Cynthia Li
EE 371
May 5, 2021

Lab 2 Report

Procedure

This lab mainly involves two tasks: 1) understand the basic use of VGA, follow Bresenham's algorithm to write systemVerilog code that can draw a line composed of pixels nearest to the mathematically calculated line onto VGA when two endpoints are specified; use modelsim for simulation and test it on FPGA board; 2) make sure that the code in task 1 can draw line with any slope and implement a clear function to clear the screen of FPGA.

Task1: write code that can approximate (round) each point on the mathematically calculated line between two given endpoints to integer coordinates using signed numbers. Specifies the endpoints for the line_drawer to draw and specifies the drawing action with a user control (SW/KEY). Wire the line_drawer, the endpoints instructions (lines.sv), the VGA_framebuffer together in the top module.

Task 2: This task's main function is to implement the clear function and ensure that the code in task 1 can draw a line with any slope. First, I personally chose to implement the clear function by clearing the line one by one (draw a line first, clear it, then move to the next), which only requires using a register to update the clear status and pass it into VGA_framebuffer. Second, besides the fast clock used to draw the line, a slower clock is needed to prevent clear function and drawing happening too fast on VGA board for the demonstration purpose.

Result:

Task 1:

I first implemented the line drawer that is using a FSM to update the coordinates of a moving point on the line between the two given endpoints' coordinates. It produces waveforms as below in Fig.1-4.

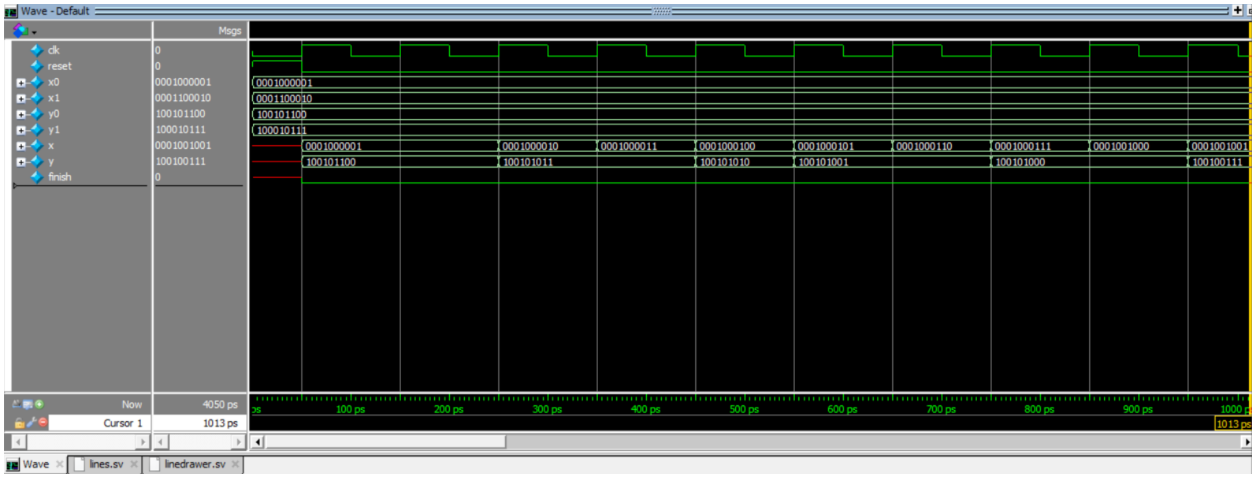

Fig. 1. line_drawer_testbench simulation waveform (part1)

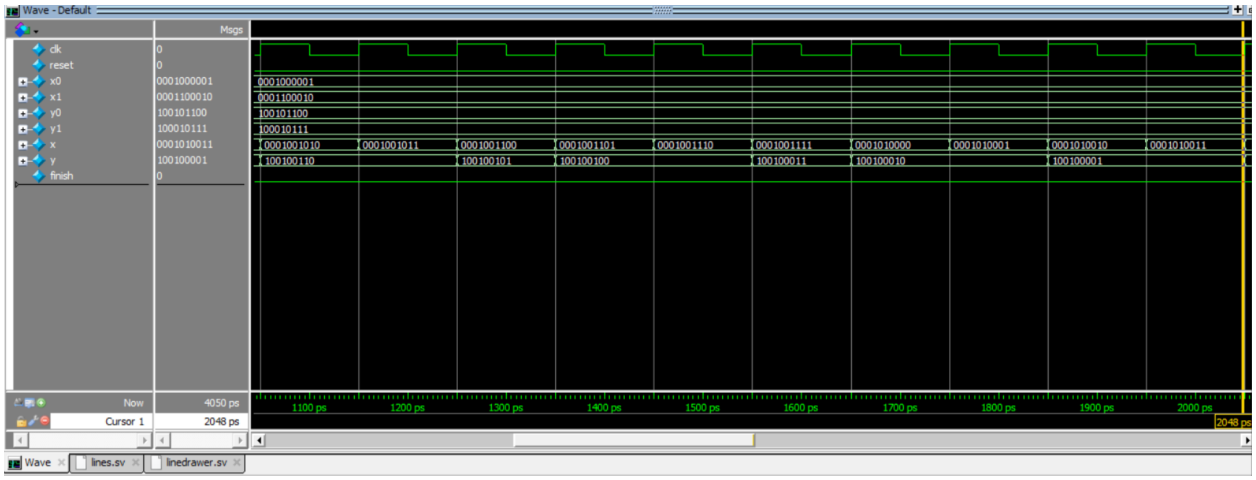

Fig. 2. line_drawer_testbench simulation waveform (part2)

