Abstract

We have created a Vegas-style slot machine, given its popularity in casinos and the popularity of online gambling. Our slot machine relies on camera-controlled inputs from the user, instead of physical touch, like the traditional slot machine uses. In order to implement this, we divided our project into two parts: the inputs from the camera and the outputs to the screen. In the end, although the full project was not implemented as intended, an interactive and fun demo/alternative was created.
# Table of Contents

Description of the System..................................................................................................................1  
Modules—Daneaya............................................................................................................................1  
Modules—Laura................................................................................................................................4  
Testing and Debugging of Video Input..............................................................................................7  
Conclusions........................................................................................................................................8  
Appendix............................................................................................................................................9  

Description of the System

Overview

The project is divided into two major components, and these are broken up into modules. The first component is the video input component. The video input component processes all of the data coming from the camera and it also determines the center of mass of any red pixels present in the view of the camera screen. This is then passed along to the second component of the system. The second component is the game output component. This component handles the processing of the input and is responsible for game functionality.

NTSC Decoder (by Daneaya)

This module, named ntsc_decode.v, is taken from the 6.111 website. This module was not modified for our particular project. Using the 6.111 labkit’s composite-in port, this module takes in the camera’s video input and outputs a 30-bit YCrCb signal. This signal contains the Luminance value (Y) in bits 29 through 20. The Chrominance values (Cr and Cb) are located in bits 19 through 0. The module also outputs the signals for field, vertical and horizontal sync, and a data valid signal.

YCrCb to RGB Converter (by Daneaya)

\[
\begin{bmatrix}
Y \\
Cr \\
Cb
\end{bmatrix} =
\begin{bmatrix}
0.299 & 0.587 & 0.114 \\
-0.147 & -0.0289 & 0.436 \\
0.615 & -0.515 & -0.100
\end{bmatrix}
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

A module for the YCrCb to RGB converter was already provided by Xilinx. This module takes in the 10-bit Y, Cr, and Cb signals and on its output, produces red, green, and blue signals, which are each 8-bits. This 24-bit signal is then concatenated by cleaving the last 2 bits of each of the red, green, and blue signals. The reasoning for this is explained in the next section. The final product is the 18-bit RGB signal. This signal is what is passed through the remaining modules and is also what is used to make future calculations.

NTSC to ZBT (by Daneaya)

The ZBT RAM is able to store 36-bits per address, but the information that we want to store is 24-bits in length. We could store one RGB pixel value per address, but that was decided to be a waste of RAM. A better use of the ZBT would be to try and store two pixels per address. To
do this, we need to cut the information in each pixel down to 18 bits. It was decided to cut each of the red, green, and blue signals by its two lowest bits. This would not add any negative affects because we would not need to display the actual camera image to the screen, so preciseness was not necessary. The 18-bit RGB signal is input into the ntsc2zbt module. This module prepares the data to be stored into the ZBT RAM. Other inputs to the module are, the 65 MHz clock, 25 MHz camera clock, field, vertical and horizontal sync, and data valid signals. This module outputs a 19-bit address to write data to, 36-bits of data, and a write enable signal to the ZBT RAM.

The functionality of the original 6.111 ntsc2zbt module only stored the Luminance value (Y) of the YCrCb signal, which provided no color. Also, since the module did not have to account for color, it was formed to store 4 pixels per address. Some modifications were made to be able to store color as well as 2 bits per address.

**ZBT_6.111 (by Daneaya)**

The data provided by the ntsc2zbt module gets written to the ZBT RAM with the help of a module called zbt_6111.v. The write enable is delayed by two clock cycles which is caused by the two-stage pipeline for the data. This module was not modified to fit with our project.

**VRAM_Display (by Daneaya)**

This module is responsible for reading the data from the ZBT RAM. The module makes sure the data is latched at a specific time, since there is a two-cycle read-write latency on the information. The module outputs a 19-bit address signal that it wants to read from memory. It receives the data stored at that address and outputs each pixel, which is the modified 18-bit RGB signal. Modifications to this module were done to handle the two pixels per address instead of the original four. This module operates at 65 MHz.

**Red Pixel/Center of Mass Detector (by Daneaya)**

The Red Pixel/Center of Mass Finder (com.v) takes in the 18-bit RGB signal from the VRAM_Display module along with hcount and vcount. First, the module adds two zeros to the ends of each of the red, green, and blue signals. This is to make each signal into 8 bits again. The pixel is then passed through a threshold test. This is minimum/maximum for the values of red, green, and blue, to be considered a red pixel. If the pixel passes this test, it is then determined if the pixel is in fact within the camera range. The camera’s field of vision and output is 720x480 pixels, whereas the VGA display is 1024x768. This means that we don’t want to consider pixels beyond hcount = 720 and vcount = 480. If the pixel is in the proper range, the number of passing pixels is incremented by one and the pixel’s hcount and vcount are added to the sum of the x and y positions, respectively.

This is continued until the end of the camera’s frame is reached. If the total number of passing pixels (weight) is less than 40, the module outputs the center of mass as zero. This is to
signify that there is no hand present in the view of the camera. If the weight is greater than 40, two
dividers are used to calculate the center of mass. This is done by dividing the total sum of the x or
y positions by the total number of passing pixels. This produces the same calculation as the center
of mass calculation. The only problem is this center of mass is in reference to the 720x480 camera
screen. We actually need the center of mass to be up-scaled to the entire 1024x768 screen. It was
calculated that the 1024x768 screen is 1.42 times the 720x480 in the horizontal direction and 1.6
times larger in the y direction. To fix this problem two more dividers and two multipliers are used.
The x-position center of mass for the 720x480 screen is multiplied by 142 and then divided by 100.
This gives the center of mass for the x-position of the entire screen. The same is done for the y
position, except the number multiplied by is 160. The resulting x and y positions are then passed
along to the rest of the modules in the project. These modules are those that deal with game
functionality and video display on screen.
Module: game_logic_FSM
Creator: Laura
Inputs: vclock, reset, hcount, vcount, hsync, vsync, blank, phsync, pvsync, pblank, payout, x_lever_in, y_lever_in
Outputs: amount_won, cash_pot, current_bet_amount, state, pixel
Description:
The FSM within this module controls the slot machine's behavior. The FSM became more straightforward the more I made the code modular and delegated tasks to other modules. This module is also where the “OR-ing” of the various pixels takes place, before I pass the final pixel value to the topmost module (lab5.v).
This module would have interfaced with Daneaya's part of the project—she would have been the one providing x_lever_in and y_lever_in (better names for these would have been “x_center_of_mass” and “y_center_of_mass”), and the rest would have been taken care of. Instead, I hard-coded values to the switches on the board and used those as x_lever_in and y_lever_in for our demo.

Module: show_me_title
Creator: Laura
Inputs: vclock, reset, hcount, vcount
Outputs: addr, out_of_bounds
Description:
This module provides the address for the display of the title of our slot machine (“Show Me SevenS”) from its ROM.

Module: show_bet25
Creator: Laura
Inputs: vclock, reset, hcount, vcount
Outputs: addr, out_of_bounds
Description:
This module provides the address for the display of the three bet buttons (bet25, bet50, and bet100) from their ROMs. The buttons' ROMs are the same size, so this module works for all three, despite this module's name.

Module: reel_animation
Creator: Laura
Inputs: vclock, reset, x, y, width, height, rspeed, hcount, vcount, reel_go, start_addr, stop_addr
Outputs: addr, out_of_bounds, has_stopped
Description:
This elegant module is in charge of starting a reel in the correct place, stopping it at the correct place, and animating the reel so that it seems circular at whatever speed you desire. The out_of_bounds signal lets me know that hcount and vcount are not within the rectangular window that reveals the current state of the reel, and thus, the color black should be displayed. The has_stopped signal lets the game logic FSM know that a reel has stopped. This is important because before I had this signal, the reels would stop in whatever order they liked because one reel
would find its stop address before the first reel did, and that's not how I wanted the slot machine to behave.

