Nithya Subramanian
EE371
February 9, 2025
Lab 3 Report

Section 1: Procedure

This lab asked to modify and implement and draw various lines from arbitrary points on a VGA display. This system used VGA output to display the animation, movement, and drawing of a line. I divided the assignment into the following tasks, based on the major components:

1. Drawing any arbitrary line
2. Move and Animate arbitrary lines

Task #1: Drawing any arbitrary lines

Task one we implemented Bresenham's line algorithm to be able to draw any line from arbitrary points on a VGA display. To do this I created a line_drawer module from the pseudo-code provided by the lab document which helped me format/understand how to implement this algorithm. I also added to the already given DE1_SoC module and kept the VGA_framebuffer module the same.

1. The first module is called "DE1_SoC" This is the top-level module that connects/instantiates the line_drawer and VGA_frambuffer modules. It also instantiates the draw color by making the draw color black if SW[9] is off and white if SW[9] is on. The reset is also instantiated to KEY[3] (since active low it's ~KEY[3]) so when KEY[3] is pressed it resets the board by coloring whatever arbitrary points are given to white.
2. The second module was the "line_drawer" module that dealt with each case of arbitrary points, whether it was gradual or steep as well as dealing with where the points start and end. This module also uses these arbitrary points and cycles through each point between the end and beginning by incrementing the x and y coordinates based on whether the line was steep, gradual, neg slop, pos slop, etc.
3. The "VGA_framebuffer" module is the last module and it is taken from the skeleton provided by the lab manual for lab 3. This is the driver module that connects the VGA display to show visible output to the user.

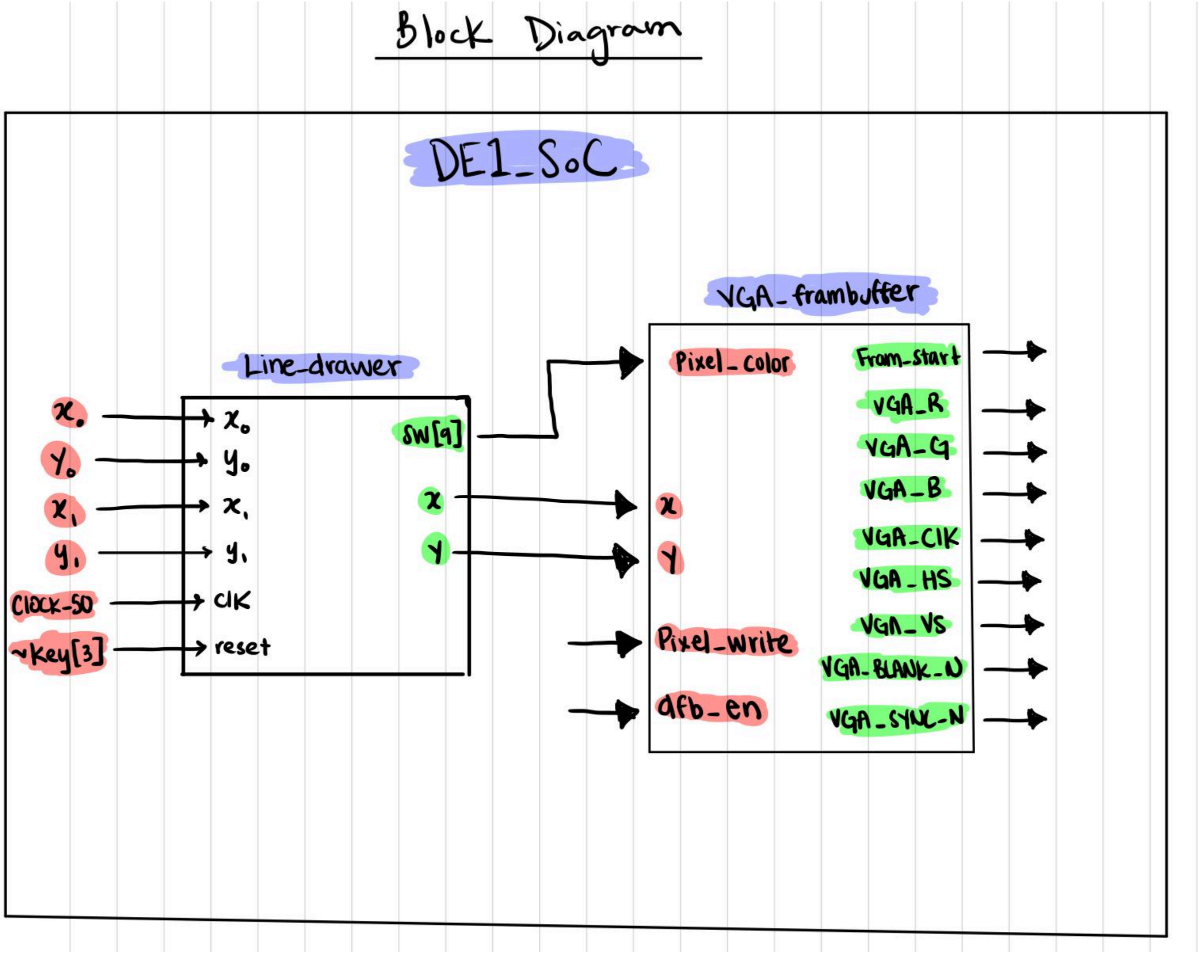


Figure 1: Block diagram of Task 1 - Draw Arbitary Lines

Task #2: Moving/Animating Arbitrary Lines

The second task was to create dual-port RAM by implementing an existing 32x4 RAM. We used 4 modules (3 modules in block diagram, as they were all initialized following this format on the 4th DE1_SoC module)

1. The “line_drawer” module stays the same where the module deals with each case of arbitrary points and cycles through each point between the end and beginning by incrementing the x and y coordinates based on whether the line was steep, gradual, neg slop, pos slop, etc.
2. The “VGP_framebuffer” module stays the same as the last task and it is taken from the skeleton provided by the lab manual for lab 3.
3. The “line_animation” module uses the x0, x1, y1, and y2 start and ending coordinates and increases their values periodically to show the line moving. This module uses a counter to slow down the time each variable is updated (and the line moves) so that the user can see the movement. It takes in the reset and clock signal and outputs 4 new x0, x1, y1, and y2 values.

4. The “DE1_SoC” module initlizations all of these modules and also creates an inProgress singal that serves as a signal that is true when a drawing is in progress so that it can’t be reset when the switch is on. It also uses a RAM to store the coordinates of the lines that are being animated and uses a state machine to go between each set of coordinates in a loop.
5. The “clear_display” module clears the display on two different occasions, it first clears the display after key3 is pressed or when SW9 turns off. It clears the display after a certain amount of time to show the animation recurring over and over again when key 3 is pressed.
6. The “button_press” module uses a FSM to make sure that any press of the button is captured and not overlooked incase it is pressed and finished pressing during the neg edge of a clock.

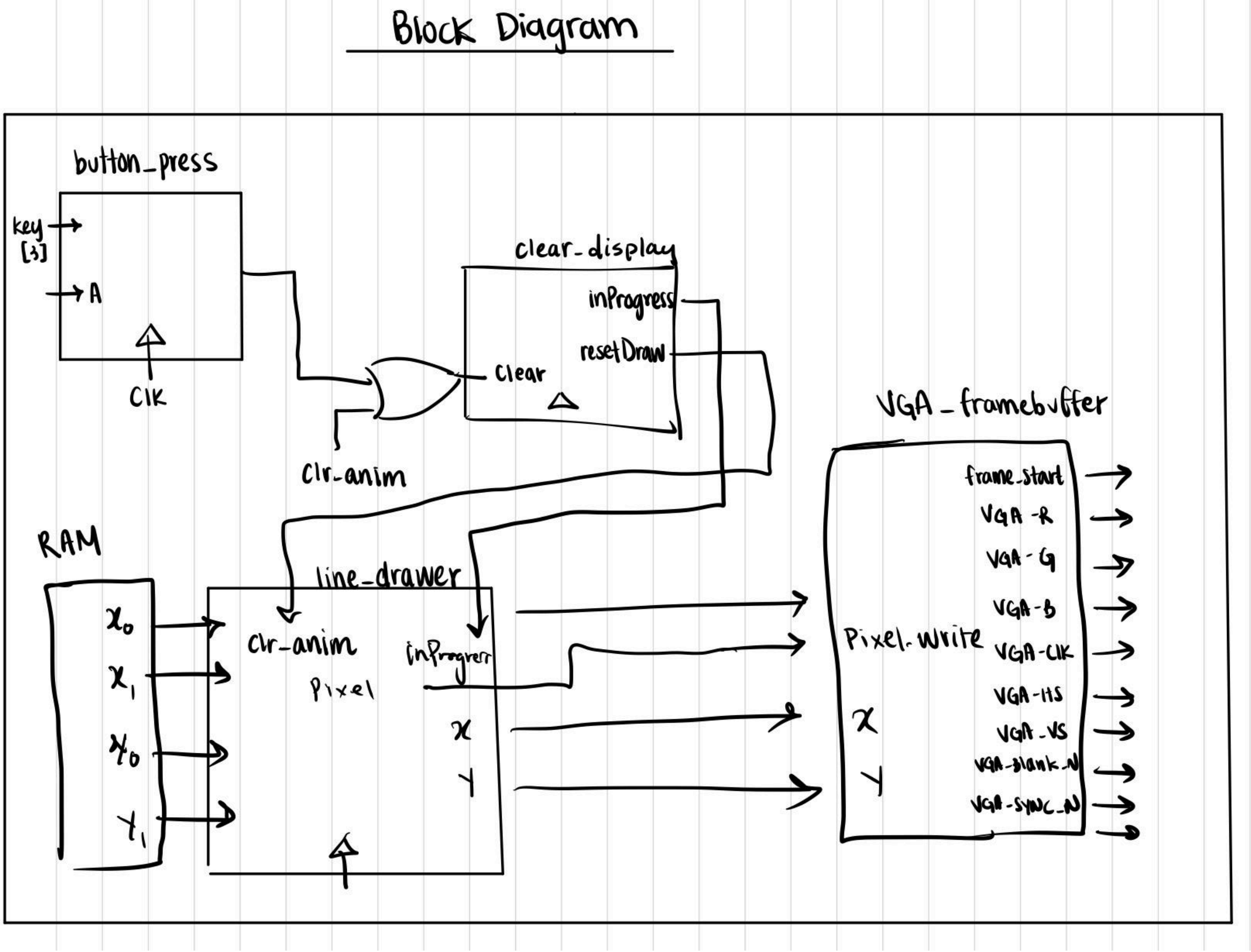


Figure 2: Block Diagram of Task 2 - Animation of Arbitrary Lines

Section 2: Results

Task #1: Drawing Arbitaray Lines

Each of the testbench simulations tested the following situations:

1. **Line_drawer_testbench**: This testbench simulates 7 different types of line drawings, each of which are described below with figures that go along with the description.

