ECE241 Final Report

Project Title	Piano Tiles
Tutorial section and station number	PRA05
TA	Roberto Dicecco
Prepared by	Shihan Zhang 1002055795 Ankita Singal 1002478838

Introduction

Description of Project [Appendix[1])

Our project was to build a single-player game similar to Piano Tiles - Don't Tap the Black Tile. We wanted to create a game that involved a piano since both of us love to play the piano. With that in mind, we decided to replicate a very well known game "Piano Tiles" for this project. The objective of the game is to tap on the white tiles as they appear from the top of the screen while avoiding tapping on the black tiles. We used the keyboard as our input and the monitor as output. The game consists of four columns, each of which are controlled by a key on the keyboard (A, S, D, F). The player presses the respective key as the white tiles reach the bottom of the screen. Upon hitting the key on a black tile, the player loses and and the game ends.

The Design **(Appendix[4] for block diagram)**

Controls

For this project, we used KEY[1] on the DE1 Board and four keys (A, S, D, F) on the keyboard as input to the game. KEY[1] on the DE1 Board is used to start the game. The four keys on the keyboard are used to tap the white tile as it reaches the bottom of the screen. Refer to Appendix [2] for the specific key assignment.

Verilog Code

The major task in this project was controlling the display since our game was all about animation, which required understanding the use of the VGA adapter. Our code consisted of a controller and datapath module that mapped out a FSM which defines the majority of flow of the game. We also had a few helper functions that were used to perform various tasks of the game. The modules are outlined in two separate tables below:

Table 1: Description of Major modules and their Functions

Module	Description
Control (Finite State Machine)	• Controls the flow of the game by defining the game state. • Some of the game states include start_game, choose_column, delete_old, start_animation, shift_down etc. • Sends load signals to the datapath module based on the game state. The load signals are used to perform actions like drawing a white tile during the game.

	• Receives control signals from datapath module which indicate the completion of the action item. These signals allows the FSM to proceed onto the next state.
Datapath	• Contains the algorithm for performing action items in the game • Examples of action items include: drawing and deleting a white tile as it moves down the screen, drawing game over screen, etc. • Contains multiple always blocks that decide the action based on the control signals received from the control module. • Determines the end of game if the user presses a key on the keyboard on a black tile. • Draws the game over screen from a RAM module. (Appendix[3]) • Computes score every time the user presses the key on a white tile. • Receives input from random number generator. Assigns values from RNG to a respective column.

Table 2: Description of Helper modules and their Functions

Module	Description
draw_pixel	• Contains an instance of vga_adapter that is used to send x,y coordinates and the colour for output to the monitor.
PS2_Controller	• Used to read input when a key on the keyboard is pressed. The data from keyboard is read every 2 milliseconds.
LSFR	• Contains timer that reads a random 13-bit number from RandomNum module every 6 milliseconds. • Sends the random number to datapath

	module to choose an appropriate column.
RandomNum (Linear Feedback Shift Register) (Appendix[5])	● Implements a shift register that operates on CLOCK_50. When clocked, it advances the signal through the register from one bit to the next most significant bit. ● A exclusive-OR is performed on two of the flip-flops and the output is feed back into the inputs of the first flip flop as shown in Appendix 5.
Ram32x4	● Stores the game over mif file in a 32768 word with 3 bits wide memory block. ● Used to read colour of bits for specific x,y coordinates when drawing the game over screen (Appendix[3]).
Hex_decoder	● Print a binary number on the hex display. ● Used to print the player's score on HEX[1] and HEX[0].

FSM States (Appendix [6])

I. **Start_Game:**
In this state, the game waits for the user to press KEY[1] on the DE1 Board to begin the game. The key triggers the start of the FSM.

II. **Choose_Column**
In this state, a 13-bit random number is read from the LSFR. Based on the random number, a column in which the next white tile will be dropped iis chosen.

III. **Delete_Old**
In this state, the white tile drawn on the screen is deleted by colouring it black. An 8-bit counter is used to iterate through each pixel of the white tile with bits [3:0] corresponding to the change in x coordinates and bits [7:4] corresponding to the change in y coordinates. The coordinates and the 3 bit colour black are sent to the VGA module.

IV. **Shift_Down**
In this state, the y-coordinate for the tile being drawn is incremented by one. The new value is stored in the register.

V. **Print_New**
In this state, a white tile is drawn on the screen for the new shifted y coordinate. An 8-bit counter is used to draw the tile with bits [3:0] corresponding to the change in x coordinates and bits [7:4] corresponding to the change in y coordinates. The coordinates and the 3 bit colour white are sent to the VGA module.

VI. **Done**
In this state, a check is performed to see if the tile has reached the bottom of the screen. If yes, then the next state is Choose_Column. Otherwise, the next state is Delete_Old. It

also performs a check to see if the game is over (i.e., the user pressed the key on a black tile and the game over flag is up). If yes, the FSM goes to the Game_Over state.

VII. **Game_Over**

In this state, the game over screen is drawn on the monitor. A 15-bit counter is used to keep track of the address of the pixel being drawn. Another variable x_counter counts up to 160 bits after which it is reset to 0 and the y_counter is incremented by 1. The colour of each pixel is read from the RAM module with the game over mif preloaded.

Report on success

Our measure of success is based on the working functionality of the game that was determined at the beginning of the project. There are a couple things that could not be achieved during the timeline of the project. The functionality is outlined below:

Functionality	Achieved?
The game starts when KEY[1] is pressed.	✓
The white tiles appear randomly from top of the screen.	✓
The project reads keyboard input A,S,D,F for 4 different columns.	✓
When the white tiles reaches the bottom and the corresponding key is pressed at the same time, score is computed and is displayed on hex increment.	✓
If the black tiles are hit, this game ends properly and displays the game over screen	✓
Generating audio every time a key is pressed on the keyboard that corresponds to a piano note Note: We were successfully able to load different sounds when switches [3:0] on the DE1 board were toggled. This was tested as independent project. When we integrated the audio code with our project, the sounds were not being produced. Due to lack of time, we couldn't debug the code to make the audio work with our project.	
Displaying the score on the monitor	

