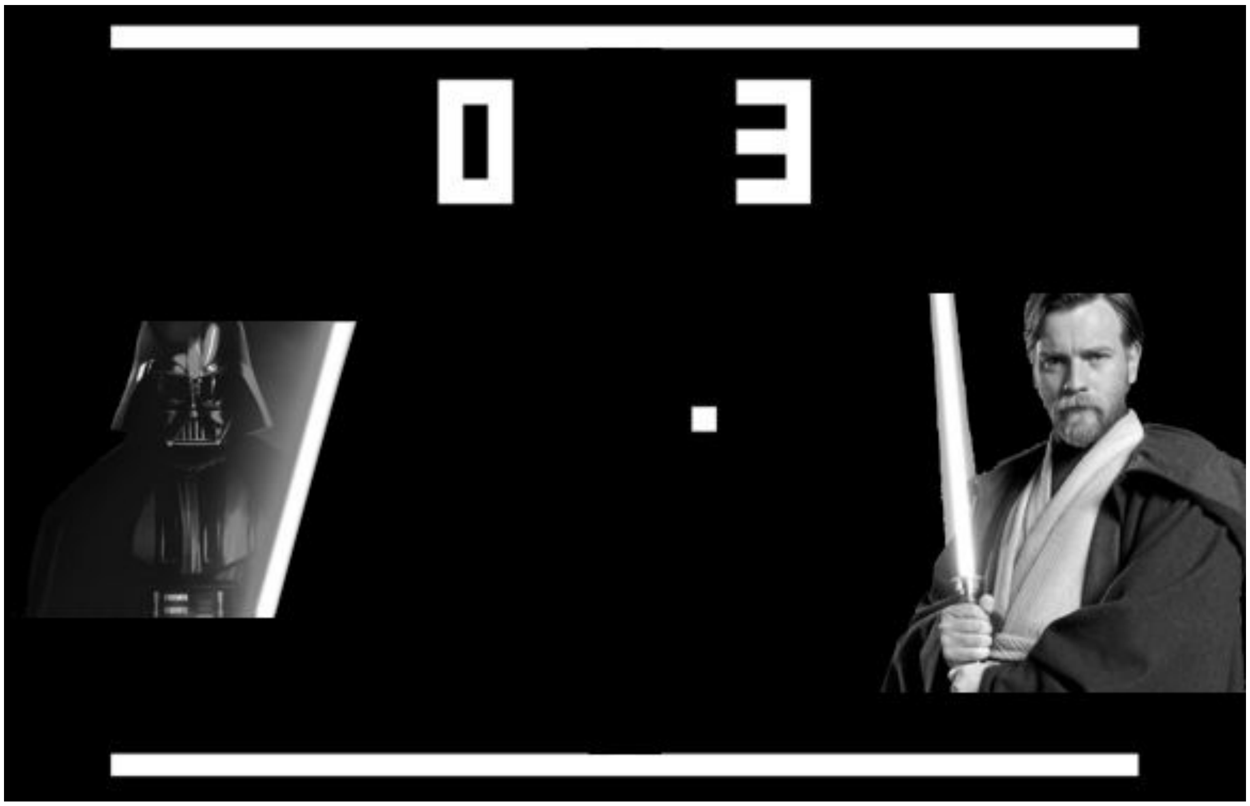


Live Action Pong

12.13.2017

Mike Wang

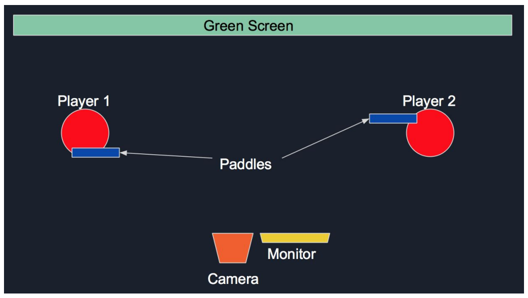
Nicholas Waltman

6.111 Fall 2017

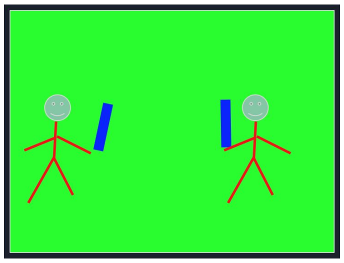
Abstract

We plan to program a pong or tennis-style game that incorporates both the physical and virtual world. Players will hold a paddle (tube of coloured construction paper) and stand at opposite ends of a green screen. A camera looking at the screen and players will pick up the position of the paddles via the magic of chroma-keying, and using that information control the location of a virtual paddle which will interact with a virtual ball. The gameplay is similar to pong, where the players hit the ball back and forth and points are lost if the ball hits the wall behind you. Since the paddles being controlled by a physical object allows for the paddle to be tilted at arbitrary angles, we plan to implement simple physics with respect to ball collision with angled surfaces to spice up gameplay. Leveraging further the power of chroma-keying we will also replace the green screen with a custom image - allowing pong to be played in a locale of your choosing.

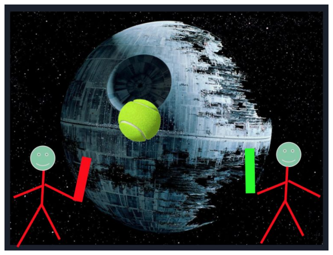
Some stretch goals include more advanced physics (having the paddle's velocity affect the velocity of the ball, for instance), allowing the players to choose between different paddles for different interactions with the ball, and a background that responds to game events.



Play area setup



Camera view



Monitor display

Goals

Minimum Product

- ✓ Isolate players from green screen (Mike)
    - ✓ Get RGB values from NTSC camera
    - ✓ Achieve RGB to HSV conversion
    - ✓ Create interface to debug and set thresholds for chroma keying
    - ✓ Ensure proper pipelining
- ✓ Locate paddles (Mike)
- ✓ Balls returned as if the paddle was flat (Nick)

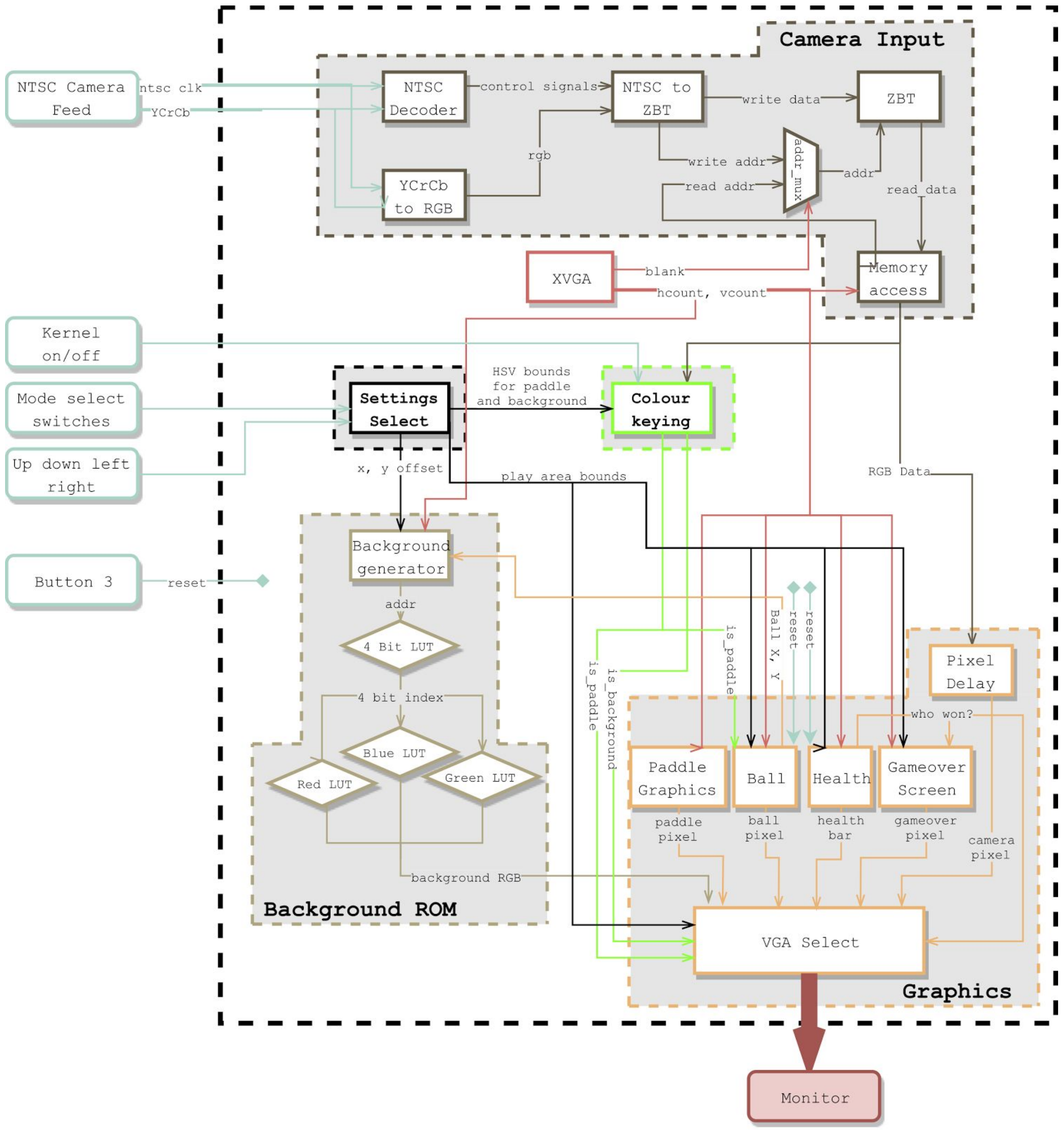
Expected Functionality

- ✓ Paddle angle affects ball direction of motion (Nick)
- ✓ Render paddle at specified angle (Mike)
    - ✓ Create graphics for paddles
- ✓ Display game on monitor (Mike)
    - ✓ Ball, players, custom background

Stretch Features

- ❏ Ball speed affected by paddle speed (Nick)
- ✓ Background reacts to game (Mike and Nick)
    - ✓ Background responds to in-game events (Mike)
    - ✓ Displays game-over state and who won (Mike)
    - ✓ Displays player health (Nick)
- ❏ Paddle selection affects ball interaction (Nick)

System Block Diagram



Everything other than NTSC decoder and YCrCb2RGB runs at the system clock, these two modules run at the camera clock so the control signals and RGB output are in sync.

Subsystems

I. Camera input (Mike)

Modules: *zbt_6111, vram_display, adv7185init, ntsc_decode*, YCrCb2RGB, ntsc_to_zbt (provided by staff)*

This subsystem takes the raw NTSC output and converts it into a usable form via the **adv7185** module, then parses out the control signals with **ntsc_decode**. Simultaneously it converts the YCrCb data from the camera to RGB using the **YCrCb2RGB** module, then the control signals corresponding to the right RGB values are fed into **ntsc_to_zbt** for storage into the ZBT memory (**zbt_6111**). The pixels needed by the VGA display are retrieved via the **vram_display** module.

This section relies heavily on staff-provided Verilog (otherwise it would've been a almost a final project in and of itself) , however significant changes needed to be made to **ntsc_to_zbt** since the original only stored black and white data, and this project required colour. In summary, the changes were to make this module store 2 pixels worth of data (18 bits each) per address of ZBT memory (6 bits of R, G, and B), instead of 4 pixels worth of data per address, since black and white data only required 8 bits of information per pixel. We still had to double up on addresses since at 2^19 addresses the ZBT memory was not large enough to afford storing only one pixel's worth of data per address.

In addition to the following changes, one must obviously change the size of wires and registers as appropriate (line numbers in reference to **ntsc_to_zbt.v**).

a) We need not change anything before line 86, since this section only deals with generating the appropriate control signals which are not be affected by whether we are storing 4 or 2 pixels of data per address.
b) Only concatenate 2 pixels of data into the register that holds the data to store into ZBT memory.
c) In the ZBT address computation section, remove the leading 0 and expand the section pulling from x_addr to [9:1] from [9:2], since we're saving every 2 addresses instead of every 4.
d) Update the write enable logic to reflect writing every 2 addresses instead of every 4.

vram_display needed to be changed to read a new address every 2 pixels instead of every 4 as well.

Changes were also needed to mirror (or de-mirror, depending on your perspective) the stored data. Specifically, this change was decrementing the col counter instead of incrementing it (line 79), making the data write from right to left instead of left to right. COL_START also had to be changed since it indicates the maximum X coordinate of the image after the change, instead of the minimum X before the change. Therefore it needs to be increased to something like 800, otherwise the left half of the image would be cut off.

The address signal for the **vram_display** module needs to be muxed since we must use this port for both read and write addresses. This feature (also provided by staff) exists in the labkit module where vram_addr is assigned. Cleverly, read requests are interleaved between write requests using the write enable signal.

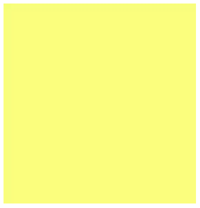
II. Colour Keying (Mike)

Modules: binarizer, *rgb2hsv*

The **binarizer** module performs the colour keying and is the foundation of the whole project. The module is set up to be able to detect two different colours - the green screen and the paddles - using the HSV boundary information from the settings select subsystem. Pixels that have HSV values within the range for the background, or paddle, are classified accordingly, allowing this module to output the status of one pixel (i.e. if it is a background or paddle pixel, or neither).

Why HSV instead RGB?

HSV ranges were used for colour keying, as opposed to RGB ranges, since the characteristics of a colour are more intuitive when understood through its hue, value, and saturation as opposed to its red, green, and blue components. Hue can be understood as the colour of a pixel, value as how dark it is, and saturation as its intensity (high saturation meaning the colour is very apparent, low meaning that it is washed out). This information is easy to ballpark by eye, which is important for figuring out how to adjust these thresholds so that the right colours are picked up, whereas it is much harder to determine a pixel's RGB components just by looking.



For example, what are the RGB components of this colour? You probably didn't guess (251, 253, 124). However, with HSV you can reason that this is close to yellow, and so look up the hue range for yellow. It looks somewhat washed out, so the saturation is probably somewhere around half, and it's fairly bright so the value is likely very high. As it turns out, looking at the HSV diagram on the left this is exactly where the colour is, in the red square in *Figure 1*.

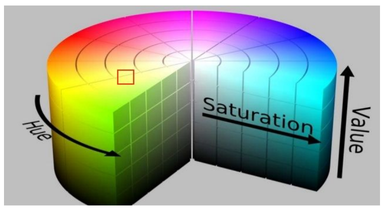


Figure 1: HSV colour space

Now let's say we're trying to adjust the thresholds to treat blue as the background as opposed to green, in the scene shown in *Figure 2a.* Using HSV thresholds allows us to do this by simply changing the hue boundaries without worrying about the saturation and brightness as shown in *Figure 2c, 2d*. The **settings select** subsystem lets us do this in real time!

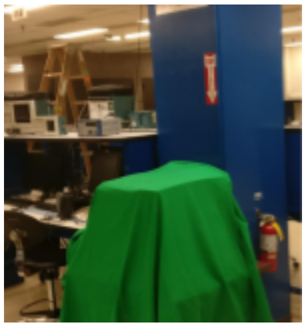


Figure 2a: The scene

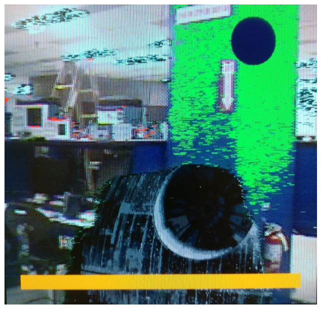


Figure 2b: Replacing green pixels with the background image. The bright green pixels are the result of the program replacing blue pixels with the paddle colour.

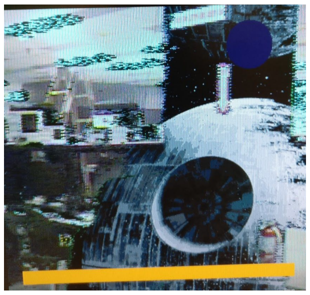


Figure 2c: Increasing the hue upper bound now includes the blue parts of the image, so the green and blue areas are replaced by the background image. Blue pixels no longer treated as paddle pixels since classification as background takes priority.

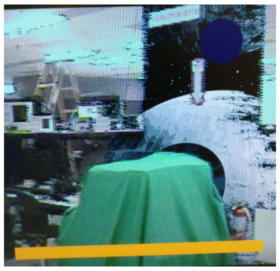


Figure 2d: Increasing the hue lower bound now completely excludes the green from being detected, and now only blue pixels are replaced with the background image.

In a similar vein, if we had the value at the right range but weren't picking up dark areas caused by wrinkles, it's a simple matter of decreasing the lower bound of the value parameter. The same adjustment would be much more difficult in the RGB color space.

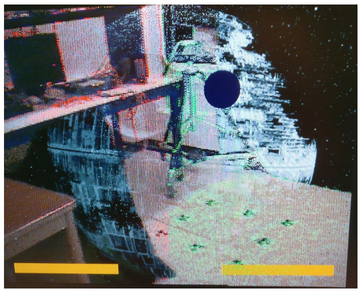
How does it work?

The module first sends the data to the **rgb2hsv** module provided by staff which, as its name suggests, converts the RGB data from the ZBT memory into HSV data. This module requires 2x16 bit divider IP cores. This module is pipelined so it will continuously shift in new data every clock cycle, and adds a 23 cycle delay to the whole pipeline. We then check if the pixel falls within the HSV boundaries of the background, and if it does, shift a 1 into the 5 bit register holding the initial binarization of the background pixels, and shift 0 into the corresponding register for paddle pixels (a pixel is either paddle or background, never both, with background taking precedence). Else check if it fits the HSV of a paddle, and if it does shift 1 into the paddle register instead and 0 into the background register.

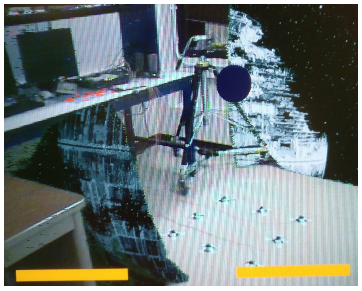
Now that we have the initial binarization results we can perform erosion. The next section of code shifts a 1 into the erosion output buffer only if all of the bits in the initial binarization buffer are 1, else shifts in a 0. This is done for both the paddle and background pixels.

Given the output of the erosion kernel, apply dilation by outputting a 1 if the erosion output is >0, else output 0. This is equivalent to classifying neighbouring pixels as 1 if any nearby pixel is 1. Again, this is applied to the erosion kernel outputs for both the paddle and background pixels. This yields the final classification of the pixel as a background or paddle.

