Jiovany Soliman, 1951338

Chloe' Miller, 2031648

EE/CSE 371 Lab 5 Report: Display Interface

November 17, 2023

Design Procedure

The purpose of this lab was to utilize the VGA capabilities of the DE1_SoC board. The VGA controller supports a screen resolution of 640 x 480 pixel, and the image that's displayed is controlled by 2 sources: a character buffer, and a pixel buffer. We were provided with an already-implemented VGA pixel buffer, and we were told not to worry about the character buffer for this lab. The main idea of the VGA display is that it's a continuous loop through each pixel of the display, horizontally in the x direction, then vertically in the y direction. For each index, the VGA driver can set the color of that pixel (black or white for this lab) if pixel_write is asserted. The main tasks for this lab were to be able to handle drawing lines with various slopes and animate a line moving on the screen. Additionally, we needed to be able to clear the screen by making it all black.

Task #1:

For task 1, we simply followed the instructions and ensured that we were able to communicate with the DE1_SoC board on LabsLand and made sure that the VGA display works. No modification of the code provided was necessary to complete this task.

Task #2:

For task 2, we started by reading the Wikipedia pages to gain some understanding of Bresenham's line algorithm. We then went through the provided pseudocode line by line and translated it into SystemVerilog code. We started our code implementation by figuring out the combinational portions which were the swapping (if x0 > x1), checking for isSteep (abs(y1-y0) > abs(x1-x0)), and calculating the deltax and deltay variables. The sequential portions were the for-loop mechanism, adjusting the error, and stepping x and y. As part of the implementation of task 2, we tested out code and drew lines from left to right, up to down, right to left, and down to up, with both steep and gradual slopes.

```
// swap (x0, y0) and (x1, y1)
always_comb begin
  if(isSteep) begin
        x0temp1 = y0;
        y0temp1 = x0;
        x1temp1 = y1;
        y1temp1 = x1;
  end else begin
        x0temp1 = x0;
        x1temp1 = x1;
        y0temp1 = y0;
        y1temp1 = y1;
  end

  // swap (x0, x1) and (y0, y1)
  if(x0temp1 > x1temp1) begin
        x0temp2 = x1temp1;
        x1temp2 = x0temp1;
        y0temp2 = y1temp1;
        y1temp2 = y0temp1;
  end else begin
        x0temp2 = x0temp1;
        x1temp2 = x1temp1;
        y0temp2 = y0temp1;
        y1temp2 = y1temp1;
  end

  // deterine if y will decrement or increment
  if(y0temp2 < y1temp2)
     y_step = 1;
  else
     y_step = -1;

end // always_comb
```

Figure 1: Snippet showing line_drawer combinational logic

The above figure shows our combinational logic for the line corrections of Bresenham's line algorithm. The first if-else block is to account for slopes that are greater than 1. This is important because if the line is steep, there will be multiple y values calculated for one x value, thus we must swap x0 with y0 and x1 with y1 to iterate over the y values instead. This allows for only one x value to be mapped to only one y value.

The second if-else blocks accounts for the scenario in which the user has passed in coordinates where the end point occurs before the start point (when x0 > x1). We simply swap x0 with x1 and y0 and y1 to ensure that the line being drawn is within bounds.

```
    // calculate detlax and deltay for the algorithm
    assign deltax = (x1temp2 - x0temp2);
    assign deltay = (y1temp2 > y0temp2) ? (y1temp2 - y0temp2) : (y0temp2 - y1temp2); // absolute value

    always_ff @(posedge clk) begin
      if(reset) begin // reset state, initialize nextX and nextY, error, and doneSignal
         nextX <= x0temp2;
         nextY <= y0temp2;
         error <= -(deltax/2);
         doneSignal <= 0;
      end else if(nextX <= (x1temp2)) begin // for x from x0 to x1 (inclusive)
         nextX <= nextX + 1;

         // swap x and y if the slope is steep (prevents multiple y's from being assigned to a single x)
         if(isSteep) begin
            x <= nextY;
            y <= nextX;
         end else begin
            x <= nextX;
            y <= nextY;
         end

         // algorithm to determine if y should be stepped or not for best approximation of the line
         if(error + deltay >= 0) begin
            nextY <= nextY + y_step;
            error <= error + deltay - deltax;
         end else begin
            nextY <= nextY;
            error <= error + deltay;
         end

         // if we reached the end point, then line is done drawing
         if((nextX == x1temp2) && (nextY == y1temp2))
            doneSignal <= 1;
      end // end (nextX <= (x1temp2)
    end // always_ff

    assign done = doneSignal;

endmodule
```

Figure 2: Snippet showing line_drawer sequential logic

The above figure shows our sequential logic for Bresenham's line drawing algorithm. We have a reset state that initializes the variables nextx and nexty, error, and done. After resetting, the algorithm determines whether x and y should be swapped based on if the slope is steep or not. Then, it uses the error variable and the calculated deltax and deltay (from the combinational logic) to determine if y needs to change by 1 or not. Lastly, the algorithm signals that it is done drawing the line when the outputs x and y have reached the specified (and adjusted) endpoints.