**Module: lever_display**
*Creator: Laura*
*Inputs: reset, vclock, x, y, hcount, vcount,*
*Outputs: pixel, bet_placed, hit, original_pos, bet_amount*
*Description:*
This module contains an FSM that takes care of the user's on-screen button selections, lever pull-down, and animation of the lever as it returns to its original position near the top of the slot machine. It interacts with the game logic FSM, described above.

**Module: show_pointer**
*Creator: Laura*
*Inputs: vclock, reset, x, y, user_off, hcount, vcount*
*Outputs: addr, out_of_bounds*
*Description:*
This module provides the address of the ROM for the pointer, whose corresponding “dout” can be displayed after that dout is translated into a 24-bit RGB value. The pointer is the on-screen representation of the user's center of mass. At first, the pointer was a “blob,” and it didn't require this module, but to make the slot machine more visually-appealing, I made the pointer look like a hand and put that image in a ROM.

**Module: hover_two_seconds**
*Creator: Laura*
*Inputs: reset, vclock, off, user_x, user_y, x_upper, y_upper, width, height*
*Outputs: two_seconds*
*Description:*
This module's output returns a “1” the moment a user has entered the rectangle formed on-screen by x_upper, y_upper, width, and height. (It has nothing to do with two seconds anymore.) This is useful for knowing if the user has activated any of the bet buttons or the lever.

**Module: get_rgb**
*Creator: Laura*
*Inputs: dout, out_of_bounds*
*Outputs: pixel*
*Description:*
This module takes in a “dout” from a ROM, which represents 1 out of 16 colors and returns its associated 24-bit RGB value.

**Module: get_stop_addr**
*Creator: Laura*
*Inputs: rand_num*
*Outputs: stop_addr*
**Description:**

This module takes in a random number between 0 and 31 and provides the stop address of the ROM that corresponds to that random number's associated symbol (cherries, diamond, seven, or blank). This is called “virtual reel mapping” because the random number can “land” on a number from 0 to 31, and these random numbers have to be mapped to the “stops” on the actual reel, and usually there are less actual stops than there are virtual stops. As you can see in the table below, most of the random numbers correspond to blank stops, so that the player will not win so often. Also, a player will never be able to land on “Cherries15”–it is simply on the reel for show!

<table>
<thead>
<tr>
<th>Actual Reel</th>
<th>32 virtual reel stops</th>
<th>Stop_addr (of ROM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diamond 1</td>
<td>5,6,7</td>
<td>272140</td>
</tr>
<tr>
<td>Blank 2</td>
<td>15,16,17,18</td>
<td>0</td>
</tr>
<tr>
<td>Seven3</td>
<td>1</td>
<td>18040</td>
</tr>
<tr>
<td>Blank4</td>
<td>11,12,13,14</td>
<td>36300</td>
</tr>
<tr>
<td>Cherries5</td>
<td>9</td>
<td>54340</td>
</tr>
<tr>
<td>Blank6</td>
<td>10</td>
<td>72600</td>
</tr>
<tr>
<td>Diamond7</td>
<td>4</td>
<td>90640</td>
</tr>
<tr>
<td>Blank8</td>
<td></td>
<td>108900</td>
</tr>
<tr>
<td>Cherries9</td>
<td>8</td>
<td>126940</td>
</tr>
<tr>
<td>Blank10</td>
<td>31,0</td>
<td>145200</td>
</tr>
<tr>
<td>Seven11</td>
<td>2</td>
<td>163240</td>
</tr>
<tr>
<td>Blank12</td>
<td>24,25,26,27,28,29,30</td>
<td>181500</td>
</tr>
<tr>
<td>Seven13</td>
<td>3</td>
<td>199540</td>
</tr>
<tr>
<td>Blank14</td>
<td>21,22,23</td>
<td>217800</td>
</tr>
<tr>
<td>Cherries15</td>
<td></td>
<td>235840</td>
</tr>
<tr>
<td>Blank16</td>
<td>19,20</td>
<td>254100</td>
</tr>
</tbody>
</table>

**Module: get_winnings**

**Creator:** Laura  
**Inputs:** bet_amount, stop_addr1, stop_addr2, stop_addr3  
**Outputs:** winnings  
**Description:**  
This module takes in the addresses of where the reels stopped on the ROM and figures out if they correspond to winning combinations. It then outputs the winnings based on the combination and the amount of the bet placed. The table below shows the pay table for our slot machine.

<table>
<thead>
<tr>
<th>Winning Combos</th>
<th>Bet=25</th>
<th>Bet=50</th>
<th>Bet=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three 7’s</td>
<td>500</td>
<td>1000</td>
<td>1500</td>
</tr>
<tr>
<td>Three Diamonds</td>
<td>200</td>
<td>400</td>
<td>600</td>
</tr>
<tr>
<td>Three Cherries</td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>Any 2 Cherries</td>
<td>50</td>
<td>100</td>
<td>150</td>
</tr>
<tr>
<td>Any Cherry</td>
<td>25</td>
<td>50</td>
<td>75</td>
</tr>
</tbody>
</table>

**Module: get_rand**

*Creator:* Laura  
*Inputs:* reset, velock, x, y  
*Outputs:* rand_num  
*Description:*  
This module provides a random number between 0 and 31 based on x and y (the user's center of mass) and an internal count. It is necessary so that the slot machine will have a random feel to it.

**Testing and Debugging of Video Input** *(by Daneaya)*

This part of the project was considerably a lot harder than expected. Initially, a lot of time was spent just trying to get a live feed from the video camera. After hours of thinking the coding was wrong, I figured out that the plug for the camera was put into a faulty outlet. Other issues dealt with the conversion from storing black and white pixels to color pixels. Modifying the nts2zbt and vram-display modules was a lot harder than it seemed. There were lots of subtle changes that needed to be made and I kept forgetting to make a small change before I started compiling, so time was an issue.

Testing and debugging the center of mass detector was fairly straightforward, as I used testing as a way to see what was happening on screen. For testing, I displayed the color image from the camera on screen and I used the vga_output signals in the wrapper module to output a green square of where the system thought the center of mass was. In the beginning the center of mass seemed to be all wrong. After doing a lot of experimental testing, I found the problem to be an overflow of registers in the system. I made this fix by making the registers larger. Another problem that I continually encountered was the red color detection. Initially I was using trial and error to figure out what a good threshold was, but I realized that there was a better way to do it. Using a cursor that moved around the screen, the RGB values of the intersection of the cursors was displayed onto the 64-bit hex display. Using this, I placed the red paper to be used for the glove in front of the cursor and read the values. These were used as the threshold values.

For integration, there were a lot of problems. When we tried to integrate, the camera input did not operate at all. It produced gibberish, which in turn could not get passed on through the rest of the system. Pinpointing the cause of this problem was a challenge and proved to be the hardest part of the project. In the end, after spending much time trying to figure out what went wrong, being that both of our individual parts work, we decided to just make a demo of our project.
Conclusions

By the end of the project, we learned a lot. Working separately on the different components made the project more specialized for each of us. We each understand a lot more about the different components and how they relate to each other. In the beginning, we underestimated the time consuming nature of the project. Toward the end when trying to fit everything together and integrate, we realized that more time should have been dedicated to testing and debugging the final product and not just the individual parts. We assumed that if our parts worked individually, that there was not going to be much needed when we integrate because only two signals were crossing boundaries. Our thinking on this was clearly skewed, and because of this experience we have learned more about design than anything – to allot plenty of time to debugging.
Appendix

Laura’s Code: (significant modules are in bold)

/////////////////////////////////////////////////////////////////////
// Pushbutton Debounce Module (video version)
/////////////////////////////////////////////////////////////////////
module debounce (reset, clock_65mhz, noisy, clean);
    input reset, clock_65mhz, noisy;
    output clean;
    reg [19:0] count;
    reg new, clean;
    always @(posedge clock_65mhz)
        if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
        else if (noisy != new) begin new <= noisy; count <= 0; end
        else if (count == 650000) clean <= new;
        else count <= count+1;
endmodule

