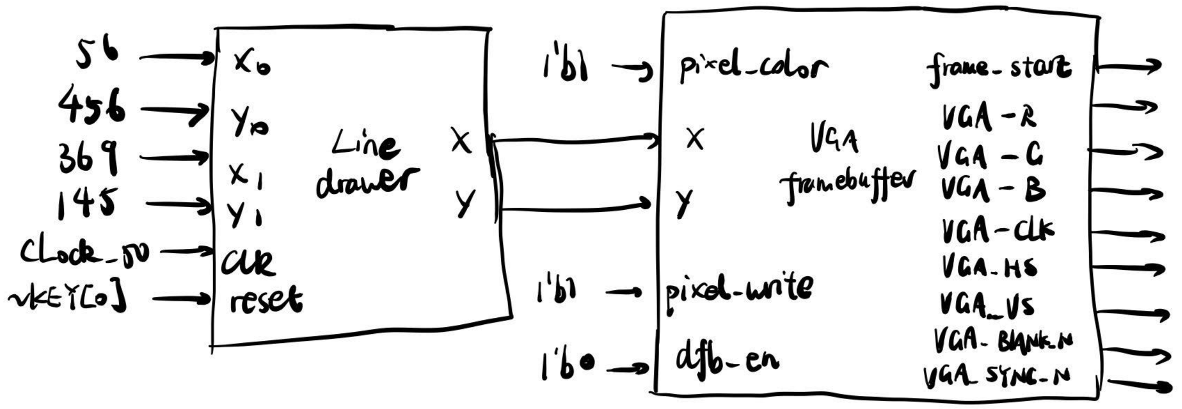
Peter Zhong
EE 371
May. 5th, 2021
Lab 3 Report

Procedure

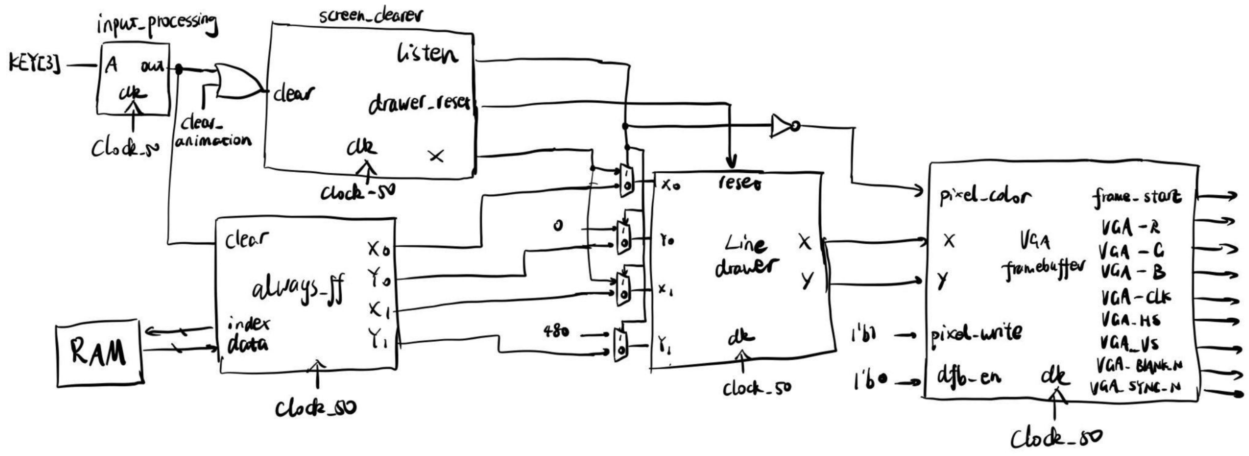
Task #1: Drawing an Arbitrary Line with line_drawer



(Figure 1: Block Diagram of the top-level module for task 1)

Task 1 asks me to implement Bresenham's line algorithm in hardware. To implement this algorithm, I first referred to the pseudo code of the Bresenham's algorithm provided in the lab manual. Because we can only use integers in Verilog, this pseudo code was extremely helpful for me to understand how the algorithm works by using an integer "error" variable to determine whether an increment should be applied to the desired variable. Then, I implemented the algorithm in line_drawer.sv in 4 different parts: determining the steepness of the line, finding starting and ending point based on the steepness, determining positive or negative step values for x and y, and an always_ff block with logics to determine the incrementation of x and y. Please refer to line_drawer.sv for more details regarding my implementation.

Task #2:



(Figure 2: Block Diagram of the top-level module for task 2)

Task 2 asks me to implement a line animation on the VGA display and a "clear" function to clear out the entire display by painting everything black.

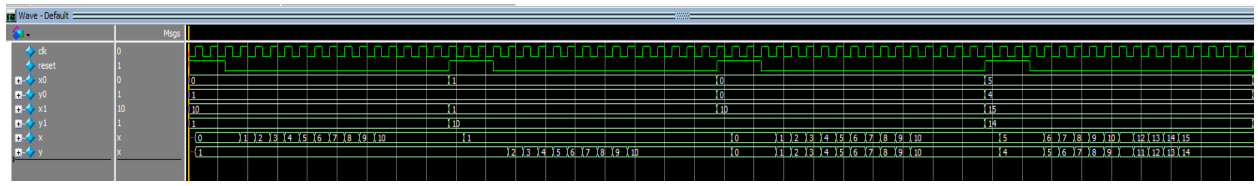
<u>To implement the clear function:</u> I realized that clearing the monitor can be achieved by setting $y_0 = 0$, $y_1 = 480$, and incrementing x_0 and x_1 from 0 to 640 (since the monitor is 640 x 480 pixels). Therefore, to implement this functionality, I created the "screen_clearer" module to perform the operations above when a "clear" signal is passed to the module. Within this module, I added a "listen" signal so that the line_drawer module knows which logic to listen to, and I also added a "drawer_reset_clearer" signal to tell the line_drawer module when to pick up the new coordinates. Outside of this module, I used a mux that uses the "listen" output of the clearer module as the selection signal to control whether the line drawer is drawing in black or white.

