Name: John Espedido, Jiawei Hu **Date:** December 2, 2019

Station Number: 33

ECE241 – FINAL REPORT

SLOT MACHINE

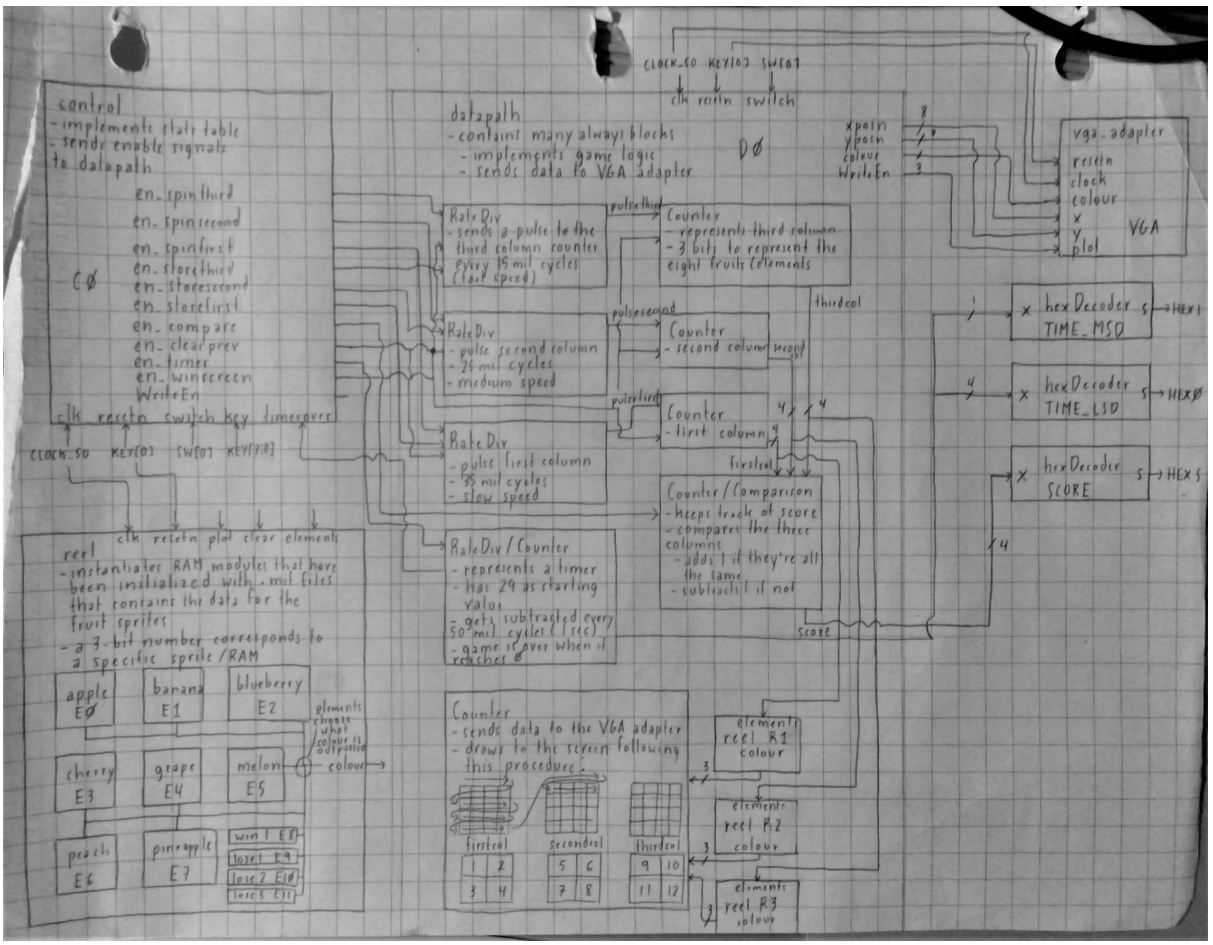
Introduction

The quality and success of a project is often measured by its ability to meet expectations, specifications and needs. When it comes to deciding a project idea, many people are quick to believe that quality implies difficulty – a good project depends on complexity. As the final project of ECE241, my partner and I sought to make our project idea unique, fancy while maintaining a moderate level of difficulty that is within our scope of ability developed through the course.

Our first objective in creating our idea was to incorporate as much of what we learned throughout the course into our project as possible to demonstrate our firm understanding of class materials. Next was to make the idea interesting and eye-catching. We started by viewing some of the past projects through the links on piazza to gather inspiration and to witness the elements that make a project captivating. We took note of the key visuals, colors and animations of some remarkable creations. Eventually we agreed upon a game that everyone is familiar with: slot machine game. It is a simple yet addicting game that has brought people around the globe to play in casinos. It is a game that would embody everything we learned in ECE241 from adders to FSM as well as the potential to have stunning visuals to be produced on the VGA display – a type of fanciness unique to slot machines. The idea stood out to us and we were genuinely excited to start the project. This became our motivation to make this project as good as we imagined.

