Justin Sim and Mina Gao (Lab Group 4)
EE 371
February 2, 2024
Lab 3 Report

Procedure

This lab consists of two tasks. In the first, we are using the DE1_SoC's VGA display to show a line from any two coordinates. To do this, we needed to implement Bresenham's line algorithm. In the second task, we created an animation by alternating the position of a line segment between two start/end coordinate pairs.

Task #1

In the first task, I implemented Bresenham's line algorithm in the `line_drawer.sv` module. The module takes inputs `x0`, `y0`, `x1`, `y1`, corresponding to the coordinate pairs `(x0, y0)` and `(x1, y1)`, and my task was to draw the line of pixels connecting these pairs.

I created local registers to keep track of the algorithm's progress, ensuring that the coordinates `x` and `y` changed by at most one pixel each clock cycle. I paid attention to the signed arithmetic aspect, crucial for proper implementation, as indicated in the example.

To tackle the special cases of Bresenham's algorithm such as steep slopes and vertical lines, I approached the problem step by step. First, I handled cases where `x0 = x1` or `y0 = y1`, drawing perfectly straight lines. Then, I focused on diagonal lines, assuming `(x0, y0)` as `(0,0)` and `x1 = y1`, gradually expanding to handle diagonal lines from any starting point.

Finally, I modified the algorithm to handle lines with gradual slopes, such as from `(0,0)` to `(100, 20)`. This stepwise approach allowed me to tackle the problem effectively and efficiently, ensuring the module could draw lines between arbitrary points on the monitor regardless of direction or slope.

For the demonstration of task 1, I have written a 'switches' module that interprets switch inputs to generate a variety of lines by determining specific sets of points ((x0, y0), (x1, y1)). This module takes the state of the switches (SW0 to SW5) as input and translates them into corresponding sets of points: (x0, y0) and (x1, y1). The module is designed to showcase six types of lines. When SW0 is active, it generates a vertical line; SW1 produces a horizontal line;

SW2 creates a diagonal line with a positive slope; SW3 forms a diagonal line with a negative slope; SW4 leads to a gradual line with a positive slope; and SW5 results in a gradual line with a negative slope. The output coordinates for each line type are predefined in the module, and the appropriate line is drawn based on the switch that is turned on.

Task #2

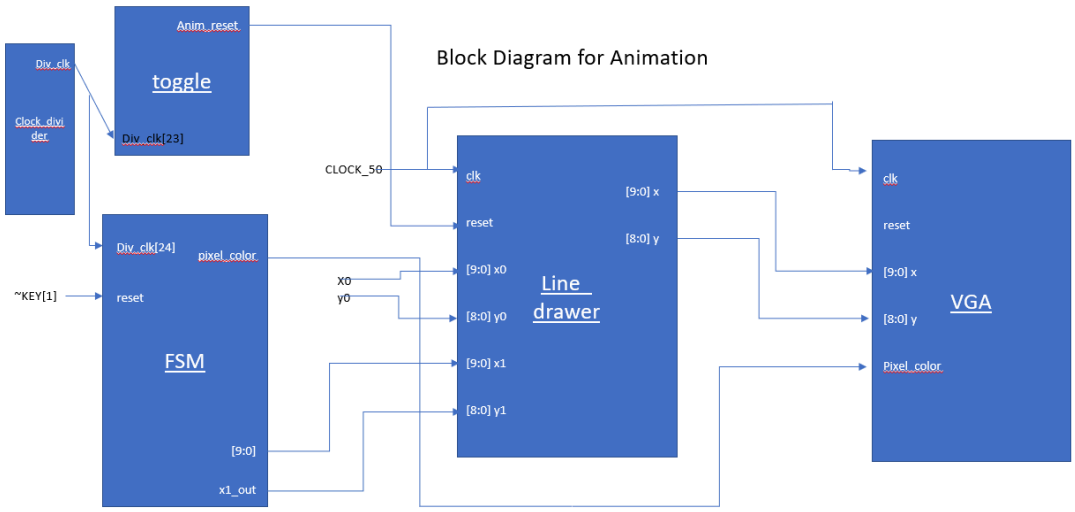

Figure 1: Top level Block Diagram

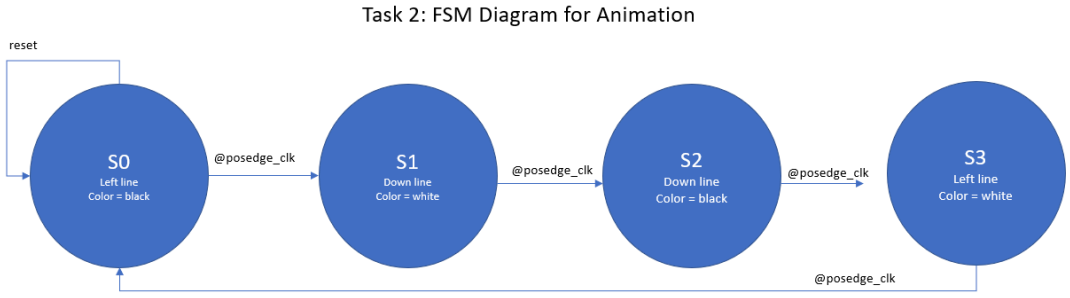

Figure 2: FSM Block Diagram

In the second task, we were asked to animate a line on the VGA screen. I decided to animate a counter-clockwise rotation of a line in 4 segments per 360 degree rotation. In other words, a line that spins around the center of the screen. Additionally, the screen would be all black when the reset is switched to on. In order to implement these functionalities, I created an FSM where each state represents the next line segment for the animation. This FSM would be synced to a “slower” clock frequency such that it does not create persistence of vision.