<u>To implement the animation:</u> I wanted the animation to repeatedly draw a pattern. Therefore, I figured that it would be the best if I can instantiate RAMs (or 2D arrays) to store a pattern of starting and ending coordinates. After loading the RAMs with proper values of the shape that I want to draw (see the shape in my task 2 demo video), I implemented a counting logic inside of my always_ff block in the top-level module to control the animation. Please refer to the DE1_SoC_2.sv for more detail regarding the implementation of my screen clearing module and the animation logics inside of my top-level module.

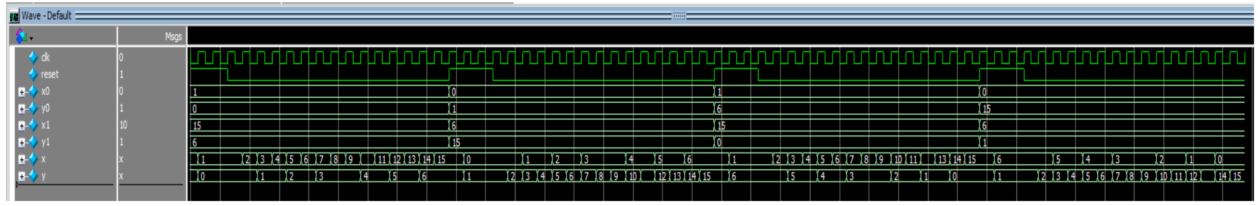
Results

Task #1 - Demonstration Video: https://youtu.be/87gd4jXx1qs

Screenshot for the line_drawer module simulation:



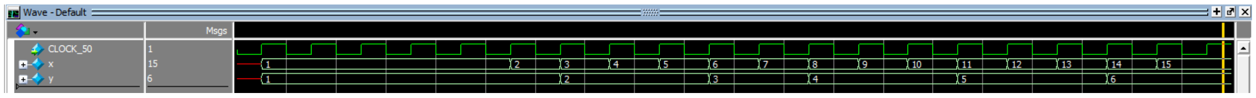
(Figure 3: Screenshot of line_drawer Module Simulation, Scenario 1-4)



(Figure 4: Screenshot of line_drawer Module Simulation, Scenario 5-8)

- For the line_drawer module, I tested 8 different scenarios in my testbench to verify its functionality. These 8 scenarios are:
    1. Horizontal line
    2. Vertical line
    3. Perfectly diagonal line from origin
    4. Perfectly diagonal line from an arbitrary point
    5. Arbitrary line, positive and gradual slope
    6. Arbitrary line, positive and steep slope
    7. Arbitrary line, negative and gradual slope
    8. Arbitrary line, negative and steep slope
- Scenario 1-4 can be seen in Figure 3, and scenario 5-8 can be seen in Figure 4. As can be seen from the figures above, here is a summary of the performance of the module under each scenario:
    1. y stays constant and only x is incremented.
    2. x stays constant and only y is incremented.
    3. Both x and y increments simultaneously.
    4. Both x and y increments simultaneously.
    5. 1-by-1 incrementation for x, incrementation only at proper times for y.
    6. 1-by-1 incrementation for y, incrementation only at proper times for x.
    7. 1-by-1 incrementation for x, decrementation only at proper times for y.
    8. 1-by-1 incrementation for y, decrementation only at proper times for x.

Screenshot for top-level simulation:

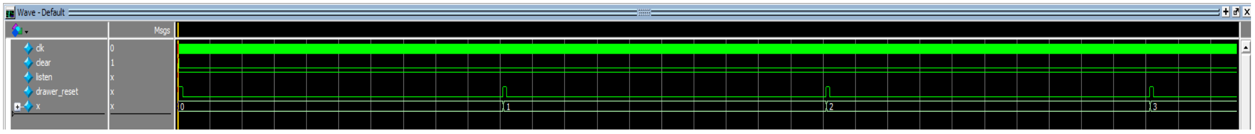


(Figure 5: Screenshot of top-level Simulation)

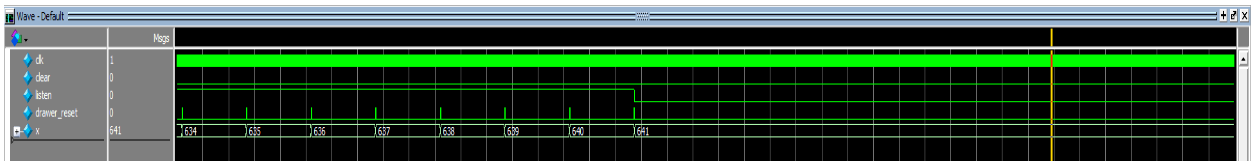
- For the top-level module, I drew an arbitrary line from (1,1) to (15,6). As can be seen from the simulation above, the top-level module properly outputs the coordinates of the pixels on the line over 15 clock cycles.

Task #2 - Demonstration Video: https://youtu.be/o7fsbg2s9cQ

Screenshot for the screen_clearer module simulation:



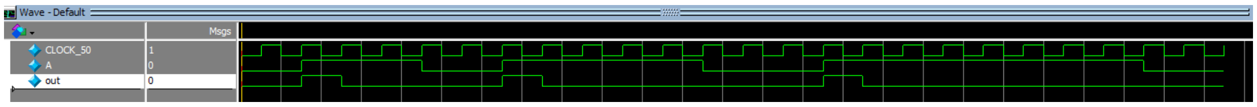
(Figure 6: Screenshot of line_drawer Module Simulation, Regular Operation)



(Figure 7: Screenshot of line_drawer Module Simulation, Reaching the End)

- For the screen clearer, I sent it a clear signal and let it run for 400,000 clock cycles. As can be seen from the screenshot above, the clearer correctly increments the x coordinate of the row by 1 for every 500 clock cycles. It also holds the value of x at 641 (out of screen) after the clearing has been completed. The listen signal goes true when a clear signal is received, and it goes back to false when the clearing has been completed. The drawer_reset signal goes true for 5 cycles every time when x is updated.