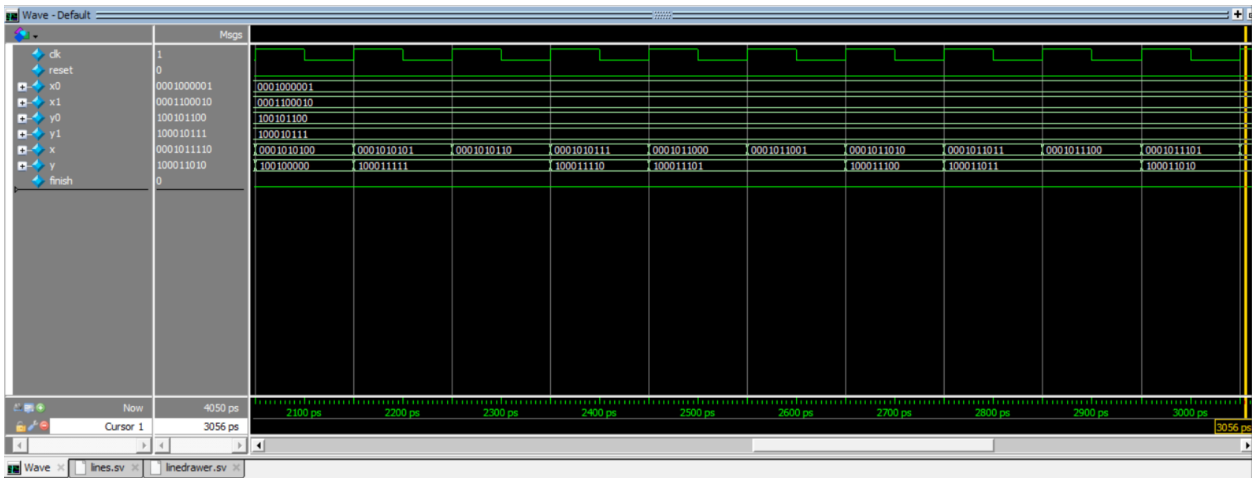

Fig. 3. line_drawer_testbench simulation waveform (part3)

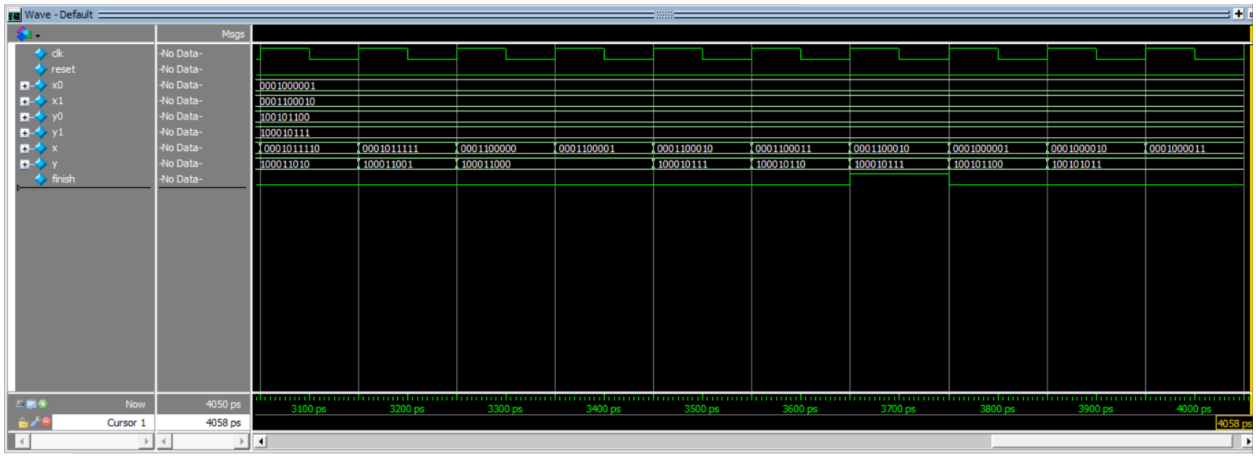

Fig. 4. line_drawer_testbench simulation waveform (part4)

For this part, I simply tested whether the line_drawer can update the correct x/y coordinates for nearest pixels, and whether the status 'finish' updates when the drawing action ends for each pair. I randomly picked 4 coordinates representing the two endpoints that form a smooth diagonal. I expect to see that x and y gradually increment from x0, y0 to x1, y1, that x changes increment more frequently than y (since the difference in x-coordinate is larger), and that finish turns true when it reaches the end. From Fig. 1-4, we can see that it works as expected.

Then, to provide instructions for different endpoints, I implemented a FSM which has 8 states, each state specifies different endpoint pair's coordinates. It produces waveforms as below in Fig.5-7.

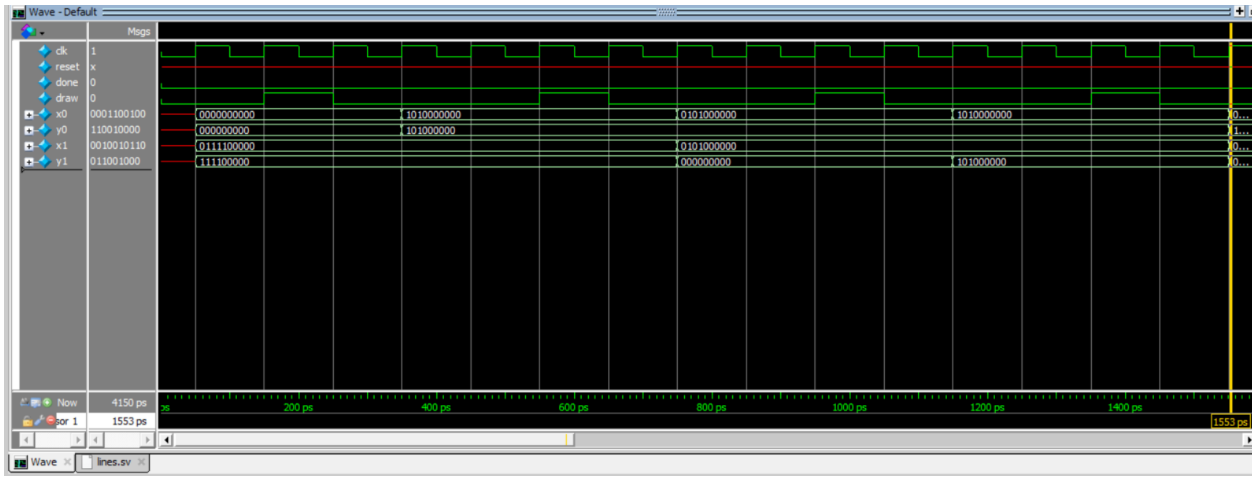

Fig. 5. lines_testbench simulation waveform (part1)

