Laser Shot: Video-based Alternative to Arcade Light Guns

Tiffany Chen, Spencer Sugimoto, Paul Yang
6.111 Introduction to Digital Systems Laboratory, Fall 2006
December 13, 2006

Abstract

Laser shot is an improvement of the arcade light gun system often used in shooting arcade games such as Duck Hunt. In Duck Hunt, a light-sensor gun is used to track the aiming point on a CRT monitor. However, such an approach is incompatible with LCD and plasma screens. An alternative approach is to use laser tracked by a video camera system. The video camera was placed with a clear view of the screen, and the image captured was processed to determine the location of the laser on the screen. To demonstrate the functionality of the laser-based aiming system, a modified version of Duck Hunt was be implemented, complete with video and audio features. This report details the specifications, design, and development of the Laser Shot and offers methods of improvements for the design and debugging processes.

Table of Contents

List of Tables

List of Figures

Overview

Laser Shot is an alternative to the traditional light gun system used in arcade shooting games. Current light gun systems used in arcade systems are incompatible with LCD screens. To resolve this problem, Laser Shot uses a video camera system to track a laser point aimed on the screen. This shooting system is demonstrated though implementation of Duck Hunt, in which ducks are targets for the laser-based gun.

The implementation of the project is divided into two major components: the dot finder, which tracks the laser, and the Duck Hunt system. The user is able to interact with the system using labkit buttons and a laser gun. The laser gun is wired to emit a brief 1/20 second pulse when the trigger is pulled. Figure 1 shows the setup of Laser Shot.

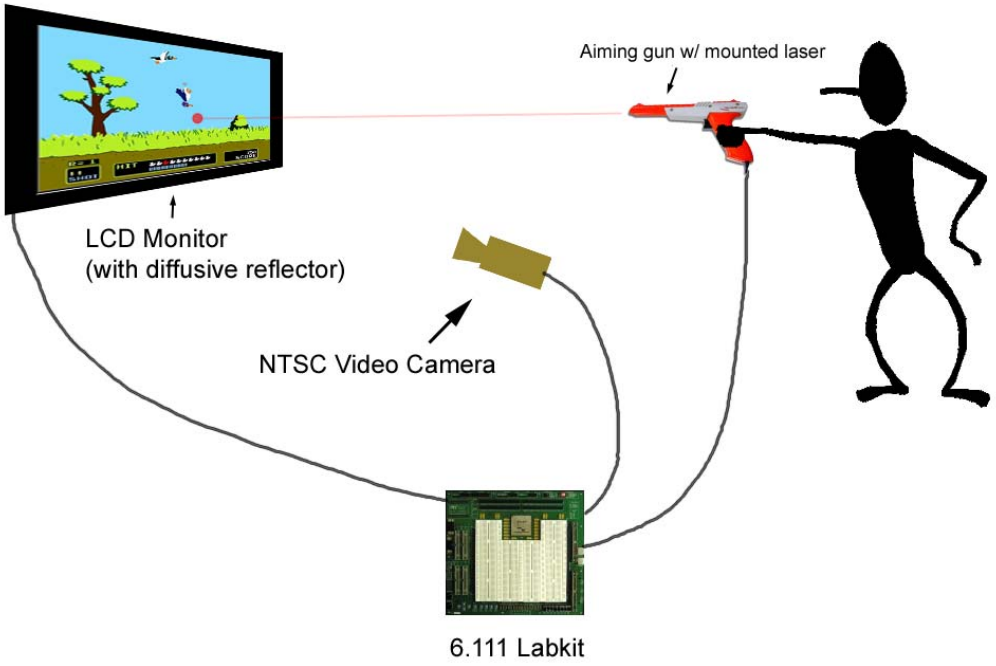

Figure 1 - Laser Shot Setup

Dot Finder System Overview

The dot finder system is responsible for figuring out where the laser dot is located on the LCD screen. The system calculates the estimated location of the laser dot though the image captured by an NTSC camera. The camera is positioned in front of the display and is angled such that it has a view of the full screen. Furthermore, a sanded sheet of thin plastic is overlaid on top of the screen to improve reflectivity. When the laser is pointed at a location on the screen, the camera sees the bright red laser dot and the resulting image is processed and transformed to extract the location of the laser dot on the LCD screen. The dot finder subsystem is divided into two major parts: the video capture

and the point calculation subsystems. An auxiliary display subsystem is used to output the resulting data on a VGA display.

Video Capture Subsystem

The video image is captured from the camera by decoding signals that are produced by the AD7185 NTSC decoder IC. The data from the decoder IC is converted to a series of signals that contain the video frame data in the YCrCb color space. The YCrCb image is then converted into a RGB image to aid in processing. Since the laser is red, the blue and the red channels are filtered out to remove parts of the image that are not related to the laser. Once the green and the blue value are removed, a threshold filter sets all pixels above a specified value to 255 and all other pixels to zero. The resulting image is sent to the point calculation system to extract the location of the laser dot on the LCD screen.

Point Calculation System

The point calculation system uses the threshold image to produce the average (x,y) coordinates of all non-zero pixels. The average (x,y) coordinate represents the "center of mass" and is assumed to be the center location of the laser dot. A coordinate transform then converts the center of mass location in the NTSC frame into a location on the LCD screen. Figure 2 shows the location of the laser dot on the LCD screen at a head-on angle whereas figure 3 shows the location of the laser dot as seen by the video camera. Note that the captured image exhibits perspective distortion. The coordinate transform corrects the perspective distortion using eight coefficients that are generated from the four corner points of the LCD display in the captured video frame. The transformed coordinate is used to display a centered blob on the LCD.

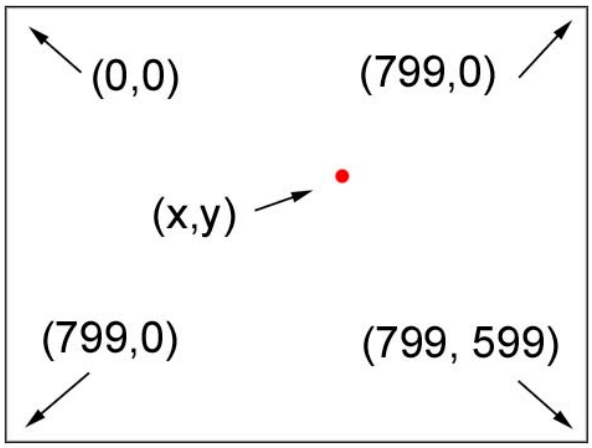

Figure 2 - Coordinates of laser dot on the LCD frame

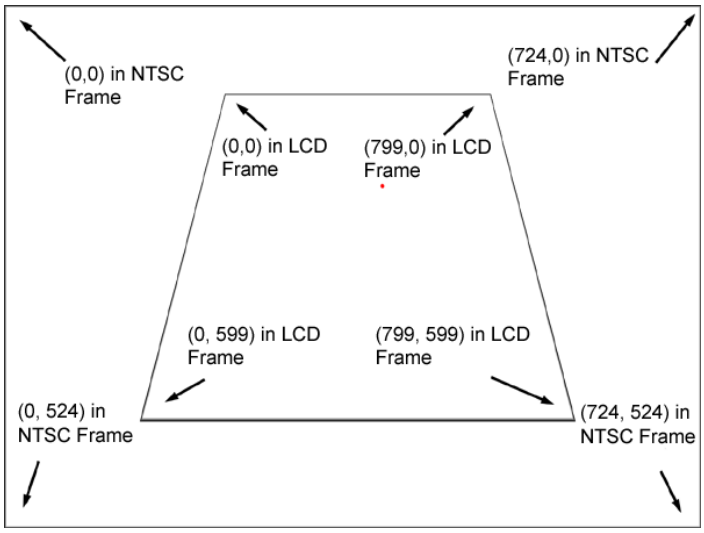

Figure 3 - The LCD screen as seen by the video camera with labeled corner points in both frames

Duck Hunt System

The duck hunt video system uses 4-bit color to display the duck animation and video on the LCD screen. An 800 by 600 pixel resolution was used at a refresh rate of 60 Hz with a pixel clock of 40 MHz. This resolution matches most closely with the dimensions of the screen capture using the video camera that was used to track the laser dot on the LCD screen.

The duck hunt game starts on the title frame, displaying the duck hunt logo, a picture of the duck, and the project title. After eight seconds, the duck hunt game begins. One of four bushes shudders to warn the user to watch for a duck, and after a variable number of seconds, the duck flies out of the bush and into the sky. The duck's wings flap up and down, and upon reaching the edges of the screen, the duck reverses its direction and the animation changes accordingly. No more than one duck is on the screen at any given time. In addition to ducks, the game system also consists of stationary and animated bushes as well as clouds that scroll across the sky at constant speeds.

The user has 45 seconds to kill six ducks. Each time the duck is hit, the duck freezes briefly on the screen in a state of shock and then continues to fly across the sky. On the third hit, however, the duck dies and falls to the ground. Upon falling onto the ground, another random bush shudders to signify the appearance of the next duck, and the sequence of animations continues in such a cycle. If the user is not able to hit the duck eight seconds after the duck appears from the bushes, the duck flies away off the screen, and the user must wait until the next duck appears to score. The duck also flies off the screen if more than eight seconds elapse between subsequent shots of the duck. The game becomes progressively more difficult: the first set of three ducks fly slowly, while the second set of ducks fly at medium speed. The last two ducks move very quickly across the screen.

The color to the VGA is controlled using a set of modules. The XVGA module determines the horizontal and vertical sync signals for the VGA and generates horizontal and vertical pixel counts to the video system. To determine the color to output, the video controller takes in coordinates of the clouds, ducks, bushes, and character strings and layers them according to their priority. The video controller also sets the transparency of the images displayed. The video controller outputs 4-bit pixel data, which is then sent to the colormap to match up the pixel data to the appropriate red, green, and blue (RGB) output values.

The movement and animation of the duck is controlled by a collection of modules. ROMs are used to store the various images of the duck: the duck dying, the duck getting shot, the duck with its wings up facing east, the duck with its wings down facing east, the duck with its wings up facing west, and the duck with its wings up facing east. The duck animation is created using the duck controller module, which is a finite state machine (FSM) that transitions between the various duck image and controls the coordinates of the duck on the screen. A timer module determines when the duck controller should transition between the duck images. The appear counter controls when the duck should fly out of the bushes and when eight seconds has elapsed after consecutive shots or after the bush flies out of the bushes. The number of shots the duck has sustained is kept track of in the duck health module, and when the duck has sustained three shots, the module signals the duck controller for the duck to die. Finally, the speed at which the duck flies is determined by the velocity module.

Other components of the duck hunt system include the clouds and bushes. The cloud image is stored in a ROM, and its pixel data is read off of using the cloud controller module. The cloud controller module also controls the horizontal position of the cloud so that it scrolls across the screen. Similarly, the image of the bush is stored in a ROM, and the pixel data is read off of with the bush controller module.

The rules of play of the Lasershot game are handled by the Scorekeeper System. The time of the game is decremented every second. The user's score, is also determined by the Scorekeeper. Whenever a duck is shot down, the scorekeeper stores this information as the score.

The Audio System allows the FPGA to play 48-khz sound effects at any desired time or signal. Original sound data is first down-sampled to a portable 8-bit, 8khz wav file, by Windows Sound Recorder. Using a simple java program, the sound data is converted into coefficient data (the .coe format), as specified in the data sheet for the FPGA's AC97 sound processing chip. The new .coe files are stored in individually created ROMs on the FPGA. The .coe sound is interpreted and up-sampled from 8khz back to 48khz samples using a 6-step interpolation codec, whenever the sound is needed. Using this method, a number of sound modules are instantiated with specific logic to determine the behavior of the playback of the sound.

In the final version of the Lasershot game, a laser gun sound is played whenever the physical gun is shot. The trigger of the gun is tied to the FPGA through a user input port on the labkit. The laser gun sound file plays for approximately one second, but it is very possible for a user to pull the trigger at a rate faster than one shot per second. To account for this, the Audio System's laser gun sound component uses logic that allows the sound file to be played in rapid succession. That is to say, whenever the trigger is

pulled, a new laser sound will play, seamlessly overriding any previous laser sound that may be playing. This allows for more realistic trigger functionality.

Losing the game, a user receives the game over sound from the original Duck Hunt game. Based on the rules of the game, when the Scorekeeper System ends the game, the game over sound logic detects the game over and outputs the sound file. The game over sound file plays for approximately 1.5 seconds. At this point, the Lasershot game is completely frozen, so there is no need for the game over sound to have the rapid play functionality of the laser gun sound component.

Additional modules for the duck hunt video system include a random number generator, a seed generator, a frame module, and a signal to pulse converter. A random number generator, or more specifically a linear feedback shift register (LFSR), and seed generator are used to create randomness and unpredictability in the timing of the appearance of the duck and the bush the duck flies out of. The frame module signals to the duck hunt system when a new frame is created, and a signal to pulse converter is used throughout the system to convert a constant high signal to a pulse one clock cycle high.

This report details the various modules used to create the laser shot project, which consists of the laser dot finder system and the duck hunt system. Testing and debugging procedures used in the development of the project are described. The report concludes with possible ways the design could be improved as well as more efficient methods for the development of the laser shot system.

Description of Laser Shot System

The laser shot system consists of two major components: the laser dot finder system and the duck hunt game system. The two systems and their implementation are detailed in this section.

Video Capture Subsystem Modules (Paul Yang)

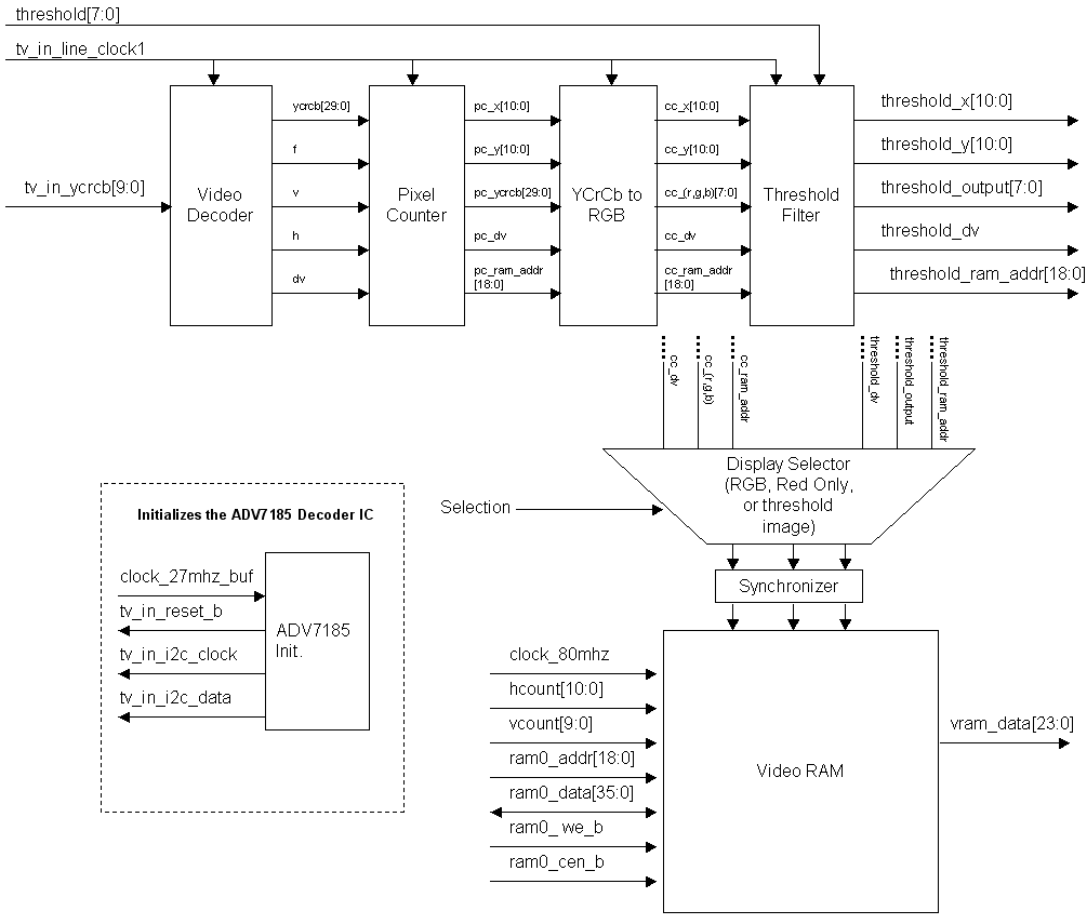

Figure 4 - Block Diagram for Video Capture Subsystem

ADV7185 Video Decoder Configuration Init (adv7185init in video_decoder.v)

The ADV7185 video decoder configuration init module configures the ADV7185 decoder using the I2C interface so that the decoder properly decodes the NTSC signal from the video camera. The module runs on the 27MHz clock and has an input *source* that selects between the composite input and the s-video input on the labkit. The

tv_in_reset_b output is used to reset the ADV7185 and *tv_in_i2c_clock*, and *tv_in_i2c_data outputs* are used to communicate on the I2C bus with the decoder. The ADV7185 Video Decoder module was supplied as a part of a sample application and was used without modifications in the project

Video Decoder (module ntsc_decode in video_decoder.v)

The video decoder module is used to decode the 10-bit data stream that is output by the ADV7185. The data stream contains a series of maker bytes that indicates the start and end of the active video frame. YCrCb data is transmitted between the markers. The video decoder uses an FSM to detect when the active video frame starts and has several states to appropriately latch data at correct points in the data stream. The module outputs four five signals, *ycrcb*, *f*, *v*, *h*, and *data_valid*. *ycrcb* is a 30 bit output that packs luma in the upper 10 bits, red chroma in the middle 10 bits, and blue chroma in the lower 10 bits. *f* indicates whether the sample is in the even or odd field and *v*,*h* indicates a horizontal and vertical sync, respectively. Finally the *dv* output indicates whether the output data from the module is valid or not. The video decoder module was supplied as a part of a sample application and was used without modifications in the project.

Pixel Counter (module pixel_counter in pixel_counter.v)

The pixel counter uses the *ycrcb*, *f*, *v*, *h*, and *dv* outputs of the video decoder modules to calculate what pixel in the 720x525 NTSC video frame a particular sample belongs to. The pixel counter has internal x and y coordinate counters that are reset on a valid vertical sync. The x counter is always reset to zero where as the y coordinate is reset to either zero or a one depending on the field. On every horizontal sync pulse, the x coordinate counter is reset to zero and the y counter is incremented by two. In addition to keeping track of the x,y coordinate of each pixel, the module also keeps a running count of the RAM address that corresponds to each sample. The RAM address is later used by the video RAM module to write the pixel data to the correct location. The RAM address is computed in a similar method as the x,y coordinates. An internal counter is reset to 0 or 800 on a vertical sync pulse, again depending on the field. The address of 800 corresponds to the first pixel of the second line in the video RAM. At each clock cycle, a valid sample increments the counter by one. A horizontal sync increments the address counter by 75 + 800. Incrementing the RAM address counter by 75 moves the RAM address to next line in the VRAM and adding another 800 moves the address by one more line. Using this method to compute the x,y coordinate and the RAM address results in invalid data until the first vertical sync pulse but since video data is repeated, invalid values are quickly overwritten. The outputs of the module are an 11 bit x coordinate (*x*), a 10 bit y coordinate (*y*), and the corresponding ycrcb (*ycrcb)*, data valid (*dv_out*), and RAM address (*ram_addr*).

YCrCb To RGB Color Converter (module YCrCb2RGB in mult_model.v)

The YCrCb color data must be converted into RGB color before it can be displayed on a VGA monitor. The following equations are used to perform the conversion:

$$R = 1.164(Y-16) + 1.596(Cr-128)$$
$$G = 1.164(Y-16) - 0.813(Cr-128) - 0.392(Cb-128)$$
$$B = 1.164(Y-16) + 2.017(Cb-128)$$

The YCrCb to RGB color converter performs the above calculation though the use of several multipliers in a pipelined circuit that has a latency of five clock cycles. However, the module produces a color conversion every clock cycle. The original YCrCb2RGB module from Xilinx was modified to include x,y coordinates, RAM address, and data valid signals from the pixel counter as inputs and outputs. The module does not modify signals but does delay them by five clock cycles to keep the coordinates in sync with the color data.

Threshold Filter (module threshold in threshold.v)

The threshold filter compares the 8-bit input value *input_val* with the 8 bit threshold *threshold_val.* If *input_val* is greater than *threshold_val,* then 255 is output on *outpu_ val.* Otherwise, 0 is output on *output_val*. The *threshold_val* input is driven by a threshold value register that keeps track of the user selected threshold value. The threshold register is appropriately wired to pushbuttons so that the value can be changed to suit lighting conditions as the system is running. Since *output_val* is registered, the result is not available until the next clock cycle. Similar to the YCrCb2RGB module, the threshold filter does accept x,y coordinates, data valid, and RAM address but does not modify them. Instead, they are delayed by one clock cycle to keep all data for each pixel in sync.

Video RAM (module vram in vram.v)

The video RAM module is responsible for storing the video captured data as it is produced by the YCrCb2RGB and threshold modules and for providing appropriate data to the ADV7125 to display the captured image. The video RAM uses the built in ZBT RAM to function as the primary storage element. Within the video RAM module, the zbt_6111 sample driver was instantiated to provide the proper signals to the RAM. The zbt_6111 module was slightly modified to register read data as timing requirements were not met when running at high frequencies when using the original sample driver. Using the ZBT RAM for video frame storage requires accommodating characteristics of the RAM while designing the respective video display circuitry. Because the data width of the SRAM is 36 bits and each color pixel requires 24 bits, only data for a single pixel location can be stored into each memory location without loss of precision. Since the ADV7125 requires a pixel to be read at 40MHz for 800x600 resolution, the SRAM is clocked at 80MHz to enable both reading and writing to a pixel at 40MHz. Running the single port memory at 80MHz effectively emulates a dual port ram at 40MHz. The 80MHz clock must be in phase with the 40Mhz clock to maintain synchronous logic

design. DCMs are used in a clock-doubling configuration with feedback to generate the clock signals. Finally, in addition to the doubled-clock requirement, the ZBT SRAMS have a two-cycle delay between the input of addresses and enables to the output of the data. The delay is taken into account when designing the VGA output circuitry to delay the output of the horizontal sync, vertical sync, and blanking signals by two clock cycles.

The dual port functionality is achieved through the use of an internal parity bit that indicates whether data from the video capture modules should be written, or if data should be read from the memory to send to the ADV7125. The parity bit alternates read-write cycles and changes value with the 80MHz clock. The parity bit is set to read on the first instance when the inputs *hcount* and *vcount* are zero. By resetting the parity bit at a known point on the screen, the display will not shift due to the changing order of reads and writes as *hcount* and *vcount* cycle though multiple screen refreshes. The data corresponding to the given *hcount* and *vcount* are output on the *read_data* output port when the parity bit is set to read. When the parity bit is set to write, *write_data* is written to the *write_address* of the ZBT SRAM. The *write_data* and *write_address* inputs are multiplexed to get values from the RGB color output of the YCrCb2RGB module or the threshold-adjusted output of the threshold filter module. Because the YCrCb2RGB and threshold filter modules run off the TV in line clock, the signals from those modules must be synchronized to the 40MHz clock to avoid metastability issues.

Point Calculation Modules (Paul Yang)

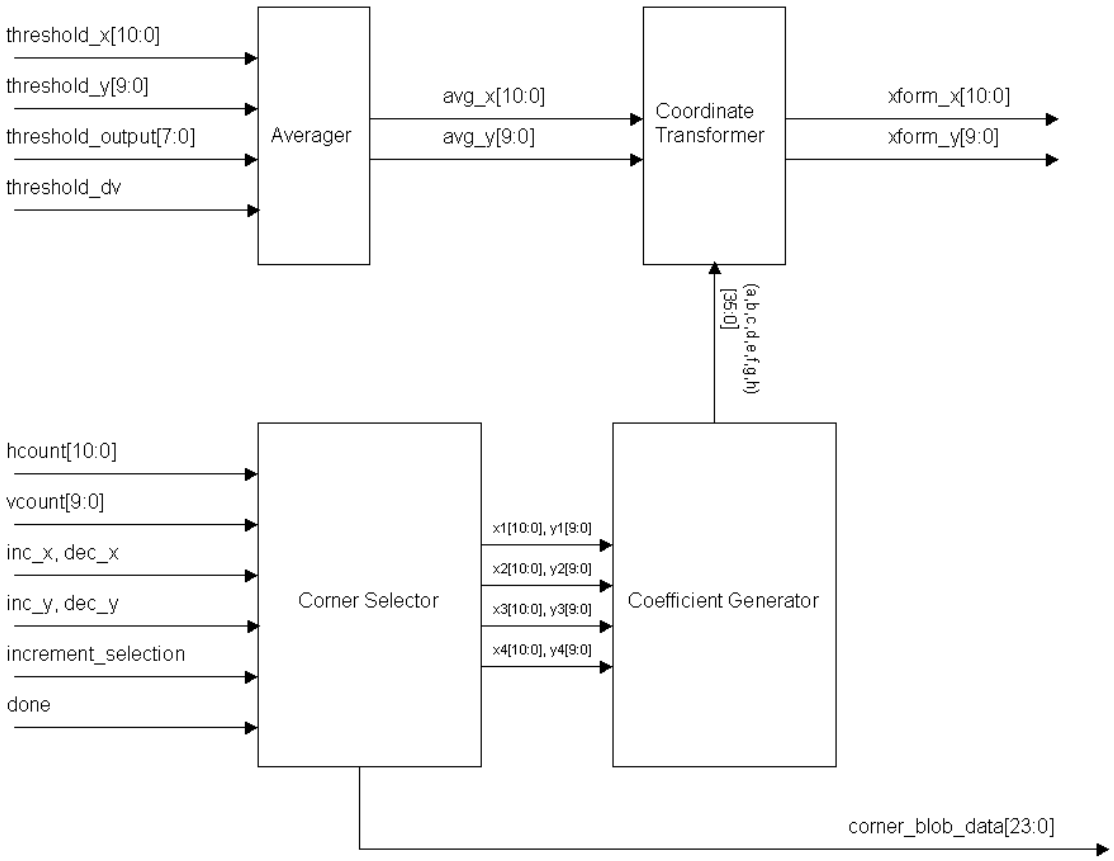

Figure 5 - Block Diagram for Point Calculator

Averager (module averager in averager.v)

The averager module is responsible to finding the center of mass of the threshold-filtered image. The inputs to the averager module are the x,y coordinates, data valid, and output value signals from the threshold module. When data valid is true and output value is 255, the module adds the x and y values to respective accumulation registers. At the same time, a counter keeps track of the number of pixels meeting the requirement. At the end of the frame, the accumulated x and y values are divided by the number of pixels to produce and average x and y values of all pixels that were marked as 255 by the threshold filter.

In an effort to reduce the logic required to compute the average, that assumption was made that there will be at most 65535 pixels that have a value of 2555 after the threshold filter. The sum of 65535, x coordinates can be at most 400 * 65535 and consequently will require $\log_2(400 * 65535) \approx 25$ bits to store (400 is the mean value of the x coordinate). Likewise, the counter for the number of pixels can be set to only 16 bits. Though COREGen, a divider with a 25-bit dividend and a 16-bit divisor was created to compute the average. The generated divider was also set to use 8 clock cycles per division to further reduce the logic requirements. Because the average is computed once every frame, there are no strict timing requirements on the divisor.

Coordinate Transformer (module coord_transform in coord_transform.v)

The coordinate transformer converts a coordinate in the 720x525 NTSC frame to a coordinate on the 800x600 LCD screen. As mentioned in the overview, the NTSC frame contains a perspective-distorted view of the 800x600 LCD screen with a laser dot somewhere on the image. The calculation that is performed is as follows:

$$x' = \frac{Ax + Bx + C}{Gx + Hy + 1} \qquad y' = \frac{Dx + Ex + F}{Gx + Hy + 1}$$

(x,y) are coordinates in the NTSC frame and (X,Y) are the estimated coordinates on the LCD screen. The coefficients in the equations are generated from known mappings from (x,y) to (X, Y). However, since the coefficients can only be represented as integer values, a scale factor of 2^{18} is applied to the numerator and denominator to approximate the coefficients as integers. The coordinate transform equations are then:

$$x' = \frac{ax + bx + c}{gx + hy + s} \qquad Y = \frac{dx + ex + f}{gx + hy + s}$$

Where $\lfloor s * \{A, B, C, D, E, F, G, H\} \rfloor = \{a,b,c,d,e,f,g,h\}$ and $s = 2^{18}$

The scale factor was chosen based on estimates for the range of coefficient values. A larger scale factor results in less round-off errors at the expense of complexity. Through informal testing, a scale factor of 2^{18} proved to be sufficiently accurate to perform the required transformation. After applying the scale factor, the coefficients can represented using 36-bit signed integers. Fewer bits cannot be used, as it would cause an overflows with the given scale factor.

The coordinate transformer has seven 36-bit inputs for the coefficients *a,b,c,d,e,f,g,h*, as well as an input for the x,y coordinate, *x_in* and *y_in*. The outputs of the module are *x_out* and *y_out*, the transformed (x,y) coordinates. The output is available 35 clock cycles after the data is input. The delay corresponds to the latency of the divider with a 32-bit dividend and a 32-bit divisor.

Coefficient Generator (module coeff_gen in coeff_gen.v)

The coefficient generator creates the coefficients for the coordinate transformer using 4 points on the NTSC frame: p1 = (x1, y1) is defined as the pixel location of the upper left hand corner of the LCD screen in the NTSC frame. p2 = (x2, y2) is defined as the pixel location of the upper right hand corner of the LCD screen in the NTSC frame. p3 = (x3, y3) is defined as the lower left hand corner of the LCD screen in the NTSC frame. Finally p4 = (x4, y4) is defined as the lower right hand corner of the LCD screen in the NTSC frame. x1, y1, x2, y2, x3, y3, x4, y4 are all defined as inputs to the coefficient generator.

Following those defined points, the transformed points should correspond to (X1, Y1)=(0,0), (X2, Y2)=(799,0), (X3, Y3)=(0,599), and (X4, Y4)=(799,599) respectively. The coefficients are determined by solving the following matrix equation:

$$\begin{bmatrix} x1 & y1 & 1 & 0 & 0 & 0 & -X1*x1 & -X1*y1 \\ 0 & 0 & 0 & x1 & y1 & 1 & -Y1*x1 & -Y1*y1 \\ x2 & y2 & 1 & 0 & 0 & 0 & -X2*x2 & -X2*y2 \\ 0 & 0 & 0 & x2 & y2 & 1 & -Y2*x2 & -Y2*y2 \\ x3 & y3 & 1 & 0 & 0 & 0 & -X3*x3 & -X3*y3 \\ 0 & 0 & 0 & x3 & y3 & 1 & -Y3*x3 & -Y3*y3 \\ x4 & x4 & 1 & 0 & 0 & 0 & -X4*x4 & -X4*y4 \\ 0 & 0 & 0 & x4 & y4 & 1 & -Y4*x4 & -Y4*y4 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{bmatrix} = \begin{bmatrix} X1 \\ Y1 \\ X2 \\ Y2 \\ X3 \\ Y3 \\ X4 \\ Y4 \end{bmatrix}$$

In general, the above matrix is very difficult to solve. However, with the added constraint that y1 = y2 and y3 = y4, the equations reduce to the following form (via MATLAB):

$$a = \frac{799*(y1 - y3)}{(-y3*x2 + x4*y1 + y3*x1 - x3*y1)}$$

$$b = \frac{-799*(x1 - x3)}{(-y3*x2 + x4*y1 + y3*x1 - x3*y1)}$$

$$c = \frac{799*(y3*x1 - x3*y1)}{(-y3*x2 + x4*y1 + y3*x1 - x3*y1)}$$

$$d = 0$$

$$e = \frac{599*(-x4 + x3)}{(-y3*x2 + x4*y1 + y3*x1 - x3*y1)}$$

$$f = \frac{-599*(-x4 + x3)*y1}{(-y3*x2 + x4*y1 + y3*x1 - x3*y1)}$$

$$g = 0$$

$$h = \frac{-(x1 - x2 + x4 - x3)}{(-y3*x2 + x4*y1 + y3*x1 - x3*y1)}$$

The six non-zero coefficients are calculated using a signed divider with a 32-bit dividend, a 32-bit divisor, and a 19-bit fractional remainder. The signed fraction is converted into an unsigned form by setting the highest bit to zero and subtracting one from the quotient. Then, the lower 18-bits of the fractions are concatenated with the lower 18 bits of the quotients to from 36-bit numbers that are effectively the 2^{18} scaled coefficients (a, b, c, d, e, f, g, h) required by the coordinate transformer.

Due to the high logic requirements of the divider, there is only one instance of the divider. The dividend and divisor for each coefficient are fed into the divisor by a 6-state FSM that also latches the correct output data from the divider at each state. The six non-zero coefficients, *a, b, c, e, f,* and *h,* are the outputs of the module.

Corner Selector (module corner_select_fsm in corner_select_fsm.v)

The corner selector is a FSM that allows the user select the four corner points of the LCD screen in the NTSC frame by the using the push buttons on the lab kit. In addition, the corner selector generates appropriate data based on *hcount* and *vcount* to show centered blobs at the location of the currently set corner points. Figure 6 shows a screen shot of the video data generated by the corner selector, overlaid on top of an image of the screen. The four blue dots represent the selected corner points whereas the horizontal green lines are alignment guides that are drawn when *vcount* is equal to y1 or y4.

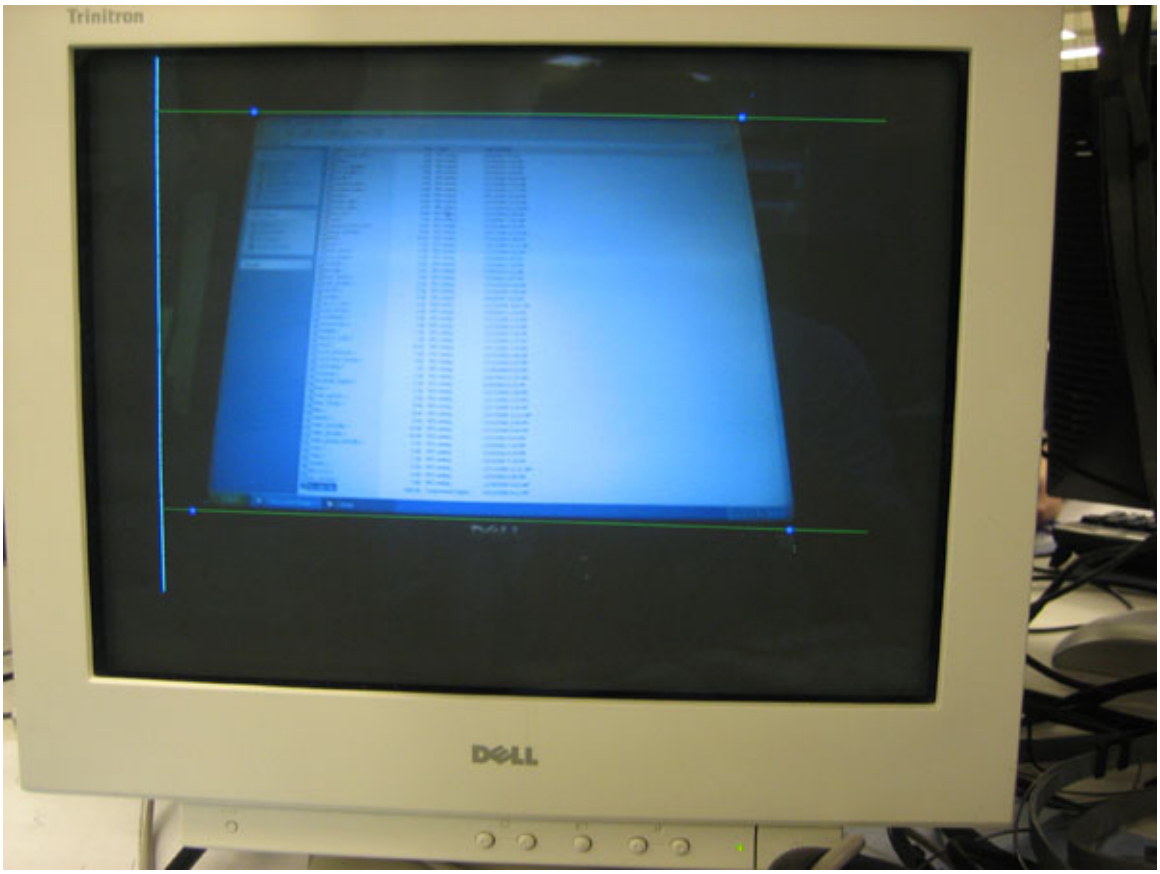

Figure 6 - Picture of Corner Select Screen

The module has five inputs for moving the locations of the corner points: *inc_x / dec_x / inc_y / inc_x* for incrementing or decrementing the x and y coordinates of the currently selected corner point, *increment_size* signal input for selecting whether the *inc/dec* inputs shift the selected corner point by 1 or 10 pixels. The increment and decrement inputs are assumed to be wired to the push buttons using the edge detector module so that each button press generates a single clock-cycle pulse. The corner selector has five states that are listed in table 1.

