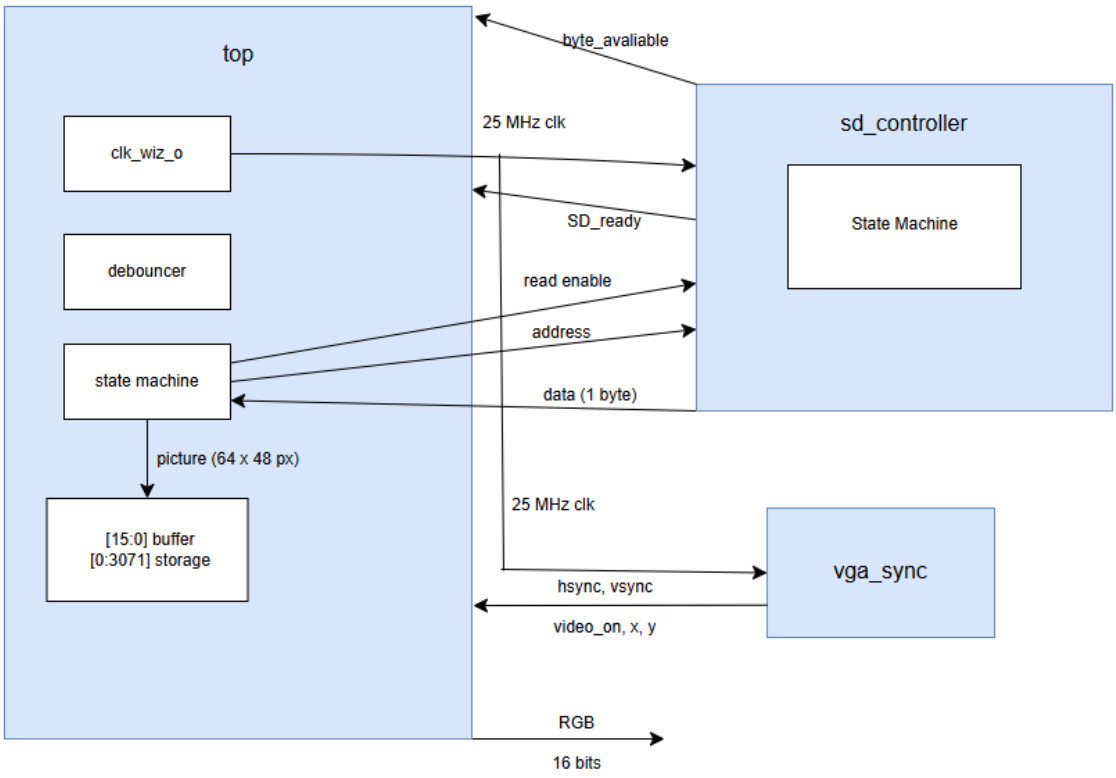
Group Member

Group name : ตาลอยเพราะซอยเปลี่ยว

Name	Student Number
Kris Bunuabenjamas	6631301821
Tanphan Krupistrai	6631323621
Thanaphat Intarapuk	6631324221
Tatthon Limmonthol	6631325921

Overall Design Block Diagram



Original picture : https://drive.google.com/file/d/1TD7IQv_B_17KxaUctUiCpuT0lYRwn6w-/view?usp=sharing

Design Decision.

- Module Architecture
    - State Machine-Based Control
      Advantage:

        - **Ease of debugging**: You can trace system behavior step-by-step using `main_state`.

        - **Deterministic control**: FSMs ensure predictable transitions, ideal for hardware.

    - Divided the system into clearly separated modules

      Advantage:

        - **Modularity** improves code readability, reusability, and testability.

        - Each module can be developed, simulated, or debugged independently.

- Communication Protocols
  We using SPI protocol for communication here this is why
    - **Wide Compatibility**
      SPI mode is supported by almost all SD cards. It is part of the SD card specification, ensuring compatibility even with older or low-capacity cards.