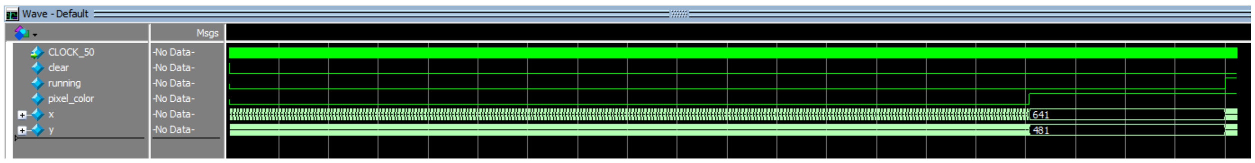
Screenshot for input_processing simulation:



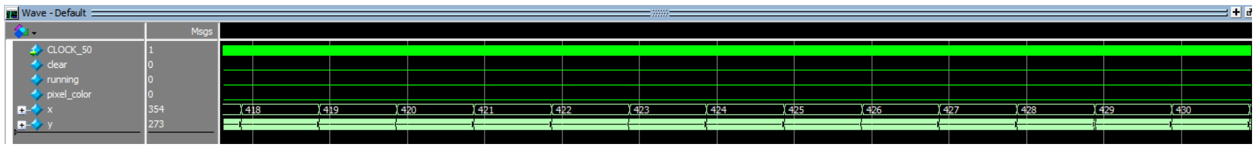
(Figure 8: Screenshot of input_processing Module Simulation)

- To test the functionality of input_processing, I pressed and released the virtual "input" button for random amounts of durations. As can be seen from the simulation above, the input processing module correctly reduced the user input to 1 clock cycle long for all instances of user input.

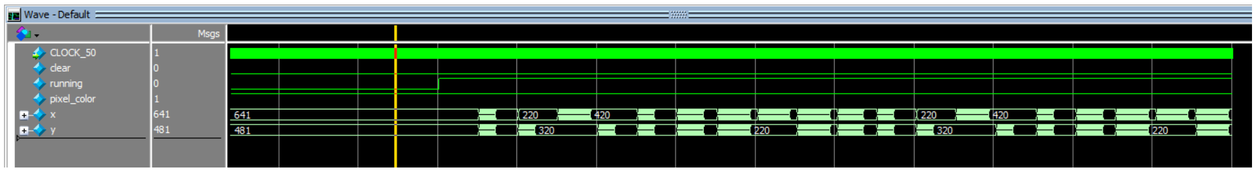
Screenshot for top-level simulation:



(Figure 9: Screenshot of top-level Module Simulation, Overview)



(Figure 10: Screenshot of top-level Module Simulation, Zoomed in on clear operation)



(Figure 11: Screenshot of top-level Module Simulation, Zoomed in on line-drawing operation)

- For the top-level module, I first cleared the screen, and then let the module run for around 5000 cycles.
- Interpreting Figure 9: the module is clearing the screen when "running" is 0, and the module is drawing the line when "running" is 1. As can be seen in Figure 9, the clearer module works properly, for it stops at an out-of-bound inactive coordinate after clearing the entire screen. Also, the pixel_color signal immediately goes back to 1 (white) after the screen has been cleared with the 0 (black) signal.
- Interpreting Figure 10: this figure is a close-up of the operation of the clearer. As can be seen in Figure 10, for every x coordinate, the line_drawer paints all the y pixels from 0 to 480 in black, and then increments x to the next x coordinate.
- Interpreting Figure 11: this figure is a close-up of the operation of the drawing process. As can be seen from Figure 11, it accurately draws out the lines given by the pre-configured RAM (see in the .sv code below) pixel by pixel. It also paints in white, as can be seen from the color signal.