You may notice that, instead of a 2 dimensional kernel, this process essentially applies a 1 dimensional kernel by only checking the pixels in the same line. I found that this kernel was quite effective at removing noise, and was much simpler to implement than a 2 dimensional kernel, so there was no need to try a 2D kernel. Check out the image with and without applying the kernel:

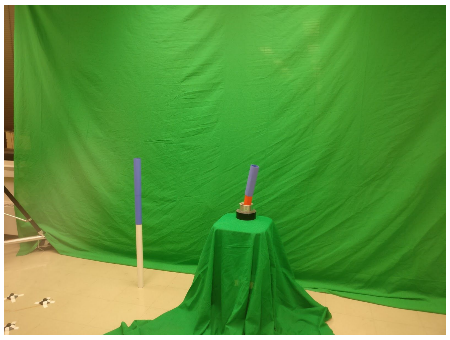


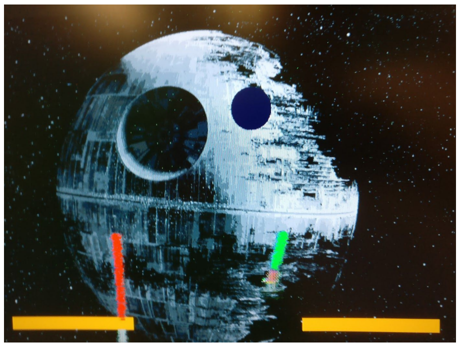
No Kernel Applied: appears to classify parts of the lab bench and floor as paddle pixels (the red and green areas). Seems to think part of the lab bench should be showing the death star too.



Kernal On: Vast majority of the artifacts removed (as if millions of pixels suddenly cried out in terror, and were suddenly silenced)

This module is also quite adept at identifying 2 different colours - the background and the paddle (green and blue respectively) - at the same time. As shown:





One thing that I neglected to implement was accounting for the circular nature of the hue parameter. The logic checks for hueLowerBound < pixelHue < hueUpperBound, which is impossible to satisfy if the lowerBound is higher than the upperBound but is actually valid because hue is a circular range. The concrete manifestation of this issue is that it's almost impossible to key to colours close to red, since red straddles the 0/360 degree discontinuity.

III. Settings Select (Mike)

Modules: parameter_select, incrementor, *debounce*

This module takes the switches as input and uses the combination to change which parameter is being adjusted by the directional buttons. Some of these parameters, such as the x and y bounds of the game area, are fairly large and it would be impractical to press a button a few hundred times to get the parameter to where you want it. Yet we can't increment those settings every rising edge of the clock else it'd be impossible to control. Thus the **incrementor** module was born. This module takes in a button as input, and has an output that rises high at most a limited number of times per second. The registers that hold the various parameters are then controlled by this output (which register specifically is determined by the switches), so one can hold down the up button and have the parameter increment at a fast, but controllable, rate. The switches also control the output connected to the hex displays, so that the user can see the values of the parameter they are currently selecting.

This module plays an important role in getting accurate colour keying, since it lets us adjust the HSV boundaries for background and pixel detection on the go without having to constantly reprogram the labkit (as we saw in the changing hue example in the previous section). Similarly, we can adjust the game boundaries and the positioning of the background image without being cornered into using hardcoded values, and the game boundary information is what allows many of the graphics generating modules to dynamically adjust their position.

See the real-time rekeying and resizing in action here:

https://drive.google.com/open?id=1TkwXHgcmDBtb-M3THLptw1auiFVwdjKW

https://drive.google.com/open?id=12dFehfPY92KX1ufqfOXDYepuQNAh-487

IV. Background ROM (Mike)

Modules: background_gen

This module implements ROM that contains a 4 bit index for every pixel in the preloaded background image, and ROMs to look up the red, green, and blue values for all 16 possible indices. It also contains logic to "bounce" the background following the ball's motion, by translating the address being looked up by an amount proportional to the ball's current position.

The 4 bit version of the background image was generated using Gimp on the lab computers, and the .coe files were created using the staff provided Matlab. The background image was 1024x512 pixels, which synthesizes fairly quickly but takes up quite a bit of BRAM memory.

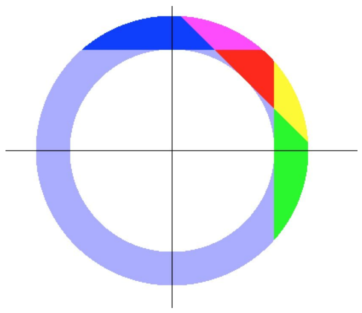
V. Physics Simulation - Collision Detection and Angle Approximation (Nick)

Modules: vga_select, ball

Ball Module

In order to have greater player control over the return of the ball, the approximate angle of the strike was estimated. This was done by dividing the edge of the ball into several segments. Once a frame, pixels were checked to see if they were both inside one of the segments and part of a player's paddle. If enough pixels overlapped, we were able to determine that there was a collision between the paddle and the ball, and we're not just picking up on noise. When this occurred, the number of pixels in each segment were compared to determine which segment had the greatest number of overlapping pixels.

Using eight different overlapping segments, we had 45º regions, so that discounting noise error would be no higher than ± 22.5º, and the average angle error would be 11.25º. Verilog has no built-in function to find the maximum of an array, so this had to be done by hand. Using continuous combinatorial logic was discovered to be necessary, so that on each frame transition the logic could be complete without getting in the way of the pipeline, preventing any pixels from being displayed on the monitor.



Ball Segments: Three segments are shown in the primary colors red, green, and blue. Overlapping regions between two segments, in purple and yellow, prevented a small change from causing a significant change in predicted angle.

In addition to being able to accomplish two tasks at once, this method had the advantage of not requiring an estimate of the actual angle of the paddle, which appeared to be either be to noisy or to require more complex mathematics than would be easily programmable in hardware (linear regression, edge detection, etc). A drawback of this was that only a finite number of angles could be implemented, however in actual gameplay this did not seem to be too significant of a drawback.

On each frame, it was first seen if there was a collision with the paddle. If not, the ball's future position was calculated to see if it remained "in bounds". If it would leave the bounds set by the game parameters, it would update its internally stored velocities and use the new velocities to determine its next position. This prevented the ball from temporarily going into the background, which looked sloppy. Additionally, if the ball would go out of bounds on the left or the right, a one clock width signal was raised so that the appropriate player would lose health.

If there was a collision with the paddle, new velocities were determined using precoded logic taking into account the angle of the ball's velocity as well as the paddle. Because we were had only a finite number of angles, we were able to improve gameplay by massaging the return angle. For example, if there were too shallow of an angle between the paddle and the ball's approach, a physics simulator could potentially allow for the ball to continue to travel towards their wall. If the paddle detection was noisy and the angle determined was off, the player could be quite upset about this happening. To prevent this, the physics was "massaged" so that if the player managed to hit the ball, it would be returned, but keeping the goal of the player being able to control the angle.

VI. Graphics (Mike and Nick)

Modules: vga_select, ball, paddle_pixel, victory_gen, defeat_gen, health_bar, delay, *debounce*

VGA Select (Mike and Nick)

This module generates the graphics for the ball, paddles, heath bars, and game over screens, and internally takes care of the game logic (i.e. updating health bars and switching to a game over state when a player loses). It also implements a reset signal that has a delay before the game restarts, giving a player time to walk to the labkit, press the reset button, and return to the playing area.

Ball Display (Nick)

The ball module also uses its internal x and y coordinates, as well as a parameterized radius variable, to determine if a given VGA hcount and vcount was part of the ball. The center was found by adding the radius to top-left x and y, and the Pythagorean Theorem was used to compare the distance from the center to the radius. Both sides of the equation were squared so that no square-root was needed.

Health Bar Module (Nick)

The module took the current pixel display values (hcount and vcount), player healths, and the boundary parameters to determine if the given pixel should be replaced by a health bar. The health bars each had three bounded regions determined by distance from y-max for the top and bottom, and distance from x-min or x-max for the left or right extreme. The last bound was taken by multiplying health by a scalar value (2), and adding it to the other horizontal extreme. The health bars were evaluated separately, but the module returns whether it is a health pixel for either player's health bar simultaneously. The bars move if the window size changes mid-game.

Paddle Graphics (Mike)

Display different colour paddles for either player by replacing the detected paddle with green pixels on one side of the screen, and red pixels on the other side of the screen. No complex blob detection algorithms needed.

Defeat/Victory screens (Mike)

Generates the defeat and victory text when a player loses all health. These two modules, each 20000 lines long, contain a massive case expression that is a lookup table for what colour each pixel should be within the bounds of the defeat/victor graphic (100px x 200px). These statements were generated using Java and copy pasted into verilog. This module is designed to automatically reposition itself if the game boundaries change.

Delay (Mike)

Delays a specified size input signal by a specified number of clock cycles. This is used to pipeline the camera RGB data for a single pixel so that it arrives at the VGA selector at the same time as the corresponding is_background and is_pixel signals arrive.

Conclusion

Mike

My biggest takeaway from this class, aside learning Verilog starting from having absolutely no experience with the language, is the way of thinking when it comes to dealing with low level systems. Coming into this project I was still thinking of the implementation in a very high-level, sequential way: first get the entire frame from the camera, then threshold the entire frame on HSV, then apply the kernels to the binary output, and so on until an entire frame's worth of output was produced. I was stuck in the programming mentality of thinking that one line of code must execute before the next can begin.

The more I worked on the project however, the more I realized that the key to Verilog's speed was the ability to treat everything as a massive pipeline: once we receive a new pixel from the camera we should begin processing it immediately, and simply add delays where necessary if data needed to be synchronized to other information computed elsewhere in a slower process. Coming to this key insight about Verilog would've streamlined the design process immensely, since for a while I was stuck on storing the output of the colour keying module to RAM and subsequently retrieving it, but had I gotten into the mindset of everything being one huge pipeline I would have realized that storing an entire frame's worth of information was an unnecessary since the subsequent modules didn't need to wait on all the information being ready for them to work properly. Though I got a taste of this kind of thinking in 6.004, this class really made it stick for me because of the sheer amount of practice it provided in terms of thinking about how data flows in a clocked system.

Acknowledgements

Gim Hom and Joe Steinmeyer for a fun and well-taught class, and Joe especially for providing a last-minute RJ45 extension cable.

Diana Wofk for getting us started with chroma-keying and pointing us toward existing resources.

Appendix I: Usage Instructions

Setting up the area

For testing we taped a 5′ x 7′ green screen to a portable projector screen, but for optimal play we taped a massive 7′ x 12′ green screen to the wall with the window in the large open space in lab. To get the camera and the labkit into the right position required moving a lab bench to face the green screen however, which needed an extension cord and ethernet extender to connect the computer to the wired network in order to log in.

Making the paddles

We used blue construction paper tubes for our paddles, which seemed to work better than most colours in terms of being properly detected by the HSV detection scheme. In theory any highly saturated colour would work well.

Buttons and Switches

Which parameter you are adjusting depends on SW[4, 2:0]. In all cases, unless otherwise specified, up and down respectively increase and decrease the upper bound of that parameter, while right and left do the same for the lower bound. The hex display will show first the upper bound, then the lower bound.

sw[4:0]	parameter
0?000	Green screen hue range
0?001	Green screen saturation range
0?010	Green screen value range
1?000	Paddle screen hue range
1?001	Paddle screen saturation range
1?010	Paddle screen value range
0?011	Play area Y limits
0?100	Play area X limits
0?101	Up/down: +/- Y coordinate of top left corner of background image left/right: +/- X coordinate of top left corner of background image

Sometimes it is useful to turn off the background replacement effect, this can be toggled with sw[5].

Turning off camera pixels (what the monitor shows where the pixel is neither paddle nor green screen) can be helpful for seeing how noisey the filtering is. This can be accomplished with sw[6]

Finally, sw[7] toggles whether the kernel is applied.

To reset the game, hit button 3. The ball will return to the middle of the screen and player health bars will reset, and the ball will start moving after 5 seconds.

Appendix II: Verilog Listing

File	Modules Contained	Purpose
background_gen.v	background_gen	Gets the pixel with which to replace the green screen
ball.v	ball	Creates the ball graphic
bianarizer.v	binarizer	Detects background and paddles
blut.v (IP core)	blut	Blue lookup table
debounce.v	debounce	Debounces the button inputs
defeat_gen.v	defeat_gen, victory_gen	Creates defeat and victory graphics
display_16hex.v	display_16hex	Shows data on hex displays
divider.v (IP core)	divider	16 bit divider
glut.v (IP core)	glut	Green lookup table
health_bar.v	health_bar	Create health bar graphics
ntsc2zbt.v	ntsc2zbt	Stores data from camera into ZBT
parameter_select.v	parameter_select	Selects the parameter to adjust
rgb2hsv.v	rgb2hsv	Converts rgb to hsv
rlut.v (IP core)	rlut	Red lookup table
rom.v	rom	4 bit indices for each pixel in background image
vga_select.v	vga_select paddle_pixel	Generate graphics and choose what to display at that pixel
video_decoder.v	ntsc_decode adv7185init	Decodes ntsc signals
ycrcb.v	YCrCb2RGB	Converts YCrCb to RGB
zbt_6111_sample.v	zbt_6111_sample *xvga* delay *vram_display*	The global module, instantiates everything and creates wires for all modules to communicate

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Generates the background (the pixels the green screen is replaced by) using an
// image preloaded in ROM
//
// Mike Wang

module background_gen(input clk,
                      input [10:0] hcount,
                                   x_offset,  // how much to shift the image in x
                                   x_min,  // x bounds of the play area
                                   x_max,
                                   ball_track_x,  // ball x, used to make background
 move with ball
                      input [9:0] vcount,
                                  y_offset,
                                  y_min,
                                  ball_track_y,
                      input mode,
                      output [7:0] r,g,b);

    wire [3:0] index;  // the colour index (4 bit colour) of the pixel
    wire [9:0] translated_v;  // translate vcount and hcount by the proper offsets
    wire [10:0] translated_h;
    reg [10:0] time_delta = 0;  // this is used for the mode where the background si
mple moves from left to right and loops over
    // translate differently based on the mode
                                                      // this mode causes the image to
 bounce around with the ball
    assign translated_v = mode ? vcount-y_offset : vcount-y_offset-{1'b0, ball_track
_y[9:1]};
    assign translated_h = mode ? hcount-x_offset+time_delta : hcount-x_offset-{1'b0,
 ball_track_x[10:1]};

    // logic to move the background continuously to the right at a slow pace
    parameter INCR = 1000000;
    reg [30:0] counter = 0;
    always @(posedge clk) begin

        if (counter == INCR) begin
            time_delta <= time_delta + 1;
            counter <= 0;
        end else begin
            counter<= counter + 1;
        end
    end

    // the data in the following ROMs were generated using the Matlab code generousl
y provided by 6.111 staff
    //
    // it may be worth pointing out on the course website that, in order to get a 8
or 4 bit image from a 24 bit image,
    // one can use Gimp which comes installed on all lab computers. Matlab is also i
nstalled on the lab computers
    // but accessing it is quite involved (or maybe I did something wrong it is poss
ible)

    // the 4 bit info for each pixel
    rom index_rom(.clka(clk), .addra({translated_v[8:0],translated_h[9:0]}), .douta(
index));
    // look up the r, g, b data using the 4 bit index
    rlut red_tab(.clka(clk), .addra(index), .douta(r));
    glut green_tab(.clka(clk), .addra(index), .douta(g));
    blut blue_tab(.clka(clk), .addra(index), .douta(b));

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Generate the ball and implements simple physics
//
// Nicholas Waltman
//

module ball #(parameter INIT_VX = 5,
               parameter INIT_VY = 3,
               parameter INIT_X = 500,
               parameter INIT_Y = 200,
               parameter R = 30,
               parameter COLLIDE_COLOUR = 24'b00111111_00001111_00000011,
               parameter COLOUR = 24'b00000011_00001111_00111111)
             (input clk,
              input [10:0] hcount, x_min, x_max,
              input [9:0] vcount, y_max, y_min,
              input paddle_px,
              input [23:0] cur_px,
              output reg[23:0] new_px,
              output reg [63:0] dispdata,
              output reg [7:0] contact,
              output reg signed [12:0] x,
              output reg signed [12:0] y,
              input reset,
              input stop_moving,
              input [2:0] new_vx, new_vy);

    reg signed [5:0]  vx = INIT_VX;
    reg signed [5:0]  vy = INIT_VY;

    wire signed [12:0] px;
    assign px = hcount;
    wire signed [12:0] py;
    assign py = vcount;
    wire signed [12:0] sx_min, sx_max, sy_min, sy_max;
    assign sx_min = x_min;
    assign sx_max = x_max;
    assign sy_min = y_min;
    assign sy_max = y_max;

    wire signed [12:0] cx;
    wire signed [12:0] cy;
    assign cx = px - x - R;
    assign cy = py - y - R;

    reg [10:0] contact_cooldown;

    wire pixel_is_ball;
    //assign pixel_is_ball = !( (px < x)  || (px > x + 2*R) || (py < y) || (py > y +
 2*R)); // square
    assign pixel_is_ball = (x + R - px) * (x + R -px) + (y + R - py) * (y + R -py) <
= R*R; //circle

    reg [15:0] total;
    reg [7:0] seg0, seg1, seg2, seg3, seg4, seg5, seg6, seg7;
    wire seg0m, seg1m, seg2m, seg3m, seg4m, seg5m, seg6m ,seg7m; // segxm is max
    wire seg0c, seg1c, seg2c, seg3c, seg4c, seg5c, seg6c ,seg7c; // currently segxc

    assign seg0m = (seg0 >= seg1) && (seg0 >= seg2) && (seg0 >= seg3) && (seg0 >= se
g4) && (seg0 >= seg5) && (seg0 >= seg6) && (seg0 >= seg7);
    assign seg1m = (seg1 >= seg0) && (seg1 >= seg2) && (seg1 >= seg3) && (seg1 >= se
g4) && (seg1 >= seg5) && (seg1 >= seg6) && (seg1 >= seg7);
    assign seg2m = (seg2 >= seg1) && (seg2 >= seg0) && (seg2 >= seg3) && (seg2 >= se
g4) && (seg2 >= seg5) && (seg2 >= seg6) && (seg2 >= seg7);
    assign seg3m = (seg3 >= seg1) && (seg3 >= seg2) && (seg0 >= seg3) && (seg3 >= se
g4) && (seg3 >= seg5) && (seg3 >= seg6) && (seg3 >= seg7);
    assign seg4m = (seg4 >= seg1) && (seg4 >= seg2) && (seg4 >= seg3) && (seg4 >= se
g0) && (seg4 >= seg5) && (seg4 >= seg6) && (seg4 >= seg7);
    assign seg5m = (seg5 >= seg1) && (seg5 >= seg2) && (seg5 >= seg3) && (seg5 >= se
g4) && (seg5 >= seg0) && (seg5 >= seg6) && (seg5 >= seg7);
    assign seg6m = (seg6 >= seg1) && (seg6 >= seg2) && (seg6 >= seg3) && (seg6 >= se
g4) && (seg6 >= seg5) && (seg6 >= seg0) && (seg6 >= seg7);
    assign seg7m = (seg7 >= seg1) && (seg7 >= seg2) && (seg7 >= seg3) && (seg7 >= se
g4) && (seg7 >= seg5) && (seg7 >= seg6) && (seg7 >= seg0);

    assign seg0c = cx < -24;
    assign seg1c = cy - 17 > x + 17;
    assign seg2c = cy < -24;
    assign seg3c = cy - 17 > -cx + 17;
    assign seg4c = cy > 24;
    assign seg5c = cy + 17 < cx - 17;
    assign seg6c = cy > 24;
    assign seg7c = cy + 17 < - (cx + 17 ) ;

    reg signed [1:0] going_right, going_down;

    wire will_hit_left, will_hit_right, will_hit_vert;
    assign will_hit_left = x +going_right* vx <sx_min;
    assign will_hit_right = x + going_right*vx + 2*R > sx_max;

