Garrett Tashiro

EE 371

November 5, 2021

Lab 3 Report

Procedure

Lab three was comprised of two tasks: the first task was to implement Bresenham's line algorithm and draw a line on a VGA display. There were two examples to base the line algorithm off of and they both did the same thing. The second task for this lab was to take the line algorithm and create an animation for a line that we draw. For the second task, the line drawing algorithm was changed slightly to be able to shift the line after it was drawn, and a clear screen module was created as well to be able to make the screen blank upon reset. Both tasks used Bresenham's line algorithm, and a VGA display. This lab gave an understanding about using the VGA display for DE1_SoC board, as well as being able to implement a line drawing algorithm.

Task #1

The first task we were given was to implement Bresenham's line algorithm in SystemVerilog, and draw a line on a VGA display. The way I approached this first task was by first reading over the information in sections "Pixel Buffer" and "Drawing" on the lab document. I wanted to have an understanding of this information prior to going over task 1. Once I had an understanding of both sections, I read over task 1. At first, I referred heavily to Figure 3 of the lab document to implement the line algorithm, but after spending many hours trying with the pseudo code, I ended up going to link provided on the lab document and following the C code version, which was much easier. The C code used conditional operators, which I hadn't seen much of before, and have never used. I went onto chip verify to learn about what they were, and how to implement them. I used conditional operators in a number of places in line_drawer.sv, and an example of some conditional operators that I used for this module can be seen below in Figure 1.

```
//Assigning dx with a conditional operator
//to get the absolute value of delta x
assign dx = (x1 > x0) ? x1 - x0 : x0 - x1;

//Assigning dy with a conditional operator
//to get the absolute value of delta y
assign dy = (y1 > y0) ? y1 - y0 : y0 - y1;
```

Fig. 1: Conditional Operators for Absolute Values of Delta Y and Delta X

Since the C code used a for loop, I knew I needed to use an FSM to be able to update the 10-bit output x, and the 9-bit output y every clock cycle. I followed along with the C code, making minor adjustments to be able to put the FSM in an always_ff block. I used two states in my FSM: start, and draw. Upon reset, I had the state set to start, which is where all the values for the outputs were assigned. If reset was low, then the FSM would go straight to the start state. As long as 10-bit inputs x0 and x1 weren't equal, as well as 9-bit inputs y0 and y1 were equal, then the FSM will go to the draw state. In the draw state, the FSM checks if the line is done being drawn by using checking if 10-bit output x is equal to 10-bit input x1 and if 9-bit output y is equal to 9-bit input y1. If x == x1 and y == y1 then the state is updated to start. An else statement is used after which has three if statements inside of it to test e2 (two times the error value) against dx, and -dy. Each statement has their own updates to either 10-bit output x, 9-bit output y, or both x and y, while also updating the value for error in each

of the if statements. After the if statements state is set to draw. The diagram for the FSM can be seen below in Figure 2.

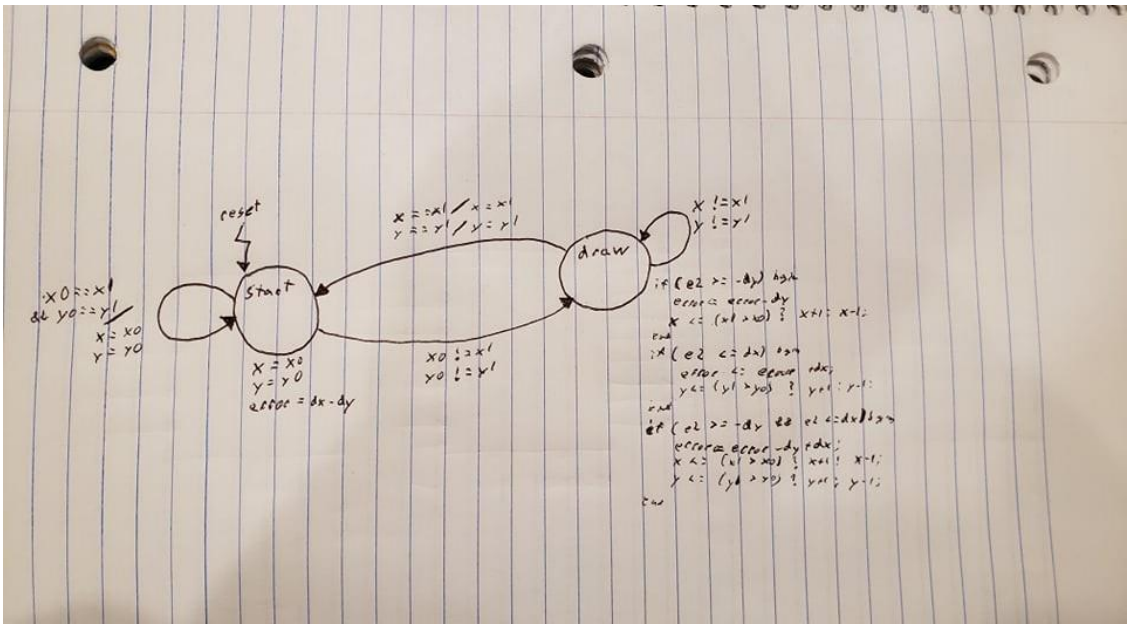

Fig. 2: Line Drawer FSM for Task 1

The reason as to why I setup my code the way I did was because I found using the C code to be very easy compared to the pseudo code that was provided. The FSM was needed to be able to update the outputs for x and y along with the error value every clock cycle. Since I wasn't using purely combinational logic, I chose to use and always_ff block. I put 12-bit logic e2 in an always comb, so that it would update with the new value for error on the new clock cycle. The code for line_drawer can be seen in Appendix 1.A. With the line_drawer module done, I created a testbench for DE1_SoC and tested to make sure that the correct values were being passed from line_drawer to VGA_framebuffer. The code for DE1_SoC was given, and the only thing changed were the values for x0, x1, y0, and y1. The code for DE1_SoC for task 1 can be found in Appendix 1.C. The VGA_framebuffer module was another given module. This code was untouched by me, and the code can be found in Appendix 3.A since both tasks 1 and 2 use this module.

Task #2

For the second task, we had to take the line algorithm that was implemented in task 1 and draw a line, but also animate the line on the VGA display. How I did this was I first read what was asked of us in the task. I knew that I was going to either change my line_drawer module, or create a new module to be able to shift the line after the line was drawn. I figured that CLOCK_50 was going to be too fast for the line, so I decided to use my clock_divider module that I had from previous labs to slow down the line_drawer module. Since we had to animate a line, I figured that I wanted to just shift a line from left to right across the screen, so I started by creating a module called shiftx. This module would take in the 10-bit values x0 and x1 that were in DE1_SoC, and the output from shiftx would be new, shifted x0 and x1 values that would be passed to line_drawer. I had a problem with this method though, and had to scrap the idea. I decided to primarily focus on making changes to the FSM in line_drawer to be able to control everything. I started by adding a third state to the FSM, which was the done state. In this state, the line would be done drawing and would also control the color of the line. I knew I wanted to have some sort of delay for the white line in the done state so that the white line would show up for a moment before being drawn over in black. I decided to implement a counter in the done state

for when the color of the line was white and it would just loop in the done state until the counter hit 1700. After the counter hit 1700, or there was a 1700 clock cycle delay, the color would change to black, and the state would be set to start. This would then draw over the white line in black completely, and once the black line was finished, the color would be set back to white, and then the state would be set to start again to draw another white line. The diagram for my FSM can be seen below in Figure 3.