    - **Simplicity**
      SPI is a much simpler protocol than the native SD protocol. It uses a basic 4-wire interface (MISO, MOSI, SCLK, CS), which is easy to

implement in hardware such as FPGAs or microcontrollers.

- **No Licensing Requirements**
  Unlike the SD native protocol, which requires licensing from the SD Association, SPI mode can be used freely without any licensing fees or legal constraints.

- **Low Resource Usage**
  Implementing SPI requires fewer logic resources and memory compared to a full SD host controller, making it ideal for resource-constrained systems like small FPGAs or microcontrollers.

- **Master-Controlled Communication**
  SPI is a master-driven protocol, allowing the host (FPGA/microcontroller) to fully control the timing and data flow, which is useful in embedded systems where precise control is needed.

- **Widely Supported in Development Tools**
  Most FPGA development boards, IP cores, and microcontroller libraries provide ready-to-use SPI modules, making development faster and easier.

- Clock and Data Transfer Rate

  We selected a **25 MHz clock** because it provides a balanced trade-off between compatibility, system performance, and reliability for both **SD card communication** and **VGA display timing**.

1. SD Card Compatibility

- The **SPI mode** of SD cards typically supports clock frequencies up to 25 MHz for full-speed operation.

- **25 MHz** is within the **safe and standard operating range** recommended by SD card specifications for SPI, ensuring reliable communication without

errors.

- Using a higher clock may cause instability or data corruption, especially with cheaper or older SD cards.

2. VGA Display Timing Requirements

- For **VGA 640x480 @ 60Hz**, a **pixel clock of 25.175 MHz** is standard.

- Using **25 MHz** allows us to approximate this closely, maintaining proper horizontal and vertical sync timing for most monitors.

- It simplifies synchronization with the VGA controller and pixel pipeline without needing fractional frequency generation.

- Resource Utilization
    - Using reg [15:0] buffer [0:3071];
    - **Memory in registers (Distributed RAM)**: Instead of using BRAMs, the design uses FPGA registers (LUTRAM) to implement a **small, tightly controlled image buffer** (only 6 KB).
    - **Trade-off**: This keeps BRAMs available for other tasks but slightly increases LUT usage. However, it's acceptable due to the limited size (only 3072 pixels = 64×48 image).

1. Using SPI Mode to Communicate with SD Card

Decision: Implement SD card interface using SPI (not native SD mode).

Why:

- SPI is simpler and license-free.
- Easier to implement in FPGA hardware.
- Compatible with almost all SD cards.

2. Use internal registers (not BRAM) to store pixel data.

Why:

- Small image size (64x48 = 3072 pixels) allows for register storage.
- Keeps BRAM free for other uses.
- Allows direct access and simple logic for read/write.

3. Downscale VGA resolution from 640×480 to 64×48.

Why:

- Reduces memory footprint for frame buffer.
- Simplifies pixel addressing.
- Enables real-time updates even with limited memory and bandwidth.

Decision: Calculate checksum only after all sectors have been read.

Why:

- Reduces interference with SD read or display logic.

- Ensures checksum is based only on valid, complete data.

Design Decision : Gray Mode Toggle

Why This Fits Our Design

- This feature demonstrates dynamic VGA behavior modification.
- Grayscale conversion is implemented by reusing bits from RGB to produce a unified value.