Appendix

Task #1

1) line_drawer.sv

```
// Peter Zhong
// 5/5/2021
// EE 371
// Lab #3. Task 1

// line_drawer uses Bresenham's line algorithm to draw a line based on two sets of
// coordinates given by the input. It outputs the coordinates of the pixels over
// several clock cycles. The number of cycles is equal to the number of pixels in the line.

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
   );

   // determining steepness of the line
   logic is_steep;
   logic signed [10:0] xgap;
   logic signed [9:0] ygap;
   // calculating absolute value of the deltas
   assign xgap = (x0 > x1) ? (x0 - x1) : (x1 - x0);
   assign ygap = (y0 > y1) ? (y0 - y1) : (y1 - y0);
   assign is_steep = (ygap > xgap) ? 1'b1 : 1'b0;

   // finding starting and ending point
   logic signed [10:0] xstart, xend;
   logic signed [9:0] ystart, yend;

   always_comb begin
      // starting on the smaller y value if the line is steep
      if (is_steep) begin
         xstart = (y0 <= y1) ? x0 : x1;
         ystart = (y0 <= y1) ? y0 : y1;
         xend = (y0 > y1) ? x0 : x1;
         yend = (y0 > y1) ? y0 : y1;
      end
      // starting on the smaller x value if the line is gradual
      else begin
         xstart = (x0 <= x1) ? x0 : x1;
         ystart = (x0 <= x1) ? y0 : y1;
         xend = (x0 > x1) ? x0 : x1;
         yend = (x0 > x1) ? y0 : y1;
      end
   end

   // determining positive or negative steps for x and y coordinates
   integer xstep, ystep;
   assign xstep = (xend > xstart) ? 1 : -1;
   assign ystep = (yend > ystart) ? 1 : -1;
   // logic to output x and y coordinates
   logic signed [11:0] error;
   always_ff @(posedge clk) begin
      // reset x and y to starting points, reset error to initial value
      if (reset) begin
         x <= xstart;
         y <= ystart;
         if (is_steep == 1'b1)
            error <= -ygap/2;
         else
            error <= -xgap/2;
      end
      // logic for if the line is steep
      // increment y one by one, conditional step for x
      else if ((is_steep == 1'b1) & (y < yend)) begin
         y <= y + 1;
         if ((error + xgap) >= 0) begin // increment x
            x <= x + xstep;
            error <= error + xgap - ygap;
         end
         else // update error
            error <= error + xgap;
      end
      // logic for if the line is gradual
      // increment x one by one, conditional step for y
      else if (x < xend) begin
         x <= x + 1;
         if ((error + ygap) >= 0) begin // increment y
            y <= y + ystep;
            error <= error + ygap - xgap;
         end
         else // update error
            error <= error + ygap;
      end
   end
endmodule
```

```
// Testbench for line_drawer module
module line_drawer_testbench();
   logic [9:0] x0, x1, x;
   logic [8:0] y0, y1, y;
   logic clk, reset;

   line_drawer dut (.*);

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

   initial begin
      // horizontal line
      reset <= 1; x0 <= 0; y0 <= 1; x1 <= 10; y1 <= 1; repeat(3) @(posedge clk);
      reset <= 0;                                      repeat(15) @(posedge clk);
      // vertical line
      reset <= 1; x0 <= 1; y0 <= 1; x1 <= 1; y1 <= 10; repeat(3) @(posedge clk);
      reset <= 0;                                      repeat(15) @(posedge clk);
      // perfectly diagonal line from origin
      reset <= 1; x0 <= 0; y0 <= 0; x1 <= 10; y1 <= 10; repeat(3) @(posedge clk);
      reset <= 0;                                      repeat(15) @(posedge clk);
      // perfectly diagonal line from arbitrary point
      reset <= 1; x0 <= 5; y0 <= 4; x1 <= 15; y1 <= 14; repeat(3) @(posedge clk);
      reset <= 0;                                      repeat(15) @(posedge clk);
      // positive slope gradual slope line (arbitrary)
      reset <= 1; x0 <= 1; y0 <= 0; x1 <= 15; y1 <= 6; repeat(3) @(posedge clk);
      reset <= 0;                                      repeat(15) @(posedge clk);
      // positive slope steep slope line (arbitrary)
      reset <= 1; x0 <= 0; y0 <= 1; x1 <= 6; y1 <= 15; repeat(3) @(posedge clk);
      reset <= 0;                                      repeat(15) @(posedge clk);
      // negative slope gradual slope line (arbitrary)
      reset <= 1; x0 <= 1; y0 <= 6; x1 <= 15; y1 <= 0; repeat(3) @(posedge clk);
      reset <= 0;                                      repeat(15) @(posedge clk);
      // negative slope steep slope line (arbitrary)
      reset <= 1; x0 <= 0; y0 <= 15; x1 <= 6; y1 <= 1; repeat(3) @(posedge clk);
      reset <= 0;                                      repeat(15) @(posedge clk);
      $stop;
   end
endmodule
```

2) DE1-SoC.sv

```
// Peter Zhong
// 5/5/2021
// EE 371
// Lab #3. Task 1

// DE1_SoC instantiates VGA_framebuffer and line_drawer modules, and draws an arbitrary line
// on the VGA display output. The coordinates of that arbitrary line is defined within the
// DE1_SoC module.

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(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);

   // draw lines between (x0, y0) and (x1, y1)
   line_drawer lines (.clk(CLOCK_50), .reset(~KEY[0]),
            .x0, .y0, .x1, .y1, .x, .y);

   // draw an arbitrary line
   assign x0 = 56;
   assign y0 = 456;
   assign x1 = 369;
   assign y1 = 145;

   // for testbench
   // assign x0 = 1;
   // assign y0 = 1;
   // assign x1 = 15;
   // assign y1 = 6;
   assign pixel_color = SW[0] ? 1'b1 : 1'b0;

endmodule
```