The Design

The following block diagram is shown in more detail as necessary. Some notes may overlap with the following paragraphs.



Control:

The control module implements the state logic of the slot machine and sends the corresponding enable signals. The design has five states in total. Win/loss screen shows a corresponding image whether the user has successfully matched three of the same elements or not (en_winscreen). Clear screen sets all of the columns back to the first element and resets the timer to 29 seconds (en_clearprev). Start spin enables the counters for the columns so that the user can attempt to match three of the same elements once again (en_spinfirst, en_spinsecond, en_spinthird). It also enables the timer to start counting down to zero (en_timer). Stop first column and stop second column perform similar functions as they stop the counter and store the value of their respective columns on separate registers (en_storefirst, en_storesecond) when their corresponding key is pressed. Stop third column is similar to these states but with the added function of comparing the three columns to see if they all match (en_storethird, en_compare). If the timer reaches zero it stops the first, second, and third column for the user. The logic for each state is then implemented by the next module.

Datapath:

The datapath module implements the game logic using always blocks which are activated by the enable signals received from the control module. The always blocks are as follows:

- Rate Divider (columns)

    - Three 27-bit rate dividers are used to send pulses to three 3-bit counters. The rate dividers correspond to the different columns and how quickly it changes elements. The one for the first column pulses every 35 million clock cycles for a slow speed while the second and third pulses every 25 million and 15 million respectively to achieve the medium and fast speeds.
- 3-bit counter (columns)
    - Like the previous always block, there are three 3-bit counters that represent each of the columns. The 3 bits allows for eight elements to be displayed in total and holds 3'd0 as its default value. It only counts up when it receives a pulse from their respective Rate Divider and loops back to the first element after the eighth element. It also has an additional function of displaying the win state.
    - For most cases, this always block functions same as above but the columns have an extra bit to be used by the reel module (explained later in document). When compare is enabled, the first, second, and third column will hold the value of 4'd9 when they all equal to each other. It will hold the values of 4'd11, 4'd12, and 4'd13 respectively when they do not. This will access the appropriate RAM sprites to be displayed in the win/loss screen state.
- 4-bit counter (score)
    - When compare is enabled, it either adds or subtracts one to the score depending on whether the user matched all three columns or not.
- Rate Divider / Down Counter (Timer)
    - The timer starts with 29 seconds on the clock. When 50 million clock cycles / one second has passed, this will get subtracted by one. In order to correctly show this on the hex display, the time remaining is split into two 4-bit numbers rather than one 5-bit number. The least significant digit is decremented every second, looping back to 9 after 0, while the most significant is decremented after 10 seconds. When there is no time left, it sends a signal to the datapath which stops the first, second, and column automatically.
- Counter (VGA Display)
    - This sends colour, and x and y position data to the VGA adapter in order to display the columns. It uses a counter that continuously loops in this sequence:

1	2	3		10	11	12		19	20	21
4	5	6		13	14	15		22	23	24
7	8	9		16	17	18		25	26	27

The colour at each individual pixel location is decided by an instantiated reel module.

Hex Decoder:

- Receives a 4-bit number as an input and turns on the appropriate segments in a seven segment display. Instantiated three times in the top-level module: HEX5 displays the user's score while HEX1 and HEX0 displays the most and least significant digit of the timer respectively.

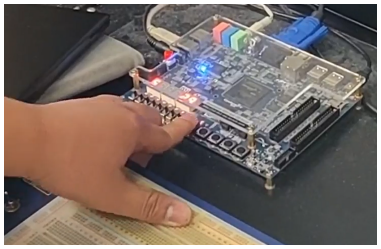
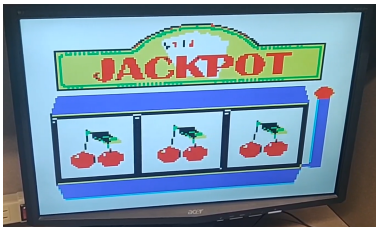
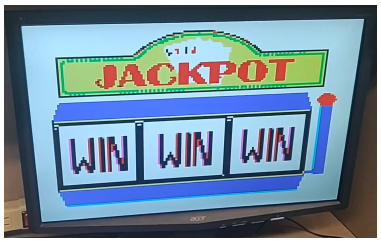
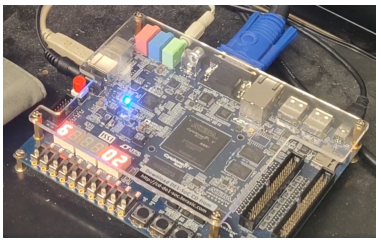
RAM (Sprites)

- A total of 12 RAM modules have been created for this design: 8 to represent the different column fruits (apple, banana, blueberry, cherry, grape, melon, peach, and pineapple) and 4 for the win/loss states. Each one has 900 3-bit words of memory that have been initialized using a .MIF file. This was produced using a program that converted our 30x30 bitmap images to the .MIF format.