Implementation Detail

Module : sd_controller

Module code :

```
/* SD Card controller module. Allows reading from and writing to a microSD card
through SPI mode. Fixed version according to lecture specifications. */

`timescale 1ns / 1ps

module sd_controller(
   output reg cs,
   output mosi,
   input miso,
   output sclk,
   input rd,
   output reg [7:0] dout,
   output reg byte_available,
   output reg ready_for_next_byte,
   input reset,
   output ready,
   input [31:0] address,
   input clk
);
   parameter RST = 0;
   parameter INIT = 1;
   parameter CMD0 = 2;
   parameter CMD8 = 3;
   parameter CMD55 = 4;
   parameter CMD41 = 5;
   parameter POLL_CMD = 6;

   parameter IDLE = 7;
   parameter READ_BLOCK = 8;
   parameter READ_BLOCK_WAIT = 9;
   parameter READ_BLOCK_DATA = 10;
   parameter READ_BLOCK_CRC = 11;
   parameter SEND_CMD = 12;
   parameter RECEIVE_BYTE_WAIT = 13;
   parameter RECEIVE_BYTE = 14;
   parameter CMD58 = 20;  // Added CMD58 state for OCR register check
```

```
parameter WRITE_DATA_SIZE = 515;

reg [4:0] state = RST;
reg [4:0] return_state;
reg sclk_sig = 0;
reg [55:0] cmd_out;
reg [7:0] recv_data;

reg [9:0] byte_counter;
reg [9:0] bit_counter;

//boot SDcard
reg [26:0] boot_counter = 27'd100_000_000;
always @(posedge clk) begin
    if(reset == 1) begin
        state <= RST;
        sclk_sig <= 0;
        boot_counter <= 27'd100_000_000;
    end
    else begin
        case(state)
            RST: begin
                if(boot_counter == 0) begin
                    sclk_sig <= 0;
                    cmd_out <= {56{1'b1}};
                    byte_counter <= 0;
                    byte_available <= 0;
                    ready_for_next_byte <= 0;
                    //need at least 74clk
                    bit_counter <= 160;
                    cs <= 1;
                    state <= INIT;

                end
                else begin
                    boot_counter <= boot_counter - 1;
                end
            end
            INIT: begin
                if(bit_counter == 0) begin
                    cs <= 0;
```

```
                state <= CMD0;
            end
            else begin
                bit_counter <= bit_counter - 1;
                sclk_sig <= ~sclk_sig;
            end
        end
        CMD0: begin
            cmd_out <= 56'hFF_40_00_00_00_00_95;
            bit_counter <= 55;
            return_state <= CMD8;
            state <= SEND_CMD;
        end
        CMD8: begin
            cmd_out <= 56'hFF_48_00_00_01_AA_87;
            bit_counter <= 55;
            return_state <= CMD58;
            state <= SEND_CMD;
        end
        CMD58: begin
            cmd_out <= 56'hFF_7A_00_00_00_00_75;
            bit_counter <= 55;
            return_state <= CMD55;
            state <= SEND_CMD;
        end
        CMD55: begin
            cmd_out <= 56'hFF_77_00_00_00_00_65;
            bit_counter <= 55;
            return_state <= CMD41;
            state <= SEND_CMD;
        end
        CMD41: begin
            cmd_out <= 56'hFF_69_40_00_00_00_77;
            bit_counter <= 55;
            return_state <= POLL_CMD;
            state <= SEND_CMD;
        end
        POLL_CMD: begin
            if(recv_data[0] == 0) begin
                state <= IDLE;
            end
```

```
                else begin
                    state <= CMD55;  // loop cmd55 -> cmd41
                end
            end
            IDLE: begin
                if(rd == 1) begin
                    state <= READ_BLOCK;
                end
                else begin
                    state <= IDLE;
                end
            end
            READ_BLOCK: begin
                cmd_out <= {16'hFF_51, address, 8'hFF};
                bit_counter <= 55;
                return_state <= READ_BLOCK_WAIT;
                state <= SEND_CMD;
            end
            READ_BLOCK_WAIT: begin
                if(sclk_sig == 1 && miso == 0) begin
                    byte_counter <= 511;
                    bit_counter <= 7;
                    return_state <= READ_BLOCK_DATA;
                    state <= RECEIVE_BYTE;
                end
                sclk_sig <= ~sclk_sig;
            end
            READ_BLOCK_DATA: begin
                dout <= recv_data;
                byte_available <= 1;
                if (byte_counter == 0) begin
                    bit_counter <= 7;
                    return_state <= READ_BLOCK_CRC;
                    state <= RECEIVE_BYTE;
                end
                else begin
                    byte_counter <= byte_counter - 1;
                    return_state <= READ_BLOCK_DATA;
                    bit_counter <= 7;
                    state <= RECEIVE_BYTE;
                end
```

```
            end
            READ_BLOCK_CRC: begin
                bit_counter <= 7;
                return_state <= IDLE;
                state <= RECEIVE_BYTE;
            end
            SEND_CMD: begin
                if (sclk_sig == 1) begin
                    if (bit_counter == 0) begin
                        state <= RECEIVE_BYTE_WAIT;
                    end
                    else begin
                        bit_counter <= bit_counter - 1;
                        cmd_out <= {cmd_out[54:0], 1'b1};
                    end
                end
                sclk_sig <= ~sclk_sig;
            end
            RECEIVE_BYTE_WAIT: begin
                if (sclk_sig == 1) begin
                    if (miso == 0) begin
                        recv_data <= 0;
                        bit_counter <= 6;
                        state <= RECEIVE_BYTE;
                    end
                end
                sclk_sig <= ~sclk_sig;
            end
            RECEIVE_BYTE: begin
                byte_available <= 0;
                if (sclk_sig == 1) begin
                    recv_data <= {recv_data[6:0], miso};
                    if (bit_counter == 0) begin
                        state <= return_state;
                    end
                    else begin
                        bit_counter <= bit_counter - 1;
                    end
                end
                sclk_sig <= ~sclk_sig;
            end
```

```
            endcase
        end
    end

    assign sclk = sclk_sig;
    assign mosi = cmd_out[55];
    assign ready = (state == IDLE);
endmodule
```