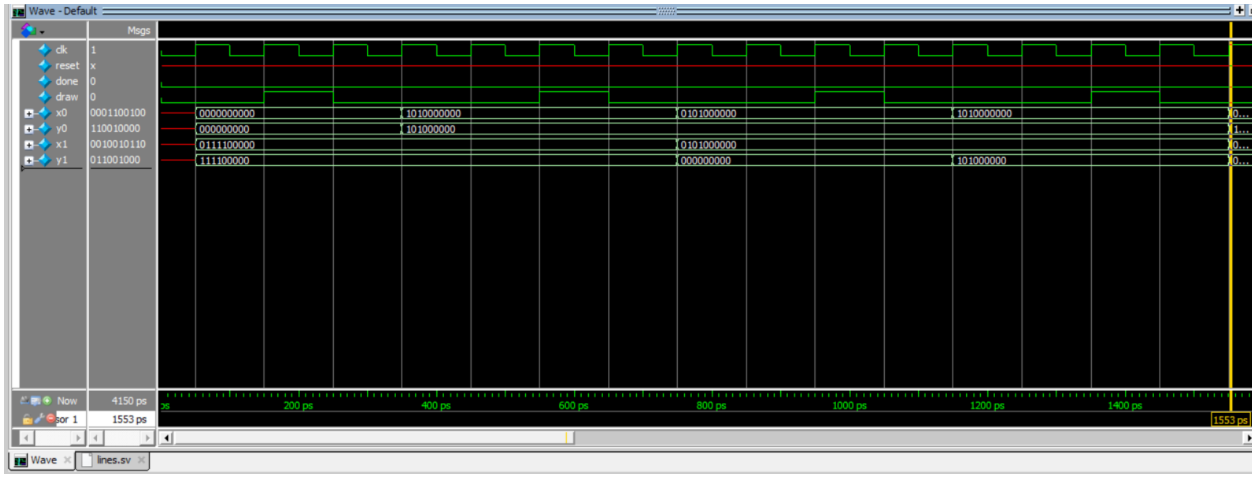

Fig. 6. lines_testbench simulation waveform (part2)

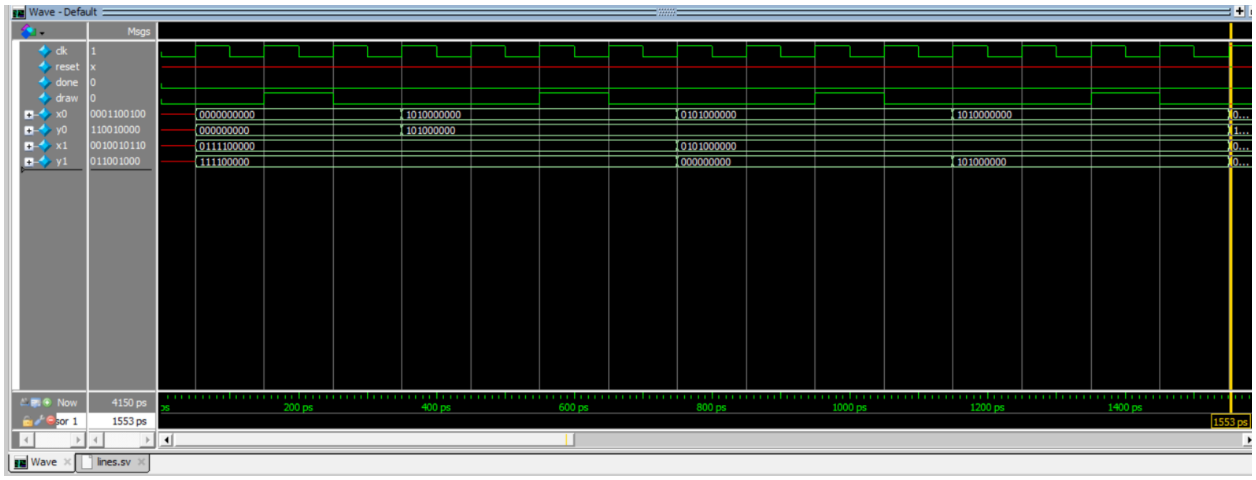

Fig. 7. lines_testbench simulation waveform (part3)

Here I simply tested when the drawing action is triggered (draw is true), whether the four coordinates x0, x1, y0, y1 updates correctly. I expect to see after each time the ‘draw’ turns true, the x0, x1, y0, y1 will become the binary value that corresponds to the endpoints of the next state, also expect to see that the sequence of the next state is correctly showing up. From Fig. 5-7. We can see that the x0, x1, y0, y1 are updating in the correct sequence and as the correct value of the endpoints. Thus, it works correctly.

Since I am using a fast clock, I would like to make my press, which is long compared to the clock cycle, true for only one cycle to trigger the drawing action. Thus, I am using the KEY processor previously used in EE 271, and 371 lab2. It produces waveforms as below in Fig. 8. (as shown in the previous lab as well).