Simulation waveforms of line_drawer_testbench:

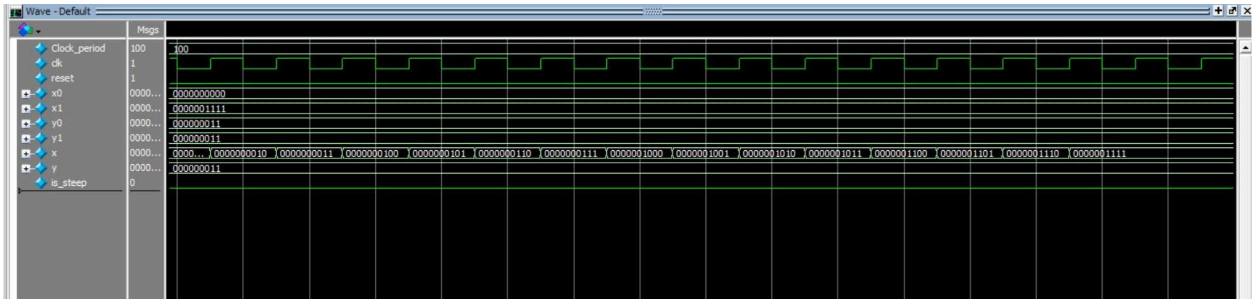


Figure 3.1: ModelSim waveform of Horizontal Line

Shows Situation where a coordinates for x0, x1, y0, y1 are given such that a horizontal line is drawn. As you can see the x coordinates increase by one but the y coordinates stay the same which shows how the horizontal line is being drawn.

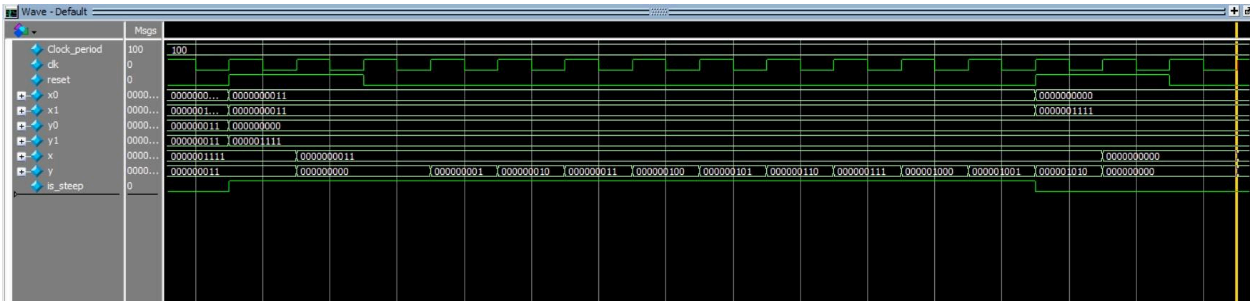


Figure 3.2: ModelSim waveform of Vertical Line

Shows Situation where a coordinates for x0, x1, y0, y1 are given such that a vertical line is drawn. As you can see the y coordinates increase by one but the x coordinates stay the same which shows how the vertical line is being drawn.

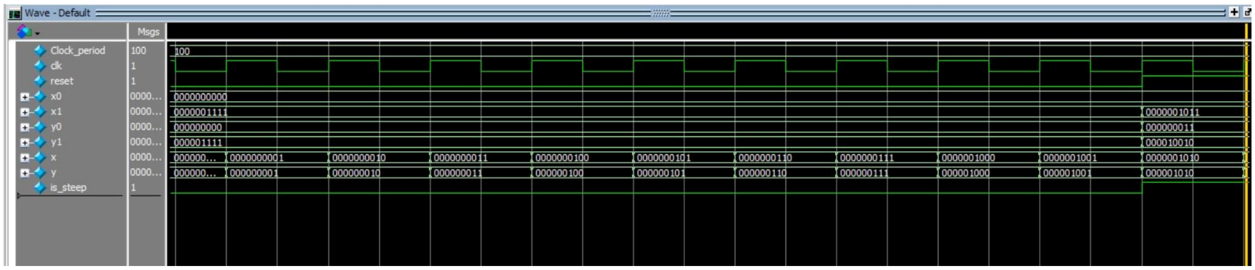


Figure 3.3: ModelSim waveform of diagonal from origin

Shows Situation where a coordinates for x0, x1, y0, y1 are given such that a perfectly diagonal line is drawn from the origin. As you can see the x coordinates and y coordinates increase at the same time/ratio- linearly.

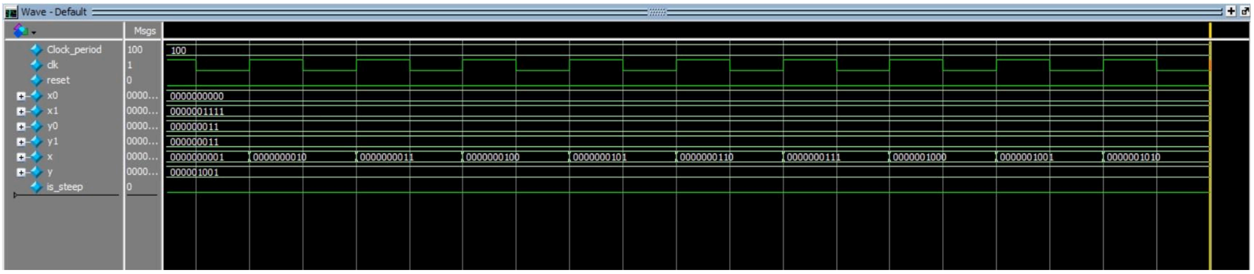


Figure 3.4: ModelSim waveform of gradual negative slope

Shows Situation where a coordinates for x0, x1, y0, y1 are given such that a gradual line with a negative slope is drawn not from the axis of the display. As you can see the x coordinates increase ever so often but the y coordinates increase steadily(a slope that is not the same for rise and run).

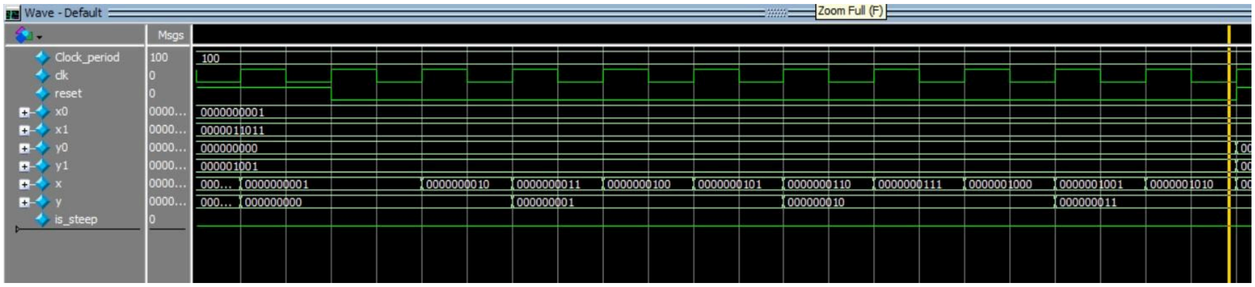


Figure 3.5: ModelSim waveform of gradual positive slope

Shows Situation where a coordinates for x0, x1, y0, y1 are given such that a gradual line with a positive slope is drawn not from the axis of the display. As you can see the y coordinates increase ever so often but the x coordinates increase steadily(a slope that is not the same for rise and run).