    // position and velocity
    initial begin
        x <= INIT_X;
        y <= INIT_Y;

        going_right <= 1;
        going_down <= 1;
    end

    always @(posedge clk) begin
        if (reset) begin
            going_right <= new_vx[0];
            going_down <= new_vy[0];
            x <= (x_max+x_min)>>1;
            y <= (y_max+y_min)>>1;
            vx <= new_vx;
            vy <= new_vy;
        end

        if (stop_moving) begin
            contact <= 0;
        end

        if ((px == 0) && (py == 0) && !(reset) && !stop_moving) begin// frame update

            total <= 0; // reset overlap counts
            // update positions
            x <= x + going_right*vx;
            y <= y + going_down*vy;

            // update velocities if it will hit edge
            if (contact_cooldown > 0) begin
                contact_cooldown <= contact_cooldown -1;
            end else if ( total > 25) begin
                contact_cooldown <= 30;
                //dispdata <= {seg0, seg1, seg2, seg3, seg4, seg5, seg6, seg7};
                if (seg0m || seg4m) begin  // vertical paddle
                    going_right <= -going_right;
                //end else if (seg2m || seg6m) begin // horizontal paddle
                //  going_down <= -going_down;
                end else if (   (seg1m && (going_down ==  1)) || (seg3m && (going_do
wn ==  1)) ||
                                        (seg5m && (going_down == -1)) || (seg5m && (goin
g_down == -1))  )begin
                    going_right <= -going_right;
                end else begin //angled paddle, flat hit
                    going_right <= -going_right;
                    going_down <= -going_down;
                end
            end

            if (will_hit_left) begin
                going_right <= -going_right;
                contact <= 10;
            end else if (will_hit_right) begin
                going_right <= -going_right;
                contact <= 20;
            end
            if ((y + going_down*vy <sy_min) ||( y + going_down*vy + 2*R > sy_max ))
begin
                going_down <= -going_down;
            end

            if ( !(will_hit_left || will_hit_right))
                contact <= 0;
        end // frame updates
        else  begin
            // ****************************
            // happens every update of clock
            // ****************************

            if (pixel_is_ball && paddle_px) begin  // overlapping pixel
                total <= total + 1;
                seg0 <= seg0 + seg0c;
                seg1 <= seg1 + seg1c;
                seg2 <= seg2 + seg2c;
                seg3 <= seg3 + seg3c;
                seg4 <= seg4 + seg4c;
                seg5 <= seg5 + seg5c;
                seg6 <= seg6 + seg6c;
                seg7 <= seg7 + seg7c;

            end // overlapping pixel


            // pixel selection for vga selecter
            if (!pixel_is_ball) begin
                new_px <= cur_px;
            end else begin
                new_px <= COLOUR;
            end

        end
    end
endmodule
```

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Detects if this pixel is a background or a paddle pixel. Implements extremely
// basic erosion and dilation filtering: only checks pixels to the left and right,
// not above and below, the pixel in queston. Gets the job done though.
// Outputs whether this pixel is a background or paddle pixel.
//
// Mike Wang
//

module binarizer (input clk,
                  input [7:0] r, g, b, // input rgb values for this pixel
                  input [10:0] hcount,  // x coord of the data
                  input [9:0] vcount,   // y coord of the data
                  input [7:0] hue_max, hue_min, // hsv bounds for background
                              sat_max, sat_min,
                              val_max, val_min,
                  input [7:0] pad_hue_max, pad_hue_min, // hsv bounds for paddle
                              pad_sat_max, pad_sat_min,
                              pad_val_max, pad_val_min,
                  output reg is_background,
                  output reg is_paddle,
                  input activate_kernel
                  );

    wire [7:0] h, s, v;

    reg [4:0] initial_binary; // hold the initial binarization, i.e. does this pixel
 fall within the hsv bounds for the background
    reg [4:0] eroded;  // hold the result of erosion

    // the same, for the paddle pixels
    reg [4:0] initial_paddle;
    reg [4:0] eroded_paddle;

    // instatiate a converter module from rgb to hsv, graciously provided by 6.111 s
taff
    rgb2hsv converter(.clock(clk), .r(r), .g(g), .b(b), .h(h), .s(s), .v(v));

    always @(posedge clk) begin
        // if pixel is within hsv bounds for background, shift 1 into the background
 reg and 0 into the paddle reg
        if (h<=hue_max && h>=hue_min &&
            s<=sat_max && s>=sat_min &&
            v<=val_max && v>=val_min) begin
            initial_binary <= {1, initial_binary[4:1]};
            initial_paddle <= {0, initial_paddle[4:1]};
        // else check if in paddle bounds
        end else if (h<=pad_hue_max && h>=pad_hue_min &&
                     s<=pad_sat_max && s>=pad_sat_min &&
                     v<=pad_val_max && v>=pad_val_min) begin
             initial_paddle <= {1, initial_paddle[4:1]};
             initial_binary <= {0, initial_binary[4:1]};
        // otherwise it's neither
        end else begin
             initial_paddle <= {0, initial_paddle[4:1]};
             initial_binary <= {0, initial_binary[4:1]};
        end

        // erode the initial binary: only shift in a 1 if all 1s
        if (initial_binary == 'b11111) begin
            eroded <= {1,eroded[4:1]};
        end else begin
            eroded <= {0,eroded[4:1]};
        end

        if (initial_paddle == 'b11111) begin
            eroded_paddle <= {1,eroded_paddle[4:1]};
        end else begin
            eroded_paddle <= {0,eroded_paddle[4:1]};
        end

        // dilate the eroded result: output 1 if any of the neighbouring pixels was
```

```
1
        if (activate_kernel) begin
            is_background <= (eroded > 0) ? 1:0;
        end else begin
            is_background <= initial_binary;
        end

        if (activate_kernel) begin
            is_paddle <= (eroded_paddle >0) ? 1:0;
        end else begin
            is_paddle <= initial_paddle;
        end
    end

endmodule

//
///////////////////////////////////////////////////////////////////////////////////
```

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2007 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synthesis directives "translate_off/translate_on" specified below are
// supported by Xilinx, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file blut.v when simulating
// the core, blut. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module blut(
	clka,
	addra,
	douta);


input clka;
input [3 : 0] addra;
output [7 : 0] douta;

// synthesis translate_off

      BLK_MEM_GEN_V2_8 #(
		.C_ADDRA_WIDTH(4),
		.C_ADDRB_WIDTH(4),
		.C_ALGORITHM(1),
		.C_BYTE_SIZE(9),
		.C_COMMON_CLK(0),
		.C_DEFAULT_DATA("0"),
		.C_DISABLE_WARN_BHV_COLL(0),
		.C_DISABLE_WARN_BHV_RANGE(0),
		.C_FAMILY("virtex2"),
		.C_HAS_ENA(0),
		.C_HAS_ENB(0),
		.C_HAS_MEM_OUTPUT_REGS_A(0),
		.C_HAS_MEM_OUTPUT_REGS_B(0),
		.C_HAS_MUX_OUTPUT_REGS_A(0),
		.C_HAS_MUX_OUTPUT_REGS_B(0),
		.C_HAS_REGCEA(0),
		.C_HAS_REGCEB(0),
		.C_HAS_SSRA(0),
		.C_HAS_SSRB(0),
		.C_INIT_FILE_NAME("blut.mif"),
		.C_LOAD_INIT_FILE(1),
        .C_MEM_TYPE(3),
        .C_MUX_PIPELINE_STAGES(0),
        .C_PRIM_TYPE(1),
        .C_READ_DEPTH_A(16),
        .C_READ_DEPTH_B(16),
        .C_READ_WIDTH_A(8),
        .C_READ_WIDTH_B(8),
        .C_SIM_COLLISION_CHECK("ALL"),
        .C_SINITA_VAL("0"),
        .C_SINITB_VAL("0"),
        .C_USE_BYTE_WEA(0),
        .C_USE_BYTE_WEB(0),
        .C_USE_DEFAULT_DATA(0),
        .C_USE_ECC(0),
        .C_USE_RAMB16BWER_RST_BHV(0),
        .C_WEA_WIDTH(1),
        .C_WEB_WIDTH(1),
        .C_WRITE_DEPTH_A(16),
        .C_WRITE_DEPTH_B(16),
        .C_WRITE_MODE_A("WRITE_FIRST"),
        .C_WRITE_MODE_B("WRITE_FIRST"),
        .C_WRITE_WIDTH_A(8),
        .C_WRITE_WIDTH_B(8),
        .C_XDEVICEFAMILY("virtex2"))
    inst (
        .CLKA(clka),
        .ADDRA(addra),
        .DOUTA(douta),
        .DINA(),
        .ENA(),
        .REGCEA(),
        .WEA(),
        .SSRA(),
        .CLKB(),
        .DINB(),
        .ADDRB(),
        .ENB(),
        .REGCEB(),
        .WEB(),
        .SSRB(),
        .DOUTB(),
        .DBITERR(),
        .SBITERR());


// synthesis translate_on

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of blut is "black_box"

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module
//
////////////////////////////////////////////////////////////////////////////////

module debounce (reset, clk, noisy, clean);
   input reset, clk, noisy;
   output clean;

   parameter NDELAY = 650000;
   parameter NBITS = 20;

   reg [NBITS-1:0] count;
   reg xnew, clean;

   always @(posedge clk)
     if (reset) begin xnew <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != xnew) begin xnew <= noisy; count <= 0; end
     else if (count == NDELAY) clean <= xnew;
     else count <= count+1;

endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// This module drives the labkit hex displays and shows the value of
// 8 bytes (16 hex digits) on the displays.
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 02-Nov-05 Ike: updated to make it completely synchronous
//
// Inputs:
//
//   reset       - active high
//   clock_27mhz - the synchronous clock
//   data        - 64 bits; each 4 bits gives a hex digit
//
// Outputs:
//
//    disp_*     - display lines used in the 6.111 labkit (rev 003 & 004)
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data_in,
        disp_blank, disp_clock, disp_rs, disp_ce_b,
        disp_reset_b, disp_data_out);

   input reset, clock_27mhz;    // clock and reset (active high reset)
   input [63:0] data_in;        // 16 hex nibbles to display

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Display Clock
   //
   // Generate a 500kHz clock for driving the displays.
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [5:0] count;
   reg [7:0] reset_count;
//   reg         old_clock;
   wire      dreset;
   wire      clock = (count<27) ? 0 : 1;

   always @(posedge clock_27mhz)
     begin
    count <= reset ? 0 : (count==53 ? 0 : count+1);
    reset_count <= reset ? 100 : ((reset_count==0) ? 0 : reset_count-1);
//  old_clock <= clock;
     end

   assign dreset = (reset_count != 0);
   assign disp_clock = ~clock;
   wire   clock_tick = ((count==27) ? 1 : 0);
//   wire   clock_tick = clock & ~old_clock;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Display State Machine
   //
   ////////////////////////////////////////////////////////////////////////////
```

```
reg [7:0] state;     // FSM state
reg [9:0] dot_index;     // index to current dot being clocked out
reg [31:0] control;      // control register
reg [3:0] char_index;    // index of current character
reg [39:0] dots;     // dots for a single digit
reg [3:0] nibble;        // hex nibble of current character
reg [63:0] data;

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clock_27mhz)
  if (clock_tick)
    begin
   if (dreset)
     begin
        state <= 0;
        dot_index <= 0;
        control <= 32'h7F7F7F7F;
     end
   else
     casex (state)
       8'h00:
     begin
        // Reset displays
        disp_data_out <= 1'b0;
        disp_rs <= 1'b0; // dot register
        disp_ce_b <= 1'b1;
        disp_reset_b <= 1'b0;
        dot_index <= 0;
        state <= state+1;
     end

       8'h01:
     begin
        // End reset
        disp_reset_b <= 1'b1;
        state <= state+1;
     end

       8'h02:
     begin
        // Initialize dot register (set all dots to zero)
        disp_ce_b <= 1'b0;
        disp_data_out <= 1'b0; // dot_index[0];
        if (dot_index == 639)
          state <= state+1;
        else
          dot_index <= dot_index+1;
     end

       8'h03:
     begin
        // Latch dot data
        disp_ce_b <= 1'b1;
        dot_index <= 31;     // re-purpose to init ctrl reg
        state <= state+1;
     end

       8'h04:
     begin
        // Setup the control register
        disp_rs <= 1'b1; // Select the control register
        disp_ce_b <= 1'b0;
        disp_data_out <= control[31];
        control <= {control[30:0], 1'b0};    // shift left
        if (dot_index == 0)
          state <= state+1;
        else
          dot_index <= dot_index-1;
     end

       8'h05:
        begin
           // Latch the control register data / dot data
           disp_ce_b <= 1'b1;
           dot_index <= 39;     // init for single char
           char_index <= 15;        // start with MS char
           data <= data_in;
           state <= state+1;
        end

          8'h06:
        begin
           // Load the user's dot data into the dot reg, char by char
           disp_rs <= 1'b0;         // Select the dot register
           disp_ce_b <= 1'b0;
           disp_data_out <= dots[dot_index]; // dot data from msb
           if (dot_index == 0)
                 if (char_index == 0)
                   state <= 5;          // all done, latch data
             else
               begin
              char_index <= char_index - 1; // goto next char
              data <= data_in;
              dot_index <= 39;
               end
           else
             dot_index <= dot_index-1;  // else loop thru all dots
        end

        endcase // casex(state)
       end

   always @ (data or char_index)
     case (char_index)
       4'h0:        nibble <= data[3:0];
       4'h1:        nibble <= data[7:4];
       4'h2:        nibble <= data[11:8];
       4'h3:        nibble <= data[15:12];
       4'h4:        nibble <= data[19:16];
       4'h5:        nibble <= data[23:20];
       4'h6:        nibble <= data[27:24];
       4'h7:        nibble <= data[31:28];
       4'h8:        nibble <= data[35:32];
       4'h9:        nibble <= data[39:36];
       4'hA:        nibble <= data[43:40];
       4'hB:        nibble <= data[47:44];
       4'hC:        nibble <= data[51:48];
       4'hD:        nibble <= data[55:52];
       4'hE:        nibble <= data[59:56];
       4'hF:        nibble <= data[63:60];
     endcase

   always @(nibble)
     case (nibble)
       4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
       4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
       4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
       4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
       4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
       4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
       4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
       4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
       4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
       4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
       4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
       4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
       4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
       4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
       4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
       4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
     endcase