Table 1 - States for Corner Select Module

State Name	State Value	Description
S_IDLE	4	Increment and decrement inputs have no effect
S_UPPER_LEFT	0	Increment and decrement inputs changes the coordinates for corner point (x1,y1)
S_UPPER_RIGHT	1	" for corner point (x2,y2)
S_LOWER_RIGHT	2	" for corner point (x3,y3)
S_LOWER_LEFT	3	" for corner point (x4,y4)

The FSM starts out in the S_IDLE state after a reset and only transitions to the next state when the *done* input is high. The states always transition from S_IDLE to S_UPPER_LEFT to S_UPPER_RIGHT to S_LOWER_RIGHT to S_LOWER_LEFT and then finally back to S_IDLE.

The outputs of the module are the coordinates of the corner points, *x1, y1, x2, y2, x3, y3, x4, y4,* and they are routed to coefficient generator to create the constants required by the coordinate transform.

Dot Finder Display Modules (Paul Yang)

The dot finder display modules relate to the generation of video data that helps to visualize the outputs of the video capture and point calculation subsystems.

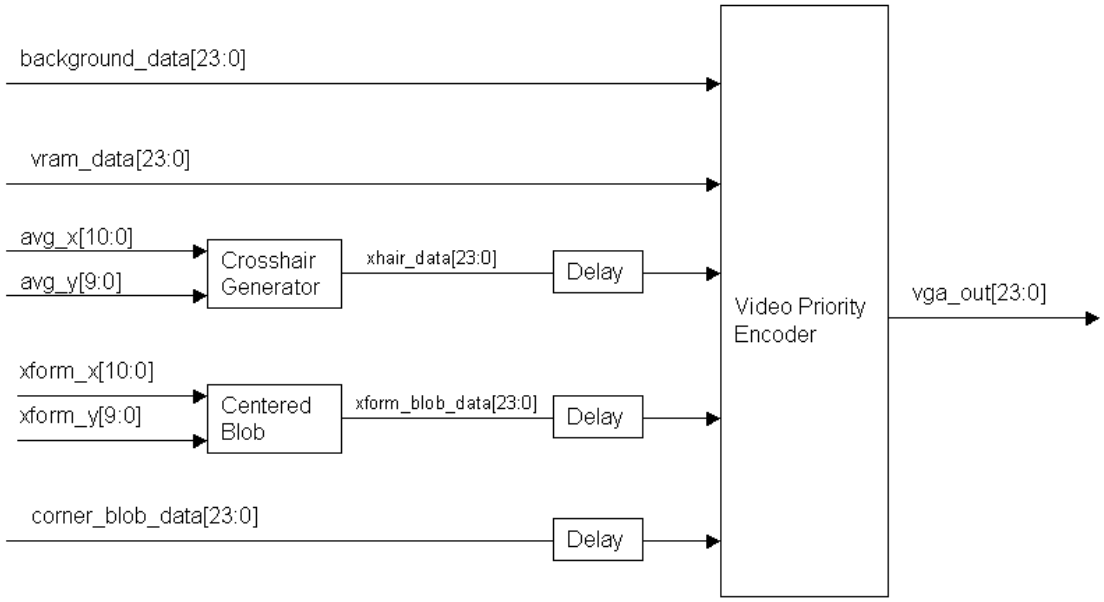

Figure 7 - Display Block Diagram

Crosshair Generator (module xhair in xhair.v)

The crosshair generator has *hcount, vcount, x,* and *y* inputs and outputs the color white on the *out* port whenever *hcount* and *vcount* are equal to *x* and *y* respectively. Otherwise, it outputs a zero. The crosshair generator is used to generate a crosshair on the center of mass in the threshold-filtered image.

Centered Blob Generator (module cblob in blob.v)

The centered blob generator has *hcount, vcount, x,* and *y* inputs and outputs a parameter COLOR value whenever *hcount, vcount* are within a parameter SIZE pixels from *x* and *y*. Effectively, the module generates a square with sides 2*SIZE pixels centered on the given (*x*,*y*) on the screen. This module is used to display a red square at the transformed coordinate.

Video Priority Encoder (module video_priority_encoder in video_priority_encoder.v)

The video priority encoder overlays pixel values from various layers to produce a merged pixel value that is sent to the VGA DAC. The encoder has a 24-bit input for each layer: *background_data* (first layer), *video_data* (second layer), *crosshair_overlay* (third layer),

xform_overlay (forth layer), and *calib_overlay* (fifth layer). The *background_data* is a single color that does not depend on the value of *hcount* and *vcount*. The *video_data* is the output of the video RAM module and represents the image captured from the video camera. The *crosshair_overlay* is the data generated by the crosshair module. The *xform_overlay* is the data for the blob centered at the transformed coordinate and finally the *calib_overlay* is the data produced by the corner selector module. All data inputs except for the *video_data* input are delayed by two clock cycles to keep all display signals in sync with the output of the video RAM module.

The merging method is to give priority to higher layers and let data from the higher layer overwrite from lower layers. However if a pixel in a higher layer has a value of 0, then the data from the previous layer is retained and passed on to the next layer. The video priority encoder uses a pipeline to restrict the logic delays incurred from the use of cascaded multiplexers in the overlay decision logic.

Duck Hunt System for Tiffany's Part

The block diagrams for Tiffany's section of the duck hunt game system can be seen in Figures 8 and 9. These block diagrams illustrate the interaction between the modules that will be discussed in this section as well as the major signals that are passed between the modules.

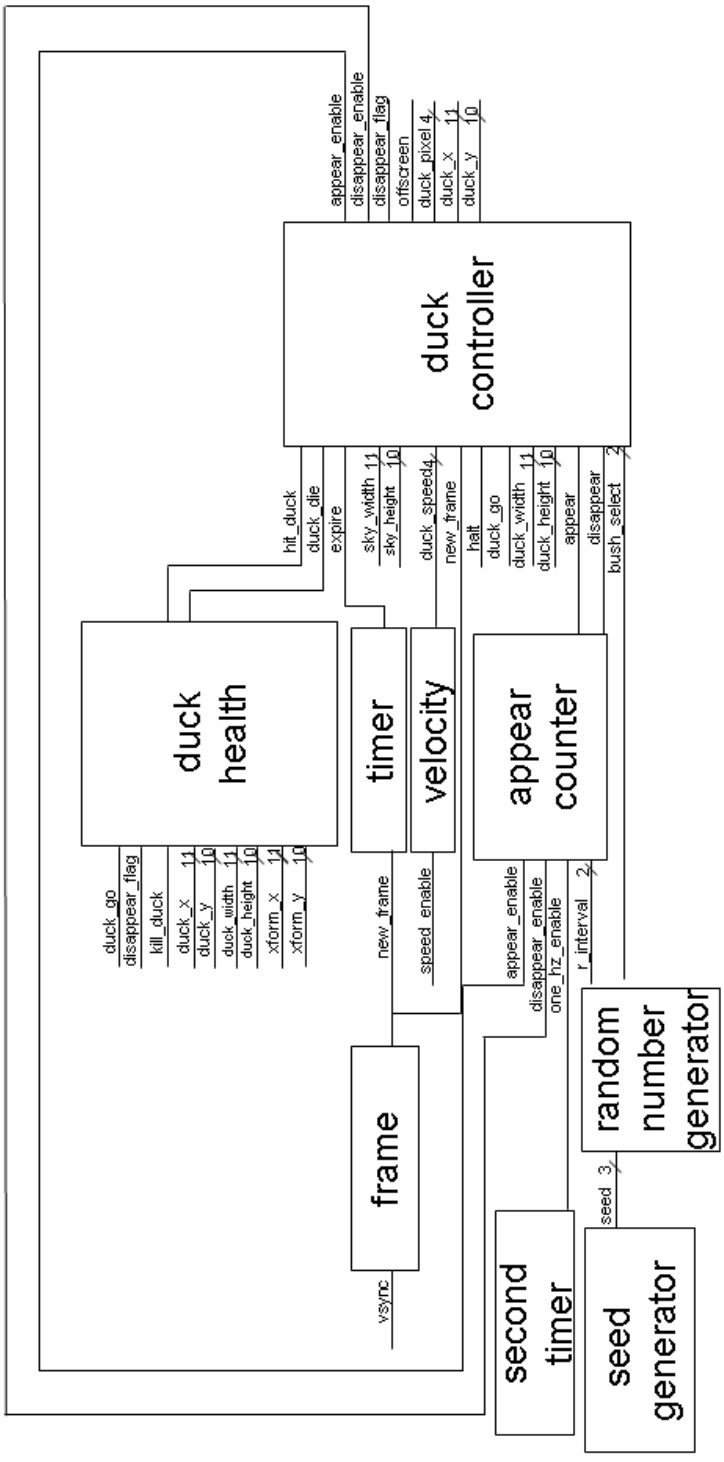

Figure 8 - Tiffany's Block Diagrams, Part I

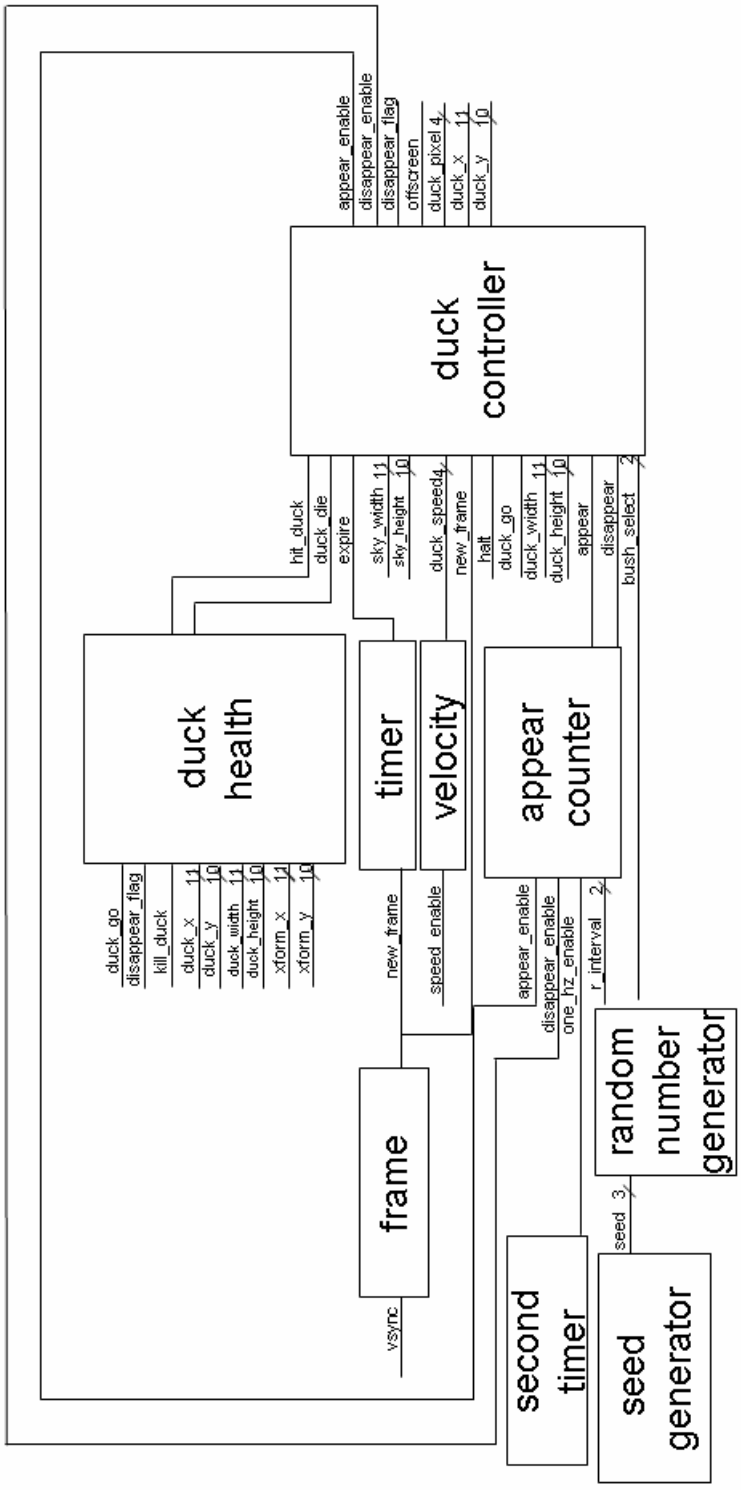

Figure 9 - Tiffany's Block Diagrams, Part II

Second Timer (by Tiffany Chen), in second_timer.v

The second timer is necessary for the system to properly time the number of seconds for the duck hunt game. The second timer takes as inputs the 40 MHz clock and the reset signal. As the pixel clock for 60 Hz video refresh rate is 40 MHz, the second timer has an internal 26-bit counter that increments at each positive clock edge. The output signal one_hz_enable is high for one clock cycle when the counter reaches 39,999,999, since the counter starts counting at 0. After it reaches 39,999,999, the counter resets to 0 and starts counting again. This counting method ensures that the output signal is high for one clock cycle every 40 million clock cycles, which is one second. This system enables the second timer to count the time in seconds. In addition, when the game is reset, the internal counter in the second timer module is set to 0, which allows the second timer essentially to reset its one_hz_enable output signal.

Random Number Generator & Seed Generator (by Tiffany Chen), in randgen.v and seedgen.v

The random number generator is to generate two signals: the bush_select signal that determines which bush the duck should fly out of and the r_interval signal that determines the time between the bush animation and when the duck appears from the bushes. These signals are generated by the coordination between the random number generator and the seed generator.

The random number generator module is generated using the CoreGenerator's linear shift feedback register (LFSR) utility in the Xilinx tool chain. The LFSR is initialized at 7 when the FPGA is loaded and outputs a 3-bit pseudorandom number, meaning that the number will appear to be random but is actually repeated as one number of a long cycle of numbers. The seed generator (seedgen.v) generates a 3-bit seed for the random number generator by incrementing the seed at every positive edge of the clock. Every time the LFSR is loaded through its load port on the load_enb signal, which is wired to the game's reset button, the seed from the seed generator is loaded into the LFSR as the current state through the pd_in port. Because the time at which the user presses the game's reset button is arbitrary, the seed is random, thus creating the unpredictability of the system.

When the LFSR's clock enable port (ce) is asserted, the last two bits of the current 3-bit number are XORed, and the next 3-bit number is generated by concatenating this number to the most significant bits of the current number. The resulting number is a 3-bit pseudorandom number that is generated at the pd_out port.

The first instantiation of the random number generator generates the r_interval signal that determines the time between the bush animation and when the duck appears from the bushes. Because this time ranges only from 0 to 3 seconds and a 2-bit LFSR generates only signal that counts down at every clock enable (asserted after the bush is done with its animation), a 3-bit LFSR is used instead to increase variability of the system. The LFSR continually outputs 0 if the seed is 0, so to prevent the seed from being 0, a check is performed before seeding the LFSR. If the seed is 0, the seed is set to

1. The seed is then fed into the random number generator, and each time the duck flies off the screen, a new 3-bit signal is generated. The r_interval signal is then determined by the two least significant bits of this 3-bit random number.

The second instantiation of the random number generator generates the bush_select signal that determines which bush the duck should come out of. The seed for this random number generator is the same seed fed to the other random generator incremented by 1, and the same check is performed to prevent the seed from being 0. If the seed for this random number generator is 0, the seed is set to 2 instead so that the two random number generators do not generator the same numbers. Each time a duck flies off the screen, a new 3-bit signal is generated, and the bush_select signal is determined by the two least significant bits of this 3-bit random number.

Colormap (by Tiffany Chen), in colormap.v

The colormap module takes as input the pixel information through the game_pixel signal and matches up the appropriate RGB values to output to VGA through the rgb signal. As the duck hunt game implements 16-color, the game_pixel is a four-bit value that corresponds to a certain 24-bit color rgb value determined using the colormap. The first set of eight bits of the rgb signal correspond to the VGA's red outputs, the second set of eight bits of the rgb signal correspond to the VGA's green outputs, and the last set of eight bits of the rgb signal correspond to the VGA's blue outputs. The mapping of colors for the colormap correspond to standard 16-bit color RGB values except for one: when the game_pixel signal is 1, its usual color on the standard 16-bit color map is replaced by a shade of brown necessary for coloring the ground for the duck hunt system. The transparent color for the images corresponds to a pixel value of 13 in the 16-color system, and this color is only used for the background color of the images stored in ROMs.

The hsync, vsync, blank, hcount, dhsync, dvsync, dblank, vcount, and switch signals are also fed to the colormap for the video display system. The horizontal sync, vertical sync, and blanking signals, which are hsync, vsync, and blank signals respectively, are generated through a xvga module according to the FPGA labkit's specifications for a 800x600 60 Hz video display. The xvga module also creates an 11-bit horizontal count, hcount, and a 10-bit vertical count that display the horizontal and vertical line numbers, and these signals are also fed into the colormap. The dhsync, dvsync, and dblank signals are the duck system's horizontal sync, vertical sync, and blanking signals, and when the appropriate combination of switches are implemented, the colormap outputs these signals as hs, vs, and b to the VGA. Otherwise the outputs hs, vs, and b are used to debug the colormap.

Frame (by Tiffany Chen), in frame.v

The frame module takes in the vertical sync signal (vsync) from the xvga and signals when a new frame is begins through new_frame. Because the vsync signal is active low, by detecting the negative edge, new_frame is one clock cycle high one clock

cycle after the vsync signal changes from high to low. This negative edge detection is implemented by checking the previous vsync value. If the previous vsync value is different from the current vsync value and the current vsync value is 0, the new_frame signal is asserted for one clock cycle. With this new_frame signal fed to other modules, images will only update their coordinates at the start of a frame.

Stationary Bush Controller (by Tiffany Chen), in bush_controller_tiff.v

Eight stationary bushes are created for the duck hunt game system, each of which are instantiations of the stationary bush controller. The bush controller takes in bush_x, bush_y, hcount, vcount, hsync, vsync, blank, bush_height and bush_width signals and reads the bush pixel information off the bush ROM. The bush ROM is 4-bit wide with a depth of 6650, as the bush is 50 pixels tall and 133 pixels wide, and the coe file was generated using Matlab to convert the pixel data into hexadecimal numbers corresponding to the appropriate color. Figure 10 illustrates an example of the 16-color bush image used for the duck hunt system.

Figure 10 - Bush Image

The 11-bit bush_x and bush_width and 10-bit bush_y and bush_height signals are asserted as wires in the labkit to specify the location and dimensions of the bush, and for each instantiation of the bush, different bush_x and bush_y coordinates are specified. The bush_x and bush_y signals determine the upper left corner location of the duck image. The horizontal count (hcount) is within the dimensions of the bush when it is between bush_x and bush_x + bush_width, and the vertical count (vcount) is within the dimensions of the bush when it is between bush_y and bush_y + bush_height. When both hcount and vcount are within the range of the bush image, the bush address is calculating by multiplying the bush width with the difference between vcount and bush_y and adding it to the difference between hcount and bush_x. In such a manner for every pixel within the range of the bush image, the bush address is calculated and the appropriate pixel information is correctly sent from the bush ROM.

The hsync, vsync, and blank signals are assigned to the duck's dhsync, dvsync, and dblank signals, respectively.

Cloud Controller (by Tiffany Chen), cloud_controller.v

In the duck hunt game system, three clouds scroll across the sky at constant speeds, and each cloud is generated using the cloud controller. The cloud controller takes

in a 11-bit initial x position (cloud_initx) and cloud width (cloud_width) information as well as a 10-bit initial y position (cloud_inity) and cloud height information (cloud_height). Additional signals that are fed into this module include the cloud_enable signal, 11-bit horizontal count (hcount), 10-bit vertical count (vcount), a 4-bit cloud speed (cloud_speed), the dimensions of the sky area through sky_height and sky_width, the east signal to signify the direction the cloud moves, the new_frame signal, and the halt signal. The cloud controller outputs the cloud's x and y coordinates a well as the 4-bit pixel data.

The cloud controller reads off pixel information from the cloud ROM, which is 4-bit side with a depth of 4320, as the cloud is 108 pixels wide and 40 pixels tall, and the coe file was generated using Matlab to convert the pixel data into hexadecimal numbers corresponding to the appropriate color. Figure 11 illustrates an example of the 16-color cloud image used for the duck hunt system.

Figure 11 - Cloud Image

When the horizontal and vertical counts are within the dimensions of the cloud, the address is calculated by multiplying the cloud width with the difference between vcount and the y coordinate of the cloud and adding it to the difference between hcount and the x coordinate of the cloud. In such a manner for every pixel within the range of the cloud image, the cloud address is calculated and the appropriate pixel information is correctly sent from the cloud ROM.

On reset, the cloud's top left corner is initialized according to cloud_initx and cloud_inity. At each new frame, as signified by the new_frame signal, the cloud's x and y coordinates are calculated using an 11-bit cloud_x count. This check is necessary to ensure that the cloud does not change its coordinates in the middle of the frame so that one complete image of the cloud is displayed before the cloud moves. If the cloud is traveling east and the cloud's x coordinate is within the range of the sky, the cloud increments its x coordinate by a number of pixels per frame specified by cloud_speed to move the cloud east. Otherwise, if the cloud scrolls off the screen, the x coordinate of the cloud is set to 0 so that the cloud continues to scroll across the sky. Likewise, if the cloud is traveling west and the cloud's x coordinate is within the range of the sky, the cloud decrements its x coordinate by a number of pixels per frame specified by cloud_speed to make the cloud move west. If the cloud hits the left edge of the screen, the x coordinate of the cloud is set to the width of the sky so that the cloud continues to scroll across the sky.

If the halt signal is asserted, meaning that either the player has won or lost the game, the cloud is stationary: its x coordinate is kept at the same value. The hsync, vsync, and blank signals generated from the xvga are assigned to the duck hunt system's corresponding dhysnc, dvsync, and dblank signals.

Duck Subsystem (by Tiffany Chen)

The duck subsystem consists of the timer module, appear counter, duck health, velocity, and duck controller modules. Together, they coordinate the timing and animation of the duck.

Timer, in timer.v

The timer is responsible for timing the rate at which the duck image changes to create the duck animation effect. The timer takes in the new_frame signal and asserts the expire signal every 16 frames. A decrement counter counts keeps track of the number of frames. Each time a new frame is created, the counter decrements, and when the count reaches 0, the expire signal is asserted and reset to 16. This method ensures that the expire signal is high once every 16 frames, so that every 16 frames, the duck state transitions and a different duck image is displayed accordingly.

Appear Counter, in appear_counter.v

The appear counter is responsible for counting to r_interval number of seconds, as specified by the random number generator, for the duck to appear. Furthermore, if the user does not hit the duck 8 seconds after the duck emerges from the bushes or after the previous hit, the duck flies away off the screen. The module uses the r_interval signal from the random number generator, the one_hz_enable signal from the second timer, appear_enable and disappear_enable signals form the duck controller. The appear counter asserts an appear signal and disappear signal to the duck controller.

Essentially, the appear counter is two decrement counters combined: count down to r_interval, and the other to count down to eight seconds. When the game is reset, the internal counters for the duck to appear (appear_count) and disappear (disappear_count) are reset. To properly time the number of seconds between the bush animation and the duck's emergence from the bushes, the appear_count is decremented each second as specified by the one_hz_enable signal, and the appear signal is asserted only when appear_count reaches 0, after which the appear_count is reset to r_interval. The appear_enable signal, which is asserted after the bush animation, also resets the counter to r_interval. In such a manner, the appear signal is high only after r_interval number of seconds.

As for the disappear signal, the disappear_count is decremented each second as specified by the one_hz_enable signal from the second timer, and the disappear signal is asserted only when disappear_count reaches 0, after which the disappear_count is reset to 8 seconds. In such a manner, the disappear signal is high every 8 seconds. The disappear_enable signal, which is asserted each time the duck is hit or each time the duck emerges from the bushes, also resets the disapper_count counter.

Velocity, in velocity.v

The velocity module determines the speed at which the duck flies across the sky. The first three ducks fly at slow speed, the second three ducks fly at medium speed and the last two ducks fly at high speed. This is enough ducks for the user to shoot within the 45 second time frame of the game.

The velocity module keeps track of the number of ducks. On each reset, the duck count is reset to 0, and at each clock cycle, the duck count is checked to see if it is between 0 to 2, 3 to 5, or 6 to 7, inclusive. When the duck count is between 0 and 2, the duck speed is 2 pixels/frame; when the duck count is between 3 and 5, the duck speed is 4 pixels/frame, and when the duck count is between 6 and 7, the duck speed is 6 pixels/frame. Each time the speed_enable signal, which is asserted each time the bush finishes its animation, is high, the duck count increments. The duck count is also output through the velocity module onto the hex display, and the duck speed is sent to the duck controller module.

Duck Health, in duck_health.v

The duck health module keeps track of the number of hits the duck has sustained and indicates to the duck controller when the duck should die. The duck requires three shots to be killed, and the duck health module uses an internal counter (duck_shots) to keep track of the number of shots sustained. When a new duck appears the number of shots the duck has sustained is reset. In addition to the duck coordinates and dimensions, the x and y coordinates of the laser dot are also inputs to the module as xform_x and xform_y.

The duck health module also accounts for the lag the laser dot system takes to calculate the x and y coordinates of the laser. When the duck is not immune to shots and the trigger of the gun is pulled (kill_duck), an internal counter is set to 4 million and decrements at each clock cycle. Until the internal counter reaches 0, the x and y coordinates of the laser dot is compared with the x and y coordinates of the duck as well as its dimensions: if the x and y coordinates of the laser dot fall within the range of the duck image, a hit is registered, and the internal counter is set to 0 so that the counter will reset at the next kill_duck signal. This system of registering a hit allows the module to sample the x and y coordinates of the laser dot for 1/10 of a second after the trigger signal is asserted to allow enough time of the laser dot finder system to calculate the coordinates of the laser dot.

When a hit is registered, a hit_duck signal is asserted for one clock cycle, and when the duck dies, the duck_die signal is asserted for one clock cycle. After the duck has sustained three shots or is flies off the screen, as determined through the signal disappear_flag from the duck controller because the user did not hit the duck in time, the duck is immune to shots, and the hits are not registered in the internal counter duck_shots.

Duck Controller, in duck_controller.v

The various signals from the rest of the duck subsystem is fed to the duck controller module, which is responsible for the animation of the duck on the screen. The duck controller is a FSM with 14 different states for the duck: the duck with its wings up heading northeast (State 0); the duck with its wings down heading northwest (State 1); the duck with its wings down heading northwest (State 2); the duck with its wings up heading northwest (State 3); the duck with its wings up heading southeast (State 4); the duck with its wings down heading southeast (State 5); the duck with its wings up heading southwest (State 6); the duck with its wings down heading southwest (State 7); three states for when the duck is hit (State 8, State 9, and State 10); the duck dying (State 11); the duck flying away (State 12); and the duck waiting to come out of the bushes (State 13).

Six duck images are stored in separate ROMs: the duck with its wings up facing left, the duck with its wings down facing left, the duck with its wings up facing right, the duck with its wings down facing right, the duck in a state of shock when it is hit, and the duck with its head pointed towards the ground when it is dead. Each ROM is 4 bits wide with a depth of 12100, as each duck image is 110 pixels wide and 110 pixels long. The coe files were generated using Matlab to convert the pixel data into hexadecimal numbers corresponding to the appropriate color. Figure 12 illustrates the images stored in the ROMs for the duck.

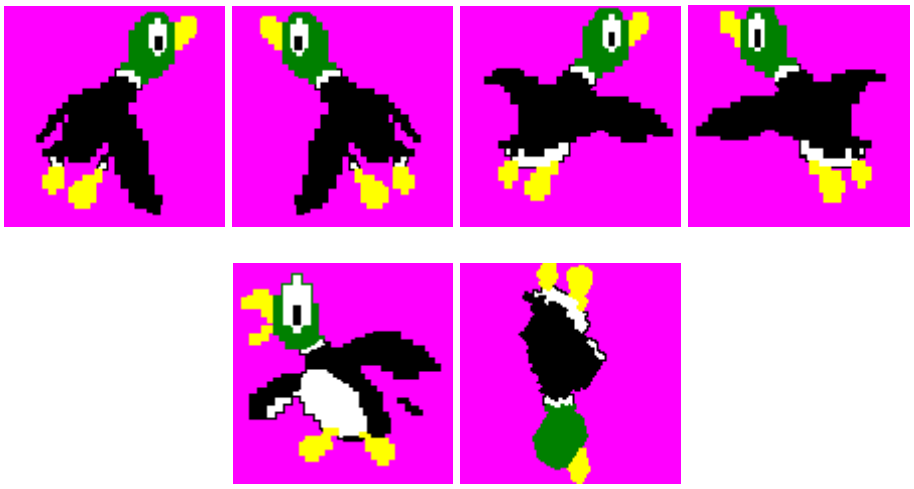

Figure 12 - Duck Images

When the horizontal and vertical counts are within the dimensions of the duck image, the address is calculated by multiplying the duck width with the difference between vcount and the y coordinate of the duck and adding it to the difference between hcount and the x coordinate of duck. In such a manner for every pixel within the range of the duck image, the duck address is calculated and the appropriate pixel information is correctly sent from the duck ROM.

When the game is reset or when a new duck appears, the x coordinates of the duck are determined by which bush is selected through the signal bush_select. The y coordinate of the duck is set to the height of the sky so that the duck image is out of the range of the sky. The duck in initially facing northeast, and its state is set to its dying

state so that the x and y coordinates of the duck do not change. When the duck_go signal is asserted, the duck begins to fly out of the bush traveling northeast.

The direction of the duck and its x and y coordinates are kept track of using internal registers. The disappear_flag high when the duck is flying away to signal to the duck health module that the duck is immune to shots. When the duck is out of the range of the sky, an offscreen signal is asserted. The disappear_enable signal is asserted when the appear counter asserts the appear signal and the duck is in its waiting state (State 13), or when the duck is flying away (State 12). The duck_die_flag is a signal that registers when the new_frame signal is low and when the duck_die signal has been declared and is reset after the duck enters the its dying state (State 11) or when the duck is off the sky (offscreen signal is high). This signal prevents the duck controller from missing the duck_die signal when the new_frame signal is not asserted.

The 4-bit duck_pixel output is determined by the state and direction the duck is in. When the duck is in any state in which its wings are supposed to be up, the south and east flags are checked to determine the direction the duck is supposed to be pointing and assigns duck_pixel to the corresponding ROM output that has the duck with its wings up. Likewise, when the duck is in any state in which its wing are supposed to be down, the south and east flags are checked to determine the direction the duck is supposed to be pointing and assigns duck_pixel to the corresponding ROM output that has the duck with its wings down. When the duck is in any one of its 3 hit states, the duck_pixel is assigned to the output of the ROM that stores the duck in its state of shock. When the duck is dead, the duck_pixel is assigned to the output of the ROM that stores the dead image of the duck. Finally, when the duck in its state to fly away off the screen, the duck_pixel to the ROM output according to which direction it is facing using the south and east flags.

For the normal animation of the duck flapping its wings, when both the new_frame and expire signals are asserted, if the duck is in one of its wings-up states (States 0, 3, 4, 6), the duck transitions between one of its wings-up state to the corresponding wings-down state (States 1, 2, 5,7), or if the duck is in one of its wings-down state, transitions to the corresponding wings-up state. This creates the effect of the duck flapping its wings as it moves across the sky.

When the game is over or when the user wins, the halt signal is asserted and the duck controller maintains its state. Otherwise, when the duck is in its waiting state and the appear signal from the appear counter is asserted, the duck transitions to the state with its wings up and heading northeast (State 0). When the duck is hit, the duck transitions to its first hit state (State 8), and at expire signal transitions to its second hit state (State 9), which then enters the third hit state (State 10). These multiple hit states allow the duck to display its state of shock for a longer period of time so that the user clearly knows when he or she has hit the duck. After three hits, the duck transitions to its dying state (State 11). Otherwise, if the disappear signal is asserted and the duck is not in its waiting state or in its dead state, the duck enters the state where it flies off the screen (State 12).

The duck controller is also responsible for changing the coordinates of the duck so that the duck flies across the sky. When the halt signal is asserted, the duck's x and y coordinates are kept the same so that the duck freezes on the screen. Otherwise, when the duck is in its waiting state (State 13), its x coordinate is set according to which bush is selected form the bush_select signal, and the y coordinate is set so that the duck is just below the topmost edge of the bushes.

When the duck enters its dying state (State 11), if this is the first frame the duck is in this state, the duck controller increments the duck's x coordinate (duck_x) by duck_speed as determined by the velocity module if the duck is traveling east; if the duck is traveling west, the duck controller decrements the duck's x coordinate by duck_speed. This creates a smooth transition between the duck's dying state and its previous state. The duck_die_flag is reset to 0 so that after the x coordinate of the duck is incremented or decremented accordingly, the x coordinate of the duck will no longer change on the next frames until the duck falls off the area of the sky. When the duck is in its dying state, the y coordinate (duck_y) is incremented by a constant parameter of 4 pixels per frame so that the duck falls down from the sky.

When the duck is in its state to fly away (State 12) because the user did not manage to hit the duck in time, duck_x is kept at the same value, while the duck y is decremented by the constant parameter of 4 pixels per frame so that the duck flies rises out of the sky, indicating to the user that his or her time to shoot that particular duck has run out.

Otherwise, if the duck is in one of its regular animation states with its wings flapping up and down (State 0-7), the x coordinates of the duck is incremented if the duck is traveling east and decremented if the duck is traveling west. If the duck reaches the right edge of the screen while it is hitting east, the duck's x coordinate is set to the sky width minus the duck width so that the entire image of the duck is displayed to the right-most part of the screen possible. Similarly, if the duck is traveling west and the duck reaches the left edge of the screen, the duck's x coordinate is set to 0 so that the entire image of the duck is displayed to the left-most part of the screen possible.

Analogous logic applies to determining the y coordinate of the duck when it is in one of its regular animation states with its wings flapping up and down. The y coordinate of the duck is incremented if the duck is traveling south and decremented if the duck is traveling north. If the duck reaches the bottom edge of the sky, the duck's y coordinate is set to the sky height minus the duck height so that the entire image of the duck is displayed to the bottom-most part of the sky possible. Similarly, if the duck is traveling north and the duck reaches the top edge of the sky, the duck's y coordinate is set to 0 so that the entire image of the duck is displayed to the top-most part of the sky possible.

The hsync, vsync, and blank signals from the XVGA are assigned to the duck hunt's dhsync, dvsync, and dblank signals.

Video Controller (by Tiffany Chen), in video_controller.v

The video controller takes in all the pixel and x and y coordinate information from all the modules related to the video display for the duck hunt game system and prioritizes the pixel information so that the duck flies behind the clouds and appears behind the bushes. The video controller also controls the transparency of the backgrounds of the images from the duck, clouds, and bushes. In addition, the video controller implements a simple FSM that transitions between the title frame and the duck hunt game and controls the speed at which the clouds move.

The title image, game over image, and you win image are instantiated in the video controller. These images are illustrated in Figures 13-15 and were generated using Matlab to create the coe files to load the ROMs with.

Figure 13 - Duck Hunt Title Image

Figure 14 - Duck Hunt Game Over Image

Figure 15 - Duck Hunt You Win Image

Each image is stored in a separate ROM that is 4 bits wide with a depth of 5000, as each image is 100 pixels wide and 50 pixels tall.

When the game is reset, the game is in the start mode. When the start_game signal is asserted, the game transitions to the game mode.

When the game is in the start mode, the title frame is displayed. When the hcount and vcount signals are within the range of the dimensions of the title image, the address for the title image is calculated. For the start frame, the title image is magnified horizontally and vertically by eight times to fill most of the screen. The address to the ROM is calculated by first bit-shifting the difference between vcount and the y coordinate of the upper left corner of the title image to the right by 3 bits. This number is then multiplied by the width of the title image and added to the difference between hcount and the x coordinate of the upper left corner of the title image bit-shifted by 3 bits. The resulting number is the address sent to the ROM.

The duck image is also displayed on the start frame and magnified horizontally and vertically two times, but instead of bit shifting by 3 bits, only a 1-bit shift is necessary because the image is only being magnified by a factor of two.

The game_pixel information for each hcount and vcount value is then determined by a series of if-else checks that check if the pixel lies within the ranges of pixels required for the various images. The topmost layer consists of the title image, followed by the duck image and the character strings. If none of the game pixels lie within the ranges of pixels required for the various images, then the pixel is set to sky blue for the background color.

When the video controller is in the game mode, analogous if-else checks determine the layering of the images. The blob that records the last position of the shot is on the topmost layer, followed by the you win or game over images that are displayed if the halt signal is declared. If the win signal is declared while the halt signal is declared, the game pixel is assigned to the output of the ROM that stores the you win image; otherwise, if the gameover signal is declared while the halt signal is declared, the game pixel is assigned to the output of the ROM that stores the gameover image. Both images are magnified horizontally and vertically using the method previously described, except involving bit-shifting to the right by 2 bits because the image is magnified by four times.

For determining the game pixel data when the game is running, the video controller first determines the pixel information for the sky area. Clouds are layered first, followed by the bushes (both animated and stationary), and finally by the ducks. When the hcount and vcount are checked to see if they lie within the range of the various images, the transparency color (color 13 in the colormap) is also checked so that if the pixel is equal to color 13, the image behind it is displayed. If no pixel information is assigned for a given hcount or vcount, the sky color is displayed.

Then, the video controller determines the pixel information for the scorekeeping area. The title image, magnified once in the horizontal and vertical directions, is displayed at the lower left corner of the scorekeeping section of the string, which also displays the time remaining and score. When the hcount and vcount are checked to see if they lie within the range of the character display strings, additional checks to see if the pixel is black or white are conducted to change the color of the character string background and text so that the contrast is reduced for more accurate tracking of the laser dot. If no pixel information is assigned for a given hcount or vcount, the brown color for the ground is displayed.