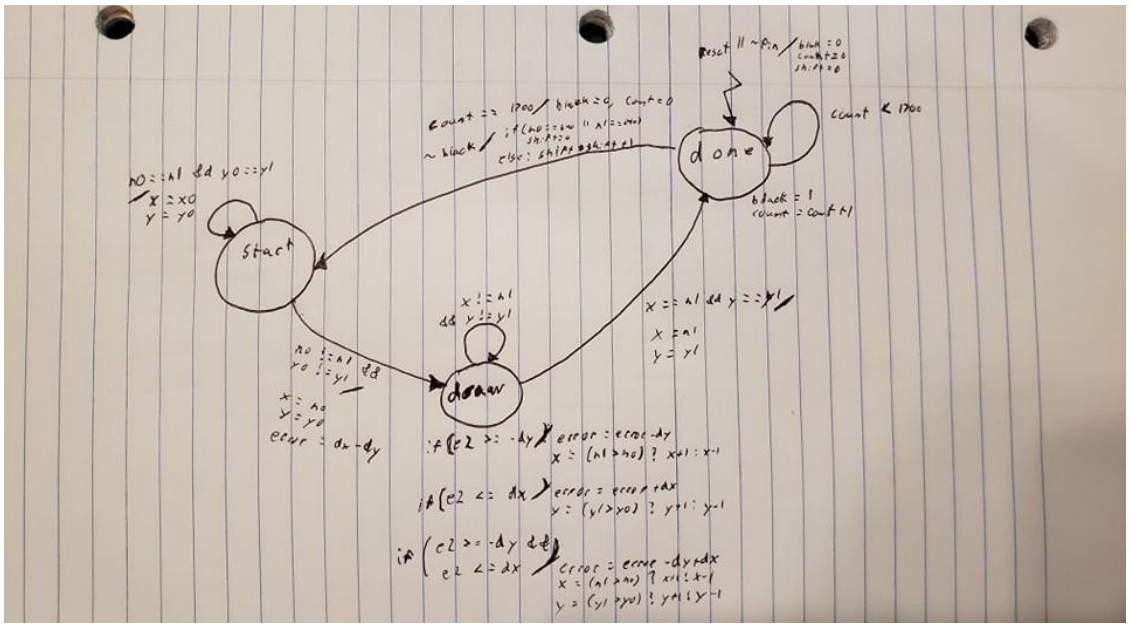

Fig. 3: Line Drawer FSM for Task 2

I had some troubles figuring out the logic for actually shifting the line at first, so I primarily focused on having a white line be drawn, then have the line be "erased" by having a black line go over the white line. I slowed down the clock speed for the line using the clock_divider and made sure that everything was working accordingly. Once I had that working, I moved onto shifting the line. I wanted to hold onto the initial 10-bit values for x0 and x1 that were passed, along with being able to update those values after the white line was drawn over with the black line. I decided to create two new 10-bit logics n0, and n1 which would be the new x0, and x1 values. These values were updated with 11-bit logic shift. I had n0, and n1 in an always comb and they their respective x value added with shift. The code for assigning 10-bit logic n0, and n1 can be seen below in Figure 4.

```
//This always_comb block updates the 12-bit value
//e2 whenever error is changed, update n0, and n1
//whenever a line is drawn over and needs to be shifted
//once shift is updated
always_comb begin
   e2 = 2*error;
   n0 = (x0 + shift);
   n1 = (x1 + shift);
end
```

Fig. 4: Assignment of Logic n0 and n1

The shift value would be updated after the black line drew over the white line, and I used and if/else statement in the done state to do this. Since the VGA display was 640x480, I had an if statement while in the done state to check if the value of n0 was equal to 640, or if n1 was equal to 640. If n0, or n1 were equal to 640 then shift would be set to 0, otherwise shift would increment by 1. The state would then be set to start and the location of the new line would be shifted over by 1, or set back to the original x0 and x1 values. This can be seen below in Figure 5, and the full code for the line_drawer module can be found in Appendix 2.A.

```
if(~black) begin
   black <= 1'b1;
   if((n0 == 640) || (n1 == 640)) begin
       shift <= '0;
   end

   else begin
       shift <= shift + 1;
   end
   state <= start;
end
```

Fig. 5: Logic for Shifting the Line Across the VGA Display

Once I was able to draw a white line, draw over the line with a black line, and shift the line to the right, I moved onto implementing a clear screen module that I called clr_scr. I knew that I wanted to use the 10-bit x, and 9-bit y outputs from this module upon reset instead of the ones from line_drawer. This caused a little bit of confusion for me for a bit, but I was able to use two separate conditional operators to assign x and y in DE1_SoC. I also assigned 1-bit pixel_color in DE1_SoC as well. I controlled what values were being passed as inputs into VGA_framebuffer by using a flag in clr_scr called done. This 1-bit value for the output done is held in a 1-bit logic called finished in DE1_SoC. The conditional operators can be seen below in Figure 6.

```
//Conditional operator to assign x. If 1-bit finished being
//passed from clr_scr is true, then the x value output from
//line_drawer will be used. If finished is false, the x value
//output from clr_scr will be used.
assign x = (finished) ? xl : xc;

//Conditional operator to assign y. If 1-bit finished being
//passed from clr_scr is true, then the y value output from
//line_drawer will be used. If finished is false, the y value
//output from clr_scr will be used.
assign y = (finished) ? yl : yc;

//Conditional operator to assign pixel_color. If 1-bit finished
//is true, then use the 1-bit l_color from output black in
//line_drawer. If it is false, use 1-bit c_coloc from output
//color of clr_scr.
assign pixel_color = (finished) ? l_color : c_color;
```

Fig. 6: Conditional Operators for Assigning Logic x, y, and pixel_color

Upon reset, 1-bit done, 1-bit color, 10-bit x, and 9-bit y outputs inside clr_scr are all set to 0. After they are set to 0, the moves across every pixel on the VGA display and changes the color of each pixel to black. This is done by using if/else statements in an always_ff block. The first test is if 10-bit output x is less than 641. If it is less than 641, then there is a nested if statement to check if 9-bit ouput y is less than 480. If output y is less than 480, then y is incremented by 1. If y is not less than 480, then y is set 0 and x is incremented by 1. Once 10-bit output x is greater than, or equal to 641, both 1-bit color, and 1-bit done are set to 1. This is to set the color back to white, and then also have done pass the value to finished in DE1_SoC so that the values for x and y take on the values passed from line_drawer now instead and start drawing lines again. The code for the always_ff block in clr_scr can be seen below in Figure 7.

```
always_ff @(posedge clk) begin
   if(reset) begin
      x <= '0;
      y <= '0;
      color <= 1'b0;
      done <= 1'b0;
   end

   else begin
      if(x < 641) begin
         if(y < 480) begin
      if(x < 6) begin
         if(y < 4) begin
            y <= y + 1;
         end

         else begin
            x <= x + 1;
            y <= 0;
         end
      end

      if(x >= 641) begin
      if(x >= 6) begin
         color <= 1'b1;
         done <= 1'b1;
      end
   end
end
```

Fig. 7: The Logic for Clearing the Screen by Changing All Pixels to Black

After I had the line_drawer module updated to draw a white line and erase it, as well as the clr_scr module to clear the screen and change every pixel to black, both of which were in the DE1_SoC module, I created a testbench for the DE1_SoC module. I changed the values for the if statements inside clr_scr, as well as changed what the count went up to in line_drawer to create a smaller testbench that was more manageable to see if there were errors. I wanted to check if the modules were all working together on a smaller scale, prior to running it on the DE1_SoC board using the VGA display. The reason as to why I did this task the way I did was to make sure I did what was asked of me in the task. I wanted to break the task down into smaller tasks so I could make sure that each piece was working by itself prior to putting them together. The full code for the clr_scr module can be found in Appendix 2.C. The code for the clock_divider module can be found in Appendix 2.E. The code for the DE1_SoC module can be found in Appendix 2.G. Lastly, the code for VGA_framebuffer was untouched, and shared with task one, so the code for this module can be found in Appendix 3.A.