endmodule
```

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2007 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synthesis directives "translate_off/translate_on" specified below are
// supported by Xilinx, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file glut.v when simulating
// the core, glut. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module glut(
    clka,
    addra,
    douta);


input clka;
input [3 : 0] addra;
output [7 : 0] douta;

// synthesis translate_off

      BLK_MEM_GEN_V2_8 #(
        .C_ADDRA_WIDTH(4),
        .C_ADDRB_WIDTH(4),
        .C_ALGORITHM(1),
        .C_BYTE_SIZE(9),
        .C_COMMON_CLK(0),
        .C_DEFAULT_DATA("0"),
        .C_DISABLE_WARN_BHV_COLL(0),
        .C_DISABLE_WARN_BHV_RANGE(0),
        .C_FAMILY("virtex2"),
        .C_HAS_ENA(0),
        .C_HAS_ENB(0),
        .C_HAS_MEM_OUTPUT_REGS_A(0),
        .C_HAS_MEM_OUTPUT_REGS_B(0),
        .C_HAS_MUX_OUTPUT_REGS_A(0),
        .C_HAS_MUX_OUTPUT_REGS_B(0),
        .C_HAS_REGCEA(0),
        .C_HAS_REGCEB(0),
        .C_HAS_SSRA(0),
        .C_HAS_SSRB(0),
        .C_INIT_FILE_NAME("glut.mif"),
        .C_LOAD_INIT_FILE(1),
        .C_MEM_TYPE(3),
        .C_MUX_PIPELINE_STAGES(0),
        .C_PRIM_TYPE(1),
        .C_READ_DEPTH_A(16),
        .C_READ_DEPTH_B(16),
        .C_READ_WIDTH_A(8),
        .C_READ_WIDTH_B(8),
        .C_SIM_COLLISION_CHECK("ALL"),
        .C_SINITA_VAL("0"),
        .C_SINITB_VAL("0"),
        .C_USE_BYTE_WEA(0),
        .C_USE_BYTE_WEB(0),
        .C_USE_DEFAULT_DATA(0),
        .C_USE_ECC(0),
        .C_USE_RAMB16BWER_RST_BHV(0),
        .C_WEA_WIDTH(1),
        .C_WEB_WIDTH(1),
        .C_WRITE_DEPTH_A(16),
        .C_WRITE_DEPTH_B(16),
        .C_WRITE_MODE_A("WRITE_FIRST"),
        .C_WRITE_MODE_B("WRITE_FIRST"),
        .C_WRITE_WIDTH_A(8),
        .C_WRITE_WIDTH_B(8),
        .C_XDEVICEFAMILY("virtex2"))
    inst (
        .CLKA(clka),
        .ADDRA(addra),
        .DOUTA(douta),
        .DINA(),
        .ENA(),
        .REGCEA(),
        .WEA(),
        .SSRA(),
        .CLKB(),
        .DINB(),
        .ADDRB(),
        .ENB(),
        .REGCEB(),
        .WEB(),
        .SSRB(),
        .DOUTB(),
        .DBITERR(),
        .SBITERR());


// synthesis translate_on

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of glut is "black_box"

endmodule
```

```
`timescale 1ns / 1ps
///////////////////////////////////////////////////////////////////////////////////
//
// Generate and update the health bar
//
// Nicholas Waltman
//

module health_bar(
    input clk,
    input signed [7:0] player_health,
    input [10:0] x_min,
    input [10:0] x_max,
    input [9:0] y_max,
    input [10:0] hcount,
    input [10:0] vcount,
     input right_align,
    output is_health
    );
    wire signed [12:0] px;
    assign px = hcount;
    wire signed [12:0] py;
    assign py = vcount;
    wire signed [12:0] sx_min, sx_max, sy_max;
    assign sx_min = x_min;
    assign sx_max = x_max;

    assign sy_max = y_max;


    wire not_dead, right_height, left_align_ok, right_align_ok;
    assign not_dead =(player_health > 0);
    assign right_height = (sy_max - py < 40) && (sy_max - py > 20);
    assign left_align_ok = !right_align && (px - sx_min > 20 ) && (px - sx_min - 20
< 2 * player_health);
    assign right_align_ok = right_align && (sx_max - px  > 20 ) && (sx_max - 2*playe
r_health - 20 < px);

    assign is_health = not_dead && right_height && (left_align_ok || right_align_ok)
;

endmodule
```

```
//¿
// File:   ntsc2zbt.v¿
// Date:   27-Nov-05¿
// Author: I. Chuang <ichuang@mit.edu>¿
//¿
// Example for MIT 6.111 labkit showing how to prepare NTSC data¿
// (from Javier's decoder) to be loaded into the ZBT RAM for video¿
// display.¿
//¿
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to¿
// store 4 bytes of black-and-white intensity data from the NTSC¿
// video input.¿
//¿
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>¿
// Date    : 11-May-09  // gph mod 11/3/2011¿
//¿
// ¿
// Bug due to memory management will be fixed. It happens because¿
// the memory addressing protocol is off between ntsc2zbt.v and¿
// vram_display.v. There are 2 solutions:¿
// -. Fix the memory addressing in this module (neat addressing protocol)¿
//    and do memory forecast in vram_display module.¿
// -. Do nothing in this module and do memory forecast in vram_display¿
//    module (different forecast count) while cutting off reading from ¿
//    address(0,0,0).¿
//¿
// Bug in this module causes 4 pixel on the rightmost side of the camera¿
// to be stored in the address that belongs to the leftmost side of the ¿
// screen.¿
// ¿
// In this example, the second method is used. NOTICE will be provided¿
// on the crucial source of the bug.¿
//¿
/////////////////////////////////////////////////////////////////////////////¿
// Prepare data and address values to fill ZBT memory with NTSC data¿
¿
module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);¿
¿
   input      clk;   // system clock¿
   input      vclk;  // video clock from camera¿
   input [2:0]   fvh;¿
   input      dv;¿
   input [17:0]      din; // mw: from 7 to 17¿
   output [18:0] ntsc_addr;¿
   output [35:0] ntsc_data;¿
   output     ntsc_we;    // write enable for NTSC data¿
   input      sw;         // switch which determines mode (for debugging)¿
¿
//   parameter   COL_START = 10'd30; // 171210
   parameter     COL_START = 10'd800;¿
¿
   parameter     ROW_START = 10'd30;¿
¿
   // here put the luminance data from the ntsc decoder into the ram¿
   // this is for 1024 * 788 XGA display¿
¿
   reg [9:0]     col = 0;¿
   reg [9:0]     row = 0;¿
   reg [17:0]    vdata = 0;  // mw: from 7 to 17¿
   reg        vwe;¿
   reg        old_dv;¿
   reg        old_frame; // frames are even / odd interlaced¿
   reg        even_odd;  // decode interlaced frame to this wire¿
   ¿
   wire       frame = fvh[2];¿
   wire       frame_edge = frame & ~old_frame;¿
¿
   always @ (posedge vclk) //LLC1 is reference¿
     begin¿
    old_dv <= dv;¿
    vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it¿
    old_frame <= frame;¿
    even_odd = frame_edge ? ~even_odd : even_odd;¿
```

```
¿
    if (!fvh[2])¿
      begin¿
         col <= fvh[0] ? COL_START : ¿
//          (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
             (!fvh[2] && !fvh[1] && dv && (col > 0)) ? col - 1 : col; //171210: chan
ged this line and COL_START so that the image is no longer reversed¿
         row <= fvh[1] ? ROW_START : ¿
            (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;¿
         vdata <= (dv && !fvh[2]) ? din : vdata;¿
      end¿
   end¿
¿
   // synchronize with system clock¿
¿
   reg [9:0] x[1:0],y[1:0];¿
   reg [17:0] data[1:0];  // mw: from 7 to 17¿
   reg       we[1:0];¿
   reg       eo[1:0];¿
¿
   always @(posedge clk)¿
     begin¿
    {x[1],x[0]} <= {x[0],col};¿
    {y[1],y[0]} <= {y[0],row};¿
    {data[1],data[0]} <= {data[0],vdata};¿
    {we[1],we[0]} <= {we[0],vwe};¿
    {eo[1],eo[0]} <= {eo[0],even_odd};¿
     end¿
¿
   // edge detection on write enable signal¿
¿
   reg old_we;¿
   wire we_edge = we[1] & ~old_we;¿
   always @(posedge clk) old_we <= we[1];¿
¿
   // shift each set of four bytes into a large register for the ZBT¿
   ¿
   reg [31:0] mydata;¿
   always @(posedge clk)¿
     if (we_edge)¿
       mydata <= { mydata[17:0], data[1] };  //mw: from 7 to 17¿
   ¿
   // NOTICE : Here we have put 4 pixel delay on mydata. For example, when:¿
   // (x[1], y[1]) = (60, 80) and eo[1] = 0, then:¿
   // mydata[31:0] = ( pixel(56,160), pixel(57,160), pixel(58,160), pixel(59,160) )¿
   // This is the root of the original addressing bug. ¿
   ¿
   ¿
   // NOTICE : Notice that we have decided to store mydata, which¿
   //          contains pixel(56,160) to pixel(59,160) in address¿
   //          (0, 160 (10 bits), 60 >> 2 = 15 (8 bits)).¿
   //¿
   //          This protocol is dangerous, because it means¿
   //          pixel(0,0) to pixel(3,0) is NOT stored in address¿
   //          (0, 0 (10 bits), 0 (8 bits)) but is rather stored¿
   //          in address (0, 0 (10 bits), 4 >> 2 = 1 (8 bits)). This¿
   //          calculation ignores COL_START & ROW_START.¿
   //          ¿
   //          4 pixels from the right side of the camera input will¿
   //          be stored in address corresponding to x = 0.¿
   // ¿
   //          To fix, delay col & row by 4 clock cycles.¿
   //          Delay other signals as well.¿
   ¿
   reg [39:0] x_delay;¿
   reg [39:0] y_delay;¿
   reg [3:0] we_delay;¿
   reg [3:0] eo_delay;¿
   ¿
   always @ (posedge clk)¿
   begin¿
     x_delay <= {x_delay[29:0], x[1]};¿
     y_delay <= {y_delay[29:0], y[1]};¿
     we_delay <= {we_delay[2:0], we[1]};¿
     eo_delay <= {eo_delay[2:0], eo[1]};¿
   end¿
   ¿
   // compute address to store data in¿
   wire [8:0] y_addr = y_delay[38:30];¿
    wire [9:0] x_addr = x_delay[39:30];¿
    ¿
   wire [18:0] myaddr = {y_addr[8:0], eo_delay[3], x_addr[9:1]}; // mw more addresse
s¿
   ¿
   // Now address (0,0,0) contains pixel data(0,0) etc.¿
   ¿
      ¿
   // alternate (256x192) image data and address¿
   wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};¿
   wire [18:0] myaddr2 = {1'b0, y_addr[8:0], eo_delay[3], x_addr[7:0]};¿
¿
   // update the output address and data only when four bytes ready¿
¿
   reg [18:0] ntsc_addr;¿
   reg [35:0] ntsc_data;¿
   wire ntsc_we = sw ? we_edge : (we_edge & (x_delay[30]==0));  // mw writing on xde
lay[30] == 0
¿
   always @(posedge clk)¿
     if ( ntsc_we )¿
       begin¿
          ntsc_addr <= sw ? myaddr2 : myaddr;    // normal and expanded modes¿
          ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};¿
       end¿
   ¿
endmodule // ntsc_to_zbt¿
¿
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Mike Wang
//
// Create Date:    19:31:04 12/04/2017
// Design Name:
// Module Name:    fsm
// Project Name:
// Target Devices:
// Tool versions:
// Description: FSM for parameter selection (HSV thresholds, play area limits,
//              various mode selects
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module parameter_select(input clk,
                        input [4:0] switch,                     // mode selection
                        input u, d, l, r,                        // button_input
                        output reg [7:0] hue_max, hue_min,  //hsv space bounds for b
ackground
                                         sat_max, sat_min,
                                         val_max, val_min,
                        output reg [10:0] x_max, x_min,     // game space bounds
                        output reg [9:0] y_max, y_min,
                        output reg [63:0] dispdata,         // display the current s
ettings on the hex display
                        output reg [7:0] pad_hue_max, pad_hue_min,  //hsv space boun
ds for paddle
                                         pad_sat_max, pad_sat_min,
                                         pad_val_max, pad_val_min,
                        output reg [10:0] x_offset,       // offset values for posit
ioning the background image
                        output reg [9:0] y_offset);

    wire incr_u, incr_d, incr_r, incr_l;  // wires to indicate which registers were
pressed
    wire fast;  // whether to increment quickly or not
    reg faster = 0;  // allow things in the always block to set the fast setting
    assign fast = faster;

    // instantitate an incrementor module for each button (see below for description
 of incrementor module
    incrementor up(clk, u, incr_u, fast);
    incrementor down(clk, d, incr_d, fast);
    incrementor left(clk, l, incr_l, fast);
    incrementor right(clk, r, incr_r, fast);

    // give default values of the settings
    initial x_min = 'hBE;
    initial x_max = 'h269;
    initial y_max = 'h233;
    initial y_min = 'hB6;

    initial hue_max = 8'h89;
    initial hue_min = 8'h2A;
    initial val_max = 8'hFF;
    initial val_min = 8'h27;
    initial sat_max = 8'hFF;
    initial sat_min = 8'h0;

    initial pad_hue_max = 8'hC2;
    initial pad_hue_min = 8'h74;
    initial pad_val_max = 8'hFF;
    initial pad_val_min = 8'h69;
    initial pad_sat_max = 8'hFF;
    initial pad_sat_min = 8'h1b;
    initial x_offset = 'h8F;
    initial y_offset = 70;

    always @(posedge clk) begin
        // begin the huge state machine
        // for all cases, up = increment upper bound, down = decrement upper bound
        //                    left = decrement lower bound, right = increment lower bo
und

        // there's a ? on bit 4 since I had to retroactively use that switch to adju
st some other parameter
        casex(switch)
            5'b0?000:begin // set the hue bounds of the background
                        faster <= 0;  // slowly, since we need to have fine control
of this setting
                        if (incr_u) begin
                            hue_max <= hue_max + 1;
                        end
                        if (incr_d) begin
                            hue_max <= hue_max - 1;
                        end
                        if (incr_l) begin
                            hue_min <= hue_min - 1;
                        end
                        if (incr_r) begin
                            hue_min <= hue_min + 1;
                        end
                        dispdata <= {4'd10, 8'b0, hue_max, hue_min};
                  end
            5'b0?001:begin  // set saturation bounds of background
                        faster <= 0;

                        if (incr_u) begin
                            sat_max <= sat_max + 1;
                        end
                        if (incr_d) begin
                            sat_max <= sat_max - 1;
                        end
                        if (incr_l) begin
                            sat_min <= sat_min - 1;
                        end
                        if (incr_r) begin
                            sat_min <= sat_min + 1;
                        end
                        dispdata <= {4'd11, 8'b0, sat_max, sat_min};
                  end
            5'b0?010:begin  // set value bounds of background
                        faster <= 0;

                        if (incr_u) begin
                            val_max <= val_max + 1;
                        end
                        if (incr_d) begin
                            val_max <= val_max - 1;
                        end
                        if (incr_l) begin
                            val_min <= val_min - 1;
                        end
                        if (incr_r) begin
                            val_min <= val_min + 1;
                        end
                        dispdata <= {4'd12, 8'b0, val_max, val_min};

                  end
            5'b0?011:begin  // set the x bounds of the play area
                        faster <= 1;

                        if (incr_u) begin
                            x_max <= x_max + 1;
                        end
                        if (incr_d) begin
                            x_max <= x_max - 1;
                        end
                        if (incr_l) begin
                            x_min <= x_min - 1;
                        end
                        if (incr_r) begin
                            x_min <= x_min + 1;
                        end
                        dispdata <= {4'd13, 8'b0, {1'b0,x_max}, {1'b0,x_min}};

                  end
            5'b0?100:begin  // set the y bounds of the play area
                        faster <= 1;
                        if (incr_u) begin
                            y_max <= y_max + 1;
                        end
                        if (incr_d) begin
                            y_max <= y_max - 1;
                        end
                        if (incr_l) begin
                            y_min <= y_min - 1;
                        end
                        if (incr_r) begin
                            y_min <= y_min + 1;
                        end
                        dispdata <= {4'd14, 8'b0, {2'b0,y_max}, {2'b0,y_min}};
                  end
            5'b1?000:begin  // set the hue bounds of the paddle
                        faster <= 0;
                        if (incr_u) begin
                            pad_hue_max <= pad_hue_max + 1;
                        end
                        if (incr_d) begin
                            pad_hue_max <= pad_hue_max - 1;
                        end
                        if (incr_l) begin
                            pad_hue_min <= pad_hue_min - 1;
                        end
                        if (incr_r) begin
                            pad_hue_min <= pad_hue_min + 1;
                        end
                        dispdata <= {4'd10, 8'b0, pad_hue_max, pad_hue_min};
                  end
            5'b1?001:begin  // set the saturation bounds of the paddle
                        faster <= 0;

                        if (incr_u) begin
                            pad_sat_max <= pad_sat_max + 1;
                        end
                        if (incr_d) begin
                            pad_sat_max <= pad_sat_max - 1;
                        end
                        if (incr_l) begin
                            pad_sat_min <= pad_sat_min - 1;
                        end
                        if (incr_r) begin
                            pad_sat_min <= pad_sat_min + 1;
                        end
                        dispdata <= {4'd11, 8'b0, pad_sat_max, pad_sat_min};
                  end
            5'b1?010:begin  // set the value bounds of the paddle
                        faster <= 0;

                        if (incr_u) begin
                            pad_val_max <= pad_val_max + 1;
                        end
                        if (incr_d) begin
                            pad_val_max <= pad_val_max - 1;
                        end
                        if (incr_l) begin
                            pad_val_min <= pad_val_min - 1;
                        end
                        if (incr_r) begin
                            pad_val_min <= pad_val_min + 1;
                        end
                        dispdata <= {4'd12, 8'b0, pad_val_max, pad_val_min};
                    end
            5'b0?101: begin  // adjust the location of the background image
                        faster <= 1;
                        if (incr_u) begin
                            y_offset <= y_offset + 1;
                        end
                        if (incr_d) begin
                            y_offset <= y_offset - 1;
                        end
                        if (incr_l) begin
                            x_offset <= x_offset - 1;
                        end
                        if (incr_r) begin
                            x_offset <= x_offset + 1;
                        end
                        dispdata <= {4'd12, 8'b0, x_offset, y_offset};
                        end
            default: begin
                        end
        endcase
    end

endmodule

//
////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////
//
// This module allows a button held down to continuously increment a register
// x times per second, rather than every cycle. This way we can hold down the
// buttons to adjust the parameters in a controlled fashion, instead of having
// to repeatedly press the button to increment the parameter.

module incrementor(input clk,
                   input trigger, // button input
                   output action, // signal to increment appropriate register
                   input fast);   // high means to increment quickly (HSV adjustment
 needs to be finer,
                                  // and thus increment slower, than adjusting the p
osition of the
                                  // background image or the x/y bounds of the play
area

    reg bang = 0;  // stores the current state: 1 means triggered, 0 means ready to
be triggered again
    reg action_reg = 0;  // stores the output state
    reg [30:0] bang_cooldown = 0; // saves the cycles since action was last triggere
d
    reg [30:0] cd;  // stores the cycles to wait before consecutive triggers of acti
on
    always @(posedge clk) begin
        if (fast) begin // wait less time if faster incrementation desired
            cd = 2375000;
        end else begin
            cd = 6750000;
        end
        if (!bang) begin // since buttons are active low, only raise action if bang=
0 (ready to trigger) and trigger=0 (button is pressed)
            if (!trigger) begin
                bang <= 1; // prevent it from triggering again consecutively
                action_reg <= 1; // raise action high
            end
        end else begin // this is the cooldown state, in which action is held low an
d cannot rise again until cooldown cycles have ellapsed or the button is released
            action_reg <= 0;
            if (trigger) begin  // if button released, we are allowed to trigger act
ion again
                bang <= 0;
                bang_cooldown <= 0;
            end else if (bang_cooldown > cd) begin // or if the cooldown has elapsed
```

```
                bang <= 0;
                bang_cooldown <= 0;
            end else begin  // otherwise increment the cooldown counter
                bang_cooldown <= bang_cooldown + 1;
            end
        end
    end
    assign action = action_reg;
endmodule

//
////////////////////////////////////////////////////////////////////////////////////
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Kevin Zheng Class of 2012
//           Dept of Electrical Engineering &  Computer Science
//
// Create Date:    18:45:01 11/10/2010
// Design Name:
// Module Name:    rgb2hsv
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module rgb2hsv(clock, reset, r, g, b, h, s, v);
        input wire clock;
        input wire reset;
        input wire [7:0] r;
        input wire [7:0] g;
        input wire [7:0] b;
        output reg [7:0] h;
        output reg [7:0] s;
        output reg [7:0] v;
        reg [7:0] my_r_delay1, my_g_delay1, my_b_delay1;
        reg [7:0] my_r_delay2, my_g_delay2, my_b_delay2;
        reg [7:0] my_r, my_g, my_b;
        reg [7:0] min, max, delta;
        reg [15:0] s_top;
        reg [15:0] s_bottom;
        reg [15:0] h_top;
        reg [15:0] h_bottom;
        wire [15:0] s_quotient;
        wire [15:0] s_remainder;
        wire s_rfd;
        wire [15:0] h_quotient;
        wire [15:0] h_remainder;
        wire h_rfd;
        reg [7:0] v_delay [19:0];
        reg [18:0] h_negative;
        reg [15:0] h_add [18:0];
        reg [4:0] i;
        // Clocks 4-18: perform all the divisions
        //the s_divider (16/16) has delay 18
        //the hue_div (16/16) has delay 18

        divider hue_div1(
        .clk(clock),
        .dividend(s_top),
        .divisor(s_bottom),
        .quotient(s_quotient),
            // note: the "fractional" output was originally named "remainder" in thi
s
        // file -- it seems coregen will name this output "fractional" even if
        // you didn't select the remainder type as fractional.
        .fractional(s_remainder),
        .rfd(s_rfd)
        );
        divider hue_div2(
        .clk(clock),
        .dividend(h_top),
        .divisor(h_bottom),
        .quotient(h_quotient),
        .fractional(h_remainder),
        .rfd(h_rfd)
        );
        always @ (posedge clock) begin
```

```
            // Clock 1: latch the inputs (always positive)
            {my_r, my_g, my_b} <= {r, g, b};

            // Clock 2: compute min, max
            {my_r_delay1, my_g_delay1, my_b_delay1} <= {my_r, my_g, my_b};

            if((my_r >= my_g) && (my_r >= my_b)) //(B,S,S)
                max <= my_r;
            else if((my_g >= my_r) && (my_g >= my_b)) //(S,B,S)
                max <= my_g;
            else    max <= my_b;

            if((my_r <= my_g) && (my_r <= my_b)) //(S,B,B)
                min <= my_r;
            else if((my_g <= my_r) && (my_g <= my_b)) //(B,S,B)
                min <= my_g;
            else
                min <= my_b;

            // Clock 3: compute the delta
            {my_r_delay2, my_g_delay2, my_b_delay2} <= {my_r_delay1, my_g_delay1, my
_b_delay1};
            v_delay[0] <= max;
            delta <= max - min;

            // Clock 4: compute the top and bottom of whatever divisions we need to
do
            s_top <= 8'd255 * delta;
            s_bottom <= (v_delay[0]>0)?{8'd0, v_delay[0]}: 16'd1;


            if(my_r_delay2 == v_delay[0]) begin
                h_top <= (my_g_delay2 >= my_b_delay2)?(my_g_delay2 - my_b_delay2) *
8'd255:(my_b_delay2 - my_g_delay2) * 8'd255;
                h_negative[0] <= (my_g_delay2 >= my_b_delay2)?0:1;
                h_add[0] <= 16'd0;
            end
            else if(my_g_delay2 == v_delay[0]) begin
                h_top <= (my_b_delay2 >= my_r_delay2)?(my_b_delay2 - my_r_delay2) *
8'd255:(my_r_delay2 - my_b_delay2) * 8'd255;
                h_negative[0] <= (my_b_delay2 >= my_r_delay2)?0:1;
                h_add[0] <= 16'd85;
            end
            else if(my_b_delay2 == v_delay[0]) begin
                h_top <= (my_r_delay2 >= my_g_delay2)?(my_r_delay2 - my_g_delay2) *
8'd255:(my_g_delay2 - my_r_delay2) * 8'd255;
                h_negative[0] <= (my_r_delay2 >= my_g_delay2)?0:1;
                h_add[0] <= 16'd170;
            end

            h_bottom <= (delta > 0)?delta * 8'd6:16'd6;


            //delay the v and h_negative signals 18 times
            for(i=1; i<19; i=i+1) begin
                v_delay[i] <= v_delay[i-1];
                h_negative[i] <= h_negative[i-1];
                h_add[i] <= h_add[i-1];
            end

            v_delay[19] <= v_delay[18];
            //Clock 22: compute the final value of h
            //depending on the value of h_delay[18], we need to subtract 255 from it
 to make it come back around the circle
            if(h_negative[18] && (h_quotient > h_add[18])) begin
                h <= 8'd255 - h_quotient[7:0] + h_add[18];
            end
            else if(h_negative[18]) begin
                h <= h_add[18] - h_quotient[7:0];
            end
            else begin
                h <= h_quotient[7:0] + h_add[18];
            end

            //pass out s and v straight
            s <= s_quotient;
            v <= v_delay[19];
        end
endmodule
```

```
// The synthesis directives "translate_off/translate_on" specified below are
// supported by Xilinx, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file rlut.v when simulating
// the core, rlut. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module rlut(
    clka,
    addra,
    douta);


input clka;
input [3 : 0] addra;
output [7 : 0] douta;

// synthesis translate_off

      BLK_MEM_GEN_V2_8 #(
        .C_ADDRA_WIDTH(4),
        .C_ADDRB_WIDTH(4),
        .C_ALGORITHM(1),
        .C_BYTE_SIZE(9),
        .C_COMMON_CLK(0),
        .C_DEFAULT_DATA("0"),
        .C_DISABLE_WARN_BHV_COLL(0),
        .C_DISABLE_WARN_BHV_RANGE(0),
        .C_FAMILY("virtex2"),
        .C_HAS_ENA(0),
        .C_HAS_ENB(0),
        .C_HAS_MEM_OUTPUT_REGS_A(0),
        .C_HAS_MEM_OUTPUT_REGS_B(0),
        .C_HAS_MUX_OUTPUT_REGS_A(0),
        .C_HAS_MUX_OUTPUT_REGS_B(0),
        .C_HAS_REGCEA(0),
        .C_HAS_REGCEB(0),
        .C_HAS_SSRA(0),
        .C_HAS_SSRB(0),
        .C_INIT_FILE_NAME("rlut.mif"),
        .C_LOAD_INIT_FILE(1),
        .C_MEM_TYPE(3),
        .C_MUX_PIPELINE_STAGES(0),
        .C_PRIM_TYPE(1),
        .C_READ_DEPTH_A(16),
        .C_READ_DEPTH_B(16),
        .C_READ_WIDTH_A(8),
        .C_READ_WIDTH_B(8),
        .C_SIM_COLLISION_CHECK("ALL"),
        .C_SINITA_VAL("0"),
        .C_SINITB_VAL("0"),
        .C_USE_BYTE_WEA(0),
        .C_USE_BYTE_WEB(0),
        .C_USE_DEFAULT_DATA(0),
        .C_USE_ECC(0),
        .C_USE_RAMB16BWER_RST_BHV(0),
        .C_WEA_WIDTH(1),
        .C_WEB_WIDTH(1),
        .C_WRITE_DEPTH_A(16),
        .C_WRITE_DEPTH_B(16),
        .C_WRITE_MODE_A("WRITE_FIRST"),
        .C_WRITE_MODE_B("WRITE_FIRST"),
        .C_WRITE_WIDTH_A(8),
        .C_WRITE_WIDTH_B(8),
        .C_XDEVICEFAMILY("virtex2"))
    inst (
        .CLKA(clka),
        .ADDRA(addra),
        .DOUTA(douta),
        .DINA(),
        .ENA(),
        .REGCEA(),
        .WEA(),
        .SSRA(),
        .CLKB(),
        .DINB(),
        .ADDRB(),
        .ENB(),
        .REGCEB(),
        .WEB(),
        .SSRB(),
        .DOUTB(),
        .DBITERR(),
        .SBITERR());


// synthesis translate_on

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of rlut is "black_box"

endmodule
```