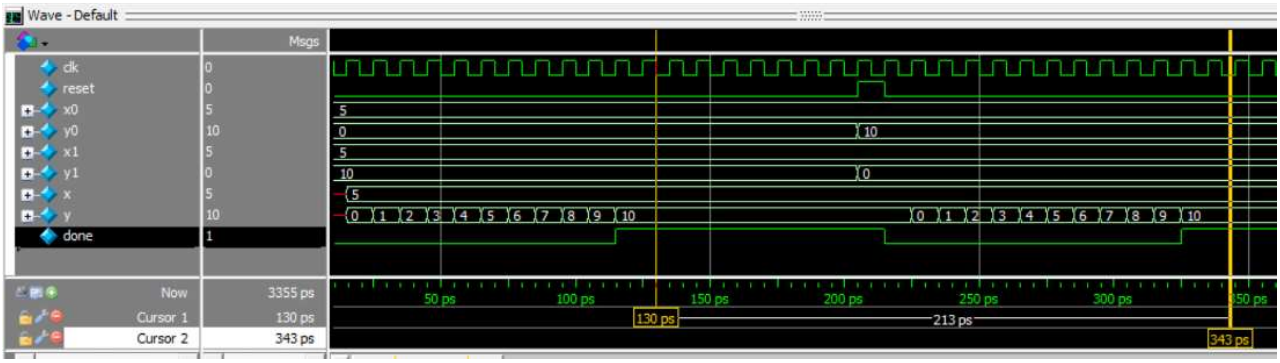


Figure 3: Snippet of ModelSim showing vertical line test @ coordinates (5, 10), (5, 0)

The above figure shows the waveform of our line drawer module after running a vertical line test, where the points (5,10) and (5,0) are swapped to ensure that we achieve the same results. Indeed, we can see from 0-130ps and from 225-343ps our code steps y from 0 to 10 while keeping x at 5, no matter the order of the points x0, y0, x1, and y1. We can also see that at 115ps and 325ps the module asserts done, indicating that the line has completed drawing.

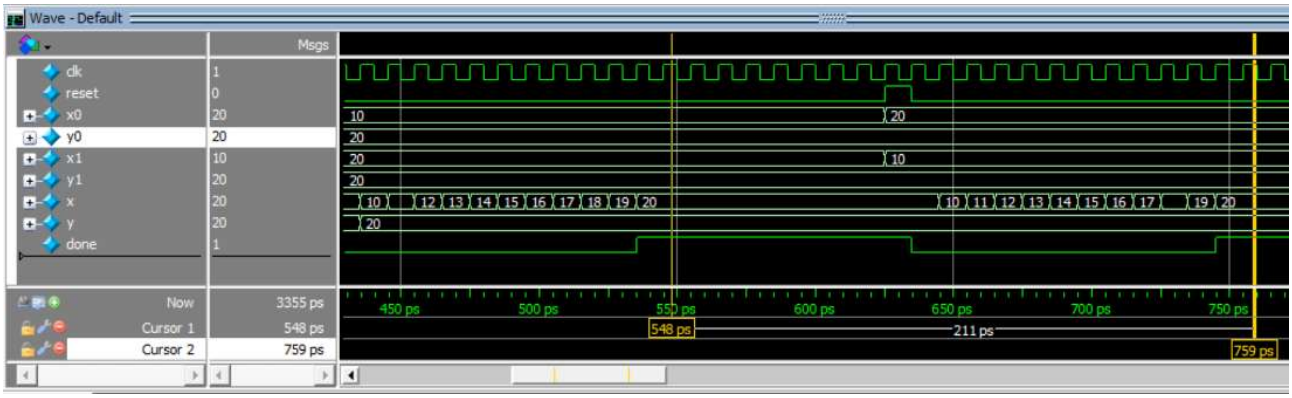


Figure 4: Snippet of ModelSim showing horizontal line test @ coordinates (10, 20), (20, 20)

The above figure shows the waveform of our line drawer module after running a horizontal line test, where the points (10,20) and (20,20) are again swapped to ensure we achieve the same results. We can see from 425-550ps and 625-775ps that our code steps x from 10 to 20 while keeping y at 20, regardless of the order of the points issued, done is also asserted at 560ps and 750ps to indicate that the line has been drawn.

