Ziyi Ma (1001479777)
Suah Lee (1003180149)
Lab Section: Monday 9-12
TA: Zissis Poulos

THE MAZE
ECE241 PROJECT REPORT

1. Introduction

To demonstrate the knowledge learned through ECE241's lectures and labs, we built upon existing lab modules such as the ram block, FSM and datapath to create a maze game. The user will input the desired direction (left, right, up, or down) with the DE1-Soc key buttons to move the character through the maze to find the exit. The major milestones and accomplishments for our project is listed in the table below:

MileStone	Verilog	Functionality	VGA Display
1	● FSM module for pixel movement ● VGA modules for pixel and maze display ● Ram block modules for new maze background	● A single pixel can move either up, down, left or right based on user input	● Basic maze background ● A single coloured pixel can shift on the screen
2	● More FSM states for 2x2 square movement and deletion ● Added RAM block for boundary detection	● Boundaries are set for the maze so player cannot walk through walls ● Four pixels can move left, right, up or down based on user input	● A new updated maze background ● Four pixels can shift on the screen
3	● Optimized FSM states and state table ● Additional RAM block and counter added to switch between reset, play and beginning screens	● Start, maze and end screens were added ● The player can be reset to the desired location	● Three distinct screens: Start, maze and end ● Cartoon Characters were added to the maze background

Table 1.1: Summary of Milestones

2. The Design

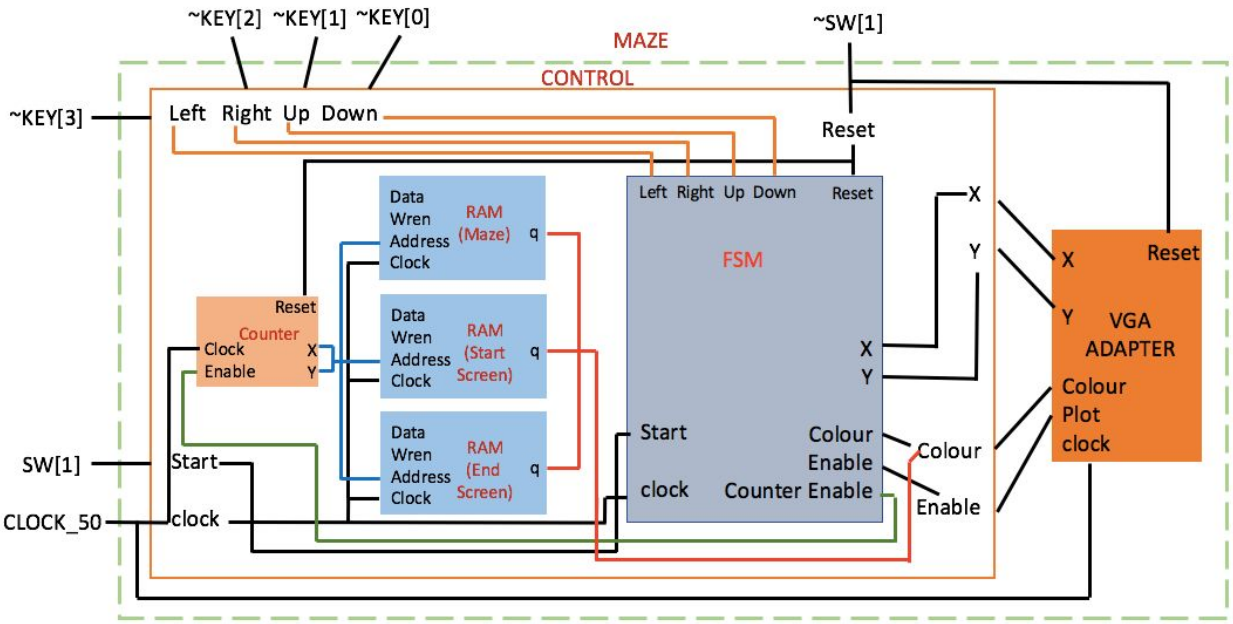


Table 2.1: Block Diagram

1.0 Maze Module

The Maze module is our top level module and it instantiates the VGA Adapter and the control module. The VGA Adapter displays our graphics to the VGA Display (monitor). The top module also connects the correct switches and keys on the DE1-Soc board as inputs to our game.