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2007 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synthesis directives "translate_off/translate_on" specified below are
// supported by Xilinx, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file rom.v when simulating
// the core, rom. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module rom(
    clka,
    addra,
    douta);


input clka;
input [18 : 0] addra;
output [3 : 0] douta;

// synthesis translate_off

      BLK_MEM_GEN_V2_8 #(
        .C_ADDRA_WIDTH(19),
        .C_ADDRB_WIDTH(19),
        .C_ALGORITHM(1),
        .C_BYTE_SIZE(9),
        .C_COMMON_CLK(0),
        .C_DEFAULT_DATA("0"),
        .C_DISABLE_WARN_BHV_COLL(0),
        .C_DISABLE_WARN_BHV_RANGE(0),
        .C_FAMILY("virtex2"),
        .C_HAS_ENA(0),
        .C_HAS_ENB(0),
        .C_HAS_MEM_OUTPUT_REGS_A(0),
        .C_HAS_MEM_OUTPUT_REGS_B(0),
        .C_HAS_MUX_OUTPUT_REGS_A(0),
        .C_HAS_MUX_OUTPUT_REGS_B(0),
        .C_HAS_REGCEA(0),
        .C_HAS_REGCEB(0),
        .C_HAS_SSRA(0),
        .C_HAS_SSRB(0),
        .C_INIT_FILE_NAME("rom.mif"),
        .C_LOAD_INIT_FILE(1),
        .C_MEM_TYPE(3),
        .C_MUX_PIPELINE_STAGES(0),
        .C_PRIM_TYPE(1),
        .C_READ_DEPTH_A(524288),
        .C_READ_DEPTH_B(524288),
        .C_READ_WIDTH_A(4),
        .C_READ_WIDTH_B(4),
        .C_SIM_COLLISION_CHECK("ALL"),
        .C_SINITA_VAL("0"),
        .C_SINITB_VAL("0"),
        .C_USE_BYTE_WEA(0),
        .C_USE_BYTE_WEB(0),
        .C_USE_DEFAULT_DATA(0),
        .C_USE_ECC(0),
        .C_USE_RAMB16BWER_RST_BHV(0),
        .C_WEA_WIDTH(1),
        .C_WEB_WIDTH(1),
        .C_WRITE_DEPTH_A(524288),
        .C_WRITE_DEPTH_B(524288),
        .C_WRITE_MODE_A("WRITE_FIRST"),
        .C_WRITE_MODE_B("WRITE_FIRST"),
        .C_WRITE_WIDTH_A(4),
        .C_WRITE_WIDTH_B(4),
        .C_XDEVICEFAMILY("virtex2"))
    inst (
        .CLKA(clka),
        .ADDRA(addra),
        .DOUTA(douta),
        .DINA(),
        .ENA(),
        .REGCEA(),
        .WEA(),
        .SSRA(),
        .CLKB(),
        .DINB(),
        .ADDRB(),
        .ENB(),
        .REGCEB(),
        .WEB(),
        .SSRB(),
        .DOUTB(),
        .DBITERR(),
        .SBITERR());


// synthesis translate_on

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of rom is "black_box"

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//
// Mux the vga output pixel based on what is supposed to be displayed
//
// Nicholas Waltman and Mike Wang
//

module vga_select(input clk,
                  input reset,  // reset the game
                  input is_background,  // is this a background pixel or not
                  input [10:0] hcount, x_min, x_max,  // x bounds
                  input [9:0] vcount, y_max, y_min,  // y bounds of playing area
                  input [23:0] background,   // from the background generator
                  input [23:0] zbt,  // what we're getting from the zbt memory
                  input show_zbt,  // should we show the camera output or not
                  input replace_background,  // should we chroma key the background
or not
                  output [23:0] pixel,  // the pixel to display
                  output reg [10:0] ball_x_delta,
                  input paddle_px, // is this a paddle pixel?
                  output reg [9:0] ball_y_delta);

    // out of bounds colour
    parameter GRAY = 'b00001111_00001111_00001111;

    // ball generation
    reg [23:0] cur_px; // the pixel before going through the ball
    wire signed [12:0] ball_x;
    wire signed [12:0] ball_y;
    wire [7:0] contact;

    // state variables: has the game started, and if the game has ended
    reg start_game=0;
    reg [1:0] state = 0;

    // required info to delay the ball movement until 5 seconds after reset is press
ed, to give the player time
    // to reset the game and walk back into position
    parameter CLKPERSECOND = 'd60000000;
    parameter SECONDS = 5;
    reg [30:0] countdown = SECONDS*CLKPERSECOND;

    assign countdown_t = {init_y[3:0], init_x[3:0], 3'b0, ((state!=0)||(!start_game)
||(countdown!=0))?1:0, 1'b0, countdown};

    ball test_ball(.clk(clk), .hcount(hcount), .x_min(x_min), .x_max(x_max), .paddle
_px(paddle_px), .contact(contact),
                    .vcount(vcount), .y_min(y_min), .y_max(y_max), .cur_px(cur_px), .
new_px(pixel),
                    .x(ball_x), .y(ball_y), .reset(reset), .stop_moving(((state!=0)||
(!start_game)||(countdown!=0))?1:0), .new_vx('d5), .new_vy('d3));

    // paddle generation
    wire [23:0] paddle_colour;
    paddle_pixel paddle_display(hcount, (x_max+x_min)>>1, vcount, clk, paddle_colour
);

    // victory/defeat display
    wire [23:0] win_px, defeat_px, text_px;
    victory_gen win_bg(clk, hcount, (state==1) ? x_min:(x_max-'d100), vcount, y_min,
 win_px);
    defeat_gen defeat_bg(clk, hcount, (state==1) ? (x_max-'d100):x_min, vcount, y_mi
n, defeat_px);
    assign text_px = win_px|defeat_px;

    // health bar stuff
    reg signed [7:0] player_1_health = 100, player_2_health = 100;
    wire px_is_health1, px_is_health2;
    health_bar p1_health_bar(.clk(clk), .player_health(player_1_health), .x_min(x_mi
n), .x_max(x_max), .y_max(y_max),
                              .hcount(hcount), .vcount(vcount), .right_align(0), .is_
health(px_is_health1));
```

```
    health_bar p2_health_bar(.clk(clk), .player_health(player_2_health), .x_min(x_mi
n), .x_max(x_max), .y_max(y_max),
                             .hcount(hcount), .vcount(vcount), .right_align(1), .is_
health(px_is_health2));

    wire new_frame;
    assign new_frame = (hcount == 0 ) && (vcount == 0);

    wire out_of_bounds;
    assign out_of_bounds = hcount < x_min || hcount > x_max || vcount < y_min || vco
unt > y_max;

    always @(posedge clk) begin
        // output ball position data for the background_gen module
        ball_x_delta <= ball_x[10:0] - x_min;
        ball_y_delta <= ball_y[9:0] - y_min;

        // reset the game
        if (reset) begin
            player_1_health <= 100;
            player_2_health <= 100;
            state <= 0;
            countdown <= SECONDS*CLKPERSECOND;
            start_game <= 1;
        // check for changes to the game state
        end else if (new_frame && (!reset)) begin
            if ( (state == 0) && (player_2_health <= 0) )
                state <= 1;
            if ( (state == 0) && (player_1_health <= 0) )
                state <= 2;

            if (contact == 10)
                player_1_health <= player_1_health - 8;
            if (contact == 20)
                player_2_health <= player_2_health - 8;
        end else begin
            countdown <= (countdown > 0) ? countdown - 1: 0;
        end

        // mux the output pixel accordingly:
        /* if out of bounds show GRAY, else
            if text is being displayed show the text, else
            if the healthbar is supposed to be displayed show the health bar, else
            if the background image is supposed to be displayed show it, else
            if the real-world is supposed to be shown show it
        */
        cur_px <= out_of_bounds?GRAY:
                  (text_px!=0&&state!=0)? text_px:
                  paddle_px ? paddle_colour:
                  (px_is_health1 || px_is_health2) ? 'b11111111_10101010_00000000 :
                  (replace_background ? is_background : 0) ? background :
                  show_zbt ? zbt:0;
    end
endmodule

//
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////
//
// Simple module to display different colour paddles for each player

module paddle_pixel #(parameter PLAYER1 = 'b11111111_00000000_00000000,
                                PLAYER2 = 'b00000000_11111111_00000000)
                     (input [10:0] hcount, midpoint,
                      input [9:0] vcount,
                      input clk,
                      output reg [23:0] pixel_px);
    always @(posedge clk) begin
        pixel_px <= (hcount < midpoint) ? PLAYER1 : PLAYER2;
    end
endmodule
```

```
//
///////////////////////////////////////////////////////////////////////////////////////
```

```
//
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
//
// This file contains the ntsc_decode and adv7185init modules
//
// These modules are used to grab input NTSC video data from the RCA
// phono jack on the right hand side of the 6.111 labkit (connect
// the camera to the LOWER jack).
//

/////////////////////////////////////////////////////////////////////////////
//
// NTSC decode - 16-bit CCIR656 decoder
// By Javier Castro
// This module takes a stream of LLC data from the adv7185
// NTSC/PAL video decoder and generates the corresponding pixels,
// that are encoded within the stream, in YCrCb format.

// Make sure that the adv7185 is set to run in 16-bit LLC2 mode.

module ntsc_decode(clk, reset, tv_in_ycrcb, ycrcb, f, v, h, data_valid);

   // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
   // reset - system reset
   // tv_in_ycrcb - 10-bit input from chip. should map to pins [19:10]
   // ycrcb - 24 bit luminance and chrominance (8 bits each)
   // f - field: 1 indicates an even field, 0 an odd field
   // v - vertical sync: 1 means vertical sync
   // h - horizontal sync: 1 means horizontal sync

   input clk;
   input reset;
   input [9:0] tv_in_ycrcb; // modified for 10 bit input - should be P[19:10]
   output [29:0] ycrcb;
   output   f;
   output   v;
   output   h;
   output   data_valid;
   // output [4:0] state;

   parameter    SYNC_1 = 0;
   parameter    SYNC_2 = 1;
   parameter    SYNC_3 = 2;
   parameter    SAV_f1_cb0 = 3;
   parameter    SAV_f1_y0 = 4;
   parameter    SAV_f1_cr1 = 5;
   parameter    SAV_f1_y1 = 6;
   parameter    EAV_f1 = 7;
   parameter    SAV_VBI_f1 = 8;
   parameter    EAV_VBI_f1 = 9;
   parameter    SAV_f2_cb0 = 10;
   parameter    SAV_f2_y0 = 11;
   parameter    SAV_f2_cr1 = 12;
   parameter    SAV_f2_y1 = 13;
   parameter    EAV_f2 = 14;
   parameter    SAV_VBI_f2 = 15;
   parameter    EAV_VBI_f2 = 16;




   // In the start state, the module doesn't know where
   // in the sequence of pixels, it is looking.