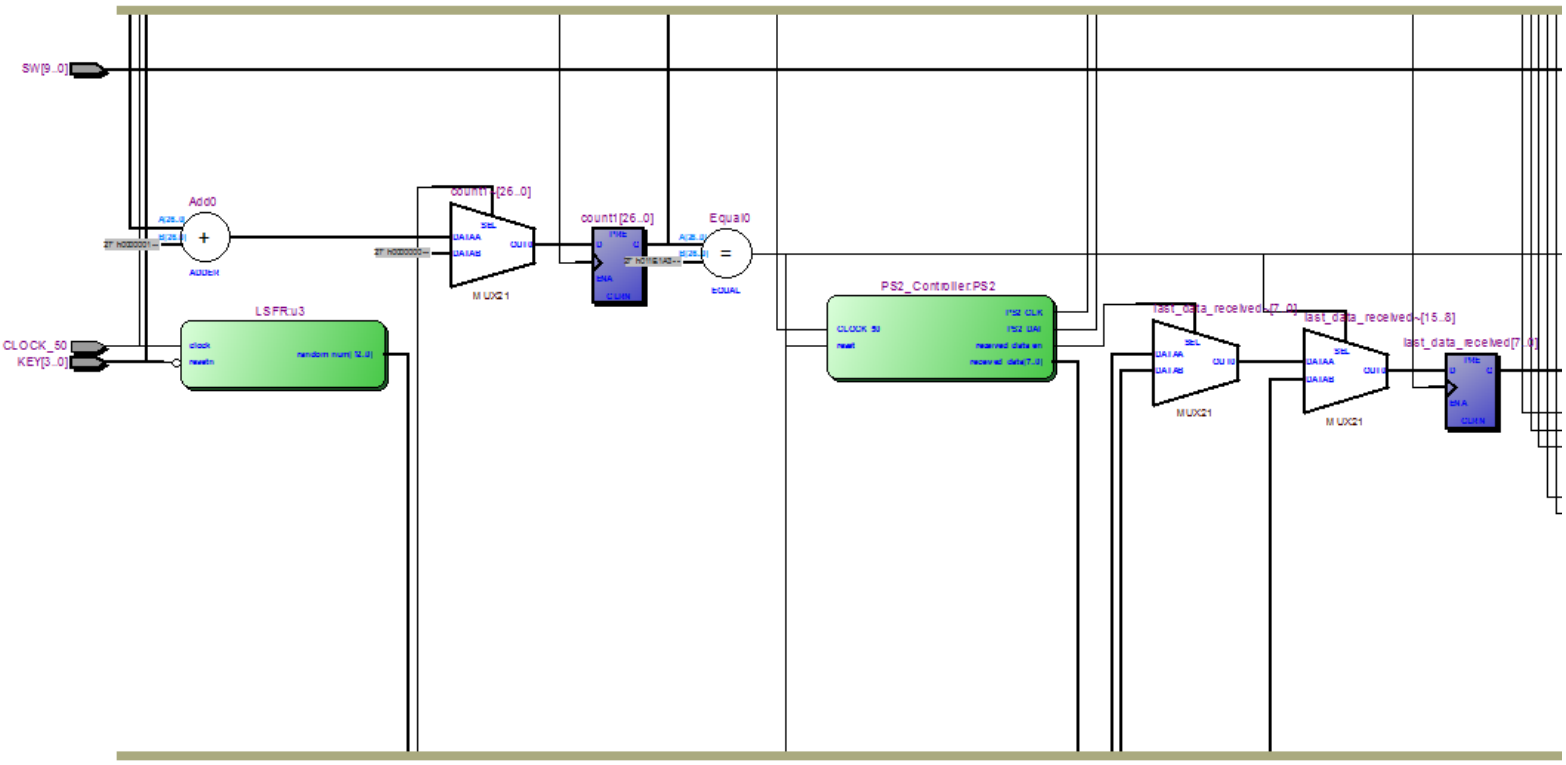
What would you do differently?

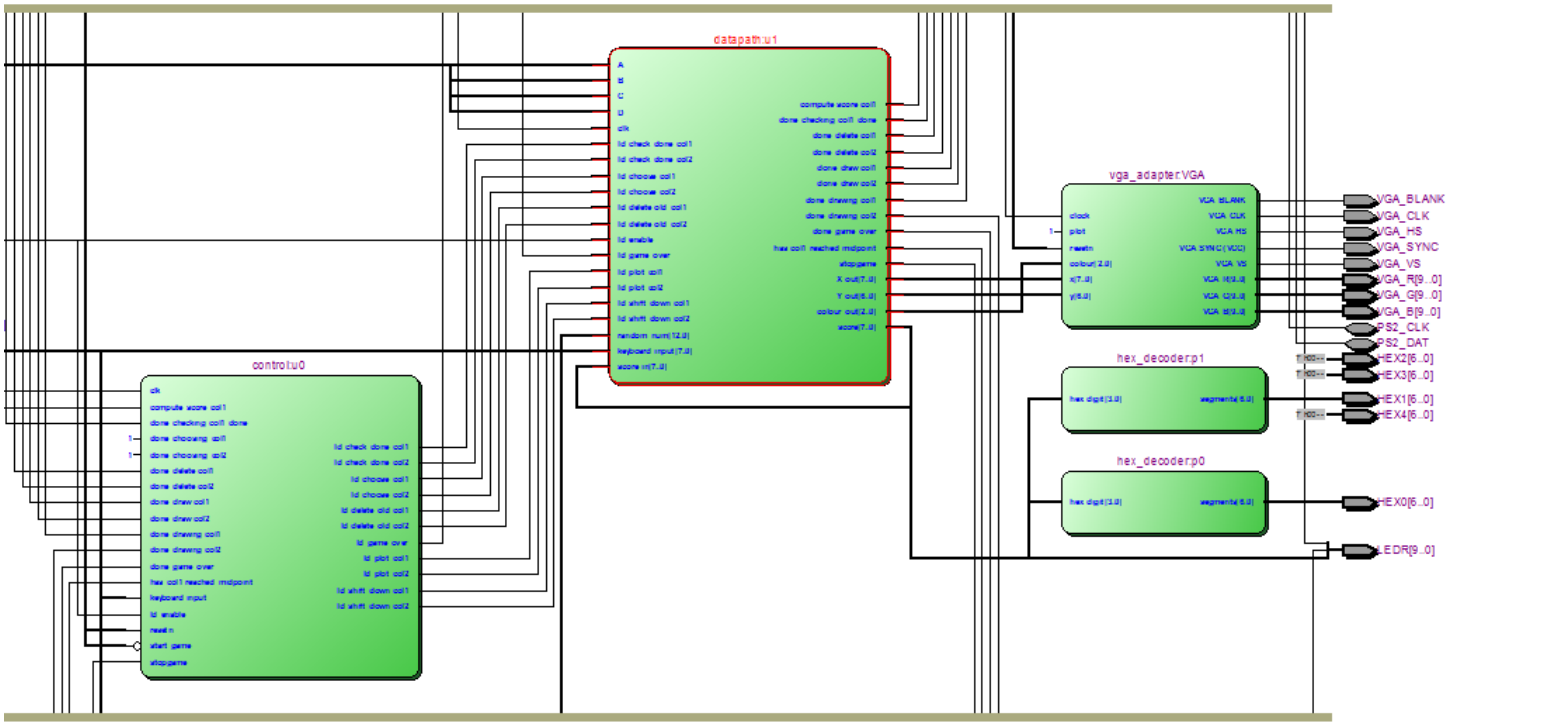
We had a working deliverable for this project. However, there are a few improvements that we can make that will enhance the game. Firstly, having more than one white tile fall down at once will make the game harder. By enhancing the FSM, we could have implemented the game such

that as soon as the white tile in one column reaches the middle, another one is drawn in another column.
Secondly, even division of work could increase productivity greatly. Also, improving communication amongst ourselves would give us a better idea on the progress made in our parts. Next time, we would have a quick scrum meeting every Thursday to see how far have we gotten on our parts and discuss any issues.