2.0 Control Module

The Control Module instantiates and connects the counter, multiple RAM blocks, and finite state machine (FSM).

2.1 Counter

The counter module is used to access and display the images stored inside the RAM blocks. It counts from (0,0) to (159,119), the aspect ratio of our VGA display. When the FSM wants to call a certain image, it enables the counter and the counter starts from x=0 y=0 and increments x by 1 until the edge of the screen is reached at x=159. The counter then resets x and increments y by 1. This process is repeated until x=159 and y=119. At the same time the counter increments, we used the relationship: RAM Address = (y*160) +x to to convert the x y coordinates to reference the memory address for the correct RAM with the desired image. The colour of the pixel at the certain address is received and is then used as inputs to the VGA adapter along with the x , y locations of the pixel on the screen. The counter ensures that all the pixels inside the RAM block are properly plotted to the VGA display.

2.2 RAM

There are three RAM blocks as we used three different images for our game: start screen, maze, and end screen (“Play Again” screen). The images are stored inside the RAM blocks and counter is used to display the images on the VGA Display. The RAM for the Maze is also used to set the boundaries for the user. The FSM checks the RAM of the maze image if the user is beside a wall and ensures that the user cannot move forward through a wall after the user direction input.

2.3 FSM

The FSM Module consisted of 19 states used to delete, move and colour pixels as well as shifting between the screens. We divided the state table into four functionality sections; waiting state, player deletion, player movement and update/redraw player. The waiting states consisted of start, play and end screens which waited for the user’s key input, otherwise, it looped in the current state. Once a key is pressed, the game will switch from the start screen to “play” mode called the A_START state. Inside the A_START state, the game will wait again for a positive edge trigger from one of the keys which indicated player movement in a certain direction. The game will then loop in DEL_WAIT state and react on the negative edge trigger of the pressed key before moving on to its first deletion state. The deletion states will start from a reference point of the pixel and return to the reference point after it looped throughout the entire player image and colored every pixel black. After deletion, the player will be redrawn on the screen with the updated x or y coordinate corresponding to the specific key pressed at the A_START state. Once the player reaches the end of the maze, the “play again” screen will be displayed on the VGA until KEY[3] is pressed to return to the “play” mode.

3. Report on Success

3.1 Accomplishments

In the end, we successfully completed our 241 project with a fully functioning maze game. Our player pixel can move left, right, up or down based on the key input from the DE1_soc board. We first begin with a start screen which asked the user for a KEY[3] input before moving on to the maze. When the player reaches the end of the maze, the user will be asked to play again with a positive KEY[3] edge trigger to return to the maze screen. At last, we achieved all of our milestones and completed our maze game project.

3.2 Problems and Resolutions

3.2.1 Incorrect VGA Display

During Milestone 1, we encountered a glitch where multiple pixels would fly across the screen when one key was pressed by the user. The solution was to add an additional waiting state in between the A_START state and the DEL state in the FSM. This problem occurred because of human reaction time in comparison to machine reaction time. The clock we used for this project was CLOCK_50 which meant 50 million clock cycles per second. While the A_START state

waited only for a single positive edge of one of the four keys during one clock cycle, humans are not fast enough to generate a single key edge in the duration of a single clock cycle. Instead, this translated to multiple movements upon one key trigger which showed up on the monitor as the flying pixel. To resolve this problem, we added an additional DEL_WAIT state right after A_START to wait for the negative edge of the pressed key before continuing to the next logic state. In the end, our player object would only shift 1 bit in the x or y direction with each key trigger which is our desired output.

3.2.2 Redundant FSM States

After completing our milestone 2, we decided to reduce the number of FSM states. Errors arose due to excessive trigger of the VGA module. At the time, the FSM we had was lengthy with 45 states in total and 10 states each for up, down, left and right movement. Since most of the states had redundant functionalities, we decided to optimize our finite state machine by combining logic. In the end, we were left with 19 essential states; five states for deletion, nine for drawing and five more used for transition purposes. After fixing our FSM, the original glitches disappeared and in addition, we were able to add other states such as reset, Start and finish screens that improved the overall design of our project.