Reel:

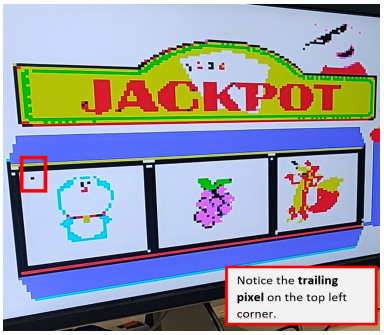
- Works in tandem with the previous VGA Display Counter in order to properly display the columns. Whereas the counter is in charge of the x and y coordinates, this is responsible for sending the correct colour data. It takes a 4-bit number as an input named "elements" and decides which of the previous 12 RAM modules is having its data read and outputted. Bits 3'd0 to 3'd7 are reserved for the elements of the columns, 4'd9 for the win screen and 4'd11 to 4'd13 for the lose screen. All of the addresses in the RAM are read using an up counter that starts from 4 and ends at 900. Value 4 was chosen as the starting value as beginning from 0 shifted the sprites to the right.

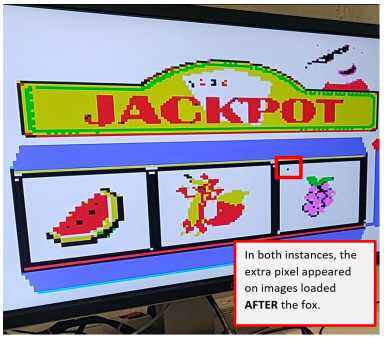
Report on Success

<table>
<tr>
<td> 
Switch is turned on to clear the previous columns and reset the timer</td>
<td> 
When switch is turned off, the machine starts spinning beginning with the first element (apple)</td>
<td> 
When the user matches three of the same elements...</td>
</tr>
<tr>
<td>  </td>
<td>  </td>
<td>  </td>
</tr>
</table>

a win screen is displayed.	When a user fails to match all three...	or the timer runs out...
 a lose screen is displayed		

The design worked as it was first outlined in the TA meeting and was achieved after days of debugging parts that did not work. The first one that we encountered was the clear screen state enabling counters when it was not supposed to. To fix this, we added a switch as an input to the datapath and made a condition to only start the columns and the timers when the switch is off. The display also had trailing pixels as shown in the following pictures:





Upon further inspection, the sprite for the fox was the only one that had pixels right at its edge. When a 1 pixel border was applied, the pixel disappeared. All of the other sprites were also changed to fruits since the characters looked washed out on the 3-bit colour we were working with.

What would you do differently?

Although we were mostly content with the final product of our VGA game, there are minor things that we could have done differently. If there is one thing that stands out about casino games, it is the noise, music and the sound of falling coins that makes you feel like you won something big! Slot machines are perhaps the best representation of it all. While we did attempt to implement audio files into our game and the FPGA, we were not able to get it to work. Unfortunately due to time constraints and more important issues, the idea did not make it into the final game. This is certainly one of our biggest regrets and something we would have loved to have. Another element that could have vastly improved and added more depth to our project would be scrolling animations. This was an idea that we considered but compromised towards the end due to its difficulty and requirement of time which we did not have.