   // Once we determine where to start, the FSM goes through a normal
   // sequence of SAV process_YCrCb EAV... repeat

   // The data stream looks as follows
   // SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | EAV
 sequence
   // There are two things we need to do:
```

```
   //   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
   //   2. Decode the subsequent data

   reg [4:0]    current_state = 5'h00;
   reg [9:0]    y = 10'h000;  // luminance
   reg [9:0]    cr = 10'h000; // chrominance
   reg [9:0]    cb = 10'h000; // more chrominance

   assign   state = current_state;

   always @ (posedge clk)
     begin
    if (reset)
      begin

      end
    else
      begin
         // these states don't do much except allow us to know where we are in the s
tream.
         // whenever the synchronization code is seen, go back to the sync_state bef
ore
         // transitioning to the new state
         case (current_state)
           SYNC_1: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_2 : SYNC_1;
           SYNC_2: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_3 : SYNC_1;
           SYNC_3: current_state <= (tv_in_ycrcb == 10'h200) ? SAV_f1_cb0 :
                    (tv_in_ycrcb == 10'h274) ? EAV_f1 :
                    (tv_in_ycrcb == 10'h2ac) ? SAV_VBI_f1 :
                    (tv_in_ycrcb == 10'h2d8) ? EAV_VBI_f1 :
                    (tv_in_ycrcb == 10'h31c) ? SAV_f2_cb0 :
                    (tv_in_ycrcb == 10'h368) ? EAV_f2 :
                    (tv_in_ycrcb == 10'h3b0) ? SAV_VBI_f2 :
                    (tv_in_ycrcb == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;

           SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y
0;
           SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cr
1;
           SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y
1;
           SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cb
0;

           SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y
0;
           SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cr
1;
           SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y
1;
           SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cb
0;

           // These states are here in the event that we want to cover these signals
           // in the future. For now, they just send the state machine back to SYNC_
1
           EAV_f1: current_state <= SYNC_1;
           SAV_VBI_f1: current_state <= SYNC_1;
           EAV_VBI_f1: current_state <= SYNC_1;
           EAV_f2: current_state <= SYNC_1;
           SAV_VBI_f2: current_state <= SYNC_1;
           EAV_VBI_f2: current_state <= SYNC_1;

         endcase
      end
     end // always @ (posedge clk)

   // implement our decoding mechanism

   wire y_enable;
   wire cr_enable;
   wire cb_enable;
   // if y is coming in, enable the register
   // likewise for cr and cb
   assign y_enable = (current_state == SAV_f1_y0) ||
                     (current_state == SAV_f1_y1) ||
                     (current_state == SAV_f2_y0) ||
                     (current_state == SAV_f2_y1);
   assign cr_enable = (current_state == SAV_f1_cr1) ||
                      (current_state == SAV_f2_cr1);
   assign cb_enable = (current_state == SAV_f1_cb0) ||
                      (current_state == SAV_f2_cb0);

   // f, v, and h only go high when active
   assign {v,h} = (current_state == SYNC_3) ? tv_in_ycrcb[7:6] : 2'b00;

   // data is valid when we have all three values: y, cr, cb
   assign data_valid = y_enable;
   assign ycrcb = {y,cr,cb};

   reg    f = 0;

   always @ (posedge clk)
     begin
    y <= y_enable ? tv_in_ycrcb : y;
    cr <= cr_enable ? tv_in_ycrcb : cr;
    cb <= cb_enable ? tv_in_ycrcb : cb;
    f <= (current_state == SYNC_3) ? tv_in_ycrcb[8] : f;
     end

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ADV7185 Video Decoder Configuration Init
//
// Created:
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
// Register 0
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SELECT                           4'h0
  // 0: CVBS on AIN1 (composite video in)
  // 7: Y on AIN2, C on AIN5 (s-video in)
  // (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE                             4'h0
  // 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
  // 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
  // 2: Autodetect: NTSC or PAL (N), w/o pedestal
  // 3: Autodetect: NTSC or PAL (N), w/pedestal
  // 4: NTSC w/o pedestal
  // 5: NTSC w/pedestal
  // 6: NTSC 4.43 w/o pedestal
  // 7: NTSC 4.43 w/pedestal
  // 8: PAL BGHID w/o pedestal
  // 9: PAL N w/pedestal
  // A: PAL M w/o pedestal
  // B: PAL M w/pedestal
  // C: PAL combination N
  // D: PAL combination N w/pedestal
  // E-F: [Not valid]

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 1
///////////////////////////////////////////////////////////////////////////////

`define VIDEO_QUALITY                          2'h0
```

```
  // 0: Broadcast quality
  // 1: TV quality
  // 2: VCR quality
  // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE                        1'b0
  // 0: Normal mode
  // 1: Square pixel mode
`define DIFFERENTIAL_INPUT                          1'b0
  // 0: Single-ended inputs
  // 1: Differential inputs
`define FOUR_TIMES_SAMPLING                         1'b0
  // 0: Standard sampling rate
  // 1: 4x sampling rate (NTSC only)
`define BETACAM                                     1'b0
  // 0: Standard video input
  // 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE                    1'b1
  // 0: Change of input triggers reacquire
  // 1: Change of input does not trigger reacquire

`define ADV7185_REGISTER_1 {`AUTOMATIC_STARTUP_ENABLE, 1'b0, `BETACAM, `FOUR_TIMES_S
AMPLING, `DIFFERENTIAL_INPUT, `SQUARE_PIXEL_IN_MODE, `VIDEO_QUALITY}

///////////////////////////////////////////////////////////////////////////////
// Register 2
///////////////////////////////////////////////////////////////////////////////

`define Y_PEAKING_FILTER                            3'h4
  // 0: Composite =  4.5dB,  s-video =  9.25dB
  // 1: Composite =  4.5dB,  s-video =  9.25dB
  // 2: Composite =  4.5dB,  s-video =  5.75dB
  // 3: Composite =  1.25dB, s-video =  3.3dB
  // 4: Composite =  0.0dB,  s-video =  0.0dB
  // 5: Composite = -1.25dB, s-video = -3.0dB
  // 6: Composite = -1.75dB, s-video = -8.0dB
  // 7: Composite = -3.0dB,  s-video = -8.0dB
`define CORING                                      2'h0
  // 0: No coring
  // 1: Truncate if Y < black+8
  // 2: Truncate if Y < black+16
  // 3: Truncate if Y < black+32

`define ADV7185_REGISTER_2 {3'b000, `CORING, `Y_PEAKING_FILTER}

///////////////////////////////////////////////////////////////////////////////
// Register 3
///////////////////////////////////////////////////////////////////////////////

`define INTERFACE_SELECT                            2'h0
  // 0: Philips-compatible
  // 1: Broktree API A-compatible
  // 2: Broktree API B-compatible
  // 3: [Not valid]
`define OUTPUT_FORMAT                               4'h0
  // 0: 10-bit @ LLC, 4:2:2 CCIR656
  // 1: 20-bit @ LLC, 4:2:2 CCIR656
  // 2: 16-bit @ LLC, 4:2:2 CCIR656
  // 3: 8-bit @ LLC, 4:2:2 CCIR656
  // 4: 12-bit @ LLC, 4:1:1
  // 5-F: [Not valid]
  // (Note that the 6.111 labkit hardware provides only a 10-bit interface to
  // the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS                     1'b0
  // 0: Drivers tristated when ~OE is high
  // 1: Drivers always tristated
`define VBI_ENABLE                                  1'b0
  // 0: Decode lines during vertical blanking interval
  // 1: Decode only active video regions

`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS, `OUTPUT_FORMAT, `
INTERFACE_SELECT}

///////////////////////////////////////////////////////////////////////////////
```

```
// Register 4
///////////////////////////////////////////////////////////////////////////////

`define OUTPUT_DATA_RANGE                         1'b0
  // 0: Output values restricted to CCIR-compliant range
  // 1: Use full output range
`define BT656_TYPE                                1'b0
  // 0: BT656-3-compatible
  // 1: BT656-4-compatible

`define ADV7185_REGISTER_4 {`BT656_TYPE, 3'b000, 3'b110, `OUTPUT_DATA_RANGE}

///////////////////////////////////////////////////////////////////////////////
// Register 5
///////////////////////////////////////////////////////////////////////////////


`define GENERAL_PURPOSE_OUTPUTS                   4'b0000
`define GPO_0_1_ENABLE                            1'b0
  // 0: General purpose outputs 0 and 1 tristated
  // 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE                            1'b0
  // 0: General purpose outputs 2 and 3 tristated
  // 1: General purpose outputs 2 and 3 enabled
`define BLANK_CHROMA_IN_VBI                       1'b1
  // 0: Chroma decoded and output during vertical blanking
  // 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE                              1'b0
  // 0: GPO 0 is a general purpose output
  // 1: GPO 0 shows HLOCK status

`define ADV7185_REGISTER_5 {`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI, `GPO_2_3_ENABLE, `G
PO_0_1_ENABLE, `GENERAL_PURPOSE_OUTPUTS}

///////////////////////////////////////////////////////////////////////////////
// Register 7
///////////////////////////////////////////////////////////////////////////////

`define FIFO_FLAG_MARGIN                          5'h10
  // Sets the locations where FIFO almost-full and almost-empty flags are set
`define FIFO_RESET                                1'b0
  // 0: Normal operation
  // 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET                      1'b0
  // 0: No automatic reset
  // 1: FIFO is autmatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME                       1'b1
  // 0: FIFO flags are synchronized to CLKIN
  // 1: FIFO flags are synchronized to internal 27MHz clock

`define ADV7185_REGISTER_7 {`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET, `FIFO_RESET
, `FIFO_FLAG_MARGIN}

///////////////////////////////////////////////////////////////////////////////
// Register 8
///////////////////////////////////////////////////////////////////////////////

`define INPUT_CONTRAST_ADJUST                           8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register 9
///////////////////////////////////////////////////////////////////////////////

`define INPUT_SATURATION_ADJUST                         8'h8C

`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register A
///////////////////////////////////////////////////////////////////////////////
```

```
`define INPUT_BRIGHTNESS_ADJUST                         8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register B
///////////////////////////////////////////////////////////////////////////////

`define INPUT_HUE_ADJUST                                8'h00

`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register C
///////////////////////////////////////////////////////////////////////////////

`define DEFAULT_VALUE_ENABLE                      1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values
`define DEFAULT_VALUE_AUTOMATIC_ENABLE            1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values if lock is lost
`define DEFAULT_Y_VALUE                           6'h0C
  // Default Y value

`define ADV7185_REGISTER_C {`DEFAULT_Y_VALUE, `DEFAULT_VALUE_AUTOMATIC_ENABLE, `DEFA
ULT_VALUE_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// Register D
///////////////////////////////////////////////////////////////////////////////

`define DEFAULT_CR_VALUE                          4'h8
  // Most-significant four bits of default Cr value
`define DEFAULT_CB_VALUE                          4'h8
  // Most-significant four bits of default Cb value

`define ADV7185_REGISTER_D {`DEFAULT_CB_VALUE, `DEFAULT_CR_VALUE}

///////////////////////////////////////////////////////////////////////////////
// Register E
///////////////////////////////////////////////////////////////////////////////

`define TEMPORAL_DECIMATION_ENABLE                1'b0
  // 0: Disable
  // 1: Enable
`define TEMPORAL_DECIMATION_CONTROL               2'h0
  // 0: Supress frames, start with even field
  // 1: Supress frames, start with odd field
  // 2: Supress even fields only
  // 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE                  4'h0
  // 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE, `TEMPORAL_DECIMATION_CO
NTROL, `TEMPORAL_DECIMATION_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// Register F
///////////////////////////////////////////////////////////////////////////////

`define POWER_SAVE_CONTROL                        2'h0
  // 0: Full operation
  // 1: CVBS only
  // 2: Digital only
  // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY                1'b0
  // 0: Power-down pin has priority
  // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                      1'b0
  // 0: Reference is functional
  // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR                  1'b0
```

```
  // 0: LLC generator is functional
  // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                           1'b0
  // 0: Chip is functional
  // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                          1'b0
  // 0: Normal operation
  // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP                                1'b0
  // 0: Normal operation
  // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP, `POWER
_DOWN_LLC_GENERATOR, `POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY, `POWER_SAVE
_CONTROL}

///////////////////////////////////////////////////////////////////////////////
// Register 33
///////////////////////////////////////////////////////////////////////////////

`define PEAK_WHITE_UPDATE                         1'b1
  // 0: Update gain once per line
  // 1: Update gain once per field
`define AVERAGE_BIRIGHTNESS_LINES                 1'b1
  // 0: Use lines 33 to 310
  // 1: Use lines 33 to 270
`define MAXIMUM_IRE                               3'h0
  // 0: PAL: 133, NTSC: 122
  // 1: PAL: 125, NTSC: 115
  // 2: PAL: 120, NTSC: 110
  // 3: PAL: 115, NTSC: 105
  // 4: PAL: 110, NTSC: 100
  // 5: PAL: 105, NTSC: 100
  // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL                                1'b1
  // 0: Disable color kill
  // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRE, `AVERAGE_BIRIGHT
NESS_LINES, `PEAK_WHITE_UPDATE}

`define ADV7185_REGISTER_10 8'h00
`define ADV7185_REGISTER_11 8'h00
`define ADV7185_REGISTER_12 8'h00
`define ADV7185_REGISTER_13 8'h45
`define ADV7185_REGISTER_14 8'h18
`define ADV7185_REGISTER_15 8'h60
`define ADV7185_REGISTER_16 8'h00
`define ADV7185_REGISTER_17 8'h01
`define ADV7185_REGISTER_18 8'h00
`define ADV7185_REGISTER_19 8'h10
`define ADV7185_REGISTER_1A 8'h10
`define ADV7185_REGISTER_1B 8'hF0
`define ADV7185_REGISTER_1C 8'h16
`define ADV7185_REGISTER_1D 8'h01
`define ADV7185_REGISTER_1E 8'h00
`define ADV7185_REGISTER_1F 8'h3D
`define ADV7185_REGISTER_20 8'hD0
`define ADV7185_REGISTER_21 8'h09
`define ADV7185_REGISTER_22 8'h8C
`define ADV7185_REGISTER_23 8'hE2
`define ADV7185_REGISTER_24 8'h1F
`define ADV7185_REGISTER_25 8'h07
`define ADV7185_REGISTER_26 8'hC2
`define ADV7185_REGISTER_27 8'h58
`define ADV7185_REGISTER_28 8'h3C
`define ADV7185_REGISTER_29 8'h00
`define ADV7185_REGISTER_2A 8'h00
`define ADV7185_REGISTER_2B 8'hA0
`define ADV7185_REGISTER_2C 8'hCE
`define ADV7185_REGISTER_2D 8'hF0
`define ADV7185_REGISTER_2E 8'h00
`define ADV7185_REGISTER_2F 8'hF0
```

```
`define ADV7185_REGISTER_30 8'h00
`define ADV7185_REGISTER_31 8'h70
`define ADV7185_REGISTER_32 8'h00
`define ADV7185_REGISTER_34 8'h0F
`define ADV7185_REGISTER_35 8'h01
`define ADV7185_REGISTER_36 8'h00
`define ADV7185_REGISTER_37 8'h00
`define ADV7185_REGISTER_38 8'h00
`define ADV7185_REGISTER_39 8'h00
`define ADV7185_REGISTER_3A 8'h00
`define ADV7185_REGISTER_3B 8'h00

`define ADV7185_REGISTER_44 8'h41
`define ADV7185_REGISTER_45 8'hBB

`define ADV7185_REGISTER_F1 8'hEF
`define ADV7185_REGISTER_F2 8'h80


module adv7185init (reset, clock_27mhz, source, tv_in_reset_b,
            tv_in_i2c_clock, tv_in_i2c_data);

   input reset;
   input clock_27mhz;
   output tv_in_reset_b; // Reset signal to ADV7185
   output tv_in_i2c_clock; // I2C clock output to ADV7185
   output tv_in_i2c_data; // I2C data line to ADV7185
   input source; // 0: composite, 1: s-video

   initial begin
      $display("ADV7185 Initialization values:");
      $display("  Register 0:  0x%X", `ADV7185_REGISTER_0);
      $display("  Register 1:  0x%X", `ADV7185_REGISTER_1);
      $display("  Register 2:  0x%X", `ADV7185_REGISTER_2);
      $display("  Register 3:  0x%X", `ADV7185_REGISTER_3);
      $display("  Register 4:  0x%X", `ADV7185_REGISTER_4);
      $display("  Register 5:  0x%X", `ADV7185_REGISTER_5);
      $display("  Register 7:  0x%X", `ADV7185_REGISTER_7);
      $display("  Register 8:  0x%X", `ADV7185_REGISTER_8);
      $display("  Register 9:  0x%X", `ADV7185_REGISTER_9);
      $display("  Register A:  0x%X", `ADV7185_REGISTER_A);
      $display("  Register B:  0x%X", `ADV7185_REGISTER_B);
      $display("  Register C:  0x%X", `ADV7185_REGISTER_C);
      $display("  Register D:  0x%X", `ADV7185_REGISTER_D);
      $display("  Register E:  0x%X", `ADV7185_REGISTER_E);
      $display("  Register F:  0x%X", `ADV7185_REGISTER_F);
      $display("  Register 33: 0x%X", `ADV7185_REGISTER_33);
   end

   //
   // Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)
   //

   reg [7:0] clk_div_count, reset_count;
   reg clock_slow;
   wire reset_slow;

   initial
     begin
    clk_div_count <= 8'h00;
    // synthesis attribute init of clk_div_count is "00";
    clock_slow <= 1'b0;
    // synthesis attribute init of clock_slow is "0";
     end

   always @(posedge clock_27mhz)
     if (clk_div_count == 26)
       begin
      clock_slow <= ~clock_slow;
      clk_div_count <= 0;
       end
     else
       clk_div_count <= clk_div_count+1;
   always @(posedge clock_27mhz)
     if (reset)
       reset_count <= 100;
     else
       reset_count <= (reset_count==0) ? 0 : reset_count-1;

   assign reset_slow = reset_count != 0;

   //
   // I2C driver
   //

   reg load;
   reg [7:0] data;
   wire ack, idle;

   i2c i2c(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
       .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
       .sda(tv_in_i2c_data));

   //
   // State machine
   //

   reg [7:0] state;
   reg tv_in_reset_b;
   reg old_source;

   always @(posedge clock_slow)
      if (reset_slow)
    begin
       state <= 0;
       load <= 0;
       tv_in_reset_b <= 0;
       old_source <= 0;
    end
      else
    case (state)
      8'h00:
        begin
           // Assert reset
           load <= 1'b0;
           tv_in_reset_b <= 1'b0;
           if (!ack)
        state <= state+1;
        end
      8'h01:
        state <= state+1;
      8'h02:
        begin
           // Release reset
           tv_in_reset_b <= 1'b1;
           state <= state+1;
                end
      8'h03:
        begin
           // Send ADV7185 address
           data <= 8'h8A;
           load <= 1'b1;
           if (ack)
        state <= state+1;
        end
      8'h04:
        begin
           // Send subaddress of first register
           data <= 8'h00;
           if (ack)
        state <= state+1;
        end
      8'h05:
        begin
           // Write to register 0
           data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
           if (ack)
         state <= state+1;
        end
      8'h06:
        begin
           // Write to register 1
           data <= `ADV7185_REGISTER_1;
           if (ack)
         state <= state+1;
        end
      8'h07:
        begin
           // Write to register 2
           data <= `ADV7185_REGISTER_2;
           if (ack)
         state <= state+1;
        end
      8'h08:
        begin
           // Write to register 3
           data <= `ADV7185_REGISTER_3;
           if (ack)
         state <= state+1;
        end
      8'h09:
        begin
           // Write to register 4
           data <= `ADV7185_REGISTER_4;
           if (ack)
         state <= state+1;
        end
      8'h0A:
        begin
           // Write to register 5
           data <= `ADV7185_REGISTER_5;
           if (ack)
         state <= state+1;
        end
      8'h0B:
        begin
           // Write to register 6
           data <= 8'h00; // Reserved register, write all zeros
           if (ack)
         state <= state+1;
        end
      8'h0C:
        begin
           // Write to register 7
           data <= `ADV7185_REGISTER_7;
           if (ack)
         state <= state+1;
        end
      8'h0D:
        begin
           // Write to register 8
           data <= `ADV7185_REGISTER_8;
           if (ack)
         state <= state+1;
        end
      8'h0E:
        begin
           // Write to register 9
           data <= `ADV7185_REGISTER_9;
           if (ack)
         state <= state+1;
        end
      8'h0F: begin
         // Write to register A
         data <= `ADV7185_REGISTER_A;
       if (ack)
         state <= state+1;
      end
      8'h10:
        begin
           // Write to register B
           data <= `ADV7185_REGISTER_B;
           if (ack)
         state <= state+1;
        end
      8'h11:
        begin
           // Write to register C
           data <= `ADV7185_REGISTER_C;
           if (ack)
         state <= state+1;
        end
      8'h12:
        begin
           // Write to register D
           data <= `ADV7185_REGISTER_D;
           if (ack)
         state <= state+1;
        end
      8'h13:
        begin
           // Write to register E
           data <= `ADV7185_REGISTER_E;
           if (ack)
         state <= state+1;
        end
      8'h14:
        begin
           // Write to register F
           data <= `ADV7185_REGISTER_F;
           if (ack)
         state <= state+1;
        end
      8'h15:
        begin
           // Wait for I2C transmitter to finish
           load <= 1'b0;
           if (idle)
         state <= state+1;
        end
      8'h16:
        begin
           // Write address
           data <= 8'h8A;
           load <= 1'b1;
           if (ack)
         state <= state+1;
        end
      8'h17:
        begin
           data <= 8'h33;
           if (ack)
         state <= state+1;
        end
      8'h18:
        begin
           data <= `ADV7185_REGISTER_33;
           if (ack)
         state <= state+1;
        end
      8'h19:
        begin
           load <= 1'b0;
           if (idle)
         state <= state+1;
        end