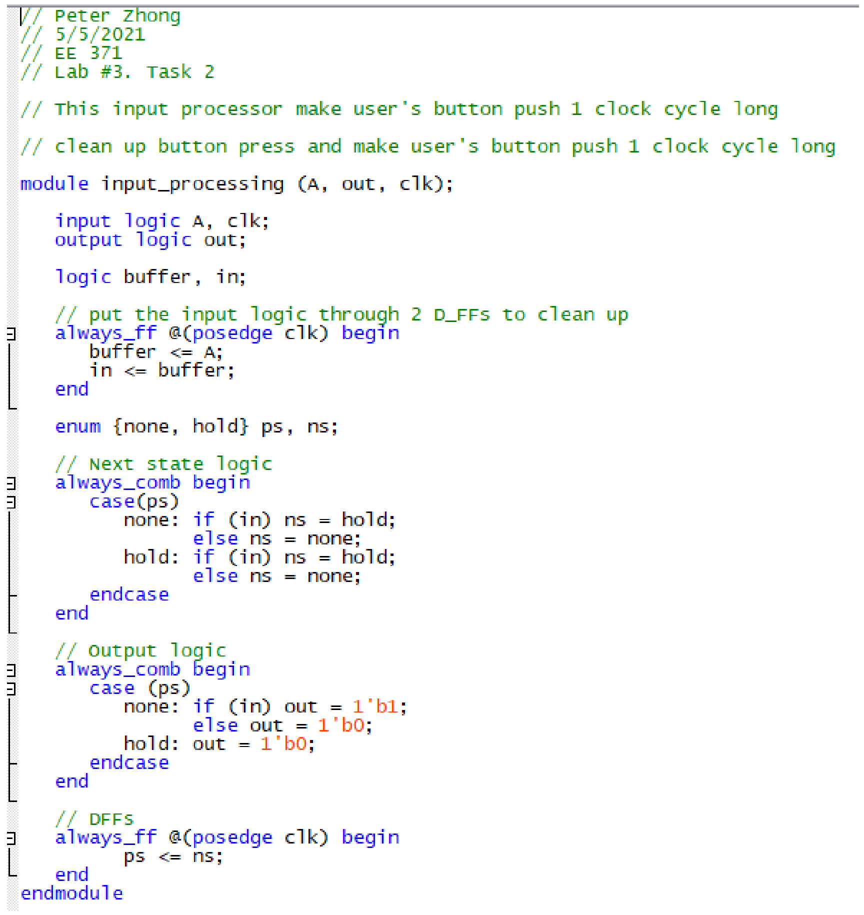
```
// Testbench for the top-level module
module DE1_SoC_testbench();
   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [9:0] LEDR;
   logic [3:0] KEY;
   logic [9:0] SW;

   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 (.*);

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   logic reset;
   assign KEY[0] = ~reset;

   initial begin
      reset <= 1; SW[0] <= 0; repeat (5) @(posedge CLOCK_50);
      reset <= 0; SW[0] <= 1; repeat(500) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

3) VGA_framebuffer.sv

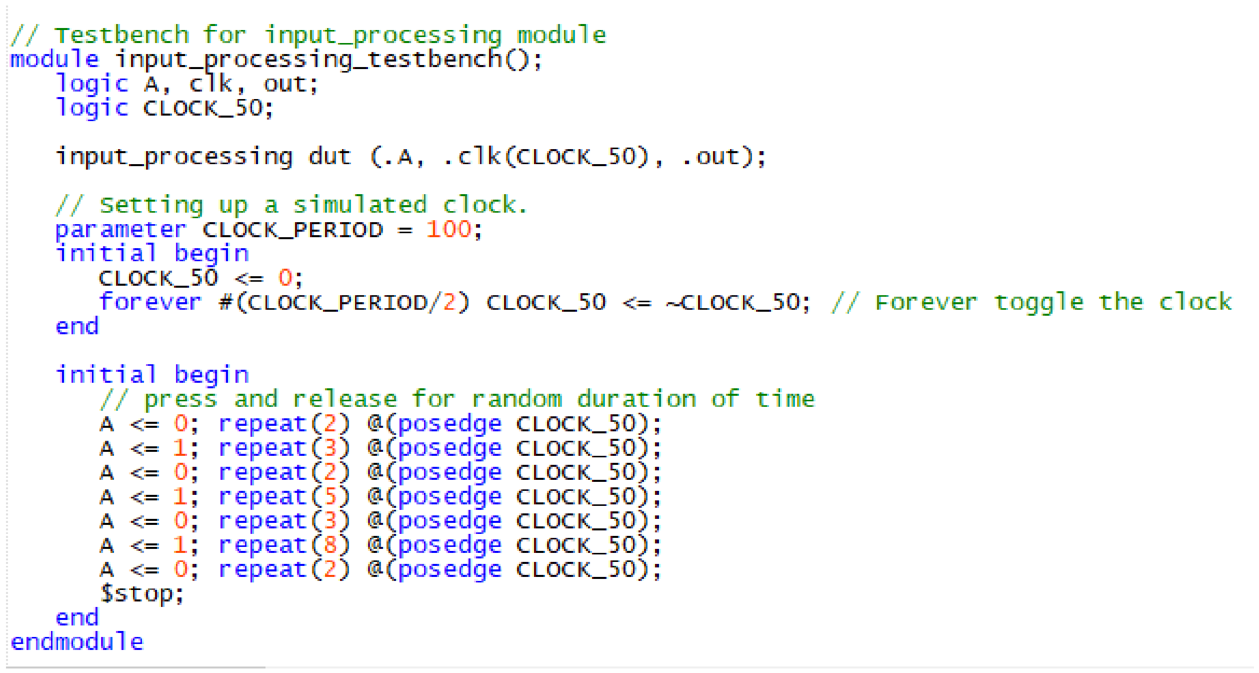
```
// Peter Zhong
// 5/5/2021
// EE 371
// Lab #3. Task 1 and 2

// 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;

   // Double-buffering.
   logic buffer[640*480*2-1:0];
   logic [19:0] wr_addr, rd_addr;
   logic rd_data;

   assign wr_addr = {y * 19'd640 + x, (!front_odd & dfb_en)};
   assign rd_addr = {v_count * 19'd640 + (h_count / 19'd2), (front_odd & dfb_en)};

   always_ff @(posedge clk) begin
      if (pixel_write) buffer[wr_addr] <= pixel_color;
      if (VGA_CLK) begin
         rd_data <= buffer[rd_addr];
         VGA_BLANK_N <= ~blank;
      end
   end