Module Description : The sd_controller module implements a simplified SPI-mode interface to communicate with an SD card for block-based data reading. It handles SD card initialization, command transmission, block read requests, and serial data transfer at the bit level. This controller is tailored for reading 512-byte sectors and can be integrated into systems such as image/video frame loaders for embedded display applications.

The controller first sends at least 74 SPI clocks with CS high to wake the SD card, then follows the SD card initialization sequence (CMD0 → CMD8 → CMD58 → CMD55/CMD41 loop).

Module : vga_sync

Module code :

```
module vga_sync (
    input wire clk, reset,
    output wire hsync, vsync, video_on,
    output wire p_tick,
    output wire [9:0] x, y
);
    localparam H_DISPLAY = 640;
    localparam H_L_BORDER = 48;
    localparam H_R_BORDER = 16;
    localparam H_RETRACE = 96;
    localparam H_MAX = H_DISPLAY + H_L_BORDER + H_R_BORDER + H_RETRACE - 1;
    localparam START_H_RETRACE = H_DISPLAY + H_R_BORDER;
    localparam END_H_RETRACE = H_DISPLAY + H_R_BORDER + H_RETRACE - 1;

    localparam V_DISPLAY = 480;
    localparam V_T_BORDER = 10;
    localparam V_B_BORDER = 33;
    localparam V_RETRACE = 2;
    localparam V_MAX = V_DISPLAY + V_T_BORDER + V_B_BORDER + V_RETRACE - 1;
```

```
  localparam START_V_RETRACE = V_DISPLAY + V_B_BORDER;
  localparam END_V_RETRACE = V_DISPLAY + V_B_BORDER + V_RETRACE - 1;

  assign p_tick = 1'b1;

  reg [9:0] h_count_reg, h_count_next, v_count_reg, v_count_next; // next location
  reg vsync_reg, hsync_reg;
  wire vsync_next, hsync_next;

  always @(posedge clk, posedge reset)
      if(reset) begin // reset
          v_count_reg <= 0;
          h_count_reg <= 0;
          vsync_reg <= 0;
          hsync_reg <= 0;
      end
      else begin // next
          v_count_reg <= v_count_next;
          h_count_reg <= h_count_next;
          vsync_reg <= vsync_next;
          hsync_reg <= hsync_next;
      end

  always @* begin
      h_count_next = h_count_reg == H_MAX ? 0 : h_count_reg + 1; // next line
      v_count_next = h_count_reg == H_MAX ?
                     (v_count_reg == V_MAX ? 0 : v_count_reg + 1) :
                     v_count_reg; // next image
  end

  assign hsync_next = h_count_reg >= START_H_RETRACE && h_count_reg <= END_H_RETRACE;
  assign vsync_next = v_count_reg >= START_V_RETRACE && v_count_reg <= END_V_RETRACE;

  assign video_on = (h_count_reg < H_DISPLAY) && (v_count_reg < V_DISPLAY);

  assign hsync = hsync_reg;
  assign vsync = vsync_reg;
  assign x = h_count_reg;
  assign y = v_count_reg;
endmodule
```