3.2.3 Flickering Images

In order to display the images stored inside the RAM blocks, we initially decided to instantiate a counter that counts from 0 to 19199 to access the address of the RAM. Since the RAM blocks do not reference the pixel locations as x and y coordinates like the VGA Adapter does, we had to determine a method to convert the memory address into x,y coordinates. This was important as the VGA adapter took x and y locations of the pixel as its input. Thus, we derived a relationship where x = (Address of RAM) % 160 and y = (Address of RAM) / 160. For example, address 161 of the RAM block would be (1,1) as coordinates on the VGA display screen. Although this worked and correctly displayed the images that were stored in the RAM blocks to the monitor, it was very glitchy and the images continuously flickered aggressively. However, we were able to fix this glitch as I have remembered from the lectures that the division and remainder operators are very complex in hardware, requiring many clock cycles. Therefore, we changed the counter so that it would count from (0,0) to (160,120) for the inputs of the VGA adapter. Since the multiplication and addition operators are much simpler than division and remainder operator, we decided to access the RAM addresses with the relationship:
Address of RAM = (y*160)+x. After this modification, the images displayed on the VGA became stable.

4. Improvements for Next time

4.1 Bigger pixel/user

If we were given the opportunity to redo the project, we would modify the FSM to create a bigger player character with different shapes to make it more aesthetically pleasing. In addition, it would require less time to move around the maze with a bigger character because of the maze and character size comparison.