Top-Level Block Diagrams

Task #1

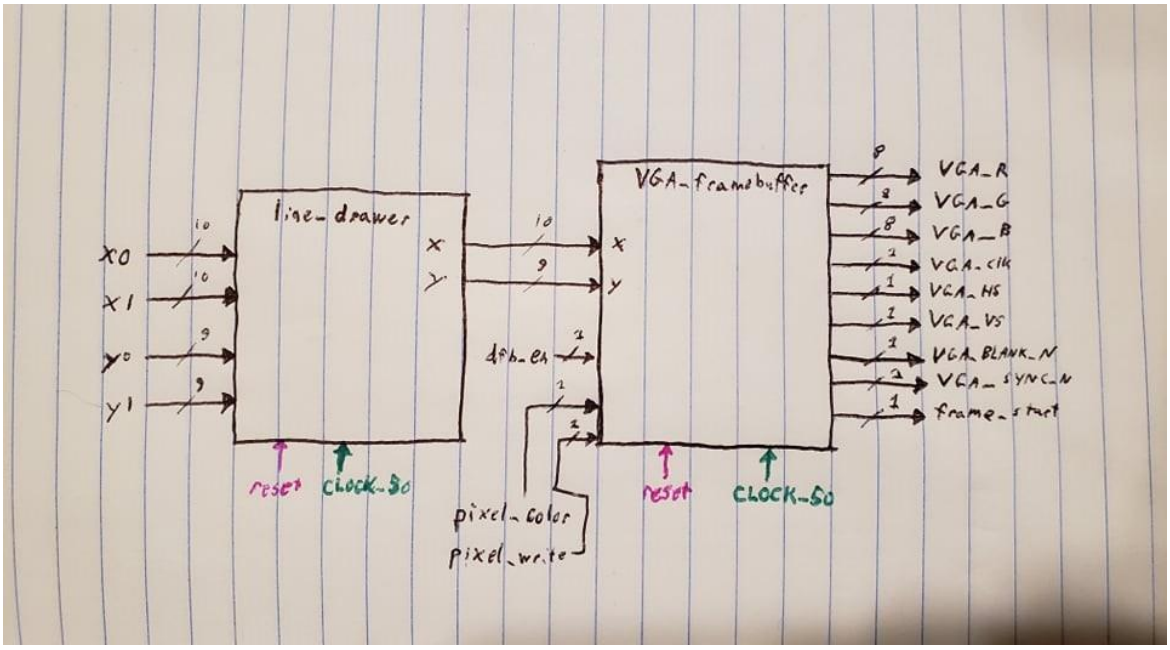

Fig. 8: DE1_SoC Block Diagram for Task 1

Task #2

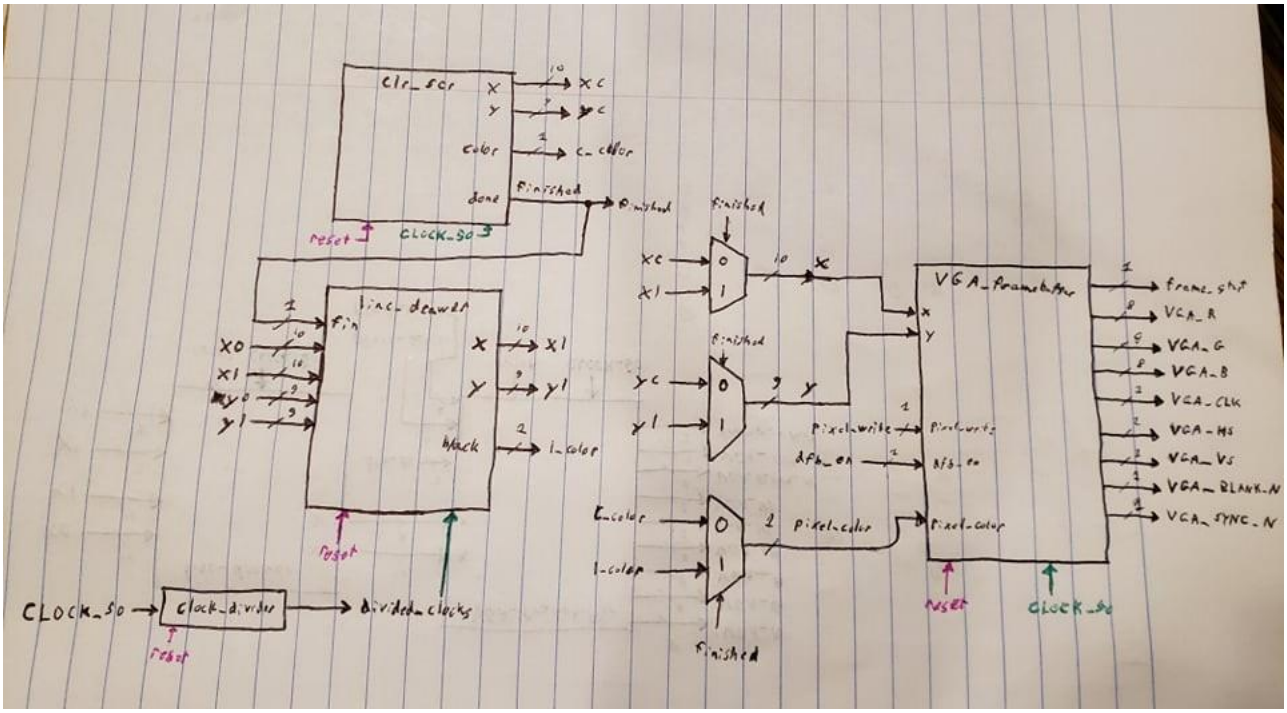

Fig. 9: DE1_SoC Block Diagram for Task 2

Results

Task #1

The first testbench I made for task 1 was for line_drawer. For this testbench I first started by setting 10-bit x0, x1, 9-bit y0, and y1 input values to 0. I then set 1-bit reset high for one clock cycle, then to low. Once reset was low, I set input x1 to 2, and y1 to 3. I kept these values for seven clock cycles in total to so I could see 10-bit output x, and 9-bit output y update accordingly, along with the state transitioning as it should. After that I set reset high for one clock cycle, then set it to low. After that I set both 10-bit x1, and 9-bit y1 to 0 and set 10-bit x0 to 4 and 9-bit y0 to 5. Since these values were a bit farther apart than the first set of coordinate pairs, I held these values for 11 clock cycles to make sure that 10-bit output x, 9-bit output y, and the state for the FSM updated accordingly. For the third test I set x0 and y0 to 0, so both coordinate pairs were 0, I set reset high for a clock cycle and then low, and held those values for five clock cycles. This was to make sure that outputs x and y wouldn't change at all after going to 0, since both x0 was equal to x1, and y0 was equal to y1. The final test I set x0, x1, and y0 to 0 and set y1 to 8. I set reset high for a clock cycle and then low for 15 clock cycles. This was to check to see if outputs x and y updated accordingly while the line was vertical. The results from these tests were as expected. The error and e2 values updated properly to then change the outputs x and y accordingly. The state changes happened when they were supposed. I was able to change the input values x0, x1, y0, and y1 and have outputs x, and y increment/decrement according to the error, and e2 values. This can be seen below in Figure 10.

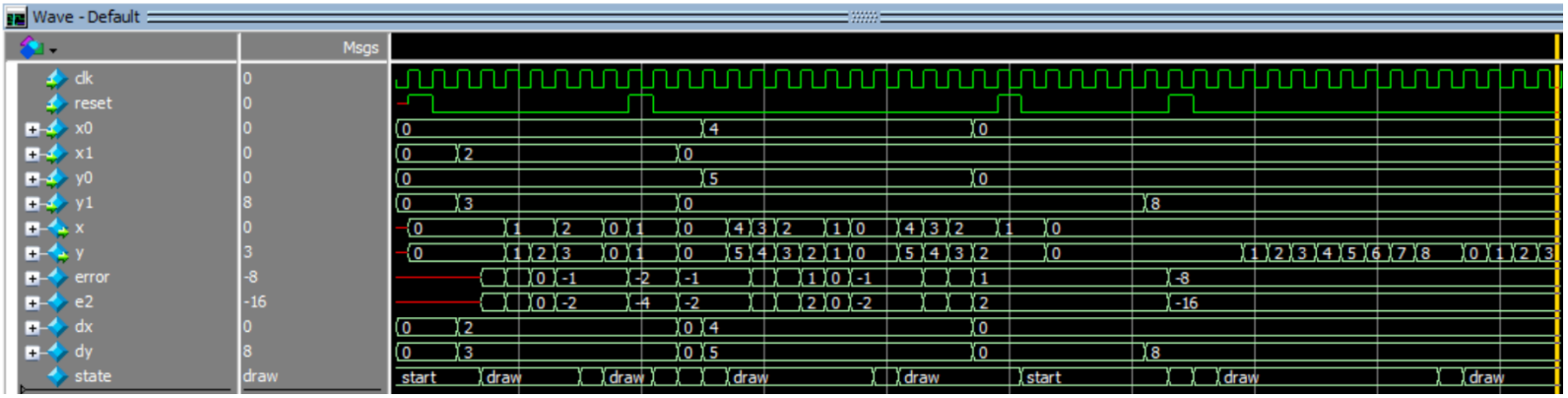

Fig. 10: line_drawer ModelSim Simulation

The second testbench I made for task 1 was for the DE1_SoC module. I set the values for x0 and y0 to 0, and set x1 to 10, and y1 to 20 inside the DE1_SoC module. I used coordinate pairs with small numbers in order to easily see the behavior of the system. In the testbench I set reset high for one clock cycle, and then low for 50 clock cycles. This was done so I could see 10-bit x, and 9-bit y updating, and see if the values cycled back to their starting points. The results for this test were as expected. The values for x and y updated accordingly, and after reaching the endpoint they reset to the starting point. They went from the starting point to the end point as they should. This can be seen below in Figure 11.