Module description : The vga_sync module generates timing signals for a standard 640x480 VGA display using a 25 MHz pixel clock. It provides horizontal and vertical synchronization pulses (hsync,vsync), pixel coordinates (x,y), and a video_on signal that indicates whether the current pixel is within the visible display region.

Module : top
Module code :

```
module top(
   input wire          clk100mhz,
   input wire          reset,
   input wire          btn,
   input wire          btnd,
   input               miso,
   output              mosi,
   output              sclk,
   output              cs,
   output reg          [15:0] led,

   // VGA outputs
   input wire [15:0]   sw,
   output wire         hsync,              // VGA horizontal sync
   output wire         vsync,              // VGA vertical sync
   output wire         [3:0] vga_r,        // VGA red channel
   output wire         [3:0] vga_g,        // VGA green channel
   output wire         [3:0] vga_b         // VGA blue channel
);
   reg        done;
   // Clock and reset signals
   wire       clk;            // 25MHz clock for SD card operation
   wire       locked;         // PLL locked signal
   wire       rst = ~locked | reset;  // Reset is active when PLL is not locked OR
reset input is high

   // Debounced button signal
   wire       btn_debounced;
   reg        btn_prev = 0;
   reg        btn_pressed = 0;

   reg btnd_prev = 0;
   reg btnd_pressed = 0;
```

```
  reg greyscale_mode = 0;

  // Buffer storage instead of block RAM
  reg [15:0] buffer [0:3071];
  reg [14:0] buffer_addr_write = 0;   // Address for writing to buffer
  wire [14:0] buffer_addr_read;       // Address for reading from buffer (for VGA)

  // Byte pairing for 16-bit buffer storage
  reg [7:0]  byte_buffer;     // Buffer for first byte in the pair
  reg        byte_ready = 0;  // Flag indicating byte buffer has data

  // Checksum calculation
  reg [31:0] checksum = 0;        // Holds the running sum (needs to be 32-bit to
handle overflow before modulo)
  reg [14:0] checksum_addr = 0;   // Address counter for reading buffer during
checksum calculation
  reg [1:0]  checksum_state = 0;  // Sub-state for checksum calculation

  // SD card controller signals
  reg        rd = 0;          // Read enable for SD controller
  wire [7:0] sd_dout;         // Data from SD controller
  wire       byte_available;  // New byte available from SD controller
  wire       ready;           // SD card is ready for operations

  // State machine variables
  reg [3:0]  main_state = INIT;
  parameter  INIT = 0,
             SD_WAIT_READY = 1,
             READ_SD = 2,
             WAIT_SECTOR = 3,
             DONE = 4,
             CALCULATE_CHECKSUM = 5,
             DISPLAY_CHECKSUM = 6;

  // Sector reading logic
  reg [31:0] current_sector = 32'd0;    // Current sector being read (0-71)
  reg [31:0] sector_base = 32'd0;       // Base sector for the current group
  reg [9:0]  bytes_read = 0;            // Counter for bytes read in current sector
  reg        reading = 0;               // Flag to indicate reading in progress
  reg [6:0]  sectors_read = 0;          // Counter for number of sectors read (0-71)

  // Flag to indicate button was pressed, used to communicate between always blocks
```

```
reg         change_sector_group = 0;

// VGA-related signals
wire        video_on;         // VGA display active area
wire        p_tick;           // 25MHz pixel clock tick
wire [9:0] pixel_x, pixel_y; // Current pixel coordinates

// Debug register for pixel data display - separate from main LED output
reg [15:0] debug_pixel;

// Clock wizard instance for 25MHz clock
clk_wiz_0 u_clk_wiz_0 (
    .reset       (reset),
    .clk_in1     (clk100mhz),       // input 100MHz
    .locked      (locked),
    .clk_out1    (clk)              // output 25MHz
);

// Button debouncer
debounce btn_debouncer (
    .clk         (clk),
    .reset       (rst),
    .btn_in      (btn),
    .btn_out     (btn_debounced)
);

wire btnd_debounced;
debounce btnd_debouncer (
    .clk         (clk),
    .reset       (rst),
    .btn_in      (btnd),
    .btn_out     (btnd_debounced)
);