////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_yrcrb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
// "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
// output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
/////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
/////////////////////////////////////////////////////////////////////////////
//
module lab5  (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock,
             vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
             vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
             vga_out_vsync,
             tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
             tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
             tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,
             tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
             tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
             tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
             tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,
ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,
ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,
clock_feedback_out, clock_feedback_in,
flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash sts, flash_byte_b,
rs232_txd, rs232_rxd, rs232_rts, rs232_cts,
mouse_clock, mouse_data, keyboard_clock, keyboard_data,
clock_27mhz, clock1, clock2,
disp_blank, disp_data_out, disp_clock, disp rs, disp_ce_b, disp_reset_b, disp_data_in,
button0, button1, button2, button3, button enter, button_right, button_left, button_down, button_up,
switch,
led,
user1, user2, user3, user4,
daughtercard,
systemace_data, systemace_address, systemace_ce_b, systemace_we_b, systemace_oe_b, systemace_irq,
systemace_mpbrdy,
analyzer1_data, analyzer1_clock,
analyzer2_data, analyzer2_clock,
analyzer3_data, analyzer3 clock,
analyzer4_data, analyzer4_clock);
output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input ac97_bit_clock, ac97_sdata_in;
output [7:0] vga out_red, vga out_green, vga out_blue;
output vga out sync b, vga out blank b, vga out pixel clock, 
vga out hsync, vga out vsync;
output [9:0] tv out ycrccb;
output tv out reset b, tv out clock, tv out i2c clock, tv out i2c data, 
tv out pal ntsc, tv out hsync b, tv out vsync b, tv out blank b, 
tv out subcar reset;
input [19:0] tv in ycrccb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output  [18:0] ram0_address;
output  ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output  [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output  [18:0] ram1_address;
output  ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output  [3:0] ram1_bwe_b;

input  clock_feedback_in;
output  clock_feedback_out;

inout  [15:0] flash_data;
output  [23:0] flash_address;
output  flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
in
input  flash_sts;

output rs232_txd, rs232_rts;
in  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
in  disp_data_in;
output  disp_data_out;

input  button0, button1, button2, button3, button_enter, button_right, button_left, button_down, button_up;
in  [7:0] switch;
output  [7:0] led;
inout  [31:0] user1, user2, user3, user4;

inout  [43:0] daughtercard;

inout  [15:0] systemace_data;
output  [6:0] systemace_address;
output  systemace_ce_b, systemace_we_b, systemace_oe_b;
in  systemace_irq, systemace_mpbrdy;

output  [15:0] analyzer1_data, analyzer2_data, analyzer3_data, analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;
// Audio Input and Output
assign beep = 1'b0;
assign audio_Reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign rami_data = 36'hZ;
assign rami_address = 19'h0;
assign rami_adv_ld = 1'b0;
assign rami_clk = 1'b0;
assign rami_cen_b = 1'b1;
assign rami_ce_b = 1'b1;
assign rami_oe_b = 1'b1;
assign rami_we_b = 1'b1;
assign rami_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
// assign disp_blank = 1'b1;
// assign disp_clock = 1'b0;
// assign disp_rs = 1'b0;
// assign disp_ce_b = 1'b1;
// assign disp_reset_b = 1'b0;
// assign disp_data_out = 1'b0;
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
// lab3 assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs
// Logic Analyzer
assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

/////////////////////////////////////////////////////////////////////
//
// lab5 : a simple pong game
//
/////////////////////////////////////////////////////////////////////

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
                 .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce db1(power_on_reset, clock_65mhz, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;

// UP and DOWN buttons for pong paddle
wire up,down;
debounce db2(reset, clock_65mhz, ~button_up, up);
debounce db3(reset, clock_65mhz, ~button_down, down);

// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0] vcount;
wire hsync,vsync,blank;
xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);
wire phsync,pvsync,pblank;

15
```verilog
wire [23:0] pixel;

//debug***************
wire bet_placed;
wire left;
debounce bb ( reset, clock_65mhz, ~button_left, left );
assign bet_placed = left;

//wire [6:0]  bet_amount;
//assign bet_amount = switch[6:0];

wire payout;
wire right;
debounce pb ( reset, clock_65mhz, ~button_right, right );
//assign payout = right;

wire [10:0] x_lever_in;
wire [9:0] y_lever_in;
assign x_lever_in = ( switch[7:0] == 0 ) ? 11'd0 : 
( (switch[7:0] == 1) ? 11'd510 :
( (switch[7:0] == 2) ? 11'd815 :
( (switch[7:0] == 3) ? 11'd791 : 11'd0 ) )
);
assign y_lever_in = ( switch[7:0] == 0 ) ? 10'd0 :
( (switch[7:0] == 1) ? 10'd650 :
( (switch[7:0] == 2) ? 10'd175 :
( (switch[7:0] == 3) ? 10'd610 : 10'd0 ) )
);


////////////////////////////////////////////////////
//debug
//display cash pot and bet amount
wire [6:0]  current_bet_amount;
wire [13:0] cash_pot;

wire [63:0] data;
display_16hex display16hex(reset,
clock_65mhz,
data,
disp_blank,
disp_clock,
disp_rs,
disp_ce_b,
disp_reset_b,
disp_data_out);
```
assign      data[7:0] = { 1'b0, current_bet_amount };  
assign      data[23:8] = { 2'b00, cash_pot };  
assign      data[63:24] = 40'd0;  
//debug************************

//DANEAYA
START************************************************************************
***************
//   wire [10:0] x_lever_in;
//   wire [9:0]  y_lever_in;
//assign      x_lever_in = ; //Daneaya x goes here
//assign      y_lever_in = ; //Daneaya y goes here

school---------------------------------------------

wire [10:0] x_out;
wire [9:0]  y_out;
wire [23:0] pixel_debug;

wire      up1, down1, left1, right1;
assign      up1 = up;
assign      down1 = down;
assign      left1 = left;
assign      right1 = right;

/* debug debug_1 ( vclock, reset, hcount, vcount, up1, down1, left1,  
right1, x_out, y_out, pixel_debug );
assign      x_lever_in = x_out;
assign      y_lever_in = y_out;;*/

*******

game_logic_FSM gl( clock_65mhz, reset, hcount, vcount, hsync, vsync,  
blank, phsync, pvsync, pblank,  
0, x_lever_in, y_lever_in,  
amount_won, cash_pot, current_bet_amount,  
state, pixel );

reg [23:0]  rgb_final;
reg         b,hs,vs;
always @ ( posedge clock_65mhz )
begin
  hs <= phsync;
  vs <= pvsync;
end
// VGA Output. In order to meet the setup and hold times of the // AD7125, we send it ~clock_65mhz.
assign vga_out_red = rgb_final[23:16];
assign vga_out_green = rgb_final[15:8];
assign vga_out_blue = rgb_final[7:0];
assign vga_out_sync_b = 1'b1; // not used
assign vga_out_blank_b = ~b;
assign vga_out_pixel_clock = ~clock_65mhz;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;
assign led = ~(3'b000,up,down,reset,switch[1:0]);
endmodule // lab5

module xvga(vclock,hcount,vcount,hsync,vsync,blank);

input vclock;
output [10:0] hcount;
output [9:0] vcount;
output vsync;
output hsync;
output blank;

reg hsync, vsync, hblank, vblank, blank;
reg [10:0] hcount; // pixel number on current line
reg [9:0] vcount; // line number

// horizontal: 1344 pixels total // display 1024 pixels per line
wire hsyncon, hsyncoff, hreset, hblankon;
assign hblankon = (hcount == 1023);
assign hsyncon = (hcount == 1047);
assign hsyncoff = (hcount == 1183);
assign hreset = (hcount == 1343);

// vertical: 806 lines total // display 768 lines
wire vsyncon, vsyncoff, vreset, vblankon;
assign vblankon = hreset & (vcount == 767);
assign vsyncon = hreset & (vcount == 776);
module led_view (vblank, hblank, vsync, hsync, x_out, y_out, pixel);

input [10:0] vcount;
input [9:0] hcount;
input vclock, reset;
output [10:0] x_out;
output [9:0] y_out;
output [23:0] pixel;
reg [10:0] x_paddle;
reg [9:0] y_paddle;

always @(posedge vclock)
begin
    if (reset)
    begin
        x_paddle <= 510;
        y_paddle <= 650;
    end
end
endmodule
else
    begin
      if ( ( hcount == 1023 ) && ( vcount == 767 ) )
        begin
          y_paddle <= y_paddle - 1;
        end
    end // else: !if( reset )
end // always @ ( posedge vclock )

