Brian Dallaire
EE371
5/7/2021

Lab 3 Report

Procedure

This lab was compromised of two tasks. The first task was to implement the line_drawer module so that when inputting any pair of coordinates within the VGA display, a line will be drawn from the first pair of coordinates to the other. This is done by following Bresenham's Line Algorithm to distribute the pixels being drawn in a way that makes the line look as straight as possible. The main challenge of this task is to design line_drawer so that it can draw any kind of slope from any position on the display. The second task was to animate lines on the VGA display. This was a more open-ended task, but the main goal was to have lines change in slope, position, etc. in real time so the line or lines look animated. Additionally, a reset has to make the entire screen dark or erased. Overall, this lab was about understanding how to display simple lines on the VGA display and fully understanding this concept allows for "animating" lines on the display. Although we are not expected to understand how the VGA buffer works, the main goal of this lab was to learn how to manage inputs into the VGA buffer so that it draws expected behavior onto the display.

Task #1

The first task was to implement the line_drawer module to draw a single line from any given pair of (x,y) coordinates. Because there are a limited number of pixels on the VGA display and there is no way to partially light up one individual pixel, the lines will not be perfect. To determine which pixels should be lit up to form the line, I had to understand Bresenham's Line Algorithm. As shown in Figure 1, the algorithm finds which pixel best represents the current position of the line. That pixel will be the one that is written into. While it may not look perfect, on a 640x480 display, it will look moderately accurate.

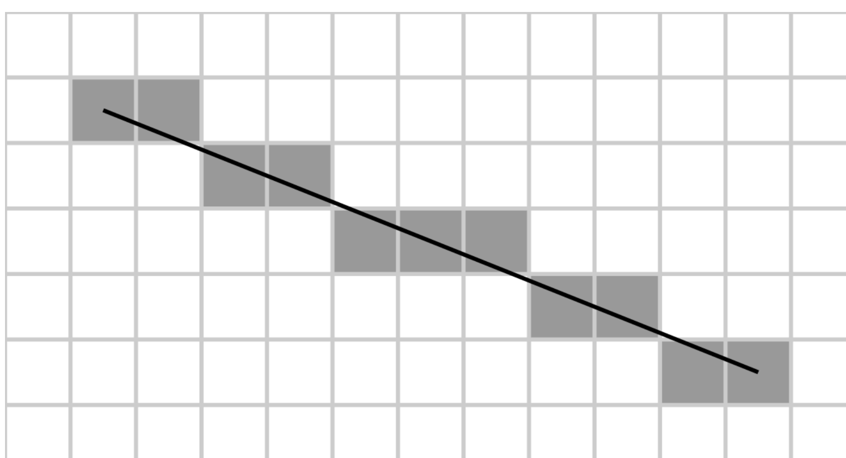

Figure 1. Bresenham's Line Algorithm and the Distribution of Pixels in a Line.

For this interpretation of the line algorithm, I start at (x0, y0) and move along the x-axis to compute the y coordinate for the line. For my interpretation, the x value will increment by 1 every time, but the y-values have to be calculated using slope and difference in x, as shown in Figure 2.

$$Slope = \frac{y_1 - y_0}{x_1 - x_0}$$

$$y = y_0 + slope * (x - x_0)$$

Figure 2. Equations for Calculating y-value

There is an issue with reading along the x-axis when a line is considered to be steep. A line is considered to be steep when the change in y values is greater than the change in x values. When this occurs, I have to make it so that I move along the y-axis to compute the coordinate for the x coordinate. Essentially, I do the same thing as before but flip the x and y values. I implemented this in SystemVerilog code using conditional operators as shown in Figure 3. Conditional operators are a great tool to assign different values based on a condition without using the space required for an if else statement.

```
deltax   = (x1 > x0) ? (x1 - x0) : (x0 - x1);
deltay   = (y1 > y0) ? (y1 - y0) : (y0 - y1);
is_steep = (deltay > deltax);
dx = !(is_steep) ? deltax : deltay;
dy = !(is_steep) ? deltay : deltax;

xa = (!is_steep & (x1<x0)) ? x1 : (!is_steep & (x1>x0)) ? x0 : (is_steep & (y1<y0)) ? y1 : y0;
xb = (!is_steep & (x1<x0)) ? x0 : (!is_steep & (x1>x0)) ? x1 : (is_steep & (y1<y0)) ? y0 : y1;
ya = (!is_steep & (x1<x0)) ? y1 : (!is_steep & (x1>x0)) ? y0 : (is_steep & (y1<y0)) ? x1 : x0;
yb = (!is_steep & (x1<x0)) ? y0 : (!is_steep & (x1>x0)) ? y1 : (is_steep & (y1<y0)) ? x0 : x1;
```

Figure 3. Code showing how x initial, x final, y initial, y final values are swapped depending on conditions such as if it is steep or not.

Using conditional operators, I can determine when to flip the x and y values. Checking to see if the slope is steep or not is what primarily determines what should be reassigned to each x and y value.

Lastly, I just needed to calculate the error and increment x and y accordingly. The code I used for this can be seen in Figure 4. To calculate the error, I followed the pseudo code given to me in the Lab Spec. Initially, error is simply -dx/2 just as a rough estimate. So long as x is not x1, error will update to be error + dy. Then, the next error is determined depending on if the current error value is greater than or equal to 0. If it is greater than or equal to zero, the next error is error – dx. If it is not, then the next error remains the same. From here all I do is increment x and y. Since we implemented a system for flipping the x and y values when necessary, in the code we will always be following the “x” value. The next x value will simply be incrementing it by 1. The next y value is determined by whether the initial y value is less than the final y value or not and if the error is greater than or equal to zero.

```
err_Temp = -(dx/2);
err_Val = err + dy;
err_Next = (err_Val >= 0) ? err_Val - dx : err_Val;

step_y = (ya < yb) ? 1 : -1;

next_x = xval + 1'b1;
next_y = (err_Val >= 0) ? yval + step_y : yval;

x = !(is_steep) ? xval : yval;
y = !(is_steep) ? yval : xval;
```

Figure 4. Code showing the calculation for error and incrementing/decrementing x and y values for the next cycle, as well as updating the output x and y values.

To update the system, I updated key variables in an always_ff block as seen in Figure 5. This block dictates the behavior of the line_drawer module when reset is true, when the line is finished, and when the line needs to be updated to the next x and y values. The reset and unfinished line portions are straight forward. To finish the line, it is as simple as comparing the current x value to the final x value and the current y value to the final y value. If both of these comparisons are equal, then the last pixel that needs to be written into has been written into, so the line is done. From here, the line will remain in place and not write any more pixels, effectively finishing the line.

```
    always_ff @(posedge clk) begin
        if(reset) begin
            xval <= xa;
            yval <= ya;
            err  <= err_Temp;
        end else begin
            if(((xb == xval) && (yb == yval))) begin
                xval <= xval;
                yval <= yval;
                err  <= err;
            end else begin
                xval <= next_x;
                yval <= next_y;
                err  <= err_Next;
            end
        end
    end
endmodule
```

Figure 6. Code for updating the behavior in line_drawer.sv

Task #2

The second task was to create an animation for drawing lines onto the VGA display. This was a rather open-ended task, so approaching this task was difficult. My first thought was that to show multiple lines to create an animation effect, I would have to erase each line after I draw it, then update the coordinates that go into line_drawer for the next line. To do this, I would have to set the variable pixel_color to 0, which goes into the VGA_framebuffer module and writes into the pixels in black, essentially "erasing" the line that was drawn when pixel_color was set to 1, which draws pixels in white. Immediately, I think of a system that can determine when a line is done drawing to erase it then update the coordinates to draw a new line. The type of animation I chose to do was a line that goes from edge to edge, spinning in circles. It will start from (0,0) – (640,480) and turn clockwise towards (640,0) – (0,480), Then, it will continue to spin clockwise back to (0,0) – (640,480). I decided to go with this animation because when the end points of the line are on the opposite edges, you only need to update either the x or the y values.