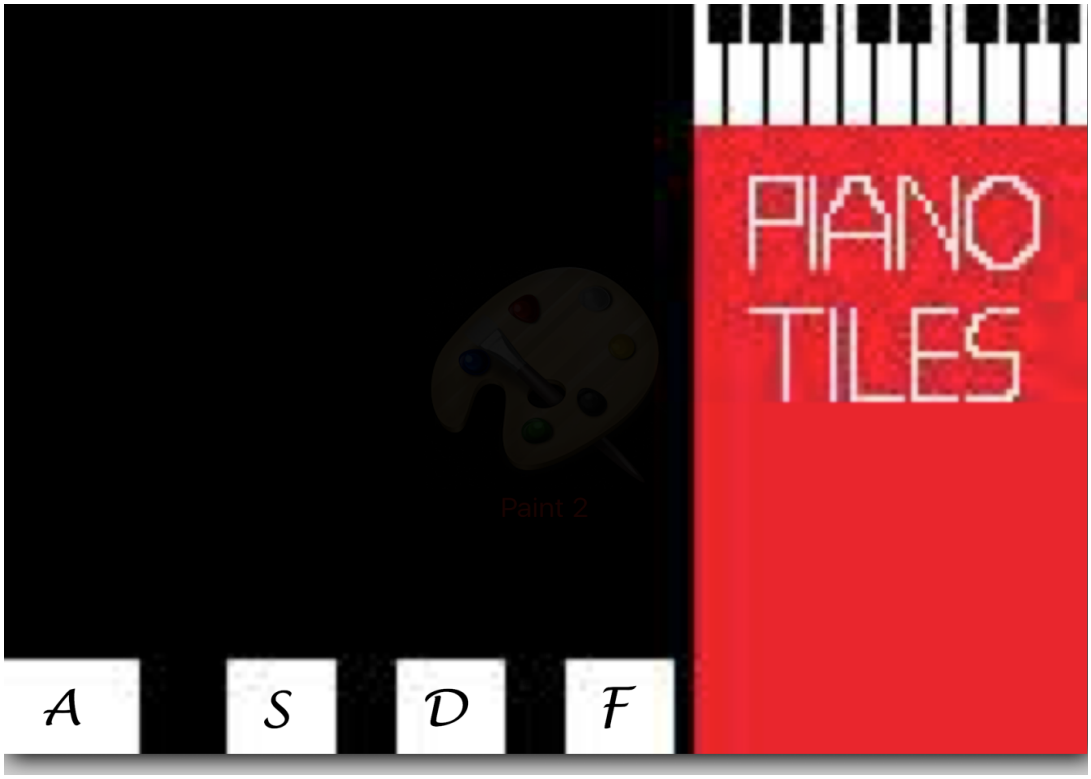
APPENDIX

1. Schematics of project





2. Key assignment



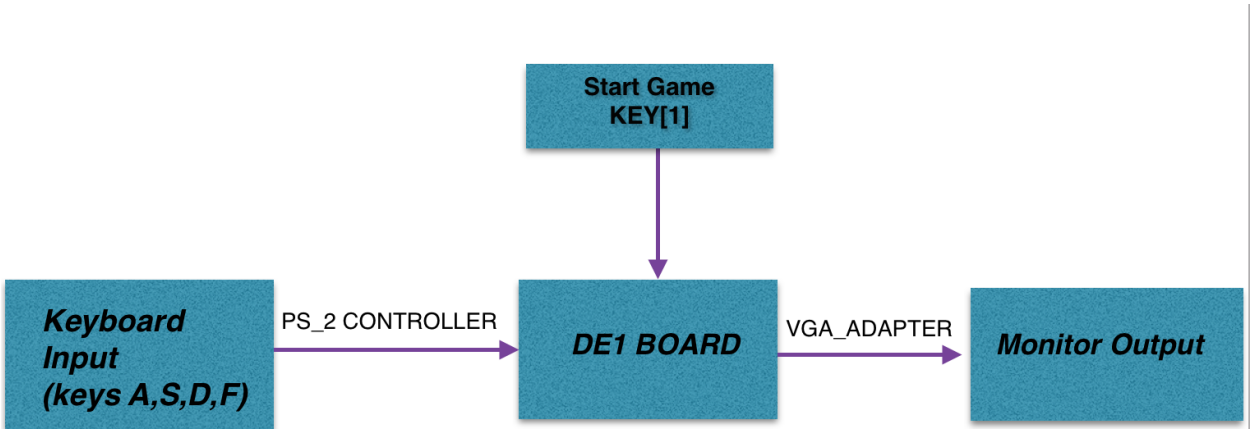
KEYBOARD INPUT

Letters	PS_2 DATA [7:4]	PS_2 DATA [3:0]
A	1	C
S	2	3
D	1	B
F	2	B

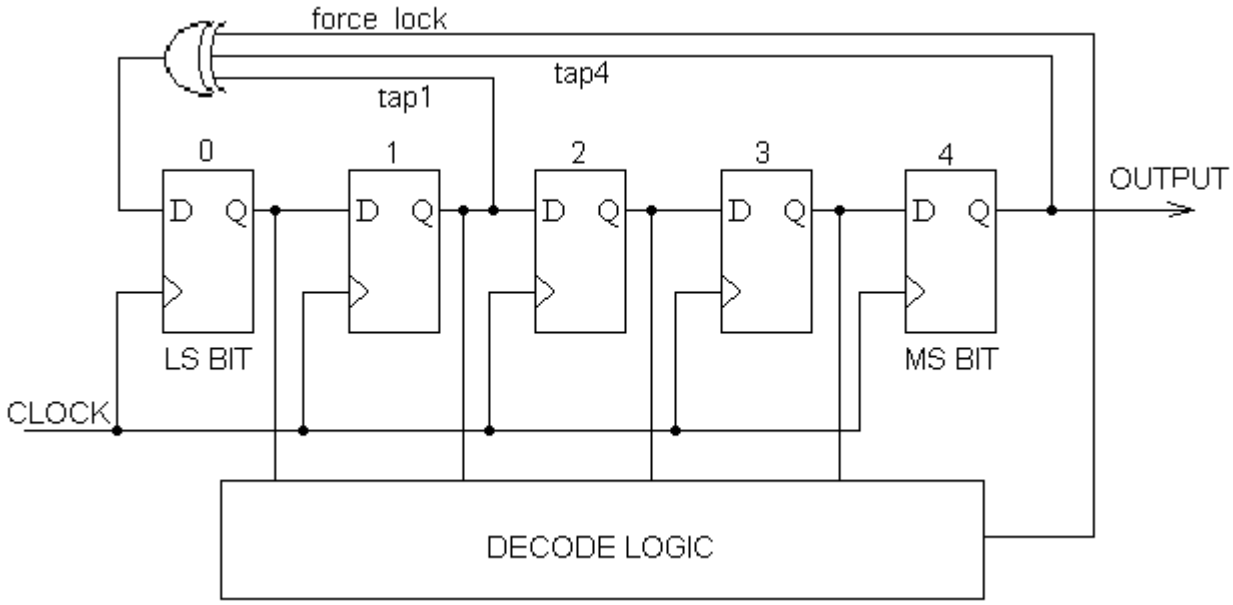
Unique Data sent as output by PS_2 controller, indicating the key pressed on keyboard (the value is in hexadecimal).