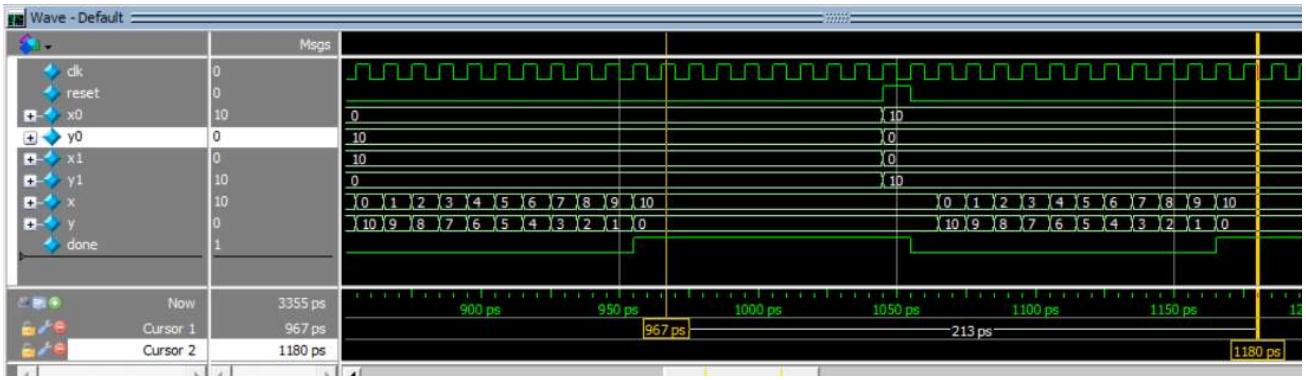


Figure 5: Snippet of ModelSim showing positive diagonal line test @ coordinates (0, 10), (10, 0)

The above waveform shows the results of our line drawer module after a positive diagonal line test with the points (0,10) and (10,0) swapped. When visualizing the graph such that the positive x axis is the top horizontal axis and the positive y axis is the left vertical axis and reading the graph from left to right, we can verify from 800-960ps and from 1050-1175ps that x increments by 1 while y decrements by 1 after each clock edge. Done is also properly asserted at 955ps and 1165ps to indicate that the line is done drawing.

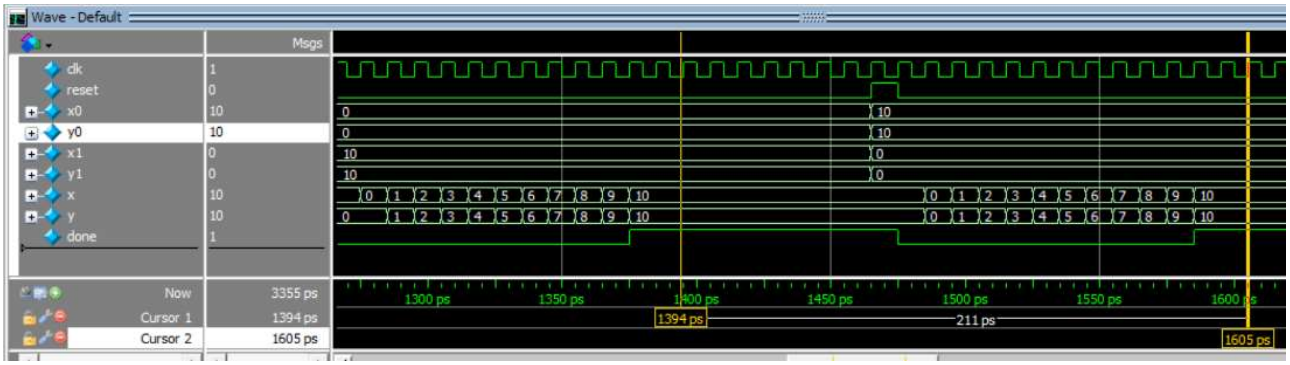


Figure 6: Snippet of ModelSim showing negative diagonal line test @ coordinates (0, 0), (10, 10)

The above waveform shows the results of our line drawer module after a negative diagonal line test with the points (0,0) and (10,10) swapped. Visualizing the graph, the same way as in the previous paragraph, we can verify that x and y both increment by 1 to produce a negative-sloped diagonal line from 1275-1375ps and 1475-1600ps, and that done is asserted properly when reaching the end point (10,10).

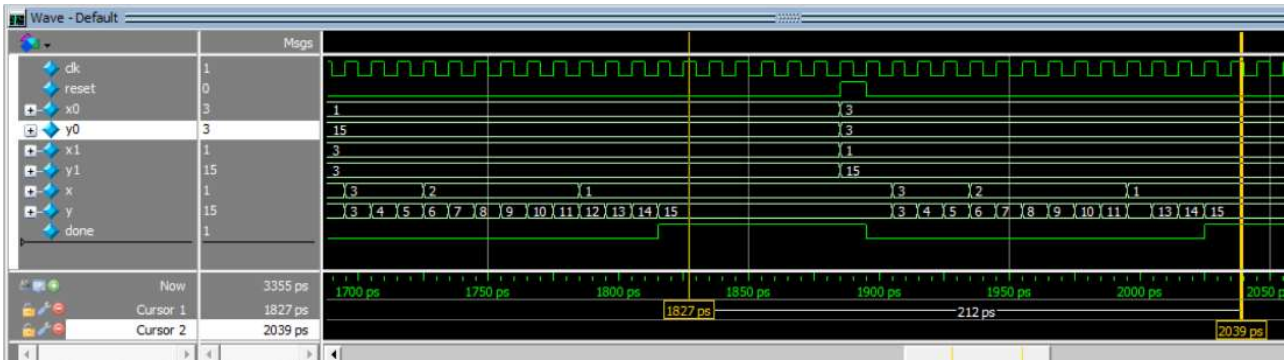


Figure 7: Snippet of ModelSim showing left-up and right down steep line test @ coordinates (1, 15), (3, 3)

The above waveform shows the results of our line drawer module after a steep line test with the points (1,15) and (3,3) swapped. Visualizing the graph as in previous paragraphs, we can see that the algorithm corrects for the steep slope (>1) and iterates over y from 3 to 15 every clock cycle while decrementing x from 3 to 1 every few clock cycles. The same results happen regardless of the order of the points, as seen in 1700-1825ps and 1900-2025ps and done is asserted once the end point (1,15) is reached.