Appendix:
Verilog Code (.v file also attached in email to view with proper formatting in Quartus):

```
module slotmachine_fpga (SW, KEY, CLOCK_50, HEX5, HEX4, HEX3, HEX2, HEX1,
HEX0, xposn, yposn, colour, WriteEn);
      input [9:0] SW;
      input [3:0] KEY;
      input CLOCK_50;

      output [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

      output [7:0] xposn;
      output [6:0] yposn;
      output [2:0] colour;
      output WriteEn;

      wire [3:0] firstcol, secondcol, thirdcol;
      wire [3:0] score;
      wire [3:0] timerMSD, timerLSD;

      slotmachine S0(
      .clk (CLOCK_50),
      .resetn (KEY[0]),
      .switch (SW[0]),
      .key (KEY[3:1]),
      .firstcol (firstcol),
      .secondcol (secondcol),
      .thirdcol (thirdcol),
      .score (score),
      .timerMSD (timerMSD),
      .timerLSD (timerLSD),
      .xposn (xposn),
      .yposn (yposn),
      .colour (colour),
      .WriteEn (WriteEn)
      );

      // First column - fastest

      // hexDecoder COL1 (HEX4, firstcol);
      // hexDecoder COL2 (HEX3, secondcol);
      // hexDecoder COL3 (HEX2, thirdcol);

  assign HEX4 = 7'd127;
  assign HEX3 = 7'd127;
```

```
  assign HEX2 = 7'd127;
  hexDecoder TIME_MSD (HEX1, timerMSD);
  hexDecoder TIME_LSD (HEX0, timerLSD);

  hexDecoder SCORE (HEX5, score);
endmodule

module slotmachine (
      input clk,
      input resetn,
      input switch,
      input [2:0] key,

      output [3:0] firstcol, secondcol, thirdcol,
      output [3:0] score,
      output [3:0] timerMSD, timerLSD,

      output [7:0] xposn,
      output [6:0] yposn,
      output [2:0] colour,
      output WriteEn
      );

      //lots of wires to connect our datapath and control
      wire timerover;
      wire en_spinfirst, en_spinsecond, en_spinthird;
      wire en_storefirst, en_storesecond, en_storethird;
      wire en_compare, en_clearprev, en_timer, en_winscreen;

  control C0(
      clk,
      resetn,
      switch,
      key [2:0],
      timerover,
      en_spinfirst,
      en_spinsecond,
      en_spinthird,
      en_storefirst,
      en_storesecond,
      en_storethird,
      en_compare,
      en_clearprev,
      en_timer,
      en_winscreen,
      WriteEn
```

```
	);
	datapath D0(
		clk,
		resetn,
		switch,
		en_spinfirst,
		en_spinsecond,
		en_spinthird,
		en_storefirst,
		en_storesecond,
		en_storethird,
		en_compare,
		en_clearprev,
		en_timer,
		en_winscreen,
		firstcol,
		secondcol,
		thirdcol,
		score,
		timerMSD,
		timerLSD,
		xposn,
		yposn,
		colour,
		timerover
	);
endmodule


module control(
	input clk,
	input resetn,
	input switch,
	input [2:0] key,

	input timerover,

	output reg en_spinfirst, en_spinsecond, en_spinthird,
	output reg en_storefirst, en_storesecond, en_storethird,
	output reg en_compare, en_clearprev, en_timer, en_winscreen,
	output reg WriteEn
	);
	reg [2:0] current_state, next_state;

	localparam  SHOW_PREV_SCREEN = 3'd0,
		    CLEAR_SCREEN = 3'd1,
```

```
                START_SPIN = 3'd2,
                STOP_FIRST_COL = 3'd3,
                STOP_SECOND_COL = 3'd4,
                STOP_THIRD_COL = 3'd5;

      // Next state logic aka our state table
      always@(*)
      begin: state_table
            case (current_state)
                  SHOW_PREV_SCREEN: if (switch) next_state = CLEAR_SCREEN;
                  else next_state = SHOW_PREV_SCREEN;

                  CLEAR_SCREEN: if (!switch) next_state = START_SPIN;
                  else next_state = CLEAR_SCREEN;

                  START_SPIN: if (!key[2]) next_state = STOP_FIRST_COL;
                  else if (timerover) next_state = STOP_FIRST_COL;
                  else next_state = START_SPIN;

                  STOP_FIRST_COL: if (!key[1]) next_state =
STOP_SECOND_COL;
             else if (timerover) next_state = STOP_SECOND_COL;
             else next_state = STOP_FIRST_COL;
                  STOP_SECOND_COL: if (!key[0]) next_state =
