_VECTOR(1 DOWNTO 0); s_axi_arvalid : IN STD_LOGIC;
s_axi_arready : OUT STD_LOGIC; s_axi_rid : OUT STD_LOGIC_VECTOR(3
DOWNTO 0); s_axi_rdata : OUT STD_LOGIC_VECTOR(3 DOWNTO 0); s_axi_rresp :
OUT STD_LOGIC_VECTOR(1 DOWNTO 0); s_axi_rlast : OUT STD_LOGIC;
s_axi_rvalid : OUT STD_LOGIC; s_axi_rready : IN STD_LOGIC; s_axi_injectsbiterr :
IN STD_LOGIC; s_axi_injectdbiterr : IN STD_LOGIC; s_axi_sbiterr : OUT
STD_LOGIC; s_axi_dbiterr : OUT STD_LOGIC; s_axi_rdaddrecc : OUT

STD_LOGIC_VECTOR(18 DOWNTO 0)); END COMPONENT blk_mem_gen_v8_4_8; ATTRIBUTE X_CORE_INFO : STRING; ATTRIBUTE X_CORE_INFO OF blk_mem_gen_0_arch: ARCHITECTURE IS "blk_mem_gen_v8_4_8,Vivado 2024.1.2"; ATTRIBUTE CHECK_LICENSE_TYPE : STRING; ATTRIBUTE CHECK_LICENSE_TYPE OF blk_mem_gen_0_arch : ARCHITECTURE IS "blk_mem_gen_0,blk_mem_gen_v8_4_8,{}"; ATTRIBUTE CORE_GENERATION_INFO : STRING; ATTRIBUTE CORE_GENERATION_INFO OF blk_mem_gen_0_arch: ARCHITECTURE IS "blk_mem_gen_0,blk_mem_gen_v8_4_8,{x_ipProduct=Vivado 2024.1.2,x_ipVendor=xilinx.com,x_ipLibrary=ip,x_ipName=blk_mem_gen,x_ipVersion=8.4,x_ipCoreRevision=8,x_ipLanguage=VERILOG,x_ipSimLanguage=MIXED,C_FAMILY=artix7,C_XDEVICEFAMILY=artix7,C_ELABORATION_DIR=./,C_INTERFACE_TYPE=0,C_AXI_TYPE=1,C_AXI_SLAVE_TYPE=0,C_USE_BRAM_BLOCK=0,C_ENABLE_32BIT_ADDRESS=0,C_CTRL_ECC_ALGO=NONE,C_HAS_AXI_ID=0,C_AXI_ID_WIDTH=4,C_MEM_TYPE=1,C_BYTE_SIZE=9,C_ALGORITHM=1,C_PRIM_TYPE=1,C_LOAD_INIT_FILE=0,C_INIT_FILE_NAME=no_" &
"coe_file_loaded,C_INIT_FILE=blk_mem_gen_0.mem,C_USE_DEFAULT_DATA=0,C_DEFAULT_DATA=0,C_HAS_RSTA=0,C_RST_PRIORITY_A=CE,C_RSTRAM_A=0,C_INITA_VAL=0,C_HAS_ENA=1,C_HAS_REGCEA=0,C_USE_BYTE_WEA=0,C_WEA_WIDTH=1,C_WRITE_MODE_A=NO_CHANGE,C_WRITE_WIDTH_A=4,C_READ_WIDTH_A=4,C_WRITE_DEPTH_A=307200,C_READ_DEPTH_A=307200,C_ADDRA_WIDTH=19,C_HAS_RSTB=1,C_RST_PRIORITY_B=CE,C_RSTRAM_B=0,C_INITB_VAL=0,C_HAS_ENB=1,C_HAS_REGCEB=0,C_USE_BYTE_WEB=0,C_WEB_WIDTH=1,C_WRITE_MODE_B=WRITE_FIRST,C_WRITE_WIDTH_B=4,C_READ_WIDTH_" &
"B=4,C_WRITE_DEPTH_B=307200,C_READ_DEPTH_B=307200,C_ADDRB_WIDTH=19,C_HAS_MEM_OUTPUT_REGS_A=0,C_HAS_MEM_OUTPUT_REGS_B=1,C_HAS_MUX_OUTPUT_REGS_A=0,C_HAS_MUX_OUTPUT_REGS_B=0,C_MUX_PIPELINE_STAGES=0,C_HAS_SOFTECC_INPUT_REGS_A=0,C_HAS_SOFTECC_OUTPUT_REGS_B=0,C_USE_SOFTECC=0,C_USE_ECC=0,C_EN_ECC_PIPE=0,C_READ_LATENCY_A=1,C_READ_LATENCY_B=1,C_HAS_INJECTERR=0,C_SIM_COLLISION_CHECK=ALL,C_COMMON_CLK=0,C_DISABLE_WARN_BHV_COLL=0,C_EN_SLEEP_PIN=0,C_USE_URAM=0,C_EN_RDADDRA_CHG=0,C_EN_RDADDRB_CHG=0,C_EN_DEEPSLE" &