While I thought this would be simple, it was very much the opposite. Immediately, there were flaws in my idea. First, I needed to find a way for the system to know when an individual line was done drawing. To do this, I added a new output variable to the line_drawer module called "done". "done" indicates that the line is done drawing, and now the system can either redraw it in black to erase it, or if it erased it and the line is "done" again, it will increment either the x values or y values and draw a new line in white with the new coordinates. Additionally, when the line is "done", line_drawer will no longer write into any pixels and needs to be reset to start again. To solve this issue, I created a variable in my line_controller module called "start". This is an output logic from line_controller that connects to the "reset" port in line_drawer. This way, if I make "start" equal to 1, it will reset line_drawer. Every time "done" becomes true, I make "start" true to redraw the line in either white or black with the same or a new pair of coordinates. This is the essence of my animation, but the most complicated part of my animation happens when the animation makes a full circle. When the animation starts, (x0, y0) = (0, 0) and (x1, y1) = (640, 480). However, after a full circle without any changes, (x0, y0) = (640, 480) and (x1, y1) = (0,0). This is an issue since I cannot increment the same values or it will go past the boundaries of the VGA display. To tackle this issue, I made it so that the coordinates flip at this point. As soon as (x0, y0) = (640, 480), it will flip with x1 and y1 and go back to (0,0). Now, it can increment as usual and continue the cycle, completing my animation for task 2. To visualize my explanation through code, refer to Appendix 2.C.

Top-Level Diagrams

Task 1 Top-Level Diagram:

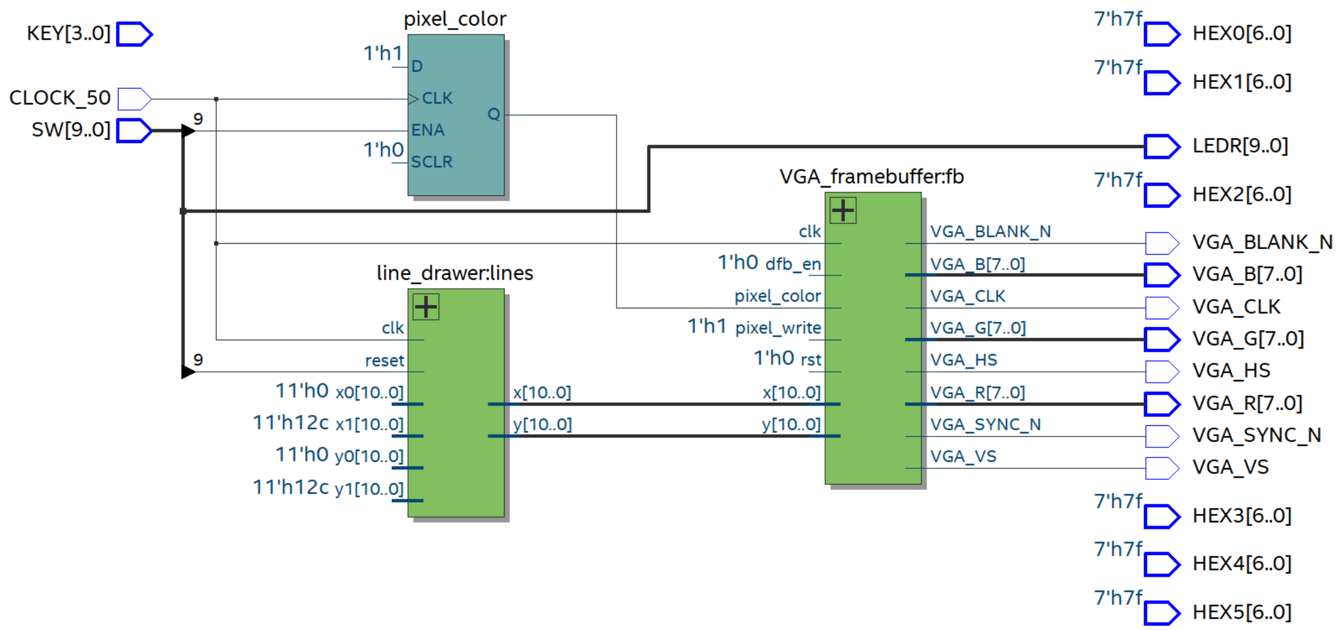

Figure 7. Top-Level Diagram for Task 1

Task 2 Top-Level Diagram:

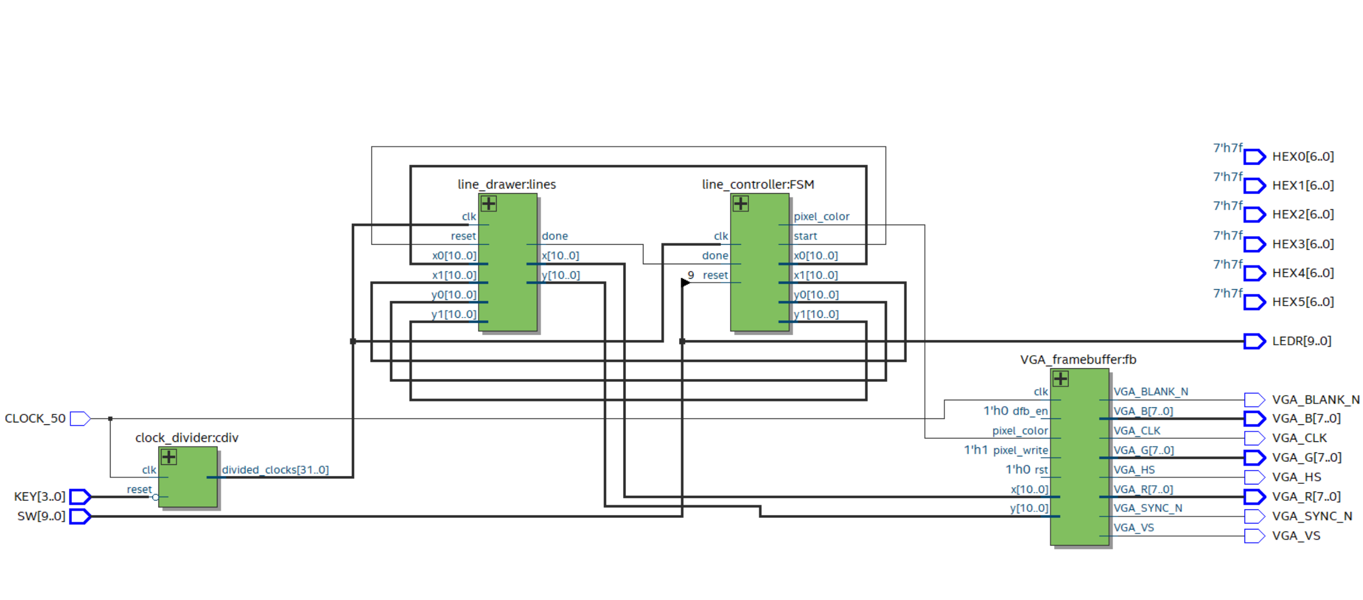

Figure 8. Top-Level Diagram for Task 2

Results

Task #1

For the first task, I created two testbenches. One for line_drawer and one for DE1_SoC. It was said that a testbench for the VGA_framebuffer was not necessary.

My line_drawer_testbench for task #1 tests a few things. The primary cases for drawing a line would be drawing a line with a positive slope, drawing a line with a negative slope, drawing a horizontal line, and drawing a vertical line. These tests were successful as shown in Figure 9. As seen, the values of the output coordinates to write into stop at the coordinates they need to stop at. For my negative slope test, I even changed the (x0, y0) to be to the right of (x1, y1) to show that no matter what coordinates you put in, even if the order is seemingly flipped, the line will output correctly.