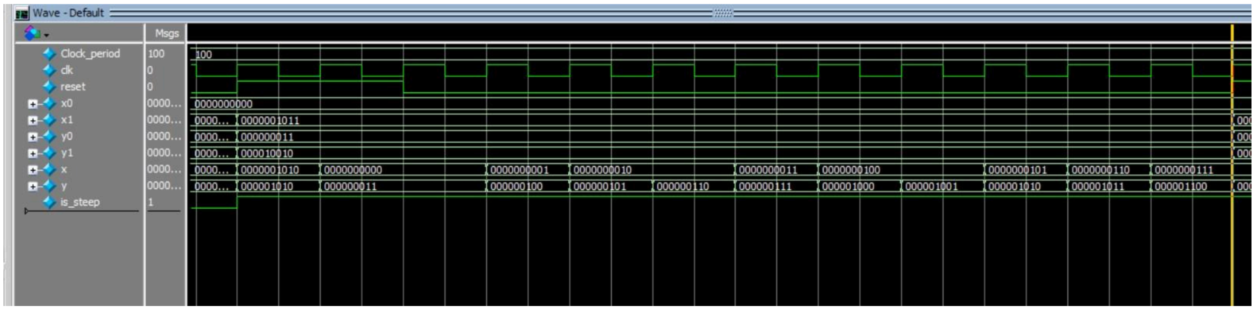

Wave - Default
Msgs
Clock_period 100 100
clk 0
reset 0
x0 0000... 0000000000
x1 0000... 0000... 0000001011
y0 0000... 0000... 000000011
y1 0000... 0000... 000010010
x 0000... 0000... 0000001010 0000000000 0000000001 0000000010 0000000011 0000000100 0000000101 0000000110 0000000111
y 0000... 0000... 000001010 000000011 000000100 000000101 000000110 000000111 000001000 000001001 000001010 000001011 000001100
is_steep 1

Figure 3.6: ModelSim waveform of steep positive slope

Shows Situation where a coordinates for x0, x1, y0, y1 are given such that a steep line with a positive slope is drawn not from the axis of the display. As you can see the y coordinates increase ever so often but the x coordinates increase steadily(a slope that is not the same for rise and run).

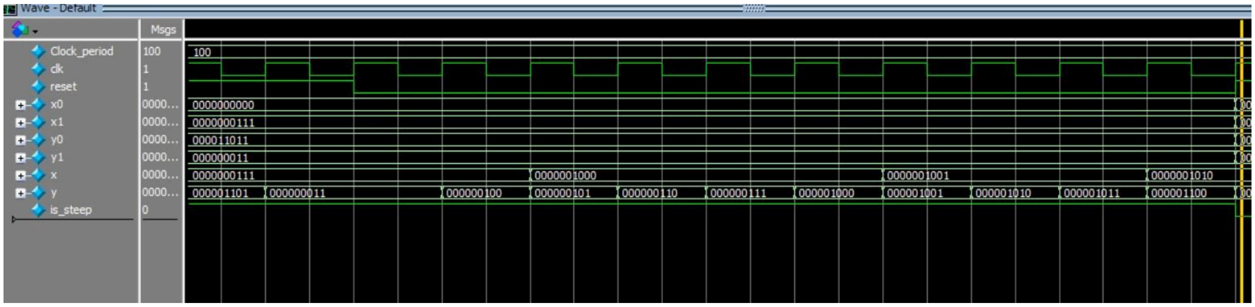

Wave - Default
Msgs
Clock_period 100 100
clk 1
reset 1
x0 0000... 0000000000
x1 0000... 0000000111
y0 0000... 000011011
y1 0000... 000000011
x 0000... 0000000111 0000001000 0000001001 0000001010
y 0000... 000001101 000000011 000000100 000000101 000000110 000000111 000001000 000001001 000001010 000001011 000001100
is_steep 0

Figure 3.7: ModelSim waveform of steep negative slope

Shows Situation where a coordinates for x0, x1, y0, y1 are given such that a steep line with a negative slope is drawn not from the axis of the display. As you can see the x coordinates increase ever so often but the y coordinates increase steadily(a slope that is not the same for rise and run).

2. **DE1_SoC_testbench**: This testbench simulates 7 different types of line drawings, each of which are described below with figures that go along with the description.

Task #2: Moving/Animating Arbitrary Lines

Each of the testbench simulations tested the following situations:

1. **DE1_SoC_testbench**: This testbench simulates
2. **button_press_testbench**: This testbench simulates
3. **clear_display_testbench**: This testbench simulates

Simulation waveforms of DE1_SoC_testbench:

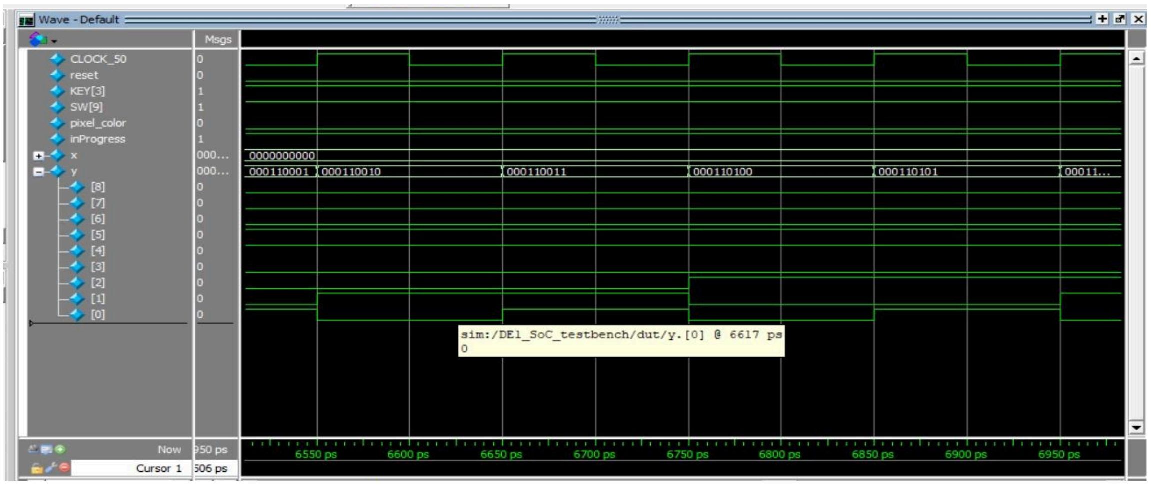


Figure 4: ModelSim waveform of DE1_SoC_testbench

Shows Situation where SW[9] is off and reset is on so there would be no drawing, a blank screen. It also shows where reset is off and SW[9] is on for 100 repeats of a posedge which shows how the display will draw the coordinates stored in the RAM.

Simulation waveforms of buton_press_testbench:

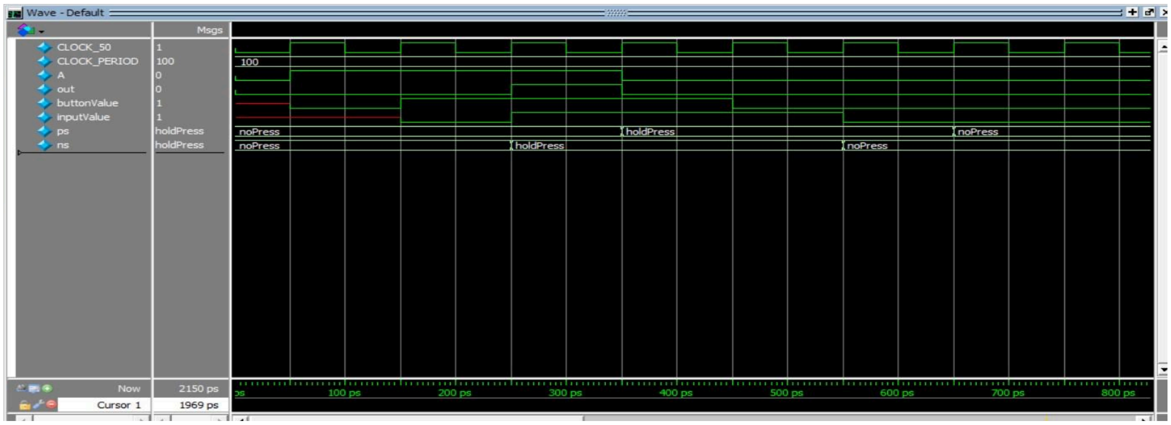
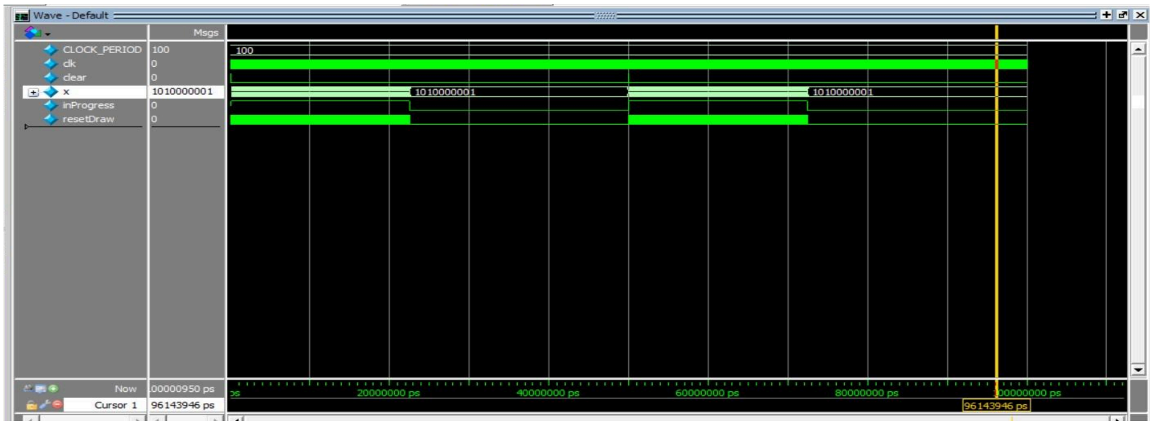


Figure 4: ModelSim waveform of button_press_testbench

Shows Situation where random button presses, A being on and off, are simulated for different amounts of time. When there is a button press you can see that the ps state goes from noPress to holdPress and continues to stay in that state until A is turned off in which it transitions to state noPress again.