The difficulty with this task was in ensuring that the previous line segment went to black once the next line segment was set. To do this, we needed to implement a new clock that would reset the line_drawer module. This way, the line drawer will be reset at the same time as the fsm is returning the new set of coordinates.

Results

Task 1: line_drawer_testbench():

Inputs:

1) clk
2) reset
3) [9:0] x0, x1
4) [8:0] y0, y1

Outputs:

1) [9:0] x
2) [8:0] y

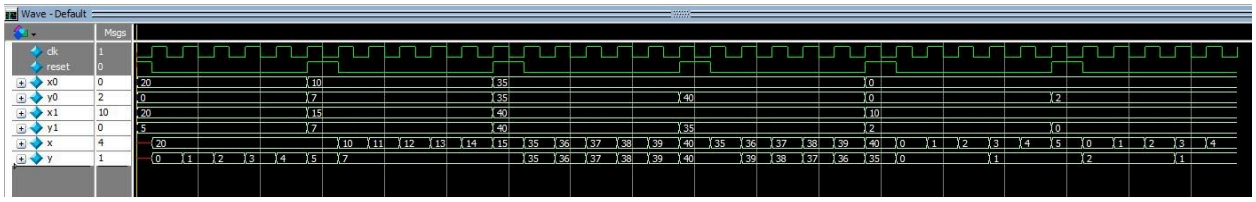

Figure 3: line_drawer testbench ModelSim

The image above shows a testbench for the line drawer module. In our testbench code, we wanted to simulate if the line drawer module can successfully implement the Bresenham’s line algorithm with any given coordinates.

. The tested conditions:

1) A vertical line, from (20, 0) to (20, 5)
2) A horizontal line, from (10, 7) to (15, 7)

3) A diagonal line with positive slope, from (35, 35) to (40, 40)
4) A diagonal line with negative slope, from (35, 40) to (40, 35)
5) A gradual line with positive slope, from (0,0) to (10, 2)
6) A gradual line with negative slope, from (0,2) to (10, 0)

We ran 5 clock cycles for each test condition. As we can see from the waveform, the module shows successful validation. Since when given a set of points, x and y are getting updated correctly every clock cycle along the expected line according to Bresenham's line algorithm.

Task 1: switches_testbench():

Inputs:

1) clk
2) [9:0] SW

Outputs:

1) [9:0] x0, x1
2) [8:0] y0, y1

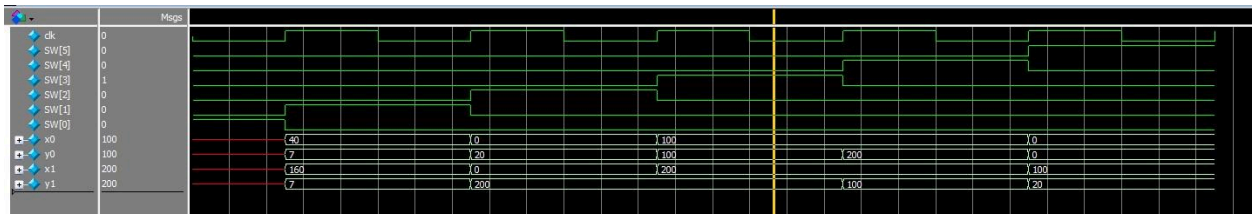

Figure 4: switches testbench for task 1

The image above shows a testbench for the switches module. In our testbench code, we wanted to test each switch (SW0 to SW5) individually to ensure that they correctly set the output points (x0, y0) (x1, y1).

. The tested conditions:

1) SW 0 is on (coordinates for a horizontal line)
2) SW 1 is on (coordinates for a vertical line)
3) SW 2 is on (coordinates for a diagonal line with positive slope)
4) SW 3 is on (coordinates for a diagonal line with negative slope)
5) SW 4 is on (coordinates for a gradual line with positive slope)
6) SW 5 is on (coordinates for a gradual line with negative slope)

As we can see from the waveform, the module shows successful validation, because the x and y coordinates are getting updated as expected while each corresponding switch is on.

Task 1: DE1_SoC_testbench():

Inputs:

1) CLOCK_50
2) [6:0] HEX: irrelevant
3) [9:0] LEDR; irrelevant
4) [3:0] KEY; irrelevant
5) [9:0] SW; irrelevant

Outputs:

1) [7:0] VGA_R; Red Pixels: VGA Display
2) [7:0] VGA_G; Green Pixels: VGA Display
3) [7:0] VGA_B; Blue Pixels: VGA Display
4) VGA_BLANK_N; default
5) VGA_CLK; default
6) VGA_HS; default
7) VGA_SYNC_N; default
8) VGA_VS; default

It's challenging to discern the direct interactions between modules from the DE1_SoC testbench since it does not directly show how x and y are getting updated. This difficulty arises when trying to confirm whether the correct set of points is being transferred from the switches module to the line_drawer, updated x y values are passing into the VGA_framebuffer and then into the top level DE1_SoC module. To verify if the coordinates are being updated as expected, we can examine the line_drawer waveform within the DE1_SoC testbench.