STOP_THIRD_COL;
           else if (timerover) next_state = STOP_THIRD_COL;
           else next_state = STOP_SECOND_COL;
                  STOP_THIRD_COL: next_state = SHOW_PREV_SCREEN;
            default:    next_state = SHOW_PREV_SCREEN;
      endcase
      end // state_table

      // Output logic aka all of our datapath control signals
      always @(*)
      begin: enable_signals
      // By default make all our signals 0 to avoid latches.
      // This is a different style from using a default statement.
      // It makes the code easier to read.  If you add other out
      // signals be sure to assign a default value for them here.
      // Enables RateDiv's, spins the columns
      en_spinfirst = 1'b0;
      en_spinsecond = 1'b0;
      en_spinthird = 1'b0;

      // Stores the columns in a register for comparison
      en_storefirst = 1'b0;
```

```
      en_storesecond = 1'b0;
      en_storethird = 1'b0;
      // Enables comparison between the three columns, adds/subtracts
score
      en_compare = 1'b0;
      // Clears previous columns
      en_clearprev = 1'b0;
      // Enables timer (99 seconds)
      en_timer = 1'b0;

      // Enables VGA
      WriteEn = 1'b0;

      // Enables win screen to be displayed
      en_winscreen = 1'b0;
      case (current_state)
            SHOW_PREV_SCREEN: begin
     en_winscreen = 1'b1;
     WriteEn = 1'b1;
            end
            CLEAR_SCREEN: begin
     en_clearprev = 1'b1;
     en_spinfirst = 1'b0;
     en_spinsecond = 1'b0;
     en_spinthird = 1'b0;
     en_timer = 1'b0;
     WriteEn = 1'b1;
            end
            START_SPIN: begin
     en_timer = 1'b1;
     en_spinfirst = 1'b1;
     en_spinsecond = 1'b1;
     en_spinthird = 1'b1;
     WriteEn = 1'b1;
            end

            STOP_FIRST_COL: begin
     en_timer = 1'b1;
     en_storefirst = 1'b1;
     en_spinsecond = 1'b1;
     en_spinthird = 1'b1;
     WriteEn = 1'b1;
            end
            STOP_SECOND_COL: begin
     en_timer = 1'b1;
     en_storesecond = 1'b1;
```

```
      en_spinthird = 1'b1;
      WriteEn = 1'b1;
            end
            STOP_THIRD_COL: begin
      en_timer = 1'b1;
      en_storethird = 1'b1;
      en_compare = 1'b1;
      WriteEn = 1'b1;
            end
       // default: // don't need default since we already made sure all of
our outputs were assigned a value at the start of the always block
       endcase
       end // enable_signals

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

module datapath(
       input clk,
       input resetn,
  input switch,

  input en_spinfirst, en_spinsecond, en_spinthird,
       input en_storefirst, en_storesecond, en_storethird,
       input en_compare, en_clearprev, en_timer, en_winscreen,

       output reg [3:0] firstcol, secondcol, thirdcol,
       output reg [3:0] score, // Currently capped at 15, for testing only
       output reg [3:0] timerMSD, timerLSD,

  output reg [7:0] xposn,
  output reg [6:0] yposn,
  output reg [2:0] colour,
       output reg timerover
       );

       // Registers for RateDiv always block
       reg [26:0] speedfirst, speedsecond, speedthird;
       reg pulsefirst, pulsesecond, pulsethird;
```

```
    // Register for timer always block
    reg [26:0] timerseconds;

 // Registers for enabling the drawing (or clearing) of columns
 reg plotfirst, plotsecond, plotthird;
 reg contfirst, contsecond, contthird;
 reg clearfirst, clearsecond, clearthird;

 wire [2:0] colourfirst, coloursecond, colourthird;

 // Instantiate reel modules to allow for drawing of columns
 reel R1(
 .clk(clk),
 .resetn(resetn),
 .plot(plotfirst),
 .clear(clearfirst),
 .elements(firstcol),
 .colour(colourfirst)
 );

 reel R2(
 .clk(clk),
 .resetn(resetn),
 .plot(plotsecond),
 .clear(clearsecond),
 .elements(secondcol),
 .colour(coloursecond)
 );

 reel R3(
 .clk(clk),
 .resetn(resetn),
 .plot(plotthird),
 .clear(clearthird),