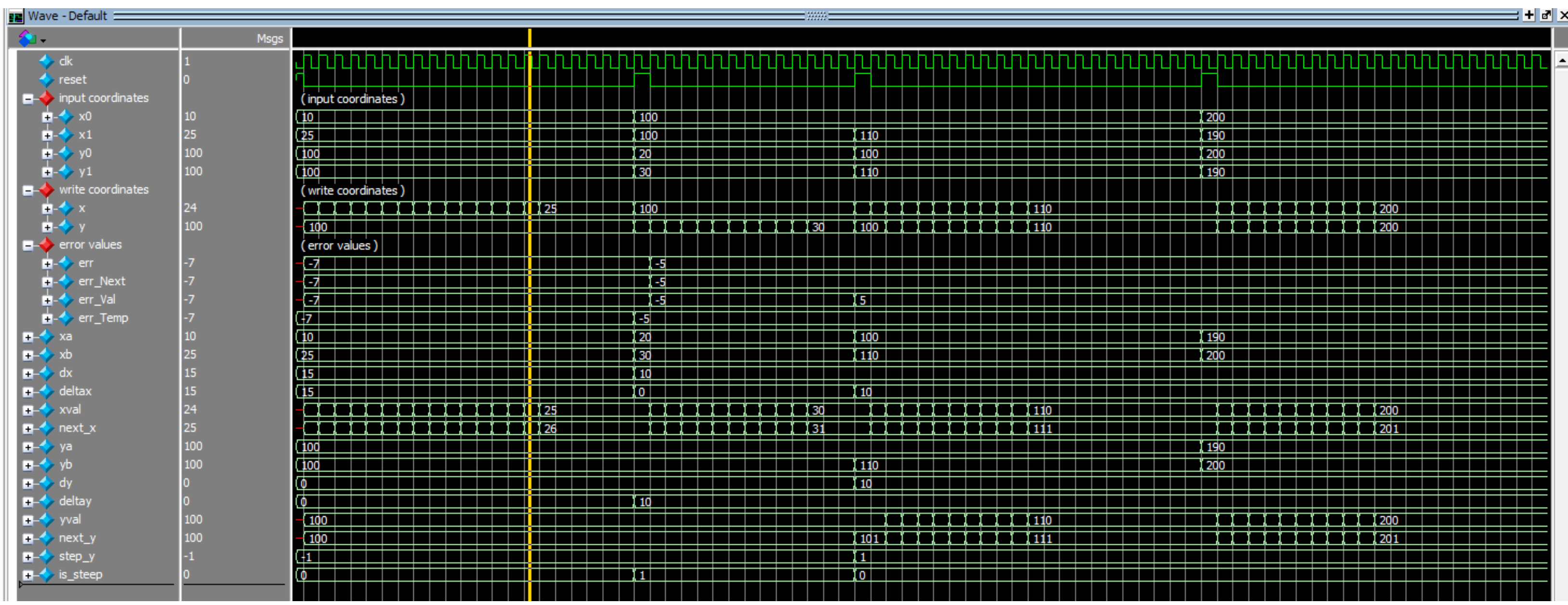

Figure 9. ModelSim for line_drawer.sv in task 1

My DE1_SoC_testbench was used to see if the VGA_framebuffer and the output of line_drawer connected properly. It was also used to see a more overall outlook of the line and its outputs. For DE1_SoC_testbench, I tested a slope magnitude of 1, a steeper slope, and a not steep slope. All of the tests were successful, as shown in Figure 10.

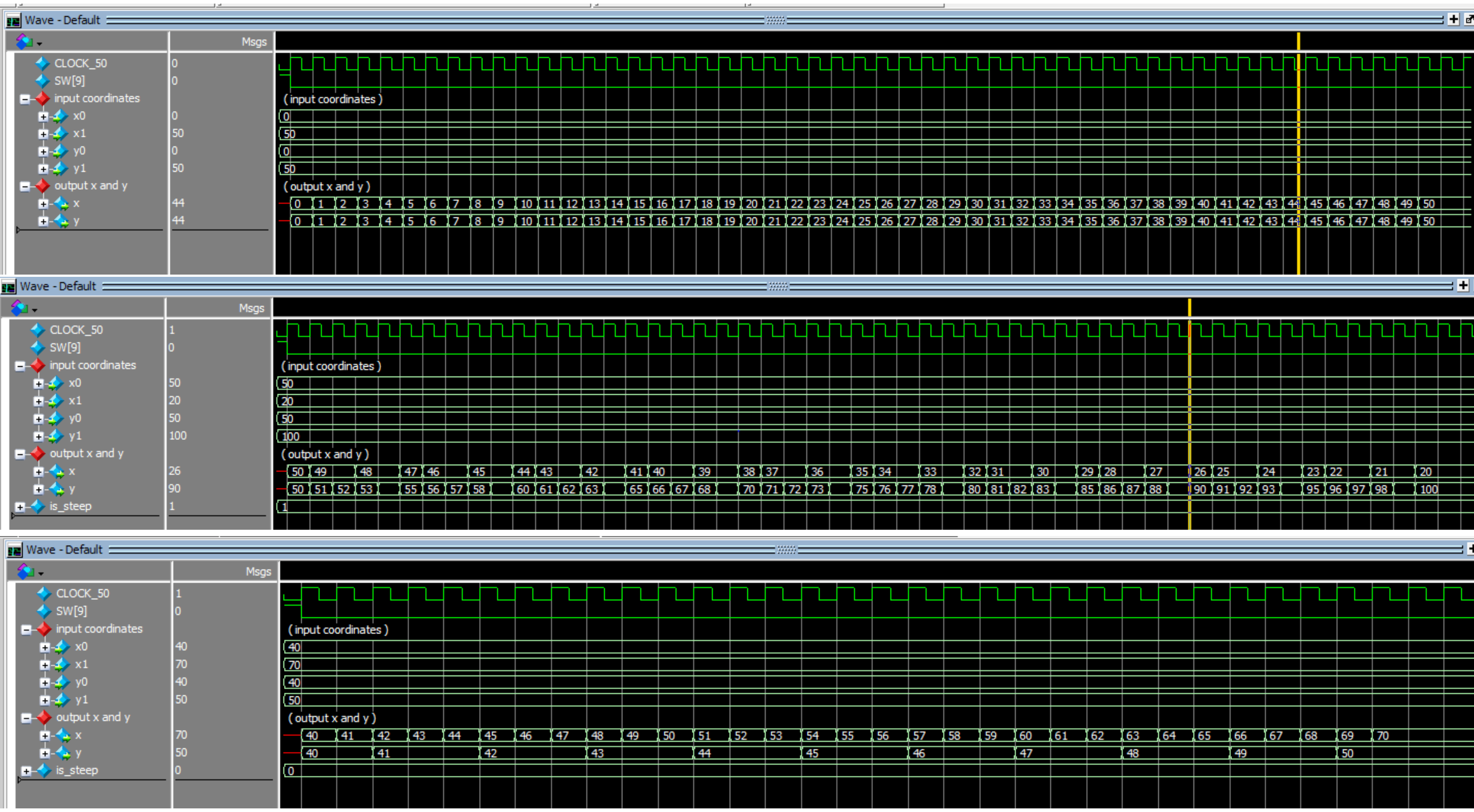

Figure 10. ModelSim simulation for DE1_SoC testbench in task 1. Top image is full waveform, middle image shows a steep slope example, and the bottom image shows a non-steep slope example.

Task #2

For task 2 I made three test benches. One for line_drawer (with a new variable), one for line_controller, and one for DE1_SoC. Similar to task 1, I did not make a testbench for VGA_framebuffer since it was said to be unnecessary.

The line_drawer_testbench for task 2 tests the same cases as task 1. The primary cases for drawing a line would be drawing a line with a positive slope, drawing a line with a negative slope, drawing a horizontal line, and drawing a vertical line. These tests were successful as shown in Figure 11. As seen, the values of the output coordinates to write into stop at the coordinates they need to stop at. For my negative slope test, I even changed the (x0, y0) to be to the right of (x1, y1) to show that no matter what coordinates you put in, even if the order is seemingly flipped, the line will output correctly. Additionally, the new output variable "done" is true when the line is done drawing and false while it is still drawing.