Figure 8: Snippet of ModelSim showing left-up and right-down gradual line test @ coordinates (1, 15), (10, 10)

The above waveform shows the results of our line drawer module after a gradual line test with the points (1,15) and (10,10) swapped. Visualizing the graph as in previous paragraphs, we can verify that our code increments x at every clock cycle while incrementing y every two clock cycles since the slope < 0, and this happens regardless of the order of the points, from 2100-2205ps and 2305-2415ps. Done is again asserted when the end point (10,10) is reached.

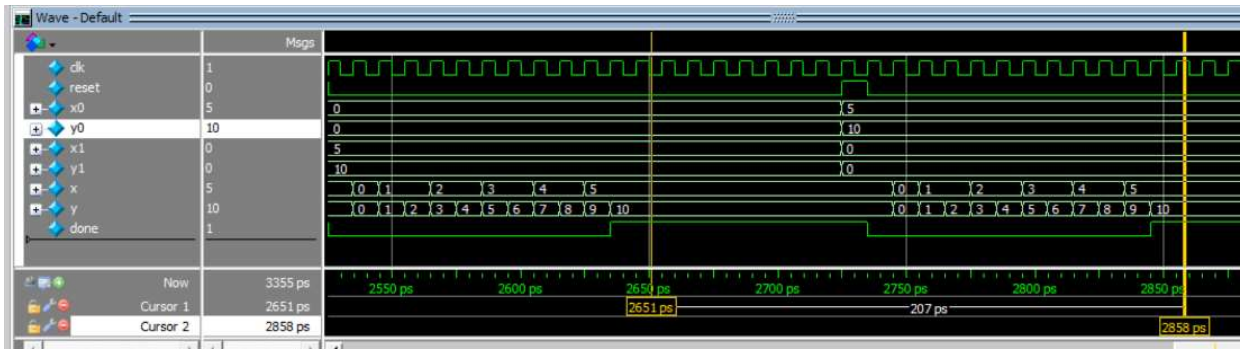


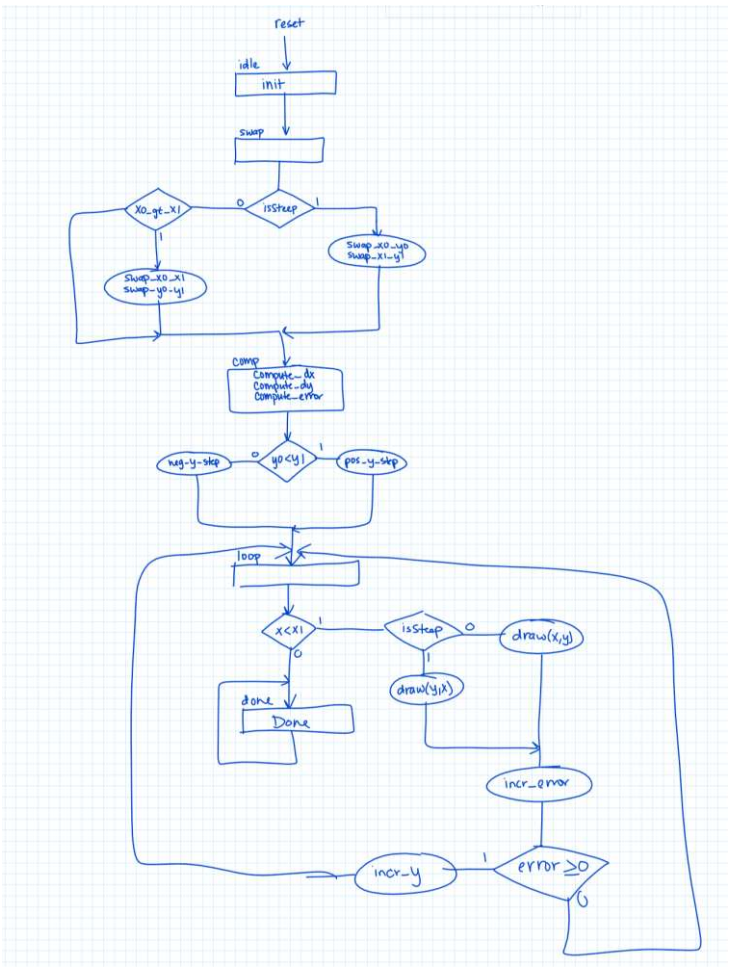
Figure 9: Snippet of ModelSim showing right-up and left-down steep line test @ coordinates (0, 0), (5, 10)

The above waveform shows the results of our line drawer module after a steep line test with the points (0,0) and (5,10) swapped. Visualizing the graph as in previous paragraphs, we can see that the algorithm corrects for the steep slope (>1) and iterates over y from 0 to 10 every clock cycle while incrementing x every two clock cycles. The same results happen regardless of the order of the points, as seen from 2525-2630ps and 2725-2850ps. Done is asserted when the end point (5,10) is reached.