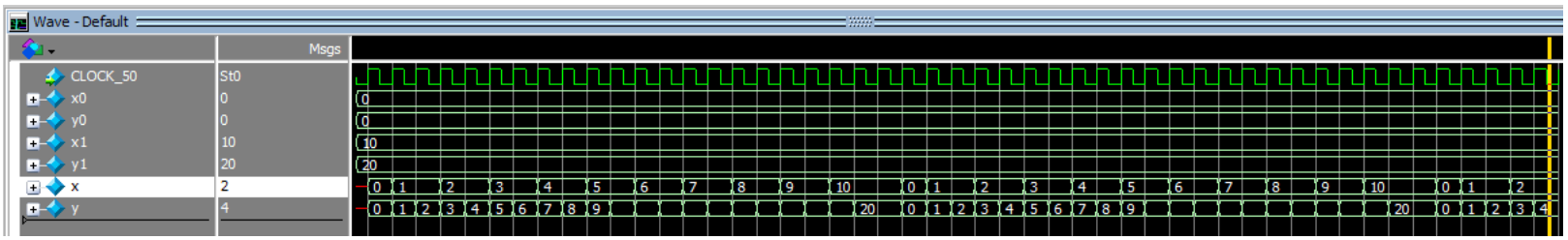

Fig. 11: DE1_SoC ModelSim Simulation

Task #2

The first testbench I made for task 2 was for line_drawer. In the done state of the line_drawer module I set the max count to 10. This was done so that the tests wouldn't run for over 1700 clock cycles. For the first test I set 10-bit x0 to 0, 10-bit x1 to 3, 9-bit y0 to 0, and 9-bit y1 to 3. I held these vales for 40 clock cycles. After that, I set reset high for a clock cycle, and then low. I then set x0 to 2, x1 to 0, y0 to 3, and y1 to 0. I held these values for 40 clock cycles. The last test I set 1-bit fin low, then high, and set x0, x1, y0, y1 all to 0. I held these values for 6 clock cycles. The results from these tests were as expected. I was able to change 10-bit x0, x1, 9-bit y0, and y1 input values and the line drew as it should. Reset also worked accordingly, as well as fin. I could set fin to 0, and the system would reset. After a line is drawn in white, it held for the 10 clock cycles that I set count to, then it would change the color of the line, redraw it in black, and then update 10-bit n0, n1, and 11-bit shift. This shifts the line after the black line draws over the white line. This can be seen below in Figure 12.

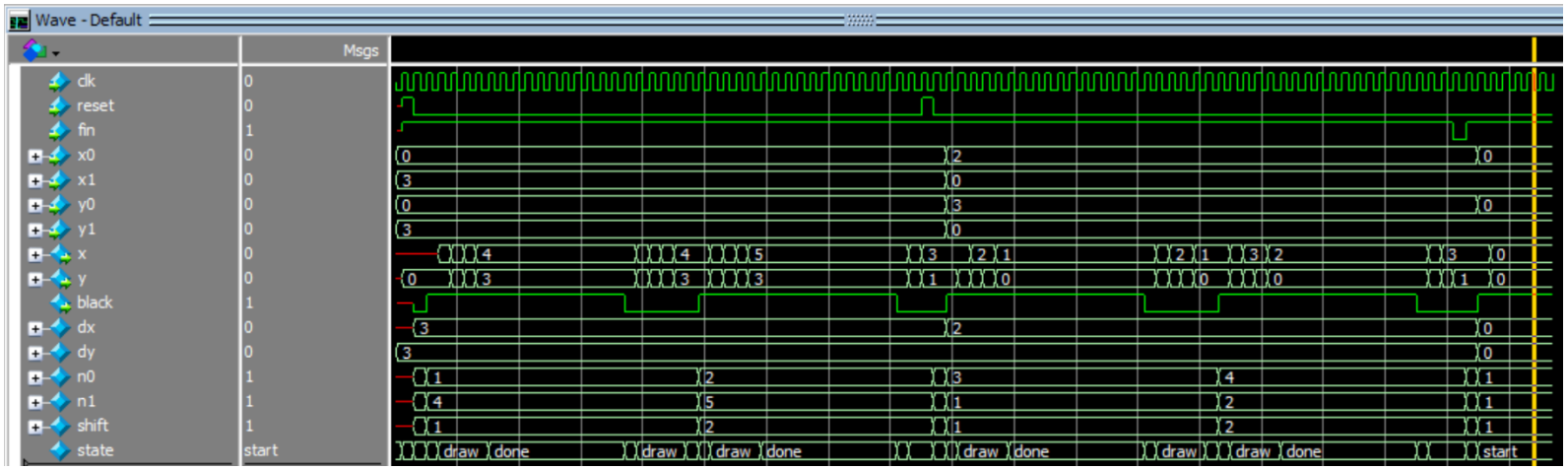

Fig. 12: line_drawer ModelSim Simulation

From there I moved onto clock_divider. The clock_divider testbench is very simple. Reset is set high and then low, and then 100 clock cycles pass. When the div_clk array is full, the that will count as one clock cycle. Without this clock_divider I wouldn't have been able to display memory addresses and the associated data using CLOCK_50. The testbench for this can be seen below in Figure 13.

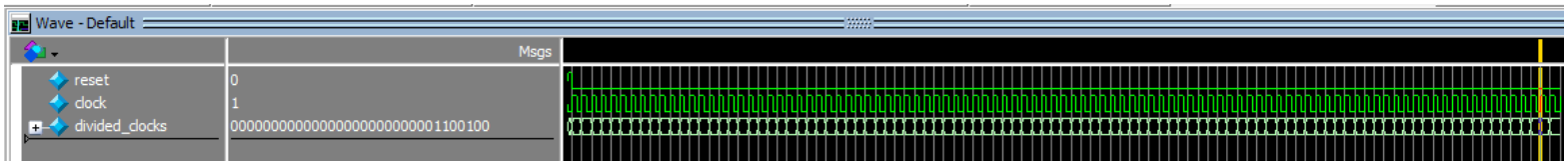

Fig. 13: clock_divider ModelSim Simulation

The next module I worked on was the clr_scr module. For testing, I changed the max values that 10-bit output x, and 9-bit output y counted up to. Inside the always_ff block inside the module I had x go to a max of 6, and y for to a max of 4. This was done to see the behavior of the test easier. In the testbench, I set reset high, and then low for 35 clock cycles. This was done so that the behavior of outputs 10-bit x, 9-bit y, 1-bit color, and 1-bit done could be observed. The results from this testbench were as expected. The values for x and y counted up to their max values like expected, and once x reached its max value, it stopped counting up. Color and done both changed from 0 to 1 at this point as well. This can be seen below in Figure 14.

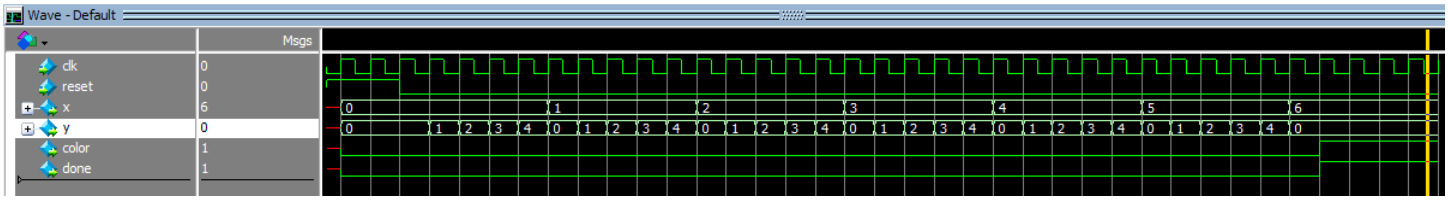

Fig. 14: clr_src ModelSim Simulation

The last module I worked on was the DE1_SoC module. For this test I changed values inside line_drawer and clr_scr for a smaller, and easier to view test. In line_drawer I had the max count in the done state be set to 10. In clr_scr I had the max value for x be set to 6, and the max value for y be set to 4. Inside the DE1_SoC module I set 10-bit x0 to 0, 10-bit x1 to 0, 9-bit y0 to 0, and 9-bit y1 to 10. To start the test, I set reset high for one clock cycle, and then low for 150. I ran this test twice. The results from this test were as expected. Upon reset, the 10-bit x, and 9-bit y values being output from clr_scr were used. Once the clr_scr module was finished, it's done flag was raised, which then had x and y take on the values being output from line_drawer. The line_drawer module drew the line in white, held the white line for 10 clock cycles, then drew over the line in black. After that, the x values were shifted. The x value increments properly after the line_drawer shifts. This can be seen below in Figure 15.