 .elements(thirdcol),
 .colour(colourthird)
 );

// Third column's RateDiv - 15mil clock cycles - fast
always @(posedge clk) begin
 if (resetn == 1'b0) begin
  speedthird <= 27'd0;
  pulsethird <= 1'b0;
 end
 else if (speedthird == 27'd14999999) begin
  speedthird <= 27'd0;
```

```
  pulsethird <= 1'b1;
 end
 else if (en_spinthird == 1'b1 && switch == 1'b0) begin
  speedthird <= speedthird + 1;
  pulsethird <= 1'b0;
 end
 else if (en_storethird == 1'b1) begin
  speedthird <= 27'd0;
  pulsethird <= 1'b0;
 end
 else if (en_clearprev == 1'b1) begin
  speedthird <= 27'd0;
  pulsethird <= 1'b0;
 end

 else begin
  speedthird <= 27'd0;
  pulsethird <= 1'b0;
 end

end

// Second column's RateDiv - 25mil clock cycles - medium
always @(posedge clk) begin
 if (resetn == 1'b0) begin
  speedsecond <= 27'd0;
  pulsesecond <= 1'b0;
 end
 else if (speedsecond == 27'd24999999) begin
  speedsecond <= 27'd0;
  pulsesecond <= 1'b1;
 end
 else if (en_spinsecond == 1'b1 && switch == 1'b0) begin
  speedsecond <= speedsecond + 1;
  pulsesecond <= 1'b0;
 end
 else if (en_storesecond == 1'b1) begin
  speedsecond <= 27'd0;
  pulsesecond <= 1'b0;
 end
 else if (en_clearprev == 1'b1) begin
  speedsecond <= 27'd0;
  pulsesecond <= 1'b0;
 end
 else begin
  speedsecond <= 27'd0;
```

```
  pulsesecond <= 1'b0;
 end

end

// First column's RateDiv - 35mil clock cycles - slow
always @(posedge clk) begin
 if (resetn == 1'b0) begin
  speedfirst <= 27'd0;
  pulsefirst <= 1'b0;
 end
 else if (speedfirst == 27'd34999999) begin
  speedfirst <= 27'd0;
  pulsefirst <= 1'b1;
 end
 else if (en_spinfirst == 1'b1 && switch == 1'b0) begin
  speedfirst <= speedfirst + 1;
  pulsefirst <= 1'b0;
 end
 else if (en_storefirst == 1'b1) begin
  speedfirst <= 27'd0;
  pulsefirst <= 1'b0;
 end
 else if (en_clearprev == 1'b1) begin
  speedfirst <= 27'd0;
  pulsefirst <= 1'b0;
 end

 else begin
  speedfirst <= 27'd0;
  pulsefirst <= 1'b0;
 end
end

// Three 3-bit counters that represent the three columns
always @(posedge clk) begin
 if (resetn == 1'b0) begin
  firstcol <= 3'd0;
  secondcol <= 3'd0;
  thirdcol <= 3'd0;

  clearfirst <= 1'b0;
  clearsecond <= 1'b0;
  clearthird <= 1'b0;
 end
 if (pulsefirst == 1'b1) begin
```

```
   firstcol <= firstcol + 1;
  end

  else if (firstcol == 3'd7) begin
   firstcol <= 3'd0;
  end
  if (pulsesecond == 1'b1) begin
   secondcol <= secondcol + 1;
  end

  else if (secondcol == 3'd7) begin
   secondcol <= 3'd0;
  end
  if (pulsethird == 1'b1) begin
   thirdcol <= thirdcol + 1;
  end

  else if (thirdcol == 3'd7) begin
   thirdcol <= 3'd0;
  end
  if (en_clearprev == 1'b1) begin
   firstcol <= 3'd0;
   secondcol <= 3'd0;
   thirdcol <= 3'd0;

   clearfirst <= 1'b1;
   clearsecond <= 1'b1;
   clearthird <= 1'b1;
  end

  if (en_winscreen == 1'b1 && firstcol == secondcol && firstcol ==
thirdcol) begin
   firstcol <= 4'd9;
   secondcol <= 4'd9;
   thirdcol <= 4'd9;
  end

  if (en_winscreen == 1'b1 && (firstcol != secondcol || firstcol !=
thirdcol)) begin
   firstcol <= 4'd11;
   secondcol <= 4'd12;
   thirdcol <= 4'd13;
  end

  else begin
   clearfirst <= 1'b0;
```

```
  clearsecond <= 1'b0;
  clearthird <= 1'b0;
 end

end

// Compares the three columns - keeps track of score
always @(posedge clk) begin
 if (resetn == 1'b0) begin
  score <= 4'd5;
 end
 else if (en_compare) begin
  if (firstcol == secondcol && firstcol == thirdcol) score <= score + 1;
  else score <= score - 1;
 end
 else if (score == 4'd0 || score == 4'd15) begin
  // Need to change behaviour after testing
  // Display LOSE on hex display or show lose screen on VGA
  score <= 4'd5;
 end
end

// Timer - user has 29 seconds to stop all three columns
always @(posedge clk) begin
 if (resetn == 1'b0) begin
  timerseconds <= 27'd0;
  timerMSD <= 4'd2;
  timerLSD <= 4'd9;
  timerover <= 1'b0;
 end
 else if (timerMSD == 4'd0 && timerLSD == 4'd0) begin
  timerMSD <= 4'd2;
  timerLSD <= 4'd9;