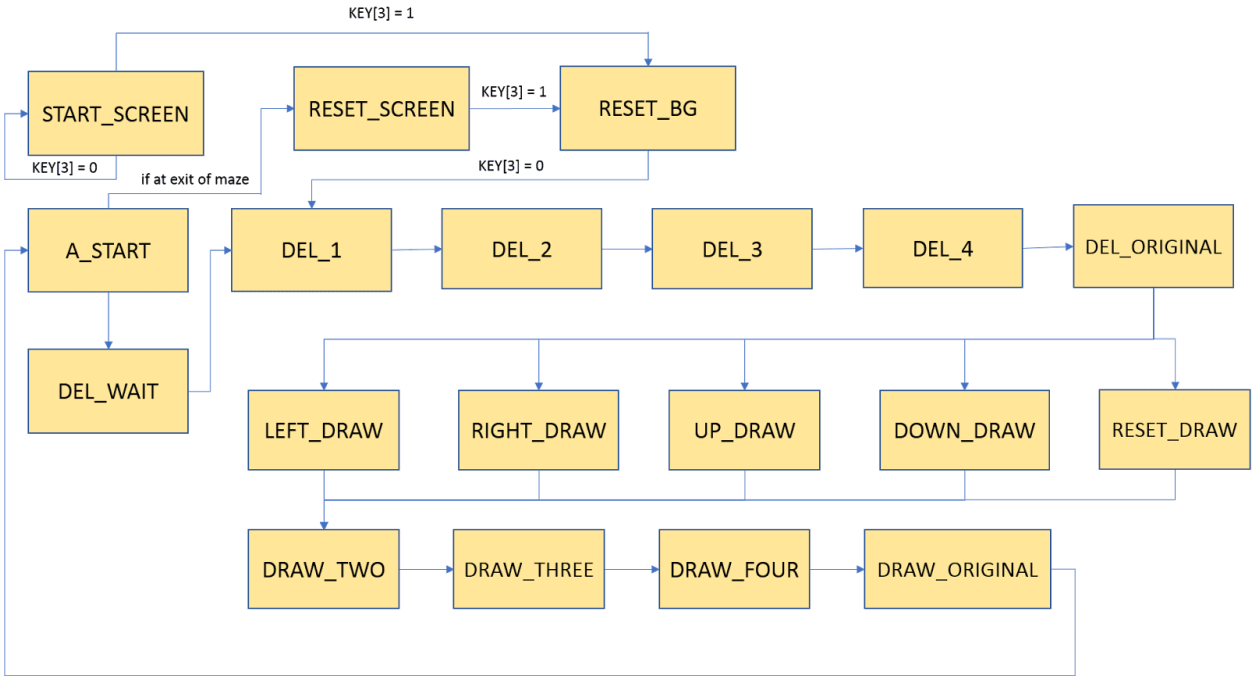
4.2 Optimize FSM

One of the major modifications we had to make before completing the project was to combine different logic states inside the FSM. If we could start all over again, we would have researched FSM structures online before attempted to write our own. We spent a lot of time rewriting the FSM towards the end because it was hard to write a fully functional FSM on the first try without experience and practice.

5. Conclusion

In conclusion, we designed a fully functioning maze game for our ECE241 final project. Through communication and task division, we ensured our deadlines were met for each milestone. We each worked individually on certain aspects of the design and made sure our parts were fully functional before bringing it together on the final copy. Through this final project, we were able to learn more about the applications of hardware and fully experience the design aspect of an electrical and computer engineer. At last, we were able to finish our maze game because of our time management, organizational and collaborative skills which resulted in the successful completion of our project.