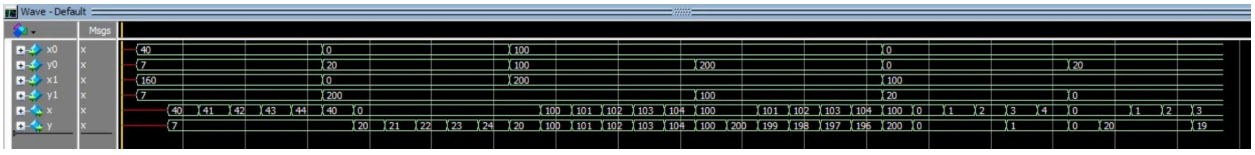

Figure 5: line_drawer waveform under DE1_SoC testbench

According to the figure, we can verify that the correct coordinates are being passed into the line_drawer and line_drawer is updating x y as expected and passing these values to the VGA_framebuffer module.

Task 2: DE1_SoC_testbench():

Inputs:

1) CLOCK_50
2) [6:0] HEX: irrelevant
3) [9:0] LEDR; irrelevant
4) [3:0] KEY; irrelevant
5) [9:0] SW; irrelevant

Outputs:

1) [7:0] VGA_R; Red Pixels: VGA Display
2) [7:0] VGA_G; Green Pixels: VGA Display
3) [7:0] VGA_B; Blue Pixels: VGA Display
4) VGA_BLANK_N; default
5) VGA_CLK; default
6) VGA_HS; default
7) VGA_SYNC_N; default
8) VGA_VS; default

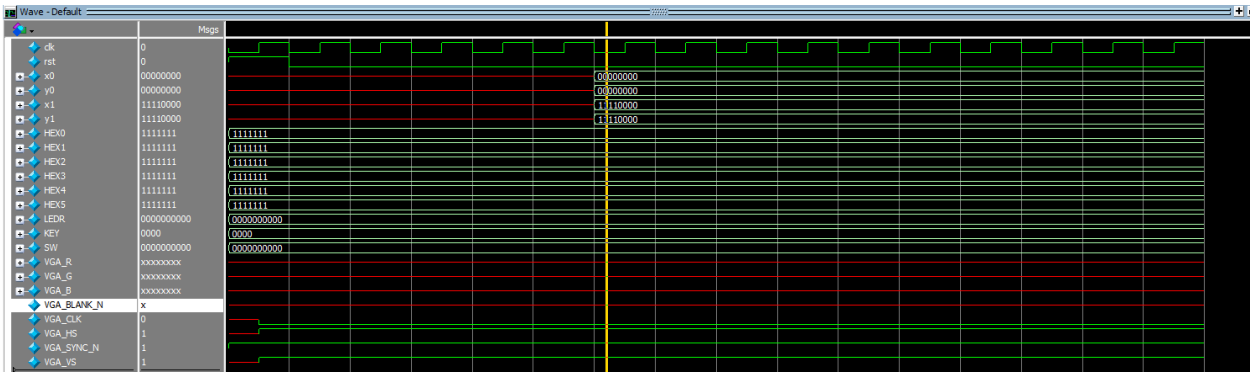

Figure 6: DE1_SoC testbench for task 2

The image above shows a testbench for the module. The intended flow was as such:

1) Maintain a low reset for 5 clock cycles
2) Set an example line to draw

    x0 = 0;

    y0 = 0;

    x1 = 240;

    y1 = 240;

According to the figure, it is easy to see at the yellow line that the xy coordinate pairs are outputted as expected.

Task 2: fsm_testbench:

Inputs:

1) SW[9] enable write
2) SW[8:4] input address
3) SW[3:0] input data
4) KEY[0] - clock

Outputs:

1) HEX[5:4] – address value (HEX5 is first digit)
2) HEX 2 – data being written to memory
3) HEX0 – read memory out

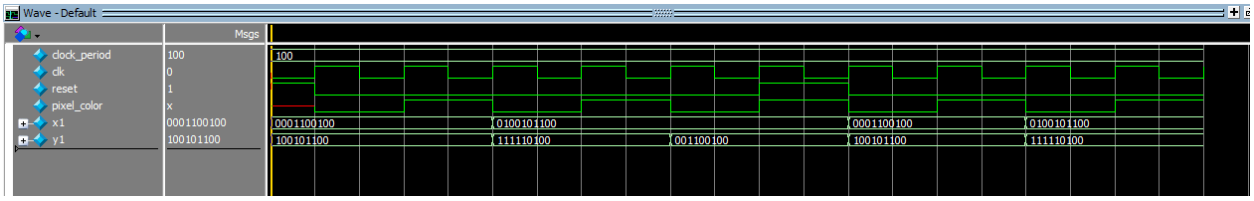

Figure 7: fsm_testbench for Task 2

The image above shows a testbench for the module. It tests the specific states that correspond to a new pair of coordinates. As shown, we can see that the coordinates for x1 and y1 change upon a given clock cycle. Additionally, we can see that the reset initializes the first state, all are aligned with out

Task 2: toggle_testbench():

Depending on the 5-bit input, the module will return a number from 0 - 31 in Hexadecimal format. The strategy to returning numbers beyond 0-9 is to pass a second adjacent HEX input such that it acts as the second digit such that each HEX can reach from 0 to 9. It was useful for Task 2 where we were working with 32 addresses and needed a way to identify each of them.