Figure 10: Snippet of ModelSim showing right-up and left-down gradual line test @ coordinates (0, 0), (10, 5)

The above waveform shows the results of our line drawer module after a gradual line test with the points (0,0) and (10,5) swapped. Visualizing the graph as in previous paragraphs, we can verify that our code increments x at every clock cycle while incrementing y every two clock cycles since the slope < 0, and this happens regardless of the order of the points, from 2950-3060ps and 3160-3265ps. Done is again asserted when the end point (10,5) is reached.

In retrospect, we probably should have made an ASM chart at the begining instead of just trying to implement the code from the get-go, since we spent a significant amount of time debugging our line drawer module. Below is the line drawing algorithm modeled as an ASMD chart that we could have used to implement our code.



Task #3:

For task 3, we started by implementing an FSM that makes a line and sets the color of the VGA to white, then it makes the same line with the same coordinates but with the color set to black. Essentially, writing over the white line with black. We repeat that sequence of steps a couple more times with different coordinates to demonstrate full range of possibilities for task 2. We start by making it go from left to right at a positive slope. We then erase that and make a vertical line. After that, we make it go from left to right at a negative slope. We then erase that and make it draw a horizontal line. This sequence of states gives the appearance of a twirling stick. It also shows how the Bresenham's algorithm handles both steep and gradual lines.

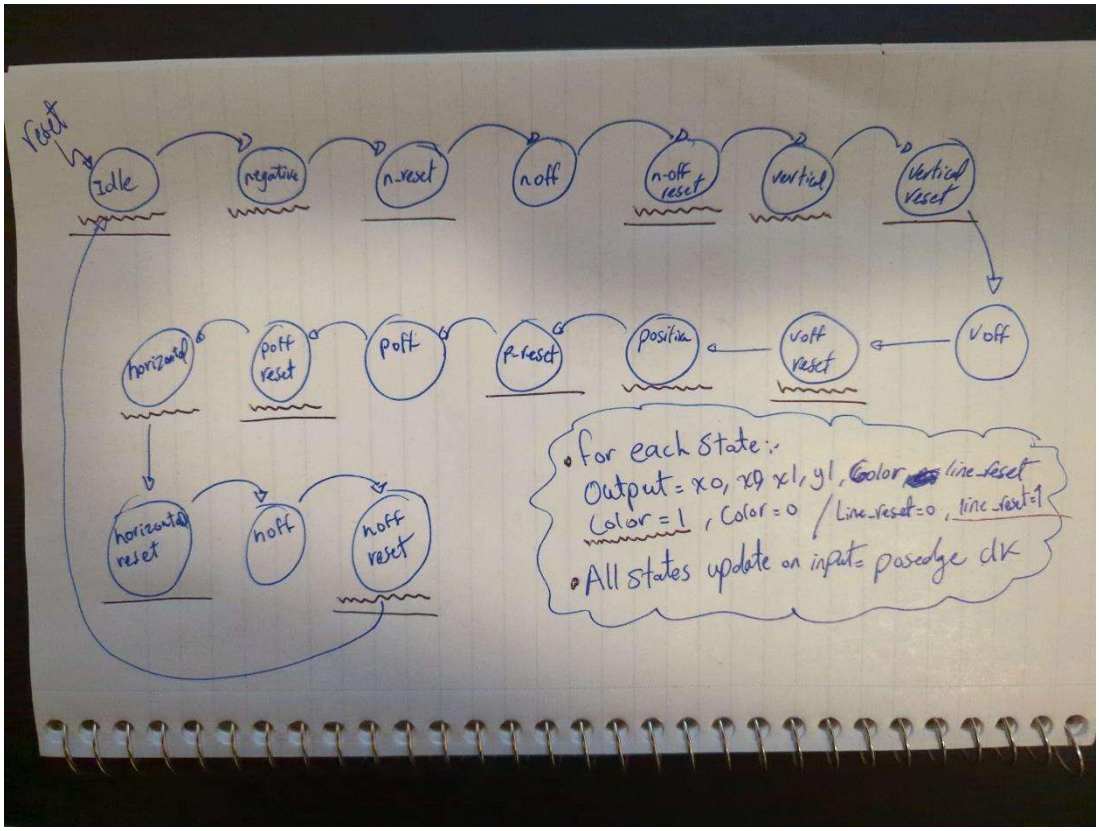


Figure 111: Snippet showing the animation FSM used in our DE1_SoC top level module. Underlines indicate Moore output color = 1, wiggly line indicates Moore output line_reset = 1 for easier viewing

As shown in the above FSM, we have an initial state *idle* that simply resets the line_drawer module to ensure that the proper coordinates are loaded. We then go through our desired movements (negative, vertical, positive, horizontal) in a set of 4 states (draw, draw reset, erase, erase reset). For example, in the first movement, we start by first drawing the line that we want by passing the coordinates to the line_drawer module, we then do a reset state to load in the new coordinates. After that, we pass the same exact coordinates but with signal color set to 0 to draw a black line instead of a white line. This effectively erases the line. We then do one more state of reset to load in the new coordinates for the next movement and so on and so forth until we're done with all 4 movements of the animation.