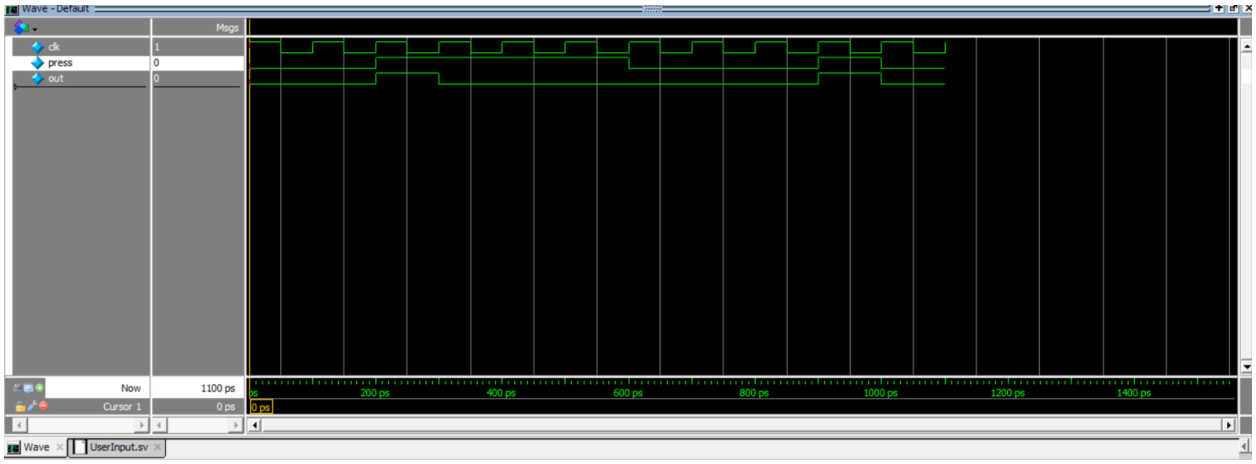

Fig. 8. UserInput_testbench simulation waveform

I am testing cases when the key is not being pressed, when there is a long press that lasts for many clock cycles, when there is a short press that lasts for only one clock cycle. I expect to see that the 'out' is only true for one cycle whenever the press turns true, but stay false otherwise. From Fig. 8, we can see it function properly.

Last I wired the VGA buffer, and the three modules above in the top module DE1_SoC. However, the most important output in VGA is hard for us to understand, and it doesn't provide much useful information. So I am simply running the simulation to see that the user control works, and the VGA will turn on correspondingly. Thus, it produces a kind of useless waveform as shown in Fig. 9. However, since each individual module works fine, and there is no complicated wiring in this module, it should be fine.

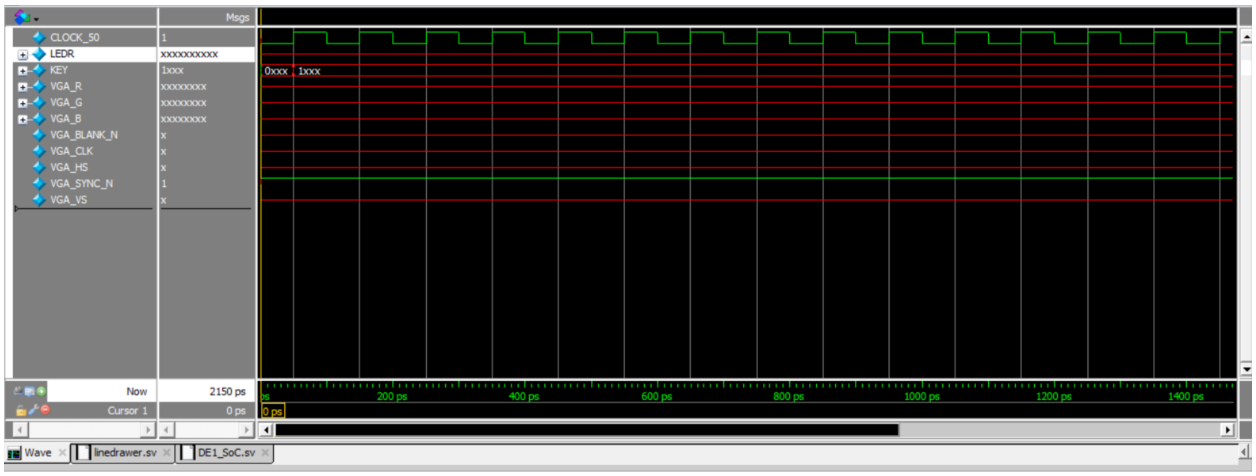

Fig.9. DE1_SoC_testbench simulation waveform

The overall functionality can be referred to demo_task1. In all, we have built up the basics of a Bresenham's line drawer and displayed it on VGA, and can say that it functions as expected and as described in the lab instructions.

Below is the block diagram.