"EP_PIN=0,C_EN_SHUTDOWN_PIN=0,C_EN_SAFETY_CKT=1,C_DISABLE_WARN_BHV_RANGE=0,C_COUNT_36K_BRAM=36,C_COUNT_18K_BRAM=3,C_EST_POWER_SUMMARY=Estimated Power for IP _ 16.198881 mW}"; ATTRIBUTE X_INTERFACE_INFO : STRING; ATTRIBUTE X_INTERFACE_PARAMETER : STRING; ATTRIBUTE X_INTERFACE_INFO OF addra: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTA ADDR"; ATTRIBUTE X_INTERFACE_INFO OF addrb: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTB ADDR"; ATTRIBUTE X_INTERFACE_PARAMETER OF clka: SIGNAL IS "XIL_INTERFACENAME BRAM_PORTA, MEM_ADDRESS_MODE BYTE_ADDRESS, MEM_SIZE 8192, MEM_WIDTH 32, MEM_ECC NONE, MASTER_TYPE OTHER, READ_LATENCY 1"; ATTRIBUTE X_INTERFACE_INFO OF clka: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTA CLK"; ATTRIBUTE X_INTERFACE_PARAMETER OF clkb: SIGNAL IS "XIL_INTERFACENAME BRAM_PORTB, MEM_ADDRESS_MODE

BYTE_ADDRESS, MEM_SIZE 8192, MEM_WIDTH 32, MEM_ECC NONE, MASTER_TYPE OTHER, READ_LATENCY 1"; ATTRIBUTE X_INTERFACE_INFO OF clkb: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTB CLK"; ATTRIBUTE X_INTERFACE_INFO OF dina: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTA DIN"; ATTRIBUTE X_INTERFACE_INFO OF doutb: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTB DOUT"; ATTRIBUTE X_INTERFACE_INFO OF ena: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTA EN"; ATTRIBUTE X_INTERFACE_INFO OF enb: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTB EN"; ATTRIBUTE X_INTERFACE_INFO OF rstb: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTB RST"; ATTRIBUTE X_INTERFACE_INFO OF wea: SIGNAL IS "xilinx.com:interface:bram:1.0 BRAM_PORTA WE"; BEGIN U0 : blk_mem_gen_v8_4_8 GENERIC MAP (C_FAMILY => "artix7", C_XDEVICEFAMILY => "artix7", C_ELABORATION_DIR => "./", C_INTERFACE_TYPE => 0, C_AXI_TYPE => 1, C_AXI_SLAVE_TYPE => 0, C_USE_BRAM_BLOCK => 0, C_ENABLE_32BIT_ADDRESS => 0, C_CTRL_ECC_ALGO => "NONE", C_HAS_AXI_ID => 0, C_AXI_ID_WIDTH => 4, C_MEM_TYPE => 1, C_BYTE_SIZE => 9, C_ALGORITHM => 1, C_PRIM_TYPE => 1, C_LOAD_INIT_FILE => 0, C_INIT_FILE_NAME => "no_coe_file_loaded", C_INIT_FILE => "blk_mem_gen_0.mem", C_USE_DEFAULT_DATA => 0, C_DEFAULT_DATA => "0", C_HAS_RSTA => 0, C_RST_PRIORITY_A => "CE", C_RSTRAM_A => 0, C_INITA_VAL => "0", C_HAS_ENA => 1, C_HAS_REGCEA => 0, C_USE_BYTE_WEA => 0, C_WEA_WIDTH => 1, C_WRITE_MODE_A => "NO_CHANGE", C_WRITE_WIDTH_A => 4, C_READ_WIDTH_A => 4, C_WRITE_DEPTH_A => 307200, C_READ_DEPTH_A => 307200, C_ADDRA_WIDTH => 19, C_HAS_RSTB => 1, C_RST_PRIORITY_B => "CE", C_RSTRAM_B => 0, C_INITB_VAL => "0", C_HAS_ENB => 1, C_HAS_REGCEB => 