Duck Subsystem (by Spencer Sugimoto)

scorekeeper.v

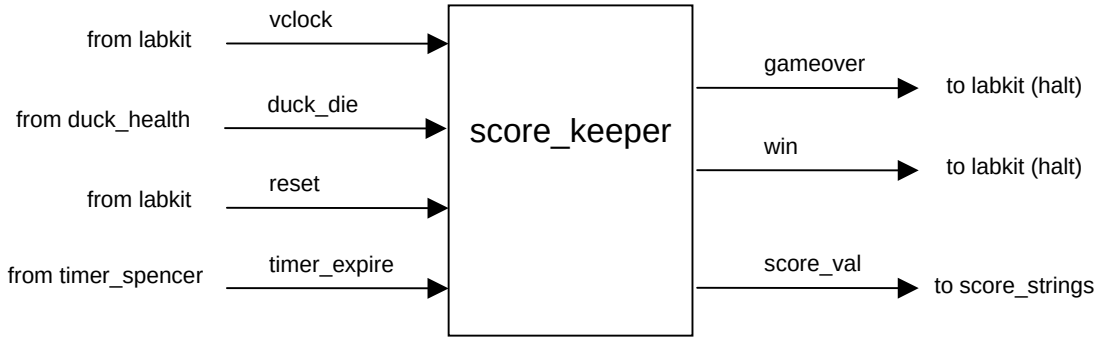

Figure 16 - Scorekeeper Block Diagram

Whenever the x and y coordinates of the Dot Finder system are matched with the x and y coordinates of a duck three times, upon the third hit, the duck dies. The duck_health module sends out a duck_die signal, high for one clock cycle. The scorekeeper module detects this single high pulse and increments the internal stored value of the score. The score increments in this way, until the scorekeeper disallows any more valid shots, when the user wins or loses the game. Controlling the rules of the game, the scorekeeper module store the high score required to win the game as a parameter within the module. If enough ducks are killed to meet the required score, the win signal is set high and remains high to keep the game halted, until a reset is input.

The scorekeeper is also responsible for the losing of the Lasershot game. When the timer of the game is expended, the timer module set the timer_expire high. The scorekeeper module detects this signal and the scorekeeper's gameover signal is set high. As the scorekeeper continues to run, it keep the gameover signal high, until a reset is indicated by the user. This halt of the entire game is exactly the same as the halt initialized by the win signal.

Upon a high reset signal, the scorekeeper module returns the score to zero and brings the game back from being halted by returning both the gameover signal and the win signal to zero values. This resets all conditions, allowing the whole game to reset, just like the original Nintendo Entertainment System's reset button feature. With all of the logic stated above, the scorekeeper module allows for accurate, consistent, and fair execution of the Lasershot game.

cstringdisp.v

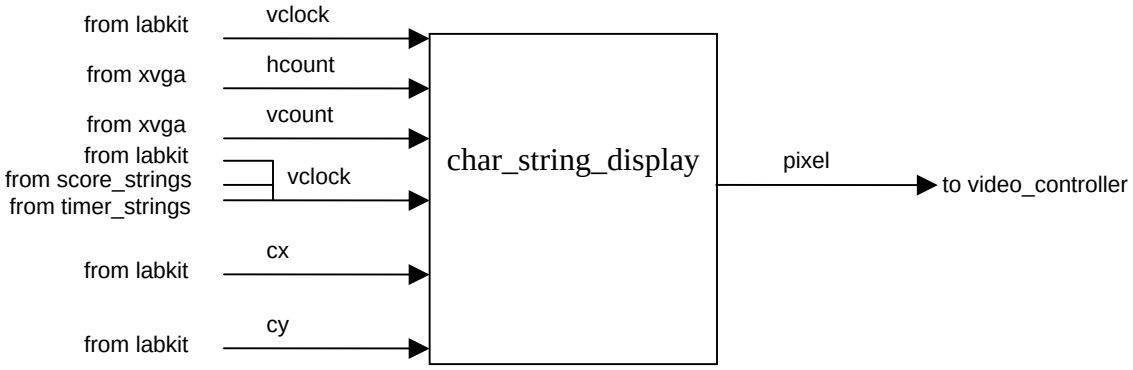

Figure 17 - cstringdisp Block Diagram

The char_string_display module takes in cstring, a string of 8-bit characters, and displays the string on the screen at specific x and y coordinates. The char_string_display module takes in cx and cy, which are the coordinates, where the top left corner of the text should be displayed. The characters are 8x12 pixels in size, and they are all stored in font_rom. Upon display, the characters are doubled in size to 16x24 pixels on the actual screen. Due to the fact that all of the characters are stored in the ROM, exact addressing

is very important. Both the number of characters in the cstring and the number of bits required to count that number of characters in cstring are both parameters that must be redefined for every instantiation of the module. Normal binary values are not legal inputs to the char_string_display module, so special translation modules must be constructed to communicate with it.

score_strings.v

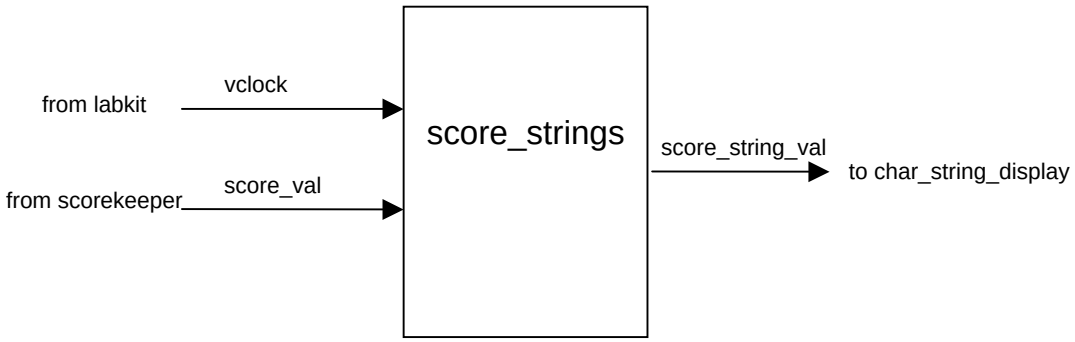

Figure 18 - score_strings Block Diagram

The score_strings module acts as the interpreter between the char_string_display module and the scorekeeper module. As stated above, the char_string_display module requires an input in the form of a string. Even when outputting numbers to the screen, binary number value cannot be input to the char_string_display module. In order to facilitate communication between the scorekeeper module and the char_string_display module, the score_strings module takes in the 4-bit binary score_val value from the scorekeeper and outputs score_string_val, the appropriate, corresponding string. For example, an input of 4'd3 will yield the string "03" to the char_string_display module. The final Lasershot game only requires a score value of six for the user to win the game; however, the score_strings module is ready for expansion, accommodating up to a score more than double the current value, 16.

Divider Module

The divider takes in a reset signal, a clock, and outputs the second counter *one_hz_enable*. Using the 40 mhz clock, the divider module simply counts up to 39999999, which take roughly one second. The timing of the Lasershot game is terms of seconds, so the Divider module is very simple, but crucial to the game. At reset, the count goes back to zero, and the second counter begins over again.

timer_spencer.v

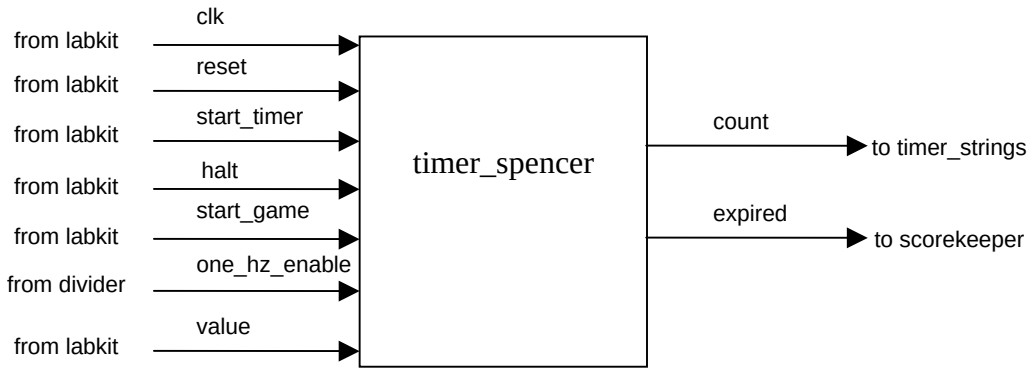

Figure 19 - timer_spencer Block Diagram

The timer_spencer module takes in a 6-bit value, up to which it will count before outputting an expired signal. The one_hz_enable signal (provided by the divider module) allows the module to increment once every second. The timer_spencer module begins to count up from zero at a reset signal, the start_timer signal, or the start_game signal. Once initialized, the counter counts to the input value. When the value is reached, the output expired signal goes high, which remains high until any of the aforementioned begin signals also goes high. It is important to note the function that once the count reaches the input value, the count remains equal to the input value, until reset. When the expired signal goes high, the scorekeeper outputs a constant gameover signal, which halts the entire game, so the count should not reset until a manual reset comes from a user.

timer_strings.v

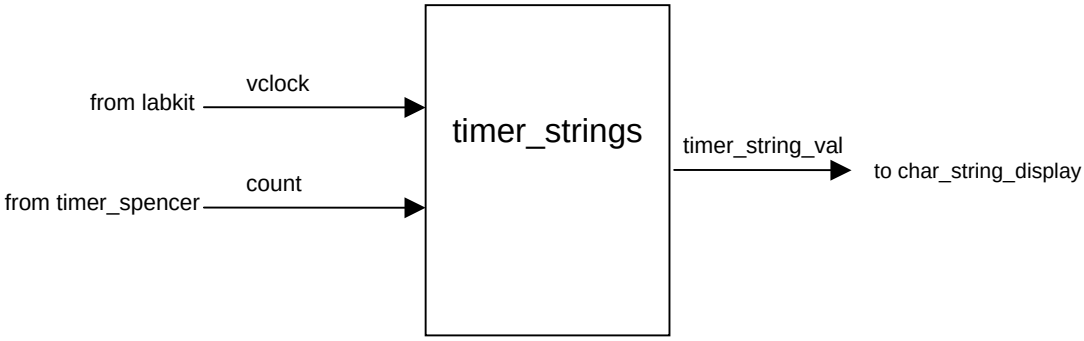

Figure 20 - timer_strings Block Diagram

The timer_strings module is the interpreter between the char_string_display module and the timer_spencer module. The timer_strings module is similar, in concept, to the score_strings module, but functions differently. The timer_strings module takes in a 6-bit binary timer_val value and outputs a 16-bit string, but the values are inverted. The timer_spencer module counts up from zero, but the rules of the Lasershot are best suited for a countdown timer. In order to accommodate this functionality, the timer_strings module first displays the time limit of the Lasershot system, then displays decremented values. For example, an input of 6'd0 will yield a "45" output, and an input of 6'd1 will yield a "44" output. With six bits, the timer_val can reach up to 63, but because the lowest input outputs the highest string, the timer strings must be completely reworked, if the time_val needs to start higher than 45.

Bush Animation

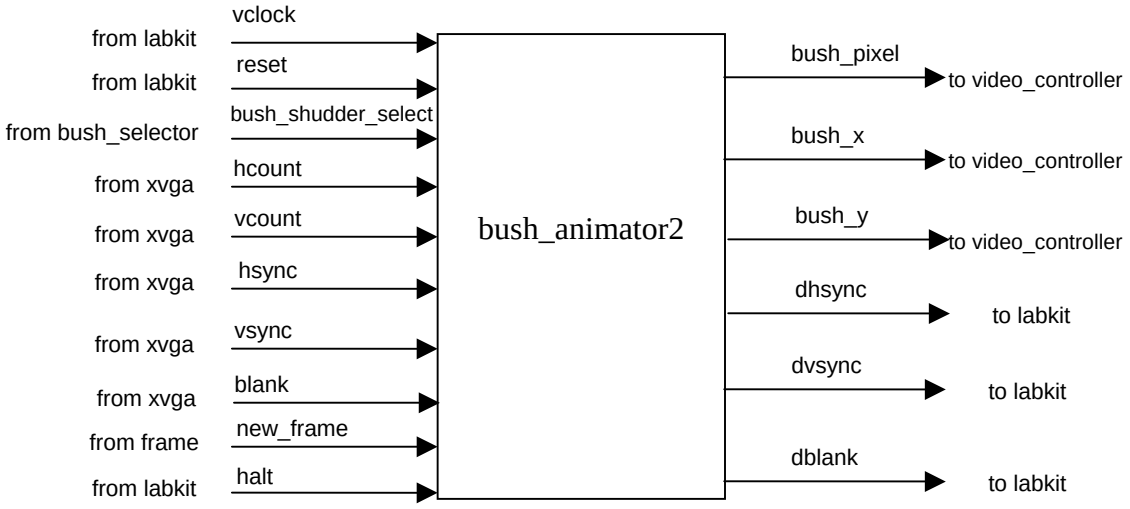

Figure 21 - Bush Animation Block Diagram

The bush_animator2 module instantiates a moving bush on the screen of the game. The video signals hsync, vsync, and blank are input and assigned to those of the Lasershot game system, dhsync, dvsync, and dblank. The bush_animator2 module takes in a set of x and y coordinates and outputs another set of x an y coordinates. The bush_animator2 module takes in 11-bit hcount and 10-bit vcount, the screen's current x and y values, which are used to test whether or not the pixel in the bush ROM should be output to the video controller. The bush_animator2 module outputs the 11-bit bush_x and 10-bit bush_y, which are the current x and y coordinates of the bush. The bush_x and bush_y coordinates dictate the current location of the top left pixel of the bush image that is being displayed on the screen. The parameters telling the size of the bush image are hard-coded into the module: `BUSH_WIDTH` and `BUSH_HEIGHT`. If the pixels of the bush are within the following constraints:

```
((bush_x <= hcount) and
 (hcount < bush_x + BUSH_WIDTH) and
 (bush_y <= vcount) and
 (vcount < bush_y + BUSH_HEIGHT)),

the bush's pixel should be output from the bush_animator2 module.
```

Beyond determining whether or not the module should be outputting the pixel of the bush ROM at a given position, the bush_animator2 module is responsible for the motion of the bushes. Before trying to move the bush, the bush_animator2 module checks that the new_frame signal is high, which ensures that the movement of the bush sprite is synchronized with the vsync signal.

The bush_animator2 module handles movement in the x- and y- planes separately. In the x-plane, the bush sprite sways left and right slowly, as if with light breezes within the game. The pixels/cycle speed of the swaying is stored in the parameter SWAY_SPEEDX. However, if the bush_animator2 module receives a high bush_shudder_select signal, this means that a duck is moving around in the bush, causing the bush to shudder vigorously. In order to show this x behavior in the bush sprite, the bush_animator2 module increases the speed of the left-right movement by 40 times to SHUDDER_SPEED. This creates the horizontal shuddering effect.

In the y-plane, the bush sprite remains at the constant parameter value, `BUSH_Y_START,` during the slow swaying effect. When the bush_animator2 module receives a high bush_shudder_select signal, the duck also causes the bush to rustle up into the sky just a bit. The bush_animator2 module shows this behavior of the sprite by subtracting the parameter value SHUDDER_Y from the y-value of the bush just a few pixels. The overall effect of the bush_shudder_select signal on the bush is a comprehensive shuddering motion of the bush sprite, clearly indicating to the user that the bush contains a duck, ready to fly away. The shuddering lasts only as long as the bush_shudder_select signal is high, before returning to the calm swaying.

The range of motion of the swaying, the speed of the swaying, the range of motion of the shuddering, and the speed of the shuddering are all abstracted into parameters, and so they are easy to modify for more or less dramatic bush movement (depending on the weather in the FPGA). The only way to break the swaying and shuddering of the bushes is with a reset or with a halt, which is a win or a gameover signal.

bush_selector.v

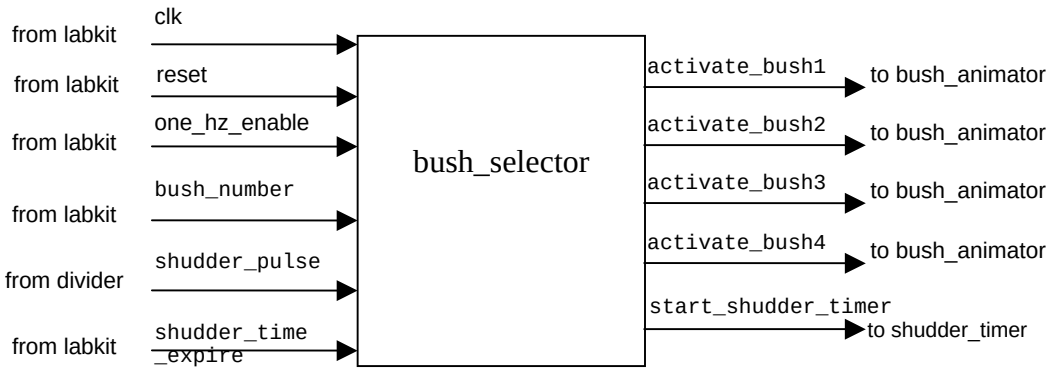

Figure 22 - Bush Selector Block Diagram

The bush_selector module receives the bush_number signal from the random number generator, choosing which bush will shudder. Upon the one-cycle high shudder_pulse, whichever activate_bush# that corresponds to the bush_number will shudder. At the same time that the shudder_pulse goes high to start the bush shuddering, the bush_selector module sends out the start_shudder_timer signal, which starts the shudder_timer on the labkit. When the shudder_timer counts to its desired value, it sends out the shudder_time_expire signal, which then causes whichever activate_bush# is shuddering to stop shuddering.

Audio System

lab4audio.v
synchronizer.v
pulse_doubler.v

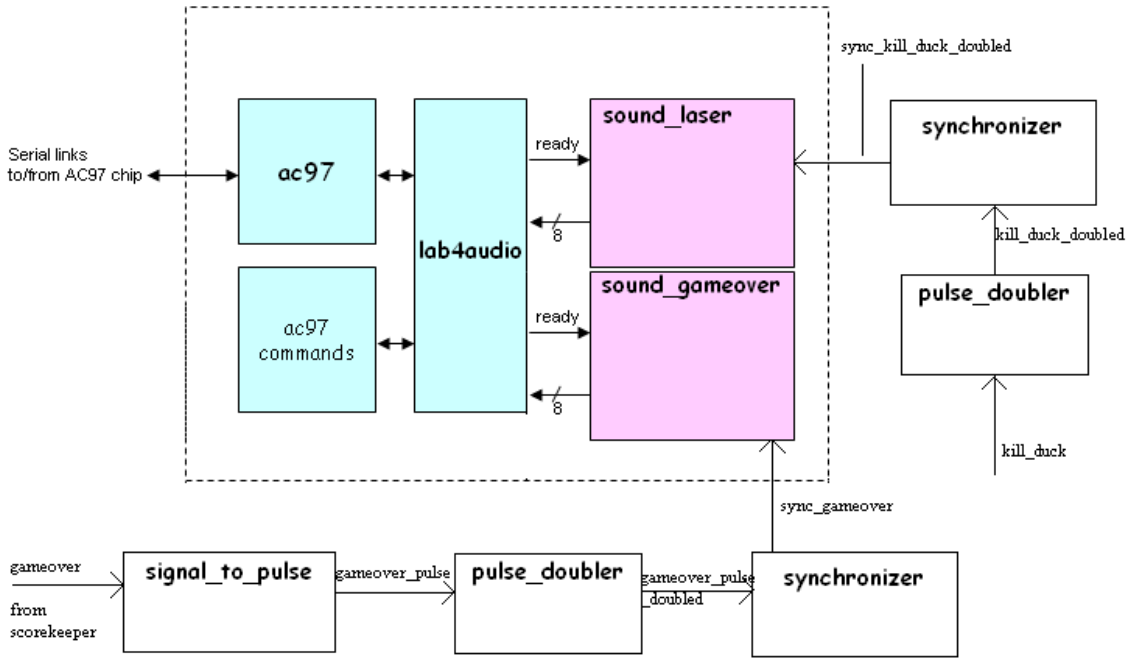

Figure 23 - Audio System Block Diagram

The lab4audio module uses the ac97 module and the ac97commands modules to create the interface between labkit.v and the ac97 audio chip on the labkit, itself. Since the Lasershot game does not require extremely high-quality sound, the sound is setup to be monaural, by setting both the left and right output data equal to the same output data. Within the lab4audio module, the volume of the overall sound output is handled by the ac97commands module.

One problem with using the ac97 sound processing chip is that it requires a 27 Mhz clock. The Lasershot video all runs on a 40 Mhz clock. Running these two different clocks, timing issues are inevitable, but fixable. A single high pulse emitted at 40Mhz, is simply too fast for a module running at 27 Mhz to reliably detect. First, the pulse_doubler module doubles the length of a single pulse to a pulse high for two clock cyles. The pulse_doubler module uses a simple counter to delay the signal high for one extra clock cycle. Although the pulse is now long enough that it might be reliable as an input to a module running at 27 Mhz, just to be sure, the signal is then passed through the synchronizer module, which synchronizes it with the 27Mhz clock. Using this 2-stage process, the ac97 is able to integrate seamlessly with any of the signals in the 40Mhz video of the Lasershot game.

sound_laser (sound_laser_backup.v)

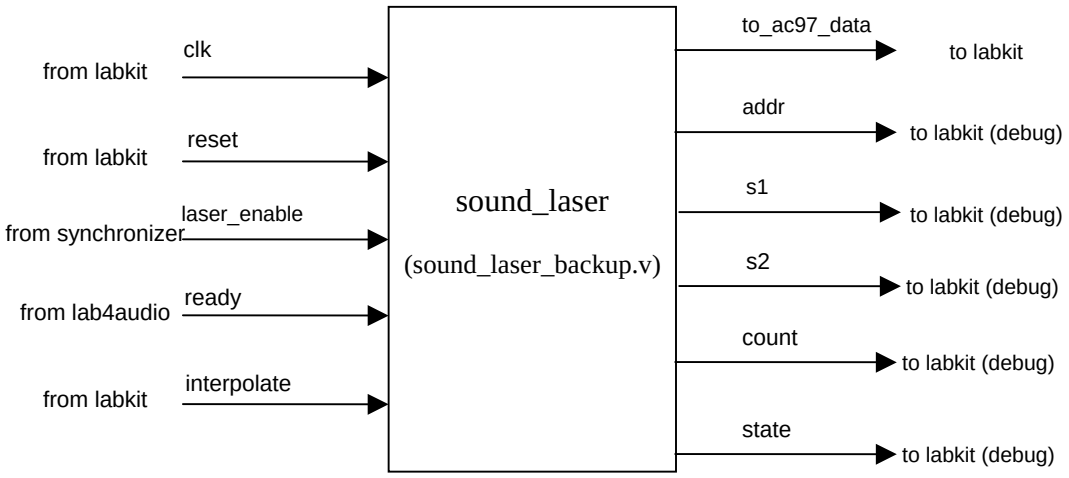

Figure 24 - sound_laser Block Diagram

The sound_laser module takes as inputs the laser_enable, ready, and interpolate signals. Using the ac97 audio chip to output sound, the sound_laser module definitely requires the synchronization described above to be triggered. The input laser_enable, which initializes the playback of the laser gun sound, gets the doubled and synchronized pulse sync_kill_duck_doubled from the synchronizer. Before being doubled and synchronized into sync_kill_duck_doubled, the original kill_duck signal is tied to the trigger of the physical laser gun, and so it is a pulse. The input ready signal is the ac97's ready signal, telling the sound_laser module when the sound driver is ready.

The interpolate signal chooses whether or not to use interpolation when playing back the laser sound effect. The interpolation codec , `((6-count)*s1 + count*s2), interpolates the data during the 5 cycles between each new pixel value, changed every six cycles.` Without interpolation the sound_laser module outputs the 8-bit, 8 khz audio data stored in the ROM, cycling the sound data from the ROM every six cycles, managed by the value of count. With interpolation, the sound is up-sampled to 48 khz by the addition of 5 cycles of data. In the labkit, interpolate is set a parameter = 1'b1 because the interpolated sound is much better than the non-interpolated data. In order to gain optimal functionality, the ROM must not try to output an address past its last data address, so the calibration of the ROM is very important. One second of playback requires approximately 8000 bits. The laser gun sound is approximately half of one second and exactly 4027 bits.

The sound_laser module contains a two-state FSM. The first state is the reset state, in which count, the laser_addr, and the stored data values s1 and s2 are all reset to zero. when the laser enable pulse is active, the FSM transitions to the Playback state. In the Playback state, if the ac97 sends its ready signal, the audio output cycle begins. Every 6^{th} count of the clock, the laser_addr is incremented, changing the audio data being output. In the chain of else-if statements, there is one condition that if the laser_enable signal goes high during the playback state, the laser_addr is reset to zero, so a new laser

sound interrupts the previous one, starting the sound for the most recent trigger pull. Otherwise, the sound plays until completion of the final laser_addr, or until a reset signal interrupts and returns the FSM to the Reset state to wait for the next signal from the trigger.

sound_gameover.v

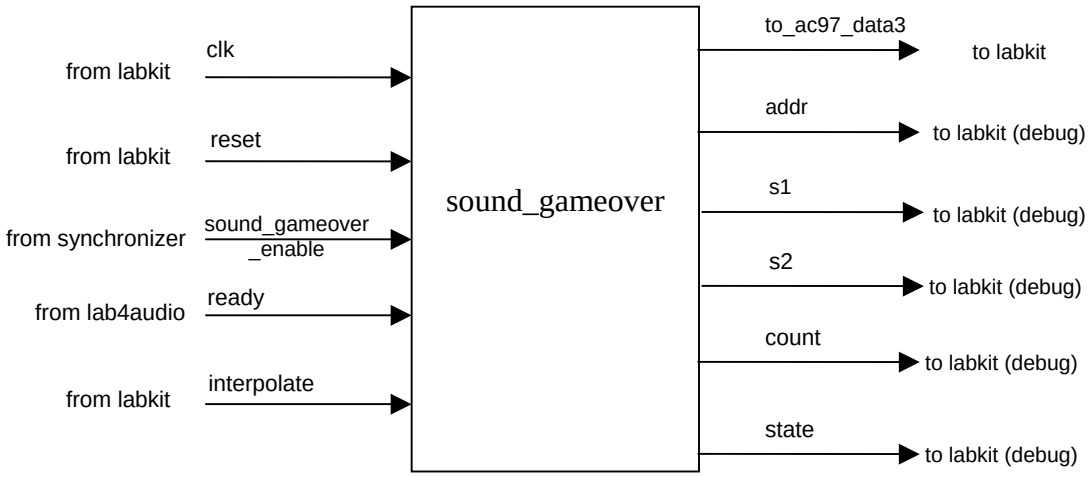

Figure 25 - sound_gameover Block Diagram

The sound_gameover module takes in gameover _enable, ready, and interpolate signals. Like the sound_laser module, the sound_ gameover module uses the ac97 audio chip to output sound. Therefore, the sound_gameover module also requires synchronization. The input gameover _enable, which initializes the playback of the gameover sound, gets the doubled and synchronized pulse, gameover_sync from the synchronizer. The original gameover signal, however, is not a pulse, so before it is synchronized, it is first converted to a pulse in 40mhz with the signal_to_pulse module, then doubled with the pulse_doubler module, and finally, synchronized with the 27 mhz clock for use with this audio module. The ready signal is the ac97's ready signal, telling the sound_gameover module when the sound driver is ready.

The interpolate signal chooses whether or not to use interpolation when playing back the laser sound effect. The interpolation codec ,`((6-count)*s1 + count*s2), interpolates the data during the 5 cycles between each new pixel value, changed every six cycles.` Without interpolation the sound_laser module outputs the 8-bit, 8 khz audio data stored in the ROM, cycling the sound data from the ROM every six cycles, managed by the value of count. With interpolation, the sound is up-sampled to 48 khz by the addition of 5 cycles of data. In the labkit, interpolate is set a parameter = 1'b1 because the interpolated sound is much better than the non-interpolated data. In order to gain optimal functionality, the ROM must not try to output an address past its last data address, so the calibration of the ROM is very important. One second of

playback requires approximately 8000 bits. The laser gun sound is approximately half of one second and exactly 12000 bits.

The sound_gameover module contains a two-state FSM. The first state is the reset state, in which count, the gameover_addr, and the stored data values s1 and s2 are all reset to zero. When the gameover_enable pulse is high, the FSM transitions to the Playback state. In the Playback state, if the ac97 is sending its ready signal, the audio output cycle begins. Every 6^{th} count of the clock, the laser_addr is incremented, changing the audio data being output. The gameover sound plays until completion of the final gameover_addr bit, or until a reset signal interrupts and returns the FSM to the Reset state to wait for the next signal user to lose the game.

Dot Finder System

Due to the nature of the modules in the dot finder system, simulations were of limited use during development because ModelSim could not emulate any of the decoder IC's or the ZBT RAM. Instead, most of the testing and debugging were carried out though the 16-character hex display on the labkit. If a particular module did not function properly, internal signals were routed as output signals and were examined on either the hex display or the logic analyzer.

One of the biggest problems encountered during project development were timing issues that arose while working with the ZBT RAM at 80MHz. The ZBT RAM interface modules were very sensitive to the routing and placing of the logic on the FPGA. In some occasions, adding an unrelated module changed the layout sufficiently to break the ZBT RAM driver module. A change is routing adds additional delays to the signals that drive the ZBT RAM which may result in invalid hold and setup times as well as clock skew. In order to combat such issues, timing constraints were researched and the follow lines were added to labkit.ucf to help the Xilinx tools create layouts that had sufficiently short delays to output pins.

```
NET "clock_27mhz" TNM_NET = "clock_27mhz";
TIMESPEC "TS_clock_27mhz" = PERIOD "clock_27mhz" 37 ns HIGH 50 %
INPUT_JITTER 0.1 ns;
      NET "ram0_we_b" OFFSET = OUT 15 ns AFTER "clock_27mhz" HIGH ;
```

Timing issues were further remedied by pipelining long combinational logic paths, especially those in the video priority encoder. Although the multiplications in the coordinate transformer and the coefficient generator were not pipelined, there were no timing issues as the modules ran off of a 27MHz clock.

Duck Hunt System (Tiffany)

Bush controller & Cloud Controller

The first step in testing the system was displaying stationary bushes. When the bush was displayed on the screen properly, the next step was to display clouds that scrolled across the sky at constant speeds. This proved to be more difficult than just displaying stationary images. When first displaying images for the cloud that moved along the width of the sky, black and white pixels from the ROM were scrabbled throughout the width of the screen. Because the bushes were addressed in the same manner, the issue had to be the additional complications introduced when moving the x

and y coordinates of the cloud so that it scrolled across the sky. After realizing that the x and y coordinates were moving at each clock cycle, and that the addressing of the ROM depending on having these coordinates constant until the entire image was displayed, the cloud controller was modified so that the coordinates of the cloud controller only changed when a new frame was generated. The frame module was thus introduced to the duck hunt video system to coordinate when the coordinates of the duck and clouds should change.

Duck Subsystem

When the duck controller was created, the various states were first represented using solid colors so that the states could be confirmed to be transitioning correctly. The various inputs to the duck controller were first simulated using debounced buttons. For the input signals that were required to be only one clock cycle high, the debounced button signals were then passed through a module signal_to_pulse that created a one-clock-cycle high pulse at the positive edge of the signal.

When the duck controller was interfaced with the appear timer, it was quickly realized that the duck controller also required a disappear signal to let the duck controller know when to transition to the state where the duck flies away off the screen, or else the duck would be on the screen until it died or the game was over. With these modifications, the duck controller was smoothly integrated with the rest of the duck subsystem.

Timing Issues

Initially, the ROMs were addressed by incrementing the address each time the hsync and vsync signals were within the range of the specified image. However, occasional problems would occur where the address did not seem to reset properly. Further investigation revealed that these rare problems occurred because the reset signal would sometimes occur when the hsync and vsync signals were exactly within the range of the image. To correct this, the multiplication and bit-shift method of addressing and magnifying the image as described previously were used. At first, the concern was that because the multipliers were located only on a specific part of the FPGA chip that the optimization of the routing on the hardware would be affected, and the lag would create timing issues for the entire system. However, upon integration with the laser dot finder system, no such issues arose.

Furthermore, when this method of addressing images was used, the images were shifted horizontally by one pixel to the right. This problem occurred because the ROM requires one clock cycle after the address is declared to output the corresponding pixel data. This issue was resolved by delaying the hsync, vsync, and blank signals by one clock cycle.

Duck Hunt System (Spencer)

Although only the gameover and laser gun sounds made it to the final Lasershot game, other sounds were successfully included in the beta versions of the game. When a bird flaps its wings, the flap sound component outputs the wing flapping sound from the original Duck Hunt game. With some beta testing, however, the constant wing flapping noise becomes increasingly aurally unaesthetic. For the sake of fun and enjoyment of the game, the flapping sound is not included in the final version.

appear duck_die

The duck falling sound is also excluded from the final version of Lasershot, but for a more logistical reason. In beta versions of the game, a whistle-like duck falling sound is output whenever a dead duck is falling from the sky. The duck falling sound file plays for approximately two seconds, but the duck does not always fall for two seconds. When a duck is killed toward the top of the screen, the two seconds fits the scenario. However, when a duck is killed closer to the bottom of the screen, the duck falls for less than even one second. Although it is possible to add a large number of time-specific duck falling sound files, this practice is wasteful with respect to memory management. For the sake of functionality and minimization of memory usage, the duck falling sound is not included in the final version of Lasershot.

cstringdisp.v

The char_string_display stores the characters as 8x12 pixel images, but doubles the size to 16x24 on output to the screen. Originally, I thought that I would be needing the text display for all of the text, including the title screen, win screen, and the game over screen. If all of the text in the entire game had been a constant size, there would be no differentiation between normal informational text and important title text. In order to accomplish differentiation, I modified the char_string_display module to quadruple the size of each character, when displayed on the screen. Although this required a couple of nights of manipulation, we decided to go with color splash screens for the winning and losing of the game. Moreover, the increased size of the text did not fit comfortably into the scorekeeper bar at the bottom of the screen. In the end, the modified char_string_disp module was not included in the final version of Laserkill, but the code is included in the appendix.

scorekeeper.v

Originally, the scorekeeper module also included the video output of the scorekeeper section of the screen, but I decided to separate the video output from the logic of the game, itself. Only the logic that dictates the rules of the game remains.

Conclusion

The objective of this final project was to design and implement a complex digital system combining video, audio, and complex user inputs. The analysis presented in the previous sections shows a fully audio and video game system along with a novel laser tracking system.

The dot finder system successfully tracked the laser dot as expected, though there were some difficulties with the reflectivity of the laser on the LCD screen. The diffusive reflector helped immensely with the contrast and the system could not work without it. The dot finder system could be improved in the future though the addition of more sophisticated noise filtering and image processing algorithms. In particular, salt and pepper noise was observed in the raw captured image that could have been removed through a median filter.

The duck hunt game system implemented various features of the original duck hunt arcade game with additional features such as animated bushes, scrolling clouds, and multiple duck shots to kill the duck. If given more time though, additional points of functionality could be implemented. For example, adding two-person duck hunt would increase the complexity of the system but also provide a more competitive gaming experience.

The laser shot system implemented all of the features that were initially planned. The sound system, with its laser and game over sound, successfully played the correct times. The dot tracking system also accurately tracked the laser dot on the screen. The duck hunt video system ensured a smooth gaming system. All in all, the laser shot project was a gratifying learning experience.