Simulation waveforms of clear_display_testbench:



```
CLOCK_PERIOD  100
clk           0
clear         0
x             1010000001
inProgress    0
resetDraw     0
Now           00000950 ps
Cursor 1      96143946 ps
```

Figure 4: ModelSim waveform of clear_display_testbench

Shows Situation that if a clear signal is high and it is run for many clock cycles it cycles through each x coordinate it draws it/colors it in black. The above diagram shows that the clear signal is being used properly as the very close together green lines show the increase in x every 350 clock cycles.

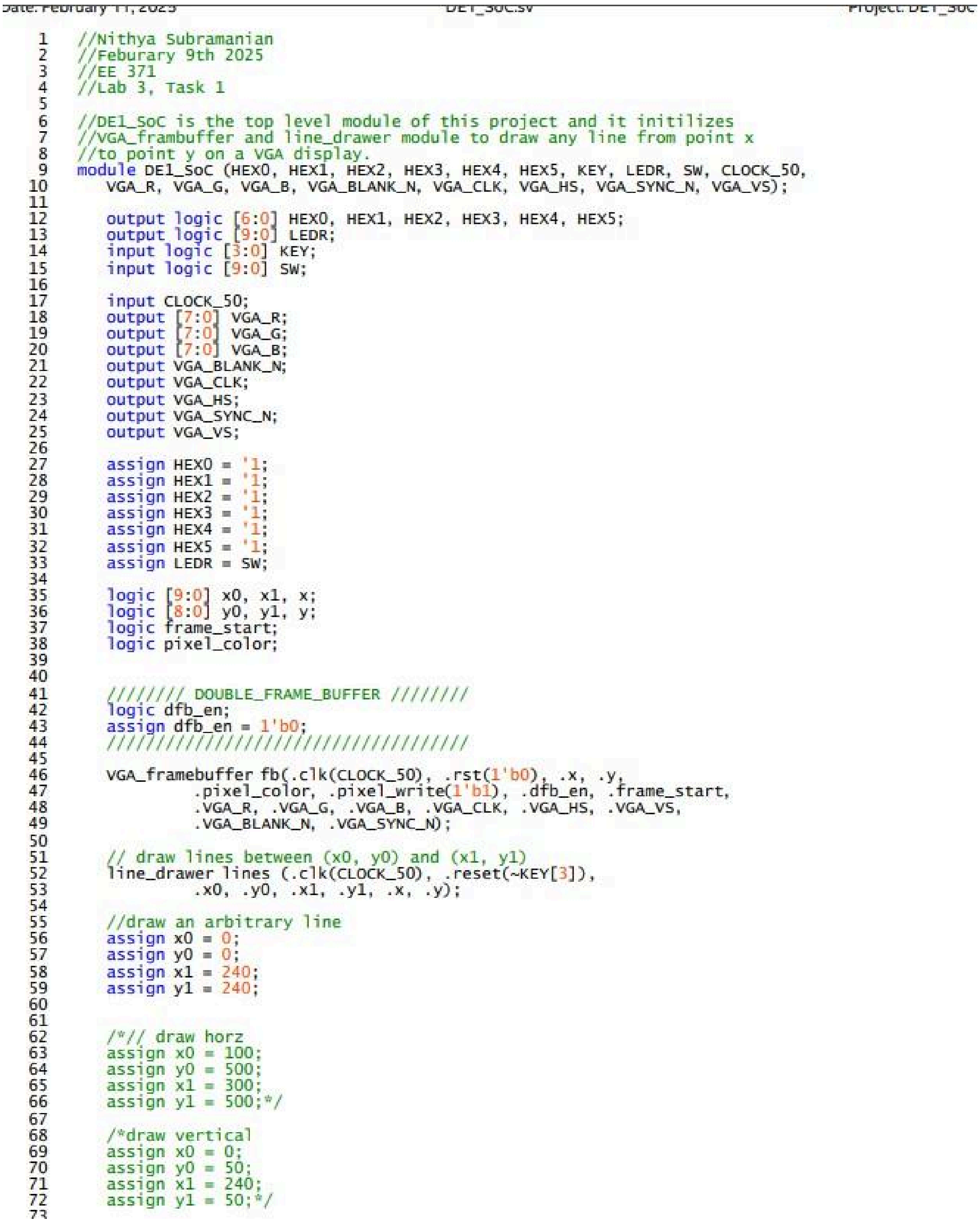
Section 3: Final Product

The final product includes a system that can draw a line from any two arbitrary coordinates/points given by implementing the Bresenham's line algorithm. This system is able to also animate these lines by using a FSM and a RAM to hold the coordinates of multiple points and cycle through them to create a repeating design that clears each time its asked to draw the design again. This system also has clear and reset capabilities where the clear makes the display all black and counties drawing from the beginging if the system is still in progress of drawing or it completely creates a blank display and keeps it that way until the system is said to draw again.

Section 4: Appendix

TASK 1:

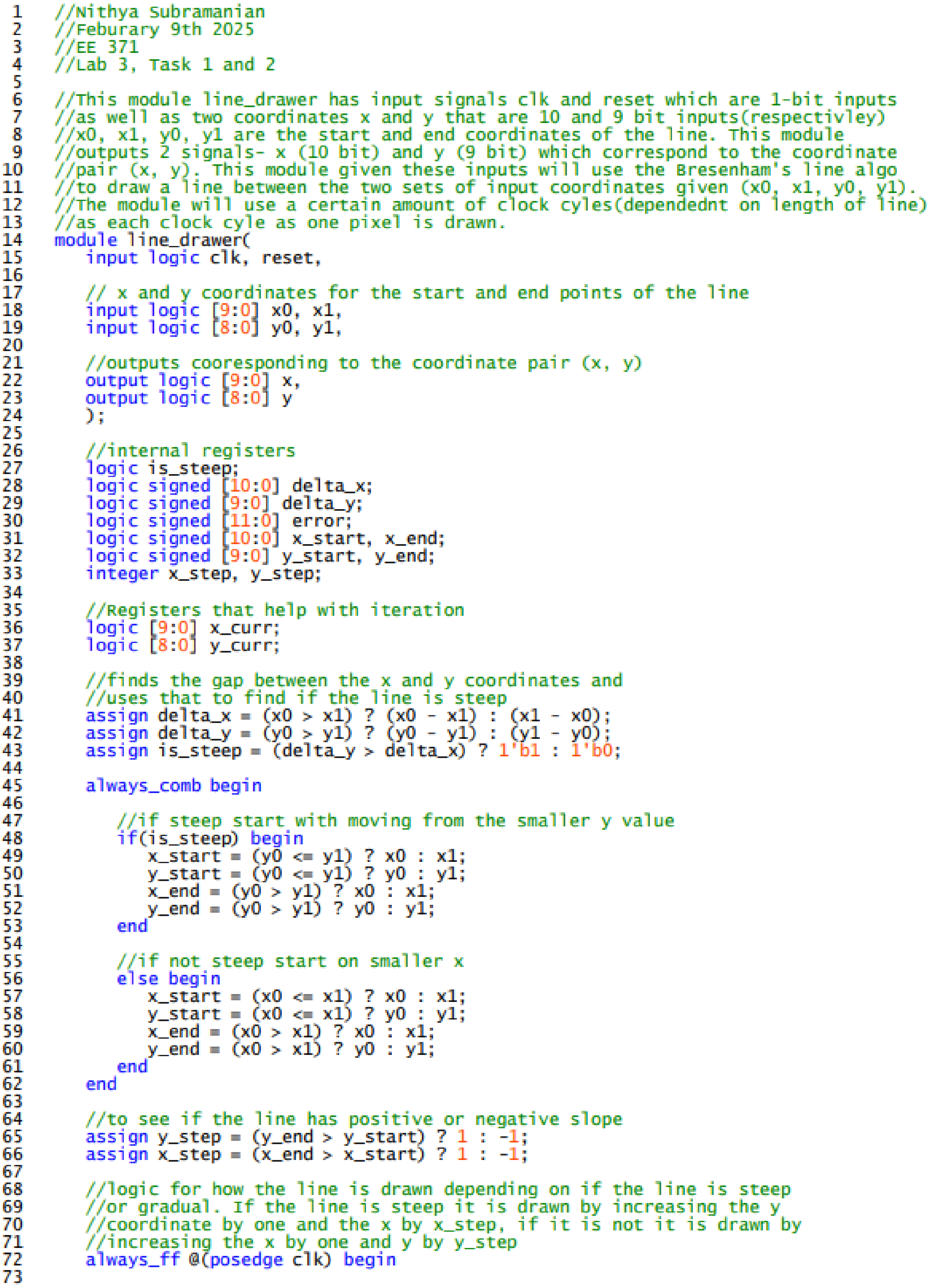
1) DE1_SoC



Date: February 11, 2025 DE1_SoC.sv Project: DE1_SoC

```
//Nithya Subramanian
//Feburary 9th 2025
//EE 371
//Lab 3, Task 1

//DE1_SoC is the top level module of this project and it initilizes
//VGA_frambuffer and line_drawer module to draw any line from point x
//to point y on a VGA display.
module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
   VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS);

   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;
   input logic [3:0] KEY;
   input logic [9:0] SW;

   input CLOCK_50;
   output [7:0] VGA_R;
   output [7:0] VGA_G;
   output [7:0] VGA_B;
   output VGA_BLANK_N;
   output VGA_CLK;
   output VGA_HS;
   output VGA_SYNC_N;
   output VGA_VS;

   assign HEX0 = '1;
   assign HEX1 = '1;
   assign HEX2 = '1;
   assign HEX3 = '1;
   assign HEX4 = '1;
   assign HEX5 = '1;
   assign LEDR = SW;

   logic [9:0] x0, x1, x;
   logic [8:0] y0, y1, y;
   logic frame_start;
   logic pixel_color;


   //////// DOUBLE_FRAME_BUFFER ////////
   logic dfb_en;
   assign dfb_en = 1'b0;
   /////////////////////////////////////

   VGA_framebuffer fb(.clk(CLOCK_50), .rst(1'b0), .x, .y,
            .pixel_color, .pixel_write(1'b1), .dfb_en, .frame_start,
            .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
            .VGA_BLANK_N, .VGA_SYNC_N);

   // draw lines between (x0, y0) and (x1, y1)
   line_drawer lines (.clk(CLOCK_50), .reset(~KEY[3]),
            .x0, .y0, .x1, .y1, .x, .y);

   //draw an arbitrary line
   assign x0 = 0;
   assign y0 = 0;
   assign x1 = 240;
   assign y1 = 240;


   /*// draw horz
   assign x0 = 100;
   assign y0 = 500;
   assign x1 = 300;
   assign y1 = 500;*/

   /*draw vertical
   assign x0 = 0;
   assign y0 = 50;
   assign x1 = 240;
   assign y1 = 50;*/
   /*draw arbitrary pos slop
   assign x0 = 50;
   assign y0 = 100;
   assign x1 = 240;
   assign y1 = 50;*/

   /*draw arbitrary neg slop
   assign x0 = 50;
   assign y0 = 25;
   assign x1 = 100;
   assign y1 = 240;*/


   // draw an arbitrary line for testbench- much smaller
   //assign x0 = 0;
   //assign y0 = 3;
   //assign x1 = 15;
   //assign y1 = 25;

   //assigns write in black if SW[9] off and white if SW[9] on
   assign pixel_color = SW[9] ? 1'b1 : 1'b0;
   //assign pixel_color = 1'b1;

endmodule

//a testbench for DE1_SoC that simulates/test all situations in that testbench
module DE1_SoC_testbench();
   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [9:0] LEDR;
   logic [3:0] KEY;
   logic [9:0] SW;

   logic CLOCK_50;
   logic [7:0] VGA_R;
   logic [7:0] VGA_G;
   logic [7:0] VGA_B;
   logic VGA_BLANK_N;
   logic VGA_CLK;
   logic VGA_HS;
   logic VGA_SYNC_N;
   logic VGA_VS;


   DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW, .CLOCK_50,
   .VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N, .VGA_CLK, .VGA_HS, .VGA_SYNC_N, .VGA_VS);

   logic reset;
   assign KEY[3] = ~reset;

   parameter CLOCK_PERIOD = 100;

   initial begin
      CLOCK_50 <= 100;
      forever #(CLOCK_PERIOD / 2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin
      //check line when reset is on and SW[9] is 0 (black)
      reset <= 1; SW[9] <= 0; repeat (10) @(posedge CLOCK_50);

      //check line drawing when reset off and SW[9] is 1 (white)
      reset <= 0; SW[9] <= 1; repeat (100) @(posedge CLOCK_50);

      $stop;
   end
endmodule
```

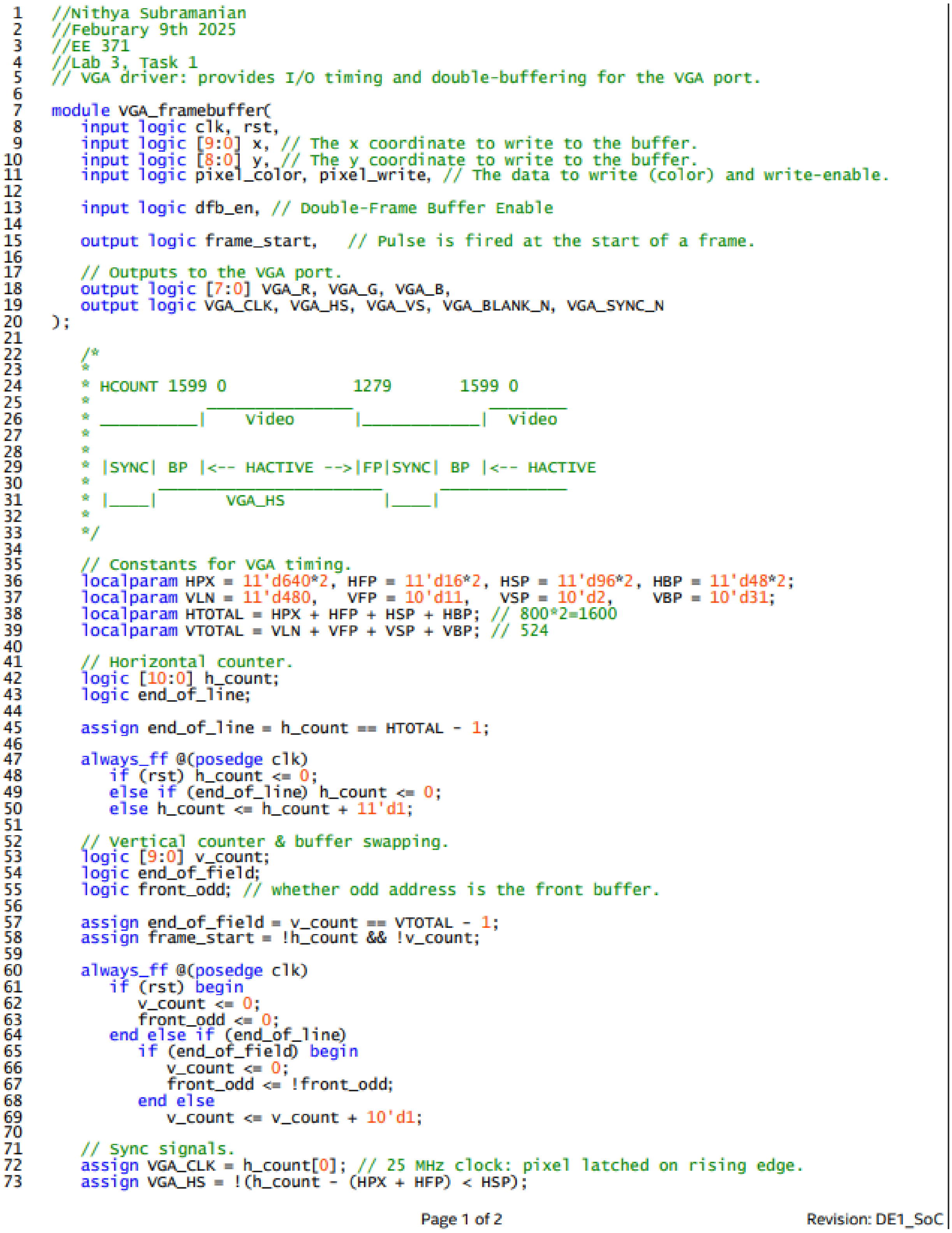
2) line_drawer



```
//Nithya Subramanian
//Feburary 9th 2025
//EE 371
//Lab 3, Task 1 and 2

//This module line_drawer has input signals clk and reset which are 1-bit inputs
//as well as two coordinates x and y that are 10 and 9 bit inputs(respectivley)
//x0, x1, y0, y1 are the start and end coordinates of the line. This module
//outputs 2 signals- x (10 bit) and y (9 bit) which correspond to the coordinate
//pair (x, y). This module given these inputs will use the Bresenham's line algo
//to draw a line between the two sets of input coordinates given (x0, x1, y0, y1).
//The module will use a certain amount of clock cyles(dependednt on length of line)
//as each clock cyle as one pixel is drawn.
module line_drawer(
   input logic clk, reset,

   // x and y coordinates for the start and end points of the line
   input logic [9:0] x0, x1,
   input logic [8:0] y0, y1,

   //outputs cooresponding to the coordinate pair (x, y)
   output logic [9:0] x,
   output logic [8:0] y
   );

   //internal registers
   logic is_steep;
   logic signed [10:0] delta_x;
   logic signed [9:0] delta_y;
   logic signed [11:0] error;
   logic signed [10:0] x_start, x_end;
   logic signed [9:0] y_start, y_end;
   integer x_step, y_step;

   //Registers that help with iteration
   logic [9:0] x_curr;
   logic [8:0] y_curr;

   //finds the gap between the x and y coordinates and
   //uses that to find if the line is steep
   assign delta_x = (x0 > x1) ? (x0 - x1) : (x1 - x0);
   assign delta_y = (y0 > y1) ? (y0 - y1) : (y1 - y0);
   assign is_steep = (delta_y > delta_x) ? 1'b1 : 1'b0;

   always_comb begin

      //if steep start with moving from the smaller y value
      if(is_steep) begin
         x_start = (y0 <= y1) ? x0 : x1;
         y_start = (y0 <= y1) ? y0 : y1;
         x_end = (y0 > y1) ? x0 : x1;
         y_end = (y0 > y1) ? y0 : y1;
      end

      //if not steep start on smaller x
      else begin
         x_start = (x0 <= x1) ? x0 : x1;
         y_start = (x0 <= x1) ? y0 : y1;
         x_end = (x0 > x1) ? x0 : x1;
         y_end = (x0 > x1) ? y0 : y1;
      end
   end

   //to see if the line has positive or negative slope
   assign y_step = (y_end > y_start) ? 1 : -1;
   assign x_step = (x_end > x_start) ? 1 : -1;

   //logic for how the line is drawn depending on if the line is steep
   //or gradual. If the line is steep it is drawn by increasing the y
   //coordinate by one and the x by x_step, if it is not it is drawn by
   //increasing the x by one and y by y_step
   always_ff @(posedge clk) begin