Inputs

1) clk

Outputs:

1) reset

Figure 8: toggle_testbench Modelsim

The image above shows a testbench for the module. The intended behavior is to display the toggle between reset and the clock. As it shows, the expected behavior is correct.

clock_divider_testbench():

clock_divider.sv was useful in Task 2 because it allowed for a longer clock cycle. It slows down the clock, so that we were able to scroll through the addresses at a reasonable rate (~1sec)

1) clk
2) reset

Outputs:

1) [31:0] divided_clocks

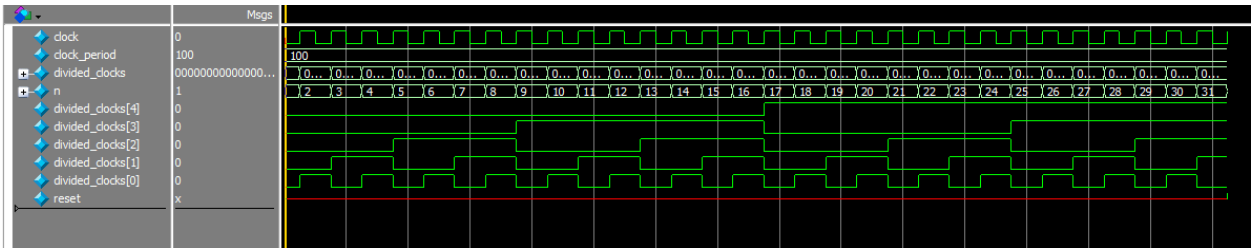

Figure 9: clock_divider testbench Modelsim

The image above shows a testbench for the module. The clock divider has 31 indices, each with a different clock frequency. One can see that divided_clock[1] has double the period of [0], [2] has double the period of [1] and so on. This is consistent with our expectations for the functionality, and it serves a great purpose in customizing the speed of our clock.

Appendix

```
/*
Justin Sim and Mina Gao
2/2/2024
EE 371 Hussein
Lab 3, Task 1

   This module will implement a line
   drawing algorithm, using the VGA Display
   in conjunction with the DE1_SoC

   inputs:
      CLOCK_50
      [6:0] HEX;          irrelevant
      [9:0] LEDR;         irrelevant
      [3:0] KEY;          irrelevant
      [9:0] SW;           irrelevant
   outputs:
      [7:0] VGA_R;        Red Pixels: VGA Display
      [7:0] VGA_G;        Green Pixels: VGA Display
      [7:0] VGA_B;        Blue Pixels: VGA Display
      VGA_BLANK_N;        default
      VGA_CLK;            default
      VGA_HS;             default
      VGA_SYNC_N;         default
      VGA_VS;             default

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

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

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

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

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


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

   VGA_framebuffer fb(.clk(CLOCK_50), .rst(~KEY[3]), .x, .y,
            .pixel_color, .pixel_write(1'b1), .dfb_en, .frame_start,
            .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
            .VGA_BLANK_N, .VGA_SYNC_N);

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