vga_sync vga_sync_unit (
    .clk(clk),
    .reset(rst),
    .hsync(hsync),
    .vsync(vsync),
    .video_on(video_on),
    .p_tick(p_tick),
    .x(pixel_x),
```

```
        .y(pixel_y)
    );

    // Color and pixel address calculation
    wire [15:0] pixel_data;     // Current pixel color data
    reg [12:0] scaled_x, scaled_y;
    reg [12:0] buffer_pixel_addr;
    // reg [14:0] frame_base_addr;
    reg [15:0] tled;
    // Scale coordinates from 640x480 to 64x48
    always @* begin
        scaled_x = pixel_x / 10;  // 640/64 = 10
        scaled_y = pixel_y / 10;  // 480/48 = 10

        // Calculate base address for current frame (each frame is 64x48 = 3072 pixels)
        // frame_base_addr = current_frame * 3072;

        // Calculate pixel address within the buffer
        buffer_pixel_addr = (scaled_y * 64) + scaled_x;
    end


    // Assign buffer read address for VGA display
    assign buffer_addr_read = (scaled_x < 64 && scaled_y < 48) ?
                            buffer_pixel_addr :
                           0; // Default to first pixel if out of bounds

    assign pixel_data = (video_on && main_state == DONE) ? buffer[buffer_addr_read] :
16'h0000;

    wire [3:0] red   = pixel_data[15:12];
    wire [3:0] green = pixel_data[10:7];
    wire [3:0] blue  = pixel_data[4:1];

    // Compute grayscale value (weighted average approximation)
    wire [7:0] r8 = {pixel_data[15:11], 3'b000}; // 5-bit to 8-bit
    wire [7:0] g8 = {pixel_data[10:5],  2'b00};  // 6-bit to 8-bit
    wire [7:0] b8 = {pixel_data[4:0],   3'b000}; // 5-bit to 8-bit

    // weighted average (approximate): (r*30 + g*59 + b*11) / 100
    wire [15:0] gray_val = (r8 * 30 + g8 * 59 + b8 * 11) / 100;
```

```
// cut to 4-bit grayscale
wire [3:0] gray4 = gray_val[7:4]; // upper 4 bits of 8-bit value

assign vga_r = video_on ? (greyscale_mode ? gray4 : pixel_data[15:12]) : 4'h0;
assign vga_g = video_on ? (greyscale_mode ? gray4 : pixel_data[10:7])  : 4'h0;
assign vga_b = video_on ? (greyscale_mode ? gray4 : pixel_data[4:1])   : 4'h0;

// Button edge detection
always @(posedge clk) begin
    if (rst) begin
        btn_prev <= 0;
        btn_pressed <= 0;
        change_sector_group <= 0;
    end else begin
        btn_prev <= btn_debounced;
        btn_pressed <= ~btn_prev & btn_debounced; // Rising edge detection

        // Set the flag when button is pressed and we're in DONE state
        if (btn_pressed && main_state == DONE) begin
            change_sector_group <= 1;
        end else begin
            change_sector_group <= 0;
        end
    end
end

always @(posedge clk) begin
    if (rst) begin
        btnd_prev <= 0;
        btnd_pressed <= 0;
        greyscale_mode <= 0;
    end else begin
        btnd_prev <= btnd_debounced;
        btnd_pressed <= ~btnd_prev & btnd_debounced; // Rising edge detection

        if (btnd_pressed && main_state == DONE) begin
            greyscale_mode <= ~greyscale_mode;
        end

    end
end
```

```
    // Main state machine
    always @(posedge clk) begin
        if (rst) begin
            main_state <= INIT;
            rd <= 0;
            reading <= 0;
            buffer_addr_write <= 0;
            bytes_read <= 0;
            led <= 16'h0000;
            done <= 0;
            byte_ready <= 0;
            checksum <= 0;
            checksum_addr <= 0;
            checksum_state <= 0;
            // Initialize sector variables
            sector_base <= 32'd0;
            current_sector <= 32'd0;
            sectors_read <= 0;
        end else begin
            // Handle sector group change request from button press
            if (change_sector_group) begin
                // Change sector group when button is pressed and we're in DONE state
                // Cycle through the 6 sector groups
                if (sector_base == 0)          sector_base <= 32'd100;    // Move to
Sectors 100
                else if (sector_base == 100)   sector_base <= 32'd200;    // Move to
Sectors 200
                else if (sector_base == 200)   sector_base <= 32'd300;    // Move to
Sectors 300
                else if (sector_base == 300)   sector_base <= 32'd400;    // Move to
Sectors 400