Appendix A – appear_counter.v

```
// appear_counter
// tiff chen

// The appear_counter is responsible for counting to r_interval seconds
for
// the duck to appear and 8 seconds for the duck to fly away off the
screen
// using the one_hz_enable signal to properly time the appearance of
the
// duck. When done counting to r_interval or 8 seconds, the appear
signal
// or disappear signal will be high to signify to the duck controller
that
// the duck should appear or disappear.

module appear_counter(clk, reset, r_interval, one_hz_enable,
      appear_enable, disappear_enable,
      appear_count, disappear_count, appear, disappear);

   input clk; // 40 MHz
      input reset; // 1 for reset
      input[1:0] r_interval; // interval in seconds from rn_generator
      input one_hz_enable; // from reset second_timer
                                                  // is 1 clock cycle
high every 1 second
      input appear_enable; // 1 to start timing for ducks to appear
      input disappear_enable; // 1 when duck begins to fly around
      output[1:0] appear_count; // for debugging
      output[2:0] disappear_count; // for debugging
   output appear; // to duck_controller
                                    // 1 for duck to appear
      output disappear; // to duck_controller
                                       // 1 for duck to disappear

   reg[1:0] appear_count;     // counter for duck to appear
      reg[2:0] disappear_count; // counter for duck to disappear

      parameter DISAPPEAR_TIME = 3'd7; // duck should fly around on
screen
                                                                      //
for 8 seconds before flying off

   always @ (posedge clk)
   begin

            // build a decrement counters appear_count &
disappear_count

            if (reset) begin
                  appear_count <= r_interval;
                  disappear_count <= DISAPPEAR_TIME;
                  end // end reset
```

```
            else begin

            if (appear_enable)
                  appear_count <= r_interval;
                        // reset appear_count to r_interval
            else if ((appear_count > 0) & (one_hz_enable))
                  appear_count <= appear_count - 1;
                        // decrement appear_count if appear_count > 0
            else if (~(appear_count > 0) & (one_hz_enable))
                  appear_count <= r_interval;
                        // reset appear_count to r_interval if
appear_count <= 0

            if (disappear_enable)
                  disappear_count <= DISAPPEAR_TIME;
            else if ((disappear_count > 0) & (one_hz_enable))
                  disappear_count <= disappear_count -1;
            else if (~(disappear_count > 0 ) & (one_hz_enable))
                  disappear_count <= DISAPPEAR_TIME;

            end // end else

   end // end begin

      assign appear = (appear_count == 3'd0);
      assign disappear = (disappear_count == 3'd0);

endmodule
```

Appendix B – averager.v

```
`timescale 1ns / 1ps

// Module: averager (averager)
// Author: Paul Yang
// Description: the averager takes in thresholded pixel data and
computes the
// average x,y coordinate of all non-zero pixels

module averager(clk, reset, x_in, y_in, value, avg_x, avg_y, dv_in);
    // Inputs:
    // clk           -  clock input @ 27MHz
    // reset         -  reset for initialization
    // x_in          -  x coordinate of the sample
    // y_in          -  y coordinate of the sample
    // value         -  value of the sample
    // dv_in         -  1 if the sample is valid, 0 o/w

    // Outputs:
    // avg_x         -  The average x coordinate of all the samples
that
    //                  have a value of 255
    // avg_y         -  The average y coordinate of all the samples
that
    //                  have a value of 255

    //                  Note: avg_x and avg_y are updated every frame
(1/30s)
    //                  A new frame is assumed when there is a sample
at a
    //                  pixel location (11'h20, 11'h20). (0,0) was not
used
    //                  as it was possible that dv might be low at that
point
    input clk;
    input reset;
    input [10:0] x_in;
    input [9:0] y_in;
    input dv_in;
    input [7:0] value;

    output [10:0] avg_x;
    output [9:0] avg_y;


    // The size of the register should be
    // 11 bits * log2(720) * log2(525) = 31 bits MAX for x's ,
    // 30 bits MAX for y's
    // However, to reduce the size of the divider, it’s assumed that
there is
    // at most 65536 pixels that needs to be counted
    // which reduces the count to 25 bits. Make the counter 26 bits in
case

    // Registers to accumulate the sum of screen coordinates
```

```
    reg [25:0] acc_x;
    reg [25:0] acc_y;

    // Register to keep track of the number of pixels
    reg [15:0] pixels;

    // Registers that hold the final accumulated sums at the end of
each frame
    reg [25:0] final_x;
    reg [25:0] final_y;
    reg [15:0] final_pixels;

    // Wire up a divider to find the average value
    wire [25:0] x_quot;
    wire [25:0] y_quot;
    divider dv1(
            .dividend(final_x),
            .divisor(final_pixels),
            .quot(x_quot),
            .clk(clk),
            .aclr(0),
            .sclr(reset),
            .ce(1'b1));
    divider dv2(
            .dividend(final_y),
            .divisor(final_pixels),
         .quot(y_quot),
            .clk(clk),
            .aclr(0),
            .sclr(reset),
            .ce(1'b1));

   assign avg_x = x_quot[10:0];
   assign avg_y = y_quot[9:0];

    always @ (posedge clk)
    begin
      // On reset
      if(reset)
      begin
         acc_x <= 26'h0;
         acc_y <= 26'h0;
         final_y <= 26'h0;
         final_x <= 26'h0;
         final_pixels <= 25'h1; // Avoid a div/0 in case there are no
pix detected
         pixels <= 0;
      end
      else
      begin
         if(dv_in)
         begin
            // Recalculate the average every new frame (0x20, 0x20)
chosen
            // arbitrarily - any coordinate will work
            if(x_in == 11'h20 && y_in == 10'h20)
            begin
```

```
                // Only give data to the divider when there are enough
pixels
                // that are are non-zero in the frame. Otherwise retain
old data
                if( pixels >= 16'd2)
                begin
                   final_x <= acc_x;
                   final_y <= acc_y;
                   final_pixels <= pixels;
                end
                // Clear the registers for each frame
                acc_x <= 26'h0;
                acc_y <= 26'h0;
                pixels <= 16'h0;
             end
             // Otherwise count the non-zero pixels
             else if( value != 8'h0)
             begin
                acc_x <= acc_x + x_in;
                acc_y <= acc_y + y_in;
                pixels <= pixels + 1;
             end
          end
       end
    end

endmodule
```

Appendix C – blob.v

```
// Module: cblob (centered blob)
// Author: Paul Yang (modified from lab5 code)
// Description: Using the video signals from the SVGA module in lab 5,
// this creates a square blob centered at the given x,y coordinate

module cblob(x,y,hcount,vcount,pixel);

   // Parameters:
   // SIZE           -  2*SIZE is the length of the square (default:4x4
square)
   // COLOR          -  the color of the blob, (default white)
   parameter SIZE = 2;
   parameter COLOR = 24'hffffff;

   // Inputs:
   // x              -  The x coordinate of the center of the blob
   // y              -  The y coordinate of the center of the blob
   // hcount         -  Horizontal count (see svga)
   // vcount         -  Vertical count

   // Outputs:
   // pixel          -  What the pixel color should be at (hcount,
vcount)
   //                   Note that 0 is assumed to be transparent
   input [10:0] x,hcount;
   input [9:0] y,vcount;
   output [23:0] pixel;

   reg [23:0] pixel;



   // Use a combinational block as the logic is simple
   always @ (x or y or hcount or vcount) begin
   // 4 Possible cases for where the blob can be - this must be done to
avoid
   // getting negative bounds to compare hcount and vcount

      // Region 1 - this is where the square gets clipped on the top
and the
      // left
      if(x < SIZE && y < SIZE)
      begin
         if(hcount <= x + SIZE && vcount <= y + SIZE )
            pixel = COLOR;
         else
            pixel = 0;
      end
      // Region 2 - this is where the square gets clipped just on the
left
      else if(x < SIZE)
      begin
         if(hcount <= x + SIZE && vcount <= y + SIZE && vcount >= y -
SIZE)
```

```
            pixel = COLOR;
         else
            pixel = 0;
      end
      // Region 3 - this is where the square gets clipped just on the
top
      else if(y < SIZE)
      begin
         if(vcount <= y + SIZE && hcount <= x + SIZE && hcount >= x -
SIZE)
            pixel = COLOR;
         else
            pixel = 0;
      end
      // Region 4 - this is where the square get clipped on the
bottom/right
      // or not at all. Clipping does not matter as the calculated
bounds
      // are never negative in this region
      else
      begin
         if( hcount <= x + SIZE && hcount >= x - SIZE &&
             vcount <= y + SIZE && vcount >= y - SIZE)
            pixel = COLOR;
         else
            pixel = 0;
      end
   end
endmodule
```

Appendix D – box_gen.v

```
`timescale 1ns / 1ps
// Module: box_gen (Box Generator)
// Author: Paul Yang
// Description: Creates the appropriate RAM addresses and the
corresponding data
// to feed to the vram module so that a black box with a 1 pixel wide
blue-green border
// is written to the ZBT RAM. The box generator is used for detecting
image alignment
// issues due to the fact that the ZBT RAM outputs its data 2 clock
cycles late
// This module assumes 800x600 resolution and is used for testing
purposes ONLY


module box_gen(clk, reset, hcount, vcount, ram_addr, data, we);
    // Inputs:
    // clk              -  the clock at 40MHz
    // reset            -  reset signal
    // hcount           -  horizontal count (see svga)
    // vcount           -  vertical count (see svga)

    // Outputs
    // ram_addr         -  The RAM address to write to
    // data             -  The data to write at the address
    // we               -  Write Enable (1 to write,)
    input clk;
    input reset;
    input [10:0] hcount;
    input [9:0] vcount;
    output [18:0] ram_addr;
    output [35:0] data;
    output we;

    reg [18:0] ram_addr;
    reg [35:0] data;
    reg we;

    // This is the color of the border to draw
    parameter COLOR = 36'h00ffff;

    always @ (posedge clk)
    begin
      // On reset, clear the registers
      if(reset)
      begin
         ram_addr <= 19'h0;
         data <= 36'h0;
         we <= 0;
      end
      else
      begin
         // Draw a white dot at (0,0) and start RAM addressing from
that point
```

```
         if(hcount == 11'h0 && vcount == 10'h0)
         begin
            ram_addr <= 19'h0;
            data <= COLOR;
            we <= 1;
         end
         else if(hcount < 11'd800 && vcount < 10'd600)
         begin
            // Draw a white line at the borders. Draw black everywhere
else
            ram_addr <= ram_addr + 1;
            we <= 1;
            data <=
               (hcount == 11'd0 || hcount == 11'd799 || vcount == 10'd0
|| vcount == 10'd599) ?
                  COLOR : 36'h0;
         end
         // When the hcount or vcount goes off screen, don't write
anything to RAM
         else
         begin
            we <= 0;
         end
      end
    end

endmodule
```

Appendix E – bush.v

```
/*********************************************************************
*********
*     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 1995-2004 Xilinx, Inc.
*
```

```
*     All rights reserved.
*
**********************************************************************
********/
// The synopsys directives "translate_off/translate_on" specified below
are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity
synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file bush.v when simulating
// the core, bush. 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 bush(
      addr,
      clk,
      dout);


input [12 : 0] addr;
input clk;
output [3 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            13,   // c_addr_width
            "0",  // c_default_data
            6650, // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "bush.mif", // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            4,    // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
```

```
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of bush is "true"

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

endmodule
```

Appendix F – bush_animator2.v

```
module bush_animator2 (
      // inputs
      vclock, reset,
      bush_shudder_select,
      hcount, vcount, hsync, vsync, blank,
      new_frame, // from frame
      halt, // = (gameover)||(win), from scorekeeper
      // outputs
      bush_pixel, bush_x, bush_y,
      dhsync, dvsync, dblank

      );
      input vclock;     // 40MHz clock
      input reset;      // 1 to initialize module
      input bush_shudder_select; //
      input [10:0] hcount;    // horizontal index of current pixel
(0...799)
      input [9:0] vcount; // vertical index of current pixel (0...599)
      input hsync;            // XVGA horizontal sync signal (active
low)
      input vsync;            // XVGA vertical sync signal (active low)
      input blank;            // XVGA blanking (1 means output black
pixel)
      input new_frame;
      input halt;

      output [3:0] bush_pixel; // bush pixels
      output [10:0] bush_x; // bush x-coordinate
      output [9:0] bush_y; // bush y-coordinate
      output dhsync;    // game's horizontal sync
      output dvsync;    // game's vertical sync
      output dblank;    // game's blanking




      // Animation Values
      parameter SWAY = 11'd40;
      parameter SWAY_SPEED = 11'd1;
      parameter SHUDDER_SPEED = 11'd18;
      parameter SHUDDER_Y = 11'd15;

      parameter SKY_WIDTH = 11'd799;
   parameter SKY_HEIGHT = 10'd499;

      parameter BUSH_HEIGHT = 10'd50;
      parameter BUSH_WIDTH = 11'd133;
      parameter BUSH_X_START = 11'd350;     // use def.param to change
      parameter BUSH_Y_START = SKY_HEIGHT - BUSH_HEIGHT -5;
      parameter BUSH_END = 13'd6649;

      wire [3:0] bush_pixel;
      reg [12:0] bush_addr = 13'd0; // depth of ROM is 6650
```

```
      reg [10:0] bush_x;
      reg [9:0] bush_y;
      reg right_left; // 1 if puck is going right, 0 if going left
      reg gameover; // 1 when the user loses and receives a game over,
0 in regular game mode

      bush bush1(.addr(bush_addr),.clk(vclock),.dout(bush_pixel));

always @ (posedge vclock) // Careful!
      if ((bush_x <= hcount) &&
                  (hcount < bush_x + BUSH_WIDTH) &&
                  (bush_y <= vcount) &&
                  (vcount < bush_y + BUSH_HEIGHT)
                  )
                  begin
         bush_addr <= (vcount - bush_y) * BUSH_WIDTH + (hcount -
bush_x);
                        //bush_addr <= (bush_addr == BUSH_END) ? 13'd0
: bush_addr + 1;
                  end



      if (reset)
            begin
            right_left <= 1'b1; // from reset, puck starts moving right
            bush_x <= BUSH_X_START;
         bush_y <= BUSH_Y_START;
            bush_addr <= 13'd0;

            end

      else if (halt)
            begin // stop all movement
            bush_x <= bush_x;
            bush_y <= bush_y;
            end // stop all movement

      else
            begin
            if (new_frame)
            begin
            if (right_left)
                  begin // right behaviour
                  if (bush_shudder_select)
                        begin
                              if (bush_x + SHUDDER_SPEED >=
BUSH_X_START + SWAY) // if puck is going to hit the wall
                                    begin // change in direction
                                    //bush_x <= (field_width -
puck_width); // puck hits the wall
                                    right_left <= 0; // puck starts
moving left
                                    end // end change in direction
                              else
                                    begin // puck move
```

```
                                    bush_x <= bush_x + SHUDDER_SPEED;
// puck moves pspeed pix/frame horizontally
                                    end //puck move
                        end // if (bush_shudder_select)
                  else
                        begin
                              if (bush_x + SWAY_SPEED >= BUSH_X_START +
SWAY) // if puck is going to hit the wall
                                    begin // change in direction
                                    //bush_x <= (field_width -
puck_width); // puck hits the wall
                                    right_left <= 0; // puck starts
moving left
                                    end // end change in direction
                              else
                                    begin // puck move
                                    bush_x <= bush_x + SWAY_SPEED; //
puck moves pspeed pix/frame horizontally
                                    end //puck move
                        end // else

                  end // right behaviour

            else if (~right_left) // puck is going left
                  begin // left behaviour
                  if (bush_shudder_select)
                        begin
                        if (bush_x - SHUDDER_SPEED <= BUSH_X_START) //
if puck is going to hit the wall
                              begin // change in direction
                              //bush_x <= (field_width - puck_width);
// puck hits the wall
                              right_left <= 1; // puck starts moving
left
                              end // end change in direction
                        else
                              begin
                              bush_x <= bush_x - SHUDDER_SPEED; //
otherwise, puck goes left, pspeed pixels/frame
                              end
                        end

                  else
                        begin
                        if (bush_x - SWAY_SPEED <= BUSH_X_START) // if
puck is going to hit the wall
                              begin // change in direction
                              //bush_x <= (field_width - puck_width);
// puck hits the wall
                              right_left <= 1; // puck starts moving
left
                              end // end change in direction
                        else
                              begin
                              bush_x <= bush_x - SWAY_SPEED; //
otherwise, puck goes left, pspeed pixels/frame
                              end
```

```
                        end // else


                  end // left behaviour


            // change in y for shuddering
            if (bush_shudder_select)
                  begin // puck going down
                        bush_y <= BUSH_Y_START - SHUDDER_Y;
                  end // puck going down

            else if (~bush_shudder_select)
                  begin // puck going up
                        bush_y <= BUSH_Y_START;
                  end // puck going up
            end // if (new_frame)
            end // end of else
      end // end of always block


      assign dhsync = hsync;
   assign dvsync = vsync;
   assign dblank = blank;

endmodule
```

Appendix G – bush_controller_tiff.v

```
// bush_controller_tiff
// tiff chen

// bush controller takes outputs the bush's pixel data from the bush
rom and
// gives the video controller the appropriate x & y coordinates and
pixel data

module bush_controller_tiff(
      // inputs
      vclock, reset, bush_x, bush_y,
      hcount, vcount, hsync, vsync, blank,
      bush_height, bush_width,
      // outputs
      bush_pixel, dhsync, dvsync, dblank
      );

      input vclock; // 40 MHz
      input reset;
      input [10:0] bush_x; // bush x coordinate
      input [9:0] bush_y; // bush y coordinate
      input [10:0] hcount;    // horizontal index of current pixel
(0...799)
   input [9:0] vcount; // vertical index of current pixel (0...599)
   input hsync;         // XVGA horizontal sync signal (active low)
   input vsync;         // XVGA vertical sync signal (active low)
   input blank;         // XVGA blanking (1 means output black pixel)
      input [10:0] bush_width;
      input [9:0] bush_height;

      output [3:0] bush_pixel; // bush's pixels
      output dhsync;    // duck game's horizontal sync
   output dvsync; // duck game's vertical sync
   output dblank; // duck game's blanking

      // pixel wires from bush ROM
      wire [3:0] bush_pixel;
      reg [12:0] bush_add = 13'd0; // depth of ROM is 6650

      // instantiating bush ROM
      bush bush1(.addr(bush_add),.clk(vclock),.dout(bush_pixel));

      always @ (posedge vclock) begin

      if (reset) begin
            // reset x & y coordintes of bush
            bush_add <= 13'd0;
            end // end reset

      // if hcount & vcount are within the range of the bush image
      if ((bush_x <= hcount) && (hcount < bush_x + bush_width) &&
             (bush_y <= vcount) && (vcount < bush_y + bush_height))
begin
```

```
              bush_add <= (vcount - bush_y) * bush_width + (hcount -
bush_x);
              end // end bush_x & bush_y

      end // always

      assign dhsync = hsync;
   assign dvsync = vsync;
   assign dblank = blank;

endmodule
```

Appendix H – bush_selector.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:59:58 12/09/06
// Design Name:
// Module Name:    bush_selector
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module bush_selector (//inputs
                                        clk, reset, one_hz_enable,
                                        bush_number, shudder_pulse,
                                        shudder_time_expire,
                                        //outputs
                                        activate_bush1,
activate_bush2, activate_bush3, activate_bush4,
                                        start_shudder_timer
                                        );


input clk;
input reset;
input one_hz_enable;


input [1:0] bush_number;
input shudder_pulse;
input shudder_time_expire;

output activate_bush1, activate_bush2, activate_bush3, activate_bush4;
output start_shudder_timer;
wire start_shudder_timer;
assign start_shudder_timer =
(activate_bush1|activate_bush2|activate_bush3|activate_bush4);


//wire activate_bush1;
//wire activate_bush2;
//wire activate_bush3;
//wire activate_bush4;
```

```
//
reg activate_bush1 = 1'b0;
reg activate_bush2 = 1'b0;
reg activate_bush3 = 1'b0;
reg activate_bush4 = 1'b0;

always @ (posedge clk)
   begin
            if (shudder_pulse)
                  begin
                        case (bush_number)
                        2'b00:      activate_bush1 <= 1'b1;
                        2'b01:      activate_bush2 <= 1'b1;
                        2'b10:      activate_bush3 <= 1'b1;
                        default: activate_bush4 <= 1'b1; // if not 00,
nor 10, nor 01, must be 11
                        endcase
                  end
            else if ((activate_bush1 == 1'b1)&&(shudder_time_expire))
                  begin
                        activate_bush1 <= 1'b0;
                  end
            else if ((activate_bush2 == 1'b1)&&(shudder_time_expire))
                  begin
                        activate_bush2 <= 1'b0;
                  end
            else if ((activate_bush3 == 1'b1)&&(shudder_time_expire))
                  begin
                        activate_bush3 <= 1'b0;
                  end
            else if ((activate_bush4 == 1'b1)&&(shudder_time_expire))
                  begin
                        activate_bush4 <= 1'b0;
                  end
//
//assign activate_bush1 = ((shudder_pulse)&&(bush_number == 2'b00)) ?
1'b1 :
//                                                ((activate_bush1 ==
1'b1)&&(shudder_time_expire)) ? 1'b0 :
//                                                1'b0;
//assign activate_bush2 = ((shudder_pulse)&&(bush_number == 2'b01)) ?
1'b1 :
//                                                ((activate_bush2 ==
1'b1)&&(shudder_time_expire)) ? 1'b0 :
//                                                1'b0;
//assign activate_bush3 = ((shudder_pulse)&&(bush_number == 2'b10)) ?
1'b1 :
//                                                ((activate_bush3 ==
1'b1)&&(shudder_time_expire)) ? 1'b0 :
//                                                1'b0;
//assign activate_bush4 = ((shudder_pulse)&&(bush_number == 2'b11)) ?
1'b1 :
//                                                ((activate_bush4 ==
1'b1)&&(shudder_time_expire)) ? 1'b0 :
//                                                1'b0;
//
```

```
end
endmodule
```

Appendix I – cloud.v

```
/*****************************************************************
*********
*     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 1995-2004 Xilinx, Inc.
*
*     All rights reserved.
*
```

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

// You must compile the wrapper file cloud.v when simulating
// the core, cloud. 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 cloud(
      addr,
      clk,
      dout);


input [12 : 0] addr;
input clk;
output [3 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            13,   // c_addr_width
            "0",  // c_default_data
            4320, // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "cloud.mif",      // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            4,    // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
```

```
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of cloud is "true"

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

endmodule
```

Appendix J – cloud_controller.v

```
// cloud_controller
// tiff chen

// cloud controller takes as input the cloud speed from the video
controller
// and outputs the cloud's pixel data from the ROM and gives the video
// controller the appropriate x & y coordinates and pixel data

module cloud_controller(
      // inputs
      vclock, reset,
      cloud_enable, east, cloud_speed,
      cloud_initx, cloud_inity,
      cloud_width, cloud_height,
      sky_width,
      hcount, vcount, hsync, vsync, blank,
      new_frame, halt,
      // outputs
      cloud_pixel, cloud_x, cloud_y,
      dhsync, dvsync, dblank,
      cloud_debug
      );

      input vclock; // 40 MHz
      input reset; //
      input cloud_enable; // 1 for cloud to show up
      input east; // 1 for east moving cloud, 0 for west moving cloud
      input [10:0] cloud_initx; // inital x coordinate of cloud
      input [9:0] cloud_inity; // initial y coordinate of cloud
      input[3:0] cloud_speed; // cloud speed
      input [10:0] hcount;    // horizontal index of current pixel
(0...799)
   input [9:0] vcount; // vertical index of current pixel (0...599)
   input hsync;         // XVGA horizontal sync signal (active low)
   input vsync;         // XVGA vertical sync signal (active low)
   input blank;         // XVGA blanking (1 means output black pixel)
      input new_frame;
      input halt; // 1 to stop everything from moving
      input [9:0] cloud_height;
      input [10:0] cloud_width;
      input [10:0] sky_width;

      output [3:0] cloud_pixel; // cloud's pixels
      output [10:0] cloud_x; // cloud's x-coordinate
      output [9:0] cloud_y; // cloud's y-coordinate
      output dhsync;    // duck game's horizontal sync
   output dvsync; // duck game's vertical sync
   output dblank; // duck game's blanking

      output cloud_debug;

      // pixel wires from cloud ROM
      wire [3:0] cloudtmp_pixel;
      reg [12:0] cloud_add = 13'd0; // depth of ROM is 4320
```

```
      wire [3:0] cloud_pixel; // width of ROM is 4
      reg [10:0] cloud_x;
      reg [9:0] cloud_y;
      reg cloud_debug = 1'b0;

      assign cloud_pixel = (cloud_enable) ? cloudtmp_pixel : 4'd13;

      // instantiating cloud ROM
      cloud
cloud1(.addr(cloud_add),.clk(vclock),.dout(cloudtmp_pixel));

      always @ (posedge vclock) begin

      if (reset) begin
            // reset x & y coordintes of cloud
            cloud_x <= cloud_initx;
            cloud_y <= cloud_inity;
            cloud_add <= 13'd0;
            end // end reset

            //cloud_x_right <= cloud_x + CLOUD_WIDTH;
            //cloud_debug <= ((~east) &&
            //                                  (cloud_x_right <=
cloud_speed) && (new_frame)) ?
            //                                  ~cloud_debug :
cloud_debug;

            // if hcount & vcount are within the range of the cloud
image
            if ((cloud_x <= hcount) && (hcount < cloud_x + cloud_width)
&&
                  (cloud_y <= vcount) && (vcount < cloud_y +
cloud_height)) begin
                  cloud_add <= (vcount - cloud_y) * cloud_width +
(hcount - cloud_x);
//                if ((cloud_add == 13'd4319) ||
//                   ((cloud_x == 11'd0) && (new_frame))) begin
//                            // reach max address for end of rom or
for scrolling effect
//                            cloud_add <= 13'd0;
//                      end // reset rom address
//                else
//                      cloud_add <= cloud_add + 1;
                  end // cloud_x & cloud_y

            // display new frame with updated cloud x & y coordinates
            if (new_frame) begin

                  // controlling the x coordinate of cloud

                  if (halt)
                        cloud_x <= cloud_x;      // cloud is stationary

                  else if (east) begin // cloud is traveling east
                        if (cloud_x + cloud_speed >= sky_width)
                              cloud_x <= 11'd0;
                        else
```

```
                              // cloud x coordinate increments to move
cloud east
                              cloud_x <= cloud_x + cloud_speed;
                              end // end east begin
                  else if (~east) begin // cloud is traveling west
                        if (cloud_x <= cloud_speed) // cloud_x_right -
cloud_speed <= 0
                              cloud_x <= sky_width;
                        else
                              // cloud x coordinate decrements to move
cloud west
                              cloud_x <= cloud_x - cloud_speed;
                        end // end west begin
                   end // end new_frame begin
            end // always

      assign dhsync = hsync;
   assign dvsync = vsync;
   assign dblank = blank;

endmodule
```

Appendix K – coeff_gen.v

```
`timescale 1ns / 1ps

// Module: coeff_gen (Coefficent Generator)
// Author: Paul Yang
// Description: Computes the a,b,c,d,e,f,g,h coefficients required for
the
// perspective transform. Note that y2 and y4 are not inputs - this is
due to
// the constraint that the camera must be lined up in the horizontal
plane.
// These coefficients are used to convert a coordinate in the NTSC
frame
// to coordinate on the LCD screen.
// The latency of this module is 6 + (latency of divider = 32 + 19 + 3)
= 60
module coeff_gen(clk, reset, x1, y1, x2, x3, y3, x4, a, b, c, d, e, f,
                    g, h);
    // Inputs:
    // clk              -  The clock @ 40Mhz
    // reset            -  Reset signal
    // x1               -  x coordinate of the upper left corner of the
screen
    // y1               -  y coordinate of the upper left corner of the
screen
    // x2               -  x coordinate of the upper right corner
    // x3               -  x coordinate of the lower left corner
    // y3               -  y coordinate of the lower left corner
    // x4               -  x coordinate of the lower right corner

    // Outputs:
    // a,b,c,d,e,f,g,h  -  Coefficients for the perspective transform,
    //                     multiplied by 2^18 to get 'rid' of the
fractional
    //                     parts. Multiplying simply shifts the decimal
point.
    //                     The output is 36 bits for consistency as the
transform
    //                     uses 36 bit multiplication to reduce round
off errors
    input clk;
    input reset;
    input [10:0] x1;
    input [9:0] y1;
    input [10:0] x2;
    input [10:0] x3;
    input [9:0] y3;
    input [10:0] x4;

    output [35:0] a;
    output [35:0] b;
    output [35:0] c;
    output [35:0] d;
    output [35:0] e;
    output [35:0] f;
```

```
    output [35:0] g;
    output [35:0] h;



    wire signed [10:0] x1;
    wire signed [9:0] y1;
    wire signed [10:0] x2;
    wire signed [10:0] x3;
    wire signed [9:0] y3;
    wire signed [10:0] x4;
    reg signed [31:0] divr;

    // The coefficients a,b,c,d,e,f are assumed to have been multiplied
by COEFF
    // to cut out the fractional parts of the coefficients. The
parameter is
    // for REFERENCE only; it is not used in this module (but is used
in others)
    parameter COEFF = 36'd262144;

    // Equations for generating the coefficients
    // assumes a mapping from 720x525 NTSC frame to 800x600 SVGA frame
    // p1 = (x1, y1) = upper left hand corner of the LCD in the NTSC
frame
    // p2 = (x2, y2) = upper right hand corner of the LCD "
    // p3 = (x3, y3) = lower left hand corner of the LCD  "
    // p4 = (x4, y4) = lower left hand corner of the LCD  "

    // Perspective transform equation
    // x' = (a*x + b*y + c)/(g*x + h*y + 1)
    // y' = (d*x + e*y + f)/(g*x + h*y + 1)
    // where (x,y) is a point in the NTSC frame
    // and (x',y') is the coordinate of the point in the 800x600 SVGA
screen

    // Using the following assumptions
    // (x1, y1) maps to (0,0)
    // (x2, y2) maps to (799, 0)
    // (x3, y3) maps to (0, 599)
    // (x4, y4) maps to (799, 599)
    // y1 = y2
    // y3 = y4

    // Matlab gives you the following equations for the coefficients

    // a = 799*(y1-y3)/(-y3*x2+x4*y1+y3*x1-x3*y1)
    // b = -799*(x1-x3)/(-y3*x2+x4*y1+y3*x1-x3*y1)
    // c = 799*(y3*x1-x3*y1)/(-y3*x2+x4*y1+y3*x1-x3*y1)
    // d = 0
    // e = 599*(-x4+x3)/(-y3*x2+x4*y1+y3*x1-x3*y1)
    // f = -599*(-x4+x3)*y1/(-y3*x2+x4*y1+y3*x1-x3*y1)
    // g = 0
    // h = -(x1-x2+x4-x3)/(-y3*x2+x4*y1+y3*x1-x3*y1)

    // d and g are always 0
    assign d = 0;
```

```
    assign g = 0;

    // a_num represents the numerator in the equation for a
    // b_num represents the numerator in the equation for b
    // ...
    reg signed [31:0] a_num;
    reg signed [31:0] b_num;
    reg signed [31:0] c_num;
    reg signed [31:0] e_num;
    reg signed [31:0] f_num;
    reg signed [31:0] h_num;

    // a_sum represents a sum that needs to be calculated a_num
    // ...
    wire signed [9:0]  a_sum;
    wire signed [10:0]  b_sum;
    wire signed [20:0] c_sum;
    wire signed [10:0]  e_sum;
    wire signed [10:0]  f_sum;

    assign a_sum = y1 - y3;
    assign b_sum = x1 - x3;
    assign c_sum = y3*x1 - x3*y1;
    assign e_sum = x3 - x4;
    assign f_sum = x3 - x4;

    // State counter for feeding numbers into the divider
    reg [7:0] fcount;

    // Register for all the constants
    // For these cases, _w indicates the whole part
    // and _f indicates the fractional part
    reg signed [31:0] a_w, b_w, c_w, e_w, f_w, h_w;
    reg signed [18:0] a_f, b_f, c_f, e_f, f_f, h_f;


    // Wires used for output calculations that merge the whole and
fractional
    // parts of the divider output
    wire [31:0] a_t1;
    wire [31:0] b_t1;
    wire [31:0] c_t1;
    wire [31:0] e_t1;
    wire [31:0] f_t1;
    wire [31:0] h_t1;


    // Wire up the divider
    reg [31:0] dividend;
    wire [31:0] quot;
    wire [18:0] frac;

    div_32x32s_19f div0(
      .dividend(dividend),
      .divisor(divr),
      .quot(quot),
      .remd(frac),
```

```
      .clk(clk),
      .rfd(),
      .aclr(1'b0),
      .sclr(1'b0),
      .ce(1'b1));


    always@(posedge clk)
    begin
      // On reset, clear all the registers
      if(reset)
      begin
         fcount <= 0;
         a_num <= 32'h0;
         b_num <= 32'h0;
         c_num <= 32'h0;
         e_num <= 32'h0;
         f_num <= 32'h0;
         h_num <= 32'h0;
         //divisor <= 32'h1; // No DIV/0! =D
      end
      else
      begin
         // Increment the state, up to 5
         fcount <= (fcount == 8'h5) ? 8'h0 : fcount + 8'h1;

         // Compute the numerator and denominator
         a_num <= 800*a_sum;
         b_num <= -800*b_sum;
         c_num <= 800*c_sum;
         e_num <= 600*e_sum;
         f_num <= -600*f_sum*y1;
         h_num <= -(x1 - x2 + x4 - x3);


         // divr is the denominator
         divr <= -y3*x2 + x4*y1 + y3*x1 - x3*y1;

//         Debugging data - used for testing
//         a_num <= 32'sd33252800;
//         a_den <= -32'sd199725;
//
//         b_num <= 32'h1;
//         b_den <= 32'h1;
//
//         c_num <= 32'h2;
//         c_den <= 32'h1;
//
//         e_num <= 32'h3;
//         e_den <= 32'h1;
//
//         f_num <= 32'h4;
//         f_den <= 32'h1;
//
//         h_num <= 32'h5;
//         h_den <= 32'h1;
```

```
         // feed numerators / denominators into the divider and also
read the
         // output data. The output of the divider each state is
completely
         // determined by the latency of the dividier. For example, at
state 0
         // the divider outputs f because the data for f was loaded
into the
         // divider n cycles before (where n is the latency of the
divider)
         case(fcount)

            8'h0:
            begin
               dividend <= a_num;
               f_w <= quot;
               f_f <= frac;
            end
            8'h1:
            begin
               dividend <= b_num;
               h_w <= quot;
               h_f <= frac;
            end
            8'h2:
            begin
               dividend <= c_num;
               a_w <= quot;
               a_f <= frac;
            end
            8'h3:
            begin
               dividend <= e_num;
               b_w <= quot;
               b_f <= frac;
            end
            8'h4:
            begin
               dividend <= f_num;
               c_w <= quot;
               c_f <= frac;
            end
            8'h5:
            begin
               dividend <= h_num;
               e_w <= quot;
               e_f <= frac;
            end

         endcase
      end
    end

    // From the whole and fractional parts, calculate the coefficient *
2 ^ 18;
```

```
    // The calculation subtracts one from the quotient (if the
fractional part

    // is negative and uses the lower 18 bits of the quotient and lower
18 bits

    // of the fractional remainder to compute the scaled coefficient

    assign a_t1 = a_w-1;
    assign a = (a_f < 0) ?  { a_t1[17:0], a_f[17:0] } : {a_w[17:0],
a_f[17:0]};

    assign b_t1 = b_w-1;
    assign b = (b_f < 0) ?  { b_t1[17:0], b_f[17:0] } : {b_w[17:0],
b_f[17:0]};

    assign c_t1 = c_w-1;
    assign c = (c_f < 0) ?  { c_t1[17:0], c_f[17:0] } : {c_w[17:0],
c_f[17:0]};

    assign e_t1 = e_w-1;
    assign e = (e_f < 0) ?  { e_t1[17:0], e_f[17:0] } : {e_w[17:0],
e_f[17:0]};

    assign f_t1 = f_w-1;
    assign f = (f_f < 0) ?  { f_t1[17:0], f_f[17:0] } : {f_w[17:0],
f_f[17:0]};

    assign h_t1 = h_w-1;
    assign h = (h_f < 0) ?  { h_t1[17:0], h_f[17:0] } : {h_w[17:0],
h_f[17:0]};

endmodule
```

Appendix L – colormap.v

```
// colormap
// tiff chen

// colormap converts the game_pixel value, which index a color, into
the
// appropriate 24-bit rgb colors

module colormap(clk, hsync, vsync, blank, hcount, vcount, switch,
      dhsync, dvsync, dblank, game_pixel,
      b, hs, vs, rgb);

      input clk; // 40 MHz
      input hsync;
      input vsync;
      input blank;
      input [10:0] hcount; // pixel number on current line
      input [9:0] vcount; // line number
      input [1:0] switch; // switches from labkit
      input dhsync; // duck hunt hsync
      input dvsync; // duck hunt vsync
      input dblank; // duck hunt blank
      input [3:0] game_pixel; // game pixel

      output b, hs, vs; // blank, horizontal sync, vertical sync
      output [23:0] rgb; // 24-bit rgb value, 8 bits/color

      // switch[1:0] selects which video generator to use:
   //  00: duck hunt game
   //  01: 1 pixel outline of active video area (adjust screen
controls)
   //  10: color bars
      //  11: debug screen (for debugging purposes only)

      reg [23:0] rgb;
      reg b,hs,vs;

   always @(posedge clk) begin
      if (switch[1:0] == 2'b01) begin
                  // 1 pixel outline of visible area (white)
                  hs <= hsync;
                  vs <= vsync;
                  b <= blank;
                  rgb <= (hcount==0 | hcount==799 | vcount==0 |
vcount==599) ?
                              24'b111111111111111111111111 : 24'd0;
      end else if (switch[1:0] == 2'b10) begin
                  // color bars
                  hs <= hsync;
                  vs <= vsync;
                  b <= blank;
                  rgb <=
{{8{hcount[8]}},{8{hcount[7]}},{8{hcount[6]}}};
      end else if (switch[1:0] == 2'b11) begin
                  // debugging mode
```

```
                  hs <= hsync;
                  vs <= vsync;
                  b <= blank;
                  rgb <= 24'b111111111111111111111111;
                  // rgb <=
{{5{pixel[2]}},{5{pixel[1]}},{5{pixel[0]}}};
            end else begin
         // default: duck hunt game
                  hs <= dhsync;
                  vs <= dvsync;
                  b <= dblank;
                  case (game_pixel)
                        // note: color names are purely for my own
purposes
                        // not their real color names
                        4'd0: rgb <= 24'b000000000000000000000000; //
black
                        4'd1: rgb <= {8'd94,8'd87,8'd38}; // CHANGED
FOR GROUND COLOR
                  4'd2: rgb <= 24'b000000001000000000000000; // dark
green
                        4'd3: rgb <= 24'b100000001000000000000000; //
dark yellow
                        4'd4: rgb <= 24'b000000000000000010000000; //
dark blue
                        4'd5: rgb <= 24'b100000000000000010000000; //
dark pink
                        4'd6: rgb <= 24'b000000001000000010000000; //
dark green-blue
                        4'd7: rgb <= 24'b100000001000000010000000; //
light grey
                        4'd8: rgb <= 24'b101111111011111110111111; //
grey
                        4'd9: rgb <= 24'b111111110000000000000000; //
red
                        4'd10: rgb <= 24'b000000001111111100000000; //
green
                        4'd11: rgb <= 24'b111111111111111100000000; //
yellow
                        4'd12: rgb <= 24'b000000000000000011111111; //
blue
                        4'd13: rgb <= 24'b111111110000000011111111; //
pink
                        4'd14: rgb <= 24'b000000001111111111111111; //
blue
                        4'd15: rgb <= 24'b111111111111111111111111; //
white
                        default: rgb <= 24'b000000000000000000000000;
// black
                  endcase
            end // end else
      end // end always

endmodule
```

