ECE241 Final Report



|  |  |
| --- | --- |
| Project Title | Piano Tiles |
| Tutorial section and station number | PRA05 #59 |
| 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])

1. **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.

1. **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.

1. **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.

1. **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.

1. **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.

1. **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.

1. **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:

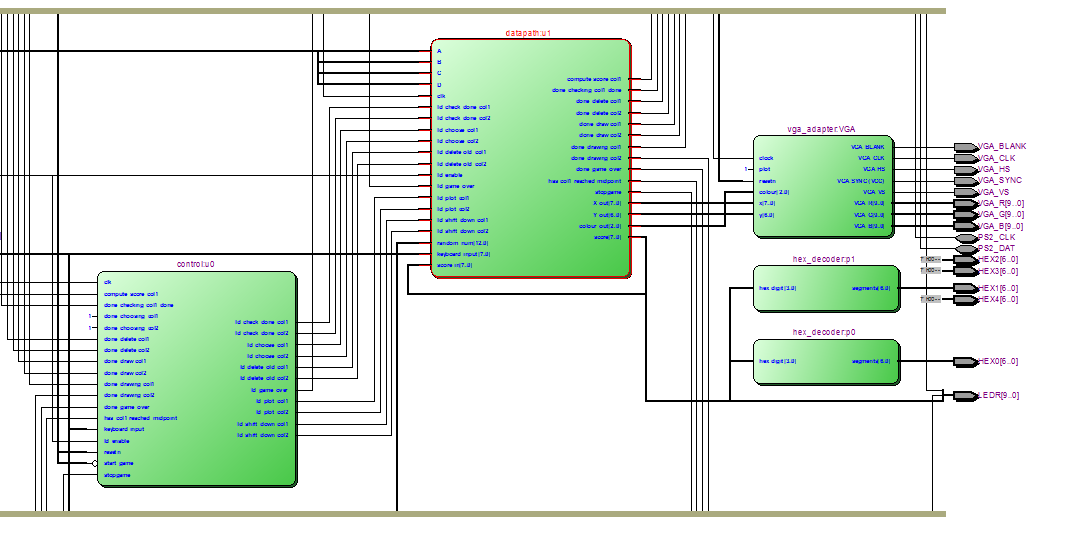
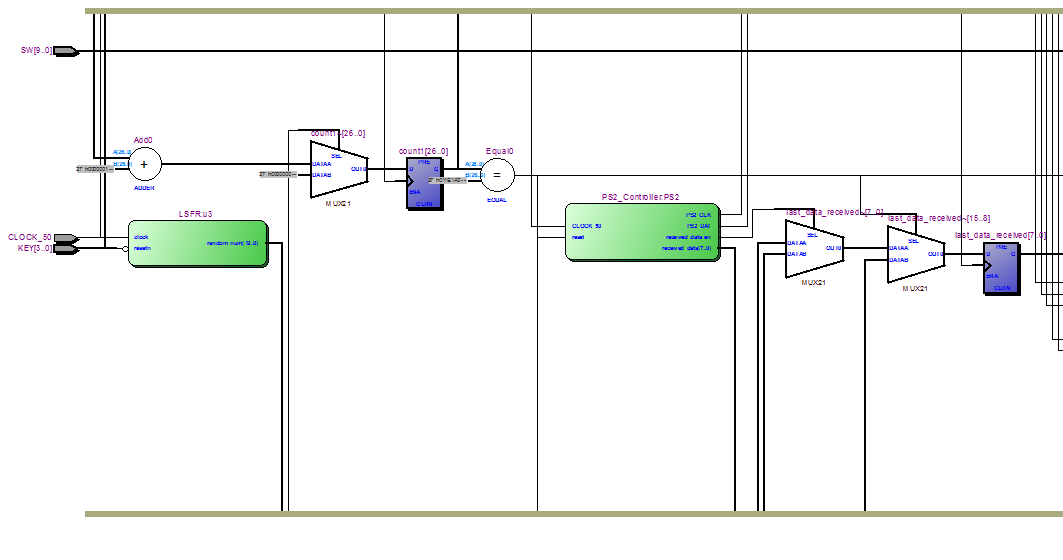
|  |  |
| --- | --- |
| Functionality | Achieved? |
| The game starts when KEY[1] is pressed. | gi.php.png |
| The white tiles appear randomly from top of the screen. | gi.php.png |
| The project reads keyboard input A,S,D,F for 4 different columns. | gi.php.png |
| When the white tiles reaches the bottom and the corresponding key is pressed at the same time, score is computed and is displayed on hex increment. | gi.php.png |
| If the black tiles are hit, this game ends properly and displays the game over screen | gi.php.png |
| 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?**