```

```
      if(reset) begin
         x <= x_start;
         y <= y_start;

         error <= (is_steep) ? -delta_y / 2 : -delta_x / 2;
      end

      //if line is step it increases y by one
      else if ((is_steep == 1) & (y != y_end)) begin
         y <= y + 1;

         if((error + delta_x) >= 0) begin
            x <= x + x_step;
            error <= error + (delta_x - delta_y);
         end

         else
            error <= error + delta_x;
      end

      //if line is gradual it increases x by one
      else if (x != x_end) begin
         x <= x + 1;

         if((error + delta_y) >= 0) begin
            y <= y + y_step;
            error <= error + (delta_y - delta_x);
         end

         else
            error <= error + delta_y;
      end
   end
endmodule

//testbench of line_drawer module that simulates/tests all scenarios for this module
module line_drawer_testbench();
   logic clk, reset;
   logic [9:0] x0, x1;
   logic [8:0] y0, y1;
   logic [9:0] x;
   logic [8:0] y;

   line_drawer dut (.clk(clk), .reset(reset), .x0(x0), .x1(x1), .y0(y0), .y1(y1), .x(x), .y(y));

   parameter CLOCK_PERIOD = 100;

   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD / 2) clk <= ~clk;
   end

   initial begin
      //draws a horizontal line from x = 0 to x = 15
      reset <= 1; x0 <= 0; y0 <= 3; x1 <= 15; y1 <= 3; repeat(2) @(posedge clk);
      reset <= 0; repeat(20) @(posedge clk);

      //draws a vertical line from y = 0 to y = 15
      reset <= 1; x0 <= 3; y0 <= 0; x1 <= 3; y1 <= 15; repeat(2) @(posedge clk);
      reset <= 0; x0 <= 3; y0 <= 0; x1 <= 3; y1 <= 15; repeat(10) @(posedge clk);

      //draws a diagonal line starting from origin with positive slope
      reset <= 1; x0 <= 0; y0 <= 0; x1 <= 15; y1 <= 15; repeat(2) @(posedge clk);
      reset <= 0; x0 <= 0; y0 <= 0; x1 <= 15; y1 <= 15; repeat(10) @(posedge clk);

      //draws a diagonal line starting from y axis with positive steep slope
      reset <= 1; x0 <= 0; y0 <= 3; x1 <= 11; y1 <= 18; repeat(2) @(posedge clk);
      reset <= 0; x0 <= 0; y0 <= 3; x1 <= 11; y1 <= 18; repeat(10) @(posedge clk);

      //draws a diagonal starting from y axis with negative steep slope
      reset <= 1; x0 <= 0; y0 <= 27; x1 <= 7; y1 <= 3; repeat(2) @(posedge clk);
      reset <= 0; x0 <= 0; y0 <= 27; x1 <= 7; y1 <= 3; repeat(10) @(posedge clk);

      //draws a line with a positive gradual slope
      reset <= 1; x0 <= 1; y0 <= 0; x1 <= 27; y1 <= 9; repeat(2) @(posedge clk);
      reset <= 0; x0 <= 1; y0 <= 0; x1 <= 27; y1 <= 9; repeat(10) @(posedge clk);

      //draws a line with a negative gradual slope
      reset <= 1; x0 <= 1; y0 <= 9; x1 <= 27; y1 <= 0; repeat(2) @(posedge clk);
      reset <= 0; x0 <= 0; y0 <= 3; x1 <= 15; y1 <= 3; repeat(10) @(posedge clk);
      $stop;
   end
endmodule
```

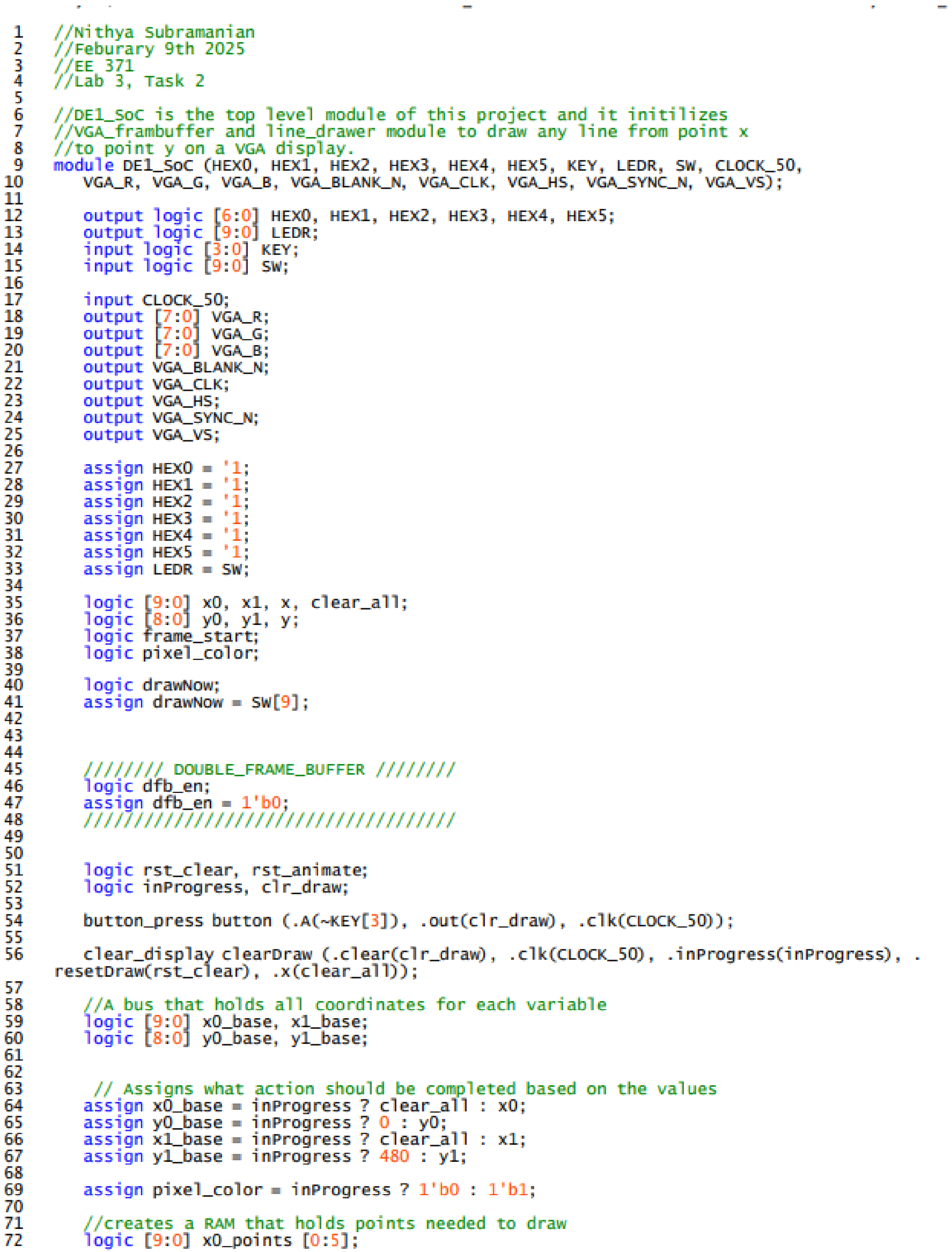
3) VGA_framebuffer



```
//Nithya Subramanian
//Feburary 9th 2025
//EE 371
//Lab 3, Task 1
// VGA driver: provides I/O timing and double-buffering for the VGA port.

module VGA_framebuffer(
   input logic clk, rst,
   input logic [9:0] x, // The x coordinate to write to the buffer.
   input logic [8:0] y, // The y coordinate to write to the buffer.
   input logic pixel_color, pixel_write, // The data to write (color) and write-enable.

   input logic dfb_en, // Double-Frame Buffer Enable

   output logic frame_start,   // Pulse is fired at the start of a frame.

   // Outputs to the VGA port.
   output logic [7:0] VGA_R, VGA_G, VGA_B,
   output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N
);

   /*
   *
   * HCOUNT 1599 0             1279       1599 0
   *           _______________              ________
   * __________|     Video     |____________|  Video
   *
   *
   * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
   *       _______________________      _____________
   * |____|       VGA_HS          |____|
   *
   */

   // Constants for VGA timing.
   localparam HPX = 11'd640*2, HFP = 11'd16*2, HSP = 11'd96*2, HBP = 11'd48*2;
   localparam VLN = 11'd480,   VFP = 10'd11,   VSP = 10'd2,    VBP = 10'd31;
   localparam HTOTAL = HPX + HFP + HSP + HBP; // 800*2=1600
   localparam VTOTAL = VLN + VFP + VSP + VBP; // 524

   // Horizontal counter.
   logic [10:0] h_count;
   logic end_of_line;

   assign end_of_line = h_count == HTOTAL - 1;

   always_ff @(posedge clk)
      if (rst) h_count <= 0;
      else if (end_of_line) h_count <= 0;
      else h_count <= h_count + 11'd1;

   // Vertical counter & buffer swapping.
   logic [9:0] v_count;
   logic end_of_field;
   logic front_odd; // whether odd address is the front buffer.

   assign end_of_field = v_count == VTOTAL - 1;
   assign frame_start = !h_count && !v_count;

   always_ff @(posedge clk)
      if (rst) begin
         v_count <= 0;
         front_odd <= 0;
      end else if (end_of_line)
         if (end_of_field) begin
            v_count <= 0;
            front_odd <= !front_odd;
         end else
            v_count <= v_count + 10'd1;

   // Sync signals.
   assign VGA_CLK = h_count[0]; // 25 MHz clock: pixel latched on rising edge.
   assign VGA_HS = !(h_count - (HPX + HFP) < HSP);
   assign VGA_VS = !(v_count - (VLN + VFP) < VSP);
   assign VGA_SYNC_N = 1; // Unused by VGA

   // Blank area signal.
   logic blank;
   assign blank = h_count >= HPX || v_count >= VLN;

   // Double-buffering.
   logic buffer[640*480*2-1:0];
   logic [19:0] wr_addr, rd_addr;
   logic rd_data;

   assign wr_addr = {y * 19'd640 + x, (!front_odd & dfb_en)};
   assign rd_addr = {v_count * 19'd640 + (h_count / 19'd2), (front_odd & dfb_en)};

   always_ff @(posedge clk) begin
      if (pixel_write) buffer[wr_addr] <= pixel_color;
      if (VGA_CLK) begin
         rd_data <= buffer[rd_addr];
         VGA_BLANK_N <= ~blank;
      end
   end

   // Color output.
   assign {VGA_R, VGA_G, VGA_B} = rd_data ? 24'hFFFFFF : 24'h000000;
endmodule