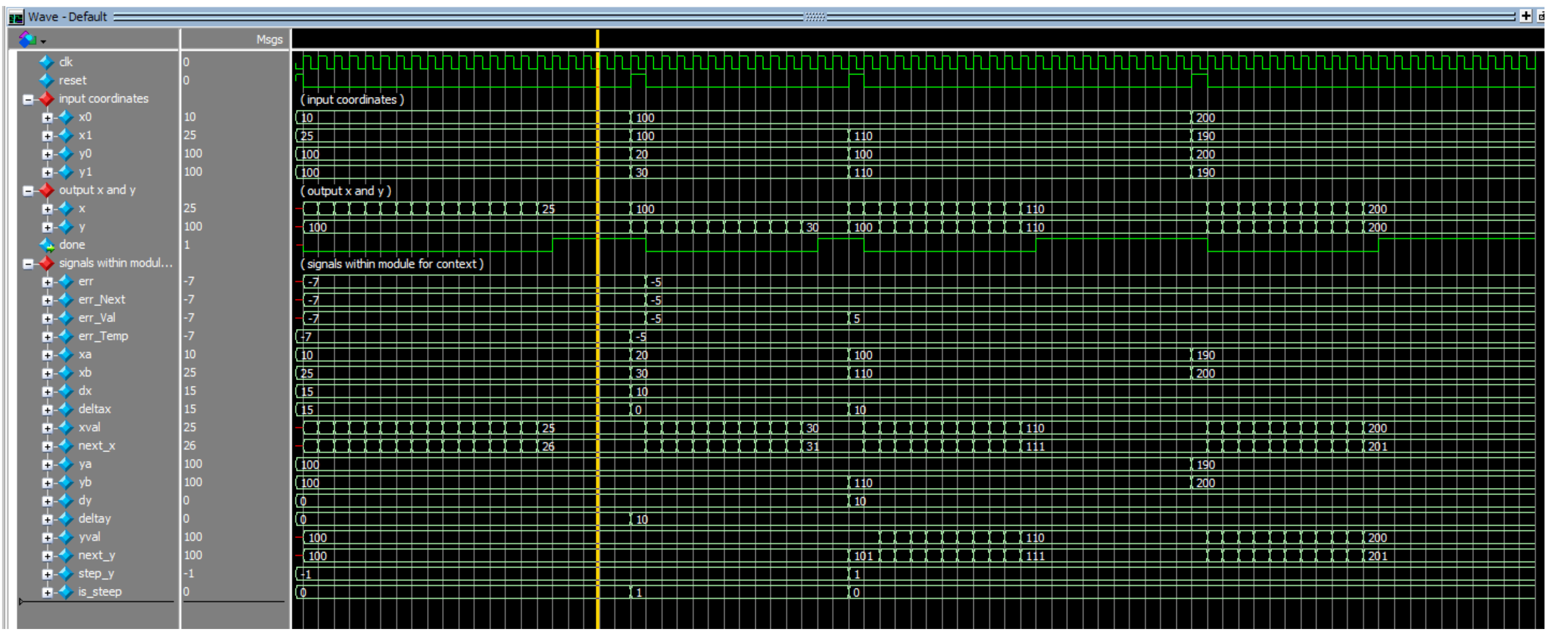

Figure 11. ModelSim simulation for line_drawer in task 2

The line_controller_testbench was used to test my system for the animation. Since the system works on its own, the only thing I can do was to toggle reset. Thus, after toggling reset I had to see if it changed from moving horizontally to vertically, then once it made a full cycle if it flipped the coordinates and moved horizontally again. These cases were successful as shown in Figure 12.

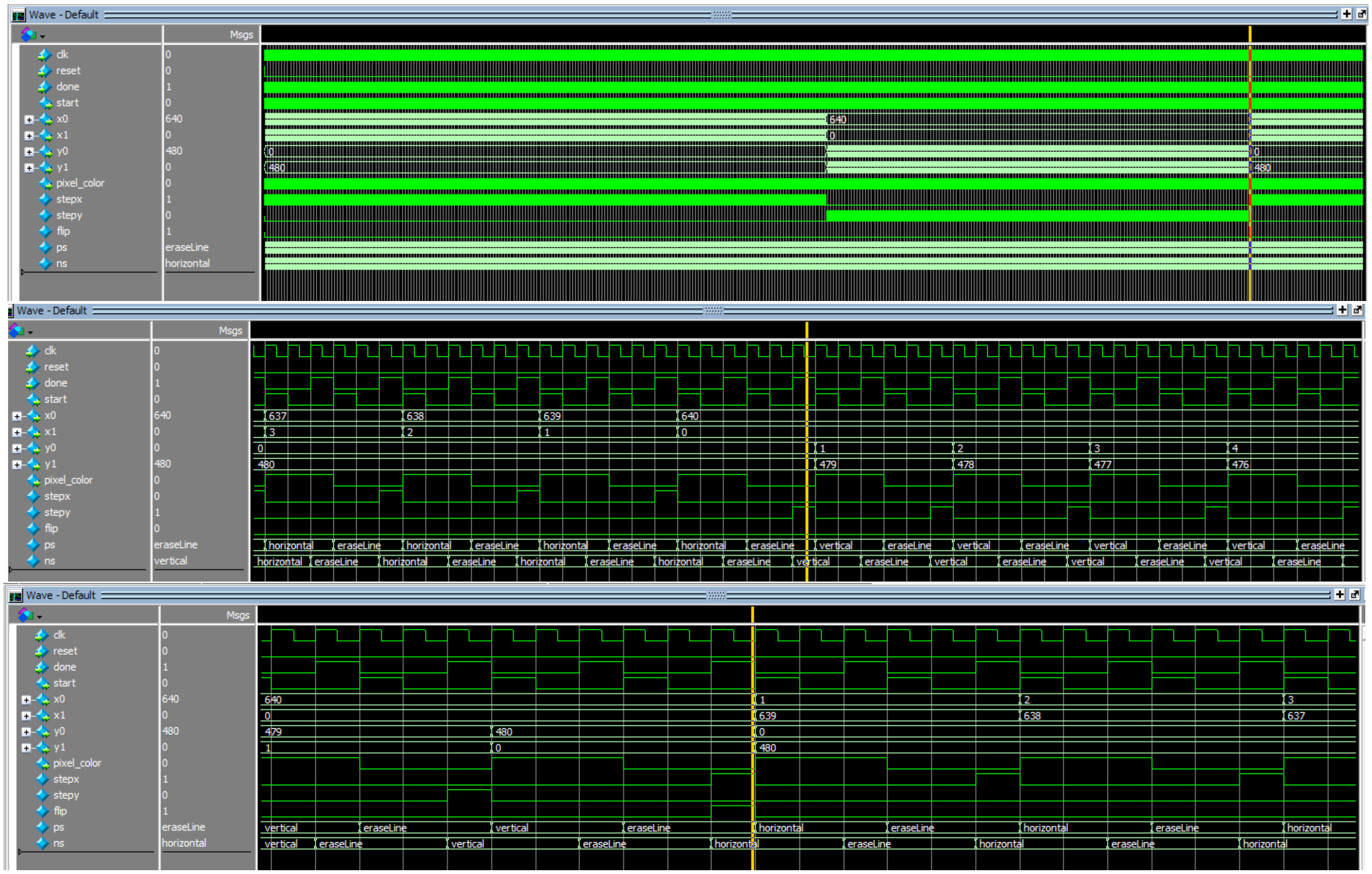

Figure 12. ModelSim simulation for line_controller. Top image is the full simulation, 4000 cycles. Middle image is the zoom in of the transition between the last erase from horizontal to vertical. The bottom image is the shift from the last erase from vertical to horizontal, where the coordinates need to flip.

The DE1_SoC testbench was rather large. In line_controller, I was able to input “done” which made cycling through the coordinates much faster. However, in DE1_SoC, I must let the system do its own work. Because my computer can handle it, I made a testbench that runs 2 million clock cycles to see if the behavior works as intended. The number is so large because it needs to write one pixel per clock cycle, and it takes twice as long since I need to erase the line every time I write it. Thankfully, my ModelSim for task 2 DE1_SoC was successful, as shown in Figure 13.