```

```
74      // controller module for demoing purposes
75      // creates bredth of line types
76      switches sw (.clk(CLOCK_50), .SW, .x0, .y0, .x1, .y1);
77
78      assign pixel_color =  1'b1;   //white
79    endmodule               //DE1_SoC
80    //-------------------------------------------------------------------------------------------
      -------------
81    //DE1_SoC_testbench tests the expected and unexpected behaviors, reset behaviors
82    module DE1_SoC_testbench ();
83      logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
84      logic [3:0] KEY;
85      logic [9:0] LEDR;
86      logic [9:0] SW;
87      logic CLOCK_50;
88      logic [7:0] VGA_R;
89      logic [7:0] VGA_G;
90      logic [7:0] VGA_B;
91      logic VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS;
92
93      logic reset;
94      assign KEY[3] = ~reset;
95
96
97      DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW, .CLOCK_50,
98                   .VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N, .VGA_CLK, .VGA_HS, .VGA_SYNC_N, .
      VGA_VS);
99
100     parameter clock_period = 100;
101     initial begin
102           CLOCK_50 <= 0;
103           forever #(clock_period / 2) CLOCK_50 <= ~CLOCK_50;
104     end
105
106     initial begin
107           reset <= 1; SW <= 10'b0000000001; repeat(2) @(posedge CLOCK_50);
108           reset <= 0;                       repeat(4) @(posedge CLOCK_50);
109           reset <= 1; SW <= 10'b0000000010; repeat(2) @(posedge CLOCK_50);
110           reset <= 0;                       repeat(4) @(posedge CLOCK_50);
111           reset <= 1; SW <= 10'b0000000100; repeat(2) @(posedge CLOCK_50);
112           reset <= 0;                       repeat(4) @(posedge CLOCK_50);
113           reset <= 1; SW <= 10'b0000001000; repeat(2) @(posedge CLOCK_50);
114           reset <= 0;                       repeat(4) @(posedge CLOCK_50);
115           reset <= 1; SW <= 10'b0000010000; repeat(2) @(posedge CLOCK_50);
116           reset <= 0;                       repeat(4) @(posedge CLOCK_50);
117           reset <= 1; SW <= 10'b0000100000; repeat(2) @(posedge CLOCK_50);
118           reset <= 0;                       repeat(4) @(posedge CLOCK_50);
119        $stop;
120     end
121   endmodule               //DE1_SoC_testbench
122
```

```
1   /*
2   Justin Sim and Mina Gao
3   2/2/2024
4   EE 371 Hussein
5   Lab 3, Task 2
6
7      This module will animate a line segment which
8      points left and down, back and forth, under
9      a specified speed. It utilizes the VGA Display
10     in conjunction with the DE1_SoC
11
12     inputs:
13        CLOCK_50
14        [6:0] HEX:           irrelevant
15        [9:0] LEDR;          irrelevant
16        [3:0] KEY;           irrelevant
17        [9:0] SW;            irrelevant
18     outputs:
19        [7:0] VGA_R;         Red Pixels: VGA Display
20        [7:0] VGA_G;         Green Pixels: VGA Display
21        [7:0] VGA_B;         Blue Pixels: VGA Display
22        VGA_BLANK_N;         default
23        VGA_CLK;             default
24        VGA_HS;              default
25        VGA_SYNC_N;          default
26        VGA_VS;              default
27
28  */
29  module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
30     VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS);
31
32     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
33     output logic [9:0] LEDR;
34     input logic [3:0] KEY;
35     input logic [9:0] SW;
36
37     input CLOCK_50;
38     output [7:0] VGA_R;
39     output [7:0] VGA_G;
40     output [7:0] VGA_B;
41     output VGA_BLANK_N;
42     output VGA_CLK;
43     output VGA_HS;
44     output VGA_SYNC_N;
45     output VGA_VS;
46
47     assign HEX0 = '1;
48     assign HEX1 = '1;
49     assign HEX2 = '1;
50     assign HEX3 = '1;
51     assign HEX4 = '1;
52     assign HEX5 = '1;
53     assign LEDR = SW;
54
55     // x/y-coordinates for line drawer algorithm
56     logic [9:0] x0, x1, x;
57     logic [8:0] y0, y1, y;
58
59     // for VGA
60     logic frame_start;
61     logic pixel_color;
62     logic reset;
63     assign reset = ~KEY[0];
64
65     logic anim_reset;    // stops the animation
66
67     //////// DOUBLE_FRAME_BUFFER ////////
68     logic dfb_en;
69     assign dfb_en = 1'b0;
70     ////////////////////////////////////////
71
72     // Clock Speed Selector
73     logic [31:0] div_clk;
```

```
74      parameter whichClock = 25; // 0.75 Hz clock
75      clock_divider cdiv (.clock(CLOCK_50),
76      .reset(~KEY[3]),
77      .divided_clocks(div_clk));
78
79
80      // Setting Pivot for Animating Line Segments
81      assign x0 = 300;
82      assign y0 = 300;
83
84      // Connects to VGA Display
85      VGA_framebuffer fb(.clk(CLOCK_50), .rst(reset), .x, .y,
86               .pixel_color, .pixel_write(1'b1), .dfb_en, .frame_start,
87               .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
88               .VGA_BLANK_N, .VGA_SYNC_N);
89
90      // draws lines between (x0, y0) and (x1, y1) using Bresnhth algorithm
91      line_drawer lines (.clk(CLOCK_50), .reset(anim_reset),
92               .x0, .y0, .x1, .y1, .x, .y);
93
94      // sets reset of line drawer to high and low every clock cycle
95      // 3 Hz clk - must be double the speed of fsm
96      toggle tog (.clk(div_clk[23]), .reset(anim_reset));
97
98      // sequence the line segments to appear animated
99      // 1.5 Hz clk
100     fsm animate (.clk(div_clk[24]), .start(~KEY[1]), .x1_out(x1), .y1_out(y1), .pixel_color);
101
102
103  endmodule            //DE1_SoC
104  //------------------------------------------------------------------------------------------
     --------------
105  //DE1_SoC_testbench tests the expected and unexpected behaviors, reset behaviors
106
107  module DE1_SoC_tb();
108     // Parameters