3. Game over screen

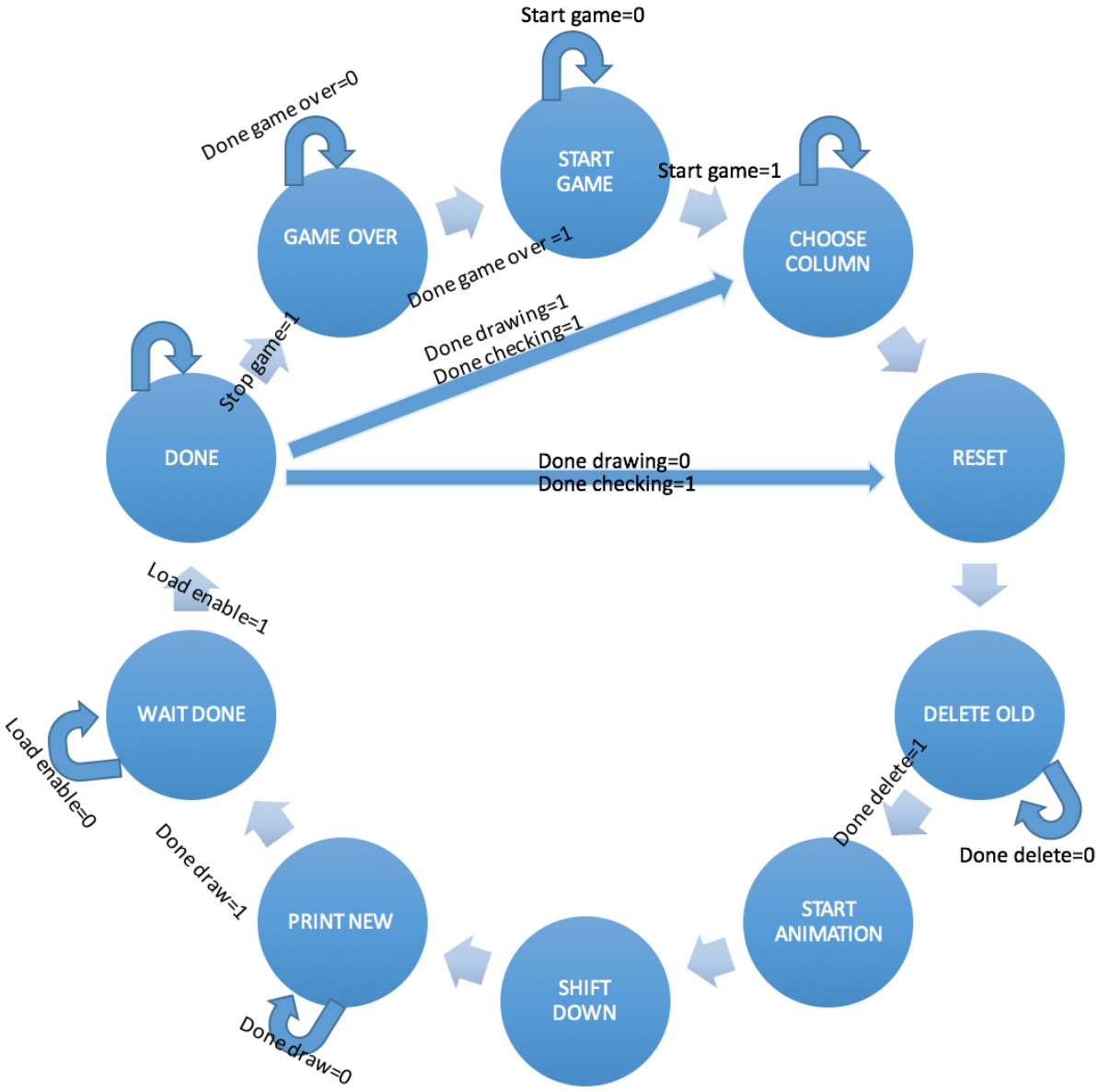
4. Block Diagram for main parts



5. Random generator block diagram



6. FSM block diagram (state diagram)



7. Verilog code

```
module FINAL_PROJECT_ECE241
	(
		CLOCK_50,				//	On Board 50 MHz
		SW,
		KEY,
```

```
                LEDR,
                HEX0,
                HEX1,
                HEX2,
                HEX3,
                HEX4,
                VGA_CLK,                                //      VGA Clock
                VGA_HS,                                 //      VGA H_SYNC
                VGA_VS,                                 //      VGA V_SYNC
                VGA_BLANK,                              //      VGA BLANK
                VGA_SYNC,                               //      VGA SYNC
                VGA_R,                                  //      VGA Red[9:0]
                VGA_G,                                  //      VGA
Green[9:0]
                VGA_B,                                  //      VGA Blue[9:0]
                PS2_CLK,
                PS2_DAT
        );

        // Declare your inputs and outputs here
        input [9:0] SW;
        input [3:0] KEY;
        input CLOCK_50;

  // Bidirectionals
        inout PS2_CLK;
        inout PS2_DAT;
        output [9:0] LEDR;
        output [6:0] HEX0,HEX1,HEX2,HEX3,HEX4;

        // Do not change the following outputs
        output                  VGA_CLK;                //      VGA Clock
        output                  VGA_HS;                 //      VGA H_SYNC
        output                  VGA_VS;                 //      VGA V_SYNC
        output                  VGA_BLANK;              //      VGA BLANK
        output                  VGA_SYNC;               //      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 [2:0] colour_out;
        wire [7:0] x;
        wire [6:0] y;
        wire ld_enable;
```

```
wire start_game;
wire ld_delete_old_col1, ld_delete_old_col2;
wire ld_shift_down_col1, ld_shift_down_col2;
wire ld_plot_col1, ld_plot_col2;
wire ld_choose_col1, ld_choose_col2;
wire ld_check_done_col1, ld_check_done_col2;
wire done_delete_col1, done_delete_col2;
wire done_draw_col1, done_draw_col2;
wire done_drawing_col1, done_drawing_col2;
wire done_choosing_col1, done_choosing_col2;
wire done_game_over;
wire clear1;
wire has_col1_reached_midpoint, done_checking_col1_done;
wire [12:0]random_num;
wire [7:0] score_in;
wire [7:0] score_out;
reg [7:0]x_coord_col;
reg [6:0]y_coord_col;
reg [26:0] count1;
wire compute_score_col1;
wire stopgame;
wire [14:0] addressInput;
wire ld_game_over;



// Internal Wires
wire            [7:0]    ps2_key_data;
wire                              ps2_key_pressed;

// Internal Registers
reg                       [7:0]    last_data_received;

//KEYBOARD INPUT
always @(posedge CLOCK_50)
begin
        if (ld_enable == 1'b1)
                last_data_received <= ps2_key_data;
        else if (ps2_key_pressed == 1'b1)
                last_data_received <= ps2_key_data;
end

//LOAD ENABLE INPUT
always @(posedge CLOCK_50) begin
```

```
    if(clear1 == 1'b1)
            count1 <= 26'd0;
    else
            count1 <= count1 + 1'b1;
  end


  assign clear1 = ld_enable;
  assign ld_enable = (count1 == 26'd1171875) ? 1'b1:1'b0;
  assign LEDR[7:0] = score_out;
  assign LEDR[8] = stopgame;
  assign LEDR[9] = ld_game_over;
  assign score_in = score_out;

  hex_decoder p0(
.hex_digit(score_out[3:0]),
.segments(HEX0)
);
  hex_decoder p1(
            .hex_digit(score_out[7:4]),
            .segments(HEX1)
            );

  LSFR u3 (
          .clock(CLOCK_50),
          .resetn(~KEY[3]),
          .random_num(random_num)
          );

  PS2_Controller PS2 (
  // Inputs
          .CLOCK_50                (CLOCK_50),
          .reset                   (ld_enable),

          // Bidirectionals
          .PS2_CLK                 (PS2_CLK),
          .PS2_DAT                 (PS2_DAT),

          // Outputs
          .received_data           (ps2_key_data),
          .received_data_en    (ps2_key_pressed)
  );
```

```
	control u0(
		// INPUTS
		.clk(CLOCK_50),
		.ld_enable(ld_enable),
		.start_game(~KEY[1]),
		.done_delete_col1(done_delete_col1),
		.done_delete_col2(done_delete_col2),
		.done_draw_col1(done_draw_col1),
		.done_draw_col2(done_draw_col2),
		.done_choosing_col1(1'b1),
		.done_choosing_col2(1'b1),
		.done_drawing_col1(done_drawing_col1),
		.done_drawing_col2(done_drawing_col2),
		.done_game_over(done_game_over),
		.has_col1_reached_midpoint(has_col1_reached_midpoint),
		.done_checking_col1_done(done_checking_col1_done),
		.resetn(KEY[0]),
		.compute_score_col1(compute_score_col1),
		.keyboard_input(last_data_received),
		.stopgame(stopgame),

		// OUTPUTS
		.ld_delete_old_col1(ld_delete_old_col1),
		.ld_delete_old_col2(ld_delete_old_col2),
		.ld_shift_down_col1(ld_shift_down_col1),
		.ld_shift_down_col2(ld_shift_down_col2),
		.ld_plot_col1(ld_plot_col1),
		.ld_plot_col2(ld_plot_col2),
		.ld_choose_col1(ld_choose_col1),
		.ld_choose_col2(ld_choose_col2),
		.ld_check_done_col1(ld_check_done_col1),
		.ld_check_done_col2(ld_check_done_col2),
		.ld_game_over(ld_game_over)
);

	datapath u1(
		//INPUTS
		.clk(CLOCK_50),
		.ld_choose_col1(ld_choose_col1),
		.ld_choose_col2(ld_choose_col2),
		.ld_delete_old_col1(ld_delete_old_col1),
		.ld_delete_old_col2(ld_delete_old_col2),
		.ld_shift_down_col1(ld_shift_down_col1),
		.ld_shift_down_col2(ld_shift_down_col2),
```

```
                .ld_plot_col1(ld_plot_col1),
                .ld_plot_col2(ld_plot_col2),
                .ld_check_done_col1(ld_check_done_col1),
                .ld_check_done_col2(ld_check_done_col2),
                .ld_game_over(ld_game_over),
                .random_num(random_num),
                .A(SW[4]),
                .B(SW[5]),
                .C(SW[6]),
                .D(SW[7]),
                .keyboard_input(last_data_received),
                .score_in(score_in),