Appendix M – coord_transform.v

```
`timescale 1ns / 1ps
// Module: coord_transform (Coordinate Transformer)
// Author: Paul Yang
// Description: The coord_transform module aims to convert a pixel
coordinate
// in the NTSC frame to a pixel coordinate in the 800x600 LCD display
frame.
// It uses a perspective transform of the form
// x' = (a*x + b*y + c)/(g*x + h*y + 1)
// y' = (d*x + e*y + f)/(g*x + h*y + 1)
// The latency of this module is the same as the latency of the divider

// (~32+3 cycles)

module coord_transform(clk, reset, a, b, c, d, e, f, g, h,
                       x_in, y_in, x_out, y_out, top_x, top_y, bottom);
      // Inputs:
    // clk              -  The clock @ 40Mhz
    // reset            -  Reset signal
    // a,b,c,d,e,f,g,h  -  The coefficients for the transform equation
    // x_in             -  The input x coordinate( x in equation)
    // y_in             -  The input y coordinate( y in equation)

    // Outputs:
    // x_out            -  The output x coordinate( x' in equation)
    // y_out            -  The output y coordinate( y' in equation)
    input clk;
    input reset;
    input [35:0] a;
    input [35:0] b;
    input [35:0] c;
    input [35:0] d;
    input [35:0] e;
    input [35:0] f;
    input [35:0] g;
       input [35:0] h;
    input [35:0] x_in;
    input [35:0] y_in;
    output [31:0] x_out;
    output [31:0] y_out;
    output [35:0] top_x;
    output [35:0] top_y;
    output [35:0] bottom;

       wire signed [35:0] a;
    wire signed [35:0] b;
    wire signed [35:0] c;
    wire signed [35:0] d;
    wire signed [35:0] e;
    wire signed [35:0] f;
    wire signed [35:0] g;
       wire signed [35:0] h;
    wire signed [35:0] x_in;
    wire signed [35:0] y_in;
```

```
       // top_x and bottom_x refer to the parts of the fraction
       // (a*x + b*y + c)/(g*x + h*y + 1)

       // top_y and bottom_y refer to the parts of the fraction
       // (d*x + e*y + f)/(g*x + h*y + 1)

       // The coefficients a,b,c,d,e,f are assumed to have been
multiplied by 2^18
    // to cut out the fractional parts of the coefficients. Since the
divider is
    // limited to 32 bits, the numerator and the denominator are
divided by
    // shifting right by 4 bits before sending it to the divider

       parameter COEFF = 36'sd262144;

    // Top_x, Top_y refer to the numerator in the transform eq.
    // bottom refers to the denominator
       assign top_x = a*x_in + b*y_in + c;
       assign top_y = d*x_in + e*y_in + f;
    assign bottom = g*x_in + h*y_in + COEFF;

    // Wire up the divider for the X

//   div_16x16s div0(
//    .dividend(top_x[35:20]),
//    .divisor(bottom[35:20]),
//    .quot(x_out),
//    .remd(),
//    .clk(clk),
//    .rfd(),
//    .aclr(1'b0),
//    .sclr(1'b0),
//    .ce(1'b1));
   div_32x32s div0(
      .dividend(top_x[35:4]),
      .divisor(bottom[35:4]),
      .quot(x_out),
      .remd(),
      .clk(clk),
      .rfd(),
      .aclr(1'b0),
      .sclr(1'b0),
      .ce(1'b1));

   div_32x32s div1(
      .dividend(top_y[35:4]),
      .divisor(bottom[35:4]),
      .quot(y_out),
      .remd(),
      .clk(clk),
      .rfd(),
      .aclr(1'b0),
      .sclr(1'b0),
      .ce(1'b1));
```

```
endmodule
```

Appendix N – corner_select_fsm.v

```
`timescale 1ns / 1ps
// Module: corner_select_fsm (Corner Selector)
// Author: Paul Yang
// Description: The corner select fsm allows the user to choose the 4
corner

// points that are needed to find the coefficients for the perspective

// transform

module corner_select_fsm(clk, reset, cen, inc_x, dec_x, inc_y, dec_y,
increment_size,
                         done, hcount, vcount, blob_data, x1, y1, x2,
y2, x3, y3,
                         x4, y4);



    // Inputs:

    // clk                 -  the clock at 40Mhz

    // reset               -  the reset signal

    // cen                 -  clock enable

    // inc_x               -  increments the x value of the currently
selected

    //                        point when asserted

    // dec_x               -  decrements the x value of the currently
selected

    //                        point when asserted

    // inc_y               -  increments the y value of the currently
selected
    //                        point when asserted

    // dec_y               -  decrements the y value of the currently
selected
    //                        point when asserted

    // increment_size      -  the increment/decrement size is 10 if 0,
1 if 1

    // done                -  signals to the FSM to change the
coordinates of

    //                        the next corner point
```

```
    // hcount              -  the horizontal count (used to display
blobs
    //                        at the corners)
    // vcount              -  the vertical count (used to display blobs
    //                        at the corners)


    // Outputs:
    // x1,y1               -  the coordinates of the upper left hand
corner
    //                        of the LCD screen in the NTSC frame
    // x2,y2               -  the coordinates of the upper right hand
corner
    //                        of the LCD screen in the NTSC frame
    // x3,y3               -  the coordinates of the lower right hand
corner
    //                        of the LCD screen in the NTSC frame
    // x4,y4               -  the coordinates of the lower left hand
corner
    //                        of the LCD screen in the NTSC frame
    // blob_data           -  video data based on hcount and vcount
that
    //                        that displays blobs centered at the
selected
    //                        corners


    input clk;
    input reset;
    input cen;
    input inc_x;
    input dec_x;
    input inc_y;
    input dec_y;
    input increment_size;
    input done;
    input [10:0] hcount;
    input [9:0] vcount;

    output wire [23:0] blob_data;
    output reg [10:0] x1;
    output reg [9:0] y1;
    output reg [10:0] x2;
    output reg [9:0] y2;
    output reg [10:0] x3;
    output reg [9:0] y3;
    output reg [10:0] x4;
    output reg [9:0] y4;
```

```
    // States of the FSM - corresponds to which corner you're selecting
    parameter S_UPPER_LEFT = 3'b00;
    parameter S_UPPER_RIGHT = 3'b01;
    parameter S_LOWER_RIGHT = 3'b010;
    parameter S_LOWER_LEFT = 3'b011;
    parameter S_IDLE = 3'b100;

    // The increments for changing the positions
    parameter SMALL_INC = 1;
    parameter BIG_INC = 10;

    // The blob color that marks the corner
    parameter BLOB_COLOR = 24'h0000ff;

    reg [2:0] state;

    always @ (posedge clk)
    begin
      if(reset)
      begin
         // On reset, set all the screen coordinates to test values
         x1 <= 48; y1 <= 45;
         x2 <= 623; y2 <= 45;
         x3 <= 28; y3 <= 437;
         x4 <= 688; y4 <= 437;
         state <= S_IDLE;
      end
      else if(cen)
      begin
         if(!done)
         begin
         case(state)
            // Move the upper left corner point
            S_UPPER_LEFT:
            begin
               x1 <= increment_size ?
                     (inc_x ? x1 + SMALL_INC : (dec_x ? x1 - SMALL_INC
: x1) ) :
                     (inc_x ? x1 + BIG_INC : (dec_x ? x1 - BIG_INC :
x1) );
               y1 <= increment_size ?
                     (inc_y ? y1 + SMALL_INC : (dec_y ? y1 - SMALL_INC
: y1) ) :
                     (inc_y ? y1 + BIG_INC : (dec_y ? y1 - BIG_INC :
y1) );
            end
            // Move the upper right corner point
            S_UPPER_RIGHT:
            begin
               x2 <= increment_size ?
                     (inc_x ? x2 + SMALL_INC : (dec_x ? x2 - SMALL_INC
: x2) ) :
                     (inc_x ? x2 + BIG_INC : (dec_x ? x2 - BIG_INC :
x2) );
               y2 <= increment_size ?
```

```
                        (inc_y ? y2 + SMALL_INC : (dec_y ? y2 - SMALL_INC
: y2) ) :
                        (inc_y ? y2 + BIG_INC : (dec_y ? y2 - BIG_INC :
y2) );
               end
               // Move the lower left corner point
               S_LOWER_LEFT:
               begin
                  x3 <= increment_size ?
                        (inc_x ? x3 + SMALL_INC : (dec_x ? x3 - SMALL_INC
: x3) ) :
                        (inc_x ? x3 + BIG_INC : (dec_x ? x3 - BIG_INC :
x3) );
                  y3 <= increment_size ?
                        (inc_y ? y3 + SMALL_INC : (dec_y ? y3 - SMALL_INC
: y3) ) :
                        (inc_y ? y3 + BIG_INC : (dec_y ? y3 - BIG_INC :
y3) );
               end
               // Move the lower right corner point
               S_LOWER_RIGHT:
               begin
                  x4 <= increment_size ?
                        (inc_x ? x4 + SMALL_INC : (dec_x ? x4 - SMALL_INC
: x4) ) :
                        (inc_x ? x4 + BIG_INC : (dec_x ? x4 - BIG_INC :
x4) );
                  y4 <= increment_size ?
                        (inc_y ? y4 + SMALL_INC : (dec_y ? y4 - SMALL_INC
: y4) ) :
                        (inc_y ? y4 + BIG_INC : (dec_y ? y4 - BIG_INC :
y4) );
               end

            endcase
            end
            // Only move on to the next point when the user presses done
            // also assign y2 to y1 and y3 to y4 on certain states for
convenience
            else
            begin
               state <= (state == 3'd4) ? 0 : state + 1;
               if(state == S_UPPER_LEFT)
                  y2 <= y1;
               else if(state == S_LOWER_RIGHT)
                  y3 <= y4;
            end
         end
      end

      // Create blobs at each of the coordinates
      wire [23:0] p1, p2, p3, p4;

      cblob c1( .x(x1), .y(y1), .hcount(hcount), .vcount(vcount),
.pixel(p1));
      cblob c2( .x(x2), .y(y2), .hcount(hcount), .vcount(vcount),
.pixel(p2));
```

```
    cblob c3( .x(x3), .y(y3), .hcount(hcount), .vcount(vcount),
.pixel(p3));
    cblob c4( .x(x4), .y(y4), .hcount(hcount), .vcount(vcount),
.pixel(p4));

    defparam c1.COLOR = BLOB_COLOR;
    defparam c2.COLOR = BLOB_COLOR;
    defparam c3.COLOR = BLOB_COLOR;
    defparam c4.COLOR = BLOB_COLOR;

    // Also draw a green horizontal line for p1, p3
    assign blob_data = p1 | p2 | p3 | p4 |
                       ( (vcount == y1 || vcount == y4) ? 24'h007f00 :
0) ;
endmodule
```

Appendix O – cstringdisp.v

```
//
// File:   cstringdisp.v
// Date:   24-Oct-05
// Author: I. Chuang, C. Terman
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// INPUTS:
//
//   vclock       - video pixel clock
//   hcount       - horizontal (x) location of current pixel
//   vcount       - vertical (y) location of current pixel
//   cstring      - character string to display (8 bit ASCII for each
char)
//   cx,cy        - pixel location (upper left corner) to display
string at
//
// OUTPUT:
//
//   pixel        - video pixel value to display at current location
//
// PARAMETERS:
//
//   NCHAR        - number of characters in string to display
//   NCHAR_BITS   - number of bits to specify NCHAR
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.
//
// Each character is 8x12, but pixels are doubled horizontally and
vertically
// so fonts are magnified 2x.  On an XGA screen (1024x768) you can fit
// 64 x 32 such characters.
//
// Needs font_rom.v and font_rom.ngo
//
// For different fonts, you can change font_rom.  For different string
// display colors, change the assignment to cpixel.


//////////////////////////////////////////////////////////////////////
///////
//
// video character string display
//
//////////////////////////////////////////////////////////////////////
///////

module char_string_display (vclock,hcount,vcount,pixel,cstring,cx,cy);

   parameter NCHAR = 8; // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 3; // number of bits in NCHAR

   input vclock;  // 65MHz clock
```

```
   input [10:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0]    vcount; // vertical index of current pixel (0..767)
   output [3:0] pixel;  // char display's pixel
   input [NCHAR*8-1:0] cstring;     // character string to display
   input [10:0] cx;
   input [9:0]    cy;

   // 1 line x 8 character display (8 x 12 pixel-sized characters)

   wire [10:0]    hoff = hcount-1-cx;
   wire [9:0]     voff = vcount-cy;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // <
NCHAR
   wire [2:0]     h = hoff[3:1];            // 0 .. 7
   wire [3:0]     v = voff[4:1];            // 0 .. 11

   // look up character to display (from character string)
   reg [7:0]  char;
   integer  n;
   always @(*)
     for (n=0 ; n<8 ; n = n+1 )           // 8 bits per character
(ASCII)
       char[n] <= cstring[column*8+n];

   // look up raster row from font rom
   wire reverse = char[7];
   wire [10:0] font_addr = char[6:0]*12 + v;    // 12 bytes per
character
   wire [7:0]  font_byte;
   font_rom f(font_addr,vclock,font_byte);

   // generate character pixel if we're in the right h,v area
   wire [3:0] cpixel = (font_byte[7 - h] ^ reverse) ? 4'd15 : 4'd0;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <=
cx+NCHAR*16)
                & (vcount < cy + 24));
   wire [3:0] pixel = dispflag ? cpixel : 4'd1;

endmodule
```

Appendix P – cstringdisp_big.v

```
//////////////////////////////////////////////////////////////////////
///////
//
// video character string display
//
//////////////////////////////////////////////////////////////////////
///////

//// below is my attempt to make the characters bigger

// char_string_display char_string_display1
(vclock,hcount,vcount,pixel,"88888888",cx,cy);
// the simple example above works!

module char_string_display (vclock,hcount,vcount,pixel,cstring,cx,cy);
   parameter NCHAR = 8; // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 3; // number of bits in NCHAR

   input vclock;  // 40MHz clock
   input [10:0] hcount; // horizontal index of current pixel (0..799)
   input [9:0]    vcount; // vertical index of current pixel (0..599)
   output [3:0] pixel;  // char display's pixel
   input [NCHAR*8-1:0] cstring;     // character string to display
   input [10:0] cx;
   input [9:0]    cy;

   // 1 line x 8 character display (8 x 12 pixel-sized characters)

   wire [10:0]    hoff = hcount-1-cx;
   wire [9:0]     voff = vcount-cy;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // <
NCHAR
//   wire [2:0]   h = hoff[3:1];      // 0 .. 7
//   wire [3:0]   v = voff[4:1];            // 0 .. 11

   wire [2:0]     h = hoff[4:2];      // 0 .. 7
   wire [3:0]     v = voff[5:2];            // 0 .. 23


   // look up character to display (from character string)
   reg [7:0]  char;
   integer  n;
   always @(*)
     for (n=0 ; n<8 ; n = n+1 )           // 8 bits per character
(ASCII)
       char[n] <= cstring[column*8+n];

   // look up raster row from font rom
   wire reverse = char[7];
   wire [10:0] font_addr = char[6:0]*12 + v;    // 12 bytes per
character
   wire [7:0]  font_byte;
   font_rom f(font_addr,vclock,font_byte);
```

```
   wire [7:0]  font_byte;
   // generate character pixel if we're in the right h,v area

      wire [3:0] cpixel = (font_byte[7 - h] ^ reverse) ? 4'd15 : 0;

//   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <=
cx+NCHAR*16)
//              & (vcount < cy + 24));
            wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount
<= cx+NCHAR*32)
                  & (vcount < cy + 48));             // double the
range of output

      wire [3:0] pixel = dispflag ? cpixel : 0;

endmodule
```

Appendix Q – delays.v

```
`timescale 1ns / 1ps

// Module: delay (delay)
// Author: Paul Yang
// Description: delays a signal by an arbitrary number of clock cycles


module delay(clk,in,out);



  // Inputs:

  // clk             -  the clock

  // in              -  the signal to delay



  // Outputs:

  // out             -  the delayed signal



  // The length of the delay must be >= 2
  parameter DELAY = 2;

  // The width of the input isgnal
  parameter WIDTH = 8;

  input clk;
  input [WIDTH-1:0] in;
  output [WIDTH-1:0] out;

  reg [DELAY-2:0] delay[WIDTH-1:0];
  reg out;

  integer i;
  always @ (posedge clk)
  begin
    for(i=0; i<WIDTH; i = i+1)
    begin
      {out[i],delay[i]} <= {delay[i][DELAY-2:0],in[i]};
    end
  end
endmodule


// Module: delay (delay)
// Author: Paul Yang
// Description: delays a signal by 1 clock cycle
```

```
module delay1(clk,in,out);
  // Inputs:
  // clk             -  the clock
  // in              -  the signal to delay

  // Outputs:
  // out             -  the delayed signal

  parameter WIDTH = 8;

  input clk;
  input [WIDTH-1:0] in;
  output [WIDTH-1:0] out;

  reg [WIDTH-1:0] out;

  always @(posedge clk)
   out <= in;
endmodule
```

Appendix R – duck_controller.v

```
// duck_controller
// tiff chen

// duck controller is responsible for all the logic for the duck
// as well as the transitioning between the various duck images
// to create the animation effect.


module duck_controller (
      // inputs
      vclock, reset,
      hit_duck, duck_die, // from duck_health
      expire,  // from timer
      appear, disappear, disappear_flag, // from appear_counter
      bush_select, // from bush_selector1
      duck_speed, // from velocity
      hcount, vcount, hsync, vsync, blank,
      new_frame, // from frame
      halt, // from button input
      duck_go,
      duck_height, duck_width,
      sky_height, sky_width,
      // outputs
      appear_enable, disappear_enable,
      duck_pixel, duck_x, duck_y, offscreen, dhsync, dvsync, dblank,
state, debug
      // temporarily ignore rom signals duck_image and duck_addr
      );
   input vclock;  // 40MHz clock
   input reset;   // 1 to initialize module
      input hit_duck; // shots at duck
      input duck_die; // dead duck signal
      input expire;     // controls rate of animation, from timer
      input appear; // 1 to signal for ducks to come out of bushes (1
clock cycle high)
      input disappear; //
      input [1:0] bush_select; // which bush the duck should come out
of
      input [3:0]duck_speed;  // controls duck speed across screen
   input [10:0] hcount; // horizontal index of current pixel (0...799)
   input [9:0] vcount; // vertical index of current pixel (0...599)
   input hsync;         // XVGA horizontal sync signal (active low)
   input vsync;         // XVGA vertical sync signal (active low)
   input blank;         // XVGA blanking (1 means output black pixel)
      input new_frame;
      input halt;
      input duck_go;
      input [9:0] duck_height;
      input [10:0] duck_width;
      input [9:0] sky_height;
      input [10:0] sky_width;

      output appear_enable; // enables appear_counter to generate
appear signal
```

```
      output disappear_enable; // duck's disappearing timer resets
      output disappear_flag; // 1 if duck is in the process of flying
off screen
                                                    // (duck is immune to
shots)
      output [3:0] duck_pixel; // duck's pixels
      output [10:0] duck_x; // duck's x-coordinate
      output [9:0] duck_y; // duck's y-coordinate
      output offscreen;
   output dhsync; // duck game's horizontal sync
   output dvsync; // duck game's vertical sync
   output dblank; // duck game's blanking

      // for debugging
      output [3:0] state;
      output debug;

      wire [10:0] duck_x_init = (bush_select == 2'd0) ? 11'd50 :
                                (bush_select == 2'd1) ? 11'd200 :
                                (bush_select == 2'd2) ? 11'd350 :
                                (bush_select == 2'd3) ? 11'd450 :
                                11'd0;

      // pixel wires from duck ROMs
      wire [3:0] duckdie_pixel;
      wire [3:0] duckdown_east_pixel;
      wire [3:0] duckdown_west_pixel;
      wire [3:0] duckup_east_pixel;
      wire [3:0] duckup_west_pixel;
      wire [3:0] duckhit_pixel;

      reg [13:0] duck_add = 14'd0; // depth of ROM is 12100
      reg [13:0] duck_add_tmp = 14'd0;
      reg [3:0] duck_pixel; // width of ROM is 4
      reg [10:0] duck_x;
      reg [9:0] duck_y;
      reg east; // 1 if duck is traveling east, 0 if traveling west
      reg south; // 1 if duck is traveling south, 0 if travling north
      reg duck_die_flag; // flag to signal dead duck if duck_die
happens to be
                                             // high during ~new_frame
      reg disappear_flag; // continuous 1 if duck is supposed to be off
screen
      reg [3:0] state; // state assignment for duck image
      reg offscreen; // 1 if duck is off sky area of screen
      reg disappear_enable;
      reg appear_enable;
      reg debug;
      // assign debug = disappear_flag; // for debuging purposes

      // state assignments for duck
      parameter UPNE = 4'd0; // wings up heading northeast
      parameter DOWNNE = 4'd1; // wings down heading northeast
      parameter DOWNNW = 4'd2; // wings up heading northwest
      parameter UPNW = 4'd3; // wings down heading northwest
      parameter UPSE = 4'd4; // wings up heading southeast
      parameter DOWNSE = 4'd5; // wings down heading southeast
```

```
      parameter UPSW = 4'd6; // wings up heading southwest
      parameter DOWNSW = 4'd7;  // wings down heading southwest
      parameter HIT1 = 4'd8; // shot duck 1
      parameter HIT2 = 4'd9; // shot duck 2
      parameter HIT3 = 4'd10; // shot duck 3    (3 needed to slow down
animation) A
      parameter DIE = 4'd11;  // dead duck
                                             B
      parameter FLYAWAY = 4'd12; // duck flies away
                               C
      parameter WAITING = 4'd13; // duck waiting to come out of bushes
             D

      // instantiating ROMs with duck images
      // these are each 4x110x110
      duckdie
duckdie1(.addr(duck_add),.clk(vclock),.dout(duckdie_pixel));
      duckdown_east duckdown_east1(.addr(duck_add),.clk(vclock),
            .dout(duckdown_east_pixel));
      duckdown_west duckdown_west1(.addr(duck_add),.clk(vclock),
            .dout(duckdown_west_pixel));
      duckup_east duckup_east1(.addr(duck_add),.clk(vclock),
            .dout(duckup_east_pixel));
      duckup_west duckup_west1(.addr(duck_add),.clk(vclock),
            .dout(duckup_west_pixel));
      duckhit
duckhit1(.addr(duck_add),.clk(vclock),.dout(duckhit_pixel));

      parameter GRAVITY = 4; // rate at which duck drops from sky

      always @ (posedge vclock) begin

      if ((reset) || (duck_go)) begin
            // reset x & y coordintes of duck
            duck_x <= duck_x_init; // start x coordinate of duck
            // CHANGEME INTEGRATION
            duck_y <= sky_height;  // start y coordinate of duck
            // duck initially travels in southeast direction
            east <= 1;
            south <= 0;
            state <= (duck_go) ? WAITING : DIE;
            duck_add <= 14'd0;
            disappear_flag <= 0;
            disappear_enable <= 1;
            appear_enable <= 1;
            offscreen <= 1;
            end // end reset

      else begin

      // logic to determine whether duck should be on screen
      // when disappear_flag = 1, duck is immune to shots
//    if (duck_go)
//          disappear_flag <= 0;
//    else if (disappear)
//          disappear_flag <= 1;
//    else disappear_flag <= disappear_flag;
```

```
      disappear_flag <= (state == FLYAWAY);


      // update pixel according to state
      // wings up
      if (state == WAITING) duck_pixel <= 4'd13;
      else if (((state == UPNE) || (state == UPNW) ||
              (state == UPSE) || (state == UPSW))) begin
              case ({south,east})
                        2'b00: duck_pixel <= duckup_west_pixel;
                        2'b01: duck_pixel <= duckup_east_pixel;
                        2'b10: duck_pixel <= duckup_west_pixel;
                        2'b11: duck_pixel <= duckup_east_pixel;
                  endcase
            end //UPNE, UPNW, UPSE, UPSW
      // wings down
      else if (((state == DOWNNE) || (state == DOWNNW) ||
           (state == DOWNSE) || (state == DOWNSW))) begin
            case ({south,east})
                  2'b00: duck_pixel <= duckdown_west_pixel;
                  2'b01: duck_pixel <= duckdown_east_pixel;
                  2'b10: duck_pixel <= duckdown_west_pixel;
                  2'b11: duck_pixel <= duckdown_east_pixel;
            endcase
            end // DOWNNE, DOWNNW, DOWNSE, DOWNSW
      else if ((state == HIT1) || (state == HIT2) || (state == HIT3))
            duck_pixel <= duckhit_pixel;
      else if (state == DIE)
            duck_pixel <= duckdie_pixel;
      else if (state == FLYAWAY)
            case ({south,east})
                  2'b00: duck_pixel <= duckdown_west_pixel;
                  2'b01: duck_pixel <= duckdown_east_pixel;
                  2'b10: duck_pixel <= duckdown_west_pixel;
                  2'b11: duck_pixel <= duckdown_east_pixel;
            endcase
      else duck_pixel <= duck_pixel;


      // if hcount & vcount are within the range of the duck image
      if ((duck_x <= hcount) && (hcount < duck_x + duck_width) &&
             (duck_y <= vcount) && (vcount < duck_y + duck_height))
begin
            //if (duck_add == 14'd12099)
//          if (((hcount == duck_x + duck_width) && (vcount == duck_y +
duck_height)) ||
//                ((appear) && (state == WAITING)))
//                duck_add <= 14'd0;
//          else
//                duck_add <= duck_add + 1;
            duck_add <= (vcount - duck_y) * duck_width + (hcount -
duck_x);
            end // end duck_add

      // 1 if duck is off the screen
      if (((duck_y >= sky_height) || (duck_y <= 13'd0)) &&
```

```
            ((state == WAITING) || (state == FLYAWAY) || (state ==
DIE)))
            offscreen <= 1;
      else
            offscreen <= 0;


//    if ((duck_x < hcount) && (hcount <= duck_x + DUCK_WIDTH) &&
//          (duck_y < vcount) && (vcount <= duck_y + DUCK_HEIGHT))
begin
//          duck_add_tmp <= (duck_add_tmp == 14'd12099) ? 14'd0 :
duck_add_tmp + 1;
//          // wings up
//          if ((state == UPNE) || (state == DOWNSE) || (state ==
UPSE) || (state == DOWNNE))
//                duck_add <= duck_add_tmp;
//          else if ((state == UPNW) || (state == UPSW) || (state ==
DOWNSW) || (state == DOWNNW))
//                duck_add <= 4'd110 - duck_add_tmp + (4'd110) *
vcount;
//          else if ((state == HIT1) || (state == HIT2) || (state ==
HIT3))
//                duck_add <= duck_add_tmp;
//          else if (state == DIE)
//                duck_add <= duck_add_tmp;
//
//          end // end duck_addr calculations


      // update duck states
      if (halt)
            state <= state;
      else if ((appear) && (state == WAITING))   begin
            state <= UPNE;
//          disappear_enable <= 1;
            end
      else if (hit_duck)
            state <= HIT1;
      else if (duck_die)
            state <= DIE;
      else if ((~(state == DIE)) && (~(state == WAITING)) &&
(disappear))
            state <= FLYAWAY;
      else begin
            // display new frame with updated duck x & y coordinates
            if (new_frame) begin

                  // update state for animation on timer's expire
signal
                  if (expire) begin
                        if (((state == UPNE) || (state == UPNW) ||
                              (state == UPSE) || (state == UPSW))
                              // && (new_frame)
                              ) begin
                              case ({south, east})
                                    2'b00: state <= DOWNNW;
                                    2'b01: state <= DOWNNE;
```

```
                                    2'b10: state <= DOWNSW;
                                    2'b11: state <= DOWNSE;
                                    endcase
                              end //UPNE, UPNW, UPSE, UPSW
                        // wings down
                        else if (((state == DOWNNE) || (state ==
DOWNNW) ||
                        (state == DOWNSE) || (state == DOWNSW))
                              // && (new_frame)
                  ) begin
                              case ({south, east})
                                    2'b00: state <= UPNW;
                                    2'b01: state <= UPNE;
                                    2'b10: state <= UPSW;
                                    2'b11: state <= UPSE;
                                    endcase
                              end // DOWNNE, DOWNNW, DOWNSE, DOWNSW
                        else if (state == HIT1) state <= HIT2;
                        else if (state == HIT2) state <= HIT3;
                        else if (state == HIT3) state <= UPNE;
                        else if (state == DIE) state <= DIE;
                        else if (state == FLYAWAY) state <= FLYAWAY;
                        else state <= state;
                        end // end expire
                  end // end new_frame
            end // end else

      // logic to reset disappear_enable
      if (
            (state == FLYAWAY) ||
            ((appear) && (state == WAITING))
            )
            disappear_enable <= 1;
      else if (~(state == HIT3))
            disappear_enable <= 0;
      else
            disappear_enable <= disappear_enable;


      // updating flags
      if (offscreen) begin
            duck_die_flag <= 0;
            end // end offscreen
      else if ((~new_frame) && (duck_die))
            duck_die_flag <= 1;
      else duck_die_flag <= duck_die_flag;


      // updating x & y coordinates of duck
      if (new_frame) begin

            if (halt) begin // stop duck's x & y coordinates from
moving
                  duck_x <= duck_x;
                  duck_y <= duck_y;
                  state <= state;
                  end // end halt begin
```

```
            else if (state == WAITING) begin
                  duck_x <= duck_x_init;
                  duck_y <= sky_height - duck_height - 1;
                  end // end WAITING
            else if ((state == HIT1) || (state == HIT2) || (state ==
HIT3)) begin
                  duck_x <= duck_x;
                  duck_y <= duck_y;
                  end // end HIT1 HIT2
            else if (state == DIE) begin
                  if (((duck_die_flag) || (duck_die)) & (east)) begin
                        duck_x <= duck_x + duck_speed;
                        duck_die_flag <= 0;
                        end // end east & dead duck
                  else if (((duck_die_flag) || (duck_die)) & (~east))
begin
                        duck_x <= duck_x - duck_speed;
                        duck_die_flag <= 0;
                        end // end west & dead duck
                  duck_y <= (duck_y > sky_height) ? duck_y : duck_y +
GRAVITY;
                  end // end DIE
            else if (state == FLYAWAY) begin
                  duck_x <= duck_x;
                  duck_y <= (duck_y >= sky_height) ? sky_height :
duck_y - GRAVITY;
                  //offscreen <= 1;
                  end // end FLYAWAY;
            else begin
                  // controlling the x coordinate of duck
                  if (east) begin // duck is traveling east
                        if (duck_x + duck_width + duck_speed >=
sky_width)  begin // duck hits right edge of screen
                        duck_x <= sky_width - duck_width;
                              east <= 0; // change horizontal direction
                              end // end hit right edge of screen begin
                  else duck_x <= duck_x + duck_speed; // duck x
coordinate increments to move duck east
                        end // end east begin
                  else if (~east) begin // duck is traveling west
                        // duck_x - duck_speed <= 0 --> duck_x <=
duck_speed
                        if (duck_x <= duck_speed) begin // duck hits
left edge of screen
                              duck_x <= 11'd0;
                              east <= 1; // change horizontal direction
                              end // end hit left edge of screen
                        else duck_x <= duck_x - duck_speed; // duck x
coordinate decrements to move duck west
                        end // end west begin
                  // controlling the y coordinate of duck
                  if (south) begin // duck is traveling south
                        if (duck_y + duck_height + duck_speed >=
sky_height) begin // duck hits bottom edge of screen
                              duck_y <= sky_height - duck_height;
                              south <= 0; // change vertical direction
```

```
                              end // end duck hits bottom edge of
screen begin
                        else duck_y <= duck_y + duck_speed; // duck y
coordinate decrements to move duck south
                        end // south begin
                  else if (~south) begin // duck is traveling north
                        if (duck_y <= duck_speed) begin // duck hits
top edge of screen
                              duck_y <= 10'd0;
                              south <= 1; // change vertical direction
                              end // end duck hits top edge of screen
                        else duck_y <= duck_y - duck_speed; // duck y
coordinate decrements to move duck north
                        end // end ~south begin
                  end // end else
            end // end new_frame begin
      end // end reset else
      end // always

      assign dhsync = hsync;
   assign dvsync = vsync;
   assign dblank = blank;

endmodule
```

Appendix S – duck_health.v

```
// duck_health
// tiff chen

// keeps track of duck's health it takes 3 shots to kill the duck
// outputs duck_die, which is high for one clock cycle
// when duck is supposed to die

module duck_health(vclock, reset, duck_go,
      disappear_flag, kill_duck,
      duck_x, duck_y, duck_width, duck_height,
      // trigger,
      dot_x,dot_y,
      hit_duck, duck_die
      , duck_shots, prev_duck_die
      );

      input vclock; // 40 mhz
      input reset;
      input duck_go;
      input disappear_flag;
      input kill_duck; // THIS WILL BE THE TRIGGER SIGNAL
      input [10:0] duck_x;
      input [9:0] duck_y;
      input [10:0] duck_width;
      input [9:0] duck_height;
      input [10:0] dot_x; // x coordinate of laser dot from dot finder
      input [9:0] dot_y; // y coordinate of laser dot from dot finder
//    input [10:0] aim_point_x;
//    input [9:0] aim_point_y;
      output hit_duck;
      output duck_die; // high for one clock cycle
                                    // when duck is supposed to die
      // debugging
      output duck_shots;
      output prev_duck_die;

      reg [1:0] duck_shots; // number of hits
      reg hit; // 1 if user hits duck
      reg prev_hit; // stores previous duck_die value
      reg hit_duck; // 1 if user hits duck, but is only
                              // high for one clock cycle
                              // delay of one clock cycle after
                              // posedge of hit
      reg duck_die;
      reg prev_duck_die;
      reg disable_shots; // 1 to disable shots
                                          // to prevent duck from
                                          // getting shot after it
                                          // dies
      reg [23:0] counter;


      always @ (posedge vclock) begin
```

```
      if ((reset) || (duck_go))  begin
            duck_shots <= 2'd0;
            hit <= 0;
            disable_shots <= 0;
            prev_duck_die <= 0;
            counter <= 0;
            end

      else begin

            if ((~disable_shots) && (kill_duck) && (counter == 24'd0))
                  counter <= 24'd4000000; // sample data for 1/10 of a
second (4 million clock cycles)
                                                                  // to
account for lag
            else if (counter != 24'd0) begin
                  counter <= counter - 1;

                  if (((duck_x <= dot_x) && (dot_x <= duck_x +
duck_width)) &&
                        ((duck_y <= dot_y) && (dot_y <= duck_y +
duck_height))) begin
                        hit <= 1;
                        counter <= 24'd0;
                        end // end duckxy

                  end // end ~disable_shots & kill_duck
            else hit <= 0;


            // make hit_duck 1 clock cycle delay after posedge
            // of hit and have the signal duration only 1 cycle high
            // duck dies at 3 shots
            disable_shots <= ((duck_shots == 2'd3) || (disappear_flag))
? 1 : disable_shots;
            hit_duck <= ((hit) && (~prev_hit));
            duck_die <= ((duck_shots == 2'd3) && (prev_duck_die ==
1'b0) && (duck_die == 1'b0));


            // duck_shots & duck_die logic
            if (hit_duck) duck_shots <= (disable_shots) ? 2'd0 :
duck_shots + 1;
            else duck_shots <= duck_shots;

            prev_hit <= hit;
            prev_duck_die <= (duck_die) ? duck_die : prev_duck_die;

            end // end else
      end // end always

endmodule
```

Appendix T – frame.v

```
// frame
// tiff chen

// frame takes in the vsync signal and generates a
// new_frame signal that is high for one clock cycle
// after vsync is low.

module frame(vclock, vsync, new_frame);

      input vclock; // 40 MHz
   input vsync;         // XVGA vertical sync signal (active low)

      output new_frame; // new frame signal

      reg prev_vsync; // previous vsync value
      reg new_frame;

      always @ (posedge vclock) begin
      // make new_frame high for one clock cycle after vsync is low
      // so that the image will only update when entire image
      // has been displayed
      new_frame <= ((~(prev_vsync == vsync)) & (vsync == 0)) ? 1 : 0;
      prev_vsync <= vsync;

      end

endmodule
```