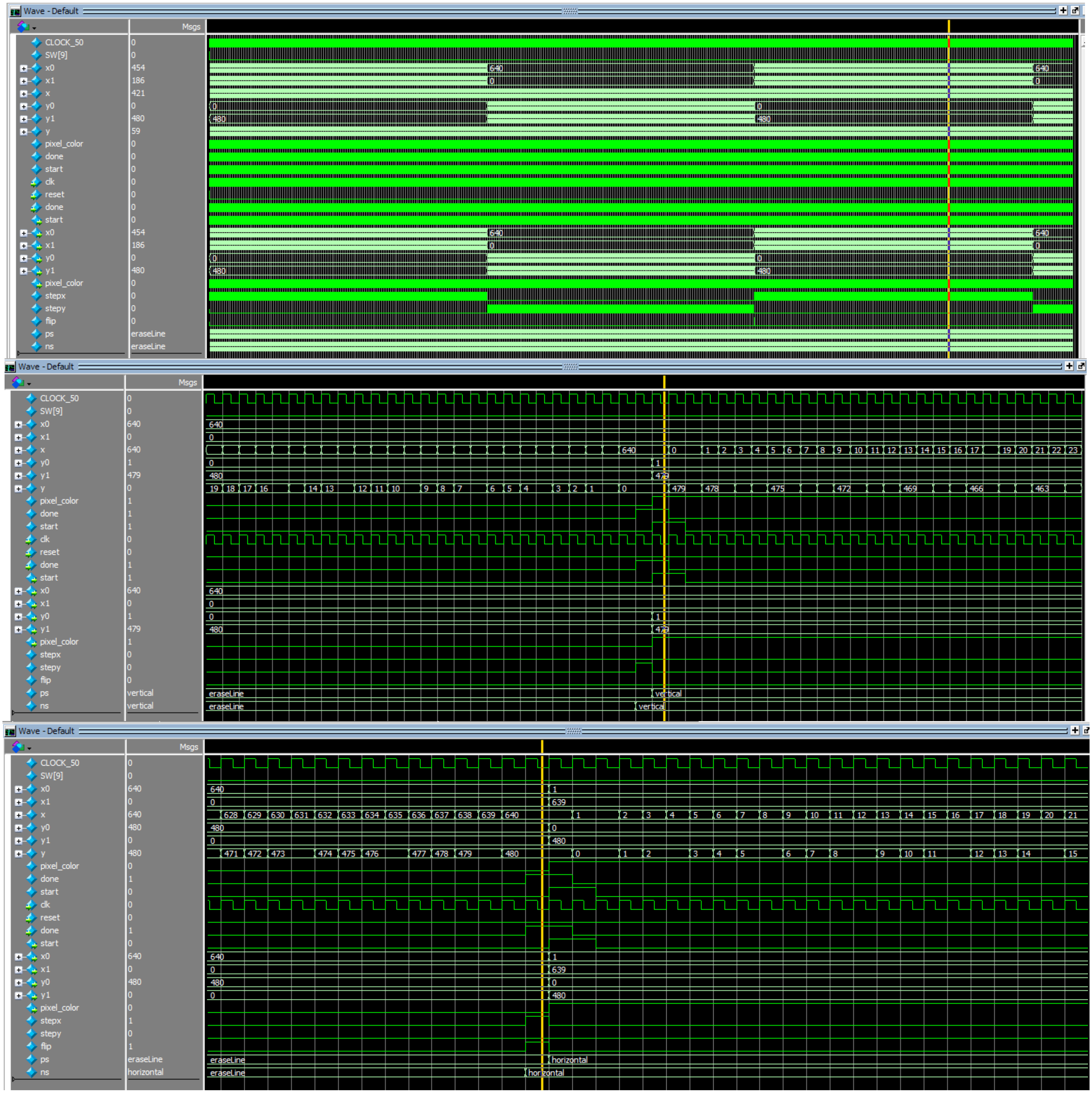

Figure 13. ModelSim for DE1_SoC in task 2. Top image is the overall image of 2 million clock cycles through my animation. Middle image is a close up of the transition between the last erase in horizontal to vertical. The bottom image is a close up of the transition between the last erase in vertical and horizontal where the coordinates need to flip.

Summary Conclusion

The main goal of this lab was to learn how to interact with the VGA display port on the FPGA. These skills were completely new to me and I had no idea the FPGA was capable of such thing. I learned how to draw onto the VGA display pixel-by-pixel, and I learned how to draw a line using Bresenham's algorithm. From there, I learned how to change the colors of the pixels in order to create an animation drawing lines. This lab was extremely challenging to me and I learned a lot of new functionalities of the FPGA board.

Overall, my lab provided the results I wanted and I believe it is sufficient in covering the requirements of this lab. The special cases were covered for and the primary functions work perfectly.

Appendix

0.A) VGA_framebuffer.sv (code) (used in all tasks)

```
// VGA driver: provides I/O timing and double-buffering for the VGA port.

module VGA_framebuffer(
    input logic clk, rst,
    input logic [10:0] x, // The x coordinate to write to the buffer.
    input logic [10:0] 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 #1 Modules

1.A) line_drawer.sv (code)

```
// Brian Dallaire
// 05/07/2021
// EE 371
// Lab 3 Task 1

// line_drawer takes the 1-bit inputs clk and reset and 11-bit inputs x0, y0, x1, and y1 and
// outputs the 11-bit outputs x and y. This module is used to draw a line in a VGA display
// using the x0, y0, x1, and y0 inputs as (x,y) coordinates. By having two coordinates in
// different locations, a line will be drawn. This is done by first calculating the change
// in x coordinates and y coordinates. Then, it will be determined if the slope is "steep"
// to see if the dx and dy values need to be flipped. Once this is done, the error will be
// calculated to determine the offset in pixels when drawing the line. Lastly, after the
// error is calculated, the output x and y values will be updated accordingly. By repeating
// this action starting from x0 y0, x and y reach the values of x1 and y1 and the line will
// stop drawing.

module line_drawer(
    input logic clk, reset,

    // x and y coordinates for the start and end points of the line
    input logic [10:0]   x0, x1,
    input logic [10:0] y0, y1,

    //outputs cooresponding to the coordinate pair (x, y)
    output logic [10:0] x,
    output logic [10:0] y
    );

    /*
     * You'll need to create some registers to keep track of things
     * such as error and direction
     * Example: */
    logic signed [11:0] err, err_Next, err_Val, err_Temp;
    logic [10:0] xa, xb, dx, deltax, xval, next_x;
    logic [10:0] ya, yb, dy, deltay, yval, next_y, step_y;
    logic [10:0] is_steep;

   // this always_comb block is the core of the functionality of this module. Here there are
   // five major components. The first component is determining dx and dy. First, change in x
   // and change in y are determined. If change in y is greater than change in x, the slope is
   // "steep". If the slope is "steep" dx is assigned change in y and dy is assigned change in x.
   // Otherwise, dx is change in x and dy is change in y. The next component simply assigns the first
   // and last values of x and y and can be change depending on conditions such as if "steep" is true
   // or x1>x0, etc. The third component is calculating the error. Since there are a limited number of
   // pixels, the error determines how the pixels are distributed to form the straightest line. Thus,
   // in the fourth component, the increments for updating the outputs x and y are determined using
   // the error from the third component. Lastly, in the fifth component, the outputs x and y are
   // updated using the combination of every component so far.
   always_comb begin
      deltax   = (x1 > x0) ? (x1 - x0) : (x0 - x1);
      deltay   = (y1 > y0) ? (y1 - y0) : (y0 - y1);
      is_steep = (deltay > deltax);
      dx = !(is_steep) ? deltax : deltay;
      dy = !(is_steep) ? deltay : deltax;

      xa = (!is_steep & (x1<x0)) ? x1 : (!is_steep & (x1>x0)) ? x0 : (is_steep & (y1<y0)) ? y1 : y0;
      xb = (!is_steep & (x1<x0)) ? x0 : (!is_steep & (x1>x0)) ? x1 : (is_steep & (y1<y0)) ? y0 : y1;
      ya = (!is_steep & (x1<x0)) ? y1 : (!is_steep & (x1>x0)) ? y0 : (is_steep & (y1<y0)) ? x1 : x0;
      yb = (!is_steep & (x1<x0)) ? y0 : (!is_steep & (x1>x0)) ? y1 : (is_steep & (y1<y0)) ? x0 : x1;