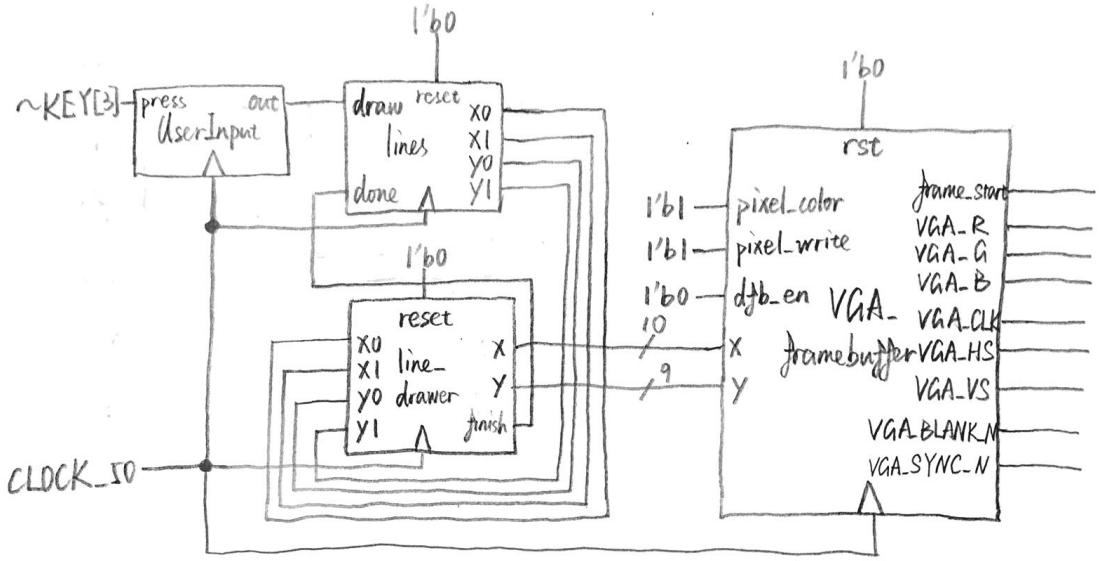

Fig. 10. Task 1 top block diagram

Task 2:

I started with the easier task which is to generate a slower clock. I used the clock divider provided by Professor Hauck in EE 271. This is also the same code used in Lab 2. In this case, there is no testbench or simulation for this part. I selected the 6Hz clock (refer to APPENDIX.

Task 2. a)

Then I implemented the clear function by using registers to keep track of whether clear action is needed, the pixel_color, etc. However, since these all happened in the top module, where the inputs and outputs are complicated to understand or not very informative, the simulation (as in Fig.11) are not very useful.

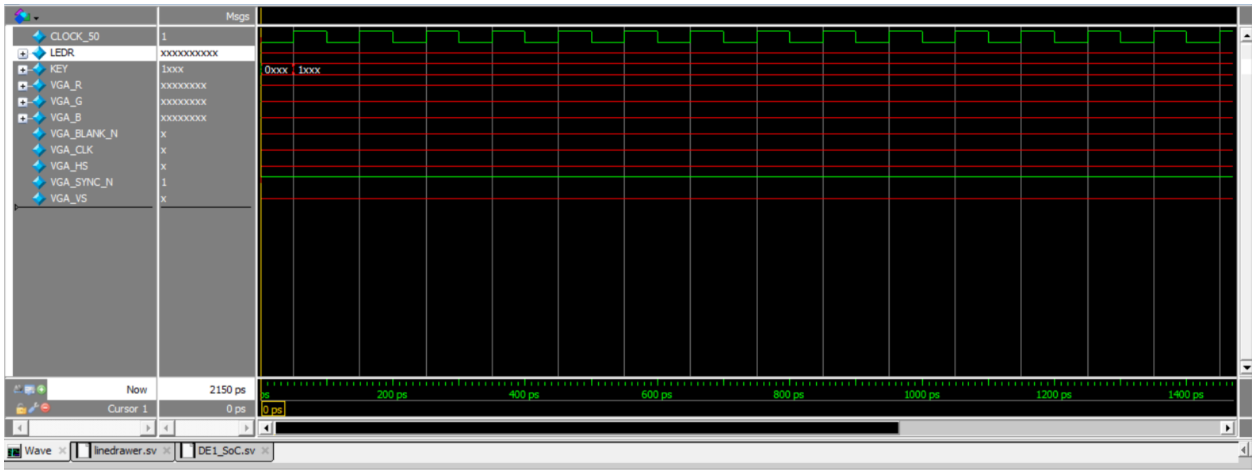

Fig.11. task2_testbench simulation waveform

Below is the synthesis report.

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits
1	|task2	395 (25)	113 (7)	614400
1	|VGA_framebuffer:fb|	182 (66)	29 (22)	614400
1	|altsyncram...ffer_rtl_0|	116 (0)	7 (0)	614400
1	|altsyncr...enerated|	116 (0)	7 (7)	614400
1	|decode...ecode2|	90 (90)	0 (0)	0
2	|mux_ehb:mux3|	26 (26)	0 (0)	0
2	|clock_divider:cdiv|	23 (23)	23 (23)	0
3	|line_drawer:lines|	165 (165)	54 (54)	0

Fig. 12 Task2 synthesis report

Conclusion: Overall, this lab extended my knowledge of memory and buffers and familiarized me with manipulating the video output of the DE1-SoC. It seem have room for improvement such as in the clear function, or find other ways to divide task2 into smaller modules for simulation. But the functionality we looked for are basically met.

APPENDIX:

Task1:

a. line_drawer.sv

```
// Cynthia Li
// EE 371 Lab3
//
// The line_drawer module takes in four 10 or 9 bit value that represent the x/y coordinates
// of two points to draw a line between the two point. It will output a 9-bit and a 10-bit
// value representing the current coordinate it's examing on the line, and output the status
// of whether the line has been finished drawing.
module line_drawer(
    input logic clk, reset,
    // x and y coordinates for the start and end points of the line
    input logic [9:0] x0, x1,
    input logic [8:0] y0, y1,
    //output x, y coordinate that represent the current point examined
    output logic [9:0] x,
    output logic [8:0] y,
    output logic finish
    );

    // the following are registers used to keep track of things
    // dx, dy keep track of the |difference in x| and -|difference in y|
    // error and temp_error keep track of the diference between x change and y change
    // to determine whether to increment x or y
    logic signed [10:0] dx, dy, error, temp_error;
    logic x_dir, y_dir;

    enum {start, draw, done} ps, ns;

    // the following specifies the next state logic
    always_comb begin
       case (ps)
          start:   ns = draw;
          draw: if ((x == x1) && (y == y1))
                   ns = done;
                else
                   ns = draw;
          done: ns = start;
          default: ns = start;
       endcase
    end

    // the following specifies the FSM behavior
    always_ff @(posedge clk) begin
       case (ps)
          start:   begin
                   // determines y change and y direction
                   dy = y1 - y0;
                   // y_dir positive for down, negative for up
                   y_dir = dy >= 0;
                   if (y_dir)
                      dy = -dy;
```

```
51
52                      // determines x change and x direction
53                      dx = x1 - x0;
54                      // x_dir positive for right, negative for left
55                      x_dir = dx >= 0;
56                      if (~x_dir)
57                          dx = -dx;
58
59                      error = dx + dy;
60
61                      x <= x0;
62                      y <= y0;
63                      finish <= 0;
64                  end
65              draw: begin
66                      temp_error = error << 1;
67
68                      // check if x need update
69                      if ((temp_error > dy)) begin
70                          error += dy;
71                          if (x_dir)
72                              x <= x + 1;
73                          else
74                              x <= x - 1;
75                      end
76
77                      // check if y need update
78                      if ((temp_error < dx)) begin
79                          error += dx;
80                          if (y_dir)
81                              y <= y + 1;
82                          else
83                              y <= y - 1;
84                      end
85                  end
86              done: begin
87                      finish <= 1;
88                      x <= x1;
89                      y <= y1;
90                  end
91              default: begin
92                      x <= x0;
93                      y <= y0;
94                      finish <= 0;
95                  end
96          endcase
97      end
98
```

```
 98
 99      // the following is a DFF used to update present state
100      always_ff @(posedge clk) begin
101         if (reset)
102            ps <= start;
103         else
104            ps <= ns;
105      end
106   endmodule
107
108   module line_drawer_testbench();
109      logic [9:0] x0, x1, x;
110      logic [8:0] y0, y1, y;
111      logic clk, reset, finish;
112
113      line_drawer dut(.*);
114
115      // Set up the clock.
116      parameter CLOCK_PERIOD = 100;
117      initial begin
118         clk <= 0;
119         forever #(CLOCK_PERIOD/2) clk <= ~clk;
120      end
121
122      // test whether a line can be drawn between two given points and whether
123      // 'finish' will be updated.
124      initial begin
125         x0 <= 65; y0 <= 300; x1 <= 98; y1 <= 279; @(posedge clk);
126                                                   repeat(40) @(posedge clk);
127         $stop;
128      end
129   endmodule
130
```

b. lines.sv

```
// Cynthia Li
// EE 371 Lab 3
//
// This module takes in a 1-bit value 'draw' and output four 10 or 9 bit value
// that represent the x/y coordinates of two points of a deliberately selected
// line. It uses a FSM to pass the endpoints for 8 lines: diagonal slope 1,
// diagonal slope -1, vertical, horizontal, steep diagonal, smooth diagonal.
module lines (
    input logic clk, reset, done, draw,
    output logic [9:0] x0, x1,
    output logic [8:0] y0, y1
);

    logic [9:0] next_x0, next_x1;
    logic [8:0] next_y0, next_y1;

    enum {line_1, line_2, line_3, line_4, line_5,line_6, line_7, line_8} ps, ns;

    // the following is the next state and output logic
    // each case specifies a line with a special slope
    always_comb begin
        case (ps)
        line_1: begin // diagonal '\' with slope 1
                if (draw) ns = line_2;
                else      ns = ps;
                next_x0 = 0;
                next_y0 = 0;
                next_x1 = 480;
                next_y1 = 480;

        end
        line_2: begin // diagonal '/' with slope -1
                if (draw) ns = line_3;
                else      ns = ps;
                next_x0 = 640;
                next_y0 = 320;
                next_x1 = 480;
                next_y1 = 480;
        end
        line_3: begin // vertical
                if (draw) ns = line_4;
                else      ns = ps;
                next_x0 = 320;
                next_y0 = 320;
                next_x1 = 320;
                next_y1 = 0;
        end
        line_4: begin // horiztonal
                if (draw) ns = line_5;
                else      ns = ps;
                next_x0 = 640;
```

```
52                  next_y0 = 320;
53                  next_x1 = 320;
54                  next_y1 = 320;
55          end
56          line_5: begin // steep '/' slope > 1
57                  if (draw) ns = line_6;
58                  else      ns = ps;
59                  next_x0 = 100;
60                  next_y0 = 400;
61                  next_x1 = 150;
62                  next_y1 = 200;
63          end
64          line_6: begin // steep'\' slope < -1
65                 if (draw) ns = line_7;
66                 else      ns = ps;
67                 next_x0 = 100;
68                 next_y0 = 200;
69                 next_x1 = 150;
70                 next_y1 = 400;
71          end
72          line_7: begin // smooth '/' slope < 1
73                  if (draw) ns = line_8;
74                  else      ns = ps;
75                  next_x0 = 450;
76                  next_y0 = 100;
77                  next_x1 = 600;
78                  next_y1 = 150;
79          end
80          line_8: begin // smooth '/' slope > -1
81                  if (draw) ns = line_1;
82                  else      ns = ps;
83                  next_x0 = 450;
84                  next_y0 = 150;
85                  next_x1 = 600;
86                  next_y1 = 100;
87          end
88          default: begin
89                  ns = line_1;
90                  next_x0 = 0;
91                  next_y0 = 0;
92                  next_x1 = 480;
93                  next_y1 = 480;
94          end
95          endcase
96      end
97
98      // the following is the DFF that update present state
99      // and the four coordinates
```

```
100     always_ff @(posedge clk) begin
101        if (reset) begin
102           ps <= line_1;
103           x0 <= 0;
104           y0 <= 0;
105           x1 <= 480;
106           y1 <= 480;
107           end
108        else begin
109           ps <= ns;
110           x0 <= next_x0;
111           y0 <= next_y0;
112           x1 <= next_x1;
113           y1 <= next_y1;
114           end
115     end
116  endmodule
117
118  module lines_testbench();
119     logic clk, reset, done, draw;
120     logic [9:0] x0, x1;
121     logic [8:0] y0, y1;
122
123     lines dut (.*);
124
125     // set up the clock
126     parameter CLK_Period = 100;
127     initial begin
128        clk <= 1'b0;
129        forever #(CLK_Period/2) clk <= ~clk;
130     end
131
132     initial begin
133     draw <= 0; done <= 0;       repeat(2)   @(posedge clk); // test default
134     draw <= 1; done <= 0;                   @(posedge clk); // test draw line 1
135     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
136     draw <= 1; done <= 0;                   @(posedge clk); // test draw line 2
137     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
138     draw <= 1; done <= 0;                   @(posedge clk); // test draw line 3
139     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
140     draw <= 1; done <= 0;                   @(posedge clk); // test draw line 4
141     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
142     draw <= 1; done <= 0;                   @(posedge clk); // test draw line 5
143     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
144     draw <= 1; done <= 0;                   @(posedge clk); // test draw line 6
145     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
146     draw <= 1; done <= 0;                   @(posedge clk); // test draw line 7
147     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
148     draw <= 1; done <= 0;                   @(posedge clk); // test draw line 8
149     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
150     draw <= 1; done <= 0;                   @(posedge clk); // test go back
151     draw <= 0; done <= 0;       repeat(3)   @(posedge clk);
152     draw <= 1; done <= 1;                   @(posedge clk); // test both
153     draw <= 0; done <= 1;       repeat(3)   @(posedge clk); // test done
154     $stop;
155     end
156  endmodule
```