Appendix



Verilog Code

```
module Maze (CLOCK_50, KEY, SW,
VGA_CLK,                    // VGA Clock
VGA_HS,                     // VGA H_SYNC
VGA_VS,                     // VGA V_SYNC
VGA_BLANK_N,                // VGA BLANK
VGA_SYNC_N,                 // VGA SYNC
VGA_R,                      // VGA Red[9:0]
VGA_G,                      // VGA Green[9:0]
VGA_B                       // VGA Blue[9:0]
);

input CLOCK_50;             // 50 MHz
input [3:0]KEY;             //User direction inputs
input [1:0]SW;              //SW[0] resets the VGA, SW[1] resets the game

output VGA_CLK;             // VGA Clock
output VGA_HS;              // VGA H_SYNC
output VGA_VS;              // VGA V_SYNC
output VGA_BLANK_N;         // VGA BLANK
output VGA_SYNC_N;          // VGA SYNC
output [9:0] VGA_R;         // VGA Red[9:0]
```

```
output [9:0] VGA_G;             // VGA Green[9:0]
output [9:0] VGA_B;             // VGA Blue[9:0]

wire [5:0] colour;
wire [7:0] x_out;
wire [7:0] y_out;

wire writeEn;
wire start;
wire clock_pulse;
wire left, right, up, down;

assign start = SW[1];

control u0(
    .clk(CLOCK_50),
    .resetn(~SW[1]),
    .left(~KEY[3]), .right(~KEY[2]), .up(~KEY[1]), .down(~KEY[0]),
    .start(start),
    .x(x_out[7:0]),.y(y_out[7:0]),      // the x,y coordinates of pixel to be plotted
    .c(colour[5:0]),                    // this is the colour
    .enable(writeEn),
    .LEFT(left),
    .RIGHT(right),
    .UP(up),
    .DOWN(down)
 );

 vga_adapter VGA(
    .resetn(~SW[0]),
    .clock(CLOCK_50),
    .colour(colour),
    .x(x_out),
    .y(y_out),
    .plot(writeEn),
                                        // Signals for the DAC to drive the monitor.
    .VGA_R(VGA_R),
    .VGA_G(VGA_G),
    .VGA_B(VGA_B),
    .VGA_HS(VGA_HS),
    .VGA_VS(VGA_VS),
    .VGA_BLANK(VGA_BLANK_N),
    .VGA_SYNC(VGA_SYNC_N),
```

```
    .VGA_CLK(VGA_CLK));
defparam VGA.RESOLUTION = "160x120";
defparam VGA.MONOCHROME = "FALSE";
defparam VGA.BITS_PER_COLOUR_CHANNEL = 2;
defparam VGA.BACKGROUND_IMAGE = "maze_2.mif";  //maze background
endmodule

module control(
   input clk,
   input resetn,
   input left, right, up, down,
   input start,

   output reg [7:0]x,y,
   output reg [5:0]c,//this is the colour
   output reg enable,//enables the drawer
   output reg UP, DOWN, LEFT, RIGHT
   );

wire Wdata, Wwren;
wire [5:0]BGcolour;
wire [5:0]resetBG;
assign Wdata = 1'b0;
assign Wwren = 1'b0;
wire[14:0]Waddress;
reg [7:0]wirex;
reg [6:0]wirey;

assign Waddress = (wirey * 8'd160) + wirex;
reg endScreenEnable;
reg bgEnable;
reg StartEnable;
wire [14:0]countOUT,countBG, countStart;
wire [5:0] PGcolour, StartColour;
wire [6:0]wy,wyBG, wyStart;
wire [7:0]wx,wxBG, wxStart;

endScreenCounter c1(resetn, clk, endScreenEnable, wx, wy);
assign countOUT = (wy * 8'd160) + wx;
endScreenCounter c2(resetn, clk, bgEnable, wxBG, wyBG);
assign countBG = (wyBG * 8'd160) + wxBG;
endScreenCounter c3(resetn, clk, StartEnable, wxStart, wyStart);
assign countStart = (wyStart * 8'd160) + wxStart;
```

```
//RAM block stored with end screen
playagain pg(
	.data(0),
	.wren(0),
	.address(countOUT),
	.clock(clk),
	.q(PGcolour));

//RAM block stored with maze. Checks for boundaries
image bg1(
	.data(0),
	.wren(0),
	.address(Waddress),
	.clock(clk),
	.q(BGcolour));

//RAM block stored with maze. Displays Maze background again.
 image bg2(
	.data(0),
	.wren(0),
	.address(countBG),
	.clock(clk),
	.q(resetBG));

//RAM block stored with start screen. Displays Start Screen
	 start bg3(
	.data(0),
	.wren(0),
	.address(countStart),
	.clock(clk),
	.q(StartColour));

reg [6:0] current_state, next_state;

localparam
	START_SCREEN   = 6'd0, //Displays Start Screen
	A_START          = 6'd1, //The starting state
	DEL_WAIT         = 6'd2, //First Deleting State - Waiting for the negative edge of the keys
	RESET_SCREEN   = 6'd3, //Displays “Play Again Screen”
	RESET_BG         = 6'd4, //Displays Maze image