      err_Temp = -(dx/2);
      err_Val = err + dy;
      err_Next = (err_Val >= 0) ? err_Val - dx : err_Val;

      step_y = (ya < yb) ? 1 : -1;

      next_x = xval + 1'b1;
      next_y = (err_Val >= 0) ? yval + step_y : yval;

      x = !(is_steep) ? xval : yval;
      y = !(is_steep) ? yval : xval;
   end

   // this always_comb block is what updates the line every clock cycle. xval, yval, and err are
   // being updated depending on the condition of the line. If reset is true, xval and yval are
   // assigned to be the first x value and the first y value. It is set this way since depending
   // on the coordinates given it is not simply xval <= x0 and yval <= y0. Next, if xval and yval
   // equal the second x value and second y value, then xval, yval, and err remain the same. This
   // indicates the line is done. If xval and yval dont equal their respective second values,
   // xval, yval, and err are updated to be variables calculated in the always_comb block. This way,
   // xval, yval, and err are incremented or decremented accordingly.
   always_ff @(posedge clk) begin
      if(reset) begin
         xval <= xa;
         yval <= ya;
         err  <= err_Temp;
      end else begin
         if(((xb == xval) && (yb == yval))) begin
            xval <= xval;
            yval <= yval;
            err  <= err;
         end else begin
            xval <= next_x;
            yval <= next_y;
            err  <= err_Next;
         end
      end
   end
endmodule
```

1.B) line_drawer.sv (testbench)

```
// line_drawer_testbench tests the expected and unexpected cases for this module. For this simulation,
// I tested what happens when the line is drawn diagonally, horizontally, and vertically. I also tested
// weird cases such as when x0 y0 and x1 y1 are making a "negative" slope. I went one step further and
// tested to see if flipping the coordinates will screw up the values.

module line_drawer_testbench();
   logic clk, reset;

   // x and y coordinates for the start and end points of the line
   logic [10:0]   x0, x1;
   logic [10:0] y0, y1;

   //outputs cooresponding to the coordinate pair (x, y)
   logic [10:0] x;
   logic [10:0] y;

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

   parameter clock_period = 100;

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

   initial begin
      x0 <= 11'd10; x1 <= 11'd25;
      y0 <= 11'd100; y1 <= 11'd100;
      reset <= 1;                                   @(posedge clk);
      reset <= 0;                                   @(posedge clk);
      repeat(20)                                    @(posedge clk);
      x0 <= 11'd100; x1 <= 11'd100;
      y0 <= 11'd20; y1 <= 11'd30;
      reset <= 1;                                   @(posedge clk);
      reset <= 0;                                   @(posedge clk);
      repeat(12)                                    @(posedge clk);
      x0 <= 11'd100; x1 <= 11'd110;
      y0 <= 11'd100; y1 <= 11'd110;
      reset <= 1;                                   @(posedge clk);
      reset <= 0;                                   @(posedge clk);
      repeat(20)                                    @(posedge clk);
      x0 <= 11'd200; x1 <= 11'd190;
      y0 <= 11'd200; y1 <= 11'd190;
      reset <= 1;                                   @(posedge clk);
      reset <= 0;                                   @(posedge clk);
      repeat(20)                                    @(posedge clk);
      $stop;
   end

endmodule
```

1.C) DE1_SoC.sv (code)

```
// Brian Dallaire
// 05/07/2021
// EE 371
// Lab 3 Task 1

// DE1_SoC takes the 1-bit input CLOCK_50, 4-bit input KEY, 10-bit input SW, and outputs
// 6, 7-bit outputs HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, 10-bit output LEDR, 8-bit outputs
// VGA_R, VGA_G, VGA_B, outputs VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, and VGA_VS. The
// main purpose of this module is to instantiate all of the modules in this project and
// connect them together to output the desired outputs onto the FGPA. DE1_SoC for task 1
// connects line_drawer to VGA_framebuffer to draw lines onto the VGA display using the
// given coordinates for x0, y0, x1, and y1.

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 [10:0] x0, x1, x;
   logic [10:0] y0, y1, y;
   logic frame_start;
   logic pixel_color;


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


   // VGA_framebuffer takes the 1-bit inputs clk, rst, pixel_color, pixel_write, and dfb_en and
   // the 10-bit inputs x and y and outputs the 1-bit output frame_start, 8-bit outputs VGA_R,
   // VGA_G, and VGA_B, and outputs the 1-bit outputs VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_N, and
   // VGA_SYNC_N. The main purpose of this module is to provide the timing and double_buffering
   // to the VGA port. For this lab and task 1, this module is used to provide I/O timing and
   // displaying the line onto the VGA display. The double buffering is not enabled in this lab.
   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);

   // line_drawer takes the 1-bit inputs clk and reset and the 11-bit inputs x0, y0, x1, and y1
   // and outputs the 11-bit outputs x and y. The main purpose of this module is to draw a line
   // between (x0, y0) and (x1, y1). To do this, the outputs x and y determine what pixel will be
   // drawn and line_drawer will output a different x and y pair many times to draw the line
   // one-pixel at a time. The outputs x and y for this module are just values, and the drawing
   // part is done through VGA_framebuffer
   line_drawer lines (.clk(CLOCK_50), .reset(SW[9]),
            .x0, .y0, .x1, .y1, .x, .y);

   // draw a line using this coordinates
   assign x0 = 40;
   assign y0 = 40;
   assign x1 = 70;
   assign y1 = 50;

   //assign pixel_color = 1'b1; // for testbench, comment out 3 lines below
   initial pixel_color = 1'b0;

   // this always_ff block is combined with the initial statement and is used to change the color
   // of the pixel from black to white after a reset. This is to prevent any unwanted lines to be
   // drawn in white before a reset is done
   always_ff @(posedge CLOCK_50) begin
      if (SW[9]) pixel_color <= 1'b1;
   end
endmodule
```

1.D) DE1_SoC.sv (testbench)

```
// DE1_SoC_testbench tests the expected and unexpected cases for this module. For this test bench,
// the assigned values for x0, y0, x1, and y1 need to be manually changed from the DE1_SoC code.
// thus, I simply reset once and let the clock cyles run until the desired output is there.

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);

   parameter clock_period = 100;

   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin
      SW[9] <= 1; @(posedge CLOCK_50);
      SW[9] <= 0; repeat(100) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