   // Color output.
   assign {VGA_R, VGA_G, VGA_B} = rd_data ? 24'hFFFFFF : 24'h000000;
endmodule
```

Task #2

1) input_processing.sv

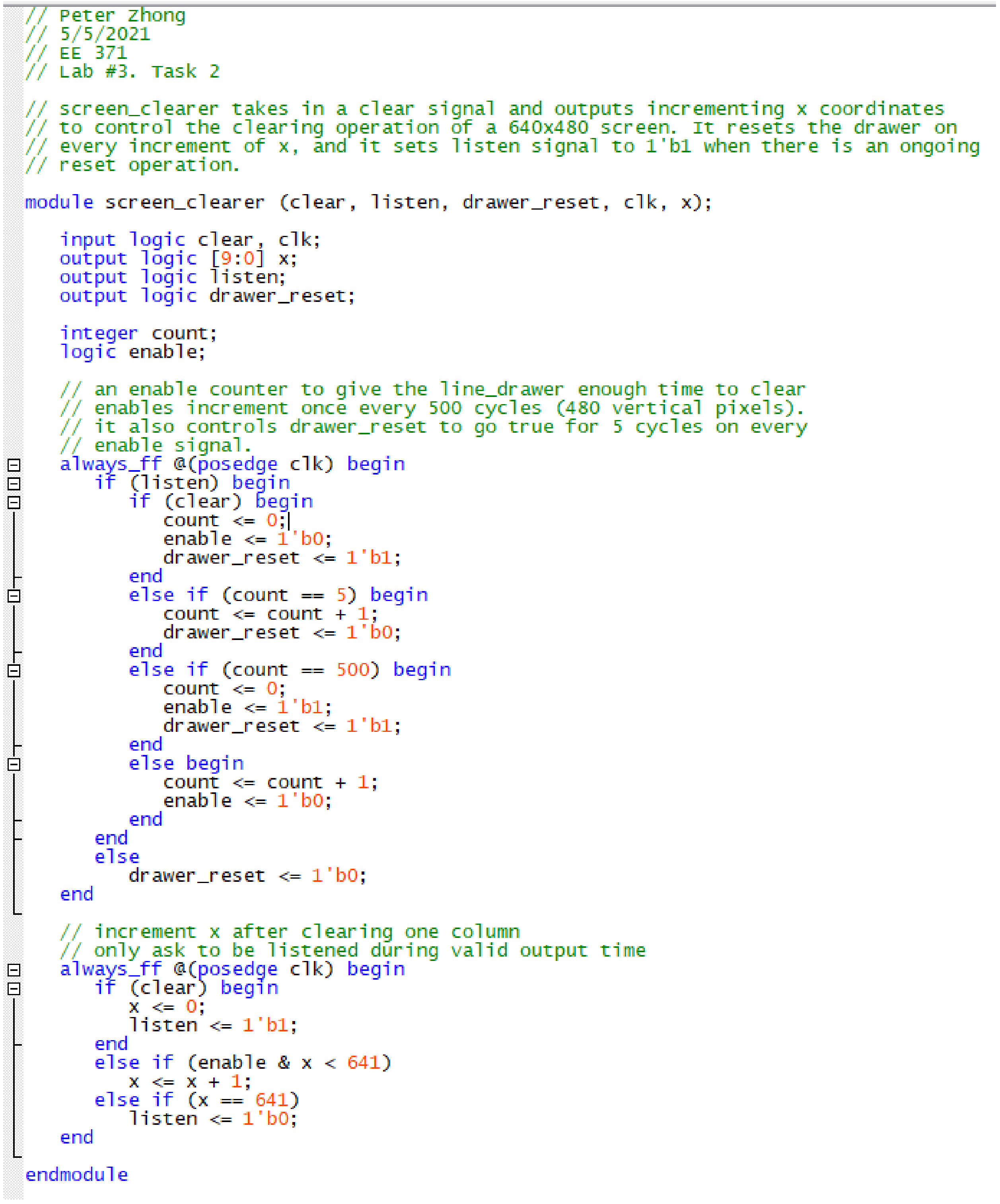


```
// Peter Zhong
// 5/5/2021
// EE 371
// Lab #3. Task 2

// This input processor make user's button push 1 clock cycle long

// clean up button press and make user's button push 1 clock cycle long

module input_processing (A, out, clk);

   input logic A, clk;
   output logic out;

   logic buffer, in;

   // put the input logic through 2 D_FFs to clean up
   always_ff @(posedge clk) begin
      buffer <= A;
      in <= buffer;
   end

   enum {none, hold} ps, ns;

   // Next state logic
   always_comb begin
      case(ps)
         none: if (in) ns = hold;
               else ns = none;
         hold: if (in) ns = hold;
               else ns = none;
      endcase
   end

   // Output logic
   always_comb begin
      case (ps)
         none: if (in) out = 1'b1;
               else out = 1'b0;
         hold: out = 1'b0;
      endcase
   end

   // DFFs
   always_ff @(posedge clk) begin
         ps <= ns;
   end
endmodule
```

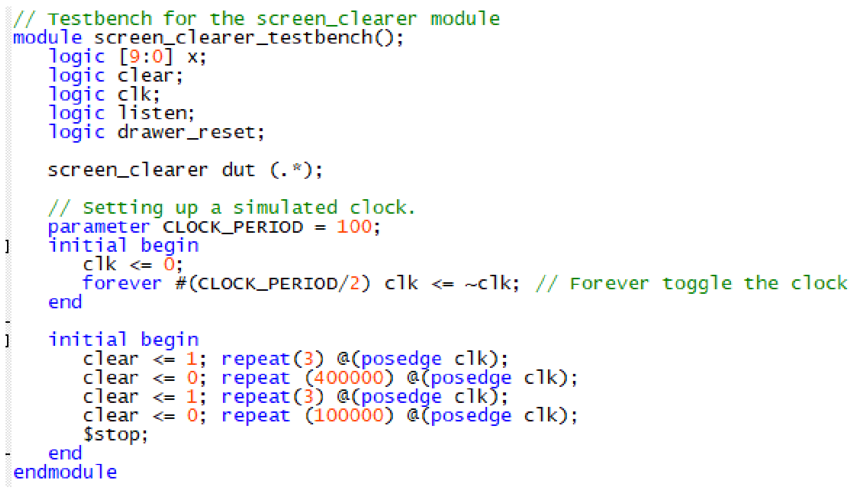


```
// Testbench for input_processing module
module input_processing_testbench();
   logic A, clk, out;
   logic CLOCK_50;

   input_processing dut (.A, .clk(CLOCK_50), .out);

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   initial begin
      // press and release for random duration of time
      A <= 0; repeat(2) @(posedge CLOCK_50);
      A <= 1; repeat(3) @(posedge CLOCK_50);
      A <= 0; repeat(2) @(posedge CLOCK_50);
      A <= 1; repeat(5) @(posedge CLOCK_50);
      A <= 0; repeat(3) @(posedge CLOCK_50);
      A <= 1; repeat(8) @(posedge CLOCK_50);
      A <= 0; repeat(2) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