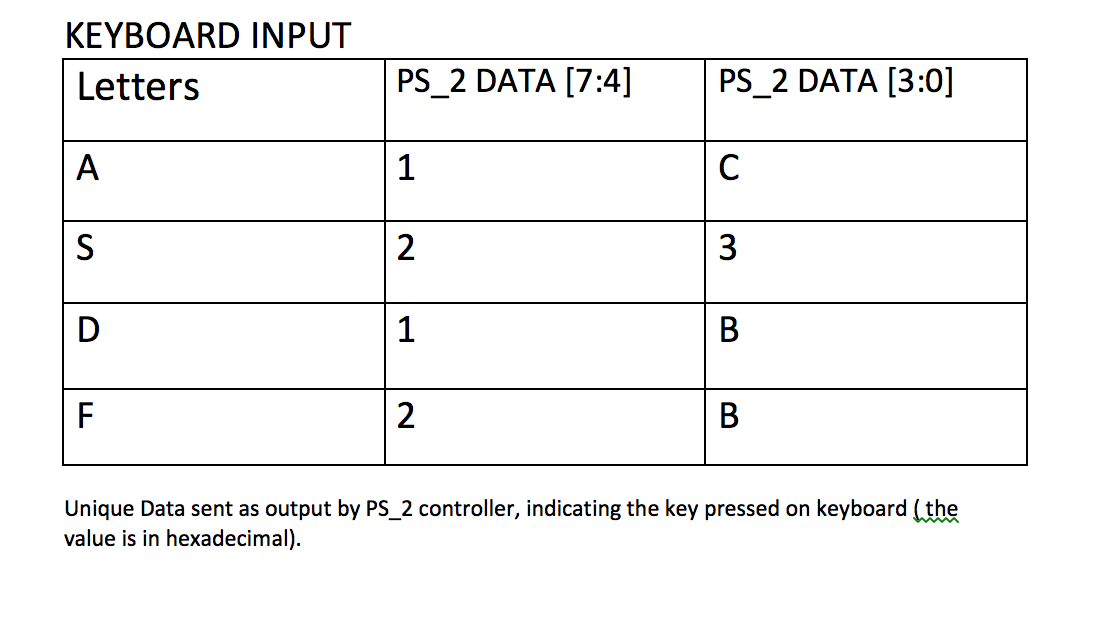
We had a working deliverable for this project. However, there are a few improvements that we can make that will enhance the game. Firstly, having more than one white tile fall down at once will make the game harder. By enhancing the FSM, we could have implemented the game such that as soon as the white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