Task #2 Modules

2.A) line_drawer.sv (code)

```
// Brian Dallaire
// 05/07/2021
// EE 371
// Lab 3 Task 2

// line_drawer takes the 1-bit inputs clk and reset and 11-bit inputs x0, y0, x1, and y1 and
// outputs the 11-bit outputs x and y and 1-bit output done. This module is used to draw a
// line in a VGA display using the x0, y0, x1, and y0 inputs as (x,y) coordinates. By having
// two coordinates in different locations, a line will be drawn. This is done by first calculating
// the change in x coordinates and y coordinates. Then, it will be determined if the slope is
// "steep" to see if the dx and dy values need to be flipped. Once this is done, the error will
// be calculated to determine the offset in pixels when drawing the line. Lastly, after the error
// is calculated, the output x and y values will be updated accordingly. By repeating this action
// starting from x0 y0, x and y reach the values of x1 and y1 and the line will stop drawing.

module line_drawer(
   input logic clk, reset,

   // x and y coordinates for the start and end points of the line
   input logic [10:0]   x0, x1,
   input logic [10:0] y0, y1,

   //outputs cooresponding to the coordinate pair (x, y)
   output logic [10:0] x,
   output logic [10:0] y,
   output logic done
   );


   logic signed [11:0] err, err_Next, err_Val, err_Temp;
   logic [10:0] xa, xb, dx, deltax, xval, next_x;
   logic [10:0] ya, yb, dy, deltay, yval, next_y, step_y;
   logic [10:0] is_steep;

   // this always_comb block is the core of the functionality of this module. Here there are
   // five major components. The first component is determining dx and dy. First, change in x
   // and change in y are determined. If change in y is greater than change in x, the slope is
   // "steep". If the slope is "steep" dx is assigned change in y and dy is assigned change in x.
   // Otherwise, dx is change in x and dy is change in y. The next component simply assigns the first
   // and last values of x and y and can be change depending on conditions such as if "steep" is true
   // or x1>x0, etc. The third component is calculating the error. Since there are a limited number of
   // pixels, the error determines how the pixels are distributed to form the straightest line. Thus,
   // in the fourth component, the increments for updating the outputs x and y are determined using
   // the error from the third component. Lastly, in the fifth component, the outputs x and y are
   // updated using the combination of every component so far.
   always_comb begin
      deltax   = (x1 > x0) ? (x1 - x0) : (x0 - x1);
      deltay   = (y1 > y0) ? (y1 - y0) : (y0 - y1);
      is_steep = (deltay > deltax);
      dx = !(is_steep) ? deltax : deltay;
      dy = !(is_steep) ? deltay : deltax;

      xa = (!is_steep & (x1<x0)) ? x1 : (!is_steep & (x1>x0)) ? x0 : (is_steep & (y1<y0)) ? y1 : y0;
      xb = (!is_steep & (x1<x0)) ? x0 : (!is_steep & (x1>x0)) ? x1 : (is_steep & (y1<y0)) ? y0 : y1;
      ya = (!is_steep & (x1<x0)) ? y1 : (!is_steep & (x1>x0)) ? y0 : (is_steep & (y1<y0)) ? x1 : x0;
      yb = (!is_steep & (x1<x0)) ? y0 : (!is_steep & (x1>x0)) ? y1 : (is_steep & (y1<y0)) ? x0 : x1;

      err_Temp = -(dx/2);
      err_Val = err + dy;
      err_Next = (err_Val >= 0) ? err_Val - dx : err_Val;

      step_y = (ya < yb) ? 1 : -1;

      next_x = xval + 1'b1;
      next_y = (err_Val >= 0) ? yval + step_y : yval;

      x = !(is_steep) ? xval : yval;
      y = !(is_steep) ? yval : xval;
   end

   // this always_comb block is what updates the line every clock cycle. xval, yval, and err are
   // being updated depending on the condition of the line. If reset is true, xval and yval are
   // assigned to be the first x value and the first y value. It is set this way since depending
   // on the coordinates given it is not simply xval <= x0 and yval <= y0. Next, if xval and yval
   // equal the second x value and second y value, then xval, yval, and err remain the same. This
   // indicates the line is done. If xval and yval dont equal their respective second values,
   // xval, yval, and err are updated to be variables calculated in the always_comb block. This way,
   // xval, yval, and err are incremented or decremented accordingly. Additionally, "done" is true
   // when xb == xval and yb == yval. This indicates the line has finished drawing, or reached the end
   always_ff @(posedge clk) begin
      if(reset) begin
         xval <= xa;
         yval <= ya;
         err  <= err_Temp;
         done <= 0;
      end else begin
         if(((xb == xval) && (yb == yval))) begin
            xval <= xval;
            yval <= yval;
            err  <= err;
            done <= 1;
         end else begin
            xval <= next_x;
            yval <= next_y;
            err  <= err_Next;
            done <= 0;
         end
      end
   end
endmodule
```

2.B) line_drawer.sv (testbench)

```
// line_drawer_testbench tests the expected and unexpected cases for this module. For this simulation,
// I tested what happens when the line is drawn diagonally, horizontally, and vertically. I also tested
// weird cases such as when x0 y0 and x1 y1 are making a "negative" slope. I went one step further and
// tested to see if flipping the coordinates will screw up the values.
module line_drawer_testbench();
   logic clk, reset;

   // x and y coordinates for the start and end points of the line
   logic [10:0]   x0, x1;
   logic [10:0] y0, y1;

   //outputs cooresponding to the coordinate pair (x, y)
   logic [10:0] x;
   logic [10:0] y;

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

   parameter clock_period = 100;

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

   initial begin
      x0 <= 11'd10; x1 <= 11'd25;
      y0 <= 11'd100; y1 <= 11'd100;
      reset <= 1;                                       @(posedge clk);
      reset <= 0;                                       @(posedge clk);
      repeat(20)                                        @(posedge clk);
      x0 <= 11'd100; x1 <= 11'd100;
      y0 <= 11'd20; y1 <= 11'd30;
      reset <= 1;                                       @(posedge clk);
      reset <= 0;                                       @(posedge clk);
      repeat(12)                                        @(posedge clk);
      x0 <= 11'd100; x1 <= 11'd110;
      y0 <= 11'd100; y1 <= 11'd110;
      reset <= 1;                                       @(posedge clk);
      reset <= 0;                                       @(posedge clk);
      repeat(20)                                        @(posedge clk);
      x0 <= 11'd200; x1 <= 11'd190;
      y0 <= 11'd200; y1 <= 11'd190;
      reset <= 1;                                       @(posedge clk);
      reset <= 0;                                       @(posedge clk);
      repeat(20)                                        @(posedge clk);
      $stop;
   end

endmodule
```

2.C) line_controller.sv (code)

```
// Brian Dallaire
// 05/07/2021
// EE 371
// Lab 3 Task 2

// line_controller takes the 1-bit inputs clk, reset, and done and outputs the 1-bit output start,
// the 11-bit outputs x0, x1, y0, and y1, and the 1-bit output pixel_color. The main purpose of this
// module is to send to the VGA display and line_drawer the initial and final coordinates and the
// pixel color. Addtionally, it will reset line_drawer using the "start" variable. Essentially,
// this module acts as a system that automatically goes through many coordinates to draw and erase
// many lines to form an animation. The goal is to form an animation where the line stretched out
// to both edges goes in circles, clockwise.

module line_controller (clk, reset, done, start, x0, x1, y0, y1, pixel_color);

   input logic clk, reset, done;
   output logic start;
   output logic [10:0] x0, x1, y0, y1;
   output logic pixel_color;

   logic stepx, stepy, flip;

   enum {eraseLine, horizontal, vertical} ps, ns;

   // this always_comb block takes the states eraseLine, horizontal, and vertical and updates specific
   // variables to achieve desired behavior. eraseLine is the core of this block, as it has the most
   // conditions and determines the most complex behaviors of the animation. When in eraseLine, pixel
   // color is 0, so the current line that was just drawn is erased. When in horizontal, the line moves
   // clockwise against the top and bottom walls of the display, only requiring x values to increment.
   // When in vertical, the line moves clockwise against the left and right walls, only requiring the
   // y values to change. Going through these states will form the desired animation behavior.
   always_comb begin
      case(ps)

         eraseLine    : begin
                           if(done & (x0 != 11'b01010000000) & !start) begin
                              pixel_color = 1'b0;
                              flip = 1'b0;
                              stepx = 1'b1;
                              stepy = 1'b0;
                              ns = horizontal;
                           end else if (done & (x0 == 11'b01010000000) & (y0 != 11'b00111100000) & !start) begin
                              pixel_color = 1'b0;
                              flip = 1'b0;
                              stepx = 1'b0;
                              stepy = 1'b1;
                              ns = vertical;
                           end else if (done & (y0 == 11'b00111100000) & (x0 == 11'b01010000000) & !start) begin
                              pixel_color = 1'b0;
                              flip = 1'b1;
                              stepx = 1'b1;
                              stepy = 1'b0;
                              ns = horizontal;
                           end else begin
                              pixel_color = 1'b0;
                              flip = 1'b0;
                              stepx = 1'b0;
                              stepy = 1'b0;
                              ns = eraseLine;
                           end
                        end

      horizontal     :  begin
                           pixel_color = 1'b1;
                           flip  = 1'b0;
                           stepx = 1'b0;
                           stepy = 1'b0;
                           if(done & !start)
                              ns = eraseLine;
                           else
                              ns = horizontal;
                        end

      vertical       :  begin
                           pixel_color = 1'b1;
                           flip  = 1'b0;
                           stepx = 1'b0;
                           stepy = 1'b0;
                           if(done & !start)
                              ns = eraseLine;
                           else
                              ns = vertical;
                        end

      endcase
   end

   // this always_ff block is used to increment multiple values and change the present state. When reset,
   // the coordinates will update to the original position. When the animation makes a full cirlce,
   // the values for (x0, y0) and (x1, y1) need to be flipped to keep the system simple. Otherwise,
   // every time a line is done, it will increment the coordinates accordingly. Whenever done or reset is
   // true, start will be true. This will reset the line_drawer module. If reset and done are not true,
   // the coordinates remain the same and line_drawer will continue to behave as normal.
   always_ff @(posedge clk) begin
      if(reset) begin
         ps <= eraseLine;
         x0 <= 11'b00000000001; // 1;
         y0 <= 11'b00000000000; // 0;
         x1 <= 11'b01001111111; // 639
         y1 <= 11'b00111100000; // 480
         start <= 1;
      end else begin
         if(flip & done) begin
            x0 <= 11'b00000000001;
            y0 <= 11'b00000000000;
            x1 <= 11'b01001111111;
            y1 <= 11'b00111100000;
            start <= 1;
         end else if (done) begin
            x0 <= x0 + stepx;
            y0 <= y0 + stepy;
            x1 <= x1 - stepx;
            y1 <= y1 - stepy;
            start <= 1;
         end else begin
            x0 <= x0;
            y0 <= y0;
            x1 <= x1;
            y1 <= y1;
            start <= 0;
         end

         ps <= ns;

      end
   end
endmodule
```