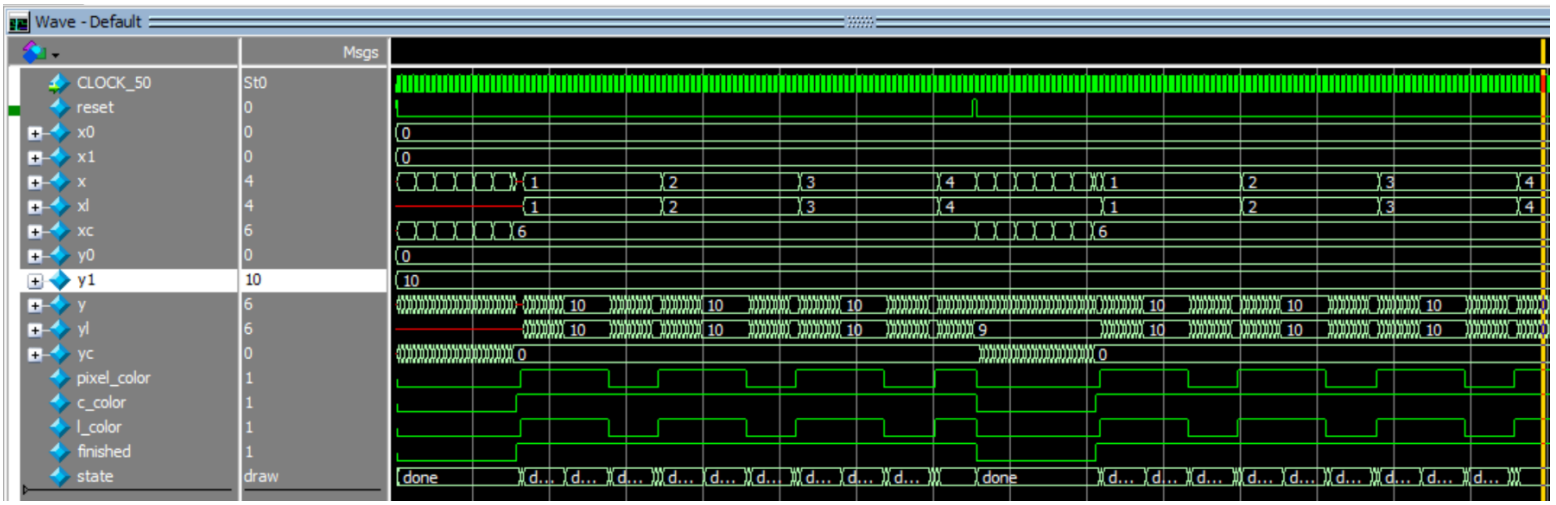

Fig. 15: DE1_SoC ModelSim Simulation

Final Project

The goal of this lab was to learn how to implement a line drawing algorithm and use it to draw a line, as well as animate a line on a VGA display that is connected to the DE1_SoC board using SystemVerilog. In the lab we learned about Bresenham's line algorithm and how to implement this algorithm in SystemVerilog. With that knowledge of the line algorithm, we learned how to animate a line on the VGA display, and clear the VGA display upon reset. The most challenging part of this lab was animating the line across the VGA display. I had issues with the line not being drawn over in black, the line not moving correctly, as well as the line not even showing up at times. This was the first time I had worked with a VGA display attached to the DE1_SoC board. I also had never used conditional operators prior to this lab, and didn't know much about them, but I learned how to use them, as well as how useful they can be.

In the end, I was mostly able to produce the result that I wanted. Task 1 worked exactly how I hoped, but task 2 was a bit different. In the end though, I do feel as if they were both sufficient to cover the requirements for lab 3.

Appendix

1.A) line_drawer.sv

```
1   //Garrett Tashiro
2   //October 30, 2021
3   //EE 371
4   //Lab 3, Task 1
5
6   //line_drawer module has 1-bit clk, reset, 10-bit x0, x1,
7   //9-bit y0, and y1 as inputs and returns 10-bit x, and
8   //9-bit y as outputs. This module takes in corrdinate
9   //pairs (x0, y0) and (x1, y1) and will update outputs x and
10  //and y to then draw a ling from one point to another.
11  module line_drawer(clk, reset, x0, x1, y0, y1, x, y);
12        input logic          clk, reset;
13        input logic [9:0]    x0, x1;
14        input logic [8:0]    y0, y1;
15        output logic [9:0]   x;
16        output logic [8:0]   y;
17
18        //12-bit logic for the error and two times the error
19        //10-bit logic for delta x and delta y
20        logic signed [11:0] error, e2;
21        logic signed [9:0] dx, dy;
22
23        //Assigning dx with a conditional operator
24        //to get the absolute value of delta x
25        assign dx = (x1 > x0) ? x1 - x0 : x0 - x1;
26
27        //Assigning dy with a conditional operator
28        //to get the absolute value of delta y
29        assign dy = (y1 > y0) ? y1 - y0 : y0 - y1;
30
31        //Two states for the line_drawer FSM
32        enum{start, draw} state;
33
34        //This always_comb block is to update 12-bit e2 to
35        //be two times what 12-bit error value is
36        always_comb begin
37           e2 = 2*error;
38        end
39
40       //This always_ff block holds the FSM and draws the line for the
41       //system. If 1-bit input reset is high, state is set to start.
42       //The else portion is the logic for the FSM. The system will start
43       //in the start state and assign 10-bit output x to 10-bit input x0,
44       //and 9-bit output y to 9-bit input y0. If the x inputs are equal
45       //as well as the y inputs, then the FSM will stay in the start state.
46       //If they are not equal, the state machine will tranistion to the draw
47       //state and increment 10-bit out x and 9-bit out y accordingly until the
48       //line is drawn.
49       always_ff @(posedge clk) begin
50          if(reset) begin
51             state <= start;
52          end
53
54          else begin
55             case(state)
56                start: begin
57                   if((x0 == x1) && (y0 == y1)) begin
58                      x <= x0;
59                      y <= y0;
60                      state <= start;
61                   end
62
63                   else begin
64                      x <= x0;
65                      y <= y0;
66                      error = dx - dy;
67                      state <= draw;
68                   end
69                end
70
71                draw: begin
72                   if((x == x1) && (y == y1)) begin
73                      x <= x1;
74                      y <= y1;
75                      state <= start;
76                   end
```

```
                else begin

                    if(e2 >= -dy) begin
                        error <= error - dy;
                        x <= (x1 > x0) ? (x + 1) : (x - 1);
                    end

                    if(e2 <= dx) begin
                        error <= error + dx;
                        y <= (y1 > y0) ? (y + 1) : (y - 1);
                    end

                    if((e2 >= -dy) && (e2 <= dx)) begin
                        error <= error - dy + dx;
                        x <= (x1 > x0) ? (x + 1) : (x - 1);
                        y <= (y1 > y0) ? (y + 1) : (y - 1);
                    end

                    state <= draw;
                end
            end
        endcase
    end
endmodule
```