109    parameter WIDTH = 8;
110    parameter HEIGHT = 8;
111    parameter MEM_DEPTH = 64;
112
113    // Signals
114    logic clk, rst;
115    logic [WIDTH-1:0] x0, y0, x1, y1;
116    logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
117    logic [9:0] LEDR;
118    logic [3:0] KEY;
119    logic [9:0] SW;
120    logic [7:0] VGA_R, VGA_G, VGA_B;
121    logic VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS;
122
123    // Instantiate the module
124    DE1_SoC dut (
125      .CLOCK_50(clk),
126      .HEX0(HEX0),
127      .HEX1(HEX1),
128      .HEX2(HEX2),
129      .HEX3(HEX3),
130      .HEX4(HEX4),
131      .HEX5(HEX5),
132      .LEDR(LEDR),
133      .KEY(KEY),
134      .SW(SW),
135      .VGA_R(VGA_R),
136      .VGA_G(VGA_G),
137      .VGA_B(VGA_B),
138      .VGA_BLANK_N(VGA_BLANK_N),
139      .VGA_CLK(VGA_CLK),
140      .VGA_HS(VGA_HS),
141      .VGA_SYNC_N(VGA_SYNC_N),
142      .VGA_VS(VGA_VS)
143    );
144
145    // Clock generation
```

```
146     initial begin
147       clk = 0;
148       forever #5 clk = ~clk;
149     end
150
151     // Test scenario
152     initial begin
153       // Initialize inputs
154       rst = 1;
155       KEY = 4'b0000;
156       SW = 10'b0000000000;
157
158       // Release reset
159       #10 rst = 0;
160
161       // Wait for a few clock cycles
162       #50;
163
164       // Set an example line to draw
165       x0 = 0;
166       y0 = 0;
167       x1 = 240;
168       y1 = 240;
169
170       #100;
171
172       $stop;
173     end
174
175   endmodule               //DE1_SoC_testbench
176
```

```
1   /*
2   Justin Sim and Mina Gao
3   1/23/2024
4   EE 371 Hussein
5   Lab 3
6
7      VGA driver: provides I/O timing and double-buffering for the VGA port.
8
9   */
10
11  module VGA_framebuffer(
12     input logic clk, rst,
13     input logic [9:0] x, // The x coordinate to write to the buffer.
14     input logic [8:0] y, // The y coordinate to write to the buffer.
15     input logic pixel_color, pixel_write, // The data to write (color) and write-enable.
16
17     input logic dfb_en, // Double-Frame Buffer Enable
18
19     output logic frame_start,   // Pulse is fired at the start of a frame.
20
21     // Outputs to the VGA port.
22     output logic [7:0] VGA_R, VGA_G, VGA_B,
23     output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N
24  );
25
26     /*
27     *
28     * HCOUNT 1599 0             1279       1599 0
29     *              _______________           ________
30     * ___________|    Video      |___________|  Video
31     *
32     *
33     * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
34     *       _____________________   _____________
35     * |____|       VGA_HS          |____|
36     *
37     */
38
39     // Constants for VGA timing.
40     localparam HPX = 11'd640*2, HFP = 11'd16*2, HSP = 11'd96*2, HBP = 11'd48*2;
41     localparam VLN = 11'd480,   VFP = 10'd11,   VSP = 10'd2,    VBP = 10'd31;
42     localparam HTOTAL = HPX + HFP + HSP + HBP; // 800*2=1600
43     localparam VTOTAL = VLN + VFP + VSP + VBP; // 524
44
45     // Horizontal counter.
46     logic [10:0] h_count;
47     logic end_of_line;
48
49     assign end_of_line = h_count == HTOTAL - 1;
50
51     always_ff @(posedge clk)
52        if (rst) h_count <= 0;
53        else if (end_of_line) h_count <= 0;
54        else h_count <= h_count + 11'd1;
55
56     // Vertical counter & buffer swapping.
57     logic [9:0] v_count;
58     logic end_of_field;
59     logic front_odd; // whether odd address is the front buffer.
60
61     assign end_of_field = v_count == VTOTAL - 1;
62     assign frame_start = !h_count && !v_count;
63
64     always_ff @(posedge clk)
65        if (rst) begin
66           v_count <= 0;
67           front_odd <= 0;
68        end else if (end_of_line)
69           if (end_of_field) begin
70              v_count <= 0;
71              front_odd <= !front_odd;
72           end else
73              v_count <= v_count + 10'd1;
```

```
74
75      // Sync signals.
76      assign VGA_CLK = h_count[0]; // 25 MHz clock: pixel latched on rising edge.
77      assign VGA_HS = !(h_count - (HPX + HFP) < HSP);
78      assign VGA_VS = !(v_count - (VLN + VFP) < VSP);
79      assign VGA_SYNC_N = 1; // Unused by VGA
80
81      // Blank area signal.
82      logic blank;
83      assign blank = h_count >= HPX || v_count >= VLN;
84
85      // Double-buffering.
86      logic buffer[640*480*2-1:0];
87      logic [19:0] wr_addr, rd_addr;
88      logic rd_data;
89
90      assign wr_addr = {y * 19'd640 + x, (!front_odd & dfb_en)};
91      assign rd_addr = {v_count * 19'd640 + (h_count / 19'd2), (front_odd & dfb_en)};
92
93      always_ff @(posedge clk) begin
94         if (pixel_write) buffer[wr_addr] <= pixel_color;
95         if (VGA_CLK) begin
96            rd_data <= buffer[rd_addr];
97            VGA_BLANK_N <= ~blank;
98         end
99      end
100
101     // Color output.
102     assign {VGA_R, VGA_G, VGA_B} = rd_data ? 24'hFFFFFF : 24'h000000;
103  endmodule
```

```
/*
Justin Sim and Mina Gao
2/2/2024
EE 371 Hussein
Lab 3

   This module implements Bersenham's Algorithm given a pairs of starting
   coordinates, and a pair of ending coordinates, it will return the next
   pair of coordinates in the direction of start to end.

   inputs:
      clk, reset
      [9:0] x0, x1,
      [8:0] y0, y1,
   outputs:
      [9:0] x,
      [8:0] y

*/
module line_drawer(
   input logic clk, reset,

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

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

   /* registers to keep track of things
      error, direction, begin and end point for both x and y
      delta x, delta y, etc
   */
   logic signed [11:0] error;
   logic is_steep;         // flag for steep
   logic signed [11:0] x_step, y_step; // direction for x and y increment