/*
if ( 1 )//up )
y_paddle <= y_paddle - 1;
else
begin
  if ( down )
    y_paddle <= y_paddle + 1;
else
begin
  if ( left )
    x_paddle <= x_paddle - 1;
else
begin
  if ( right )
    x_paddle <= x_paddle + 1;
end
end // else: !if( down )
end // else: !if( up )
end // if ( ( hcount == 0 ) && ( vcount == 0 ) )
end // else: !if( reset )
end // always @ ( posedge vclock ) */

/*
assign x_out = x_paddle;
assign y_out = y_paddle;
wire [23:0] pixel_blob;

blob gosh ( x_paddle, y_paddle, hcount, vcount, pixel_blob );
assign      pixel = pixel_blob;
endmodule // debug

module blob( x, y, hcount, vcount, pixel );
  parameter WIDTH = 64;    // default width: 64 pixels
  parameter HEIGHT = 64;   // default height: 64 pixels
  parameter COLOR = 24'h80_80_00;  //olive
input [10:0] x,hcount;
input [9:0] y,vcount;
output [23:0] pixel;
reg [23:0]    pixel;
always @ ( x or y or hcount or vcount ) begin

else
    begin
      if ( ( hcount == 1023 ) && ( vcount == 767 ) )
        begin
          y_paddle <= y_paddle - 1;
        end
    end // else: !if( reset )
end // always @ ( posedge vclock )
if ( ( hcount >= x && hcount < ( x + WIDTH ) ) &&
( vcount >= y && vcount < ( y + HEIGHT ) ) )
    pixel = COLOR;
else pixel = 0;
end
endmodule
*/

/////////////////////////////////////////////////////////////////////////////
///
// DEBUG END
///
/////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////
///
// Game Logic FSM!!
///
/////////////////////////////////////////////////////////////////////////////

module game_logic_FSM ( vclock, reset, hcount, vcount, hsync, vsync, blank,
                        phsync, pvsync, pblank,
                        payout, x_lever_in, y_lever_in,
                        amount_won, cash_pot, current_bet_amount,
                        state, pixel );

input vclock;        // 65MHz clock
input reset;
input [10:0] hcount; // horizontal index of current pixel (0..1023)
input [9:0]  vcount; // vertical index of current pixel (0..767)
input        hsync;          // XVGA horizontal sync signal (active low)
input        vsync;          // XVGA vertical sync signal (active low)
input        blank;          // XVGA blanking (1 means output black pixel)
input        payout;
input [10:0] x_lever_in;
input [9:0]  y_lever_in;

output [11:0] amount_won;
output [13:0] cash_pot;
reg [13:0]    cash_pot;
output [6:0]  current_bet_amount;

output [23:0] pixel;

output        phsync;
output        pvsync;
output        pblank;

21
output [3:0] state;
reg [3:0] state;

reg reel1_go;
reg reel2_go;
reg reel3_go;

reg [3:0] rspeed1;
reg [3:0] rspeed2;
reg [3:0] rspeed3;

reg [18:0] start_addr1;
reg [18:0] start_addr2;
reg [18:0] start_addr3;

reg [4:0] rand1, rand2, rand3;

parameter REEL1_X = 11'd50;
parameter REEL2_X = 11'd295;
parameter REEL3_X = 11'd540;
parameter REELS_Y = 10'd240;

parameter REELS_WIDTH = 11'd220;
parameter REELS_HEIGHT = 10'd340;

parameter DORMANT = 4'b0000; //0
parameter LEVER_ACTIVATION_WAIT = 4'b0001; //1
parameter LEVER_PULL = 4'b0010; //2
parameter GET_RAND1 = 4'b0011; //3
parameter GET_RAND2 = 4'b0100; //4
parameter GET_RAND3 = 4'b0101; //5
parameter ALL_REELS_SPIN = 4'b0110; //6
parameter FIRST_REEL_STOPS = 4'b0111; //7
parameter SECOND_REEL_STOPS = 4'b1000; //8
parameter THIRD_REEL_STOPS = 4'b1001; //9
parameter GAME_OVER = 4'b1010; //A
parameter PAYOUT = 4'b1011; //B

parameter DIAMOND_1 = 19'd272_140;
parameter BLANK_2 = 19'd0;
parameter SEVEN_3 = 19'd18_040;
parameter BLANK_4 = 19'd36_300;
parameter CHERRIES_5 = 19'd54_340;
parameter BLANK_6 = 19'd72_600;
parameter DIAMOND_7 = 19'd90_640;
parameter BLANK_8 = 19'd108_900;
parameter CHERRIES_9 = 19'd126_940;
parameter BLANK_10 = 19'd145_200;
parameter  SEVEN_11 = 19'd163_240;
parameter  BLANK_12 = 19'd181_500;
parameter  SEVEN_13 = 19'd199_540;
parameter  BLANK_14 = 19'd217_800;
parameter  CHERRIES_15 = 19'd235_840;
parameter  BLANK_16 = 19'd254_100;

//////////////////////////////////////////lever display instance
wire [23:0] lever_pixel;
wire        hit, original;
wire        bet_placed;
wire [6:0]  bet_amount;
lever_display ld ( reset, vclock, x_lever_in, y_lever_in, hcount, vcount,
lever_pixel, bet_placed, hit, original, bet_amount );
assign      current_bet_amount = bet_amount;

//code for displaying title******************
wire [16:0] addr_title;
wire        out_of_bounds_title;
show_me_title smt ( vclock, reset, hcount, vcount, addr_title,
out_of_bounds_title );

wire [3:0]  dout_title;
romshowme romshowme_1 ( addr_title, vclock, dout_title );
wire [23:0] pixel_title;
get_rgb get_rgb_title ( dout_title, out_of_bounds_title, pixel_title );
//end code for displaying title

//code for displaying bet25 button************
wire [13:0] addr_bet25;
wire        out_of_bounds_bet25;
show_bet25 sb25 ( vclock, reset, hcount, vcount, addr_bet25,
out_of_bounds_bet25 );

wire [3:0]  dout_bet25;
rombet25 rombet25_1 ( addr_bet25, vclock, dout_bet25 );
wire [23:0] pixel_bet25;
get_rgb get_rgb_bet25 ( dout_bet25, out_of_bounds_bet25, pixel_bet25 );
//end code for displaying bet25 button

//code for displaying bet50 button************
wire [13:0] addr_bet50;
wire        out_of_bounds_bet50;
show_bet25 sb50 ( vclock, reset, hcount, vcount, addr_bet50,
out_of_bounds_bet50 );
defparam sb50.X_START = 11'd319;

wire [3:0]  dout_bet50;
rombet50 rombet50_1 ( addr_bet50, vclock, dout_bet50 );
wire [23:0] pixel_bet50;
get_rgb get_rgb_bet50 ( dout_bet50, out_of_bounds_bet50, pixel_bet50 );
//end code for displaying bet50 button