c. UserInput.sv

```
1  // Cynthia Li
2  // 4/20/21
3  // EE 371 LAB3
4
5  // Module userInput takes a input press that indicates the status
6  // of a KEY, and should process it so that every press is true for
7  // only one cycle and otherwise stays false. It will output this
8  // processed KEY status to out.
9  module UserInput (clk, press, out);
10    input logic clk, press;
11    // press: true when the KEY is pressed
12    output logic out;
13
14    enum {on, off} ps, ns;
15
16    // the next state and output logic
17    always_comb
18       case(ps)
19          on: if (press) begin // key contiuously being pressed
20                   ns = on;    // & out has been true for 1 cycle
21                   out = 0;
22                end
23             else begin        // key is released
24                   ns = off;
25                   out= 0;
26                end
27          off:  if (press) begin  // key is pressed (the action)
28                      ns = on;    // make out true for one cycle
29                      out= 1;
30                   end
31                else begin        // key is not pressed
32                      ns = off;
33                      out = 0;
34                   end
35          default: begin
36                      ns = off;
37                      out = 0;
38                   end
39       endcase
40
41    always_ff @(posedge clk)
42       ps <= ns;
43
44 endmodule
45
46 module UserInput_testbench();
47    logic clk, press;
48    logic out;
49
50    UserInput dut (.clk, .press, .out);
51
```

```
52    // Set up the clock.
53    parameter CLOCK_PERIOD=100;
54    initial clk=1;
55    always begin
56       #(CLOCK_PERIOD/2);
57       clk = ~clk;
58    end
59
60    // Set up the inputs to the design. Each line is a clock cycle.
61    initial begin
62    press <= 0;   repeat(2) @(posedge clk);
63    press <= 1;   repeat(4) @(posedge clk);
64    press <= 0;   repeat(3) @(posedge clk);
65    press <= 1;           @(posedge clk);
66    press <= 0;           @(posedge clk);
67    $stop; // End the simulation.
68    end
69 endmodule
```

d. DE1_Soc.sv

```
// Cynthia Li
// EE 371

// This is the top module that uses KEY[3] as the user input to initizae the action to draw
// the 8 internally choosen lines and output the updated image to VGA. It follows the Bresenham's
// line drawing algorithm to determine the nearest pixel of the lines.
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;

    // the following turns off the HEX display
    assign HEX0 = '1;
    assign HEX1 = '1;
    assign HEX2 = '1;
    assign HEX3 = '1;
    assign HEX4 = '1;
    assign HEX5 = '1;
    assign LEDR = SW;

    logic pixel_color, frame_start;
    assign pixel_corlor = 1'b1;

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

    // the following instantiates the VGA frame buffer
    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);

    logic draw, finish;

    // the following process the user input to ensure each press is true for only one cycle
    UserInput process_KEY (.clk(CLOCK_50), .press(~KEY[3]), .out(draw));
```

```
52
53      // the following instantiates the line drawer
54      line_drawer lines (.clk(CLOCK_50), .reset(1'b0), .x0, .x1, .y0, .y1, .x, .y, .finish);
55
56      // the following instantiates the FSM that sets the endpoints of lines
57      lines drawing (.clk(CLOCK_50), .reset(1'b0), .done(finish), .draw, .x0, .x1, .y0, .y1);
58  endmodule
59
60  module DE1_SoC_testbench();
61      logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
62      logic [9:0] LEDR;
63      logic [3:0] KEY;
64      logic [9:0] SW;
65      logic CLOCK_50;
66      logic [7:0] VGA_R;
67      logic [7:0] VGA_G;
68      logic [7:0] VGA_B;
69      logic VGA_BLANK_N;
70      logic VGA_CLK;
71      logic VGA_HS;
72      logic VGA_SYNC_N;
73      logic VGA_VS;
74
75      DE1_SoC dut(.*);
76
77      // Simulate clock
78      parameter CLOCK_PERIOD = 100;
79      initial begin
80          CLOCK_50 <= 0;
81          forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;
82      end
83
84      // there is no real useful information in the tb
85      // simply to see that input works
86      initial begin
87          KEY[3] <= 1'b0; @(posedge CLOCK_50);
88          KEY[3] <= 1'b1; @(posedge CLOCK_50);
89          for (int i = 0; i < 20; i++) begin
90                          @(posedge CLOCK_50);
91          end
92          $stop;
93      end
94  endmodule
```