1.B) line_drawer.sv (testbench)

```

//line_drawer_testbench tests for expected and unexpected behavior.
//The testbech first sets all the inputs to 0, and resets. The first
//test is changing x1, and y1 and holds the value for 6 clock cycles.
//Next, reset is set high then low. x1, and y1 are both set to 0 while
//x0, and y0 are increase to different numbers under 10. The third test
//is having all input values be equal to zero, then setting reset high
//then low. reset stays low for 5 clock cycles. This was to check if
//the output values would stay at x0 and y1 and the FSM would stay in
//the start state. The final test I set x0, x1, and y0 to 0 and had
//y1 set to 8. I set reset high, then low for 15 clock cycles.
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, .reset, .x0, .x1, .y0, .y1, .x, .y);

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

     initial begin
        x0 <= 9'd0; x1 <= 9'd0; y0 <= 8'd0; y1 <= 8'd0;  repeat(1)   @(posedge    clk);
        reset <= 1;                 repeat(1)   @(posedge    clk);
        reset <= 0;                 repeat(1)   @(posedge    clk);
        x1 <= 9'd2; y1 <= 8'd3;     repeat(1)   @(posedge    clk);
                                    repeat(6)  @(posedge    clk);
        reset <= 1;                 repeat(1)   @(posedge    clk);
        reset <= 0;                 repeat(1)   @(posedge    clk);
        x1 <= 9'd0; y1 <= 8'd0;     repeat(1)   @(posedge    clk);
        x0 <= 9'd4; y0 <= 8'd5;     repeat(1)   @(posedge    clk);
                                    repeat(10)  @(posedge    clk);
        x0 <= 9'd0; x1 <= 9'd0; y0 <= 8'd0; y1 <= 8'd0;  repeat(1)   @(posedge    clk);
        reset <= 1;                 repeat(1)   @(posedge    clk);
        reset <= 0;                 repeat(5)   @(posedge    clk);
        x0 <= 9'd0; x1 <= 9'd0; y0 <= 8'd0; y1 <= 8'd8;  repeat(1)   @(posedge    clk);
        reset <= 1;                 repeat(1)   @(posedge    clk);
        reset <= 0;                 repeat(15)   @(posedge    clk);
        $stop; // End the simulation.
     end
endmodule
```

1.C) DE1_SoC.sv

```
1   //Garrett Tashiro
2   //October 30, 2021
3   //EE 371
4   //Lab 3, Task 1
5
6   //DE1_SoC is the top level module for Lab 3, Task 1. This module implements
7   //Bresenham's line drawing algorith to draw a line on a VGA display. DE1_SoC
8   //uses hierarchical calls to VGA_framebuffer, and line_drawer. This module
9   //doesn't use any physical inputs from the DE1_SoC board, but has values set
10  //inside the module for x0, x1, y0, and y1 which are passed to line_drawer
11  //and line_drawer returns 10-bit x and 9-bit y to VGA_framebuffer as inputs
12  //to draw the white line across the VGA display.
13  module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
14      VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS);
15
16      output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
17      output logic [9:0] LEDR;
18      input logic [3:0] KEY;
19      input logic [9:0] SW;
20
21      input CLOCK_50;
22      output [7:0] VGA_R;
23      output [7:0] VGA_G;
24      output [7:0] VGA_B;
25      output VGA_BLANK_N;
26      output VGA_CLK;
27      output VGA_HS;
28      output VGA_SYNC_N;
29      output VGA_VS;
30
31      assign HEX0 = '1;
32      assign HEX1 = '1;
33      assign HEX2 = '1;
34      assign HEX3 = '1;
35      assign HEX4 = '1;
36      assign HEX5 = '1;
37      assign LEDR = SW;
38
39      logic [9:0] x0, x1, x;
40      logic [8:0] y0, y1, y;
41      logic frame_start;
42      logic pixel_color;
43
44
45      //////// DOUBLE_FRAME_BUFFER ////////
46      logic dfb_en;
47      assign dfb_en = 1'b0;
48      ////////////////////////////////////
49
50      VGA_framebuffer fb(.clk(CLOCK_50), .rst(1'b0), .x, .y,
51              .pixel_color, .pixel_write(1'b1), .dfb_en, .frame_start,
52              .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
53              .VGA_BLANK_N, .VGA_SYNC_N);
54
55      // draw lines between (x0, y0) and (x1, y1)
56      //line_drawer will draw a line between the two
57      //corrdinates
58      line_drawer lines (.clk(CLOCK_50), .reset(1'b0),
59              .x0, .y0, .x1, .y1, .x, .y);
60
61      // draw an arbitrary line
62      assign x0 = 0;
63      assign y0 = 0;
64      assign x1 = 10;
65      assign y1 = 20;
66      assign pixel_color = 1'b1;
67
68  endmodule
```

1.D) DE1_SoC.sv (testbench)

```
70  //DE1_SoC_testbench tests for expected and unexpected behavior.
71  //This testbench just runs the clock for 50 cycles, while the
72  //values for x0, x1, y0, and y1 inside the DE1_SoC module were
73  //set to 0, 0, 10, and 20, respecfully. This testbench checked
74  //for proper updates to 10-bit value x, and 9-bit value y. This
75  //was done to be able to easily see the values updating properly.
76  module DE1_SoC_testbench();
77        logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
78        logic [9:0] LEDR;
79        logic [3:0] KEY;
80        logic [9:0] SW;
81
82        logic CLOCK_50;
83        logic [7:0] VGA_R;
84        logic [7:0] VGA_G;
85        logic [7:0] VGA_B;
86        logic VGA_BLANK_N;
87        logic VGA_CLK;
88        logic VGA_HS;
89        logic VGA_SYNC_N;
90        logic VGA_VS;
91
92
93        DE1_SoC dut(.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW, .CLOCK_50,
94                    .VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N, .VGA_CLK, .VGA_HS, .VGA_SYNC_N, .VGA_VS);
95
96        parameter CLOCK_PERIOD=100;
97        initial begin
98           CLOCK_50 <= 0;
99           forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
100       end
101
102       initial begin
103                           repeat(50)   @(posedge    CLOCK_50);
104          $stop; // End the simulation.
105       end
106 endmodule
```

2.A) line_drawer.sv

```
1   //Garrett Tashiro
2   //November 2, 2021
3   //EE 371
4   //Lab 3, Task 2
5
6   //line_drawer has 1-bit clk, reset, fin, 10-bit x1, x0, 9-bit y1,
7   //and y0 as inputs and returns 1-bit black, 10-bit x, and 9-bit y
8   //as outputs. This module implements Bresenham's line drawing
9   //algorithm in an FSM. The FSM will first draw a line in white
10  //and once the line is drawn, there will be a 1700 clock cycle
11  //delay, then it will set the color to black and draw over the white
12  //line with a black one. After the black line is drawn over the
13  //white line, the 10-bit logic shift will increment by 1 and that
14  //will increment 10-bit logics n0 and n1 which hold the values for
15  //x0 and x1, thus, shifting the line across the screen. This module
16  //is designed to shift a line from left to right across the VGA display.
17  module line_drawer(clk, reset, fin, x0, x1, y0, y1, x, y, black);
18        input logic           clk, reset, fin;
19        input logic [9:0]     x0, x1;
20        input logic [8:0]     y0, y1;
21        output logic [9:0]    x;
22        output logic [8:0]    y;
23        output logic          black;
24
25        //12-bit logic error, and e2 to hold the
26        //error and two times the error value
27        logic signed [11:0] error, e2;
28
29        //10-bit dx and dy to hold the absolute
30        //values for delta x and delta y
31        logic signed [9:0] dx, dy;
32
33        //13-bit count is a counter for when the
34        //white line is drawn
35        logic [10:0] count;
36
37        //10-bit logic n0, and n1 to hold the
38        //new values from the original shifted
39        //value. 10-bit logic shift to increment
40        //n0, and n1 for shifting.
41        logic [9:0] n0, n1, shift;
```

```
42
43          //Assiging dx and dy with conditional operators
44          //to obtain the absolute values of delta x
45          //and delta y.
46          assign dx = (n1 > n0) ? n1 - n0 : n0 - n1;  //Changed x's to n's
47          assign dy = (y1 > y0) ? y1 - y0 : y0 - y1;
48
49          //States for the FSM
50          enum{start, draw, done} state;
51
52          //This always_comb block updates the 12-bit value
53          //e2 whenever error is changed, update n0, and n1
54          //whenever a line is drawn over and needs to be shifted
55          //once shift is updated
56          always_comb begin
57             e2 = 2*error;
58             n0 = (x0 + shift);
59             n1 = (x1 + shift);
60          end
61
```

```
61
62          //This always_ff block has the logic for the FSM. It starts with an
63          //if statement to see if reset is high, or if ~fin. If either of these
64          //are true, 1-bit output blakc will be set to 0, 11-bit count, and 10-bit
65          //shift is set to 0. The state is set to done. Once fin is high and
66          //reset is low, the FSM is in start and 10-bit output x, and 9-bit output
67          //y are assigned. Depending on the input values will determine the next state.
68          //If in the draw state, the line will start to be drawn by increasing, or
69          //decreasing -bit output x, and 9-bit output y, resepectfully. Once the line
70          //is finished and x and y are at the end points, the FSM will go to the done
71          //state, where a white line will be held for 1450 clock cycles, or a black line
72          //will be drawn over the white line by going back to the start state.
73          always_ff @(posedge clk) begin
74             if(reset || ~fin) begin
75                black <= 1'b0;
76                count <= '0;
77                shift <= '0;
78                state <= done;
79             end
80
81             else begin
82                case(state)
83                   start: begin
84                      if((n0 == n1) && (y0 == y1)) begin
85                         x <= x0;
86                         y <= y0;
87                         state <= start;
88                      end
89
90                      else begin
91                         x <= n0;
92                         y <= y0;
93                         error = dx - dy;
94                         state <= draw;
95                      end
96                   end
97
```

```
98                   draw: begin
99                      if((x == n1) && (y == y1)) begin
100                        x <= n1;
101                        y <= y1;
102                        state <= done;
103                     end
104
105                     else begin
106
107                        if(e2 >= -dy) begin
108                           error <= error - dy;
109                           x <= (n1 > n0) ? (x + 1) : (x - 1);
110                        end
111
112                        if(e2 <= dx) begin
113                           error <= error + dx;
114                           y <= (y1 > y0) ? (y + 1) : (y - 1);
115                        end
116
117                        if((e2 >= -dy) && (e2 <= dx)) begin
118                           error <= error - dy + dx;
119                           x <= (n1 > n0) ? (x + 1) : (x - 1);
120                           y <= (y1 > y0) ? (y + 1) : (y - 1);
121                        end
122
123                        state <= draw;
124                     end
125                  end
126
```

```
126 
127                 done: begin
128                    if(~black) begin
129                       black <= 1'b1;
130                       if((n0 == 640) || (n1 == 640)) begin
131                           shift <= '0;
132                       end
133 
134                       else begin
135                           shift <= shift + 1;
136                       end
137                       state <= start;
138                    end
139 
140                    else begin
141                       //if the count is max, then draw a black line
142                       if(count == 11'd1700) begin
143                          black <= 1'b0;
144                          count <= '0;
145                          state <= start;
146                       end
147                       //if count isn't max, stay in done and increase count
148                       else begin
149                          black <= 1'b1;
150                          count <= count + 11'd1;
151                          state <= done;
152                       end
153                    end
154                 end
155              endcase
156           end
157        end
158  endmodule
```