//code for displaying bet100 button*************
wire [13:0]   addr_bet100;
wire          out_of_bounds_bet100;
show_bet25 sb100 ( vclock, reset, hcount, vcount, addr_bet100,
out_of_bounds_bet100 );
defparam      sb100.X_START = 11'd504;

wire [3:0]    dout_bet100;
rombet100 rombet100_1 ( addr_bet100, vclock, dout_bet100 );
wire [23:0]   pixel_bet100;
ge_rgb get_rgb_bet100 ( dout_bet100, out_of_bounds_bet100, pixel_bet100 );
//end code for displaying bet100 button

wire [4:0]    rand_num_out;
ger_rand gr ( reset, vclock, x_lever_in, y_lever_in, rand_num_out );
wire [18:0]   stop_addr1, stop_addr2, stop_addr3;
wire [4:0]    rand1_wire, rand2_wire, rand3_wire;
assign        rand1_wire = rand1;
assign        rand2_wire = rand2;
assign        rand3_wire = rand3;

ger_stop_addr gs1 ( rand1_wire, stop_addr1 );
ger_stop_addr gs2 ( rand2_wire, stop_addr2 );
ger_stop_addr gs3 ( rand3_wire, stop_addr3 );

wire [11:0]   winnings;
ger_winnings gw ( bet_amount, stop_addr1, stop_addr2, stop_addr3, winnings );
wire          has_stopped1, has_stopped2, has_stopped3;

wire [23:0]   pixel_reel1, pixel_reel2, pixel_reel3;
wire          out_of_bounds_reel1, out_of_bounds_reel2, out_of_bounds_reel3;

wire [18:0]   addr, addr_reel1, addr_reel2, addr_reel3;

reel_animation reel_animation_reel1 ( vclock, reset, REEL1_X, REELS_Y,
REELS_WIDTH, REELS_HEIGHT, rspeed1, hcount, vcount, reel1_go,
start_addr1,
stop_addr1, addr_reel1, out_of_bounds_reel1,
has_stopped1 );
reel_animation reel_animation_reel2 ( vclock, reset, REEL2_X, REELS_Y, REELS_WIDTH, REELS_HEIGHT, rspeed2, hcount, vcount, reel2_go, start_addr2, stop_addr2, addr_reel2, out_of_bounds_reel2, has_stopped2 );

reel_animation reel_animation_reel3 ( vclock, reset, REEL3_X, REELS_Y, REELS_WIDTH, REELS_HEIGHT, rspeed3, hcount, vcount, reel3_go, start_addr3, stop_addr3, addr_reel3, out_of_bounds_reel3, has_stopped3 );

wire [3:0] dout;
assign addr = out_of_bounds_reel1 ? ( out_of_bounds_reel2 ? addr_reel3 : addr_reel2 ) : addr_reel1;

rom399300x4 rom399300x4_1 ( addr, vclock, dout);

get_rgb get_rgb_reel1 ( dout, out_of_bounds_reel1, pixel_reel1 );
get_rgb get_rgb_reel2 ( dout, out_of_bounds_reel2, pixel_reel2 );
get_rgb get_rgb_reel3 ( dout, out_of_bounds_reel3, pixel_reel3 );

assign phsync = hsync;
assign pvsync = vsync;
assign pblank = blank;
assign pixel = pixel_reel1 | pixel_reel2 | pixel_reel3 | lever_pixel | pixel_title | pixel_bet25 | pixel_bet50 | pixel_bet100;

always @ ( posedge vclock )
begin
  if ( reset )
  begin
    cash_pot <= 1000;
    reel1_go <= 1;
    reel2_go <= 1;
    reel3_go <= 1;
    rspeed1 <= 4’d0;
    rspeed2 <= 4’d0;
    rspeed3 <= 4’d0;
    start_addr1 <= BLANK_2;
    start_addr2 <= SEVEN_3;
  end
end
start_addr3 <= CHERRIES_5;
state <= DORMANT;
end
else
begin
case ( state )
DORMANT:
begin
if ( bet_placed )
begin
  cash_pot <= cash_pot - bet_amount;
  state <= LEVER_ACTIVATION_WAIT;
end
end
LEVER_ACTIVATION_WAIT:
begin
  //Lever FSM takes over
  state <= LEVER_PULL;
end
LEVER_PULL:
begin
if ( hit )
  state <= GET_RAND1;
end
GET_RAND1:
begin
  rand1 <= rand_num_out;  //=1
  state <= GET_RAND2;
end
GET_RAND2:
begin
  rand2 <= rand_num_out; //=2
  state <= GET_RAND3;
end
GET_RAND3:
begin
  rand3 <= rand_num_out; //=3
  state <= ALL_REELS_SPIN;
end
ALL_REELS_SPIN:
begin
if ( original )
begin
  state <= FIRST_REEL_STOPS;
end
else
begin
    rspeed1 <= 10;
    rspeed2 <= 10;
    rspeed3 <= 10;
    reel1_go <= 1;
    reel2_go <= 1;
    reel3_go <= 1;
end
end

FIRST_REEL_STOPS:
begin
    rspeed1 <= 5;
    reel1_go <= 0;
    if ( has_stopped1 )
        state <= SECOND_REEL_STOPS;
end

SECOND_REEL_STOPS:
begin
    rspeed2 <= 5;
    reel2_go <= 0;
    if ( has_stopped2 )
        state <= THIRD_REEL_STOPS;
end

THIRD_REEL_STOPS:
begin
    rspeed3 <= 5;
    reel3_go <= 0;
    if ( has_stopped3 )
        begin
            cash_pot <= cash_pot + winnings;
            if ( cash_pot < 25 )
                state <= GAME_OVER;
            else
                state <= DORMANT;
        end
end

GAME_OVER:
begin
    //you can't do anything except press reset cuz YOU LOST, lol!!
end

PAYOUT:
begin
    //you can't do anything except press reset, get rid of this . . .
end

default: //reset
begin

state <= DORMANT;
cash_pot <= 1000;
reel1_go <= 1;
reel2_go <= 1;
reel3_go <= 1;
rspeed1 <= 4'd0;
rspeed2 <= 4'd0;
rspeed3 <= 4'd0;

start_addr1 <= BLANK_2;
start_addr2 <= SEVEN_3;
start_addr3 <= CHERRIES_5;
end
case // case( state )
  default: // else: !if( reset )
end // always @ ( posedge vclock )
endmessage // game_logic_FSM

//////////////////////////////////////////////////////////////////////
// show_me_title: module to provide address of the ROM that holds the title
// ("Show Me SevenS") for display
//////////////////////////////////////////////////////////////////////
module show_me_title ( vclock, reset, hcount, vcount, addr, out_of_bounds );
  input vclock, reset;
  input [10:0] hcount;
  input [9:0] vcount;
  output [16:0] addr;
  output out_of_bounds;
  reg out_of_bounds;

  parameter X_START = 11'd0;
  parameter Y_START = 10'd45;
  parameter WIDTH = 11'd760;
  parameter HEIGHT = 10'd150;

  reg addr;

  always @ ( posedge vclock )
  begin
    if ( reset )
      addr <= 0;
    else
      begin
        if ( ( hcount == 0 ) && ( vcount == 0 ) )
          addr <= 0;
        else

begin
  if ( (( X_START <= hcount ) && ( hcount <= ( X_START +
  ( WIDTH - 1 ) ) ) )
    && ( ( Y_START <= vcount ) && ( vcount <= ( Y_START +
  ( HEIGHT - 1 ) ) ) ) )
    begin
      addr <= addr + 1;
      out_of_bounds <= 0;
    end
  else
    out_of_bounds <= 1;
end
endmodule // show_me_title

//////////////////////////////////////////////////////////////////////
//
// show_bet25: module to provide address for display of any of the bet
buttons (bet25, bet50, and bet100) since their ROMs are the same size
//
//////////////////////////////////////////////////////////////////////
module show_bet25 ( vclock, reset, hcount, vcount, addr, out_of_bounds );
input vclock, reset;
input [10:0] hcount;
input [9:0] vcount;
output [13:0] addr;
output        out_of_bounds;
reg           out_of_bounds;

parameter     X_START = 11'd135;
parameter     Y_START = 10'd620;
parameter     WIDTH = 11'd146;
parameter     HEIGHT = 10'd90;

reg [13:0]    addr;

always @ ( posedge vclock )
begin
  if ( reset )
    addr <= 0;
  else
    begin
      if ( ( hcount == 0 ) && ( vcount == 0 ) )
        addr <= 0;
      else
        begin
          if ( (( X_START <= hcount ) && ( hcount <= ( X_START +
          ( WIDTH - 1 ) ) ) )
            begin
              addr <= addr + 1;
              out_of_bounds <= 0;
            end
          else
            begin
              out_of_bounds <= 1;
            end
        end
    end
end
& & ( ( Y_START <= vcount ) & & ( vcount <= ( Y_START + 
( HEIGHT - 1 ) ) ) ) 

begin 
  addr <= addr + 1;
  out_of_bounds <= 0;
end 
else 
  out_of_bounds <= 1;
end 
end 
endmodule // bet25