```

TASK 2:

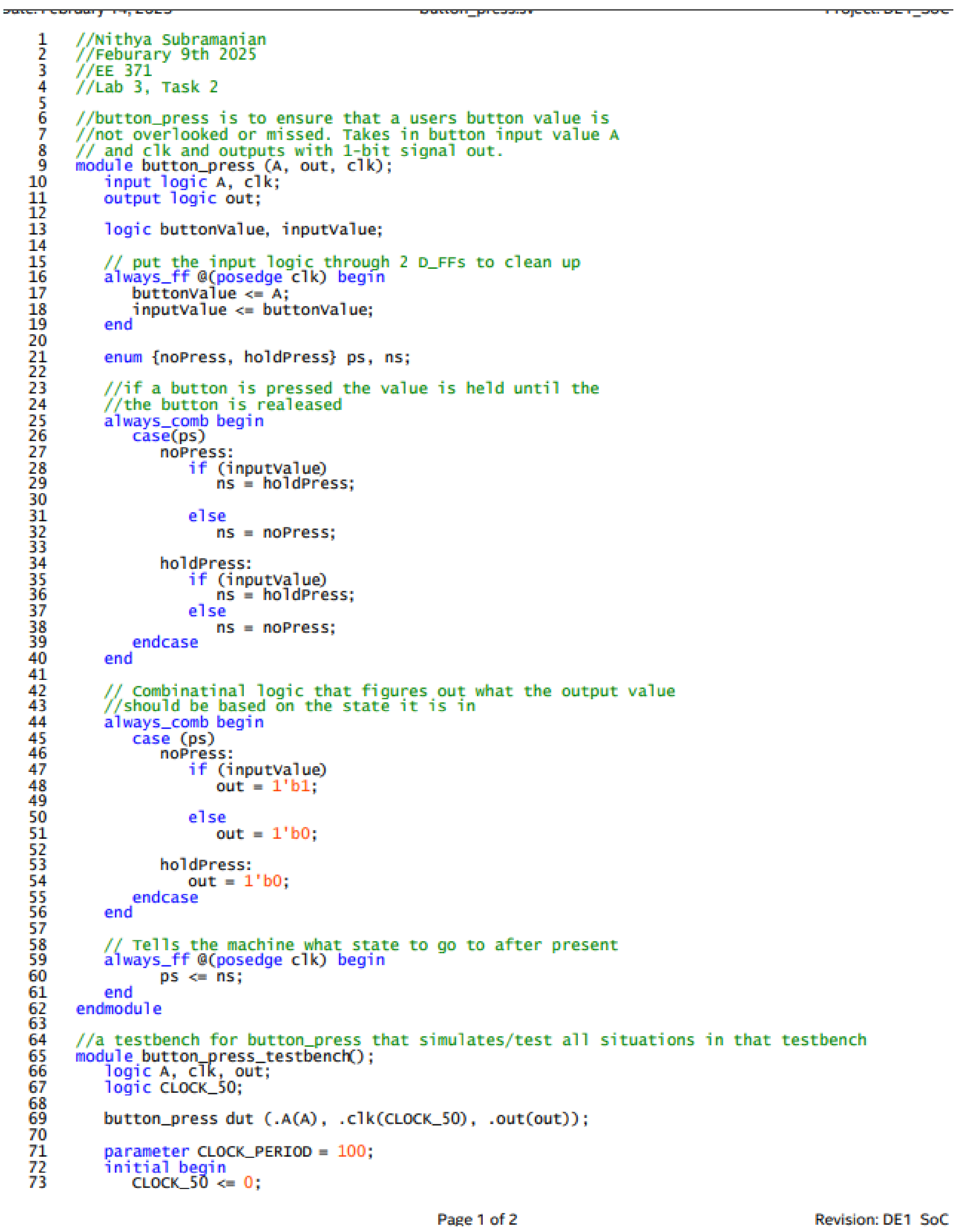
1) DE1_SoC



```
//Nithya Subramanian
//Feburary 9th 2025
//EE 371
//Lab 3, Task 2

//DE1_SoC is the top level module of this project and it initilizes
//VGA_frambuffer and line_drawer module to draw any line from point x
//to point y on a VGA display.
module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
   VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS);

   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;
   input logic [3:0] KEY;
   input logic [9:0] SW;

   input CLOCK_50;
   output [7:0] VGA_R;
   output [7:0] VGA_G;
   output [7:0] VGA_B;
   output VGA_BLANK_N;
   output VGA_CLK;
   output VGA_HS;
   output VGA_SYNC_N;
   output VGA_VS;

   assign HEX0 = '1;
   assign HEX1 = '1;
   assign HEX2 = '1;
   assign HEX3 = '1;
   assign HEX4 = '1;
   assign HEX5 = '1;
   assign LEDR = SW;

   logic [9:0] x0, x1, x, clear_all;
   logic [8:0] y0, y1, y;
   logic frame_start;
   logic pixel_color;

   logic drawNow;
   assign drawNow = SW[9];



   //////// DOUBLE_FRAME_BUFFER ////////
   logic dfb_en;
   assign dfb_en = 1'b0;
   /////////////////////////////////////


   logic rst_clear, rst_animate;
   logic inProgress, clr_draw;

   button_press button (.A(~KEY[3]), .out(clr_draw), .clk(CLOCK_50));

   clear_display clearDraw (.clear(clr_draw), .clk(CLOCK_50), .inProgress(inProgress), .
resetDraw(rst_clear), .x(clear_all));

   //A bus that holds all coordinates for each variable
   logic [9:0] x0_base, x1_base;
   logic [8:0] y0_base, y1_base;


    // Assigns what action should be completed based on the values
   assign x0_base = inProgress ? clear_all : x0;
   assign y0_base = inProgress ? 0 : y0;
   assign x1_base = inProgress ? clear_all : x1;
   assign y1_base = inProgress ? 480 : y1;

   assign pixel_color = inProgress ? 1'b0 : 1'b1;

   //creates a RAM that holds points needed to draw
   logic [9:0] x0_points [0:5];
   logic [8:0] y0_points [0:5];
   logic [9:0] x1_points [0:5];
   logic [8:0] y1_points [0:5];

   assign x0_points[0] = 240;
   assign x0_points[1] = 340;
   assign x0_points[2] = 340;
   assign x0_points[3] = 240;
   assign x0_points[4] = 240;
   assign x0_points[5] = 290;

   assign y0_points[0] = 340;
   assign y0_points[1] = 340;
   assign y0_points[2] = 240;
   assign y0_points[3] = 240;
   assign y0_points[4] = 290;
   assign y0_points[5] = 340;

   assign x1_points[0] = 240;
   assign x1_points[1] = 240;
   assign x1_points[2] = 180;
   assign x1_points[3] = 180;
   assign x1_points[4] = 240;
   assign x1_points[5] = 350;

   assign y1_points[0] = 240;
   assign y1_points[1] = 180;
   assign y1_points[2] = 180;
   assign y1_points[3] = 240;
   assign y1_points[4] = 350;
   assign y1_points[5] = 240;

   //initilizes VGA_framebuffer
   VGA_framebuffer fb(
     .clk(CLOCK_50), .rst(1'b0), .x, .y,
     .pixel_color, .pixel_write(1'b1), .dfb_en(), .frame_start,
     .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
     .VGA_BLANK_N, .VGA_SYNC_N);

   //initilizes line_drawer
   line_drawer lines (
     .clk(CLOCK_50),
     .reset(rst_clear | rst_animate),
     .x0(x0_base), .y0(y0_base), .x1(x1_base), .y1(y1_base), .x, .y);

   integer counter, index;

   //uses different signals to find whether the machine should draw, clear ect. it also uses
   //a counter to slow down the time taken to update the machine
   always_ff @(posedge CLOCK_50) begin
      if (clr_draw) begin
         counter <= 0;
         index <= 0;
         x0 <= 0;
         y0 <= 0;
         x1 <= 0;
         y1 <= 0;
      end

      else if (counter == 1) begin
         counter <= counter + 1;
         rst_animate <= 1'b1;
      end

      else if (counter == 5) begin
         counter <= counter + 1;
         rst_animate <= 1'b0;
      end

      else if (counter == (50000000 / 2)) begin
         counter <= 0;

         if (index < 5) begin
```