0, C_USE_BYTE_WEB => 0, C_WEB_WIDTH => 1, C_WRITE_MODE_B => "WRITE_FIRST", C_WRITE_WIDTH_B => 4, C_READ_WIDTH_B => 4, C_WRITE_DEPTH_B => 307200, C_READ_DEPTH_B => 307200, C_ADDRB_WIDTH => 19, C_HAS_MEM_OUTPUT_REGS_A => 0, C_HAS_MEM_OUTPUT_REGS_B => 1, C_HAS_MUX_OUTPUT_REGS_A => 0, C_HAS_MUX_OUTPUT_REGS_B => 0, C_MUX_PIPELINE_STAGES => 0, C_HAS_SOFTECC_INPUT_REGS_A => 0, C_HAS_SOFTECC_OUTPUT_REGS_B => 0, C_USE_SOFTECC => 0, C_USE_ECC => 0, C_EN_ECC_PIPE => 0, C_READ_LATENCY_A => 1, C_READ_LATENCY_B => 1, C_HAS_INJECTERR => 0, C_SIM_COLLISION_CHECK => "ALL", C_COMMON_CLK => 0, C_DISABLE_WARN_BHV_COLL => 0, C_EN_SLEEP_PIN => 0, C_USE_URAM => 0, C_EN_RDADDRA_CHG => 0, C_EN_RDADDRB_CHG => 0, C_EN_DEEPSLEEP_PIN => 0, C_EN_SHUTDOWN_PIN => 0, C_EN_SAFETY_CKT => 1, C_DISABLE_WARN_BHV_RANGE => 0, C_COUNT_36K_BRAM => "36", C_COUNT_18K_BRAM => "3", C_EST_POWER_SUMMARY => "Estimated Power for IP : 16.198881 mW") PORT MAP (clka => clka, rsta => '0', ena => ena, regcea => '1', wea => wea, addra => addra, dina => dina, clkb => clkb, rstb => rstb, enb => enb, regceb => '1', web => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 1)), addrb => addrb, dinb => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 4)), doutb => doutb, injectsbiterr => '0', injectdbiterr => '0', eccpipece => '0', sleep => '0', deepsleep => '0', shutdown =>

'0', rsta_busy => rsta_busy, rstb_busy => rstb_busy, s_aclk => '0', s_aresetn => '0', s_axi_awid => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 4)), s_axi_awaddr => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 32)), s_axi_awlen => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)), s_axi_awsize => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 3)), s_axi_awburst => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 2)), s_axi_awvalid => '0', s_axi_wdata => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 4)), s_axi_wstrb => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 1)), s_axi_wlast => '0', s_axi_wvalid => '0', s_axi_bready => '0', s_axi_arid => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 4)), s_axi_araddr => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 32)), s_axi_arlen => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)), s_axi_arsize => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 3)), s_axi_arburst => STD_LOGIC_VECTOR(TO_UNSIGNED(0, 2)), s_axi_arvalid => '0', s_axi_rready => '0', s_axi_injectsbiterr => '0', s_axi_injectdbiterr => '0'); END blk_mem_gen_0_arch;