/////////////////////////////////////////////////////////////////////
//
// reel_animation: module to control a reel's spin speed, start address, and
// stop address
//
/////////////////////////////////////////////////////////////////////
module reel_animation ( vclock, reset, x, y, width, height, rspeed, hcount, 
vcount, reel_go, start_addr, stop_addr, addr, out_of_bounds, has_stopped );

output has_stopped;
reg has_stopped;

input vclock, reset;
input [10:0] x, width;
input [9:0] y, height;
input [3:0] rspeed; //speed of reel in pixels/tick
input [10:0] hcount;
input [9:0] vcount;
input reel_go;
input [18:0] start_addr, stop_addr;
output [18:0] addr;

output out_of_bounds;
reg out_of_bounds;
reg [18:0] addr;

reg [18:0] addr_counter;
parameter OFFSET = 11'd1100;

always @ ( posedge vclock )
begin 
  if ( reset )
    begin
      addr <= start_addr;
      addr_counter <= start_addr;
      has_stopped <= 0;
    end 
end
begin
  if ( ( hcount == 0 ) && ( vcount == 0 ) ) //beginning of new frame
    begin
      if ( addr_counter >= 289_740 ) //loop back
        begin
          addr_counter <= addr_counter - 289_740; //make loop back look smooth
          addr <= addr_counter - 289_740;
          has_stopped <= 0;
          /*
           addr_counter <= 0;
           addr <= 0;/*
          end
        end
      else
        begin
          if ( ( reel_go == 0 ) &&
               ( ( stop_addr - OFFSET ) <= addr_counter ) &&
               addr_counter <= ( stop_addr + OFFSET ) )
            begin
              addr <= stop_addr;
              addr_counter <= stop_addr;
              has_stopped <= 1;
            end
        end
    end
  else
    begin
      if ( ( x <= hcount ) && ( hcount <= ( x + ( width - 1 ) )
               ) )
        && ( ( y <= vcount ) && ( vcount <= ( y + ( height - 1
count <= addr_counter + ( width * rspeed );
          addr_counter <= addr_counter + ( width * rspeed );
          has_stopped <= 0;
        end
      else
        begin
          if ( ( hcount == 0 ) && ( vcount == 0 ) )
            begin
              addr <= addr + 1;
              out_of_bounds <= 0;
            end
          else
            out_of_bounds <= 1;
        end // else: !if( ( hcount == 0 ) && ( vcount == 0 )
      end // else: !if( reset )
end // always @( posedge vclock )
endmodule // reel_animation
module lever_display ( reset, vclock, x, y, hcount, vcount, pixel,
    bet_placed, hit, original_pos, bet_amount );

    parameter WIDTH = 80;           // default width: 64 pixels
    parameter HEIGHT = 80;          // default height: 64 pixels

    input     reset, vclock;
    input [10:0] x;
    input [9:0] y;

    output       bet_placed;
    reg          bet_placed;
    output [6:0] bet_amount;
    reg [6:0]    bet_amount;

    input [10:0] hcount;
    input [9:0]  vcount;
    output       hit, original_pos;
    reg          hit;
    reg          original_pos;

    output [23:0] pixel;
    reg [10:0]   x_show;
    reg [9:0]    y_show;

    parameter   X_ORIGINAL = 11'd811;
    parameter   Y_ORIGINAL = 10'd170;
    parameter   X_THRESHOLD = 11'd780;
    parameter   Y_THRESHOLD = 10'd600;

    reg [2:0]    state;
    reg          lever_off;
    reg          user_off;
    reg          two_off, two_25_off, two_50_off, two_100_off;

    parameter   NO_BET_PLACED = 3'b000;
    parameter   ACTIVATION_WAIT = 3'b001;
    parameter   ACTIVATE = 3'b010;
    parameter   HIT = 3'b011;
    parameter   ANIMATE_TO_ORIGINAL_POS = 3'b100;
    parameter   AT_ORIGINAL_POS = 3'b101; //go back to NO_BET_PLACED

    /*wire [23:0]   pixel_lever;
     blub b ( x_show, y_show, hcount, vcount, lever_off, pixel_lever );
     defparam     b.WIDTH = WIDTH;
     defparam     b.HEIGHT = HEIGHT;*/
//Code for displaying lever ball
wire [12:0]   addr_lever;
wire          out_of_bounds_lever;
show_pointer lever_show ( vclock, reset, x_show, y_show, lever_off, hcount, vcount, addr_lever, out_of_bounds_lever );
defparam      lever_show.WIDTH = WIDTH;
defparam      lever_show.HEIGHT = HEIGHT;
wire [3:0]    dout_lever;
romleverball romleverball_1 ( addr_lever, vclock, dout_lever );
wire [23:0]   pixel_lever;
get_rgb get_rgb_lever ( dout_lever, out_of_bounds_lever, pixel_lever );
//end of code for displaying lever ball

//Code for displaying pointer ROM
wire [10:0]   x_offset;
wire [9:0]    y_offset; //user
assign        x_offset = x + 20; //user
assign        y_offset = y + 20; //user
wire [10:0]   addr_pointer;
wire          out_of_bounds_pointer;
show_pointer sp ( vclock, reset, x_offset, y_offset, user_off, hcount, vcount, addr_pointer, out_of_bounds_pointer );
wire [3:0]    dout_pointer;
rompointer rompointer_1 ( addr_pointer, vclock, dout_pointer );
wire [23:0]   user_input_pixel;
get_rgb get_rgb_pointer ( dout_pointer, out_of_bounds_pointer, user_input_pixel );
//end of code for displaying pointer ROM

wire [23:0]   pixel;
assign        pixel = ( pixel_lever | user_input_pixel );
wire          two_seconds;
hover_two_seconds hts ( reset, vclock, two_off, x, y, XORIGINAL, YORIGINAL, WIDTH, HEIGHT, two_seconds );

wire two_seconds_25;
wire two_seconds_50;
wire two_seconds_100;
parameter BUTTON_25_X = 11'd135;
parameter BUTTON_50_X = 11'd319;
parameter BUTTON_100_X = 11'd504;
parameter BUTTONS_Y = 10'd620;
parameter BUTTONS_WIDTH = 11'd146;
parameter BUTTONS_HEIGHT = 10'd90;
always @ ( posedge vclock )
begin
  if ( reset )
  begin
    lever_off <= 1;
    user_off <= 0;
    two_off <= 1;
    bet_placed <= 0;
    two_25_off <= 0;
    two_50_off <= 0;
    two_100_off <= 0;
    bet_amount <= 0;
    hit <= 0;
    original_pos <= 0;
    state <= NO_BET_PLACED;
  end
  else
  begin
    case ( state )
    NO_BET_PLACED:
    begin
      if ( two_seconds_25 )
      begin
        bet_placed <= 1;
        bet_amount <= 7'd25;
        state <= ACTIVATION_WAIT;
      end
    end
  end
end
end // else: !if( two_seconds_50 )
end // else: !if( two_seconds_25 )
end // case: NO_BET_PLACED

ACTIVATION_WAIT:
begin
  bet_placed <= 0;
  two_25_off <= 1;
  two_50_off <= 1;
  two_100_off <= 1;
  lever_off <= 0;
  user_off <= 0;
  two_off <= 0;
  x_show <= X_ORIGINAL;
  y_show <= Y_ORIGINAL;
  if ( two_seconds )
    state <= ACTIVATE;
end

ACTIVATE:
begin
  two_off <= 1;
  x_show <= x;
  y_show <= y;
  if ( ( x >= X_THRESHOLD ) && ( y > Y_THRESHOLD ) )
    begin
      hit <= 1;
      state <= HIT;
    end
  else
    if ( ( ( x < X_THRESHOLD ) || ( x > ( 1023 - WIDTH ) ) ) //x boundaries
        || y < ( Y_ORIGINAL - 60 )
    )
      state <= ACTIVATION_WAIT;
  end