2.B) line_drawer.sv (testbench)

```
160  //line_drawer_testbench tests for expected and unexpected behavior.
161  //For these tests, the value for which count is allowed to count
162  //up to in the done state was changed to 10 just for simplicity.
163  //This testbench first sets values for x0, x1, y0, y1, and then
164  //sets reset high, then low. This first test has x1 and y1 greater than
165  //x0 and y0. This ran with these values for 40 clock cycles to be able
166  //to draw the line in white, hold the white line, and then change the
167  //color to black and draw over it. The second test starts with reset
168  //being set high, then low, and then x0 and y0 are greater than x1 and
169  //y1. The test ran with these values for 40 clock cycles too have the
170  //line draw in white, the go over it black, then shift and draw in white.
171  //the last test was with all the points equal to 0. This was done to see
172  //if it would change from the (0, 0) location.
173  module line_drawer_testbench();
174        logic           clk, reset, black, fin;
175        logic [9:0]     x0, x1;
176        logic [8:0]     y0, y1;
177        logic [9:0]     x;
178        logic [8:0]     y;
179 
180        line_drawer dut(.clk, .reset, .fin, .x0, .x1, .y0, .y1, .x, .y, .black);
181 
182        parameter clk_PERIOD = 100;
183        initial begin
184              clk <= 0;
185              forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
186        end
```

```
187 
188      initial begin
189         x0 <= 9'd0; x1 <= 9'd3; y0 <= 8'd0; y1 <= 8'd3;  repeat(1)   @(posedge    clk);
190         fin <= 1;
191         reset <= 1;                repeat(1)   @(posedge    clk);
192         reset <= 0;                repeat(1)   @(posedge    clk);
193                                    repeat(40)  @(posedge    clk);
194         reset <= 1;                repeat(1)   @(posedge    clk);
195         reset <= 0;                repeat(1)   @(posedge    clk);
196         x0 <= 9'd2; y0 <= 8'd3; x1 <= 9'd0; y1 <= 8'd0;    repeat(1)   @(posedge    clk);
197                                    repeat(40)  @(posedge    clk);
198         fin <= 0;                  repeat(1)   @(posedge    clk);
199         fin <= 1;                  repeat(1)   @(posedge    clk);
200         x0 <= 9'd0; y0 <= 8'd0; x1 <= 9'd0; y1 <= 8'd0;    repeat(6)   @(posedge    clk);
201 
202         $stop; // End the simulation.
203      end
204  endmodule
```

2.C) clr_scr.sv

```
1  //Garrett Tashiro
2  //November 2, 2021
3  //EE 371
4  //Lab 3, Task 2
5
6
7  //clr_scr has 1-bit clk, ans reset as inputs and
8  //returns, 10-bit x, and 9-bit y, 1-bit color,
9  //and done as outputs. Upon reset, 10-bit x,
10 //9-bit y, 1-bit color, and done are set to 0. On
11 //the next clock cycle y in incremented every clock
12 //cycle until it is 480. Once at 480 it is set to 0
13 //and x is incremented. This process goes on until each
14 //pixel has been gone over to color them black.
15 module clr_scr(clk, reset, x, y, color, done);
16       input logic          clk, reset;
17       output logic [9:0]   x;
18       output logic [8:0]   y;
19       output logic         color, done;
20
21       //This always_ff block checks if reset is 0 or 1.
22       //If it is 0 the 10-bit x, 9-bit y, 1-bit color, and
23       //done are all set to 0. The following clock cycles
24       //x and y are incremented accordingly until x is 641.
25       //At this point color and done are both set to 1.
26       always_ff @(posedge clk) begin
27          if(reset) begin
28             x <= '0;
29             y <= '0;
30             color <= 1'b0;
31             done <= 1'b0;
32          end
33
34          else begin
35             if(x < 641) begin
36                if(y < 480) begin
37 //             if(x < 6) begin
38 //                if(y < 4) begin
39                      y <= y + 1;
40                end
41
42             else begin
43                x <= x + 1;
44                y <= 0;
45             end
46          end
47
48          if(x >= 641) begin
49 //          if(x >= 6) begin
50             color <= 1'b1;
51             done <= 1'b1;
52          end
53       end
54    end
55 endmodule
```

2.D) clr_scr.sv (testbench)

```
56
57    //clr_scr_testbench tests for expected and unexpected behavior.
58    //This test was done when in the always_ff x was capped at 6
59    //and y was capped at 4 just to have a similar test. The test
60    //sets reset high and then low. Reset stays low for 35 cycles
61    //to have x increment to 6 and for y to increment repeatedly
62    //from 0 to 4 until x hit its max and both color and done go high.
63    module clr_scr_testbench();
64          logic           clk, reset, color, done;
65          logic [9:0]     x;
66          logic [8:0]     y;
67
68          clr_scr dut (.clk, .reset, .x, .y, .color, .done);
69
70          parameter clk_PERIOD = 100;
71          initial begin
72                clk <= 0;
73                forever #(clk_PERIOD/2) clk <= ~clk;// Forever toggle the clk
74          end
75
76          initial begin
77             reset <= 1;       repeat(3)   @(posedge    clk);
78             reset <= 0;       repeat(35)   @(posedge    clk);
79
80             $stop; // End the simulation.
81          end
82    endmodule
```

2.E) clock_divider.sv

```
1     //Garrett Tashiro
2     //October 10, 2021
3     //EE 371
4     //Lab 2, Task 2.4
5
6     //clock_divider has 1-bit clock and 1-bit reset as inputs and returns
7     //divided_clocks as an output. This module allows you to change the
8     //clock rate of CLOCK_50 in order to be able to have HEX displays
9     //update roughly every second. This is a module written in 271.
10
11    // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
12    module clock_divider (clock, reset, divided_clocks);
13            input  logic          reset, clock;
14            output logic  [31:0]  divided_clocks = 0;
15
16            always_ff @(posedge clock) begin
17               divided_clocks <= divided_clocks + 1;
18            end
19    endmodule
20
```