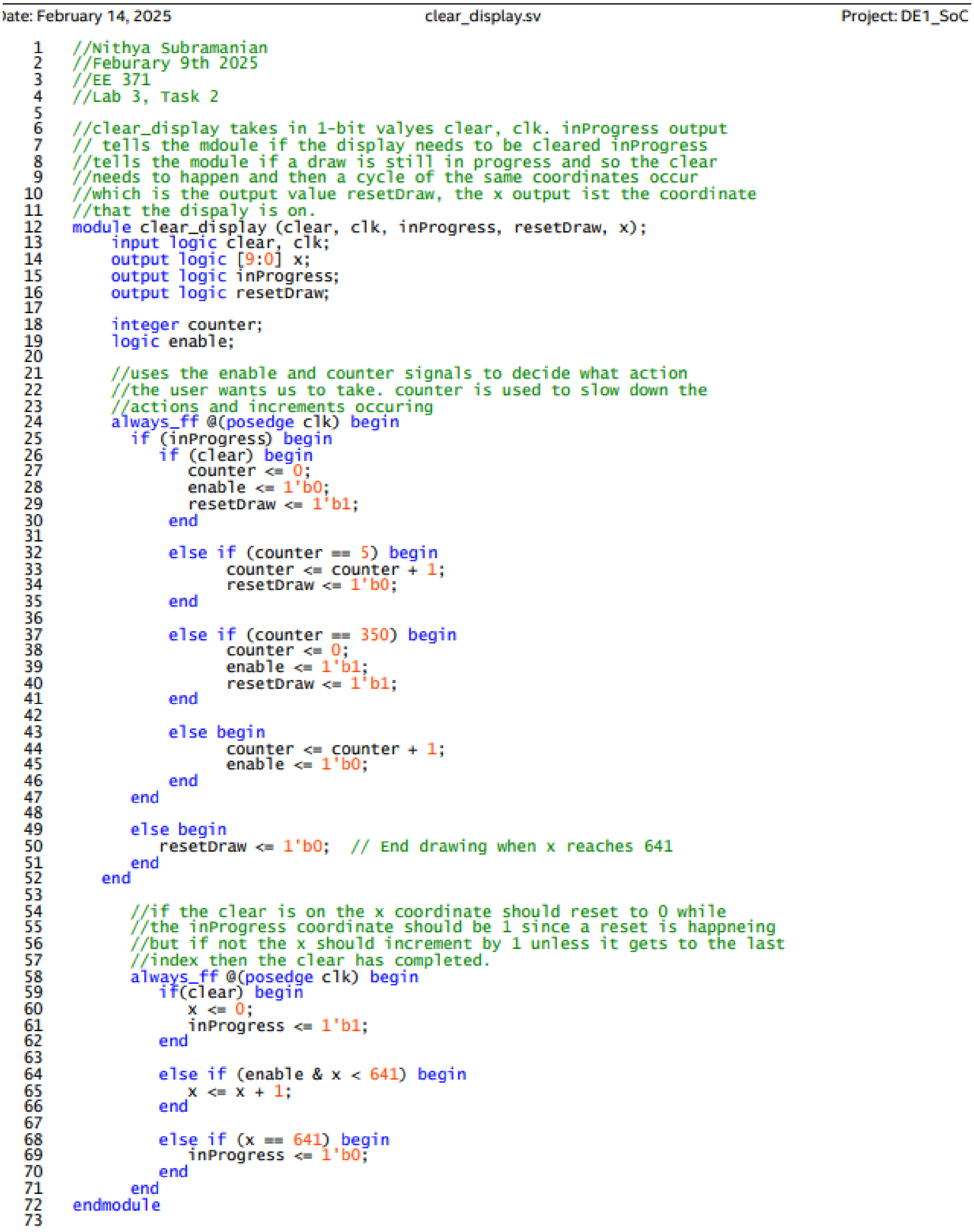
```
            index <= index + 1;
         end

         else begin
            index <= 0;
         end

            x0 <= x0_points[index];
            y0 <= y0_points[index];
            x1 <= x1_points[index];
            y1 <= y1_points[index];
      end

      else if (drawNow) begin
         counter <= counter + 1;
      end
   end
endmodule


//a testbench for DE1_SoC that simulates/test all situations in that testbench
module DE1_SoC_testbench();
   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [9:0] LEDR;
   logic [3:0] KEY;
   logic [9:0] SW;

   logic CLOCK_50;
   logic [7:0] VGA_R;
   logic [7:0] VGA_G;
   logic [7:0] VGA_B;
   logic VGA_BLANK_N;
   logic VGA_CLK;
   logic VGA_HS;
   logic VGA_SYNC_N;
   logic VGA_VS;


   DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW, .CLOCK_50,
   .VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N, .VGA_CLK, .VGA_HS, .VGA_SYNC_N, .VGA_VS);

   logic reset;
   assign KEY[3] = ~reset;

   parameter CLOCK_PERIOD = 100;

   initial begin
      CLOCK_50 <= 100;
      forever #(CLOCK_PERIOD / 2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin
      //check line when reset is on and SW[9] is 0 (black)
      reset <= 1; SW[9] <= 0; repeat (10) @(posedge CLOCK_50);

      //check line drawing when reset off and SW[9] is 1 (white)
      reset <= 0; SW[9] <= 1; repeat (100) @(posedge CLOCK_50);

      $stop;
   end
endmodule

```

2) button_press



```
//Nithya Subramanian
//Feburary 9th 2025
//EE 371
//Lab 3, Task 2

//button_press is to ensure that a users button value is
//not overlooked or missed. Takes in button input value A
// and clk and outputs with 1-bit signal out.
module button_press (A, out, clk);
   input logic A, clk;
   output logic out;

   logic buttonValue, inputValue;

   // put the input logic through 2 D_FFs to clean up
   always_ff @(posedge clk) begin
      buttonValue <= A;
      inputValue <= buttonValue;
   end

   enum {noPress, holdPress} ps, ns;

   //if a button is pressed the value is held until the
   //the button is realeased
   always_comb begin
      case(ps)
         noPress:
            if (inputValue)
               ns = holdPress;

            else
               ns = noPress;

         holdPress:
            if (inputValue)
               ns = holdPress;
            else
               ns = noPress;
      endcase
   end

   // Combinatinal logic that figures out what the output value
   //should be based on the state it is in
   always_comb begin
      case (ps)
         noPress:
            if (inputValue)
               out = 1'b1;

            else
               out = 1'b0;

         holdPress:
            out = 1'b0;
      endcase
   end

   // Tells the machine what state to go to after present
   always_ff @(posedge clk) begin
         ps <= ns;
   end
endmodule

//a testbench for button_press that simulates/test all situations in that testbench
module button_press_testbench();
   logic A, clk, out;
   logic CLOCK_50;

   button_press dut (.A(A), .clk(CLOCK_50), .out(out));

   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin

      //simulate random times of button presses
      A <= 0; repeat(7); @(posedge CLOCK_50);
      A <= 1; repeat(3) @(posedge CLOCK_50);
      A <= 0; repeat(5) @(posedge CLOCK_50);
      A <= 1; repeat(1) @(posedge CLOCK_50);
      A <= 0; repeat(6) @(posedge CLOCK_50);
      A <= 1; repeat(4) @(posedge CLOCK_50);
      A <= 0; repeat(2) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

3) Clear_display



ate: February 14, 2025 clear_display.sv Project: DE1_SoC

```
//Nithya Subramanian
//Feburary 9th 2025
//EE 371
//Lab 3, Task 2

//clear_display takes in 1-bit valyes clear, clk. inProgress output
// tells the mdoule if the display needs to be cleared inProgress
//tells the module if a draw is still in progress and so the clear
//needs to happen and then a cycle of the same coordinates occur
//which is the output value resetDraw, the x output ist the coordinate
//that the dispaly is on.
module clear_display (clear, clk, inProgress, resetDraw, x);
    input logic clear, clk;
    output logic [9:0] x;
    output logic inProgress;
    output logic resetDraw;

    integer counter;
    logic enable;

    //uses the enable and counter signals to decide what action
    //the user wants us to take. counter is used to slow down the
    //actions and increments occuring
    always_ff @(posedge clk) begin
      if (inProgress) begin
         if (clear) begin
            counter <= 0;
            enable <= 1'b0;
            resetDraw <= 1'b1;
          end

          else if (counter == 5) begin
                counter <= counter + 1;
                resetDraw <= 1'b0;
          end

          else if (counter == 350) begin
                counter <= 0;
                enable <= 1'b1;
                resetDraw <= 1'b1;
          end

          else begin
                counter <= counter + 1;
                enable <= 1'b0;
          end
      end

      else begin
         resetDraw <= 1'b0;   // End drawing when x reaches 641
      end
   end

      //if the clear is on the x coordinate should reset to 0 while
      //the inProgress coordinate should be 1 since a reset is happneing
      //but if not the x should increment by 1 unless it gets to the last
      //index then the clear has completed.
      always_ff @(posedge clk) begin
         if(clear) begin
            x <= 0;
            inProgress <= 1'b1;
         end

         else if (enable & x < 641) begin
            x <= x + 1;
         end

         else if (x == 641) begin
            inProgress <= 1'b0;
         end
      end
endmodule
```

```
//a testbench for clear_display that simulates/test all situations in that testbench
module clear_display_testbench();
   logic clear, clk;
   logic [9:0] x;
   logic inProgress;
   logic resetDraw;

   clear_display dut (.*);

   parameter CLOCK_PERIOD = 100;

   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD / 2) clk <= ~clk;
   end

   initial begin
      clear <= 1;       repeat(5)      @(posedge clk);
      clear <= 0;       repeat(500000) @(posedge clk);
      clear <= 1;       repeat(5)      @(posedge clk);
      clear <= 0;       repeat(500000) @(posedge clk);
      $stop;
   end
endmodule
```