                .ld_enable(ld_enable),

                //OUTPUTS
                .done_choosing_col1(done_choosing_col1),
                .done_choosing_col2(done_choosing_col2),
                .done_delete_col1(done_delete_col1),
                .done_delete_col2(done_delete_col2),
                .done_draw_col1(done_draw_col1),
                .done_draw_col2(done_draw_col2),
                .done_drawing_col1(done_drawing_col1),
                .done_drawing_col2(done_drawing_col2),
                .done_game_over(done_game_over),
                .has_col1_reached_midpoint(has_col1_reached_midpoint),
                .done_checking_col1_done(done_checking_col1_done),
                .X_out(x),
                .Y_out(y),
                .colour_out(colour_out),
                .score(score_out),
                .compute_score_col1(compute_score_col1),
                .stopgame(stopgame),
                .addressInput(addressInput)
);


        // Define the number of colours as well as the initial background
        // image file (.MIF) for the controller.
        vga_adapter VGA(
                        .resetn(KEY[0]),
                        .clock(CLOCK_50),
                        .colour(colour_out),
                        .x(x),
                        .y(y),
```

```
                .plot(1'b1),
                /* 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),
                .VGA_SYNC(VGA_SYNC),
                .VGA_CLK(VGA_CLK));

            defparam VGA.RESOLUTION = "160x120";
            defparam VGA.MONOCHROME = "FALSE";
            defparam VGA.BITS_PER_COLOUR_CHANNEL = 1;
            defparam VGA.BACKGROUND_IMAGE = "game_screen_orig.mif";

endmodule

module control(
    input clk,
        input ld_enable,
        input start_game,
    input done_delete_col1, done_delete_col2,
    input done_draw_col1, done_draw_col2,
        input done_choosing_col1, done_choosing_col2,
        input done_drawing_col1, done_drawing_col2,
        input done_game_over,
        input has_col1_reached_midpoint,
        input done_checking_col1_done,
        input resetn,
        input compute_score_col1,
        input keyboard_input,
        input stopgame,

    output reg  ld_delete_old_col1, ld_delete_old_col2,
        output reg ld_shift_down_col1, ld_shift_down_col2,
        output reg ld_plot_col1, ld_plot_col2,
        output reg ld_choose_col1, ld_choose_col2,
        output reg ld_check_done_col1, ld_check_done_col2,
        output reg ld_game_over
    );

    reg [5:0] current_state, next_state;
```

```
   localparam  START_GAME                                          = 5'd0,
                                CHOOSE_COLUMN_1                             = 5'd1,
                                CHOOSE_COLUMN_2                             = 5'd2,
                                RESET_1                                     = 5'd3,
           DELETE_OLD_1                                 = 5'd4,
                                RESET_2                                     = 5'd5,
           DELETE_OLD_2                                 = 5'd6,
           START_ANIMATION_1                    = 5'd7,
                                SHIFT_DOWN_1                                = 5'd8,
                                PRINT_NEW_1                                 = 5'd9,
           START_ANIMATION_2                    = 5'd10,
                                SHIFT_DOWN_2                                =
5'd11,
                                PRINT_NEW_2                                 =
5'd12,
                                DONE_1
           = 5'd13,
                                DONE_2
           = 5'd14,
                                WAIT_DONE_1                                 =
5'd15,
                                GAME_OVER
           = 5'd16;

   // Next state logic aka our state table
   always@(*)
   begin: state_table
          case (current_state)
                                START_GAME: next_state = start_game ?
CHOOSE_COLUMN_1 : START_GAME;
                                CHOOSE_COLUMN_1: next_state = RESET_1;
                                RESET_1: next_state = DELETE_OLD_1; // Loop in
current state until value is input
              DELETE_OLD_1: next_state = done_delete_col1 ? START_ANIMATION_1 :
DELETE_OLD_1; // Loop in current state until go signal goes low
                                START_ANIMATION_1: next_state = SHIFT_DOWN_1; //
Loop in current state until value is input
                                SHIFT_DOWN_1: next_state = PRINT_NEW_1;
                                PRINT_NEW_1: next_state = done_draw_col1 ?
WAIT_DONE_1 : PRINT_NEW_1;
                                WAIT_DONE_1: next_state = ld_enable ? DONE_1 :
WAIT_DONE_1;
                                DONE_1:
```

```
                                        if (done_checking_col1_done == 1'b1 &&
done_drawing_col1 == 1'b1)
                                                next_state = CHOOSE_COLUMN_1;
                                        else if (done_checking_col1_done == 1'b1 &&
done_drawing_col1 == 1'b0)
                                                next_state = RESET_1;
                                        else if (stopgame == 1'b1)
                                                next_state = GAME_OVER;
                                        else
                                                next_state = DONE_1;
                                GAME_OVER: next_state = done_game_over ?
START_GAME : GAME_OVER;//(start_game && done_game_over) ? START_GAME :
GAME_OVER;
            default:  next_state = START_GAME;
        endcase
    end // state_table


    // Output logic aka all of our datapath control signals
    always @(*)
    begin: enable_signals
        // By default make all our signals 0
                ld_plot_col1 = 1'b0;
                ld_plot_col2 = 1'b0;
                ld_delete_old_col1 = 1'b0;