      8'h1A: begin
         data <= 8'h8A;
         load <= 1'b1;
         if (ack)
```

```
             state <= state+1;
         end
         8'h1B:
           begin
              data <= 8'h33;
              if (ack)
            state <= state+1;
           end
         8'h1C:
           begin
              load <= 1'b0;
              if (idle)
            state <= state+1;
           end
         8'h1D:
           begin
              load <= 1'b1;
              data <= 8'h8B;
              if (ack)
            state <= state+1;
           end
         8'h1E:
           begin
              data <= 8'hFF;
              if (ack)
            state <= state+1;
           end
         8'h1F:
           begin
              load <= 1'b0;
              if (idle)
            state <= state+1;
           end
         8'h20:
           begin
              // Idle
              if (old_source != source) state <= state+1;
              old_source <= source;
           end
         8'h21: begin
            // Send ADV7185 address
            data <= 8'h8A;
            load <= 1'b1;
            if (ack) state <= state+1;
         end
         8'h22: begin
            // Send subaddress of register 0
            data <= 8'h00;
            if (ack) state <= state+1;
         end
         8'h23: begin
            // Write to register 0
            data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
            if (ack) state <= state+1;
         end
         8'h24: begin
            // Wait for I2C transmitter to finish
            load <= 1'b0;
            if (idle) state <= 8'h20;
         end
          endcase

endmodule

// i2c module for use with the ADV7185

module i2c (reset, clock4x, data, load, idle, ack, scl, sda);

   input reset;
   input clock4x;
   input [7:0] data;
   input load;
   output ack;
   output idle;
   output scl;
   output sda;

   reg [7:0] ldata;
   reg ack, idle;
   reg scl;
   reg sdai;

   reg [7:0] state;

   assign sda = sdai ? 1'bZ : 1'b0;

   always @(posedge clock4x)
     if (reset)
       begin
      state <= 0;
      ack <= 0;
       end
     else
       case (state)
     8'h00: // idle
       begin
          scl <= 1'b1;
          sdai <= 1'b1;
          ack <= 1'b0;
          idle <= 1'b1;
          if (load)
        begin
           ldata <= data;
           ack <= 1'b1;
           state <= state+1;
        end
       end
     8'h01: // Start
       begin
          ack <= 1'b0;
          idle <= 1'b0;
          sdai <= 1'b0;
          state <= state+1;
       end
     8'h02:
       begin
          scl <= 1'b0;
          state <= state+1;
       end
     8'h03: // Send bit 7
       begin
          ack <= 1'b0;
          sdai <= ldata[7];
          state <= state+1;
       end
     8'h04:
       begin
          scl <= 1'b1;
          state <= state+1;
       end
     8'h05:
       begin
          state <= state+1;
       end
     8'h06:
       begin
          scl <= 1'b0;
          state <= state+1;
       end
     8'h07:
       begin
          sdai <= ldata[6];
          state <= state+1;
       end
     8'h08:
       begin
```

```
            scl <= 1'b1;
            state <= state+1;
         end
       8'h09:
         begin
            state <= state+1;
         end
       8'h0A:
         begin
            scl <= 1'b0;
            state <= state+1;
         end
       8'h0B:
         begin
            sdai <= ldata[5];
            state <= state+1;
         end
       8'h0C:
         begin
            scl <= 1'b1;
            state <= state+1;
         end
       8'h0D:
         begin
            state <= state+1;
         end
       8'h0E:
         begin
            scl <= 1'b0;
            state <= state+1;
         end
       8'h0F:
         begin
            sdai <= ldata[4];
            state <= state+1;
         end
       8'h10:
         begin
            scl <= 1'b1;
            state <= state+1;
         end
       8'h11:
         begin
            state <= state+1;
         end
       8'h12:
         begin
            scl <= 1'b0;
            state <= state+1;
         end
       8'h13:
         begin
            sdai <= ldata[3];
            state <= state+1;
         end
       8'h14:
         begin
            scl <= 1'b1;
            state <= state+1;
         end
       8'h15:
         begin
            state <= state+1;
         end
       8'h16:
         begin
            scl <= 1'b0;
            state <= state+1;
         end
       8'h17:
         begin
            sdai <= ldata[2];
            state <= state+1;
```

```
        end
      8'h18:
        begin
           scl <= 1'b1;
           state <= state+1;
        end
      8'h19:
        begin
           state <= state+1;
        end
      8'h1A:
        begin
           scl <= 1'b0;
           state <= state+1;
        end
      8'h1B:
        begin
           sdai <= ldata[1];
           state <= state+1;
        end
      8'h1C:
        begin
           scl <= 1'b1;
           state <= state+1;
        end
      8'h1D:
        begin
           state <= state+1;
        end
      8'h1E:
        begin
           scl <= 1'b0;
           state <= state+1;
        end
      8'h1F:
        begin
           sdai <= ldata[0];
           state <= state+1;
        end
      8'h20:
        begin
           scl <= 1'b1;
           state <= state+1;
        end
      8'h21:
        begin
           state <= state+1;
        end
      8'h22:
        begin
           scl <= 1'b0;
           state <= state+1;
        end
      8'h23: // Acknowledge bit
        begin
           state <= state+1;
        end
      8'h24:
        begin
           scl <= 1'b1;
           state <= state+1;
        end
      8'h25:
        begin
           state <= state+1;
        end
      8'h26:
        begin
           scl <= 1'b0;
           if (load)
         begin
            ldata <= data;
            ack <= 1'b1;
            state <= 3;
         end
           else
         state <= state+1;
        end
      8'h27:
        begin
           sdai <= 1'b0;
           state <= state+1;
        end
      8'h28:
        begin
           scl <= 1'b1;
           state <= state+1;
        end
      8'h29:
        begin
           sdai <= 1'b1;
           state <= 0;
        end
        endcase

endmodule
```

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:34:01 11/16/2017
// Design Name:
// Module Name:    ycrcb
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
/**************************************************************************
 **
 ** Module: ycrcb2rgb
 **
 ** Generic Equations:
 ***************************************************************************/

module YCrCb2RGB ( R, G, B, clk, rst, Y, Cr, Cb);

output [7:0]  R, G, B;

input clk,rst;
input[9:0] Y, Cr, Cb;

wire [7:0] R,G,B;
reg [20:0] R_int,G_int,B_int,X_int,A_int,B1_int,B2_int,C_int;
reg [9:0] const1,const2,const3,const4,const5;
reg[9:0] Y_reg, Cr_reg, Cb_reg;

//registering constants
always @ (posedge clk)
begin
 const1 = 10'b 0100101010; //1.164 = 01.00101010
 const2 = 10'b 0110011000; //1.596 = 01.10011000
 const3 = 10'b 0011010000; //0.813 = 00.11010000
 const4 = 10'b 0001100100; //0.392 = 00.01100100
 const5 = 10'b 1000000100; //2.017 = 10.00000100
end

always @ (posedge clk or posedge rst)

   if (rst)
      begin
      Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
      end
   else
      begin
       Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
      end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
       A_int <= 0; B1_int <= 0; B2_int <= 0; C_int <= 0; X_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     A_int <= (const2 * (Cr_reg - 'd512));
     B1_int <= (const3 * (Cr_reg - 'd512));
     B2_int <= (const4 * (Cb_reg - 'd512));
     C_int <= (const5 * (Cb_reg - 'd512));
     end
```

```
always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     R_int <= X_int + A_int;
     G_int <= X_int - B1_int - B2_int;
     B_int <= X_int + C_int;
     end



/*always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     R_int <= X_int + (const2 * (Cr_reg - 'd512));
     G_int <= X_int - (const3 * (Cr_reg - 'd512)) - (const4 * (Cb_reg - 'd512));
     B_int <= X_int + (const5 * (Cb_reg - 'd512));
     end

*/
/* limit output to 0 - 4095, <0 equals o and >4095 equals 4095 */
assign R = (R_int[20]) ? 0 : (R_int[19:18] == 2'b0) ? R_int[17:10] : 8'b11111111;
assign G = (G_int[20]) ? 0 : (G_int[19:18] == 2'b0) ? G_int[17:10] : 8'b11111111;
assign B = (B_int[20]) ? 0 : (B_int[19:18] == 2'b0) ? B_int[17:10] : 8'b11111111;

endmodule
```

```
`default_nettype none
//¿
// File:   zbt_6111_sample.v¿
// Date:   26-Nov-05¿
// Author: I. Chuang <ichuang@mit.edu>¿
//¿
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT¿
// memories for video display.  Video input from the NTSC digitizer is¿
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used¿
// as the video frame buffer, with 8 bits used per pixel (black & white).¿
//¿
// Since the ZBT is read once for every four pixels, this frees up time for ¿
// data to be stored to the ZBT during other pixel times.  The NTSC decoder¿
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize¿
// signals between the two (see ntsc2zbt.v) and let the NTSC data be¿
// stored to ZBT memory whenever it is available, during cycles when¿
// pixel reads are not being performed.¿
//¿
// We use a very simple ZBT interface, which does not involve any clock¿
// generation or hiding of the pipelining.  See zbt_6111.v for more info.¿
//¿
// switch[7] selects between display of NTSC video and test bars¿
// switch[6] is used for testing the NTSC decoder¿
// switch[1] selects between test bar periods; these are stored to ZBT¿
//           during blanking periods¿
// switch[0] selects vertical test bars (hardwired; not stored in ZBT)¿
//¿
//¿
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>¿
// Date   : 11-May-09¿
//¿
// Use ramclock module to deskew clocks;  GPH¿
// To change display from 1024*787 to 800*600, use clock_40mhz and change¿
// accordingly. Verilog ntsc2zbt.v will also need changes to change resolution.¿
//¿
// Date   : 10-Nov-11¿
¿
///////////////////////////////////////////////////////////////////////////////¿
//¿
// 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_ycrcb" 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)¿
//¿
// 2011-Nov-10: Changed resolution to 1024 * 768.¿
//                   Added back ramclok to deskew RAM clock¿
//¿
// 2009-May-11: Fixed memory management bug by 8 clock cycle forecast. ¿
//              Changed resolution to  800 * 600.¿
//              Reduced clock speed to 40MHz.¿
//              Disconnected zbt_6111's ram_clk signal. ¿
//              Added ramclock to control RAM.¿
//              Added notes about ram1 default values.¿
//              Commented out clock_feedback_out assignment.¿
//              Removed delayN modules because ZBT's latency has no more effect.¿
//¿
// 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 zbt_6111_sample(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_ycrcb;¿
   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_ycrcb;¿
   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;¿
   input  flash_sts;¿
   ¿
   output rs232_txd, rs232_rts;¿
   input  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;  ¿
   input  disp_data_in;¿
   output  disp_data_out;¿
   ¿
   input  button0, button1, button2, button3, button_enter, button_right,¿
      button_left, button_down, button_up;¿
   input  [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;¿
   input  systemace_irq, systemace_mpbrdy;¿
¿
   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data, ¿
         analyzer4_data;¿
   output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;¿
¿
   ////////////////////////////////////////////////////////////////////////////¿
   //¿
   // I/O Assignments¿
   //¿
   ////////////////////////////////////////////////////////////////////////////¿
   ¿
   // 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'b1;¿
   assign tv_in_fifo_clock = 1'b0;¿
   assign tv_in_iso = 1'b1;¿
   //assign tv_in_reset_b = 1'b0;¿
   assign tv_in_clock = clock_27mhz;//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¿
¿
/* change lines below to enable ZBT RAM bank0 */¿
¿
/*¿
   assign ram0_data = 36'hZ;¿
   assign ram0_address = 19'h0;¿
   assign ram0_clk = 1'b0;¿
   assign ram0_we_b = 1'b1;¿
   assign ram0_cen_b = 1'b0;    // clock enable¿
*/¿
¿
/* enable RAM pins */¿
¿
   assign ram0_ce_b = 1'b0;¿
   assign ram0_oe_b = 1'b0;¿
   assign ram0_adv_ld = 1'b0;¿
   assign ram0_bwe_b = 4'h0; ¿
¿
/**********/¿
¿
```

```
   assign ram1_data = 36'hZ; ¿
   assign ram1_address = 19'h0;¿
   assign ram1_adv_ld = 1'b0;¿
   assign ram1_clk = 1'b0;¿
   ¿
   //These values has to be set to 0 like ram0 if ram1 is used.¿
   assign ram1_cen_b = 1'b1;¿
   assign ram1_ce_b = 1'b1;¿
   assign ram1_oe_b = 1'b1;¿
   assign ram1_we_b = 1'b1;¿
   assign ram1_bwe_b = 4'hF;¿
¿
   // clock_feedback_out will be assigned by ramclock¿
   // assign clock_feedback_out = 1'b0;  //2011-Nov-10¿
   // 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;WARNING:HDLCompilers:299 - "green_screen_test.v" line 31 To
o many digits specified in binary constant
¿
   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;¿
                ¿
   ////////////////////////////////////////////////////////////////////////////¿
   // Demonstration of ZBT RAM as video memory¿
¿
   // 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));¿
¿
//   wire clk = clock_65mhz;  // gph 2011-Nov-10¿
¿
/*   ////////////////////////////////////////////////////////////////////////////¿
   // Demonstration of ZBT RAM as video memory¿
¿
   // use FPGA's digital clock manager to produce a¿
   // 40MHz clock (actually 40.5MHz)¿
   wire clock_40mhz_unbuf,clock_40mhz;¿
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));¿
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 2¿
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 3¿
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE¿
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37¿
   BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));¿
¿
   wire clk = clock_40mhz;¿
*/¿
    wire locked;
    wire clk;¿
    //assign clock_feedback_out = 0; // gph 2011-Nov-10¿
   ¿
   ramclock rc(.ref_clock(clock_65mhz), .fpga_clock(clk),¿
                    .ram0_clock(ram0_clk), ¿
                    //.ram1_clock(ram1_clk),   //uncomment if ram1 is used¿
                    .clock_feedback_in(clock_feedback_in),¿
                    .clock_feedback_out(clock_feedback_out), .locked(locked));¿
¿
   ¿
   // power-on reset generation¿
   wire power_on_reset;    // remain high for first 16 clocks¿
   SRL16 reset_sr (.D(1'b0), .CLK(clk), .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, clk, ~button_enter, user_reset);¿
   assign reset = user_reset | power_on_reset;¿
¿
   // display module for debugging¿
¿
   reg [63:0] dispdata;¿
   display_16hex hexdisp1(reset, clk, dispdata,¿
              disp_blank, disp_clock, disp_rs, disp_ce_b,¿
              disp_reset_b, disp_data_out);¿
¿
   // generate basic XVGA video signals¿
   wire [10:0] hcount;¿
   wire [9:0]  vcount;¿
   wire hsync,vsync,blank;¿
   xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);¿
¿
   // wire up to ZBT ram¿
¿
```

```
   wire [35:0] vram_write_data;¿
   wire [35:0] vram_read_data;¿
   wire [18:0] vram_addr;¿
   wire        vram_we;¿

    ¿
   wire ram0_clk_not_used;¿
   zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,¿
           vram_write_data, vram_read_data,¿
           ram0_clk_not_used,   //to get good timing, don't connect ram_clk to zbt_6
111¿
           ram0_we_b, ram0_address, ram0_data, ram0_cen_b);¿
            ¿
   // generate pixel value from reading ZBT memory¿
   wire [17:0]  vr_pixel;¿
   wire [18:0]  vram_addr1;
¿
   vram_display vd1(reset,clk,hcount,vcount,vr_pixel,¿
            vram_addr1,vram_read_data);¿
¿
   // ADV7185 NTSC decoder interface code¿
   // adv7185 initialization module¿
   adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz), ¿
               .source(1'b0), .tv_in_reset_b(tv_in_reset_b), ¿
               .tv_in_i2c_clock(tv_in_i2c_clock), ¿
               .tv_in_i2c_data(tv_in_i2c_data));¿
¿
   wire [29:0] ycrcb;   // video data (luminance, chrominance)¿
   wire [2:0] fvh;  // sync for field, vertical, horizontal¿
   wire       dv;   // data valid¿
   wire [7:0] red, green, blue;¿
   ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),¿
               .tv_in_ycrcb(tv_in_ycrcb[19:10]), ¿
               .ycrcb(ycrcb), .f(fvh[2]),¿
               .v(fvh[1]), .h(fvh[0]), .data_valid(dv));¿

    YCrCb2RGB rgb( .R(red), .G(green), .B(blue), .clk(tv_in_line_clock1),
                     .rst(1'b0), .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]),
                          .Cb(ycrcb[9:0]) );


¿
   // code to write NTSC data to video memory¿
¿
   wire [18:0] ntsc_addr;¿
   wire [35:0] ntsc_data;¿
   wire ntsc_we;
¿
   ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, {red[7:2], green[7:2], blue[7:2
]}, ¿
            ntsc_addr, ntsc_data, ntsc_we, 1'b0);
¿
   // code to write pattern to ZBT memory¿
   reg [31:0]   count;¿
   always @(posedge clk) count <= reset ? 0 : count + 1;¿
¿
   wire [18:0]  vram_addr2 = count[0+18:0];¿
   wire [35:0]  vpat = ( switch[1] ? {  4{count[3+3:3],4'b0}}¿
              : {4{count[3+4:4],4'b0}} );¿
¿
   // mux selecting read/write to memory based on which write-enable is chosen¿
¿
   wire     sw_ntsc = 1'b1;¿
   wire     my_we = sw_ntsc ? (hcount[0]==1'b1) : blank; // mw changed hcount parame
trs¿
   wire [18:0]  write_addr = sw_ntsc ? ntsc_addr : vram_addr2;¿
   wire [35:0]  write_data = sw_ntsc ? ntsc_data : 36'b0;¿
¿
//   wire   write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;¿
//   assign     vram_addr = write_enable ? write_addr : vram_addr1;¿
//   assign     vram_we = write_enable;¿
¿
   assign   vram_addr = my_we ? write_addr : vram_addr1;¿
   assign   vram_we = my_we;¿
   assign   vram_write_data = write_data;¿
¿
   // select output pixel data¿
¿
   reg [17:0]   pixel;¿
   reg  b,hs,vs;¿
   ¿
   always @(posedge clk)¿
     begin¿
    pixel <=  vr_pixel;¿
    b <= blank;¿
    hs <= hsync;¿
    vs <= vsync;¿
     end

    wire u, d, l, r;

    debounce cleanup(.reset(reset), .clk(clk), .noisy(button_up), .clean(u));
    debounce cleandown(.reset(reset), .clk(clk), .noisy(button_down), .clean(d));
    debounce cleanleft(.reset(reset), .clk(clk), .noisy(button_left), .clean(l));
    debounce cleanright(.reset(reset), .clk(clk), .noisy(button_right), .clean(r));

    wire [7:0] hue_max, hue_min, sat_max, sat_min, val_max, val_min;
    wire [7:0] pad_hue_max, pad_hue_min, pad_sat_max, pad_sat_min, pad_val_max, pad_
val_min;
    wire [10:0] x_min, x_max;
    wire [9:0] y_min, y_max;
    wire [9:0] y_offset;
    wire [10:0] x_offset;

    wire [63:0] display_data;
    parameter_select parameter_set(.clk(clk), .switch(switch[4:0]), .u(u), .d(d), .l
(l), .r(r),
                      .hue_max(hue_max), .hue_min(hue_min), .sat_max(sat_max),
                     .sat_min(sat_min), .val_max(val_max), .val_min(val_min),
                            .x_min(x_min), .x_max(x_max),
                            .y_min(y_min), .y_max(y_max), .dispdata(display_data),
                            .pad_hue_max(pad_hue_max), .pad_hue_min(pad_hue_min),
                            .pad_sat_max(pad_sat_max), .pad_sat_min(pad_sat_min),
                            .pad_val_max(pad_val_max), .pad_val_min(pad_val_min),
                            .x_offset(x_offset), .y_offset(y_offset));

    // VGA Output.  In order to meet the setup and hold times of the¿
   // AD7125, we send it ~clk.

    wire [7:0] redp = {pixel[17:12],2'b0};¿
   wire [7:0] greenp = {pixel[11:6],2'b0};¿
   wire [7:0] bluep = {pixel[5:0],2'b0};

    wire [7:0] delayed_r, delayed_g, delayed_b;

    delay delayr(.clock(clk), .din(redp), .dout(delayed_r));
    delay delayg(.clock(clk), .din(greenp), .dout(delayed_g));
    delay delayb(.clock(clk), .din(bluep), .dout(delayed_b));

    wire [7:0] screenr, screeng, screenb;
    wire [10:0] ball_x;
    wire [9:0] ball_y;

    background_gen test(.clk(clk), .r(screenr), .g(screeng), .b(screenb), .vcount(vc
ount), .hcount(hcount),
                       .x_offset(x_offset), .ball_track_x(ball_x), .ball_track_y(ba
ll_y), .y_offset(y_offset),
                             .mode(switch[3]), .x_max(x_max), .x_min(x_min), .y_min
(y_min));

    wire is_background, is_paddle;
    binarizer detect(.clk(clk), .r(redp), .g(greenp), .b(bluep),
                     .hcount(hcount), .vcount(vcount),
                     .hue_max(hue_max), .hue_min(hue_min),
                     .sat_max(sat_max), .sat_min(sat_min),
                     .val_max(val_max), .val_min(val_min),
                     .pad_hue_max(pad_hue_max), .pad_hue_min(pad_hue_min),
                    .pad_sat_max(pad_sat_max), .pad_sat_min(pad_sat_min),
                    .pad_val_max(pad_val_max), .pad_val_min(pad_val_min),
                     .is_background(is_background), .is_paddle(is_paddle),
                     .activate_kernel(switch[7]));


    wire [23:0] vga_out;
    vga_select vga_selector(clk, ~button3, is_background, hcount, x_min, x_max,
                               vcount, y_max, y_min, {screenr, screeng, screenb},
                               {delayed_r, delayed_g, delayed_b}, switch[6], swit
ch[5],
                                  vga_out, ball_x, is_paddle, ball_y);

    ¿
   assign vga_out_red = vga_out[23:16];¿
   assign vga_out_green = vga_out[15:8];¿
   assign vga_out_blue = vga_out[7:0];
    ¿
   assign vga_out_sync_b = 1'b1;    // not used¿
   assign vga_out_pixel_clock = ~clk;¿
   assign vga_out_blank_b = ~b;¿
   assign vga_out_hsync = hs;¿
   assign vga_out_vsync = vs;¿
¿
   // debugging¿
   ¿
   assign led = ~{u,d,l,r,reset,switch[0]};¿
¿
   always @(posedge clk)¿
       dispdata <= display_data;
¿
endmodule¿
¿
////////////////////////////////////////////////////////////////////////////////¿
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)¿
¿
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);¿
   assign    vsyncoff = hreset & (vcount == 782);¿
   assign    vreset = hreset & (vcount == 805);¿
¿
   // sync and blanking¿
   wire      next_hblank,next_vblank;¿
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;¿
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;¿
   always @(posedge vclock) begin¿
      hcount <= hreset ? 0 : hcount + 1;¿
      hblank <= next_hblank;¿
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low¿
¿
```

```
      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;¿
      vblank <= next_vblank;¿
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low¿
¿
      blank <= next_vblank | (next_hblank & ~hreset);¿
   end¿
endmodule¿

//////////////////////////////////////////////////////////////////////////////
//
// Module to delay a signal of a given width by a given number of cycles
//
// Mike Wang

module delay #(parameter DELAY_CYCLES = 28,
                parameter DATA_WIDTH = 8)
                    (input clock,
                     input [DATA_WIDTH-1:0] din,
                     output [DATA_WIDTH-1:0] dout);

    reg [DATA_WIDTH-1:0] holder [DELAY_CYCLES:0]; // shift register
    integer i;  // for the for loop

    always @(posedge clock) begin
        holder[DELAY_CYCLES] <= din;
        for (i=0; i<DELAY_CYCLES; i=i+1) begin
            holder[i] <= holder[i+1];
        end
    end
    assign dout = holder[0];
endmodule

//
//////////////////////////////////////////////////////////////////////////////
                    ¿
¿
//////////////////////////////////////////////////////////////////////////////¿
// generate display pixels from reading the ZBT ram¿
// note that the ZBT ram has 2 cycles of read (and write) latency¿
//¿
// We take care of that by latching the data at an appropriate time.¿
//¿
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,¿
// decoded into four bytes of pixel data.¿
//¿
// Bug due to memory management will be fixed. The bug happens because¿
// memory is called based on current hcount & vcount, which will actually¿
// shows up 2 cycle in the future. Not to mention that these incoming data¿
// are latched for 2 cycles before they are used. Also remember that the¿
// ntsc2zbt's addressing protocol has been fixed. ¿
¿
// The original bug:¿
// -. At (hcount, vcount) = (100, 201) data at memory address(0,100,49) ¿
//    arrives at vram_read_data, latch it to vr_data_latched.¿
// -. At (hcount, vcount) = (100, 203) data at memory address(0,100,49) ¿
//    is latched to last_vr_data to be used for display.¿
// -. Remember that memory address(0,100,49) contains camera data¿
//    pixel(100,192) - pixel(100,195).¿
// -. At (hcount, vcount) = (100, 204) camera pixel data(100,192) is shown.¿
// -. At (hcount, vcount) = (100, 205) camera pixel data(100,193) is shown. ¿
// -. At (hcount, vcount) = (100, 206) camera pixel data(100,194) is shown.¿
// -. At (hcount, vcount) = (100, 207) camera pixel data(100,195) is shown.¿
//¿
// Unfortunately this means that at (hcount == 0) to (hcount == 11) data from¿
// the right side of the camera is shown instead (including possible sync signals).
¿
¿
// To fix this, two corrections has been made:¿
// -. Fix addressing protocol in ntsc_to_zbt module.¿
// -. Forecast hcount & vcount 8 clock cycles ahead and use that¿
//    instead to call data from ZBT.¿
¿
¿
```