   logic signed [10:0] delta_x, delta_y, x_begin, x_end, y_begin, y_end;

   /* find delta x and y and determine if the line is steep
      find the begin, end point, and step direction for x and y
   */
   always_comb begin
      delta_x = (x0 > x1) ? (x0 - x1) : (x1 - x0);
      delta_y = (y0 > y1) ? (y0 - y1) : (y1 - y0);
      is_steep = delta_y > delta_x;
      if (is_steep) begin
         x_begin = (y0 <= y1) ? x0 : x1;
         y_begin = (y0 <= y1) ? y0 : y1;
         x_end = (y0 > y1) ? x0 : x1;
         y_end = (y0 > y1) ? y0 : y1;
      end else begin
         x_begin = (x0 <= x1) ? x0 : x1;
         y_begin = (x0 <= x1) ? y0 : y1;
         x_end = (x0 > x1) ? x0 : x1;
         y_end = (x0 > x1) ? y0 : y1;
      end
      x_step = (x_begin < x_end) ? 1 : -1;
      y_step = (y_begin < y_end) ? 1 : -1;
   end

   // sequential logic for line drawing
   always_ff @(posedge clk) begin
      if (reset) begin
         x <= x_begin;
         y <= y_begin;
         error <= is_steep ? (-delta_y/2) : (-delta_x/2);
      // line is steep
      end else if (is_steep) begin
         if (y < y_end) begin
            y <= y + 1;
            if ((error + delta_x) >= 0) begin
```

```
74                   x <= x + x_step;
75                  error <= error + delta_x - delta_y;
76               end else begin
77                  error <= error + delta_x;
78               end
79            end
80         // line is gradual
81         end else if (x < x_end) begin
82            x <= x + 1;
83            if ((error + delta_y) >= 0) begin
84               y <= y + y_step;
85               error <= error + delta_y - delta_x;
86            end else begin
87               error <= error + delta_y;
88            end
89         end
90      end
91   endmodule               //line_drawer
92
93   // line_drawer_testbench tests the expected and unexpected behaviors, reset behaviors
94   // specifically, it tests 4 different situations, encompassing all types of scenarios
95   module line_drawer_testbench ();
96      logic clk, reset;
97      logic [9:0] x0, x1;
98      logic [8:0] y0, y1;
99      logic [9:0] x;
100     logic [8:0] y;
101
102     line_drawer ld (.clk, .reset, .x0, .x1, .y0, .y1, .x, .y);
103
104     // clock setup
105     parameter clock_period = 100;
106
107     initial begin
108        clk <= 0;
109        forever #(clock_period / 2) clk <= ~clk;
110     end
111
112     integer i;
113     initial begin
114        reset = 1; x0 = 20; y0 = 0; x1 = 20; y1 = 30; @(posedge clk); // straight line
115        for (i = 0; i < 10; i++) begin
116           reset = 0; @(posedge clk);
117        end
118        reset = 1; x0 = 30; y0 = 30; x1 = 40; y1 = 40; @(posedge clk); // diagonal line
119        for (i = 0; i < 10; i++) begin
120           reset = 0; @(posedge clk);
121        end
122        reset = 1; x0 = 5; y0 = 7; x1 = 15; y1 = 15; @(posedge clk); // diagonal line from
   any point
123        for (i = 0; i < 10; i++) begin
124           reset = 0; @(posedge clk);
125        end
126        reset = 1; x0 = 0; y0 = 0; x1 = 10; y1 = 2; @(posedge clk); // gradual line
127        for (i = 0; i < 10; i++) begin
128           reset = 0; @(posedge clk);
129        end
130        $stop; // end simulation
131     end
132  endmodule               //line_drawer_testbench
133
```

```
1   /*
2   Justin Sim and Mina Gao
3   1/23/2024
4   EE 371 Hussein
5   Lab 3, Task 2
6
7      This module returns a sequence of pixel coordinates
8      that is sent to line_drawer. It is necessary for
9      animation.
10     Additionally, the reset allows the animation to stop
11
12     inputs:
13        clk, reset;
14     outputs:
15        [9:0] x1_out;     // sent to line_drawer
16        [8:0] y1_out;     // sent to line_drawer
17        pixel_color;      // sent to VGA_framebuffer
18
19  */
20  module fsm(clk, reset, x1_out, y1_out, pixel_color);
21
22     input logic clk;
23     input logic reset;
24     output logic [9:0] x1_out;
25     output logic [8:0] y1_out;
26     output logic pixel_color;
27
28     logic [9:0] x1temp;
29     logic [8:0] y1temp;
30     logic color;
31
32     enum {S0, S1, S2, S3} ps, ns;
33
34     always_comb begin
35        case (ps)
36           S0:   begin                // black Left / reset
37                 ns = S1;
38                 x1temp = 100;
39                 y1temp = 300;
40                 color = 1'b0;
41                 end
42           S1:   begin                // white down
43                 ns = S2;
44                 x1temp = 300;
45                 y1temp = 500;
46                 color = 1'b1;
47                 end
48           S2:   begin
49                 ns = S3;
50                 x1temp = 300;        // black right
51                 y1temp = 500;
52                 color = 1'b0;
53                 end
54           S3:   begin
55                 ns = S0;
56                 x1temp = 100;        // black horizontal
57                 y1temp = 300;
58                 color = 1'b1;
59                 end
60        endcase
61     end
62
63     assign x1_out = x1temp;
64     assign y1_out = y1temp;
65     assign pixel_color = color;
66
67     always_ff @(posedge clk) begin
68     if (reset)  begin ps <= S0; end
69     else        begin ps <= ns; end
70     end
71  endmodule
72  /*-------------------------------------------------------------*/
73  //fsm_testbench tests the expected and unexpected behaviors, reset behaviors
```

```
74  // specifically, it tests all 4 states as well as the reset
75
76  module fsm_testbench();
77     logic clk, reset;
78     logic pixel_color;
79     logic [9:0] x1;
80     logic [8:0] y1;
81
82     fsm dut (.clk(clk), .reset(reset), .x1_out(x1), .y1_out(y1), .pixel_color(pixel_color));
83     parameter clock_period = 100;
84