                ld_delete_old_col2 = 1'b0;
                ld_shift_down_col1 = 1'b0;
                ld_shift_down_col2 = 1'b0;
                ld_choose_col1 = 1'b0;
                ld_choose_col2 = 1'b0;
                ld_check_done_col1 = 1'b0;
                ld_check_done_col2 = 1'b0;
                ld_game_over = 1'b0;

        case (current_state)
                        CHOOSE_COLUMN_1: begin
                                ld_choose_col1 = 1'b1;
                        end
                        CHOOSE_COLUMN_2: begin
                                ld_choose_col2 = 1'b1;
                        end
                        DELETE_OLD_1: begin
                                ld_delete_old_col1 = 1'b1;
                        end
```

```
                                    DELETE_OLD_2: begin
                                            ld_delete_old_col2 = 1'b1;
                                    end
                                    SHIFT_DOWN_1: begin
                                            ld_shift_down_col1 = 1'b1;
                                    end
                                    PRINT_NEW_1: begin
                                            ld_plot_col1 = 1'b1;
                                    end
                                    SHIFT_DOWN_2: begin
                                            ld_shift_down_col2 = 1'b1;
                                    end
                                    PRINT_NEW_2: begin
                                            ld_plot_col2 = 1'b1;
                                    end
                                    DONE_1: begin
                                            ld_check_done_col1 = 1'b1;
                                    end
                                    DONE_2: begin
                                            ld_check_done_col1 = 1'b1;
                                            ld_check_done_col2 = 1'b1;
                                    end
                                    GAME_OVER: begin
                                            ld_game_over = 1'b1;
                                    end
        endcase
    end // enable_signals

    // current_state registers
    always@(posedge clk)
    begin: state_FFs
        if(!resetn)
            current_state <= START_GAME;
        else
            current_state <= next_state;
    end // state_FFS
endmodule

module datapath(
    input clk,
            input ld_choose_col1, ld_choose_col2,
    input ld_delete_old_col1, ld_delete_old_col2,
    input ld_shift_down_col1, ld_shift_down_col2,
            input ld_plot_col1, ld_plot_col2,
```

```
        input ld_check_done_col1, ld_check_done_col2,
        input ld_game_over,
        input [12:0]random_num,
        input A, B, C, D,
        input [7:0] keyboard_input,
        input [7:0]score_in,
        input ld_enable,
        output reg done_choosing_col1, done_choosing_col2,
        output reg done_delete_col1, done_delete_col2,
        output reg done_draw_col1, done_draw_col2,
        output reg done_drawing_col1, done_drawing_col2,
        output reg done_game_over,
        output reg has_col1_reached_midpoint, done_checking_col1_done,
output reg [7:0] X_out,
        output reg [6:0] Y_out,
        output reg [2:0] colour_out,
        output [7:0] score,
        output reg compute_score_col1,
        output reg stopgame,
        output reg addressInput
);

// input registers
        reg [7:0] x_Orig_col1;
        reg [6:0] y_Orig_col1;
        reg [7:0] x_Orig_col2;
        reg [6:0] y_Orig_col2;
        reg [7:0] x_Modified_col1;
        reg [6:0] y_Modified_col1;
        reg [7:0] x_Modified_col2;
        reg [6:0] y_Modified_col2;
        reg [9:0] counter = 8'b00000000;
        reg [9:0] counter_black = 8'b00000000;
        reg random_coord = 1'b1;
        reg [7:0]score_temp = 8'b0;
        reg [14:0]counter_game_over = 15'b0;
        reg [7:0]x_counter_game_over = 8'b00000100;
        reg [7:0]y_counter_game_over = 7'b0;

        wire [2:0] colour_out_game_over;

        assign score = score_temp;

        GAME_OVER gameover (
```

```
		.address(counter_game_over), //15 bit number
		.clock(clk),
		.data(1'b0), // 3bits
		.wren(1'b0),
		.q(colour_out_game_over)
		);

	always@(posedge clk) begin
		compute_score_col1 <= 1'b0;
		stopgame <= 1'b0;
		if (x_Modified_col1 == 8'b00000100 && y_Modified_col1 >= 7'b1011010 && A ==
1'b1 && ld_enable == 1'b1) begin // First column keyboard_input == 8'b00011100
			score_temp <= score_temp + 8'b1;
			compute_score_col1 <= 1'b1;
		end
		else if (x_Modified_col1 == 8'b00100001 && y_Modified_col1 >= 7'b1011010 &&
B == 1'b1 && ld_enable == 1'b1) begin // Second column keyboard_input == 8'b00011011
			score_temp <= score_temp + 8'b1;
			compute_score_col1 <= 1'b1;
		end
		else if (x_Modified_col1 == 8'b00111010 && y_Modified_col1 >= 7'b1011010 &&
C == 1'b1 && ld_enable == 1'b1) begin // Third column keyboard_input == 8'b00100011
			score_temp <= score_temp + 8'b1;
			compute_score_col1 <= 1'b1;
		end
		else if (x_Modified_col1 == 8'b01010011 && y_Modified_col1 >= 7'b1011010 &&
D == 1'b1 && ld_enable == 1'b1) begin // Fourth column keyboard_input == 8'b00101011
			score_temp <= score_temp + 8'b1;
			compute_score_col1 <= 1'b1;
		end
		else if ((y_Modified_col1 < 7'b1011010 || y_Modified_col1 < 7'b1011010 ||
y_Modified_col1 < 7'b1011010 || y_Modified_col1 < 7'b1011010) && ld_enable == 1'b1 && (A
== 1'b1 || B == 1'b1 || C == 1'b1 || D == 1'b1)) //(keyboard_input == 8'b00011100 ||
keyboard_input == 8'b00011011 || keyboard_input == 8'b00100011 || keyboard_input ==
8'b00101011)
			stopgame <= 1'b1;
		if (score_temp == 8'b11111111) begin
			score_temp <= 8'b00000000;
			compute_score_col1 <= 1'b0;
		end
	end

	always@(posedge clk) begin
		done_drawing_col1 <= 1'b0;
```

```
		done_drawing_col2 <= 1'b0;
		done_checking_col1_done <= 1'b0;
		has_col1_reached_midpoint <= 1'b0;
		//done_game_over <= 1'b0;