2.F) clock_divider.sv (testbench)

```
20
21    //clock_divider_testbench tests for expected and unexpected behavior.
22    //This testbench resets and just runs for 100 clock cycles
23    module clock_divider_testbench();
24          logic           reset, clock;
25          logic [31:0]    divided_clocks;
26
27          clock_divider dut(.clock, .reset, .divided_clocks);
28
29          parameter clk_PERIOD = 100;
30          initial begin
31                clock <= 0;
32                forever #(clk_PERIOD/2) clock <= ~clock;// Forever toggle the clk
33          end
34
35          initial begin
36             reset <= 1;     repeat(1)   @(posedge    clock);
37             reset <= 0;     repeat(1)   @(posedge    clock);
38                             repeat(100)   @(posedge    clock);
39             $stop; // End the simulation.
40          end
41    endmodule
```

2.G) DE1_SoC.sv

```
1   //Garrett Tashiro
2   //November 3, 2021
3   //EE 371
4   //Lab 3, Task 2
5
6
7   //DE1_SoC module is the top level module for Lab 3, Task 2. This module implements
8   //a line being drawn, and shifted across a VGA screen. The module uses hierarchical
9   //calls to clock_divider, VGA_framebuffer, line_drawer, and clr_scr. The DE1_SoC
10  //module only takes one input from SW[0], and taht is connected to reset. Upon startup,
11  //the system will start by drawing a white line on the left hand side of the VGA display
12  //and after some clock cycles the white line is drawn over with a black one, and the
13  //line is shifted to the right by one pixel. The line will go off the far right side and
14  //come back to the left side and continue the process of shifting a line acoss the VGA
15  //display. Upon reset, clr_scr's values for x and y are used, along with its color. The
16  //modle will start in the top left corner of the screen and go pixel by pixel changing
17  //the color to black and clearing the screen.
18  module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
19      VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS);
20
21      output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
22      output logic [9:0] LEDR;
23      input logic [3:0] KEY;
24      input logic [9:0] SW;
25
26      input CLOCK_50;
27      output [7:0] VGA_R;
28      output [7:0] VGA_G;
29      output [7:0] VGA_B;
30      output VGA_BLANK_N;
31      output VGA_CLK;
32      output VGA_HS;
33      output VGA_SYNC_N;
34      output VGA_VS;
35
36      assign HEX0 = '1;
37      assign HEX1 = '1;
38      assign HEX2 = '1;
39      assign HEX3 = '1;
40      assign HEX4 = '1;
41      assign HEX5 = '1;
42      assign LEDR = SW;
43
44      //Added logic xl, and xc to hold outputs from
45      //line_drawer and clr_scr
46      logic [9:0] x0, x1, x, xl, xc;
47
48      //Added logic yl, and yc to hold outputs from
49      //line_drawer and clr_scr
50      logic [8:0] y0, y1, y, yl, yc;
51      logic frame_start;
52
53      //added logic c_color, l_color, and finished
54      //to take outputs from line_drawer and clr_scr
55      logic pixel_color, c_color, l_color, finished;
56
57      //Logic reset to hold a the value from SW[0]
58      //to be the reset for the system.
59      logic reset;
60      assign reset = SW[0];
61
62      //32-bit logic to hold output from clock_divider
63      logic [31:0] div_clk;
64
65      //clock_divider has 1-bit CLOCK_50, and reset as inputs and returns 1-bit
66      //div_clk as an output.This module divides the clock to lower the frequency
67      //of CLOCK_50
68      clock_divider oneSec(.clock(CLOCK_50), .reset(reset), .divided_clocks(div_clk));
69
70      //1-bit logic clk for the clock on board or during simulation
71      logic clk;
72      assign clk = CLOCK_50;      // for simulation
73      //assign clk = div_clk[9];  // for board
74
75      //////// DOUBLE_FRAME_BUFFER ////////
76      logic dfb_en;
77      assign dfb_en = 1'b0;
78      ////////////////////////////////////////
79
80      //VGA_framebuffer is given to us
81      VGA_framebuffer fb(.clk(CLOCK_50), .rst(1'b0), .x, .y,
82              .pixel_color, .pixel_write(1'b1), .dfb_en, .frame_start,
83              .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
84              .VGA_BLANK_N, .VGA_SYNC_N);
85
86      //line_drawer will draw a line between (x0, y0) and (x1, y1).
87      //The white line will stay on the screen for 1500 clock cycles
88      //before drawing a black line over the white one, and then
89      //the a variable shift will increase and new, shifted x0 and x1
90      //values are used to shift the line across the screen.
91      line_drawer lines (.clk(clk), .reset(reset), .fin(finished),
92              .x0, .y0, .x1, .y1, .x(xl), .y(yl), .black(l_color));
93
```

```
    //clr_scr clear takes 1-bit CLOCK_50, and reset as inputs and returns
    //10-bit xc, 9-bit yc, 1-bit c_color, and done as outputs. The output
    //xc is used for assigning x conditionally, yc is used for assigning
    //y conditionally, c_color is used for assigning pixel_color conditionally
    //and finished is used as the condition in which the assignments are
    //based upon.
    clr_scr clear(.clk(CLOCK_50), .reset(reset), .x(xc), .y(yc), .color(c_color), .done(finished));

    //Conditional operator to assign x. If 1-bit finished being
    //passed from clr_scr is true, then the x value output from
    //line_drawer will be used. If finished is false, the x value
    //output from clr_scr will be used.
    assign x = (finished) ? xl : xc;

    //Conditional operator to assign y. If 1-bit finished being
    //passed from clr_scr is true, then the y value output from
    //line_drawer will be used. If finished is false, the y value
    //output from clr_scr will be used.
    assign y = (finished) ? yl : yc;

    //Conditional operator to assign pixel_color. If 1-bit finished
    //is true, then use the 1-bit l_color from output black in
    //line_drawer. If it is false, use 1-bit c_coloc from output
    //color of clr_scr.
    assign pixel_color = (finished) ? l_color : c_color;

    // draw an arbitrary line
    assign x0 = 0;
    assign y0 = 100;
    assign x1 = 0;
    assign y1 = 340;
endmodule
```

2.H) DE1_SoC.sv (testbench)

```
//DE1_SoC_testbench tests for expected and unexpected behavior. DE1_SoC
//only uses SW[0] as an input to control the system. SW[0] is linked
//to reset. The test first starts by setting SW[0] high for a cycle,
//then low for 75 cycles. The values in clr_scr were lowered to a max
//x value of 60 and a max y vlue of 40. The line being tested upon is
//x0 = 0, y0 = 0, x1 = 0, and y1 = 10. The line_drawer module pasues for 10
//cycles for these tests as well. The test is ran twice to insure
//that the system could reset after drawing lines. These values were used to create
//a smaller testbench to view the behavior of the DE1_SoC and all the
//other modules working together correctly.
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; // Forever toggle the clock
      end

      initial begin
         SW[0] <= 1;       repeat(1)   @(posedge    CLOCK_50);
         SW[0] <= 0;       repeat(75)   @(posedge    CLOCK_50);
         SW[0] <= 1;       repeat(1)   @(posedge    CLOCK_50);
         SW[0] <= 0;       repeat(75)   @(posedge    CLOCK_50);
         $stop; // End the simulation.
      end
```

3.A) VGA_framebuffer.sv

This module was given to us. It was unchanged, and used in both task 1, and task 2.

```
// 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;

```

```
77      // Double-buffering.
78      logic buffer[640*480*2-1:0];
79      logic [19:0] wr_addr, rd_addr;
80      logic rd_data;
81
82      assign wr_addr = {y * 19'd640 + x, (!front_odd & dfb_en)};
83      assign rd_addr = {v_count * 19'd640 + (h_count / 19'd2), (front_odd & dfb_en)};
84
85      always_ff @(posedge clk) begin
86          if (pixel_write) buffer[wr_addr] <= pixel_color;
87          if (VGA_CLK) begin
88              rd_data <= buffer[rd_addr];
89              VGA_BLANK_N <= ~blank;
90          end
91      end
92
93      // Color output.
94      assign {VGA_R, VGA_G, VGA_B} = rd_data ? 24'hFFFFFF : 24'h000000;
95  endmodule
```