Appendix U – mult_model.v

```
/////////////////////////////////////////////////////////////////////
****
 **
 ** Module: YCrCb2RGB
 **
 ** Instantiated Multiplier:

/////////////////////////////////////////////////////////////////////
****/

// Module: YCrCb2RGB (YCrCb to RGB color converter)
// Author: Xilinx with minor modifications by Paul Yang
// Description: Converts a YCrCb color value to RGB


module YCrCb2RGB ( R, G, B, clk, rst, Y, Cr, Cb, x_in, y_in, x_out,
y_out,

                   dv_in, dv_out,
                   ram_addr_in, ram_addr_out );

output [7:0]  R, G, B;
output [10:0] x_out;
output [9:0] y_out;
output dv_out;
output [18:0] ram_addr_out;

input clk,rst;
input[9:0] Y, Cr, Cb;
input [10:0] x_in;
input [9:0] y_in;
input dv_in;
input [18:0] ram_addr_in;

//1.164 = 01.00101001
//1.596 = 01.10011000
//0.813 = 00.11010000
//0.392 = 00.01100100
//2.017 = 10.00000100

wire [10:0] Y_int, Cr_int, Cb_int;
reg [12:0] R_int1;
reg [12:0] G_int1;
reg [12:0] B_int1;
reg [12:0] R_int2;
reg [12:0] G_int2;
reg [12:0] B_int2;
reg [9:0] Y_reg, Cr_reg, Cb_reg;
//reg [10:0] Y_reg, Cr_reg, Cb_reg;
reg [17:0] const1,const2,const3,const4,const5;

wire [35:0] P1,P2,P3,P4,P5;
reg [11:0] P1_int,P2_int,P3_int,P4_int,P5_int;
wire [12:0] P1_int_act,P2_int_act,P3_int_act,P4_int_act,P5_int_act;
```

```
reg sign_y1,sign_cr1,sign_cb1,sign_y2,sign_cr2,sign_cb2;

wire[17:0] Y_reg_in, Cr_reg_in, Cb_reg_in;

assign Y_int = Y - 'd64;
assign Cr_int = Cr - 'd512;
assign Cb_int = Cb - 'd512;

//save signs of above internal signals
always @ (posedge clk or posedge rst)
if (rst)
      begin
       sign_y1 <= 0; sign_cr1 <= 0; sign_cb1 <= 0;
       sign_y2 <= 0; sign_cr2 <= 0; sign_cb2 <= 0;
      end
else
      begin
       sign_y1 <= Y_int[10]; sign_cr1 <= Cr_int[10]; sign_cb1 <=
Cb_int[10];
       sign_y2 <= sign_y1; sign_cr2 <= sign_cr1; sign_cb2 <= sign_cb1;
      end



// 2's complement logic

always @ (posedge clk or posedge rst)
if (rst)
      begin
       Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
      end
else
      begin
         if (Y_int[10] )
             Y_reg <= (-Y_int);//if negative take complement ie.,
absolute value
         else
             Y_reg <= Y_int;
         if (Cr_int[10])
             Cr_reg <= (-Cr_int);//if negative take complement ie.,
absolute value
         else
             Cr_reg <= Cr_int;
         if (Cb_int[10])
             Cb_reg <= (-Cb_int);//if negative take complement ie.,
absolute value
         else
             Cb_reg <= Cb_int ;
      end


always @ (posedge clk)
begin
 const1 = 18'b 0100101010;
 const2 = 18'b 0110011000;
 const3 = 18'b 0011010000;
 const4 = 18'b 0001100100;
```

```
 const5 = 18'b 1000000100;
end

MULT18X18 U1_MULT18X18 ( .A(const1), .B({8'b00000000,Y_reg}), .P(P1));
// 1.164(Y-64) = 01.00101001(Y-64)
MULT18X18 U2_mult18x18 ( .A(const2), .B({8'b00000000,Cr_reg}), .P(P2));
// 1.596(Cr-512) = 01.10011000(Cr-512)
MULT18X18 U3_mult18x18 ( .A(const3), .B({8'b00000000,Cr_reg}), .P(P3));
// 0.813(Cr-512) = 00.11010000(Cr-512)
MULT18X18 U4_mult18x18 ( .A(const4), .B({8'b00000000,Cb_reg}), .P(P4));
// 0.392(Cb-512) = 00.01100100(Cb-512)
MULT18X18 U5_mult18x18 ( .A(const5), .B({8'b00000000,Cb_reg}), .P(P5));
// 2.017(Cb-512) = 10.00000100(Cb-512)


/* 10 bits * 10 bits gives 20 bit products. 8 LSBs ignored */
always @ (posedge clk or posedge rst)
if (rst)
      begin
       P1_int <= 0; P2_int <= 0; P3_int <= 0; P4_int <= 0; P5_int <= 0;
      end
else
       begin
       P1_int <= P1[19:8]; P2_int <= P2[19:8]; P3_int <= P3[19:8];
P4_int <= P4[19:8]; P5_int <= P5[19:8];
      end

assign P1_int_act = (sign_y2) ? (-P1_int) : P1_int; // if product is
negative, take 2's compl.
assign P2_int_act = (sign_cr2) ? (-P2_int) : P2_int; // if product is
negative, take 2's compl.
assign P3_int_act = (sign_cr2) ? (-P3_int) : P3_int; // if product is
negative, take 2's compl.
assign P4_int_act = (sign_cb2) ? (-P4_int) : P4_int; // if product is
negative, take 2's compl.
assign P5_int_act = (sign_cb2) ? (-P5_int) : P5_int; // if product is
negative, take 2's compl.



always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int1 <= 0; G_int1 <= 0; B_int1 <= 0;
       R_int2 <= 0; G_int2 <= 0; B_int2 <= 0;
      end
   else
     begin
      R_int1 <= {P1_int_act} + {P2_int_act};
      G_int1 <= {P1_int_act} - {P3_int_act};
      B_int1 <= {P1_int_act} + {P5_int_act};
      R_int2 <= R_int1;
      G_int2 <= G_int1 - {P4_int_act};
//G_int2 <= {P1_int_act} - {P3_int_act} - {P4_int_act};
      B_int2 <= B_int1;

     end
```

```
// Delay outputs by 5 clocks
synchronize d1(clk, x_in, x_out);
defparam d1.NSYNC = 5;
defparam d1.WIDTH = 11;
synchronize d2(clk, y_in, y_out);
defparam d2.NSYNC = 5;
defparam d2.WIDTH = 10;
synchronize d3(clk, dv_in, dv_out);
defparam d3.NSYNC = 5;
defparam d3.WIDTH = 1;
synchronize d4(clk, ram_addr_in, ram_addr_out);
defparam d4.NSYNC = 5;
defparam d4.WIDTH = 19;



/* output limiter . Limit output to 0 if Rint,Gint and Bint < 0 and
limit output to 4095(1111,1111,1111) if Rint,Gint and Bint > 4095 */

assign R =  R_int2[12] ? 8'b0 : (R_int2[11:10] == 2'b0) ? R_int2[9:2] :
8'b11111111;
assign G =  G_int2[12] ? 8'b0 : (G_int2[11:10] == 2'b0) ? G_int2[9:2] :
8'b11111111;
assign B =  B_int2[12] ? 8'b0 : (B_int2[11:10] == 2'b0) ? B_int2[9:2] :
8'b11111111;




endmodule

module MULT18X18 (A, B, P); // synthesis syn_black_box
input [17:0]A;
input [17:0]B;
output [35:0]P;


endmodule
```

Appendix V – pixel_counter.v

```
`timescale 1ns / 1ps
// Module: pixel_counter (pixel counter)
// Author: Paul Yang
// Description: Reads the output of the ntsc_decoder and creates x,y

// coordinates for each YCrCb sample



module pixel_counter(clk, reset, ycrcb_in, f, v, h, dv_in, x, y,
                     ycrcb_out, dv_out, ram_addr);
    // Inputs:
    // clk           -  the clock at 27Mhz
    // reset         -  reset
    // ycrcb_in      -  the ycrcb data from the NTSC decoder
    // f             -  field data from the NTSC decoder
    //                  1 indicates an even field, 0 an odd field
    // v             -  vertical sync from the NTSC decoder
    // h             -  horizontal sync from the NTSC decoder
    // dv_in         -  data valid from the NTSC decoder

    // Outputs:
    // x             -  the X coordinate of the data
    // y             -  the Y coordinate of the data
    // ycrcb_out     -  the color data corresponding to the X,Y
coordinate
    // dv_out        -  indicates whether or not the output data is
valid

    // ram_addr      -  the RAM address that corresponds to the (x,y)
location


    parameter NTSC_FRAME_WIDTH = 11'd720;
    parameter NTSC_FRAME_HEIGHT = 10'd525;
    parameter VIDEO_FRAME_WIDTH = 19'd800;
    parameter LINE_OFFSET = 19'd80;

    input clk;
    input reset;
    input [29:0] ycrcb_in;
    input   f;
    input   v;
    input   h;
    input   dv_in;

    output [10:0] x;
    output [9:0] y;
    output [29:0] ycrcb_out;
    output [18:0] ram_addr;
    output dv_out;

    reg [10:0] x;
    reg [9:0] y;
```

```
    reg [29:0] ycrcb_out;
    reg dv_out;

    reg [18:0] ram_addr;

    reg h_first;


    always @(posedge clk)
    begin


      // On a horizontal sync
      if(h)
      begin
         x <= 11'd0;
      end

      // Not a horizontal sync
      else
      begin
         // There is a spurious first increment - d_first cancels it
out
         if( dv_in && !v && x < NTSC_FRAME_WIDTH  && y <
NTSC_FRAME_HEIGHT)
         begin
            x <= x + 1;
            ram_addr <= ram_addr + 19'd1;
         end
      end


      // On a vertical sync
      if(v)
      begin

         // Odd field
         if(f)
         begin
            h_first <= 1;
            y <= 10'd1;
            ram_addr <= VIDEO_FRAME_WIDTH;
         end

         // Even field
         else
         begin
            h_first <= 1;
            y <= 10'd0;
            ram_addr <= 19'd0;
         end
      end

      // Not a vertical sync
      else
      begin
         // There is a spurious line increment - h_first cancels it out
```

```
         if(h_first && h && y < NTSC_FRAME_HEIGHT)
         begin
            h_first <= 0;
         end
         else if(h && y < NTSC_FRAME_HEIGHT)
         begin
            y <= y + 2;
            ram_addr <= ram_addr + VIDEO_FRAME_WIDTH + LINE_OFFSET;
         end
       end


       // Delay the dv and ycrcb signals to match the calculated x,y
coordinates
       dv_out <= dv_in;
       ycrcb_out <= ycrcb_in;
     end

endmodule
```

Appendix W – score_strings.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////
// Company:
// Engineer:
//
// Create Date:    23:03:44 12/05/06
// Design Name:
// Module Name:    score_strings
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////

module score_strings (clk, hcount, vcount,
       score_val,
      score_string_val);

      input clk; // 40 MHz
      //input [10:0] hcount; // pixel number on current line
      //input [9:0] vcount; // line number
      input [3:0] score_val; //

      output [2*8-1:0] score_string_val;

      reg [2*8-1:0] score_string_val;


   always @(posedge clk) begin
      begin

                  case (score_val)

                        4'd0: score_string_val <= "00"; //
                        4'd1: score_string_val <= "01"; //
                  4'd2: score_string_val <= "02"; //
                        4'd3: score_string_val <= "03"; //
                        4'd4: score_string_val <= "04"; //
                        4'd5: score_string_val <= "05"; //
                        4'd6: score_string_val <= "06"; //
                        4'd7: score_string_val <= "07"; //
                        4'd8: score_string_val <= "08"; //
                        4'd9: score_string_val <= "09"; //
                        4'd10: score_string_val <= "10"; //
                        4'd11: score_string_val <= "11"; //
```

```
                        4'd12: score_string_val <= "12"; //
                        4'd13: score_string_val <= "13"; //
                        4'd14: score_string_val <= "14"; //
                        4'd15: score_string_val <= "15"; //
                        default: score_string_val <= "00"; //
                    endcase
                end // end else
            end // end always

endmodule
```

Appendix X – scorekeeper.v

```
//
// File:   scorekeeper.v
// Date:   30-11-2006
// Author: Spencer Sugimoto
//
//
// INPUTS:
//
//   vclock       - video pixel clock
//   hcount       - horizontal (x) location of current pixel
//   vcount       - vertical (y) location of current pixel
//   cx,cy        - pixel location (upper left corner) to display score
at
//
// OUTPUT:
//
//   pixel        - video pixel value to display at current location
//
// PARAMETERS:
//
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.

//////////////////////////////////////////////////////////////////////
///////
//
// divider module
//
//    counts seconds
//
//////////////////////////////////////////////////////////////////////
///////



//////////////////////////////////////////////////////////////////////
///////
//
// scorekeeper module
//
//////////////////////////////////////////////////////////////////////
///////







module scorekeeper (//inputs
                                    vclock,
```

```
                                        duck_die,
                                        reset,
                                        timer_expire,
                                        //output
                                        gameover,
                                        win,
                                        score_val,
                                        // freeze
                                        );

      input vclock;                 // 40MHz clock
   input duck_die;      //    duck dies
      input reset;                  // reset
      input timer_expire; //game time limit expended

      output gameover;
      output win;
   output [3:0] score_val;
      output freeze;

      parameter WIN_SCORE = 4'd6;

      reg [3:0] score = 4'd0;
      reg win;
      reg gameover;

      always @ (posedge vclock)
   begin
            if (reset) begin
                  score <= 4'd0;
                  gameover <= 0;
                  win <= 0;
                  end // end reset
            else if ((gameover) || (win))
                  score <= score;
            else if ((duck_die) && ~(gameover) && ~(win))
                  score <= score + 4'd1; // increment score if score <
win_score
            else
                  score <= score; // if nothing, no change

            if (~reset) begin
                  win <= (score == WIN_SCORE) ? 1 : win;
                  gameover <= (timer_expire) ? 1 : gameover;

                  end

      end

//    // wire gameover = 1'b0;
//    assign gameover = (timer_expire) ? ; // expired will be high for
only one clock period
//
//    wire win;
//    assign win = (score == WIN_SCORE); // expired will be high for
only one clock period
```

```
      assign score_val[3:0] = score[3:0];
endmodule
```

Appendix Y – second_timer.v

```
// second_timer
// tiff chen

// second_timer generates the one_hz_enable pulse.
// This output signal is high for one clock cycle every 1 second
// to allow for timing mechanisms in the duck controller
// to count by the second.

module second_timer(clk, reset, one_hz_enable);

   input clk; // 40 MHz
   input reset; // global reset signal
   output one_hz_enable; // one_hz_enable to appear_counter



   reg[25:0] count;      //2^26 is 67108864
   always @ (posedge clk)
      begin
            if (reset)
                  count <= 0;
            else if ((~reset) & (count < 39999999))
                  // make sure count is less than 39999999 & we're not
resetting
                  count <= count + 1;
            else
                  count <= 0; //reset_sync resets the count
      end


      assign one_hz_enable = (count[25:0] == 26'd39999999);
            // one_hz_enable will be high for one clock cycle every
40x10^6 cycles

            // which is one second

endmodule
```

Appendix Z – seedgen.v

```
// seed generator
// tiff chen

// generates a seed for the random number generators
// for the bush_select and r_interval signals

module seedgen(clk, seed);
      input clk;
      output [2:0] seed;

      reg [2:0] seed;

      always @ (posedge clk)
            seed <= seed + 1;

endmodule
```

Appendix AA – signal_to_pulse.v

```
// signal_to_pulse
// tiff chen

// takes in a signal and generates a 1-cycle high pulse at the
// positive edge of the signal

module signal_to_pulse(clk, signal, pulse);

      input clk; // 40 MHz
   input signal;        // XVGA vertical sync signal (active low)

      output pulse; // new frame signal

      reg prev_signal; // previous vsync value
      reg pulse;

      always @ (posedge clk) begin
      // make new_frame high for one clock cycle after vsync is low
      // so that the image will only update when entire image
      // has been displayed
      pulse <= ((~(prev_signal == signal)) & (signal == 1)) ? 1 : 0;
      prev_signal <= signal;

      end

endmodule
```

Appendix BB – sound.v

```
////////////////////////////////////////////////////////////////////////
////////
//
// bi-directional monaural interface to AC97
//
////////////////////////////////////////////////////////////////////////
////////

module lab4audio (clock_27mhz, reset, volume,
                  audio_in_data, audio_out_data, ready,
                audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                  ac97_synch, ac97_bit_clock);

   input clock_27mhz;
   input reset;
   input [4:0] volume;
   output [7:0] audio_in_data;
   input [7:0] audio_out_data;
   output ready;

   //ac97 interface signals
   output audio_reset_b;
   output ac97_sdata_out;
   input ac97_sdata_in;
   output ac97_synch;
   input ac97_bit_clock;

   wire [2:0] source;
   assign source = 0;      //mic

   wire [7:0] command_address;
   wire [15:0] command_data;
   wire command_valid;
   wire [19:0] left_in_data, right_in_data;
   wire [19:0] left_out_data, right_out_data;

   reg audio_reset_b;
   reg [9:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end

   wire ac97_ready;
   ac97 ac97(ac97_ready, command_address, command_data, command_valid,
             left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
```

```
            right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
            ac97_bit_clock);

   // ready: one cycle pulse synchronous with clock_27mhz
   reg [2:0] ready_sync;
   always @ (posedge clock_27mhz) begin
     ready_sync <= {ready_sync[1:0], ac97_ready};
   end
   assign ready = ready_sync[1] & ~ready_sync[2];

   reg [7:0] out_data;
   always @ (posedge clock_27mhz)
     if (ready) out_data <= audio_out_data;
   assign audio_in_data = left_in_data[19:12];
   assign left_out_data = {out_data, 12'b000000000000};
   assign right_out_data = left_out_data;

   // generate repeating sequence of read/writes to AC97 registers
   ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                     command_valid, volume, source);
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch,
ac97_bit_clock);

   output ready;
   input [7:0] command_address;
   input [15:0] command_data;
   input command_valid;
   input [19:0] left_data, right_data;
   input left_valid, right_valid;
   output [19:0] left_in_data, right_in_data;

   input ac97_sdata_in;
   input ac97_bit_clock;
   output ac97_sdata_out;
   output ac97_synch;

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

   reg [19:0] l_cmd_addr;
   reg [19:0] l_cmd_data;
   reg [19:0] l_left_data, l_right_data;
   reg l_cmd_v, l_left_v, l_right_v;
   reg [19:0] left_in_data, right_in_data;
```

```
   initial begin
      ready <= 1'b0;
      // synthesis attribute init of ready is "0";
      ac97_sdata_out <= 1'b0;
      // synthesis attribute init of ac97_sdata_out is "0";
      ac97_synch <= 1'b0;
      // synthesis attribute init of ac97_synch is "0";

      bit_count <= 8'h00;
      // synthesis attribute init of bit_count is "0000";
      l_cmd_v <= 1'b0;
      // synthesis attribute init of l_cmd_v is "0";
      l_left_v <= 1'b0;
      // synthesis attribute init of l_left_v is "0";
      l_right_v <= 1'b0;
      // synthesis attribute init of l_right_v is "0";

      left_in_data <= 20'h00000;
      // synthesis attribute init of left_in_data is "00000";
      right_in_data <= 20'h00000;
      // synthesis attribute init of right_in_data is "00000";
   end

   always @(posedge ac97_bit_clock) begin
      // Generate the sync signal
      if (bit_count == 255)
        ac97_synch <= 1'b1;
      if (bit_count == 15)
        ac97_synch <= 1'b0;

      // Generate the ready signal
      if (bit_count == 128)
        ready <= 1'b1;
      if (bit_count == 2)
        ready <= 1'b0;

      // Latch user data at the end of each frame. This ensures that
the
      // first frame after reset will be empty.
      if (bit_count == 255)
        begin
           l_cmd_addr <= {command_address, 12'h000};
           l_cmd_data <= {command_data, 4'h0};
           l_cmd_v <= command_valid;
           l_left_data <= left_data;
           l_left_v <= left_valid;
           l_right_data <= right_data;
           l_right_v <= right_valid;
        end

      if ((bit_count >= 0) && (bit_count <= 15))
        // Slot 0: Tags
        case (bit_count[3:0])
          4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
          4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
          4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
          4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
```

```
        4'h4: ac97_sdata_out <= l_right_v; // Right data valid
          default: ac97_sdata_out <= 1'b0;
        endcase

      else if ((bit_count >= 16) && (bit_count <= 35))
        // Slot 1: Command address (8-bits, left justified)
        ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

      else if ((bit_count >= 36) && (bit_count <= 55))
        // Slot 2: Command data (16-bits, left justified)
        ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

      else if ((bit_count >= 56) && (bit_count <= 75))
        begin
           // Slot 3: Left channel
           ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
           l_left_data <= { l_left_data[18:0], l_left_data[19] };
        end
      else if ((bit_count >= 76) && (bit_count <= 95))
        // Slot 4: Right channel
           ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] :
1'b0;
      else
        ac97_sdata_out <= 1'b0;

      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

   always @(negedge ac97_bit_clock) begin
      if ((bit_count >= 57) && (bit_count <= 76))
        // Slot 3: Left channel
        left_in_data <= { left_in_data[18:0], ac97_sdata_in };
      else if ((bit_count >= 77) && (bit_count <= 96))
        // Slot 4: Right channel
        right_in_data <= { right_in_data[18:0], ac97_sdata_in };
   end

endmodule

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

   input clock;
   input ready;
   output [7:0] command_address;
   output [15:0] command_data;
   output command_valid;
   input [4:0] volume;
   input [2:0] source;

   reg [23:0] command;
   reg command_valid;

   reg [3:0] state;
```

```
   initial begin
      command <= 4'h0;
      // synthesis attribute init of command is "0";
      command_valid <= 1'b0;
      // synthesis attribute init of command_valid is "0";
      state <= 16'h0000;
      // synthesis attribute init of state is "0000";
   end

   assign command_address = command[23:16];
   assign command_data = command[15:0];

   wire [4:0] vol;
   assign vol = 31-volume;  // convert to attenuation

   always @(posedge clock) begin
      if (ready) state <= state+1;

      case (state)
        4'h0: // Read ID
          begin
             command <= 24'h80_0000;
             command_valid <= 1'b1;
          end
        4'h1: // Read ID
          command <= 24'h80_0000;
        4'h3: // headphone volume
          command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
          command <= 24'h18_0808;
        4'h6: // Record source select
          command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
        4'h7: // Record gain = max
        command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
          command <= 24'h0E_8048;
        4'hA: // Set beep volume
          command <= 24'h0A_0000;
        4'hB: // PCM out bypass mix1
          command <= 24'h20_8000;
        default:
          command <= 24'h80_0000;
      endcase // case(state)
   end // always @ (posedge clock)
endmodule // ac97commands

//////////////////////////////////////////////////////////////////////
////////
//
// generate PCM data for 750hz sine wave (assuming f(ready) = 48khz)
//
//////////////////////////////////////////////////////////////////////
////////

module tone750hz (clock, ready, pcm_data);
   input clock;
   input ready;
```

```
output [19:0] pcm_data;

reg [8:0] index;
reg [19:0] pcm_data;

initial begin
   // synthesis attribute init of old_ready is "0";
   index <= 8'h00;
   // synthesis attribute init of index is "00";
   pcm_data <= 20'h00000;
   // synthesis attribute init of pcm_data is "00000";
end

always @(posedge clock) begin
   if (ready) index <= index+1;
end

// one cycle of a sinewave in 64 20-bit samples
always @(index) begin
   case (index[5:0])
     6'h00: pcm_data <= 20'h00000;
     6'h01: pcm_data <= 20'h0C8BD;
     6'h02: pcm_data <= 20'h18F8B;
     6'h03: pcm_data <= 20'h25280;
     6'h04: pcm_data <= 20'h30FBC;
     6'h05: pcm_data <= 20'h3C56B;
     6'h06: pcm_data <= 20'h471CE;
     6'h07: pcm_data <= 20'h5133C;
     6'h08: pcm_data <= 20'h5A827;
     6'h09: pcm_data <= 20'h62F20;
     6'h0A: pcm_data <= 20'h6A6D9;
     6'h0B: pcm_data <= 20'h70E2C;
     6'h0C: pcm_data <= 20'h7641A;
     6'h0D: pcm_data <= 20'h7A7D0;
     6'h0E: pcm_data <= 20'h7D8A5;
     6'h0F: pcm_data <= 20'h7F623;
     6'h10: pcm_data <= 20'h7FFFF;
     6'h11: pcm_data <= 20'h7F623;
     6'h12: pcm_data <= 20'h7D8A5;
     6'h13: pcm_data <= 20'h7A7D0;
     6'h14: pcm_data <= 20'h7641A;
     6'h15: pcm_data <= 20'h70E2C;
     6'h16: pcm_data <= 20'h6A6D9;
     6'h17: pcm_data <= 20'h62F20;
     6'h18: pcm_data <= 20'h5A827;
     6'h19: pcm_data <= 20'h5133C;
     6'h1A: pcm_data <= 20'h471CE;
     6'h1B: pcm_data <= 20'h3C56B;
     6'h1C: pcm_data <= 20'h30FBC;
     6'h1D: pcm_data <= 20'h25280;
     6'h1E: pcm_data <= 20'h18F8B;
     6'h1F: pcm_data <= 20'h0C8BD;
     6'h20: pcm_data <= 20'h00000;
     6'h21: pcm_data <= 20'hF3743;
     6'h22: pcm_data <= 20'hE7075;
     6'h23: pcm_data <= 20'hDAD80;
     6'h24: pcm_data <= 20'hCF044;
```

```
          6'h25: pcm_data <= 20'hC3A95;
          6'h26: pcm_data <= 20'hB8E32;
          6'h27: pcm_data <= 20'hAECC4;
          6'h28: pcm_data <= 20'hA57D9;
          6'h29: pcm_data <= 20'h9D0E0;
          6'h2A: pcm_data <= 20'h95927;
          6'h2B: pcm_data <= 20'h8F1D4;
          6'h2C: pcm_data <= 20'h89BE6;
          6'h2D: pcm_data <= 20'h85830;
          6'h2E: pcm_data <= 20'h8275B;
          6'h2F: pcm_data <= 20'h809DD;
          6'h30: pcm_data <= 20'h80000;
          6'h31: pcm_data <= 20'h809DD;
          6'h32: pcm_data <= 20'h8275B;
          6'h33: pcm_data <= 20'h85830;
          6'h34: pcm_data <= 20'h89BE6;
          6'h35: pcm_data <= 20'h8F1D4;
          6'h36: pcm_data <= 20'h95927;
          6'h37: pcm_data <= 20'h9D0E0;
          6'h38: pcm_data <= 20'hA57D9;
          6'h39: pcm_data <= 20'hAECC4;
          6'h3A: pcm_data <= 20'hB8E32;
          6'h3B: pcm_data <= 20'hC3A95;
          6'h3C: pcm_data <= 20'hCF044;
          6'h3D: pcm_data <= 20'hDAD80;
          6'h3E: pcm_data <= 20'hE7075;
          6'h3F: pcm_data <= 20'hF3743;
       endcase // case(index[5:0])
   end // always @ (index)
endmodule
```

Appendix CC – sound_gameover.v

```
module sound_gameover(clk, reset, sound_gameover_enable, ready,
interpolate ,to_ac97_data, gameover_addr, s1, s2, count, state);
   input clk;              // 27mhz system clock
   input reset;                    // 1 to reset to initial state
   input sound_gameover_enable;                //
   input ready;                    // 1 when AC97 data is available
   input interpolate;                       // 1 when interpolation
switch is activated
      output [7:0] to_ac97_data;      // 8-bit PCM data to headphone
      output [13:0] gameover_addr;                      // 14-bit
address of the output
      output s1;                                        // Output of
the Bram
      output s2;                                        // Output of
the Bram
      output [3:0] count;                          // 4-bit counter:
counts 0-7
                                                                    //
needs to be 4-bit because we are using two's complement
                                                                    //
format, which means that 3'b111 is actually -1, not 7
      output [2:0] state;                          // State of the FSM

      wire [7:0] to_ac97_data;
      reg signed [7:0] s1;    // s1, s2 are used to interpolate during
playback
   reg signed [7:0] s2;    // two's complement b/c used in math of the
interpolator
      reg signed [3:0] count = 4'b0;      // count to 0 -> 7; must be
signed for multiplication

      reg [13:0] gameover_addr = 14'd0;
      reg [13:0] gameover_max_addr = 14'd12000;  // highest memory
address with data
      wire [7:0] gameover_record_data;    // 8-bit data read from bram
      gameover_long_rom gameover_long_rom1 (gameover_addr, clk,
gameover_record_data);
      //module ram64x8(addr, clk, din,    dout, we);

      parameter STATE_RESET = 3'b000;   // Reset State
      parameter STATE_PLAYBACK = 3'b001;  // Record State
      reg [2:0] state = STATE_RESET;


      always @ (posedge clk) begin
            case (state)
                  // You need to go through the reset state between any
transition between the record
                  // and playback states.  The Reset state sets s1, s2,
write enable, and count to
                  // zero, always, to keep track of where we are,
max_addr is only reset on an actual
                  // press of the reset button.  After all of this, we
change to the state, based on
```

```
                // the user input.
                STATE_RESET:      begin
                                                count <= 0;
                                                gameover_addr[13:0] <=
14'd0;
                                                s1 <= 0;    // sets the
value of the sample for interpolation back to zero on reset
                                                s2 <= 0;
                                                state <=
(sound_gameover_enable)? STATE_PLAYBACK : state;
                                                end

                // This state records (stores the audio).  Stores
data for every 8 ready cycles
                STATE_PLAYBACK:
                begin
           if (ready)

                                begin
                                s2 <= gameover_record_data; // s2 takes
the current data
                                if (count == 0) // when count is zero, it
signifies the eighth sample
                                      begin              // which is
important b/c we only recorded every 8th ready cycle
                                      count <= count + 1; // go to next
sample
                                      end
                                else if (count == 1)
                                      begin
                                      gameover_addr <= gameover_addr + 1;
// go to next address
                                      count <= count +1;
                                      end

                                else if (count == 4'd5) // count reaches
5, so must return to 0
                                      begin
                                      count <= 0;  // resets the counter
for another 6 samples
                                      s1 <= s2; // during ready cycle, s1
will be the previous cycle's data
                                      //addr <= addr + 1;
                                      end
                                else if (reset)
                                      begin
                                      gameover_addr <= 14'd0; // loops
back through the audio, if there is no more data
                                      state <= STATE_RESET;
                                      end
                                else if (gameover_addr >
gameover_max_addr)
                                      begin
                                      gameover_addr <= 14'd0; // loops
back through the audio, if there is no more data
                                      state <= STATE_RESET;
                                      end
```

```
                              else
                                    begin
                                    count <= count + 1; // increment
count
                                    end
                           end
                  end

            endcase


   end

      // this is the interpolation option
      // if the interpolate switch is on, then the output for that
cycle will be
      // determined by (((8-count)*s1 + count*s2) >> 3), where s1 is
the previous data and s2
      // is the current data.  If interpolation is not requested, then
the current s2 data
      // is output.
      assign to_ac97_data = (state == STATE_RESET) ?  8'd0:
                           ((interpolate)&&(state ==
STATE_PLAYBACK))?(((6-count)*s1 + count*s2) >> 3) :
                           s2;
      endmodule
//
//module sound_gameover(clk, reset, sound_gameover_enable, ready,
interpolate ,to_ac97_data, gameover_addr, s1, s2, count, state);
//   input clk;              // 27mhz system clock
//   input reset;                    // 1 to reset to initial state
//   input sound_gameover_enable;                //
//   input ready;                    // 1 when AC97 data is available
//   input interpolate;                     // 1 when interpolation
switch is activated
//    output [7:0] to_ac97_data;      // 8-bit PCM data to headphone
//    output [13:0] gameover_addr;                      // 14-bit
address of the output
//    output s1;                                        // Output of
the Bram
//    output s2;                                        // Output of
the Bram
//    output [3:0] count;                         // 4-bit counter:
counts 0-7
//                                                                  //
needs to be 4-bit because we are using two's complement
//                                                                  //
format, which means that 3'b111 is actually -1, not 7
//    output [2:0] state;                         // State of the FSM
//
//    wire [7:0] to_ac97_data;
//    reg signed [7:0] s1;    // s1, s2 are used to interpolate during
playback
//   reg signed [7:0] s2;        // two's complement b/c used in math
of the interpolator
//    reg signed [3:0] count = 4'b0;      // count to 0 -> 7; must be
signed for multiplication
```

```
//
//    reg [13:0] gameover_addr = 14'd0;
//    reg [13:0] gameover_max_addr = 14'd12000;  // highest memory
address with data
//    wire [7:0] gameover_record_data;    // 8-bit data read from bram
//    gameover_long_rom gameover_long_rom1 (gameover_addr, clk,
gameover_record_data);
//    //module ram64x8(addr, clk, din,    dout, we);
//
//    parameter STATE_RESET = 3'b000;   // Reset State
//    parameter STATE_PLAYBACK = 3'b001;  // Record State
//    reg [2:0] state = STATE_RESET;
//
//    reg play_once = 1'b0;
//
//    always @ (posedge clk) begin
//          case (state)
//                // You need to go through the reset state between any
transition between the record
//                // and playback states.  The Reset state sets s1, s2,
write enable, and count to
//                // zero, always, to keep track of where we are,
max_addr is only reset on an actual
//                // press of the reset button.  After all of this, we
change to the state, based on
//                // the user input.
//                STATE_RESET:      begin
//                                                  if (reset) play_once <=
1'b0;
//                                                  count <= 0;
//                                                  gameover_addr[13:0] <=
14'd0;
//                                                  s1 <= 0;    // sets the
value of the sample for interpolation back to zero on reset
//                                                  s2 <= 0;
//                                                  state <=
((sound_gameover_enable)&&~(play_once))? STATE_PLAYBACK : state;
//                                                  end
//
//                // This state records (stores the audio).  Stores
data for every 8 ready cycles
//                STATE_PLAYBACK:
//                begin
//            if (ready)
//
//                              begin
//                              s2 <= gameover_record_data; // s2 takes
the current data
//                              if (count == 0) // when count is zero, it
signifies the eighth sample
//                                    begin              // which is
important b/c we only recorded every 8th ready cycle
//                                    count <= count + 1; // go to next
sample
//                                    play_once <= 1'b1;
//                                    end
//                              else if (count == 1)
```

```
//                                    begin
//                                    gameover_addr <= gameover_addr + 1;
// go to next address
//                                    count <= count +1;
//                                    end
//
//                               else if (count == 4'd5) // count reaches
5, so must return to 0
//                                    begin
//                                    count <= 0;  // resets the counter
for another 6 samples
//                                    s1 <= s2; // during ready cycle, s1
will be the previous cycle's data
//                                    //addr <= addr + 1;
//                                    end
//                               else if (reset)
//                                    begin
//                                    play_once <= 1'b0;
//                                    gameover_addr <= 14'd0; // loops
back through the audio, if there is no more data
//                                    state <= STATE_RESET;
//                                    end
//                               else if (gameover_addr >
gameover_max_addr)
//                                    begin
//                                    gameover_addr <= 14'd0; // loops
back through the audio, if there is no more data
//                                    state <= STATE_RESET;
//                                    end
//                               else
//                                    begin
//                                    count <= count + 1; // increment
count
//                                    end
//                          end
//                end
//
//          endcase
//
//
//   end
//
//    // this is the interpolation option
//    // if the interpolate switch is on, then the output for that
cycle will be
//    // determined by (((8-count)*s1 + count*s2) >> 3), where s1 is
the previous data and s2
//    // is the current data.  If interpolation is not requested, then
the current s2 data
//    // is output.
//    assign to_ac97_data = (state == STATE_RESET) ?  8'd0:
//                          ((interpolate)&&(state ==
STATE_PLAYBACK))?(((6-count)*s1 + count*s2) >> 3) :
//                          s2;
//    endmodule
```

Appendix DD – sound_laser_backup.v

```
module sound_laser(clk, reset, laser_enable, ready, interpolate
,to_ac97_data, addr, s1, s2, count, state);
   input clk;              // 27mhz system clock
   input reset;                    // 1 to reset to initial state
   input laser_enable;                // 1 for playback, 0 for record
   input ready;                    // 1 when AC97 data is available
   input interpolate;                        // 1 when interpolation
switch is activated
      output [7:0] to_ac97_data;      // 8-bit PCM data to headphone
      output [11:0] addr;                          // 12-bit address of
the output
      output s1;                                          // Output of
the Bram
      output s2;                                          // Output of
the Bram
      output [3:0] count;                          // 4-bit counter:
counts 0-7
                                                                      //
needs to be 4-bit because we are using two's complement
                                                                      //
format, which means that 3'b111 is actually -1, not 7
      output [2:0] state;                          // State of the FSM

      wire [7:0] to_ac97_data;
      reg signed [7:0] s1;    // s1, s2 are used to interpolate during
playback
   reg signed [7:0] s2;    // two's complement b/c used in math of the
interpolator
      reg signed [3:0] count = 4'b0;      // count to 0 -> 7; must be
signed for multiplication

      reg [11:0] laser_addr = 12'd0;
      reg [11:0] laser_max_addr = 12'd4026;  // highest memory address
with data
      wire [7:0] laser_record_data; // 8-bit data read from bram
      laser laser1 (laser_addr, clk, laser_record_data);
      //module ram64x8(addr, clk, din,    dout, we);

      parameter STATE_RESET = 3'b000;   // Reset State
      parameter STATE_PLAYBACK = 3'b001;  // Record State
      reg [2:0] state = STATE_RESET;

      always @ (posedge clk) begin
            case (state)
                  // You need to go through the reset state between any
transition between the record
                  // and playback states.  The Reset state sets s1, s2,
write enable, and count to
                  // zero, always, to keep track of where we are,
max_addr is only reset on an actual
```

```
                // press of the reset button.  After all of this, we
change to the state, based on
                // the user input.
                STATE_RESET:      begin
                                                count <= 0;
                                                laser_addr[11:0] <=
12'd0;
                                                s1 <= 0;    // sets the
value of the sample for interpolation back to zero on reset
                                                s2 <= 0;
                                                state <= laser_enable ?
STATE_PLAYBACK : state;
                                                end

                // This state records (stores the audio).  Stores
data for every 8 ready cycles
                STATE_PLAYBACK:
                begin
           if (ready)