DE1_SoC:

Our top level module builds on top of our task 3 and just adds the capability to reset the VGA buffer with SW[5] and clear the screen with SW[0]. It also assigns LEDR[9] to the output signal done coming out of the line drawer module. To make our DE1_SoC animate our lines, we needed to use the clock divider module to create a slower clock in an always_ff block that would transition between the different states of the FSM. Slowing down the clock made the animation visible to the viewer. In order to have our line drawer automatically reset and draw new lines (and update done properly), we needed to create many states that toggled the line_reset variable.

If the state was a non-drawing state (idle or erase) we would assert line_reset signal which would reset our line drawer by refreshing x0 and y0.

```
// use a slower clock for animation so that it is visible to users
logic [31:0] new_clock;
clock_divider cd (.clock(CLOCK_50), .divided_clocks(new_clock));
logic animator_clk;
assign animator_clk = new_clock[8]; // slower clock to use

// state variables - drawing states, "erasing" states, idle state, and reset states to work with line drawer module
 enum {idle, negative, negative_reset, noff, noff_reset, vertical, vertical_reset, voff, voff_reset,
       positive, positive_reset, poff, poff_reset, horizontal, horizontal_reset, hoff, hoff_reset} ps, ns;

 // if color = 1, draw white line
 // if color = 0, draw black line
 always_comb begin
     case(ps)
         idle: // idle/reset state for drawing negative line
             begin
                 color = 1;
                 x0 = 120;
                 y0 = 200;
                 x1 = 320;
                 y1 = 300;
                 ns = negative;
                 line_reset = 1;
             end
         negative: // draw negative line
             begin
                 color = 1;
                 x0 = 120;
                 y0 = 200;
                 x1 = 320;
                 y1 = 300;
                 if(done) ns = negative_reset;
                 else ns = negative;
                 line_reset = 0;
             end

         negative_reset: // reset state for erasing negative line
            begin
                 color = 0;
                 x0 = 120;
                 y0 = 200;
                 x1 = 320;
                 y1 = 300;
                 ns = noff;
                 line_reset = 1;
            end
```

Figure 12: Snippet of DE1_SoC showing a portion of the FSM state logic

Results:

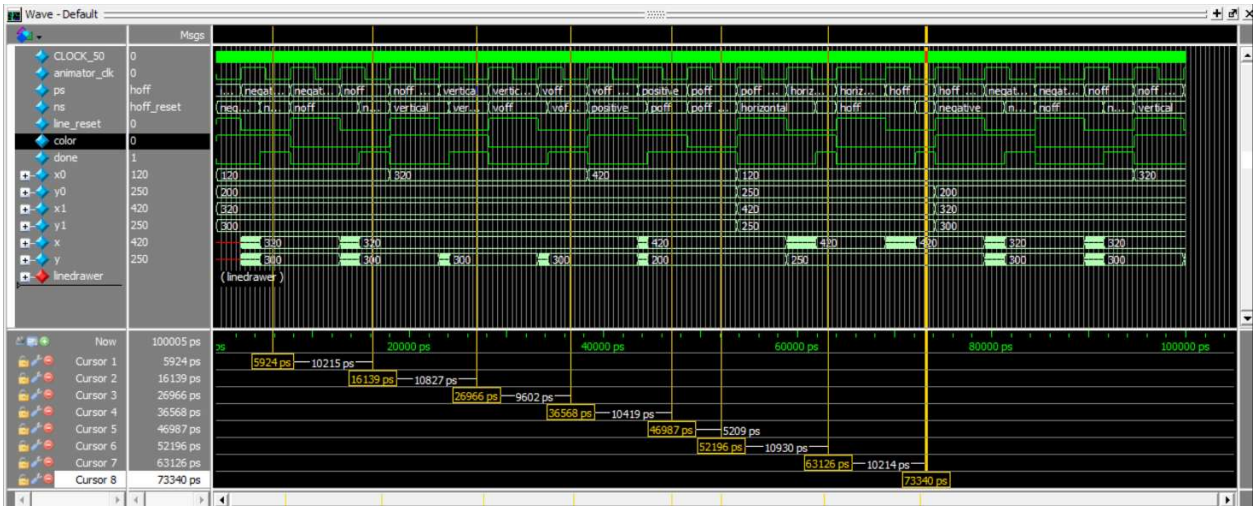


Figure 13: Snippet of ModelSim showing all cases test

The above waveform shows the simulation results of our overall design. We can see that our FSM transitions from each state as expected, first starting at idle and cycling through negative, vertical, positive, horizontal, and back to negative, including all states in between. We can also see that when new points are passed to the line drawer, the FSM does not transition to the next state until done is asserted, ensuring that the whole line is drawn before moving onto the next line.