		//CHANGING BACKGROUND
		if (ld_game_over == 1'b1)begin
			if (x_counter_game_over < 8'b10100000) begin //160
				x_counter_game_over <= x_counter_game_over + 8'b1;
				counter_game_over <= counter_game_over + 15'b1;
				done_game_over <= 1'b0;
			end
			else if (x_counter_game_over == 8'b10100000) begin //160
				x_counter_game_over <= 8'b0;
				y_counter_game_over <= y_counter_game_over + 7'b1;
				done_game_over <= 1'b0;
				if (y_counter_game_over == 7'b1111000) begin
					done_game_over <= 1'b1;
					counter_game_over <= 15'b0;
				end
			end
			X_out <= x_counter_game_over;
			Y_out <= y_counter_game_over;
			colour_out <= colour_out_game_over;
		end

		else if (ld_check_done_col1) begin
			if (y_Modified_col1 == 7'b1101001) begin
				done_drawing_col1 <= 1'b1;
				done_checking_col1_done <= 1'b1;
			end
			else if (y_Modified_col1 < 7'b1101001)
				done_checking_col1_done <= 1'b1;
			else if (y_Modified_col1 > 7'b1101001)
				y_Modified_col1 <= 7'b1101001;
		end
		if (ld_check_done_col2) begin
			if (y_Modified_col2 >= 7'b1101001)
				done_drawing_col2 <= 1'b1;
		end

	if(ld_choose_col1) begin
			if (random_num == 13'b0000000000000 && x_Orig_col2 != 8'b00000100
&& x_Orig_col1 != 8'b00000100) begin
```

```
                        x_Orig_col1 <= 8'b00000100; // first column
                        y_Orig_col1 <= 7'b0000000;
                        x_Modified_col1 <= 8'b00000100;
                        y_Modified_col1 <= 7'b0000000;
                    end
                    else if (random_num == 13'b0000000000001 && x_Orig_col2 !=
8'b00100001 && x_Orig_col1 != 8'b00100001) begin
                        x_Orig_col1 <= 8'b00100001; //second column
                        y_Orig_col1 <= 7'b0000000;
                        x_Modified_col1 <= 8'b00100001;
                        y_Modified_col1 <= 7'b0000000;
                    end
                    else if (random_num == 13'b0000000000010 && x_Orig_col2 !=
8'b00111010 && x_Orig_col1 != 8'b00111010) begin
                        x_Orig_col1 <= 8'b00111010; // third column
                        y_Orig_col1 <= 7'b0000000;
                        x_Modified_col1 <= 8'b00111010;
                        y_Modified_col1 <= 7'b0000000;
                    end
                    else if(random_num == 13'b0000000000011 && x_Orig_col2 !=
8'b01010011 && x_Orig_col1 != 8'b01010011) begin
                        x_Orig_col1 <= 8'b01010011; // fourth column
                        y_Orig_col1 <= 7'b0000000;
                        x_Modified_col1 <= 8'b01010011;
                        y_Modified_col1 <= 7'b0000000;
                    end
                end

                else if(ld_choose_col2) begin
                    if ((random_num == 13'b0000000000000 || random_num ==
13'b0000000000001) && x_Orig_col1 != 8'b00000100) begin
                        x_Orig_col2 <= 8'b00000100; // first column
                        y_Orig_col2 <= 7'b0000000;
                        x_Modified_col2 <= 8'b00000100;
                        y_Modified_col2 <= 7'b0000000;
                    end
                    else if ((random_num == 13'b0000000000010 || random_num ==
13'b0000000000011) && x_Orig_col1 != 8'b00111010) begin
                        x_Orig_col2 <= 8'b00111010; //third column
                        y_Orig_col2 <= 7'b0000000;
                        x_Modified_col2 <= 8'b00111010;
                        y_Modified_col2 <= 7'b0000000;
                    end
```

```
                        else if ((random_num == 13'b0000000000100 || random_num ==
13'b0000000000101) && x_Orig_col1 != 8'b01010011) begin
                                x_Orig_col2 <= 8'b01010011; // fourth column
                                y_Orig_col2 <= 7'b0000000;
                                x_Modified_col2 <= 8'b01010011;
                                y_Modified_col2 <= 7'b0000000;
                        end
                        else if((random_num == 13'b0000000000110 || random_num ==
13'b0000000000111) && x_Orig_col1 != 8'b00100001) begin
                                x_Orig_col2 <= 8'b00100001; // second column
                                y_Orig_col2 <= 7'b0000000;
                                x_Modified_col2 <= 8'b00100001;
                                y_Modified_col2 <= 7'b0000000;
                        end
                end

                if(ld_shift_down_col1) begin
                        y_Modified_col1 <= y_Modified_col1 + 7'b0000001; //increment
coordinate to move one down
                        y_Orig_col1 <= y_Modified_col1;

                end

                else if(ld_shift_down_col2) begin
                        y_Modified_col2 <= y_Modified_col2 + 7'b0000001; //increment
coordinate to move one down
                        y_Orig_col2 <= y_Modified_col2;
                end

                else if(ld_delete_old_col1) begin
                        if (counter_black <= 8'b11111111)begin
                                X_out <= x_Orig_col1 + counter_black[3:0];
                                Y_out <= y_Orig_col1 + counter_black[8:4];
                                colour_out <= 3'b000;
                                counter_black <= counter_black + 8'b00000001;
                                done_delete_col1 = 1'b0;
                        end
                        else begin
                                done_delete_col1 = 1'b1;
                                counter_black <= 8'b00000000;
                        end
                end

                else if(ld_delete_old_col2) begin
```

```
                        if (counter_black <= 8'b11111111)begin
                                X_out <= x_Orig_col2 + counter_black[3:0];
                                Y_out <= y_Orig_col2 + counter_black[8:4];
                                colour_out <= 3'b000;
                                counter_black <= counter_black + 8'b00000001;