2.D) line_controller.sv (testbench)

```

// line_controller_testbench tests the expected and unexpected cases for this module and if the system
// cycles through properly. Because ther are no inputs other than reset and done that can be toggled
// manually, I simply toggled reset once, then toggled done 4000 times to make sure the animation reaches
// full circle. The cases to look for were when the state transitioned from horizontal to vertical, then
// vertical back to horizontal when flip is true.
module line_controller_testbench();
   logic clk, reset, done;
   logic start;
   logic [10:0] x0, x1, y0, y1;
   logic pixel_color;

   line_controller dut(.clk, .reset, .done, .start, .x0, .x1, .y0, .y1, .pixel_color);

   parameter clock_period = 100;

   initial begin
      clk <= 0;
      forever #(clock_period /2) clk <= ~clk;
   end
   initial begin
      reset <= 1; done <= 0; @(posedge clk);
      repeat(4000) begin
      reset <= 0; done <= 0; repeat(2)@(posedge clk);
      reset <= 0; done <= 1; @(posedge clk);
      end
      reset <= 1; @(posedge clk);
      reset <= 0; repeat(20)@(posedge clk);
      $stop;
   end
endmodule

```

2.E) clock_divider.sv (code and testbench)

```
// Brian Dallaire
// 05/07/2021
// EE 371
// Lab 3 Task 2

// clock_divider takes the 1-bit inputs clk and reset and outputs the 32 bit output
// divided_clocks. The purpose of this module is to slow down the clock that is inputted
// by a specific amount. If the user calls this module and uses divided_clocks[25] for example,
// the outputted clock frequency would be 0.75Hz instead of the 50MHz clock from CLOCK_50.

/* divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ...
  [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ... */
module clock_divider (clk, reset, divided_clocks);
 input logic clk, reset;
 output logic [31:0] divided_clocks = 0;

 // always_ff block that increments divided_clocks by 1. Once divided clocks is full,
 // it will output as one clock cycle. Thus, the bigger the array the slower the clock
 always_ff @(posedge clk) begin
   divided_clocks <= divided_clocks + 1;
 end

endmodule

// clock_divider_testbench tests the functionality of divided_clocks. It showcases how
// every increment divided clocks becomes closer to full.

module clock_divider_testbench();
   logic clk, reset;
   logic [31:0] divided_clocks = 0;

   clock_divider dut(.clk, .reset, .divided_clocks);

   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
   end

   initial begin
      reset <= 1;             @(posedge clk);
      reset <= 0; repeat(2000)@(posedge clk);
      $stop;
   end
endmodule
```

2.F) DE1_SoC.sv (code)

```
// Brian Dallaire
// 05/07/2021
// EE 371
// Lab 3 Task 2

// DE1_SoC takes the 1-bit input CLOCK_50, 4-bit input KEY, 10-bit input SW, and outputs
// 6, 7-bit outputs HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, 10-bit output LEDR, 8-bit outputs
// VGA_R, VGA_G, VGA_B, outputs VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, and VGA_VS. The
// main purpose of this module is to instantiate all of the modules in this project and
// connect them together to output the desired outputs onto the FGPA. DE1_SoC for task 1
// connects line_drawer to VGA_framebuffer to draw lines onto the VGA display using the
// given coordinates for x0, y0, x1, and y1.

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 [10:0] x0, x1, x;
    logic [10:0] y0, y1, y;
    logic frame_start;
    logic pixel_color;
    logic done, start;


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

    logic [31:0] div_clk;

    // clock_divider takes the 1-bit inputs clk and reset and outputs the 32 bit output
    // div_clk. The purpose of this module is to slow down CLOCK_50 for on-board demonstration.
    // Using the value div_clk[25] will make the clock 0.75 Hz instead of 50MHz like CLOCK_50
    clock_divider cdiv (.clk(CLOCK_50),
                        .reset(~KEY[0]),
                        .divided_clocks(div_clk));
    logic clkSelect; // for easy switching between clocks
    assign clkSelect = CLOCK_50; // for simulation
    //assign clkSelect = div_clk[7]; // for board


    // VGA_framebuffer takes the 1-bit inputs clk, rst, pixel_color, pixel_write, and dfb_en and
    // the 10-bit inputs x and y and outputs the 1-bit output frame_start, 8-bit outputs VGA_R,
    // VGA_G, and VGA_B, and outputs the 1-bit outputs VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_N, and
    // VGA_SYNC_N. The main purpose of this module is to provide the timing and double_buffering
    // to the VGA port. For this lab and task 1, this module is used to provide I/O timing and
    // displaying the line onto the VGA display. The double buffering is not enabled in this lab.
    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);

    // line_controller takes the 1-bit inputs clk, reset, and done and outputs the 11-bit outputs
    // x0, x1, y0, and y1 and the 1-bit outputs start and pixel_color. The main purpose of this
    // module is to update the coordinates, tell line_drawer when to draw, and change the pixel
    // color for the VGA display to create an animation effect.
    line_controller FSM (.clk(clkSelect), .reset(SW[9]), .done, .start, .x0, .x1, .y0, .y1, .pixel_color);

    // line_drawer takes the 1-bit inputs clk, reset, and start and the 11-bit inputs x0, y0, x1, and y1
    // and outputs the 11-bit outputs x and y and 1-bit output done. The main purpose of this module is
    // to draw a line between (x0, y0) and (x1, y1). To do this, the outputs x and y determine what pixel
    // will be drawn and line_drawer will output a different x and y pair many times to draw the line
    // one-pixel at a time. The outputs x and y for this module are just values, and the drawing
    // part is done through VGA_framebuffer.
    line_drawer lines (.clk(clkSelect), .reset(start),
            .x0, .y0, .x1, .y1, .x, .y, .done);

endmodule
```

2.G) DE1_SoC.sv (testbench)

```
// DE1_SoC_testbench tests the expected and unexpected situations for this task. In this simulation,
// I tested to see if after reset the input coordinates will cycle and flip after one cycle properly.
// To do this, however, I needed many cycles. Here I did 2 million cycles simply because my computer
// can handle it, but the smart thing to do would have been to parameratize my line_controller or
// simply reduce the range the line_controller operates in. That being said, the cycle did work
// and properly flips the coordinates at the right time.
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);

   parameter clock_period = 100;

   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin
      SW[9] <= 1; @(posedge CLOCK_50);
      SW[9] <= 0; repeat(2000000) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```