                else if (sector_base == 400)   sector_base <= 32'd0;      // Back to
Sectors 0
                else                           sector_base <= 32'd0;      // Default
case

                // Reset current sector to base sector
                current_sector <= sector_base;
                sectors_read <= 0;
                buffer_addr_write <= 0;  // Reset buffer address for new sector group
                main_state <= INIT;
            end else begin
```

```
            // Normal state machine operation
            case (main_state)
                INIT: begin
                    main_state <= SD_WAIT_READY;
                    bytes_read <= 0;
                    rd <= 0;
                    reading <= 0;
                    byte_ready <= 0;

                    // Use current sector position
                    current_sector <= sector_base + sectors_read;

                    if (sectors_read >= 72) begin
                        done <= 1;  // Signal all sectors are read
                        main_state <= DONE;
                    end else begin
                        done <= 0;
                    end
                end

                SD_WAIT_READY: begin
                    // Wait for SD card to be ready
                    if (ready) begin
                        main_state <= READ_SD;
                    end
                end

                READ_SD: begin
                    // Start reading if not already reading
                    if (ready && !reading && !rd) begin
                        rd <= 1;
                        reading <= 1;
                    end else if (rd) begin
                        rd <= 0;  // Clear read signal after one clock cycle
                    end

                    // Process bytes from SD card
                    if (reading && byte_available) begin
                        // Handle 16-bit buffer writing (pair of bytes)
                        if (!byte_ready) begin
                            // Store first byte
                            byte_buffer <= sd_dout;
```

```
                            byte_ready <= 1;
                        end else begin
                            // Combine with second byte and write to buffer
                            buffer[buffer_addr_write] <= {byte_buffer, sd_dout};
                            buffer_addr_write <= buffer_addr_write + 1;
                            byte_ready <= 0;
                        end

                        bytes_read <= bytes_read + 1;

                        // Update debug LEDs with the last two bytes
//                          if (bytes_read[0])  // Every other byte
//                              led[15:8] <= sd_dout;
//                          else
//                              led[7:0] <= sd_dout;

                        // Check if we've read a full sector (512 bytes)
                        if (bytes_read == 511) begin
                            reading <= 0;      // Stop reading
                            main_state <= WAIT_SECTOR;
                            sectors_read <= sectors_read + 1;  // Increment sector
counter
                        end
                    end

                    // Handle read completion
                    if (reading && ready && !byte_available && bytes_read > 0)
begin
                        reading <= 0;
                        main_state <= WAIT_SECTOR;
                        sectors_read <= sectors_read + 1;  // Increment sector
counter
                    end
                end

                WAIT_SECTOR: begin
                    // Wait until SD controller is ready again
                    if (ready) begin
                        if (sectors_read < 12) begin
                            main_state <= INIT;  // Read next sector
                        end else begin
```

```
                                main_state <= CALCULATE_CHECKSUM;  // All sectors read,
calculate checksum
                                checksum <= 0;       // Reset checksum
                                checksum_addr <= 0;  // Start from first buffer address
                                checksum_state <= 0; // Reset checksum calculation