  timerover <= 1'b1;
 end
 else if (en_timer == 1'b1 && switch == 1'b0) begin
  if (timerseconds != 27'd49999999) begin
   timerseconds <= timerseconds + 1;
  end
  else if (timerseconds == 27'd49999999) begin
   timerseconds <= 27'd0;
   if (timerLSD != 4'd0) begin
    timerLSD <= timerLSD - 1;
   end
   else if (timerLSD == 4'd0) begin
    timerMSD <= timerMSD - 1;
```

```
     timerLSD <= 4'd9;
    end
   end
  end
  else if (en_clearprev == 1'b1) begin
   timerseconds <= 27'd0;
   timerMSD <= 4'd2;
   timerLSD <= 4'd9;
   timerover <= 1'b0;
  end
 end
 // Draws the three columns on the VGA
 // Responsible for giving x, y, and colour
 always@(posedge clk) begin
  if(!resetn) begin
   xposn <= 8'd16;
   yposn <= 7'd62;
   colour <= 3'd7;

   plotfirst <= 1'b0;
   plotsecond <= 1'b0;
   plotthird <= 1'b0;

   contfirst <= 1'b1;
   contsecond <= 1'b0;
   contthird <= 1'b0;
  end

  // Keeps looping to draw all three columns
  else if(en_clearprev || en_spinfirst || en_spinsecond || en_spinthird ||
en_winscreen) begin

   // Draws first column
   if(contfirst) begin
    colour <= colourfirst;

    // Keep accessing memory and drawing first column
    // boundary is +29 from starting pixel
    // yposn - add 1 from actual boundary
    // xposn - keep actual boundary
    if(yposn != 7'd92 && xposn != 8'd45) begin
     contfirst <= 1'b1;
     plotfirst <= 1'b1;
     xposn <= xposn + 1;
    end
```

```
 if(yposn != 7'd92 && xposn == 8'd45) begin
  contfirst <= 1'b1;
  plotfirst <= 1'b1;

  xposn <= 8'd16;
  yposn <= yposn + 1;
 end

 // Move on to second column
 if(yposn == 7'd91 && xposn == 8'd45) begin
  contfirst <= 1'b0;
  contsecond <= 1'b1;

  plotfirst <= 1'b0;

  xposn <= 8'd59;
  yposn <= 7'd62;
 end

end

// Draws second column
else if(contsecond) begin
 colour <= coloursecond;

 if(yposn != 7'd92 && xposn != 8'd88) begin
  contsecond <= 1'b1;
  plotsecond <= 1'b1;
  xposn <= xposn + 1;
 end

 if(yposn != 7'd92 && xposn == 8'd88) begin
  contsecond <= 1'b1;
  plotsecond <= 1'b1;

  xposn <= 8'd59;
  yposn <= yposn + 1;
 end

 if(yposn == 7'd91 && xposn == 8'd88) begin
  contsecond <= 1'b0;
  contthird <= 1'b1;

  plotsecond <= 1'b0;

  xposn <= 8'd102;
```

```
   yposn <= 7'd62;
  end

 end

 // Draws third column
 else if(contthird) begin
  colour <= colourthird;

  if(yposn != 7'd92 && xposn != 8'd131) begin
   contthird <= 1'b1;
   plotthird <= 1'b1;
   xposn <= xposn + 1;
  end

  if(yposn != 7'd92 && xposn == 8'd131) begin
   contthird <= 1'b1;
   plotthird <= 1'b1;

   xposn <= 8'd102;
   yposn <= yposn + 1;
  end

  if(yposn == 7'd91 && xposn == 8'd131) begin
   contthird <= 1'b0;
   contfirst <= 1'b1;

   plotthird <= 1'b0;

   xposn <= 8'd16;
   yposn <= 7'd62;
  end

 end

end

else begin
 xposn <= 8'd16;
 yposn <= 7'd62;
 colour <= 3'd7;

 plotfirst <= 1'b0;
 plotsecond <= 1'b0;
 plotthird <= 1'b0;
```

```
   contfirst <= 1'b1;
   contsecond <= 1'b0;
   contthird <= 1'b0;
  end

 end
endmodule

module hexDecoder (s, x);
 input [3:0] x;
 output [6:0] s;

 // Turns the top-middle segment on or off
 assign s[0] = (~x[3] & ~x[2] & ~x[1] & x[0]) |
      (~x[3] & x[2] & ~x[1] & ~x[0]) |
      (x[3] & ~x[2] & x[1] & x[0]) |
      (x[3] & x[2] & ~x[1] & x[0]);
 // Turns the top-right segment on or off
 assign s[1] = (~x[3] & x[2] & ~x[1] & x[0]) |
      (~x[3] & x[2] & x[1] & ~x[0]) |
      (x[3] & ~x[2] & x[1] & x[0]) |
      (x[3] & x[2] & ~x[1] & ~x[0]) |
      (x[3] & x[2] & x[1] & ~x[0]) |
      (x[3] & x[2] & x[1] & x[0]);
 // Turns the bottom-right segment on or off
 assign s[2] = (~x[3] & ~x[2] & x[1] & ~x[0]) |
      (x[3] & x[2] & ~x[1] & ~x[0]) |
      (x[3] & x[2] & x[1] & ~x[0]) |
      (x[3] & x[2] & x[1] & x[0]);
 // Turns the bottom-middle segment on or off
 assign s[3] = (~x[3] & ~x[2] & ~x[1] & x[0]) |
      (~x[3] & x[2] & ~x[1] & ~x[0]) |
      (~x[3] & x[2] & x[1] & x[0]) |