HIT:
begin
  user_off <= 1;
  state <= ANIMATE_TO_ORIGINAL_POS;
end

ANIMATE_TO_ORIGINAL_POS:
begin
  if ( y_show <= Y_ORIGINAL )
    begin
      original_pos <= 1;
      state <= AT_ORIGINAL_POS;
    end
else
    if (( hcount == 0 ) && ( vcount == 0 ))
        y_show <= y_show - 2;
end

AT ORIGINAL_POS:
    begin
        hit <= 0;
        original_pos <= 0;
        lever_off <= 1;
        state <= NO_BET_PLACED;
    end

default:
    begin
        state <= NO_BET_PLACED;
    end

endcase // case( state )
end // else: !if( reset)
end
endmodule // lever_display

module show_pointer ( vclock, reset, x, y, user_off, hcount, vcount, addr,
out_of_bounds );
    input [10:0] x;
    input [9:0] y;
    input vclock, reset, user_off;
    input [10:0] hcount;
    input [9:0] vcount;
    //output [10:0] addr;
    output [12:0] addr; //to accomodate the romleverball
    output out_of_bounds;
    reg out_of_bounds;

    parameter WIDTH = 11'd40;
    parameter HEIGHT = 10'd40;

    reg [12:0] addr;

    always @ ( posedge vclock )
        begin
            if ( reset )
                addr <= 0;
            else
                if ( user_off )
addr <= 0; // I know it's black, probably could have used out_of_bounds instead
else
begin
  if ( ( hcount == 0 ) && ( vcount == 0 ) )
    addr <= 0;
  else
  begin
    if ( ( ( x <= hcount ) && ( hcount <= ( x + ( WIDTH - 1 ) ) ) )
      && ( ( y <= vcount ) && ( vcount <= ( y + ( HEIGHT - 1 ) ) ) ) )
      begin
        addr <= addr + 1;
        out_of_bounds <= 0;
      end
      else
      begin
        out_of_bounds <= 1;
      end
  end // else: !if( user_off )
end // always @ ( posedge vclock )
endmodule // show_pointer

//////////////////////////////////////////////////////////////////////
// hover_two_seconds: tells you if user_x and user_y have been within rectangle (x_upper, y_upper, WIDTH, HEIGHT)
// (no longer two seconds, does it instantly, I didn't change the name)
//////////////////////////////////////////////////////////////////////
module hover_two_seconds ( reset, vclock, off, user_x, user_y, x_upper, y_upper, width, height, two_seconds );
input reset, vclock, off;
input [10:0] user_x, x_upper, width;
inport [9:0] user_y, y_upper, height;
output two_seconds;
reg two_seconds;

reg [25:0] counter;
reg [1:0] seconds;

always @ ( posedge vclock )
begin
  if ( reset )
    two_seconds <= 0;
  else
  begin
    if ( off )
      two_seconds <= 0;
    else
      begin

37
begin
  if ( ( x_upper <= user_x ) && ( user_x < ( x_upper +
width ) ) ) &&
( ( y_upper <= user_y ) && ( user_y < ( y_upper +
height ) ) )
  
two_seconds <= 1;
  else
  
two_seconds <= 0;
end // else: !if( off )
end // else: !if( reset )
end // always @ ( posedge vclock )
endmodule // hover_two_seconds

/////////////////////////////////////////////////////////////////////
//
// hover_two_seconds: tells you if user_x and user_y have been within rectangle (x_upper, y_upper, WIDTH, HEIGHT) for two seconds continuously
//
/////////////////////////////////////////////////////////////////////
/*module hover_two_seconds ( reset, vclock, off, user_x, user_y, x_upper,
y_upper, width, height, two_seconds );
  input reset, vclock, off;
  input [10:0] user_x, x_upper, width;
  input [9:0]  user_y, y_upper, height;
  output      two_seconds;
  reg         two_seconds;
  reg [25:0]  counter;
  reg [1:0]   seconds;
always @ ( posedge vclock )
begin
  if ( reset )
  begin
    counter <= 0;
    seconds <= 0;
    two_seconds <= 0;
  end
  else
  begin
    if ( off )
    begin
      counter <= 0;
      seconds <= 0;
      two_seconds <= 0;
    
end
end
endmodule // hover_two_seconds
end
else
begin
  if ( seconds == 2 )
  begin
    two_seconds <= 1;
    seconds <= 0;
  end
else
begin
  if ( counter == 65000000 )
  begin
    counter <= 0;
    seconds <= seconds + 1;
    two_seconds <= 0;
  end
else
  begin
    if ( ( x_upper <= user_x ) && ( user_x < ( x_upper + width ) ) ) &&
       ( y_upper <= user_y ) && ( user_y < ( y_upper + height ) )
    begin
      counter <= counter + 1;
      two_seconds <= 0;
    end
else
  begin
    counter <= 0;
    seconds <= 0;
    two_seconds <= 0;
  end
end // else: !if( counter == 65000000 )
end
end // else: !if( reset )
end // always @( posedge vclock )
endmodule // hover_two_seconds
*/

////////////////////////////////////////////////////////////////////////////////////////
///
/// Get rgb values combinationally
///
////////////////////////////////////////////////////////////////////////////////////////
///
module get_rgb ( dout, out_of_bounds, pixel );
  input[3:0] dout;
input out_of_bounds;
output [23:0] pixel;
reg [23:0] pixel;

always @ (dout or out_of_bounds)
begin
    if (out_of_bounds)
        pixel = 24'h00_00_00; //black
    else
        begin
            case (dout) //color from ROM
                4'h0: pixel = 24'h00_00_00; //black
                4'h1: pixel = 24'h80_00_80; //purple
                4'h2: pixel = 24'h00_80_00; //green
                4'h3: pixel = 24'h80_80_00; //olive
                4'h4: pixel = 24'hFF_FF_00; //yellow
                4'h5: pixel = 24'h00_00_80; //maroon
                4'h6: pixel = 24'h00_80_80; //teal
                4'h7: pixel = 24'h80_80_80; //gray
                4'h8: pixel = 24'h00_C0_C0; //silver
                4'h9: pixel = 24'hFF_00_00; //red
                4'hA: pixel = 24'h00_00_80; //navy
                4'hB: pixel = 24'h00_FF_00; //lime
                4'hC: pixel = 24'h00_00_FF; //blue
                4'hD: pixel = 24'h00_FF_FF; //fuchsia
                4'hE: pixel = 24'hFF_00_FF; //aqua
                4'hF: pixel = 24'hFF_FF_FF; //white
                default: pixel = 24'h00_00_00; //black
            endcase // case( color from Rom )
        end // else: !if( out_of_bounds )
    end // always @ (dout or out_of_bounds)
endmodule // get_rgb

/////////////////////////////////////////////////////////////////////
//
// get_stop_addr: module that takes in 1 random number and provides its
// stop_addr combinationally (virtual reel mapping)
//
/////////////////////////////////////////////////////////////////////
module get_stop_addr (rand_num, stop_addr);
input [4:0] rand_num;
output [18:0] stop_addr;
reg [18:0] stop_addr;

parameter DIAMOND_1 = 19'd272_140;
parameter BLANK_2 = 19'd0;
parameter SEVEN_3 = 19'd18_040;
parameter BLANK_4 = 19'd36_300;
parameter CHERRIES_5 = 19'd54_340;
parameter BLANK_6 = 19'd72_600;
parameter DIAMOND_7 = 19'd90_640;
parameter     BLANK_8 = 19'd108_900;
parameter     CHERRIES_9 = 19'd126_940;
parameter     BLANK_10 = 19'd145_200;
parameter     SEVEN_11 = 19'd163_240;
parameter     BLANK_12 = 19'd181_500;
parameter     SEVEN_13 = 19'd199_540;
parameter     BLANK_14 = 19'd217_800;
parameter     CHERRIES_15 = 19'd235_840;
parameter     BLANK_16 = 19'd254_100;