state
                                done <= 1;           // Signal all sectors are read
                            end
                        end
                    end

                    CALCULATE_CHECKSUM: begin
                        case (checksum_state)
                            0: begin
                                // Add the current buffer value to checksum
                                checksum <= (checksum + buffer[checksum_addr]) % 65521;
// Modulo 65521 (largest prime under 16 bits)
                                checksum_state <= 1;
                            end
                            1: begin
                                // Move to next address or finish
                                if (checksum_addr < buffer_addr_write - 1) begin //
Check against actual data stored
                                    checksum_addr <= checksum_addr + 1;
                                    checksum_state <= 0; // Go back to state 0 for next
read
                                end else begin
                                    main_state <= DISPLAY_CHECKSUM;
                                end
                                checksum_state <= 0;
                            end
                        endcase
                    end

                    DISPLAY_CHECKSUM: begin
                        // Display the checksum on LEDs
                        //led <= checksum[15:0];  // Show the 16 least significant bits
                        main_state <= DONE;     // Move to DONE state
                    end

                    DONE: begin
                        // Wait for button press to change sector group
```

```
                    done <= 1;  // Signal all sectors are read
                    //led <= current_frame;
                    // led <= buffer[frame_base_addr + ((47 * 64) + 63)];
                    // Keep displaying the checksum - LEDs are not updated here
anymore
                end

                default: main_state <= INIT;
            endcase
        end
    end
  end

  // SD card controller instance
  sd_controller sd_ctrl (
      .cs             (cs),
      .mosi           (mosi),
      .miso           (miso),
      .sclk           (sclk),

      .rd             (rd),
      .dout           (sd_dout),
      .byte_available (byte_available),

      .ready_for_next_byte(),               // Not used in top module

      .reset          (rst),
      .ready          (ready),
      .address        (current_sector),
      .clk            (clk)
  );

endmodule

// Button debouncer module
module debounce (
  input      clk,
  input      reset,
  input      btn_in,
  output reg btn_out
);
  parameter DEBOUNCE_PERIOD = 250000;  // 10ms at 25MHz
```

```
    reg [19:0] counter = 0;
    reg        btn_state = 0;

    always @(posedge clk) begin
        if (reset) begin
            counter <= 0;
            btn_state <= 0;
            btn_out <= 0;
        end else begin
            if (btn_in != btn_state) begin
                // Button state changed, start counting
                counter <= 0;
                btn_state <= btn_in;
            end else if (counter < DEBOUNCE_PERIOD) begin
                // Still counting
                counter <= counter + 1;
            end else begin
                // Debounce period elapsed, update output
                btn_out <= btn_state;
            end
        end
    end
endmodule
```

Module description :

The top module implements a complete digital image viewer system. It reads image data from a microSD card using SPI protocol, buffers the data in internal memory, and displays it on a VGA monitor at 640×480 resolution. The design supports real-time image switching and grayscale display toggling via push buttons.

Challenge Faced

1. Majority of the documents and references are too complex to understand, resulting in delayed initialization of the project.
2. SD Card protocol is hard to implement.
3. Limited hours are available for testing, most of the students don’t own monitors with VGA ports.
4. Minimal help due to shortage of TA, only 1 TA is present most of the time.