	DEL_1            = 6'd5,//Second Deleting State - deletes the reference pixel
```

```
        DEL_2                = 6'd6,//Third Deleting State - deletes ref pixel + 1'b1 in the x direction
        DEL_3                = 6'd7,//Fourth Deleting State - deletes ref pixel + 1'b1 in the y direction
        DEL_4                = 6'd8,//Fifth Deleting State - deletes ref pixel - 1'b1 in the x direction
        DEL_ORIGINAL   = 6'd9,//Sixth Deleting State - return to reference pixel location
        LEFT_DRAW        = 6'd10,//Move Left State - move reference pixel -1'b1 in the x direction
        RIGHT_DRAW     = 6'd11,//Move Right State - move reference pixel 1'b1 in the x direction
        UP_DRAW           = 6'd12,//Move Up State - move reference pixel -1'b1 in the y direction
        DOWN_DRAW     = 6'd13,//Move Down State - move reference pixel 1'b1 in the y direction
        RESET_DRAW     = 6'd14,//Reset State - move back to starting location
        DRAW_TWO         = 6'd15,//Second Draw State - draw reference pixel + 1'b1 in the x direction
        DRAW_THREE    = 6'd16, //Third Draw State - draw reference pixel + 1'b1 in the y direction
        DRAW_FOUR       = 6'd17,//Fourth Draw State - draw reference pixel -1'b1 in the x direction
        DRAW_ORIGINAL = 6'd18;//Fifth Draw State - return to the reference pixel location

// Next state logic aka our state table
always@(*)
begin: state_table
        case (current_state)
        START_SCREEN: begin
        if (left)
        next_state = RESET_BG;
        else
        next_state = START_SCREEN;
        end

        A_START: begin
        if((y == 8'd114 || y == 8'd115) && (x == 8'd73 || x== 8'd74 || x== 8'd75|| x== 8'd76|| x==
        8'd77|| x== 8'd78|| x== 8'd79 || x== 8'd80 )) begin

        next_state = RESET_SCREEN; //location of maze exit
        end
        else if(left) begin
        wirex = x - 8'd1; //the new locations of the pixel if it were to go left
        wirey = y;
        next_state = DEL_WAIT;
        //LEFT = 1'b1;
        end
        else if(right) begin
        wirex = x + 8'd2;
        wirey = y;
        next_state = DEL_WAIT;
        //RIGHT = 1'b1;
        end
        else if(up) begin
```

```
wirex = x;
		wirey = y - 8'd1;
		 next_state = DEL_WAIT;
		 //UP = 1'b1;
		end
		else if(down) begin
		wirex = x;
		wirey = y + 8'd2  ;
		 next_state = DEL_WAIT;
		 //DOWN = 1'b1;
		end
		else
		 next_state = A_START;
		end
DEL_WAIT: begin
		if(left == 1'b0 && right == 1'b0 && up == 1'b0 && down == 1'b0) begin
			next_state = DEL_1;
		end

		end
RESET_SCREEN: begin
//////////////////////////////// PRESS KEY[3] TO EXIT END SCREEN ////////////////////////////
if (left)
next_state = RESET_BG;
end
RESET_BG: begin
if(!left)
next_state = DEL_1;
end
DEL_1: begin
		next_state = DEL_2;
		end
DEL_2: begin
		next_state = DEL_3;
		end
DEL_3: begin
		next_state = DEL_4;
		end
DEL_4: begin
		next_state = DEL_ORIGINAL;
		end
DEL_ORIGINAL: begin
		if(LEFT == 1'b1) begin
```

```
            next_state = LEFT_DRAW;
           end
           else if(RIGHT == 1'b1) begin
            next_state = RIGHT_DRAW;
           end
           else if(UP == 1'b1) begin
            next_state = UP_DRAW;
           end

           else if(DOWN == 1'b1) begin
            next_state = DOWN_DRAW;
           end
           else begin
            next_state = RESET_DRAW;
           end
           end
LEFT_DRAW: begin
           next_state = DRAW_TWO;
           end
RIGHT_DRAW: begin
           next_state = DRAW_TWO;
           end
UP_DRAW: begin
           next_state = DRAW_TWO;
           end
DOWN_DRAW: begin
           next_state = DRAW_TWO;
           end
RESET_DRAW: begin
           next_state = DRAW_TWO;
           end
DRAW_TWO: begin
           next_state = DRAW_THREE;
           end
DRAW_THREE: begin
           next_state = DRAW_FOUR;
           end
DRAW_FOUR: begin
           next_state = DRAW_ORIGINAL;
           end
DRAW_ORIGINAL: begin
           next_state = A_START;