always @ ( rand_num )
begin
  case ( rand_num )
    5'd1:
      begin
        stop_addr = SEVEN_3;
      end
    5'd2:
      begin
        stop_addr = SEVEN_11;
      end
    5'd3:
      begin
        stop_addr = SEVEN_13;
      end
    5'd4:
      begin
        stop_addr = DIAMOND_7;
      end
    5'd5:
      begin
        stop_addr = DIAMOND_1;
      end
    5'd6:
      begin
        stop_addr = DIAMOND_1;
      end
    5'd7:
      begin
        stop_addr = DIAMOND_1;
      end
    5'd8:
      begin
        stop_addr = CHERRIES_9;
      end
    5'd9:
      begin
        stop_addr = CHERRIES_5;
      end
    5'd10:
      begin
        stop_addr = BLANK_6;
      end
  endcase
end
5'd11:
  begin
    stop_addr = BLANK_4;
  end
5'd12:
  begin
    stop_addr = BLANK_4;
  end
5'd13:
  begin
    stop_addr = BLANK_4;
  end
5'd14:
  begin
    stop_addr = BLANK_4;
  end
5'd15:
  begin
    stop_addr = BLANK_2;
  end
5'd16:
  begin
    stop_addr = BLANK_2;
  end
5'd17:
  begin
    stop_addr = BLANK_2;
  end
5'd18:
  begin
    stop_addr = BLANK_2;
  end
5'd19:
  begin
    stop_addr = BLANK_16;
  end
5'd20:
  begin
    stop_addr = BLANK_16;
  end
5'd21:
  begin
    stop_addr = BLANK_14;
  end
5'd22:
  begin
    stop_addr = BLANK_14;
  end
5'd23:
  begin
    stop_addr = BLANK_14;
  end
5'd24:
  begin
```plaintext
stop_addr = BLANK_12;
end
5'd25:
  begin
    stop_addr = BLANK_12;
  end
5'd26:
  begin
    stop_addr = BLANK_12;
  end
5'd27:
  begin
    stop_addr = BLANK_12;
  end
5'd28:
  begin
    stop_addr = BLANK_12;
  end
5'd29:
  begin
    stop_addr = BLANK_12;
  end
5'd30:
  begin
    stop_addr = BLANK_12;
  end
5'd31:
  begin
    stop_addr = BLANK_10;
  end
5'd0:
  begin
    stop_addr = BLANK_10;
  end
default:
  begin
    stop_addr = CHERRIES_15;
  end
endcase // case( rand_num )

end // always @ ( rand_num )
endmodule // get_stop_addr

/////////////////////////////////////////////////////////////////////
//
// get_winnings: module that takes in 3 stop_addrs and calculates winnings
// (pay table) combinationally
//
/////////////////////////////////////////////////////////////////////
module get_winnings ( bet_amount, stop_addr1, stop_addr2, stop_addr3,
  winnings );
```
input [6:0] bet_amount;
input [18:0] stop_addr1, stop_addr2, stop_addr3;
output [11:0] winnings;
reg [11:0]    winnings;

parameter     DIAMOND_1 = 19'd272_140;
parameter     BLANK_2 = 19'd0;
parameter     SEVEN_3 = 19'd18_040;
parameter     BLANK_4 = 19'd36_300;
parameter     CHERRIES_5 = 19'd54_340;
parameter     BLANK_6 = 19'd72_600;
parameter     DIAMOND_7 = 19'd90_640;
parameter     BLANK_8 = 19'd108_900;
parameter     CHERRIES_9 = 19'd126_940;
parameter     BLANK_10 = 19'd145_200;
parameter     SEVEN_11 = 19'd163_240;
parameter     BLANK_12 = 19'd181_500;
parameter     SEVEN_13 = 19'd199_540;
parameter     BLANK_14 = 19'd217_800;
parameter     CHERRIES_15 = 19'd235_840;
parameter     BLANK_16 = 19'd254_100;

parameter     ANY_CHERRIES_25 = 12'd25;
parameter     ANY_CHERRIES_50 = 12'd50;
parameter     ANY_CHERRIES_100 = 12'd75;

parameter     ANY_2_CHERRIES_25 = 12'd50;
parameter     ANY_2_CHERRIES_50 = 12'd100;
parameter     ANY_2_CHERRIES_100 = 12'd150;

parameter     THREE_CHERRIES_25 = 12'd100;
parameter     THREE_CHERRIES_50 = 12'd200;
parameter     THREE_CHERRIES_100 = 12'd300;

parameter     THREE_DIAMONDS_25 = 12'd200;
parameter     THREE_DIAMONDS_50 = 12'd400;
parameter     THREE_DIAMONDS_100 = 12'd600;

parameter     THREE_SEVENS_25 = 12'd500;
parameter     THREE_SEVENS_50 = 12'd1000;
parameter     THREE_SEVENS_100 = 12'd1500;

always @ ( stop_addr3 )                   //since stop_addr3, corresponding to reel3, is the
                                          //last one to be figured out
      begin
        //Three Cherries
        if ( ( stop_addr1 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) ) &&
             ( stop_addr2 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) ) &&
             ( stop_addr3 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
             )
        winnings = ( bet_amount == 7'd25 ) ? THREE_CHERRIES_25 :
( bet_amount == 7'd50 ) ? THREE_CHERRIES_50 :

THREE_CHERRIES_100 );
    else
        //Any Two Cherries
        if ( ( stop_addr1 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
            &&
            ( stop_addr2 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
        ) //stop_addr1 && stop_addr2 are cherries
            ||
            ( stop_addr2 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
        &&
        ( stop_addr3 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
        ) //stop_addr2 && stop_addr3 are cherries
            ||
            ( stop_addr1 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 )
        &&
        ( stop_addr3 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
        ) //stop_addr1 && stop_addr3 are cherries
        winnings = ( bet_amount == 7'd25 ) ? ANY_2_CHERRIES_25 :
            ( ( bet_amount == 7'd50 ) ? ANY_2_CHERRIES_50 :

ANY_2_CHERRIES_100 );
    else
        //Any Cherry
        if ( ( stop_addr1 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
            ||
            ( stop_addr2 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
            ||
            ( stop_addr3 == ( CHERRIES_5 || CHERRIES_9 || CHERRIES_15 ) )
        ) winnings = ( bet_amount == 7'd25 ) ? ANY_CHERRIES_25 :
            ( ( bet_amount == 7'd50 ) ? ANY_CHERRIES_50 :

ANY_CHERRIES_100 );
    else
        //Three Diamonds
        if ( ( stop_addr1 == ( DIAMOND_1 || DIAMOND_7 ) )
            &&
            ( stop_addr2 == ( DIAMOND_1 || DIAMOND_7 ) )
            &&
            ( stop_addr3 == ( DIAMOND_1 || DIAMOND_7 ) )
        )
        winnings = ( bet_amount == 7'd25 ) ? THREE_DIAMONDS_25 :
            ( ( bet_amount == 7'd50 ) ? THREE_DIAMONDS_50 :

THREE_DIAMONDS_100 );
    else
        //Three 7's
        if ( ( stop_addr1 == ( SEVEN_3 || SEVEN_11 || SEVEN_13 ) )
            &&
            ( stop_addr2 == ( SEVEN_3 || SEVEN_11 || SEVEN_13 ) )
            &&
            ( stop_addr3 == ( SEVEN_3 || SEVEN_11 || SEVEN_13 ) )
        )
        winnings = ( bet_amount == 7'd25 ) ? THREE_SEVENS_25 :
34
( ( bet_amount == 7'd50 ) ? THREE_SEVENS_50 )


else
  winnings = 0;
end // always @ ( stop_addr3 )
endmodule // get_winnings


//////////////////////////////////////////////////////////////////////
//
// get_rand: module that provides a pseudo-random number every vclock cycle
//
//////////////////////////////////////////////////////////////////////
module get_rand ( reset, vclock, x, y, rand_num );
  input reset, vclock;
  input [10:0] x;
  input [9:0] y;
  output [4:0] rand_num;
  reg [4:0] rand_num;
  reg [11:0] count;

  always @ ( posedge vclock )
  begin
    if ( reset )
      begin
        rand_num <= 0;
        count <= 20;
      end
    else
      begin
        count <= count + 1;
        rand_num <= x + count + y;
      end
  end
endmodule // get_rand