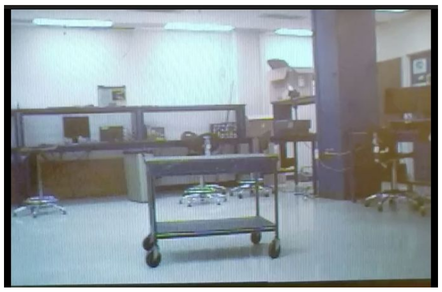
```
module vram_display(reset,clk,hcount,vcount,vr_pixel,¿
            vram_addr,vram_read_data);¿
¿
   input reset, clk;¿
   input [10:0] hcount;
   input [9:0]  vcount;¿
   output [17:0] vr_pixel;¿
   output [18:0] vram_addr;¿
   input [35:0]  vram_read_data;¿
¿
   //forecast hcount & vcount 8 clock cycles ahead to get data from ZBT¿
   wire [10:0] hcount_f = (hcount >= 1048) ? (hcount - 1048) : (hcount + 8);
   wire [9:0] vcount_f = (hcount >= 1048) ? ((vcount == 805) ? 0 : vcount + 1) : vco
unt;¿
      ¿
   wire [18:0]   vram_addr = {vcount_f, hcount_f[9:1]};¿
¿
   wire      hc4 = hcount[0];¿
   reg [17:0]    vr_pixel;¿
   reg [35:0]    vr_data_latched;¿
   reg [35:0]    last_vr_data;¿
¿
   always @(posedge clk)¿
     last_vr_data <= (hc4==1'b1) ? vr_data_latched : last_vr_data;¿
¿
   always @(posedge clk)¿
     vr_data_latched <= (hc4==1'b0) ? vram_read_data : vr_data_latched;¿
¿
   always @(*)      // each 36-bit word from RAM is decoded to 4 bytes¿
     case (hc4)¿
       1: vr_pixel = last_vr_data[17:0];¿
       0: vr_pixel = last_vr_data[35:18];¿
     endcase¿
¿
endmodule // vram_display¿
¿
/////////////////////////////////////////////////////////////////////////////¿
// parameterized delay line ¿
¿
module delayN(clk,in,out);¿
   input clk;¿
   input in;¿
   output out;¿
¿
   parameter NDELAY = 3;¿
¿
   reg [NDELAY-1:0] shiftreg;¿
   wire          out = shiftreg[NDELAY-1];¿
¿
   always @(posedge clk)¿
     shiftreg <= {shiftreg[NDELAY-2:0],in};¿
¿
endmodule // delayN¿
¿
////////////////////////////////////////////////////////////////////////////¿
// ramclock module¿
¿
///////////////////////////////////////////////////////////////////////////////¿
//¿
// 6.111 FPGA Labkit -- ZBT RAM clock generation¿
//¿
//¿
// Created: April 27, 2004¿
// Author: Nathan Ickes¿
//¿
///////////////////////////////////////////////////////////////////////////////¿
//¿
// This module generates deskewed clocks for driving the ZBT SRAMs and FPGA ¿
// registers. A special feedback trace on the labkit PCB (which is length ¿
// matched to the RAM traces) is used to adjust the RAM clock phase so that ¿
// rising clock edges reach the RAMs at exactly the same time as rising clock ¿
// edges reach the registers in the FPGA.¿
//¿
```

```
// The RAM clock signals are driven by DDR output buffers, which further ¿
// ensures that the clock-to-pad delay is the same for the RAM clocks as it is ¿
// for any other registered RAM signal.¿
//¿
// When the FPGA is configured, the DCMs are enabled before the chip-level I/O¿
// drivers are released from tristate. It is therefore necessary to¿
// artificially hold the DCMs in reset for a few cycles after configuration. ¿
// This is done using a 16-bit shift register. When the DCMs have locked, the ¿
// <lock> output of this mnodule will go high. Until the DCMs are locked, the ¿
// ouput clock timings are not guaranteed, so any logic driven by the ¿
// <fpga_clock> should probably be held inreset until <locked> is high.¿
//¿
///////////////////////////////////////////////////////////////////////////////¿
¿
module ramclock(ref_clock, fpga_clock, ram0_clock, ram1_clock, ¿
                clock_feedback_in, clock_feedback_out, locked);¿
   ¿
   input ref_clock;                  // Reference clock input¿
   output fpga_clock;                // Output clock to drive FPGA logic¿
   output ram0_clock, ram1_clock;    // Output clocks for each RAM chip¿
   input  clock_feedback_in;         // Output to feedback trace¿
   output clock_feedback_out;        // Input from feedback trace¿
   output locked;                    // Indicates that clock outputs are stable¿
   ¿
   wire  ref_clk, fpga_clk, ram_clk, fb_clk, lock1, lock2, dcm_reset;¿
¿
   ////////////////////////////////////////////////////////////////////////////¿
   ¿
   //To force ISE to compile the ramclock, this line has to be removed.¿
   //IBUFG ref_buf (.O(ref_clk), .I(ref_clock));¿
    ¿
    assign ref_clk = ref_clock;
    wire ram_clock;¿
   ¿
   BUFG int_buf (.O(fpga_clock), .I(fpga_clk));¿
¿
   DCM int_dcm (.CLKFB(fpga_clock),¿
        .CLKIN(ref_clk),¿
        .RST(dcm_reset),¿
        .CLK0(fpga_clk),¿
        .LOCKED(lock1));¿
   // synthesis attribute DLL_FREQUENCY_MODE of int_dcm is "LOW"¿
   // synthesis attribute DUTY_CYCLE_CORRECTION of int_dcm is "TRUE"¿
   // synthesis attribute STARTUP_WAIT of int_dcm is "FALSE"¿
   // synthesis attribute DFS_FREQUENCY_MODE of int_dcm is "LOW"¿
   // synthesis attribute CLK_FEEDBACK of int_dcm  is "1X"¿
   // synthesis attribute CLKOUT_PHASE_SHIFT of int_dcm is "NONE"¿
   // synthesis attribute PHASE_SHIFT of int_dcm is 0¿
   ¿
   BUFG ext_buf (.O(ram_clock), .I(ram_clk));¿
   ¿
   IBUFG fb_buf (.O(fb_clk), .I(clock_feedback_in));¿
   ¿
   DCM ext_dcm (.CLKFB(fb_clk), ¿
            .CLKIN(ref_clk), ¿
            .RST(dcm_reset),¿
            .CLK0(ram_clk),¿
            .LOCKED(lock2));¿
   // synthesis attribute DLL_FREQUENCY_MODE of ext_dcm is "LOW"¿
   // synthesis attribute DUTY_CYCLE_CORRECTION of ext_dcm is "TRUE"¿
   // synthesis attribute STARTUP_WAIT of ext_dcm is "FALSE"¿
   // synthesis attribute DFS_FREQUENCY_MODE of ext_dcm is "LOW"¿
   // synthesis attribute CLK_FEEDBACK of ext_dcm  is "1X"¿
   // synthesis attribute CLKOUT_PHASE_SHIFT of ext_dcm is "NONE"¿
   // synthesis attribute PHASE_SHIFT of ext_dcm is 0¿
¿
   SRL16 dcm_rst_sr (.D(1'b0), .CLK(ref_clk), .Q(dcm_reset),¿
             .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));¿
   // synthesis attribute init of dcm_rst_sr is "000F";¿
   ¿
¿
   OFDDRRSE ddr_reg0 (.Q(ram0_clock), .C0(ram_clock), .C1(~ram_clock),¿
              .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));¿
   OFDDRRSE ddr_reg1 (.Q(ram1_clock), .C0(ram_clock), .C1(~ram_clock),¿
              .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));¿
   OFDDRRSE ddr_reg2 (.Q(clock_feedback_out), .C0(ram_clock), .C1(~ram_clock),¿
              .CE (1'b1), .D0(1'b1), .D1(1'b0), .R(1'b0), .S(1'b0));¿
¿
   assign locked = lock1 && lock2;¿
   ¿
endmodule¿
¿
¿
```

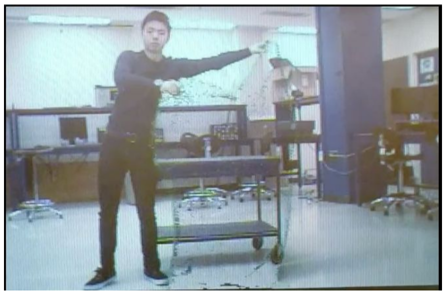
Extra: Extending the Project (Mike)

In the days after the final project checkoff I extended the our project to produce an invisibility cloak effect. Though this technically isn't part of our project and was completed after the deadline so it should not count towards the grade, I think it bears mentioning in the spirit of helping out whoever wants to implement something similar in the future.

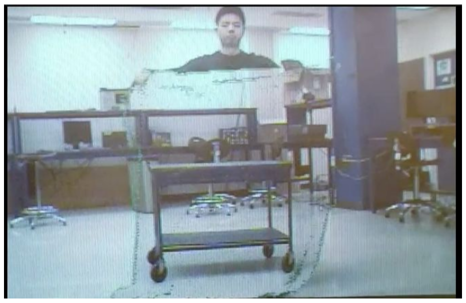
a) The scene



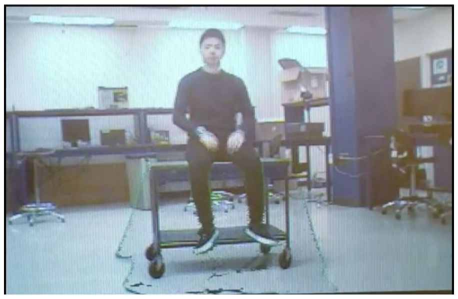
b) The cloak is transparent...



c) ... but humans are hidden!



d) No green screen in the background - that's a real cart



This effect was accomplished by activating the other ZBT memory bank, and having it record the camera output while a button was pressed. Concretely, this means that this new ZBT memory has the same write address and write data as the original ZBT memory used to record the camera output, but the write enable needs to be changed so that it is only pulled high when a button is pressed. Then, instead of replacing green pixels with the output of the ROM you replace them with the output of this new ZBT memory. Both ZBTs take the same read address.

In effect, you're taking a picture of the scene beforehand and when you walk in with the green cloth, the green would be replaced by whatever was behind it, assuming nothing in the background had changed (this is the fatal flaw: the background cannot change otherwise the effect is ruined. If the cart was moved away after the button was pressed, putting the green screen in front of the where the cart used to be would make the cart show up again on the monitor, since it was saved into the new ZBT memory).