                                begin
                                s2 <= laser_record_data; // s2 takes the
current data
                                if (count == 0) // when count is zero, it
signifies the eighth sample
                                      begin              // which is
important b/c we only recorded every 8th ready cycle
                                      count <= count + 1; // go to next
sample
                                      end
                                else if (count == 1)
                                      begin
                                      laser_addr <= laser_addr + 1;  //
go to next address
                                      count <= count +1;
                                      end

                                else if (count == 4'd5) // count reaches
5, so must return to 0
                                      begin
                                      count <= 0;  // resets the counter
for another 6 samples
                                      s1 <= s2; // during ready cycle, s1
will be the previous cycle's data
                                      //addr <= addr + 1;
                                      end
                                else if (reset||(laser_addr >=
laser_max_addr))
                                      begin
                                      state <= STATE_RESET;
                                      laser_addr <= 12'd0;    // loops
back through the audio, if there is no more data
                                      end
                                else if(laser_enable)
                begin
                laser_addr <= 0;
                count <=0;
```

```
                end
             else
                                begin
                                count <= count + 1; // increment
count
                                end
                          end
                end

          endcase


   end

      // this is the interpolation option
      // if the interpolate switch is on, then the output for that
cycle will be
      // determined by (((8-count)*s1 + count*s2) >> 3), where s1 is
the previous data and s2
      // is the current data.  If interpolation is not requested, then
the current s2 data
      // is output.
      assign to_ac97_data = (state == STATE_RESET) ?  8'sd0:
                        ((interpolate)&&(state ==
STATE_PLAYBACK))?(((6-count)*s1 + count*s2) >> 3) :
                        s2;
      endmodule
```

Appendix DD – synchronize.v

```
// pulse synchronizer
module synchronize(clk,in,out);
  parameter NSYNC = 2;  // number of sync flops.  must be >= 2
  parameter WIDTH = 8;

  input clk;
  input [WIDTH-1:0] in;
  output [WIDTH-1:0] out;

  reg [NSYNC-2:0] sync[WIDTH-1:0];
  reg out;

  integer i;
  always @ (posedge clk)
  begin
    for(i=0; i<WIDTH; i = i+1)
    begin
      {out[i],sync[i]} <= {sync[i][NSYNC-2:0],in[i]};
    end
  end
endmodule
```

Appendix EE – threshold.v

```
`timescale 1ns / 1ps
// Module: threshold (threshold filter)
// Author: Paul Yang
// Description: Sets the output value to 0 or 255 depending on whether
the

// input value is greater than a specified threshold value


module threshold(clk, reset, x_in, y_in, dv_in, input_val,
threshold_val,
                    x_out, y_out, dv_out, output_val, ram_addr_in,

                    ram_addr_out);
    // Inputs:
    // clk           -  the clock
    // reset         -  reset
    // x_in          -  the input x coordinate
    // y_in          -  the input y coordinate
    // dv_in         -  data valid for the given x,y coordinate
    // input_val     -  the pixel value for the given input pixel
    // threshold_val -  the threshold value to compare against

    // Outputs
    // x_out         -  the output x coordinate
    // y_out         -  the output y coordinate
    // dv_out        -  the data valid for the output x,y coord.
    // output_val    -  the value after thresholding

    // ram_addr_out  -  the associated RAM address of the sample


    input clk;
    input reset;
    input [10:0] x_in;
    input [9:0] y_in;
    input dv_in;
    input [7:0] input_val;
    input [7:0] threshold_val;
    input [18:0] ram_addr_in;

    output [10:0] x_out;
    output [9:0] y_out;
    output dv_out;
    output [7:0] output_val;
    output [18:0] ram_addr_out;


    reg [7:0] output_val;
    reg [10:0] x_out;
    reg [9:0] y_out;
    reg [18:0] ram_addr_out;
    reg dv_out;
```

```
    always @ (posedge clk)
    begin
      output_val <= (input_val > threshold_val) ? 8'hff  : 8'h0;
      // Delay the coodinate data to match the delay of the output_val
      x_out <= x_in;
      y_out <= y_in;
      dv_out <= dv_in;
      ram_addr_out <= ram_addr_in;
    end
endmodule
```

Appendix FF – timer.v

```
// timer

// The timer is responsible for timing the animation.
// When done counting to 16 new_frame signals,
// the expire signal will go high to signal to the duck_controller
// that it should move on to the next duck animation.


module timer(clk, reset, new_frame, expire);

   input clk; // 40 MHz
      input reset; // 1 for reset
      input new_frame; // 1 when new frame is needed for video
      output expire; // to duck_controller
                                    // 1 for duck_controller to
transition
                                    // to next animation

      reg[3:0] count; // 2^4 = 16

   always @ (posedge clk)
   begin
            // build a decrement counter
            if (reset)
                  count <= 4'd15; // reset count
            else if ((new_frame) && (count > 0))
                  count <= count - 1; // decrement count if count > 0
            else if ((new_frame) && (~(count > 0)))
                  count <= 4'd15; // reset count to TICKS if count <= 0
            else
                  count <= count; // keep count value
   end // end begin

      assign expire = (count == 0);

endmodule
```

Appendix GG – timer_spencer.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:15:51
// Design Name:
// Module Name:    timer
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module timer_spencer(reset, clk, start_timer, start_game, halt, value,
one_hz_enable, expired, count);
      input reset;
      input clk;
      input start_timer;
      input halt;
      input start_game;
      input [5:0] value;
      input one_hz_enable;
      output expired;
      output [5:0] count;

      //Timer - counts up, as opposed to down

      reg [5:0] count = 6'h0;
      always @ (posedge clk)
   begin
            if ((reset) | (start_timer) | (start_game))
                  begin
                  count <= 0;
                  end
            else if (halt)
                  count <= count;
            else if ((count < value) & (one_hz_enable))
                  count <= count + 1; // increment count if count <
value
            else if (~(count < value) & (one_hz_enable))
                  count <= count; // reset count to value if count >
value
            else
                  count <= count; // keep count value
      end
```

```
      wire expired;
      assign expired = (count == value); // expired will be high for
only one clock period

endmodule
```

Appendix HH – timer_strings.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////////
// Company:
// Engineer:
//
// Create Date:    23:03:44 12/05/06
// Design Name:
// Module Name:    score_strings
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////////

module timer_strings (clk, hcount, vcount,
       timer_val,
      timer_string_val);

      input clk; // 40 MHz
      //input [10:0] hcount; // pixel number on current line
      //input [9:0] vcount; // line number
      input [5:0] timer_val; //

      output [2*8-1:0] timer_string_val;



      reg [2*8-1:0] timer_string_val;


   always @(posedge clk) begin
      begin

                  case (timer_val)

                        6'd0: timer_string_val <= "45"; //
                        6'd1: timer_string_val <= "44"; //
                  6'd2: timer_string_val <= "43"; //
                        6'd3: timer_string_val <= "42"; //
                        6'd4: timer_string_val <= "41"; //
                        6'd5: timer_string_val <= "40"; //
                        6'd6: timer_string_val <= "39"; //
                        6'd7: timer_string_val <= "38"; //
                        6'd8: timer_string_val <= "37"; //
                        6'd9: timer_string_val <= "36"; //
```

```
                        6'd10: timer_string_val <= "35"; //
                        6'd11: timer_string_val <= "34"; //
                        6'd12: timer_string_val <= "33"; //
                        6'd13: timer_string_val <= "32"; //
                        6'd14: timer_string_val <= "31"; //
                        6'd15: timer_string_val <= "30"; //
                        6'd16: timer_string_val <= "29"; //
                        6'd17: timer_string_val <= "28"; //
                  6'd18: timer_string_val <= "27"; //
                        6'd19: timer_string_val <= "26"; //
                        6'd20: timer_string_val <= "25"; //
                        6'd21: timer_string_val <= "24"; //
                        6'd22: timer_string_val <= "23"; //
                        6'd23: timer_string_val <= "22"; //
                        6'd24: timer_string_val <= "21"; //
                        6'd25: timer_string_val <= "20"; //
                        6'd26: timer_string_val <= "19"; //
                        6'd27: timer_string_val <= "18"; //
                        6'd28: timer_string_val <= "17"; //
                        6'd29: timer_string_val <= "16"; //
                        6'd30: timer_string_val <= "15"; //
                        6'd31: timer_string_val <= "14"; //
                        6'd32: timer_string_val <= "13"; //
                        6'd33: timer_string_val <= "12"; //
                        6'd34: timer_string_val <= "11"; //
                        6'd35: timer_string_val <= "10"; //
                        6'd36: timer_string_val <= "09"; //
                        6'd37: timer_string_val <= "08"; //
                  6'd38: timer_string_val <= "07"; //
                        6'd39: timer_string_val <= "06"; //
                        6'd40: timer_string_val <= "05"; //
                        6'd41: timer_string_val <= "04"; //
                        6'd42: timer_string_val <= "03"; //
                        6'd43: timer_string_val <= "02"; //
                        6'd44: timer_string_val <= "01"; //
                        6'd45: timer_string_val <= "00"; //

//                      5'd46: timer_string_val <= "04"; //
//                      5'd47: timer_string_val <= "03"; //
//                      5'd48: timer_string_val <= "02"; //
//                      5'd49: timer_string_val <= "01"; //
//                      5'd50: timer_string_val <= "00"; //
//                      5'd51: timer_string_val <= "19"; //
//                      5'd52: timer_string_val <= "18"; //
//                      5'd53: timer_string_val <= "17"; //
//                      5'd54: timer_string_val <= "16"; //
//                      5'd55: timer_string_val <= "15"; //
//                      5'd56: timer_string_val <= "14"; //
//                      5'd57: timer_string_val <= "13"; //
//                5'd58: timer_string_val <= "12"; //
//                      5'd59: timer_string_val <= "11"; //
//                      5'd60: timer_string_val <= "10"; //
//                      5'd61: timer_string_val <= "09"; //
//                      5'd62: timer_string_val <= "08"; //
//                      5'd63: timer_string_val <= "07"; //
//
//
```

```
                        default: timer_string_val <= "00"; //
                    endcase
                end // end else
            end // end always

endmodule
```

Appendix II – title.v

```
/*****************************************************************
*********
*     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 1995-2004 Xilinx, Inc.
*
*     All rights reserved.
*
```

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

// You must compile the wrapper file title.v when simulating
// the core, title. 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 title(
      addr,
      clk,
      dout);


input [12 : 0] addr;
input clk;
output [3 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            13,   // c_addr_width
            "0",  // c_default_data
            5000, // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "title.mif",      // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            4,    // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
```

```
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of title is "true"

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

endmodule
```

Appendix JJ – velocity.v

```
// velocity

// the velocity module determines the speed at which the
// duck flies. the first 3 ducks will fly at slow speed,
// second 3 ducks will fly at medium speed, and the last
// 2 ducks will fly at high speed.

module velocity(clk, reset, speed_enable, duck_count, duck_speed);

      input clk; // 40 MHz
      input reset; // 1 to reset game
      input speed_enable; // 1 when a duck appears
      output [2:0] duck_count;
      output[3:0] duck_speed; // speed in pixel/frame

      // speed parameters in pixels/tick
      parameter LOW = 4'd2; // speed for first 3 ducks
      parameter MED = 4'd4; // speed for second 3 ducks
      parameter HI = 4'd6; // speed for last 2 ducks

      reg[3:0] duck_speed;
      reg[2:0] duck_count; // counts the number of ducks that
                                                  // have already
appeared in the game

      always @ (posedge clk)
      begin

            if (reset)
                  duck_count <= 3'd0; // reset duck count
            else begin
                  if (speed_enable) begin

                  // logic to determine duck speed
                        if (duck_count < 3'd3)
                              duck_speed <= LOW;
                        else if ((duck_count >= 3'd3) && (duck_count <
3'd6))
                              duck_speed <= MED;
                        else if (duck_count <= 3'd7)
                              duck_speed <= HI;
                        else
                              duck_speed <= 0; // for debugging
purposes

                        duck_count <= duck_count + 1;
                        // increment duck_count

                        end // end speed_enable

                  else duck_speed <= duck_speed;

                  end // end reset else
      end // end always
```

```
endmodule
```

Appendix KK – video_controller.v

```
// video_controller
// tiff chen

// The video controller take all the pixel information from the
// duck, cloud, scorekeeper, and bush and prioritizes the pixel
// information so that the duck is behind the bushes and clouds
// and that the scorekeeper is on top of everything else. The video
// controller also controls the transparency of the backgrounds
// of the images from the duck, cloud, and bushes.
// the video controller is also a simple fsm that transitions between
the
// title image of the game to the actual game.

module video_controller(
              //inputs
              vclock,reset,
              hcount,vcount,hsync,vsync,blank,
              start_game, halt, win, gameover,
              duck_pixel, duck_x, duck_y,
              cloud_pixel1, cloud_x1, cloud_y1,
              cloud_pixel2, cloud_x2, cloud_y2,
              cloud_pixel3, cloud_x3, cloud_y3,
              bush_pixel1, bush_x1, bush_y1,
              bush_pixel2, bush_x2, bush_y2,
              bush_pixel3, bush_x3, bush_y3,
              bush_pixel8, bush_x8, bush_y8,
              bush_pixel9, bush_x9, bush_y9,
              bush_pixel10, bush_x10, bush_y10,
              bush_pixel11, bush_x11, bush_y11,
              bush_pixel12, bush_x12, bush_y12,

              up,down,left,right,blob_enb,

        // transform blob
        xform_blob_data,
             // spencer's inputs

              sk_pixel, sk_x, sk_y,
              hard_score_pixel, hs_x, hs_y,
              score_pixel, score_x, score_y,
              hard_timer_thirty_pixel, hard_timer_thirty_x,
              hard_timer_thirty_y,hard_timer_thirty_x_length,
              timer_thirty_pixel, timer_thirty_x, timer_thirty_y,
              timer_thirty_x_length,
              bush_pixel4, bush_x4, bush_y4,
              bush_pixel5, bush_x5, bush_y5,
              bush_pixel6, bush_x6, bush_y6,
              bush_pixel7, bush_x7, bush_y7,
        title_pixel1, title_x1, title_y1, title_x_length1,
              title_pixel2, title_x2, title_y2, title_x_length2,
              title_pixel3, title_x3, title_y3, title_x_length3,
              title_pixel4, title_x4, title_y4, title_x_length4,
```

```
            title_pixel5, title_x5, title_y5, title_x_length5,

            //outputs
            bush_height, bush_width,
            cloud_height, cloud_width,
            sky_height, sky_width,
            duck_height, duck_width,
            cloud_speed1, cloud_speed2, cloud_speed3,
            cloud_enable1, cloud_enable2, cloud_enable3,
            dhsync,dvsync,dblank,game_pixel,gamestate,
            blob_x, blob_y);

input vclock;  // 40MHz clock
input reset;         // 1 to initialize module
input [10:0] hcount; // horizontal index of current pixel (0..799)
input [9:0] vcount; // vertical index of current pixel (0..599)
input hsync;         // XVGA horizontal sync signal (active low)
input vsync;         // XVGA vertical sync signal (active low)
input blank;         // XVGA blanking (1 means output black pixel)
   input start_game;
   input halt;
   input win;
   input gameover;

   input up;
   input down;
   input left;
   input right;
   input blob_enb;

   // duck
   input [3:0] duck_pixel; // duck pixels
   input [10:0] duck_x; // duck x-coordinate
   input [9:0] duck_y; // duck y-coordinate

   // first cloud
   input [3:0] cloud_pixel1; // cloud1 pixels
   input [10:0] cloud_x1; // cloud1 x-coordinate
   input [9:0] cloud_y1; // cloud1 y-coordinate

   // second cloud
   input [3:0] cloud_pixel2; // cloud2 pixels
   input [10:0] cloud_x2; // cloud2 x-coordinate
   input [9:0] cloud_y2; // cloud2 y-coordinate

   // third cloud
   input [3:0] cloud_pixel3; // cloud3 pixels
   input [10:0] cloud_x3; // cloud3 x-coordinate
   input [9:0] cloud_y3; // cloud3 y-coordinate

   // first stationary bush
   input [3:0] bush_pixel1; // bush1 pixels
   input [10:0] bush_x1; // bush1 x-coordinate
   input [9:0] bush_y1; // bush1 y-coordinate

   // second stationary bush
   input [3:0] bush_pixel2; // bush2 pixels
```

```
input [10:0] bush_x2; // bush2 x-coordinate
input [9:0] bush_y2; // bush2 y-coordinate

// third stationary bush
input [3:0] bush_pixel3; // bush3 pixels
input [10:0] bush_x3; // bush3 x-coordinate
input [9:0] bush_y3; // bush3 y-coordinate

// 4th stationary bush
input [3:0] bush_pixel8; // bush3 pixels
input [10:0] bush_x8; // bush3 x-coordinate
input [9:0] bush_y8; // bush3 y-coordinate

// 5th stationary bush
input [3:0] bush_pixel9; // bush3 pixels
input [10:0] bush_x9; // bush3 x-coordinate
input [9:0] bush_y9; // bush3 y-coordinate

// 6th stationary bush
input [3:0] bush_pixel10; // bush3 pixels
input [10:0] bush_x10; // bush3 x-coordinate
input [9:0] bush_y10; // bush3 y-coordinate

// 7th stationary bush
input [3:0] bush_pixel11; // bush3 pixels
input [10:0] bush_x11; // bush3 x-coordinate
input [9:0] bush_y11; // bush3 y-coordinate

// 8th stationary bush
input [3:0] bush_pixel12; // bush3 pixels
input [10:0] bush_x12; // bush3 x-coordinate
input [9:0] bush_y12; // bush3 y-coordinate

// scorekeeping
input [3:0] sk_pixel;
input [10:0] sk_x; //
input [9:0] sk_y; //

input [3:0] hard_score_pixel;
input [10:0] hs_x; //
input [9:0] hs_y; //

input [3:0] score_pixel;
input [10:0] score_x; //
input [9:0] score_y; //

input [3:0] hard_timer_thirty_pixel;
input [10:0] hard_timer_thirty_x;
input [9:0] hard_timer_thirty_y;
input [10:0] hard_timer_thirty_x_length;

input [3:0] timer_thirty_pixel;
input [10:0] timer_thirty_x;
input [9:0] timer_thirty_y;
input [10:0] timer_thirty_x_length;

// spencer's bushes
```

```
   input [3:0] bush_pixel4;
   input [10:0] bush_x4;
   input [9:0] bush_y4;

input [3:0] bush_pixel5;
   input [10:0] bush_x5;
   input [9:0] bush_y5;

   input [3:0] bush_pixel6;
   input [10:0] bush_x6;
   input [9:0] bush_y6;

   input [3:0] bush_pixel7;
   input [10:0] bush_x7;
   input [9:0] bush_y7;

input [3:0] title_pixel1;
   input [10:0] title_x1;
   input [9:0] title_y1;
   input [10:0] title_x_length1;

   input [3:0] title_pixel2;
   input [10:0] title_x2;
   input [9:0] title_y2;
   input [10:0] title_x_length2;

   input [3:0] title_pixel3;
   input [10:0] title_x3;
   input [9:0] title_y3;
   input [10:0] title_x_length3;

   input [3:0] title_pixel4;
   input [10:0] title_x4;
   input [9:0] title_y4;
   input [10:0] title_x_length4;

   input [3:0] title_pixel5;
   input [10:0] title_x5;
   input [9:0] title_y5;
   input [10:0] title_x_length5;

input [23:0] xform_blob_data;
   output [9:0] bush_height;
   output [10:0] bush_width;
   output [9:0] cloud_height;
   output [10:0] cloud_width;
   output [9:0] duck_height;
   output [10:0] duck_width;
   output [9:0] sky_height;
   output [10:0] sky_width;

   output[3:0] cloud_speed1; // cloud speed
   output[3:0] cloud_speed2; // cloud speed
   output[3:0] cloud_speed3; // cloud speed

   output cloud_enable1;
   output cloud_enable2;
```

```
   output cloud_enable3;

output dhsync; // duck hunt game's horizontal sync
output dvsync; // duck hunt game's vertical sync
output dblank; // duck hunt game's blanking
output [3:0] game_pixel;   // duck hunt game's pixel

   output [1:0] gamestate;
   output [10:0] blob_x;
   output [9:0] blob_y;


   // bottom-most layer: blue sky
   assign sky_width = 11'd800;
   assign sky_height = 10'd500;
   parameter SKY_COLOR = 4'd14;  // sky color is cyan

   // stationary bushes
   assign bush_height = 10'd50;
   assign bush_width = 11'd133;

   //    clouds
   assign cloud_height = 10'd40;
   assign cloud_width = 11'd108;

   // ducks
   assign duck_height = 10'd110;
   assign duck_width = 11'd110;

   // title
   parameter TITLE_HEIGHT = 50;
   parameter TITLE_WIDTH = 100;
   parameter TITLE_X = 70;
   parameter TITLE_Y = 500;

   // gameover
   parameter WINLOSE_WIDTH = 100;
   parameter WINLOSE_HEIGHT = 50;
   parameter WINLOSE_X = 200;
   parameter WINLOSE_Y = 100;

   // start frame
   parameter TITLE_X_START = 0;
   parameter TITLE_Y_START = 0;
   parameter DUCK_X_START = 50;
   parameter DUCK_Y_START = 400;

   parameter text_height = 10'd24;

   // game FSM states
   parameter STARTMODE = 2'd0;
   parameter GAMEMODE = 2'd1;

   reg [1:0] gamestate;
   reg [3:0] game_pixel;
   reg [12:0] title_add;
   reg [13:0] duck_add1;
```

```
      reg [12:0] winlose_add;
      reg prev_start_game;
      reg [10:0] blob_x = 11'd1;
      reg [9:0] blob_y = 10'd1;

      // duck hunt title image
      wire [3:0] title_pixel;
      wire [3:0] duckdown_east_pixel1;
      wire [3:0] gameover_pixel;
      wire [3:0] youwin_pixel;
      title title1(.addr(title_add), .clk(vclock), .dout(title_pixel));
      duckdown_east
duckdown_east_title(.addr(duck_add1),.clk(vclock),.dout(duckdown_east_p
ixel1));
      gameoverimage
gameoverimage1(.addr(winlose_add),.clk(vclock),.dout(gameover_pixel));
      youwin
youwin1(.addr(winlose_add),.clk(vclock),.dout(youwin_pixel));


      always @ (posedge vclock)
      begin

            if (reset) begin
                  title_add <= 13'd0;
                  duck_add1 <= 14'd0;
                  winlose_add <= 13'd0;
                  gamestate <= STARTMODE;
                  end // end reset

            else begin

                  // state transitions
                  if (gamestate == STARTMODE)
                        gamestate <= (start_game) ? GAMEMODE :
STARTMODE;
                  //    gamestate <= STARTMODE;
                  else if (gamestate == GAMEMODE)
                        gamestate <= GAMEMODE;
                  else
                        gamestate <= gamestate;

                  // pixel outputs
                  if (gamestate == STARTMODE) begin

                        if ((TITLE_X_START <= hcount) && (hcount <
TITLE_X_START + TITLE_WIDTH*8) &&
                              (TITLE_Y_START <= vcount) && (vcount <
TITLE_Y_START + TITLE_HEIGHT*8))
                              title_add <= ((vcount -
TITLE_Y_START)>>3) * TITLE_WIDTH + ((hcount - TITLE_X_START)>>3);
                        if  (
                              (DUCK_X_START <= hcount) && (hcount <
DUCK_X_START + duck_width*2) &&
                            (DUCK_Y_START <= vcount) && (vcount <
DUCK_Y_START + duck_height*2))
```

```
                              duck_add1 <= ((vcount - DUCK_Y_START)>>1)
* duck_width + ((hcount - DUCK_X_START)>>1);

                           if ((TITLE_X_START<= hcount) && (hcount <
TITLE_X_START + TITLE_WIDTH*8 + 1) &&
                               (TITLE_Y_START <= vcount) && (vcount <
TITLE_Y_START + TITLE_HEIGHT*8 + 1))
                                    game_pixel <= title_pixel;
                           else if  ((duckdown_east_pixel1 != 4'd13) &&
                                    (DUCK_X_START + 1 <= hcount) &&
(hcount < DUCK_X_START + duck_width*2 + 1) &&
                              (DUCK_Y_START <= vcount) && (vcount <
DUCK_Y_START + duck_height*2))
                                    game_pixel <= duckdown_east_pixel1;

                           /// spencer's text
                           else if ((vcount > title_y1) &&
                                 (vcount <= (title_y1+text_height)) &&
                                 (hcount > (title_x1 + 1))     &&
                                 (hcount <= (title_x1+title_x_length1+ 1))
&&
                                 (title_pixel1 == 4'd15)
                                   )
                                 game_pixel[3:0] <= 4'd4;

                           else if ((vcount > title_y2) &&
                                 (vcount <= (title_y2+text_height)) &&
                                 (hcount > (title_x2+ 1))      &&
                                 (hcount <= (title_x2+title_x_length2+ 1))
&&
                                 (title_pixel2 == 4'd15)
                                   )
                                 game_pixel[3:0] <= 4'd4;

                           else if ((vcount > title_y3) &&
                                 (vcount <= (title_y3+text_height)) &&
                                 (hcount > (title_x3+ 1))      &&
                                 (hcount <= (title_x3+title_x_length3+ 1))
&&
                                 (title_pixel3 == 4'd15)
                                   )
                                 game_pixel[3:0] <= 4'd4;

                           else if ((vcount > title_y4) &&
                                 (vcount <= (title_y4+text_height)) &&
                                 (hcount > (title_x4+ 1))      &&
                                 (hcount <= (title_x4+title_x_length4+ 1))
&&
                                 (title_pixel4 == 4'd15)
                                   )
                                 game_pixel[3:0] <= 4'd4;

                           else if ((vcount > title_y5) &&
                                 (vcount <= (title_y5+text_height)) &&
                                 (hcount > (title_x5+ 1))      &&
                                 (hcount <= (title_x5+title_x_length5+ 1))
&&
```

```
                              (title_pixel5 == 4'd15)
                                )
                              game_pixel[3:0] <= 4'd4;


                  /// end spencer's text

                        else game_pixel <= SKY_COLOR;

                        end // STARTMODE

                  else if (gamestate == GAMEMODE) begin

                  blob_x = (left) ? blob_x - 11'd20 :
                                    (right) ? blob_x + 11'd20 :
                                    blob_x;
                        blob_y = (up) ? blob_y - 10'd20 :
                                    (down) ? blob_y + 10'd20 :
                                    blob_y;

//                      if ((blob_enb) && (blob_x + 1 <= hcount) &&
(hcount < blob_x + 10 + 1) &&
//                             (blob_y <= vcount) && (vcount < blob_y +
10))
//                           game_pixel <= 4'd2;
            if(xform_blob_data != 24'h0)
               game_pixel <= 4'd9;

                        else if ((hcount < sky_width) && (vcount <
sky_height))      begin

                                    if ((halt) &&
                                          (WINLOSE_X <= hcount) &&
                                          (hcount < WINLOSE_X +
WINLOSE_WIDTH*4 ) &&
                                          (WINLOSE_Y <= vcount ) &&
                                          (vcount < WINLOSE_Y +
WINLOSE_HEIGHT*4))
                                          winlose_add <= ((vcount -
WINLOSE_Y) >> 2) * WINLOSE_WIDTH

      + ((hcount - WINLOSE_X) >> 2);

                                    if ((halt) &&
                                          (WINLOSE_X + 1 <= hcount) &&
                                          (hcount < WINLOSE_X +
WINLOSE_WIDTH*4 + 1) &&
                                          (WINLOSE_Y <= vcount ) &&
                                          (vcount < WINLOSE_Y +
WINLOSE_HEIGHT*4))
                                          game_pixel <= (win)?
youwin_pixel :

(gameover)? gameover_pixel : 4'd15;

                                    // cloud1 pixels
                                    else if (
```

```
                                        (~(cloud_pixel1 == 4'd13)) &&
                                        (cloud_x1 + 1 <= hcount) &&
                                        (hcount  < cloud_x1 +
cloud_width + 1) &&
                                        (cloud_y1 <= vcount) &&
                                        (vcount < cloud_y1 +
cloud_height))
                                        game_pixel <= cloud_pixel1;


                                  // cloud2 pixels
                                  else if (
                                        (~(cloud_pixel2 == 4'd13)) &&
                                        (cloud_x2 + 1 <= hcount) &&
                                        (hcount < cloud_x2 +
cloud_width + 1) &&
                                        (cloud_y2 <= vcount) &&
                                        (vcount < cloud_y2 +
cloud_height))
                                        game_pixel <= cloud_pixel2;


                                  // cloud3 pixels
                                  else if (
                                        (~(cloud_pixel3 == 4'd13)) &&
                                        (cloud_x3 + 1 <= hcount) &&
                                        (hcount < cloud_x3 +
cloud_width + 1 ) &&
                                        (cloud_y3 <= vcount) &&
                                        (vcount < cloud_y3 +
cloud_height))
                                        game_pixel <= cloud_pixel3;


                                  // bush1 pixels
                                  else if ((~(bush_pixel1 == 4'd13))
&&
                                        (bush_x1 + 1 <= hcount) &&
                                        (hcount < bush_x1 +
bush_width + 1) &&
                                        (bush_y1 <= vcount) &&
                                        (vcount < bush_y1 +
bush_height))
                                        game_pixel <= bush_pixel1;


                                  // bush2 pixels
                                  else if ((~(bush_pixel2 == 4'd13))
&&
                                        (bush_x2 + 1 <= hcount) &&
                                        (hcount < bush_x2 +
bush_width + 1) &&
                                        (bush_y2 <= vcount) &&
                                        (vcount < bush_y2 +
bush_height))
                                        game_pixel <= bush_pixel2;
```

```
                                    // bush3 pixels
                                    else if ((~(bush_pixel3 == 4'd13))
&&
                                          (bush_x3 + 1 <= hcount) &&
                                          (hcount < bush_x3 +
bush_width + 1) &&
                                          (bush_y3 <= vcount) &&
                                          (vcount < bush_y3 +
bush_height))
                                          game_pixel <= bush_pixel3;

                                    // bush8 pixels
                                    else if ((~(bush_pixel8 == 4'd13))
&&
                                          (bush_x8 + 1 <= hcount) &&
                                          (hcount < bush_x8 +
bush_width + 1) &&
                                          (bush_y8 <= vcount) &&
                                          (vcount < bush_y8 +
bush_height))
                                          game_pixel <= bush_pixel8;

                                    // bush9 pixels
                                    else if ((~(bush_pixel9 == 4'd13))
&&
                                          (bush_x9 + 1 <= hcount) &&
                                          (hcount < bush_x9 +
bush_width + 1) &&
                                          (bush_y9 <= vcount) &&
                                          (vcount < bush_y9 +
bush_height))
                                          game_pixel <= bush_pixel9;

                                    // bush10 pixels
                                    else if ((~(bush_pixel10 == 4'd13))
&&
                                          (bush_x10 + 1 <= hcount) &&
                                          (hcount < bush_x10 +
bush_width + 1) &&
                                          (bush_y10 <= vcount) &&
                                          (vcount < bush_y10 +
bush_height))
                                          game_pixel <= bush_pixel10;

                                    // bush11 pixels
                                    else if ((~(bush_pixel11 == 4'd13))
&&
                                          (bush_x11 + 1 <= hcount) &&
(hcount <bush_x11 + bush_width + 1) &&
                                          (bush_y11 <= vcount) &&
(vcount < bush_y11 + bush_height))
                                          game_pixel <= bush_pixel11;

                                    // bush12 pixels
                                    else if ((~(bush_pixel11 == 4'd13))
&&
```

```
bush_width + 1) &&

bush_height))


here

&&

(bush_y4+bush_height)) &&
      &&
+ 1))

bush_pixel4[3:0];

&&

(bush_y5+bush_height)) &&
      &&
+ 1))

bush_pixel5[3:0];

&&

(bush_y6+bush_height)) &&
      &&
+ 1))

bush_pixel6[3:0];

&&

(bush_y7+bush_height)) &&
      (bush_x12 + 1 <= hcount) &&
      (hcount < bush_x12 +

      (bush_y12 <= vcount) &&
      (vcount < bush_y12 +

      game_pixel <= bush_pixel12;


// spencer's animated bushes start

else if ((~(bush_pixel4 == 4'd13))

      (vcount >= bush_y4 ) &&
      (vcount <

      (hcount >= (bush_x4 + 1))

      (hcount < (bush_x4+bush_width

      )
      game_pixel[3:0] <=

else if ((~(bush_pixel5 == 4'd13))

      (vcount >= bush_y5) &&
      (vcount <

      (hcount >= (bush_x5 + 1))

      (hcount < (bush_x5+bush_width

      )
      game_pixel[3:0] <=

else if ((~(bush_pixel6 == 4'd13))

      (vcount >= bush_y6) &&
      (vcount <

      (hcount >= (bush_x6 + 1))

      (hcount < (bush_x6+bush_width

      )
      game_pixel[3:0] <=

else if ((~(bush_pixel7 == 4'd13))

      (vcount >= bush_y7) &&
      (vcount <
```

```
                                          (hcount >= (bush_x7 + 1))
      &&
                                          (hcount < (bush_x7+bush_width
+ 1))
                                          )
                                          game_pixel[3:0] <=
bush_pixel7[3:0];
                                    // spencer's animated bushes end
here

                                    // duck pixels
                                    else if (
                                          (~(duck_pixel == 4'd13)) &&
                                          (duck_x + 1 <= hcount) &&
                                          (hcount < duck_x + duck_width
+ 1) &&
                                          (duck_y <= vcount) && (
                                          vcount < duck_y +
duck_height))
                                          game_pixel <= duck_pixel;

                                    else game_pixel <= SKY_COLOR;

                                    end // end sky begin

                              else
                                    begin// scorekeeping

                                          // title image
                                          if ((TITLE_X + 1 <= hcount) &&
                                          (hcount < TITLE_X +
TITLE_WIDTH*2 + 1) &&
                                          (TITLE_Y <= vcount) &&
(vcount < TITLE_Y + TITLE_HEIGHT*2))
                                          game_pixel <= title_pixel;
                                    if ((TITLE_X <= hcount) &&
                                           (hcount < TITLE_X +
TITLE_WIDTH*2) &&
                                           (TITLE_Y <= vcount) &&
                                           (vcount < TITLE_Y +
TITLE_HEIGHT*2))
                                          title_add <= ((vcount -
TITLE_Y)>>1) * TITLE_WIDTH
                                                + ((hcount -
TITLE_X)>>1);

                                    else if ((~(hard_score_pixel[3:0]
== 4'd0)) &&
                                          (vcount >= hs_y) &&
                                          (vcount < (hs_y+text_height))
&&
                                          (hcount >= (hs_x + 1))  &&
                                          (hcount < (hs_x+11'd96 + 1))
                                          )
                                          game_pixel[3:0] <=
(hard_score_pixel[3:0] == 4'd15 ) ?
```

```
                                                4'd0 :
hard_score_pixel[3:0];

                                        else if ((~(score_pixel[3:0] ==
4'd0)) &&
                                              (vcount >= score_y) &&
                                              (vcount <
(score_y+text_height)) &&
                                              (hcount >= (score_x + 1))
      &&
                                              (hcount < (score_x+11'd40 +
1))
                                              )
                                              game_pixel[3:0] <=
(score_pixel[3:0] == 4'd15 ) ?
                                                    4'd0 :
score_pixel[3:0];

                                        else if
((~(hard_timer_thirty_pixel[3:0] == 4'd0))      &&
                                              (vcount >=
hard_timer_thirty_y) &&
                                              (vcount <
(hard_timer_thirty_y+text_height)) &&
                                              (hcount >=
(hard_timer_thirty_x + 1))    &&
                                              (hcount <
(hard_timer_thirty_x+hard_timer_thirty_x_length + 1))
                                              )
                                              game_pixel[3:0] <=
(hard_timer_thirty_pixel[3:0] == 4'd15 ) ?

      4'd0 : hard_timer_thirty_pixel[3:0];

                                        else if ((~(timer_thirty_pixel[3:0]
== 4'd0)) &&
                                              (vcount >= timer_thirty_y) &&
                                              (vcount <
(timer_thirty_y+text_height)) &&
                                              (hcount >= (timer_thirty_x +
1))   &&
                                              (hcount <
(timer_thirty_x+timer_thirty_x_length + 1))
                                              )
                                              game_pixel[3:0] <=
(timer_thirty_pixel[3:0] == 4'd15 ) ?

      4'd0 : timer_thirty_pixel[3:0];

                                        else game_pixel <= 4'd1; // ground
color
                           end// end scorekeeping
                        end // end gamestate
         end // end reset else
      end // end begin

   assign dhsync = hsync;
```

```
   assign dvsync = vsync;
   assign dblank = blank;

      assign cloud_enable1 = 1'b1;
      assign cloud_enable2 = 1'b1;
      assign cloud_enable3 = 1'b1;

      assign cloud_speed1 = 4'b0001;
      assign cloud_speed2 = 4'b0001;
      assign cloud_speed3 = 4'b0010;

endmodule
```

Appendix LL – video_decoder.v

```
//
// 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 stream.
           // whenever the synchronization code is seen, go back to the
sync_state before
           // 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_y0;
              SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ?
SYNC_1 : SAV_f1_cr1;
              SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ?
SYNC_1 : SAV_f1_y1;
              SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ?
SYNC_1 : SAV_f1_cb0;

              SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ?
SYNC_1 : SAV_f2_y0;
              SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ?
SYNC_1 : SAV_f2_cr1;
              SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ?
SYNC_1 : SAV_f2_y1;
              SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ?
SYNC_1 : SAV_f2_cb0;

              // 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_SAMPLING, `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, `GPO_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, `DEFAULT_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_CONTROL, `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_BIRIGHTNESS_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
```