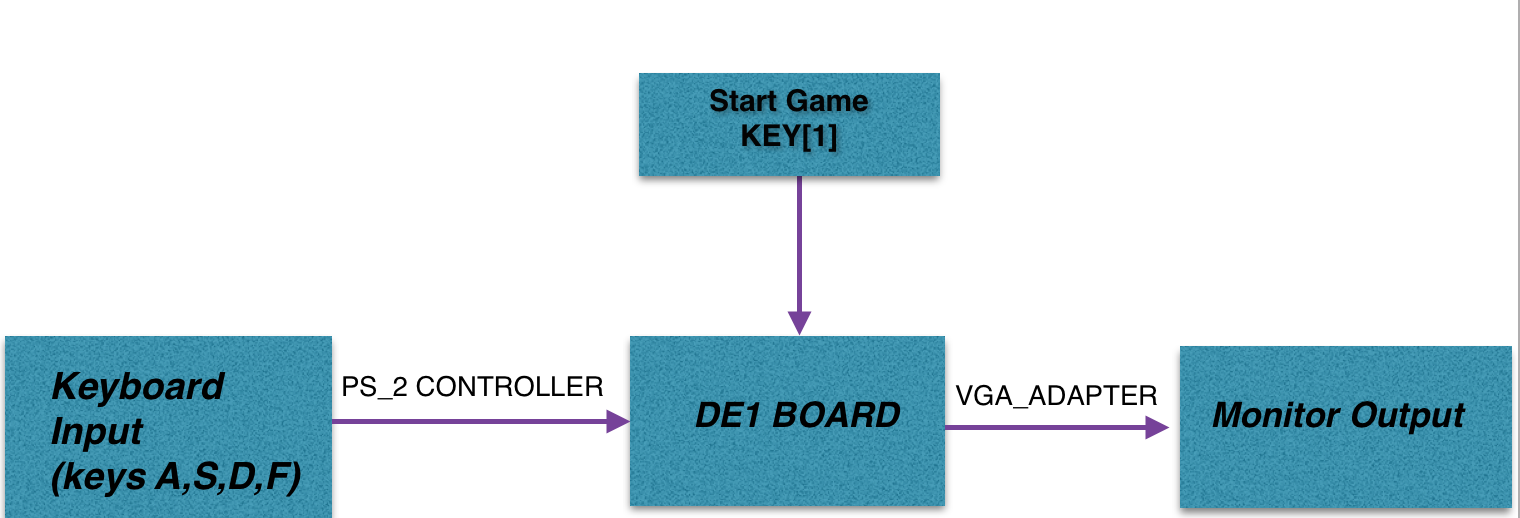




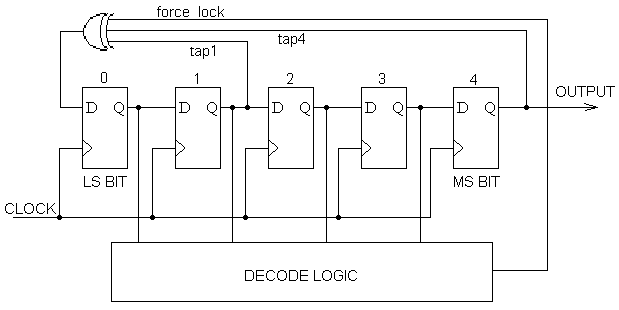
1. Game over screen



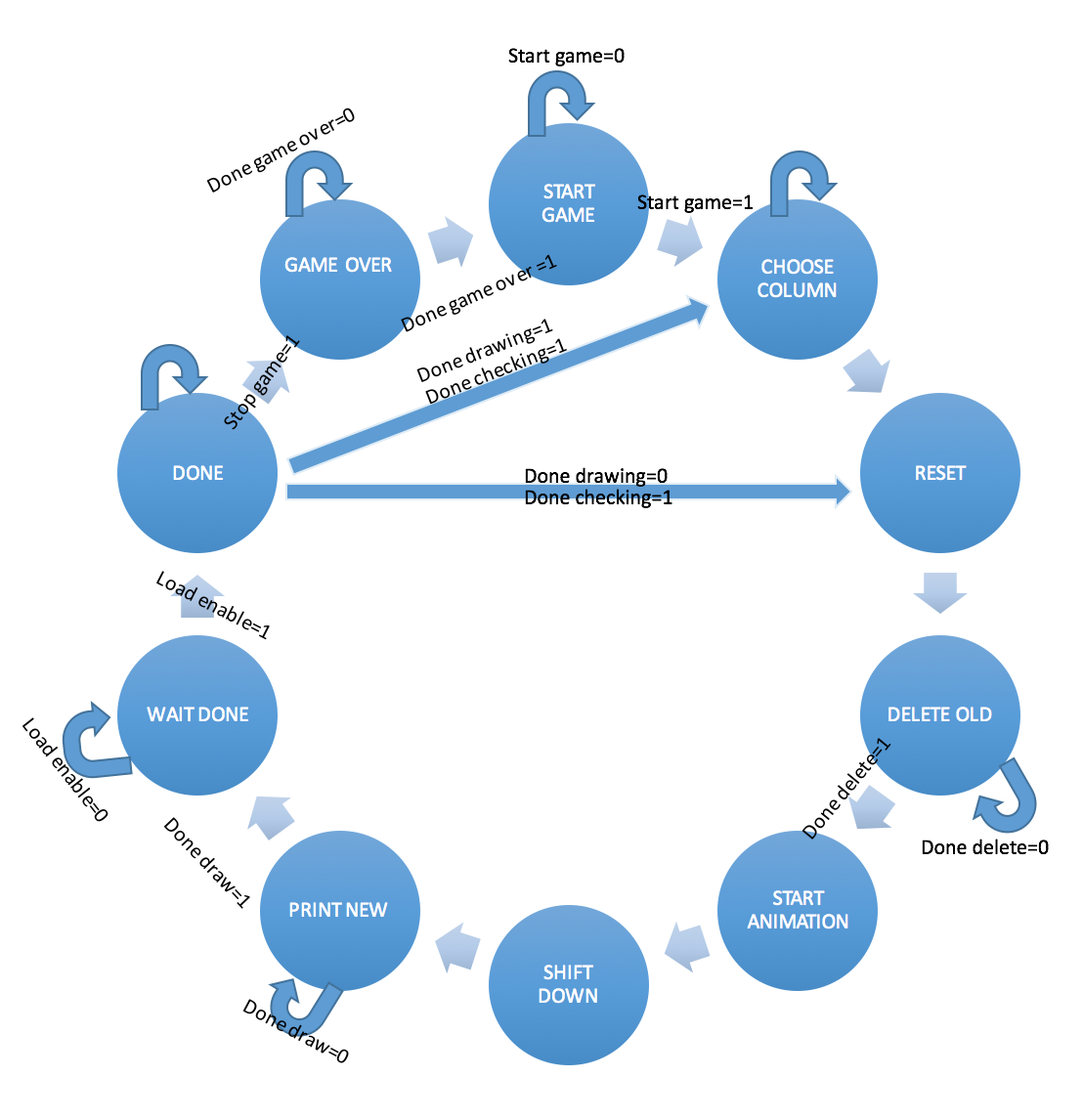
1. Block Diagram for main parts



1. Random generator block diagram



1. FSM block diagram (state diagram)



1. 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