85     initial begin clk <= 0;
86        forever #(clock_period /2) clk <= ~clk; end
87
88     initial begin
89        reset <= 0; @(posedge clk);
90     @(posedge clk);
91     @(posedge clk);
92     @(posedge clk);
93     reset <=1; @(posedge clk);
94     @(posedge clk);
95     @(posedge clk);
96     @(posedge clk);
97     @(posedge clk);
98     @(posedge clk); // return to start
99     @(posedge clk);
100    @(posedge clk);
101    @(posedge clk);
102    @(posedge clk);
103    $stop;
104    end
105 endmodule
```

```
1   /*
2   Justin Sim and Mina Gao
3   1/23/2024
4   EE 371 Hussein
5   Lab 3, Task 2
6
7      This module divides the input clock such that it can slow down the cycle:
8      divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz,[25] = 0.75Hz, ...
9
10     inputs:
11        clock
12     in/outputs:
13        [31:0] divided_clocks           array of varying clock speeds
14
15  */
16  module clock_divider (clock, reset, divided_clocks);
17     input logic reset, clock;
18     output logic [31:0] divided_clocks = 0;
19     always_ff @(posedge clock) begin
20        divided_clocks <= divided_clocks + 1;
21     end
22  endmodule               // clock_divider
23  //-------------------------------------------------------------------------------------------
    --------------
24  // clock_divider_testbench tests the expected behaviors of the clock_divider module
25
26  module clock_divider_testbench();
27  logic reset, clock;
28  logic [31:0] divided_clocks;
29
30  clock_divider dut (.clock, .reset, .divided_clocks);
31
32  // clock setup
33     parameter clock_period = 100;
34     initial begin
35        clock <= 0;
36        forever #(clock_period / 2) clock <= ~clock;
37     end
38     integer n;
39     initial begin
40        for (n=1; n<=31; n++) begin
41        @(posedge clock);
42        end
43        reset = 1;
44        $stop;
45     end
46
47  endmodule               // clock_divider_testbench
```

```
1  /*
2  Justin Sim and Mina Gao
3  2/2/2024
4  EE 371 Hussein
5  Lab 3, Task 2
6
7     This module returns a value that toggles 0 and 1 for the given
8     clock cycle. It is useful for resetting the line_drawer
9     to prepare it for the next set of coordinates from fsm.
10
11    inputs:
12       clk,
13    outputs:
14       reset;
15 */
16 module toggle(clk, reset);
17    input logic clk;
18    output logic reset;
19
20    logic [31:0] counter = 0;
21
22    always_ff @(posedge clk) begin
23       counter <= counter + 1;
24       reset <= counter[0];          // counter[0] toggles 0 and 1
25    end
26 endmodule               //toggle
27 // toggle_testbench tests the expected and unexpected behaviors, reset behaviors
28 module toggle_tb();
29    logic clk;
30    logic reset;
31
32    toggle dut (.clk, .reset);
33
34    // Clock generation
35    initial begin
36       clk = 0;
37       forever #5 clk = ~clk;
38    end
39
40    initial begin
41       #100;
42       $stop;
43    end
44 endmodule               //toggle_testbench
45
46
```

```
1   /*
2   Justin Sim and Mina Gao
3   2/2/2024
4   EE 371 Hussein
5   Lab 3
6
7      This module is given a set of controls and returns
8      a set of coordinates that encompass a broad range
9      of line types
10
11     inputs:
12        clk
13        [9:0] SW;
14     outputs:
15        [9:0] x0, x1;
16        [8:0] y0, y1;
17
18  */
19  module switches (clk, SW, x0, y0, x1, y1);
20      input logic clk;
21      input logic [9:0] SW;
22      output logic [9:0] x0, x1;
23      output logic [8:0] y0, y1;
24
25      always_ff @(posedge clk) begin
26          case(1'b1) // Priority encoder
27              SW[0]: begin // Horizontal line// Vertical line
28                  x0 <= 40;
29                  y0 <= 7;
30                  x1 <= 160;
31                  y1 <= 7;
32              end
33              SW[1]: begin // Vertical line
34                  x0 <= 0;
35                  y0 <= 20;
36                  x1 <= 0;
37                  y1 <= 200;
38              end
39              SW[2]: begin // Diagonal line with positive slope
40                  x0 <= 100;
41                  y0 <= 100;
42                  x1 <= 200;
43                  y1 <= 200;
44              end
45              SW[3]: begin // Diagonal line with negative slope
46                  x0 <= 100;
47                  y0 <= 200;
48                  x1 <= 200;
49                  y1 <= 100;
50              end
51              SW[4]: begin // Gradual line with positive slope
52                  x0 <= 0;
53                  y0 <= 0;
54                  x1 <= 100;
55                  y1 <= 20;
56              end
57              SW[5]: begin // Gradual line with negative slope
58                  x0 <= 0;
59                  y0 <= 20;
60                  x1 <= 100;
61                  y1 <= 0;
62              end
63              default: begin // Default case when no switch is active
64                  x0 <= 0;
65                  y0 <= 0;
66                  x1 <= 0;
67                  y1 <= 0;
68              end
69          endcase
70      end
71  endmodule
72  // switches_testbench tests the expected and unexpected behaviors, reset behaviors
73  // specifically, it tests all types of lines
```

```
74  module switches_testbench ();
75     logic clk;
76     logic [9:0] SW;
77     logic [9:0] x0, x1;
78     logic [8:0] y0, y1;
79
80     switches test (.clk, .SW, .x0, .y0, .x1, .y1);
81
82     // clock setup
83     parameter clock_period = 100;
84
85     initial begin
86        clk <= 0;
87        forever #(clock_period / 2) clk <= ~clk;
88     end
89
90     integer i;
91     initial begin
92        for (i = 0; i < 6; i++) begin
93           SW = 10'b0000000000;
94           SW[i] = 1; @(posedge clk);
95        end
96        $stop; // end simulation
97     end
98  endmodule
```