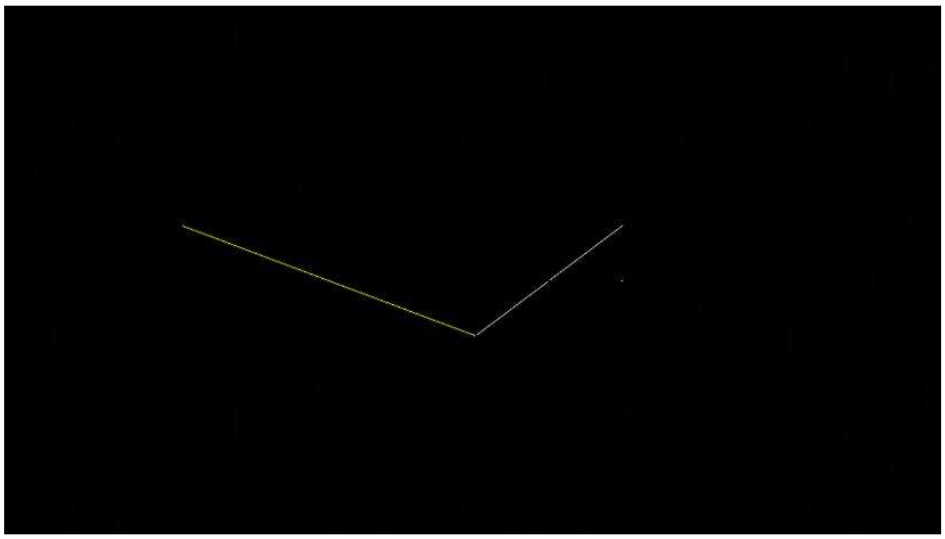


Figure 14: Snippet showing VGA display in movement 1 with ghost line

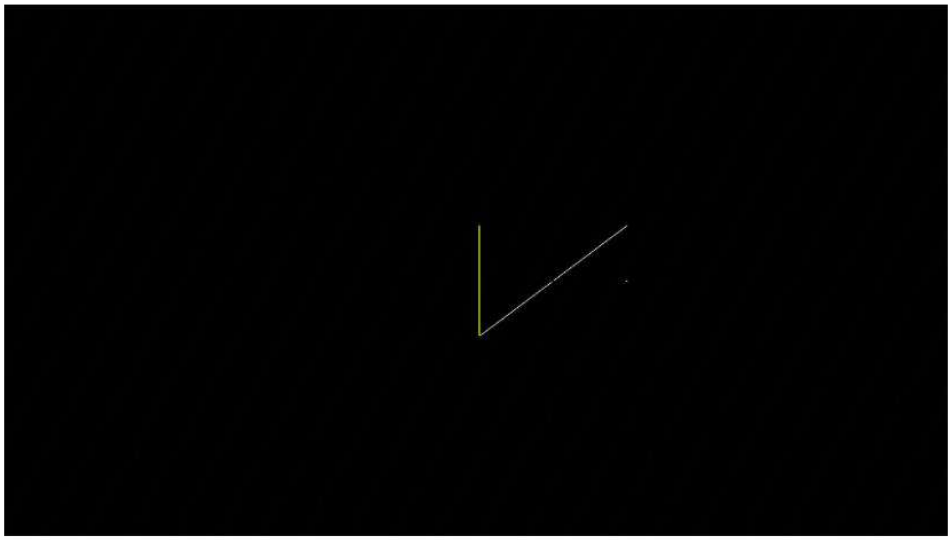


Figure 14: Snippet showing VGA display in movement 2 with ghost line

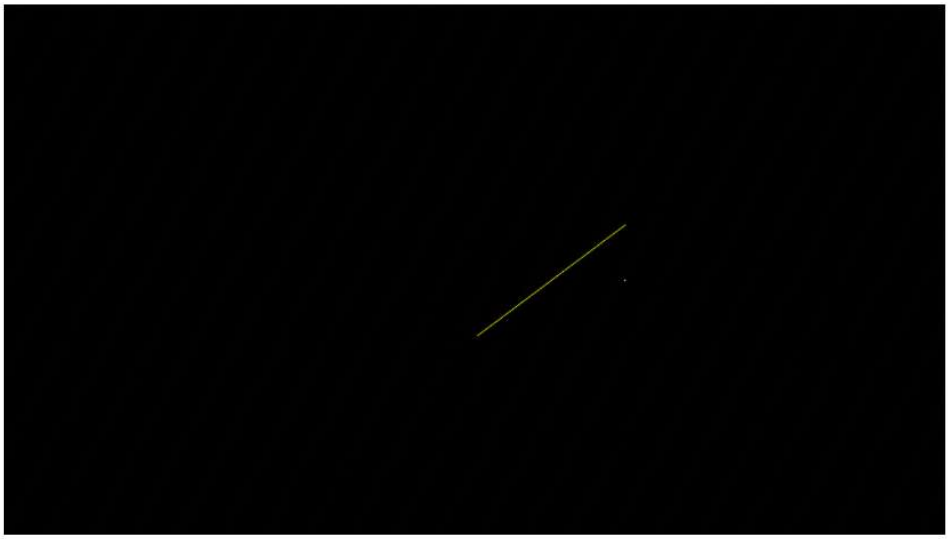


Figure 15: Snippet showing VGA display in movement 3 with ghost line

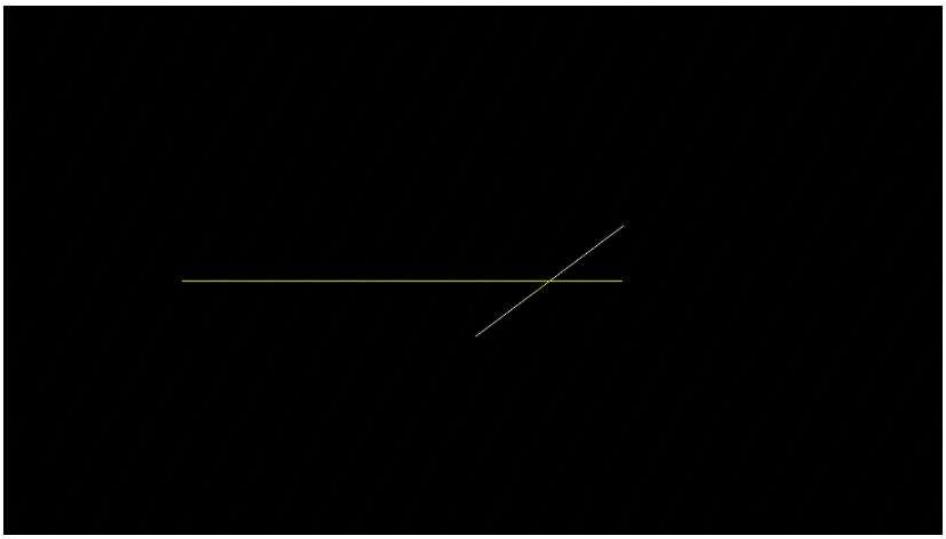


Figure 16: Snippet showing VGA display in movement 4 with ghost line

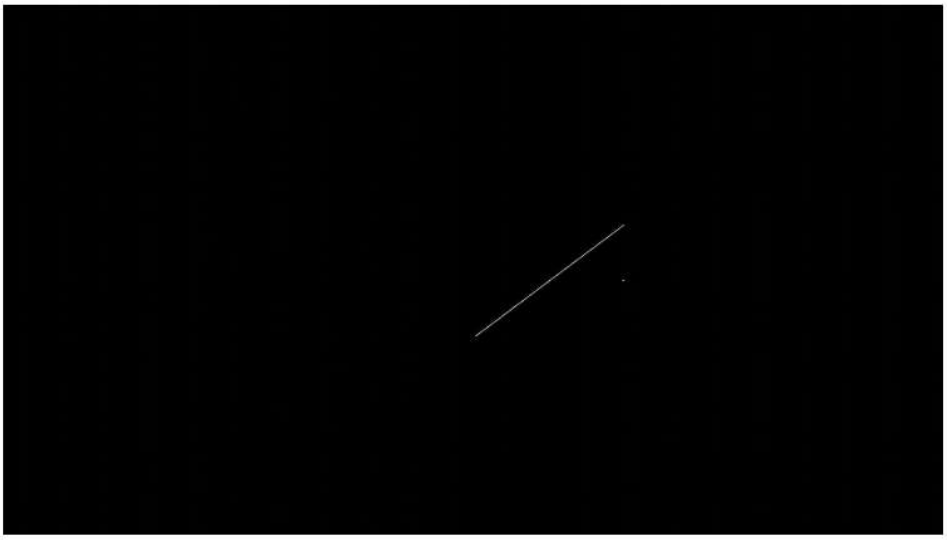


Figure 17: Snippet showing VGA display with clear switch asserted with ghost line.

As shown above the results reflect the expected behavior, we did encounter a weird "ghost line" artifact that we're unable to determine the cause of. However, the TA explained that this is likely caused by how our algorithm is interacting with the VGA buffer and we can simply ignore it. Ignoring the "ghost line" artifact we can see that movements 1-4 are being displayed and then cleared in sequence as expected. We also see our clear functionality clearing the display and the FSM being idle when reset is asserted.

Flow Summary:

Flow Summary	
<<Filter>>	
Flow Status	Successful - Fri ... 17 20:49:27 2023
Quartus Prime Version	17.0.0 Build 595 0...17 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	99
Total pins	96
Total virtual pins	0
Total block memory bits	307,200
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Figure 1318: Flow Summary

Experience Report:

Implementing the 2nd and 3rd tasks; Implement the Line-Drawing Algorithm and Animate an Object was extremely difficult. We initially encountered issues with which parts of the pseudocode are combinational and sequential. We also encountered problems with getting the DE1_SoC top level module to work with the VGA on LabsLand. We also spent a lot of time with the animation FSM. We had to review the class lecture slides multiple times throughout working on the lab. For teamwork, we used GitHub for easy collaboration of shared files which made code developing and debugging more efficient. It was not very helpful to reference the provided pseudocodes since they were incomplete and confusing. For the next lab we would like to get a head start and aim to work together live more frequently to reduce debugging time. Additionally, using GitHub's full functionality for version control may help make our code development process even smoother.

The estimated total time working on this lab was 45 hours broken as follows:

- Reading: 5 hours
- Planning: 5 hours
- Design: 5 hours
- Coding: 10 hours
- Testing: 10 hours
- Debugging: 10 hours