      (x[3] & ~x[2] & ~x[1] & x[0]) |
      (x[3] & ~x[2] & x[1] & ~x[0]) |
      (x[3] & x[2] & x[1] & x[0]);
 // Turns the bottom-left segment on or off
 assign s[4] = (~x[3] & ~x[2] & ~x[1] & x[0]) |
      (~x[3] & ~x[2] & x[1] & x[0]) |
      (~x[3] & x[2] & ~x[1] & ~x[0]) |
      (~x[3] & x[2] & ~x[1] & x[0]) |
      (~x[3] & x[2] & x[1] & x[0]) |
      (x[3] & ~x[2] & ~x[1] & x[0]);
 // Turns the top-left segment on or off
 assign s[5] = (~x[3] & ~x[2] & ~x[1] & x[0]) |
      (~x[3] & ~x[2] & x[1] & ~x[0]) |
```

```
      (~x[3] & ~x[2] & x[1] & x[0]) |
      (~x[3] & x[2] & x[1] & x[0]) |
      (x[3] & x[2] & ~x[1] & x[0]);
 // Turns the middle segment on or off
 assign s[6] = (~x[3] & ~x[2] & ~x[1] & ~x[0]) |
      (~x[3] & ~x[2] & ~x[1] & x[0]) |
      (~x[3] & x[2] & x[1] & x[0]) |
      (x[3] & x[2] & ~x[1] & ~x[0]);
endmodule


// NEW - module to draw columns
module reel(
 input clk,
 input resetn,
 input plot,
 input clear,
 input [3:0] elements,
 output reg [2:0] colour
);
 // Wire to access addresses in memory (sprite)
 reg [9:0] address;

 // Wire for the different memory's colours
 wire [2:0] applecol;
 wire [2:0] bananacol;
 wire [2:0] blueberrycol;
 wire [2:0] cherrycol;
 wire [2:0] grapecol;
 wire [2:0] meloncol;
 wire [2:0] peachcol;
 wire [2:0] pineapplecol;
 wire [2:0] win1col;
 wire [2:0] win2col;
 wire [2:0] win3col;
 wire [2:0] lose2col;
 wire [2:0] lose3col;
 apple E0(
  .address(address),
  .clock(clk),
  .wren(1'b0),
  .q(applecol)
 );

 banana E1(
  .address(address),
```

```
 .clock(clk),
 .wren(1'b0),
 .q(bananacol)
);

blueberry E2(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(blueberrycol)
);

cherry E3(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(cherrycol)
);

grape E4(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(grapecol)
);

melon E5(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(meloncol)
);

peach E6(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(peachcol)
);

pineapple E7(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(pineapplecol)
);
```

```
win1 E8(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(win1col)
);

win2 E9(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(win2col)
);

win3 E10(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(win3col)
);

lose2 E11(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(lose2col)
);

lose3 E12(
 .address(address),
 .clock(clk),
 .wren(1'b0),
 .q(lose3col)
);

// X and Y position register
always @(posedge clk) begin
 if(!resetn) begin
  address <= 10'd4;
 end

 if(plot && address != 10'd900) begin
  address <= address + 1;
 end
```

```
 if(plot && address == 10'd899) begin
  address <= 10'd4;
 end

 if(!plot) begin
  address <= 10'd4;
 end

end

// Colour register
always @(posedge clk) begin
 if(!resetn) begin
  colour <= 3'd7;
 end

 else if(plot && elements == 3'd0) begin
  colour <= applecol;
 end

 else if(plot && elements == 3'd1) begin
  colour <= bananacol;
 end

 else if(plot && elements == 3'd2) begin
  colour <= blueberrycol;
 end

 else if(plot && elements == 3'd3) begin
  colour <= cherrycol;
 end

 else if(plot && elements == 3'd4) begin
  colour <= grapecol;
 end

 else if(plot && elements == 3'd5) begin
  colour <= meloncol;
 end

 else if(plot && elements == 3'd6) begin
  colour <= peachcol;
 end

 else if(plot && elements == 3'd7) begin
  colour <= pineapplecol;
```

```
  end

  else if(plot && elements == 4'd8) begin
   colour <= win1col;
  end

  else if(plot && elements == 4'd9) begin
   colour <= win2col;
  end

  else if(plot && elements == 4'd10) begin
   colour <= win3col;
  end

  else if(plot && elements == 4'd11) begin
   colour <= win1col;
  end

  else if(plot && elements == 4'd12) begin
   colour <= lose2col;
  end

  else if(plot && elements == 4'd13) begin
   colour <= lose3col;
  end

  else if(plot && clear) begin
   colour <= 3'd7;
  end
 end

endmodule
```