                                done_delete_col2 = 1'b0;
                        end
                        else begin
                                done_delete_col2 = 1'b1;
                                counter_black <= 8'b00000000;
                        end
                end
                else begin
                        if (ld_plot_col1) begin
                                if (counter <= 8'b11111111)begin
                                        X_out <= x_Orig_col1 + counter[3:0];
                                        Y_out <= y_Orig_col1 + counter[8:4];
                                        colour_out <= 3'b111;
                                        counter <= counter + 8'b00000001;
                                        done_draw_col1 <= 1'b0;
                                end
                                else begin
                                        done_draw_col1 <= 1'b1;
                                        counter <= 8'b00000000;
                                end
                        end

                        else if (ld_plot_col2) begin
                                if (counter <= 8'b11111111)begin
                                        X_out <= x_Orig_col2 + counter[3:0];
                                        Y_out <= y_Orig_col2 + counter[8:4];
                                        colour_out <= 3'b111;
                                        counter <= counter + 8'b00000001;
                                        done_draw_col2 <= 1'b0;
                                end
                                else begin
                                        done_draw_col2 <= 1'b1;
                                        counter <= 8'b00000000;
                                end
                        end
                end
        end
endmodule
```

```
module LSFR(clock, resetn, random_num);
	input clock;
	input resetn;
	output [12:0] random_num;

	wire [12:0] rnd;
	wire clear1;
	wire ld_enable;
	reg [26:0] count1;
	wire [12:0]final_num;

	always @(posedge clock) begin
	  if(clear1 == 1'b1)
		count1 <= 26'd0;
	  else
		count1 <= count1 + 1'b1;
	end

	assign clear1 = ld_enable;
	assign ld_enable = (count1 == 26'd3125000) ? 1'b1:1'b0;
	assign random_num = rnd & 13'b0000000000111;


	RandomNum u1(
			.clock(ld_enable),
			.reset(resetn),
			.rnd(rnd)
			);

endmodule


module RandomNum(input clock, input reset, output [12:0] rnd);

	wire feedback = random[12] ^ random[3] ^ random[2] ^ random[0];

	reg [12:0] random, random_next, random_done;
	reg [3:0] count, count_next;

	always@ (posedge clock, posedge reset)
	begin
	 if (reset)
	 begin
	  random <= 13'hF;
```

```
            count <= 0;
           end

           else
           begin
            random <= random_next;
            count <= count_next;
           end
          end

          always@(*)
          begin
           random_next = random;
           count_next = count;

            random_next = {random[11:0], feedback};
            count_next = count + 1;

           if (count == 13)
           begin
            count_next = 0;
            random_done = random;
           end

          end
          assign rnd = random_done;

endmodule

module ram32x4 (
          address,
          clock,
          data,
          wren,
          q);

          input    [14:0]  address;
          input      clock;
          input    [2:0]  data;
          input      wren;
          output  [2:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
```

```
	tri1	clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

	wire [2:0] sub_wire0;
	wire [2:0] q = sub_wire0[2:0];

	altsyncram	altsyncram_component (
				.address_a (address),
				.clock0 (clock),
				.data_a (data),
				.wren_a (wren),
				.q_a (sub_wire0),
				.aclr0 (1'b0),
				.aclr1 (1'b0),
				.address_b (1'b1),
				.addressstall_a (1'b0),
				.addressstall_b (1'b0),
				.byteena_a (1'b1),
				.byteena_b (1'b1),
				.clock1 (1'b1),
				.clocken0 (1'b1),
				.clocken1 (1'b1),
				.clocken2 (1'b1),
				.clocken3 (1'b1),
				.data_b (1'b1),
				.eccstatus (),
				.q_b (),
				.rden_a (1'b1),
				.rden_b (1'b1),
				.wren_b (1'b0));
	defparam
		altsyncram_component.clock_enable_input_a = "BYPASS",
		altsyncram_component.clock_enable_output_a = "BYPASS",
		altsyncram_component.init_file = "../image.colour.game.over.mif",
		altsyncram_component.intended_device_family = "Cyclone V",
		altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
		altsyncram_component.lpm_type = "altsyncram",
		altsyncram_component.numwords_a = 32768,
		altsyncram_component.operation_mode = "SINGLE_PORT",
		altsyncram_component.outdata_aclr_a = "NONE",
		altsyncram_component.outdata_reg_a = "UNREGISTERED",
		altsyncram_component.power_up_uninitialized = "FALSE",
```

```
                altsyncram_component.read_during_write_mode_port_a =
"NEW_DATA_NO_NBE_READ",
                altsyncram_component.widthad_a = 15,
                altsyncram_component.width_a = 3,
                altsyncram_component.width_byteena_a = 1;


endmodule


module hex_decoder(hex_digit, segments);
    input [3:0] hex_digit;
    output reg [6:0] segments;

    always @(*)
        case (hex_digit)
            4'h0: segments = 7'b100_0000;
            4'h1: segments = 7'b111_1001;
            4'h2: segments = 7'b010_0100;
            4'h3: segments = 7'b011_0000;
            4'h4: segments = 7'b001_1001;
            4'h5: segments = 7'b001_0010;
            4'h6: segments = 7'b000_0010;
            4'h7: segments = 7'b111_1000;
            4'h8: segments = 7'b000_0000;
            4'h9: segments = 7'b001_1000;
            4'hA: segments = 7'b000_1000;
            4'hB: segments = 7'b000_0011;
            4'hC: segments = 7'b100_0110;
            4'hD: segments = 7'b010_0001;
            4'hE: segments = 7'b000_0110;
            4'hF: segments = 7'b000_1110;
            default: segments = 7'h7f;
        endcase
endmodule
```