           end
```

```
      default: next_state = A_START;
        endcase
    end // state_table

always @(posedge clk)
 begin: enable_signals
 case (current_state)


 START_SCREEN: begin
 //load start screen
 StartEnable <= 1'b1;
 bgEnable <= 1'b0;
 endScreenEnable <= 1'b0;
 enable <= 1'b1;
 c[5:0] <= StartColour;
 x<=wxStart;
 y<=wyStart;
 end


 A_START: begin
 if(start) begin
  x <= 8'd76;
  y <= 8'd29;
  c <= 6'b110000;
  enable <= 1'b0;
  LEFT <= 1'b0;
  RIGHT <= 1'b0;
  UP <= 1'b0;
  DOWN <= 1'b0;
 end

 else if(left) begin
   LEFT <= 1'b1;
   c[5:0] <= 6'b000000;//make the color black
   enable <= 1'b0;//dont want to draw anything
 end

 else if(right) begin
  RIGHT <= 1'b1;
  c[2:0] <= 3'b000;//make the color black
  enable <= 1'b0;//dont want to draw anything
```

```
 end

 else if(up) begin
  UP <= 1'b1;
  c[2:0] <= 3'b000;//make the color black
  enable <= 1'b0;//dont want to draw anything
 end

 else if(down) begin
  DOWN <= 1'b1;
  c[2:0] <= 3'b000;//make the color black
  enable <= 1'b0;//dont want to draw anything
 end

 else begin
  c[2:0] <= 3'b000;//make the color black
  enable <= 1'b0;//dont want to draw anything
 end
 end

 DEL_WAIT: begin
           enable <= 1'b0;
 end

 RESET_SCREEN: begin
//load “play again” screen
 endScreenEnable <= 1'b1;
 enable <= 1'b1;
 c[5:0] <=  PGcolour;
 x<=wx;
 y<=wy;
 end

 RESET_BG: begin
//load maze image again
  endScreenEnable <= 0;
  bgEnable <= 1;
  enable <= 1'b1;
  c[5:0] <=  resetBG;
  x<=wxBG;
  y<=wyBG;

  end
```

```
 DEL_1: begin
//if the “new/future” locations of the pixel is not black (not a path), don’t move
 if (BGcolour != 6'b000000)
            enable <= 1'b0;
  else  begin
  c[5:0] <= 6'b000000;
  enable <= 1'b1;

  end
 end

 DEL_2: begin
  x <= x + 1'b1;
 end

 DEL_3: begin
  y <= y + 1'b1;
 end

 DEL_4: begin
  x <= x - 1'b1;
 end

 DEL_ORIGINAL: begin
  y <= y - 1'b1;
  enable <= 1'b0;
 end

 UP_DRAW: begin
  if (BGcolour != 6'b000000) begin
             enable <= 1'b0;
             UP <= 1'b0;
           end

  else  begin
  y <= y - 1'b1;
  c[5:0] <= 6'b110000;
  enable <= 1'b1;
  UP <= 1'b0;
  end
 end
```

```
 DOWN_DRAW: begin
  if (BGcolour != 6'b000000) begin
             enable <= 1'b0;
              DOWN <= 1'b0;
            end

  else  begin
  y <= y + 1'b1;
  c[5:0] <= 6'b110000;
  enable <= 1'b1;
  DOWN <= 1'b0;
  end
 end

 RIGHT_DRAW: begin
  if (BGcolour != 6'b000000) begin
             enable <= 1'b0;
             RIGHT <= 1'b0;

end
  else  begin
  x <= x + 1'b1;
  c[5:0] <= 6'b110000;
  enable <= 1'b1;
  RIGHT <= 1'b0;
  end
 end

 LEFT_DRAW: begin
  if (BGcolour != 6'b000000) begin
             enable <= 1'b0;
             LEFT <= 1'b0;
            end
  else  begin
  x <= x - 1'b1;
  c[5:0] <= 6'b110000;
  enable <= 1'b1;
  LEFT <= 1'b0;
  end
 end

 RESET_DRAW: begin
  if (BGcolour != 6'b000000)
```

```
            enable <= 1'b0;
 else  begin
          x <= 8'd76;
          y <= 8'd29;
          c[5:0] <= 6'b110000;
          enable <= 1'b1;
          end
end

 DRAW_TWO: begin
  x <= x + 1'b1;
 end
 DRAW_THREE: begin
  y <= y + 1'b1;
 end
 DRAW_FOUR: begin
  x <= x - 1'b1;
 end

 DRAW_ORIGINAL: begin
  enable <= 1'b0;
  y <= y - 1'b1;
  c[5:0] = 6'b110000;
 end

default: begin
 enable <= 1'b0;
 x <= 8'd76;
  y <= 8'd29;
 c <= 3'b000;
 end
endcase
end // enable_signals

// current_state registers
always@(posedge clk)
begin: state_FFs
if(!resetn) begin
current_state <= START_SCREEN;
end

else
current_state <= next_state;
```

```
end // state_FFS
endmodule



module endScreenCounter (reset, clock, enable, x, y);
input reset, clock, enable;
output reg [7:0]x;
output reg [6:0]y;

always@(negedge reset, posedge clock) begin
if (!reset) begin
x<=0;
y<=0;
end

else if(x==8'd159) begin
        if(y==7'd119) begin
        y<=0;
        x<=0;
        end
        else begin
        y<=y+7'd1;
        x<=0;
        end
end

else
x<=x+8'd1;


end

endmodule
```