2) screen_clearer.sv

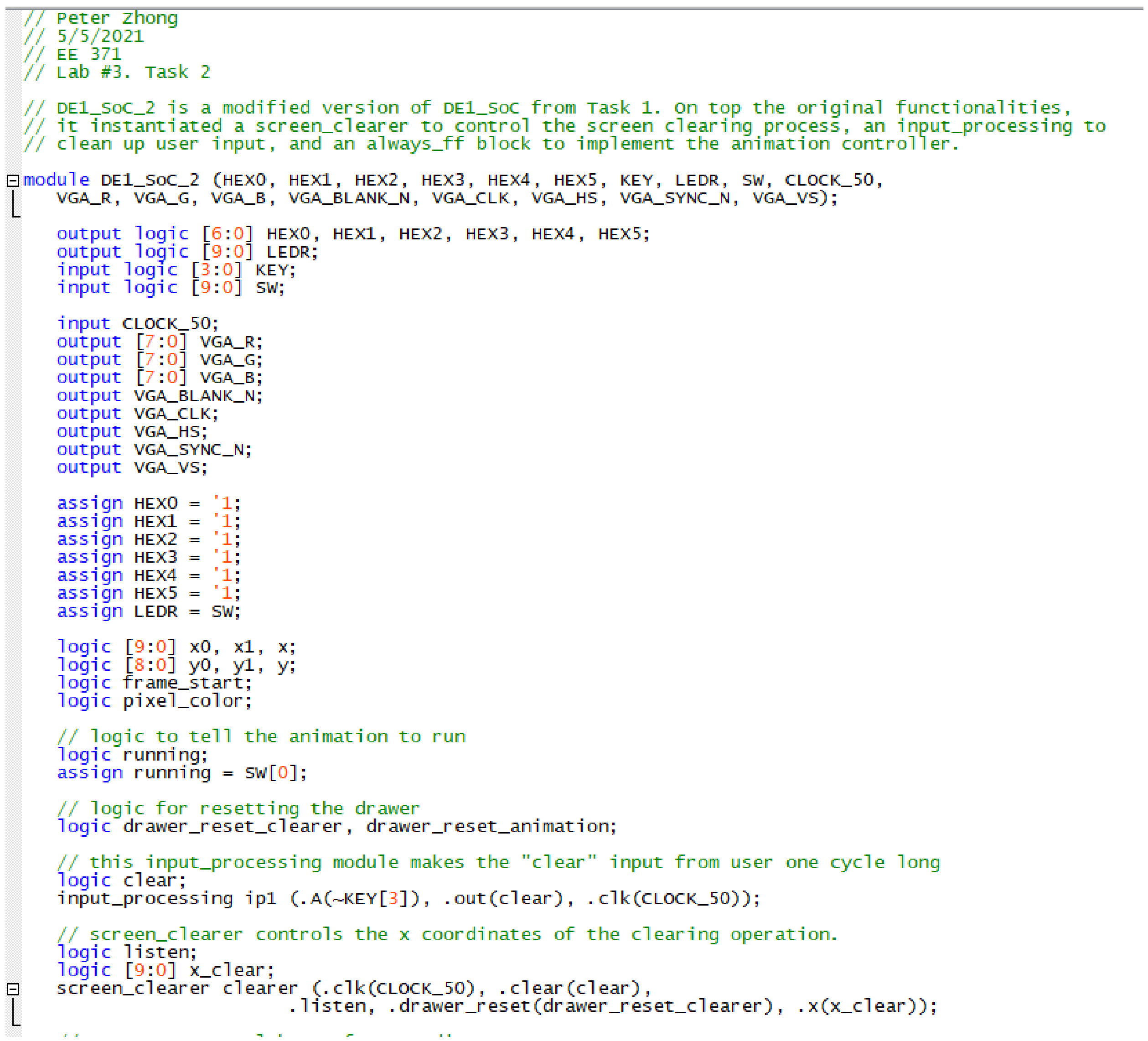


```systemverilog
// Peter Zhong
// 5/5/2021
// EE 371
// Lab #3. Task 2

// screen_clearer takes in a clear signal and outputs incrementing x coordinates
// to control the clearing operation of a 640x480 screen. It resets the drawer on
// every increment of x, and it sets listen signal to 1'b1 when there is an ongoing
// reset operation.

module screen_clearer (clear, listen, drawer_reset, clk, x);

   input logic clear, clk;
   output logic [9:0] x;
   output logic listen;
   output logic drawer_reset;

   integer count;
   logic enable;

   // an enable counter to give the line_drawer enough time to clear
   // enables increment once every 500 cycles (480 vertical pixels).
   // it also controls drawer_reset to go true for 5 cycles on every
   // enable signal.
   always_ff @(posedge clk) begin
      if (listen) begin
         if (clear) begin
            count <= 0;
            enable <= 1'b0;
            drawer_reset <= 1'b1;
         end
         else if (count == 5) begin
            count <= count + 1;
            drawer_reset <= 1'b0;
         end
         else if (count == 500) begin
            count <= 0;
            enable <= 1'b1;
            drawer_reset <= 1'b1;
         end
         else begin
            count <= count + 1;
            enable <= 1'b0;
         end
      end
      else
         drawer_reset <= 1'b0;
   end

   // increment x after clearing one column
   // only ask to be listened during valid output time
   always_ff @(posedge clk) begin
      if (clear) begin
         x <= 0;
         listen <= 1'b1;
      end
      else if (enable & x < 641)
         x <= x + 1;
      else if (x == 641)
         listen <= 1'b0;
   end

endmodule
```



```
// Testbench for the screen_clearer module
module screen_clearer_testbench();
   logic [9:0] x;
   logic clear;
   logic clk;
   logic listen;
   logic drawer_reset;

   screen_clearer dut (.*);

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

   initial begin
      clear <= 1; repeat(3) @(posedge clk);
      clear <= 0; repeat (400000) @(posedge clk);
      clear <= 1; repeat(3) @(posedge clk);
      clear <= 0; repeat (100000) @(posedge clk);
      $stop;
   end
endmodule
```