e. VGA_framebuffer.sv

```
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.
```

```
53      assign end_of_field = v_count == VTOTAL - 1;
54      assign frame_start = !h_count && !v_count;
55
56      always_ff @(posedge clk)
57         if (rst) begin
58            v_count <= 0;
59            front_odd <= 0;
60         end else if (end_of_line)
61            if (end_of_field) begin
62               v_count <= 0;
63               front_odd <= !front_odd;
64            end else
65               v_count <= v_count + 10'd1;
66
67      // Sync signals.
68      assign VGA_CLK = h_count[0]; // 25 MHz clock: pixel latched on rising edge.
69      assign VGA_HS = !(h_count - (HPX + HFP) < HSP);
70      assign VGA_VS = !(v_count - (VLN + VFP) < VSP);
71      assign VGA_SYNC_N = 1; // Unused by VGA
72
73      // Blank area signal.
74      logic blank;
75      assign blank = h_count >= HPX || v_count >= VLN;
76
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
96
```

Task2:

a. task2.sv

```
// Cynthia Li
// EE 371 TASK2

// This is the top module that uses KEY[3] as the user input to clear the VGA display
// It displays lines with different slope on VGA that look like a loading clock (so i
// closewise direction). It is a extension of Task1 as it futher implemented a clear
// function that clears a line after finish drawing.
module task2 (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;

    // the following turns off the HEX display
    assign HEX0 = '1;
    assign HEX1 = '1;
    assign HEX2 = '1;
    assign HEX3 = '1;
    assign HEX4 = '1;
    assign HEX5 = '1;
    assign LEDR = SW;

    logic [9:0] x0, x1, x;
    logic [8:0] y0, y1, y;
    logic draw, finish;

    logic pixel_color, frame_start;
    initial pixel_color <= 1'b1;

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

    // the following instantiates the VGA frame buffer
```

```
48  VGA_framebuffer fb(.clk(CLOCK_50), .rst(clear), .x, .y,
49            .pixel_color, .pixel_write(1'b1), .dfb_en, .frame_start,
50            .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
51            .VGA_BLANK_N, .VGA_SYNC_N);
52
53  logic reset, clear;
54
55  // initialize the local parameters
56  initial begin
57     clear <= 1'b0;
58     reset <= 1'b0;
59  end
60
61  // the following set a fixed end point
62  assign x0 = 320;
63  assign y0 = 240;
64
65  // the following uses two arrays to store the x/y coordinate of the second endpoint used for animation
66  int x_vals [15:0];
67  int y_vals [15:0];
68  // the following will form lines with (x0,y0) in closewise sequence.
69  assign x_vals = '{300, 260, 220, 180, 220, 260, 300, 320, 340, 380, 420, 460, 420, 380, 340, 320};
70  assign y_vals = '{125, 180, 220, 240, 260, 300, 355, 380, 355, 300, 260, 240, 220, 180, 125, 100};
71
72  // i keeps track of the second endpoint currently using
73  logic [3:0] i;
74
75  // the following determines the drawing and clearing action
76  always_ff @(posedge clk[whichClock]) begin
77     if (~KEY[3]) begin // use KEY[3] as user control of clearing action
78        clear <= ~clear;
79        i <= 0;
80     end
81
82     if (finish) begin
83        // each time finish drawing a line, switch pixel_color
84        pixel_color <= ~pixel_color;
85        // after clearing, move to next endpoint
86        if (~pixel_color) begin
87           i <= i + 1;
88        end
89        // trigger next drawing action
90        reset <= 1'b1;
91     end else begin
92        // wait for drawing action to finish
93        reset <= 1'b0;
94     end
95  end
96
```

```
97      // As i gets incremented in the main program logic, the end point of the line changes accordingly
98      assign x1 = x_vals[i];
99      assign y1 = y_vals[i];
100
101     // the following instantiates the line drawer module
102     line_drawer lines (.clk(CLOCK_50), .reset, .x0, .x1, .y0, .y1, .x, .y, .finish);
103
104     // the following generates a slower clock
105     logic [31:0] clk;
106     parameter whichClock = 22; // 6 Hz clock
107     clock_divider cdiv (.clock(CLOCK_50), .divided_clocks(clk));
108
109 endmodule
110
111 module task2_testbench();
112     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
113     logic [9:0] LEDR;
114     logic [3:0] KEY;
115     logic [9:0] SW;
116     logic CLOCK_50;
117     logic [7:0] VGA_R;
118     logic [7:0] VGA_G;
119     logic [7:0] VGA_B;
120     logic VGA_BLANK_N;
121     logic VGA_CLK;
122     logic VGA_HS;
123     logic VGA_SYNC_N;
124     logic VGA_VS;
125
126      task2 dut(.*);
127
128      // Simulate clock
129      parameter CLOCK_PERIOD = 100;
130      initial begin
131          CLOCK_50 <= 0;
132          forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;
133      end
134
135      // there is no real useful information in the tb
136      // simply to see that input works
137      initial begin
138          KEY[3] <= 1'b0; @(posedge CLOCK_50);
139          KEY[3] <= 1'b1; @(posedge CLOCK_50);
140          for (int i = 0; i < 20; i++) begin
141                          @(posedge CLOCK_50);
142          end
143          $stop;
144      end
145 endmodule
```

b. clock_divider.sv

```
1  // Cynthia Li
2  // EE 371 LAB3 TASK2
3
4  // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
5  module clock_divider (clock, reset, divided_clocks);
6     input logic reset, clock;
7     output logic [31:0] divided_clocks = 0;
8
9     always_ff @(posedge clock) begin
10       divided_clocks <= divided_clocks + 1;
11    end
12 endmodule
```