3) DE1_SoC_2.sv



```
// Peter Zhong
// 5/5/2021
// EE 371
// Lab #3. Task 2

// DE1_SoC_2 is a modified version of DE1_SoC from Task 1. On top the original functionalities,
// it instantiated a screen_clearer to control the screen clearing process, an input_processing to
// clean up user input, and an always_ff block to implement the animation controller.

module DE1_SoC_2 (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;

   // logic to tell the animation to run
   logic running;
   assign running = SW[0];

   // logic for resetting the drawer
   logic drawer_reset_clearer, drawer_reset_animation;

   // this input_processing module makes the "clear" input from user one cycle long
   logic clear;
   input_processing ip1 (.A(~KEY[3]), .out(clear), .clk(CLOCK_50));

   // screen_clearer controls the x coordinates of the clearing operation.
   logic listen;
   logic [9:0] x_clear;
   screen_clearer clearer (.clk(CLOCK_50), .clear(clear),
                           .listen, .drawer_reset(drawer_reset_clearer), .x(x_clear));
```

```
// muxes to control buses for coordinates
logic [9:0] x0_bus, x1_bus;
logic [8:0] y0_bus, y1_bus;
// logic for listening either to the clearer or to the animation generator
assign x0_bus = listen ? x_clear : x0;
assign y0_bus = listen ? 0 : y0;
assign x1_bus = listen ? x_clear : x1;
assign y1_bus = listen ? 480 : y1;

// RAM for coordinates of lines
logic [9:0] x0_coordinates [0:9];
logic [8:0] y0_coordinates [0:9];
logic [9:0] x1_coordinates [0:9];
logic [8:0] y1_coordinates [0:9];
// assign values to the x0 and y0 coordinates
assign x0_coordinates[0] = 320;
assign x0_coordinates[1] = 220;
assign x0_coordinates[2] = 220;
assign x0_coordinates[3] = 420;
assign x0_coordinates[4] = 420;
assign x0_coordinates[5] = 320;
assign x0_coordinates[6] = 220;
assign x0_coordinates[7] = 420;
assign x0_coordinates[8] = 220;
assign x0_coordinates[9] = 420;
assign y0_coordinates[0] = 160;
assign y0_coordinates[1] = 220;
assign y0_coordinates[2] = 320;
assign y0_coordinates[3] = 320;
assign y0_coordinates[4] = 220;
assign y0_coordinates[5] = 160;
assign y0_coordinates[6] = 320;
assign y0_coordinates[7] = 220;
assign y0_coordinates[8] = 220;
assign y0_coordinates[9] = 320;
// assign values to the x1 and y1 coordinates
assign x1_coordinates[0] = 220;
assign x1_coordinates[1] = 220;
assign x1_coordinates[2] = 420;
assign x1_coordinates[3] = 420;
assign x1_coordinates[4] = 320;
assign x1_coordinates[5] = 220;
assign x1_coordinates[6] = 420;
assign x1_coordinates[7] = 220;
assign x1_coordinates[8] = 420;
assign x1_coordinates[9] = 320;
assign y1_coordinates[0] = 220;
assign y1_coordinates[1] = 320;
assign y1_coordinates[2] = 320;
assign y1_coordinates[3] = 220;
assign y1_coordinates[4] = 160;
assign y1_coordinates[5] = 320;
assign y1_coordinates[6] = 220;
assign y1_coordinates[7] = 220;
assign y1_coordinates[8] = 320;
assign y1_coordinates[9] = 160;

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

// paint in black if clearing, paint in white otherwise
assign pixel_color = listen ? 1'b0 : 1'b1;
```

```
   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);

   // draw lines between (x0, y0) and (x1, y1)
   line_drawer lines (.clk(CLOCK_50), .reset(drawer_reset_clearer | drawer_reset_animation),
            .x0(x0_bus), .y0(y0_bus), .x1(x1_bus), .y1(y1_bus), .x, .y);

   // use count to count clock cycle, index to keep track of address of RAM to read off of
   integer count, index;
   integer clock_cycle = 50000000; // frequency of the clock for CLOCK_50
   // integer clock_cycle = 500; // frequency of the clock for testbench
   // logic to control the animation
   always_ff @(posedge CLOCK_50) begin
      if (clear) begin
         count <= 0;
         index <= 0;
         // set to inactive coordinates
         x0 <= 641;
         x1 <= 641;
         y0 <= 481;
         y1 <= 481;
      end
      else if (count == 1) begin // tell drawer to update as soon as there is a new line to draw
         count <= count + 1;
         drawer_reset_animation <= 1'b1;
      end
      else if (count == 3) begin // turn off the update signal
         count <= count + 1;
         drawer_reset_animation <= 1'b0;
      end
      else if (count == (clock_cycle / 2)) begin // update index every half a second
         count <= 0;
         if (index < 9)
            index <= index + 1;
         else
            index <= 0;
         // update coordinates
         x0 <= x0_coordinates[index];
         x1 <= x1_coordinates[index];
         y0 <= y0_coordinates[index];
         y1 <= y1_coordinates[index];
      end
      else if (running) begin
         count <= count + 1;
      end
   end
endmodule

// Testbench for the top-level module
module DE1_SoC_2_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_2 dut (.*);

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   logic clear, running;
   assign KEY[3] = ~clear;
   assign SW[0] = running;
   initial begin
      clear <= 1; running <= 0;         repeat(5) @(posedge CLOCK_50);
      clear <= 0;                                 @(posedge CLOCK_50);
      clear <= 1;                 repeat (400000) @(posedge CLOCK_50); // clear the VGA
      clear <= 0; running <= 1;   repeat(5000) @(posedge CLOCK_50); // let the animation run
      $stop;
   end
endmodule
```