Appendix MM – video_priority_encoder.v

```
`timescale 1ns / 1ps
// Module: pixel_counter (pixel_counter)
// Author: Paul Yang
// Description: The video priority encoder is a pipelined module that
merges

// various image layers. Each layer is a stage in a pipeline. and

// overwrites (unless the current layer color is black the data from
the

// previous layer. Due to the pipelined nature of the module, the
merged

// output is delayed by 5 clock cyles

module video_priority_encoder(clk, reset, video_data,
crosshair_overlay,
                              xform_overlay, calib_overlay,
background_data,
                              vga_out);



    // Inputs:

    // clk                 -  the clock at 40MHz

    // reset               -  the reset signal

    // background_data     -  the background pixel data (first layer)

    // crosshair_overlay   -  the crosshair data (second layer)

    // xform_overlay       -  the blob data centered at the transformed
(x,y)

    //                        coordinate

    // calib_overlay       -  the blob data from the coefficent
generator



    // Outputs:

    // vga_out             -  the final mereged pixel value
    input clk;
    input reset;

    input [23:0] background_data;
    input [23:0] video_data;
    input [23:0] crosshair_overlay;
    input [23:0] xform_overlay;
```

```
    input [23:0] calib_overlay;
    output [23:0] vga_out;

    // If a pixel in a layer is the transparent color, it keeps the
data from
    // the previous layer
    parameter TRANSPARENT_COLOR = 0;
    parameter LAYERS = 5;

    // The layer is the stage in the pipeline
    reg [23:0] layer[LAYERS-1:0];

    // Appropriately delay each layer (except for the background)

    // Later stages need to be delayed more as they encouner less
stages

    // of the pipeline
    wire [23:0] video_data_delayed;
    wire [23:0] crosshair_overlay_delayed;
    wire [23:0] xform_overlay_delayed;
    wire [23:0] calib_overlay_delayed;

    delay1 d1(clk, video_data, video_data_delayed);
    defparam d1.WIDTH = 24;
    delay d2(clk, crosshair_overlay, crosshair_overlay_delayed);
    defparam d2.DELAY = 2;
    defparam d2.WIDTH = 24;
    delay d3(clk, xform_overlay, xform_overlay_delayed);
    defparam d3.DELAY = 3;
    defparam d3.WIDTH = 24;
    delay d4(clk, calib_overlay, calib_overlay_delayed);
    defparam d4.DELAY = 4;
    defparam d4.WIDTH = 24;

    always@(posedge clk)
    begin

      // Shift data down the pipeline
      layer[0] <= background_data;
      layer[1] <= (video_data_delayed != TRANSPARENT_COLOR) ?
                  video_data_delayed : layer[0];
      layer[2] <= (crosshair_overlay_delayed != TRANSPARENT_COLOR) ?
                   crosshair_overlay_delayed : layer[1];
      layer[3] <= (xform_overlay_delayed != TRANSPARENT_COLOR) ?
                   xform_overlay_delayed : layer[2];
      layer[4] <= (calib_overlay_delayed != TRANSPARENT_COLOR) ?
                   calib_overlay_delayed : layer[3];
    end

    // Assign the output to be the last layer
    assign vga_out = layer[LAYERS-1];
endmodule
```

Appendix NN – vram.v

```
`timescale 1ns / 1ps
// Module: vram (video_ram)
// Author: Paul Yang
// Description: The video RAM module uses the ZBT SRAM on the labkit to

// emulates a dual port RAM. The module clocks the ZBT RAM at twice the

// pixel clock so that a pixel can be read and written within a single

// pixel clock cycle.


// vram - uses ZBT ram to mimic dual port video ram
module vram(clk, ram0_clk, reset, hcount, vcount, write_addr,
write_data, we,
            wr, read_data, ram_we_b, ram_address,
            ram_data, ram_cen_b);
    // Inputs:
    // clk           -  the clock at 80Mhz
    // reset         -  reset
    // hcount        -  the horizontal count

    // vcount        -  the vertical count

    // write_addr    -  the address to write to

    // write_data    -  the data to write to the given write_addr
    // we            -  whether or not data at the next clock edge
should
    //                  be written to the given write_address. value
has
    //                  no effect unless wr is true

    // Outputs:
    // wr            -  output that signals when the data will be read
    //                  from data_in. if asserted, data will be read in
the
    //                  next clock edge and written to the given
write_addr
    // read_data     -  value stored at the pixel location at hcount,
vcount
    //                  that was clocked in 2 clock cycles (@40Mhz) ago


    // Labkit ZBT related connections
    // Outputs:
    // ram_we_b      -  ZBT write enable
    // ram_address   -  ZBT address
    // ram_data      -  ZBT read/write port
    // ram_cen_b     -  ZBT enable


    input clk;
```

```
    input reset;
    input [10:0] hcount;
    input [9:0] vcount;
    input [18:0] write_addr;
    input [35:0] write_data;
    input we;

    output wr;
    output [35:0] read_data;

    // zbt_connrectsions
    output ram_we_b;
    output [18:0] ram_address;
    inout [35:0] ram_data;
    output ram_cen_b;
    output ram0_clk;

    // Wire up the ZBT ram
       reg [18:0] zbt_addr;
       reg zbt_we;
       wire [35:0] zbt_read_data;
       reg [35:0] zbt_write_data;

       zbt_6111 zbt1(.clk(clk), .ram_clk(ram0_clk), .cen(1'b1),
.we(zbt_we),
                                    .addr(zbt_addr),
.write_data(zbt_write_data),
                                    .read_data(zbt_read_data),
                                    .ram_we_b(ram_we_b),
                                    .ram_address(ram_address),
.ram_data(ram_data),
                                    .ram_cen_b(ram_cen_b));


       // Detect the first instance when hcount and vcount == 0
       reg zero_detected;
    // Flag to indicate whether to read or write to RAM (parity bit)
    reg read;

    reg [18:0] read_addr;

    always@(posedge clk)
       begin
      // Set the read such that for every pixel, a read occurs,
followed

      // by a write
            if(hcount == 0 && vcount == 0 && zero_detected == 0)
            begin
         read_addr <= 0;
               zero_detected <= 1;
                  read <= 1;
      end
            // Read from a Pixel
            else if(read && hcount < 11'd800 && vcount < 10'd600)
            begin
         zero_detected <= 0;
```

```
                  read <= ~read;

         zbt_we <= 0;
                  zbt_addr <= read_addr;
         read_addr <= read_addr + 1;
            end
      // Write to a pixel
            else if(we)
            begin
         zero_detected <= 0;
                  read <= ~read;

                  zbt_we <= 1;
                  zbt_addr <= write_addr;
         zbt_write_data <= write_data;
            end
      // If the pixel is supposed to be written to but there is no
pixel to write,
      // then just alternate the parity bit
      else
      begin
         zero_detected <= 0;
                  read <= ~read;
      end
    end

    // Assign the appropriate output connections
    assign read_data = zbt_read_data;
    assign wr = ~read;

       endmodule
```

Appendix OO – xhair.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////////
// Company:
// Engineer:
//
// Create Date:    21:45:49 11/30/06
// Design Name:
// Module Name:    xhair
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////////
module xhair(clk, reset, hcount, vcount, x, y, out);
    input clk;
    input reset;
    input [10:0] hcount;
    input [9:0] vcount;
    input [10:0] x;
    input [9:0] y;
    output  [23:0] out;

    assign out = (hcount == x || vcount == y) ? 24'hffffff: 24'h0;

endmodule
```

Appendix PP – zbt_6111.v

```
//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data
hold
// times around the clock positive edge to work reliably.
//
// Modified by Paul Yang to latch read data


////////////////////////////////////////////////////////////////////
//////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the
actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not
available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when
high.

module zbt_6111(clk, cen, we, addr, write_data, read_data,
                ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk;                 // system clock
   input cen;                 // clock enable for gating ZBT cycles
   input we;                  // write enable (active HIGH)
   input [18:0] addr;         // memory address
   input [35:0] write_data;   // data to write
   output [35:0] read_data;   // data read from memory
   output    ram_clk;   // physical line to ram clock
   output    ram_we_b;  // physical line to ram we_b
   output [18:0] ram_address; // physical line to ram address
   inout [35:0]  ram_data;    // physical line to ram data
   output    ram_cen_b; // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire      ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is
raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;
```

```
   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1,
write_data};

   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_clk = ~clk;     // RAM is not happy with our data
hold
                                   // times if its clk edges equal
FPGA's
                                   // so we clock it on the falling
edges
                                   // and thus let data stabilize
longer
   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};


   //assign      read_data = ram_data;
   // Latch read data

   // Modification to original sample code
   reg [35:0] read_data;
   wire inv_clk = ~clk;
   always @ (posedge inv_clk)
   begin
      if(~we_delay[1])
         read_data <= ram_data;
   end

endmodule // zbt_6111
```

Appendix QQ – labkit.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////
////////
//
// 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)
//
// 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 labkit(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_buf;//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 */

/* 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;
   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;

   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input
```

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

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

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

   // LED Displays

//   assign disp_blank = 1'b1;
//   assign disp_clock = 1'b0;
//   assign disp_rs = 1'b0;
//   assign disp_ce_b = 1'b1;
//   assign disp_reset_b = 1'b0;
//   assign disp_data_out = 1'b0;

   // disp_data_in is an input

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

   // Logic Analyzer
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
```

```
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;


////////////////////////////////////////////////////////////////////////
/////
   // Demonstration of ZBT RAM as video memory



      wire locked_d40, locked_d80, clock_27mhz_buf;
   wire clock_40mhz_unbuf, clock_40mhz, clock_80mhz, clock_80mhz_90;


//   dcm40mhz d40 (
//    .CLKIN_IN(clock_27mhz),
//    .RST_IN(),
//    .CLKFX_OUT(clock_40mhz_unbuf),
//    .CLKIN_IBUFG_OUT(clock_27mhz_buf),
//    .CLK0_OUT(),
//    .LOCKED_OUT(locked_d40)
//    );
//
//   dcm80mhz d80 (
//    .CLKIN_IN(clock_40mhz_unbuf),
//    .RST_IN(),
//    .CLK0_OUT(clock_40mhz),
//    .CLK2X_OUT(clock_80mhz),
//    .LOCKED_OUT(locked_d80)
//    );

   dcm80mhz d80 (
    .CLKIN_IN(clock_27mhz),
    .RST_IN(),
    .CLK0_OUT(),
    .CLKFX_OUT(clock_80mhz),
    .CLKIN_IBUFG_OUT(clock_27mhz_buf),
    .CLK90_OUT(clock_80mhz_90),
    .LOCKED_OUT(locked_d80)
    );

   dcm40mhz d40 (
    .CLKIN_IN(clock_80mhz),
    .RST_IN(),
    .CLKDV_OUT(clock_40mhz),
    .CLK0_OUT(),
    .LOCKED_OUT(locked_d40)
    );



   // synthesis attribute clock_signal of clock_80mhz is “yes”;

   // power-on reset generation
```

```
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_40mhz), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(power_on_reset, clock_40mhz, ~button_enter,
user_reset);
   assign reset = user_reset | power_on_reset | ~locked_d40 |
~locked_d80;



   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   svga svga(clock_40mhz,hcount,vcount,hsync,vsync,blank);




   // Switch between two systems
   wire duck_hunt_enable = switch[7];



   wire [63:0] hex_data;

   wire [63:0] dot_finder_disp_data;

   wire [63:0] duck_hunt_disp_data;



   assign hex_data = duck_hunt_enable ? duck_hunt_disp_data :

                                        dot_finder_disp_data;



   wire [7:0] duck_hunt_led;

   wire [7:0] dot_finder_led = ~(switch);



   assign led = duck_hunt_enable ? duck_hunt_led : dot_finder_led;



   // Declare the wires that are required even if the section is
commented out

   wire [23:0] vga_out;
```

```
   wire [31:0] xform_x;
   wire [31:0] xform_y;



   // Uncomment if my section is commented out


//   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
//   assign user1 = 32'hZ;


//////////////////////////////////////////////////////////////////////
/////
   //                       Video Processing Start
//

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

   assign tv_in_clock = clock_27mhz_buf;
   // adv7185 initialization module
   adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz_buf),
                  .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 f; // sync for field, vertical, horizontal
   wire v;
   wire h;
   wire dv; // data valid

   // NTSC decoder
   ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                  .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                  .ycrcb(ycrcb), .f(f),
                  .v(v), .h(h), .data_valid(dv));

   // Wire the ntsc decoder to the pixel counter
   wire [10:0] pc_x;
   wire [9:0] pc_y;
   wire [29:0] pc_ycrcb_out;
   wire pc_dv_out;
   wire [18:0] pc_ram_addr;

   pixel_counter pc( .clk(tv_in_line_clock1),
                     .reset(reset),
                     .ycrcb_in(ycrcb),
                     .f(f), .v(v), .h(h),
                     .dv_in(dv),
                     .x(pc_x), .y(pc_y),
                     .ycrcb_out(pc_ycrcb_out),
```

```
                    .dv_out(pc_dv_out),
                    .ram_addr(pc_ram_addr));

   // Wire the pixel counter to the color converter
   wire [7:0] cc_r, cc_g, cc_b;
   wire [10:0] cc_x;
   wire [9:0] cc_y;
   wire cc_dv_out;
   wire [18:0] cc_ram_addr;
   YCrCb2RGB color_converter( .clk(tv_in_line_clock1), .rst(reset),
                              .R(cc_r), .G(cc_g), .B(cc_b),
                              .Y(pc_ycrcb_out[29:20]),
                              .Cr(pc_ycrcb_out[19:10]),
                              .Cb(pc_ycrcb_out[9:0]),
                              .x_in(pc_x), .y_in(pc_y),
                              .x_out(cc_x), .y_out(cc_y),
                              .dv_in(pc_dv_out), .dv_out(cc_dv_out),
                              .ram_addr_in(pc_ram_addr),
                              .ram_addr_out(cc_ram_addr));


   // Wire the output of the color converter to the threshold
   wire [10:0] threshold_x;
   wire [9:0] threshold_y;
   wire [7:0] threshold_output;
   wire threshold_dv;
   wire [7:0] threshold;
   wire [18:0] threshold_ram_addr;

   threshold th(  .clk(tv_in_line_clock1), .reset(reset),
                  .x_in(cc_x), .y_in(cc_y), .dv_in(cc_dv_out),
                  .input_val(cc_r), .threshold_val(threshold),
                  .x_out(threshold_x), .y_out(threshold_y),
.dv_out(threshold_dv),
                  .output_val(threshold_output),
                  .ram_addr_in(cc_ram_addr),
                  .ram_addr_out(threshold_ram_addr));


   // Threshold using luminance
//   threshold th(  .clk(tv_in_line_clock1), .reset(reset),
//                  .x_in(pc_x), .y_in(pc_y), .dv_in(pc_dv_out),
//                  .input_val(pc_ycrcb_out[29:22]),
.threshold_val(threshold),
//                  .x_out(threshold_x), .y_out(threshold_y),
.dv_out(threshold_dv),
//                  .output_val(threshold_output),
//                  .ram_addr_in(pc_ram_addr),
//                  .ram_addr_out(threshold_ram_addr));

   // Wire the output of the threshold to the averager
   wire [10:0] avg_x;
   wire [9:0] avg_y;

   averager avg(  .clk(tv_in_line_clock1), .reset(reset),
                  .x_in(threshold_x), .y_in(threshold_y),
```

```
                  .value(threshold_output), .avg_x(avg_x),
.avg_y(avg_y),
                  .dv_in(threshold_dv));


   // Synchronize the signals coming from the pixel counter

//////////////////////////////////////////////////////////////////////
/////
   wire [10:0] sync_pc_x;
   wire [9:0] sync_pc_y;
   wire [29:0] sync_pc_ycrcb;
   wire [18:0] sync_pc_ram_addr;
   wire sync_pc_dv;

   synchronize sync_pc1(.clk(clock_40mhz),.in(pc_x),.out(sync_pc_x));
   defparam sync_pc1.WIDTH = 11;
   synchronize sync_pc2(.clk(clock_40mhz),.in(pc_y),.out(sync_pc_y));
   defparam sync_pc2.WIDTH = 10;
   synchronize
sync_pc3(.clk(clock_40mhz),.in(pc_ycrcb_out),.out(sync_pc_ycrcb));
   defparam sync_pc3.WIDTH = 30;
   synchronize
sync_pc4(.clk(clock_40mhz),.in(pc_dv_out),.out(sync_pc_dv));
   defparam sync_pc4.WIDTH = 1;
   synchronize
sync_pc5(.clk(clock_40mhz),.in(pc_ram_addr),.out(sync_pc_ram_addr));
   defparam sync_pc5.WIDTH = 19;

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

   // Synchronize the signals coming from the color converter

//////////////////////////////////////////////////////////////////////
/////
   wire [10:0] sync_cc_x;
   wire [9:0] sync_cc_y;
   wire [7:0] sync_cc_r, sync_cc_g, sync_cc_b;
   wire sync_cc_dv;
   wire [18:0] sync_cc_ram_addr;

   synchronize sync_cc1(.clk(clock_40mhz),.in(cc_x),.out(sync_cc_x));
   defparam sync_cc1.WIDTH = 11;
   synchronize sync_cc2(.clk(clock_40mhz),.in(cc_y),.out(sync_cc_y));
   defparam sync_cc2.WIDTH = 10;
   synchronize sync_cc3(.clk(clock_40mhz),.in(cc_r),.out(sync_cc_r));
   defparam sync_cc3.WIDTH = 8;
   synchronize sync_cc4(.clk(clock_40mhz),.in(cc_g),.out(sync_cc_g));
   defparam sync_cc4.WIDTH = 8;
   synchronize sync_cc5(.clk(clock_40mhz),.in(cc_b),.out(sync_cc_b));
   defparam sync_cc5.WIDTH = 8;
   synchronize
sync_cc6(.clk(clock_40mhz),.in(cc_dv_out),.out(sync_cc_dv));
   defparam sync_cc6.WIDTH = 1;
   synchronize
sync_cc7(.clk(clock_40mhz),.in(cc_ram_addr),.out(sync_cc_ram_addr));
```

```
   defparam sync_cc7.WIDTH = 19;

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

   // Synchronize the signals coming from threshold

////////////////////////////////////////////////////////////////////////
/////
   wire [10:0] sync_threshold_x;
   wire [9:0] sync_threshold_y;
   wire [7:0] sync_threshold_output;
   wire sync_threshold_dv;
   wire [18:0] sync_threshold_ram_addr;

   synchronize
sync_th1(.clk(clock_40mhz),.in(threshold_x),.out(sync_threshold_x));
   defparam sync_th1.WIDTH = 11;
   synchronize
sync_th2(.clk(clock_40mhz),.in(threshold_y),.out(sync_threshold_y));
   defparam sync_th2.WIDTH = 10;
   synchronize
sync_th3(.clk(clock_40mhz),.in(threshold_output),.out(sync_threshold_ou
tput));
   defparam sync_th3.WIDTH = 8;
   synchronize
sync_th4(.clk(clock_40mhz),.in(threshold_dv),.out(sync_threshold_dv));
   defparam sync_th4.WIDTH = 1;
   synchronize
sync_th5(.clk(clock_40mhz),.in(threshold_ram_addr),.out(sync_threshold_
ram_addr));
   defparam sync_th5.WIDTH = 19;

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


   // Synchronize the signals coming from the averager

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

   wire [10:0] sync_avg_x;

   wire [9:0] sync_avg_y;



   synchronize
sync_avg1(.clk(clock_40mhz),.in(avg_x),.out(sync_avg_x));
   defparam sync_avg1.WIDTH = 11;

   synchronize
sync_avg2(.clk(clock_40mhz),.in(avg_y),.out(sync_avg_y));
   defparam sync_avg2.WIDTH = 10;
```

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



   wire [35:0] a_coeff, b_coeff, c_coeff, d_coeff,
               e_coeff, f_coeff, g_coeff, h_coeff;
   // Coordinate transform

   // First calculate the coefficients

   wire [23:0] x1, y1, x2, y2, x3, y3, x4, y4;
   coeff_gen cg(.clk(clock_40mhz), .reset(reset),
                .x1(x1), .y1(y1), .x2(x2), .x3(x3), .y3(y3), .x4(x4),
                .a(a_coeff), .b(b_coeff), .c(c_coeff), .d(d_coeff),
.e(e_coeff),
                .f(f_coeff), .g(g_coeff), .h(h_coeff));

   // Then actually run the transform
   coord_transform cxform
      (.clk(clock_40mhz), .reset(reset),
      .a(a_coeff),
      .b(b_coeff),
      .c(c_coeff),
      .d(d_coeff),
      .e(e_coeff),
      .f(f_coeff),
      .g(g_coeff),
      .h(h_coeff),
      .x_in(sync_avg_x),
      .y_in(sync_avg_y),
      .x_out(xform_x), .y_out(xform_y), .top_x(), .top_y(), .bottom());




////////////////////////////////////////////////////////////////////////
/////
   //                       Video Processing End
//

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


   // wire up the VRAM

////////////////////////////////////////////////////////////////////////
/////
   wire [35:0] vram_read_data;
   reg [18:0] vram_read_addr;
```

```
   reg [35:0] vram_write_data;
   reg [18:0] vram_write_addr;
   reg vram_we;
   wire wr;

   wire [35:0] box_gen_data;
   wire [18:0] box_gen_addr;
   wire box_gen_we;

   box_gen box(.clk(clock_40mhz),
               .reset(reset),
               .hcount(hcount),
               .vcount(vcount),
               .ram_addr(box_gen_addr),
               .data(box_gen_data), .we(box_gen_we));

   // Select what kind of data to write to the ZBT ram
   always@(posedge clock_40mhz)
   begin
      case(switch[2:1])
         2'b00:
         begin
            vram_write_addr <= box_gen_addr;
            vram_write_data <= box_gen_data;
            vram_we <= box_gen_we;
         end
         2'b01:
         begin
            vram_write_addr <= sync_cc_ram_addr;
            vram_write_data <= {12'h0, sync_cc_r, sync_cc_g,
sync_cc_b};
            vram_we <= sync_cc_dv;
         end
         2'b10:
         begin
            vram_write_addr <= sync_cc_ram_addr;
            vram_write_data <= {12'h0, sync_cc_r, 16'h0};
            vram_we <= sync_cc_dv;
         end
         2'b11:
         begin
            vram_write_addr <= sync_threshold_ram_addr;
            vram_write_data <= {28'h0, sync_threshold_output};
            vram_we <= sync_threshold_dv;
         end
         default:
         begin
            vram_write_addr <= 19'h0;
            vram_write_data <= 36'h0;
            vram_we <= 0;
         end
      endcase
   end


   vram vram1(.clk(clock_80mhz), .clk90(clock_80mhz_90),
.ram0_clk(ram0_clk), .reset(reset),
```

```
               .hcount(hcount),
               .vcount(vcount),
               .read_data(vram_read_data),
               .write_addr(vram_write_addr),
               .write_data(vram_write_data), .we(vram_we), .wr(wr),
               .ram_we_b(ram0_we_b),
               .ram_address(ram0_address),
               .ram_data(ram0_data), .ram_cen_b(ram0_cen_b));

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


   // Cross hairs

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

   wire [10:0] xhair_x;
   wire [10:0] sync_xhair_x;
   wire [9:0] xhair_y;
   wire [9:0] sync_xhair_y;
   wire [23:0] xhair_data;

//   assign xhair_x = threshold;
//   assign xhair_y = threshold;
   assign xhair_x = avg_x;
   assign xhair_y = avg_y;

   // Synchronize the input to the crosshair
   synchronize sync_xh1(.clk(clock_40mhz),
.in(xhair_x),.out(sync_xhair_x));
   defparam sync_xh1.WIDTH = 11;
   synchronize sync_xh2(.clk(clock_40mhz),
.in(xhair_y),.out(sync_xhair_y));
   defparam sync_xh2.WIDTH = 10;

   xhair xh(.clk(clock_40mhz), .reset(reset), .hcount(hcount),
.vcount(vcount),
            .x(sync_xhair_x), .y(sync_xhair_y), .out(xhair_data));


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

   // Blobs

////////////////////////////////////////////////////////////////////////
/////
   wire [23:0] blob_data;
   wire [10:0] cursor_x;
   wire [9:0] cursor_y;
   wire [23:0] xform_blob_data;

   cblob testblob(.x(cursor_x), .y(cursor_y), .hcount(hcount),
.vcount(vcount), .pixel(blob_data));
   defparam testblob.COLOR = 24'h0000ff;
```

```
   cblob xformblob(.x(xform_x[10:0]), .y(xform_y[9:0]),
.hcount(hcount), .vcount(vcount), .pixel(xform_blob_data));
   defparam xformblob.COLOR = 24'hff0000;
   defparam xformblob.SIZE = 8;

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


   // Create edges for the user buttons (signal is high for 1 clk when
pressed)

//////////////////////////////////////////////////////////////////////
/////
   wire button_down_edge, button_up_edge, button_left_edge,
button_right_edge;
   wire button2_edge;

   pb_edge e1(clock_40mhz, reset, ~button_down, button_down_edge);
   pb_edge e2(clock_40mhz, reset, ~button_up, button_up_edge);
   pb_edge e3(clock_40mhz, reset, ~button_left, button_left_edge);
   pb_edge e4(clock_40mhz, reset, ~button_right, button_right_edge);
   pb_edge e5(clock_40mhz, reset, ~button2, button2_edge);

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

   // Everything related to the threshold

//////////////////////////////////////////////////////////////////////
/////
   // display module for debugging
   // Timing issue

   wire [63:0] dispdata;

   //assign dispdata = {5'h0, cursor_x, 6'h0, cursor_y, 24'h000000,
threshold};
   assign dispdata = {a_coeff[31:0], 24'h000000, threshold};
   controls control(.clk(clock_40mhz), .reset(reset),
.button_up(button_up),
                    .button_down(button_down),
.button_left(button_left),
                    .button_right(button_right),
.button_increment(button3),
                    .selection(~switch[6]), .threshold(),
                    .cursor_x(cursor_x), .cursor_y(cursor_y));

   threshold_register tr(.clk(clock_40mhz), .reset(reset),
.cen(switch[6]),
                         .inc(button_up_edge),
                         .dec(button_down_edge),
                         .step(button3), .threshold(threshold));

   assign dot_finder_disp_data = {32'h0, 24'h0, threshold};
```

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

   // Corner select

//////////////////////////////////////////////////////////////////////
/////
   wire [23:0] corner_blob_data;
   corner_select_fsm csfsm (.clk(clock_40mhz), .reset(reset),
.cen(~switch[6]),
                            .inc_x(button_right_edge),
.dec_x(button_left_edge),
                            .inc_y(button_down_edge),
.dec_y(button_up_edge),
                            .increment_size(~button3),
                            .done(button2_edge),
                            .hcount(hcount), .vcount(vcount),
                            .blob_data(corner_blob_data),
                            .x1(x1), .y1(y1), .x2(x2), .y2(y2),
                            .x3(x3), .y3(y3), .x4(x4), .y4(y4));

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

   // Assign the video data

//////////////////////////////////////////////////////////////////////
/////
   wire [23:0] video_data;
   // Limit the output of the video data to the size of the NTSC frame
to hide
   // sync noise
   assign video_data = (hcount < 11'd720 &&
                        vcount < 10'd525) ? vram_read_data : 24'h0;

   // Assign the overlays (xhair etc) (delay by 2 clocks to line up
with zbt

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

   // Assign the cross hair overlay
   wire [23:0] crosshair_overlay;
   assign crosshair_overlay = (switch[3] ? xhair_data : 24'h0);
   wire [23:0] crosshair_overlay_delayed;
   delay co(clock_40mhz, crosshair_overlay, crosshair_overlay_delayed);
   defparam co.WIDTH = 24;

   // Assign the transform blob overlay
   wire [23:0] xform_overlay;
   assign xform_overlay = (switch[4] ? xform_blob_data : 24'h0);
   wire [23:0] xform_overlay_delayed;
   delay xf(clock_40mhz, xform_overlay, xform_overlay_delayed);
   defparam xf.WIDTH = 24;

   // Assign the calibration overlays
```

```
   wire [23:0] calib_overlay;
   assign calib_overlay = (switch[5] ? corner_blob_data : 24'h0);
   wire [23:0] calib_overlay_delayed;
   delay calo(clock_40mhz, calib_overlay, calib_overlay_delayed);
   defparam calo.WIDTH = 24;

   // Assign the background
   wire [23:0] background_data;
   assign background_data = (switch[0] == 0) ? 24'h0 : 24'h00ffff;

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

   // Assign final VGA data

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

   video_priority_encoder vpe (.clk(clock_40mhz), .reset(reset),
                               .background_data(background_data),
                               .video_data(video_data),

.crosshair_overlay(crosshair_overlay_delayed),
                               .xform_overlay(xform_overlay_delayed),
                               .calib_overlay(calib_overlay_delayed),
                               .vga_out(vga_out));

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


   // Laser Driver

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

   wire trigger_input, trigger_edge;



   // Wire the trigger to user1[1] port

   assign user1[31:1] = 31'hZ;

   assign trigger_input = user1[1];



   // Find the edge when there is a transition from low to a high

   pb_edge pe(clock_40mhz, reset, trigger_input, trigger_edge);



   // trigger_db is high whenever the trigger is pressed
```

```
   wire out;

   assign user1[0] = ~out;


   laser_driver ld(clock_40mhz, reset, trigger_edge, out);

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


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


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




      // tiff's stuff starts here

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


   wire game_reset;

   debounce db10(power_on_reset, clock_40mhz, ~button1, game_reset);


   // controlling the shot blob: up, down, left right

   wire up_tmp,down_tmp,left_tmp,right_tmp,up,down,left,right;

   assign up = 0;

   assign down = 0;

   assign left = 0;

   assign right = 0;

//   debounce db5(power_on_reset, clock_40mhz, ~button_up, up_tmp);

//    debounce db6(power_on_reset, clock_40mhz, ~button_down,
down_tmp);

//    debounce db7(power_on_reset, clock_40mhz, ~button_left,
left_tmp);
```

```
//    debounce db8(power_on_reset, clock_40mhz, ~button_right,
right_tmp);

//    signal_to_pulse upgen(clock_40mhz, up_tmp, up);

//    signal_to_pulse downgen(clock_40mhz, down_tmp, down);

//    signal_to_pulse leftgen(clock_40mhz, left_tmp, left);

//    signal_to_pulse rightgen(clock_40mhz, right_tmp, right);




      // button0 kills the duck on screen

   wire kill_duck = trigger_edge;

//   debounce db2(power_on_reset, clock_40mhz, ~button0, kill_duck);


      // button2 is duck_appear from bush fsm

   wire duck_go; // signal for duck to appear on screen

      wire duck_go_tmp;

   assign duck_go = 0;

   assign duck_go_tmp = 0;


   //debounce db4(power_on_reset, clock_40mhz, ~button2, duck_go_tmp);

      //signal_to_pulse duck_go_pulser(.clk(clock_40mhz),
.signal(duck_go_tmp),

      //
      .pulse(duck_go));

      //reg duck_go_debug;

      //always @ (posedge duck_go)

      //begin

      //    duck_go_debug <= ~duck_go_debug;

      //end


   // feed XVGA signals to video controller for duck hunt game
```

```
wire [3:0] game_pixel;     // using 16 color (4-bit color)
wire dhsync,dvsync,dblank;

   // wires for ducks
   wire [3:0] duck_pixel;
   wire [10:0] duck_x;
   wire [9:0] duck_y;
   wire [3:0] state;
   wire [9:0] duck_height;
   wire [10:0] duck_width;
   wire speed_enable;
   wire [1:0] duck_shots;
   wire expire; // for timer
   wire one_hz_enable; // for second_timer

   // appear_counter
   wire [1:0] appear_count; // for debugging appear_counter
   wire [2:0] disappear_count; //
   wire appear_enable;
   wire disappear_enable;
   wire appear;
   wire disappear;

   // velocity
   wire [3:0] duck_speed;
   wire [2:0] duck_count;

   // wires for tiff's bushes
```

```
wire [9:0] bush_height;
wire [10:0] bush_width;
wire [3:0] bush_pixel1;
wire [10:0] bush_x1 = 11'd0;
wire [9:0] bush_y1 = 10'd450;
wire [3:0] bush_pixel2;
wire [10:0] bush_x2 = 11'd100;
wire [9:0] bush_y2 = 10'd450;
wire [3:0] bush_pixel3;
wire [10:0] bush_x3 = 11'd200;
wire [9:0] bush_y3 = 10'd450;
wire [3:0] bush_pixel8;
wire [10:0] bush_x8 = 11'd300;
wire [9:0] bush_y8 = 10'd450;
wire [3:0] bush_pixel9;
wire [10:0] bush_x9 = 11'd400;
wire [9:0] bush_y9 = 10'd450;
wire [3:0] bush_pixel10;
wire [10:0] bush_x10 = 11'd500;
wire [9:0] bush_y10 = 10'd450;
wire [3:0] bush_pixel11;
wire [10:0] bush_x11 = 11'd600;
wire [9:0] bush_y11 = 10'd450;
wire [3:0] bush_pixel12;
wire [10:0] bush_x12 = 11'd700;
wire [9:0] bush_y12 = 10'd450;

// wires for cloud_controller
```

```
// cloud1
wire cloud_enable1;
wire [3:0] cloud_pixel1;
wire [10:0] cloud_x1;
wire [9:0] cloud_y1;
wire [10:0] cloud_initx1 = 11'd300;
wire [9:0] cloud_inity1 = 10'd30;
wire [3:0] cloud_speed1;
// cloud2
wire cloud_enable2;
wire [3:0] cloud_pixel2;
wire [10:0] cloud_x2;
wire [9:0] cloud_y2;
wire [10:0] cloud_initx2 = 11'd000;
wire [9:0] cloud_inity2 = 10'd35;
wire [3:0] cloud_speed2;
//cloud3
wire cloud_enable3;
wire [3:0] cloud_pixel3;
wire [10:0] cloud_x3;
wire [9:0] cloud_y3;
wire [10:0] cloud_initx3 = 11'd700;
wire [9:0] cloud_inity3 = 10'd40;
wire [3:0] cloud_speed3;
wire [10:0] cloud_width;
wire [9:0] cloud_height;

wire [10:0] sky_width;
```

```
      wire [9:0] sky_height;


      wire gameover;

      wire win;

      wire halt = gameover || win;

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


      // interfacing between spencer's bushes and tiff's ducks

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

      wire duck_enb_tmp;

      wire duck_enb = (duck_go) || (duck_enb_tmp);

      wire offscreen_pulse;

   wire shudder_time_expire;

      signal_to_pulse duck_enb1(.clk(clock_40mhz),
.signal(shudder_time_expire),


      .pulse(duck_enb_tmp));

      signal_to_pulse bushenb(.clk(clock_40mhz), .signal(offscreen),

            .pulse(offscreen_pulse));

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


      // random number generators

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

      wire [2:0] r_interval_tmp;

      wire [1:0] r_interval = r_interval_tmp[1:0];

      wire [2:0] bush_select_tmp;

      wire [1:0] bush_select = bush_select_tmp[1:0];
```

```
      wire [2:0] seed;

      wire [2:0] interval_seed = (seed == 3'd0 ) ? 1 : seed;

      wire [2:0] bush_seed = (seed == 3'd7) ? 2 : seed + 1;

      wire load_enb = reset | game_reset;

      seedgen seedgen1(.clk(clock_40mhz), .seed(seed));

      // random number generator to determine time interval between
duck

      randgen interval_gen1(.clk(clock_40mhz), .pd_out(r_interval_tmp),

            .ce(offscreen_pulse | load_enb), .load(load_enb),
.pd_in(interval_seed));

      // random number generator to determine which bush duck should
come out of

      randgen bush_selector1(.clk(clock_40mhz),
.pd_out(bush_select_tmp),

            .ce(offscreen_pulse | load_enb), .load(load_enb),
.pd_in(bush_seed));

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



      // other wires

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

      wire [1:0] gamestate;

      // button3 is start game

   wire start_game = disappear && (gamestate == 2'd0);

   // debounce db5(power_on_reset, clock_40mhz, ~button3, start_game);

      wire [9:0] blob_y; // for displaying tiff's simulation of

      wire [10:0] blob_x; // paul's x & y coordinates

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



      // timer for duck animations
```

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

      timer timer1(.clk(clock_40mhz), .reset(reset | game_reset),

            .new_frame(new_frame), .expire(expire));

      // debugging

      reg expire_debug;

      always @ (posedge clock_40mhz)

      begin

            if (expire) expire_debug <= ~expire_debug;

      end

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



      // instantiate frame handler

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

      frame frame1(.vclock(clock_40mhz), .vsync(vsync),
.new_frame(new_frame));

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



      // instantiate cloud_controllers

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

      // cloud1

      cloud_controller cloud_controller1(

            // inputs

            .vclock(clock_40mhz), .reset(reset | game_reset),

            .cloud_enable(cloud_enable1), .east(1'b1),
.cloud_speed(cloud_speed1),

            .cloud_initx(cloud_initx1), .cloud_inity(cloud_inity1),
```

```
            .hcount(hcount), .vcount(vcount), .hsync(hsync),
.vsync(vsync),

            .blank(blank), .new_frame(new_frame), .halt(halt),

            .cloud_width(cloud_width), .cloud_height(cloud_height),

            .sky_width(sky_width),

            // outputs

            .cloud_pixel(cloud_pixel1),

            .cloud_x(cloud_x1), .cloud_y(cloud_y1),

            .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank),

            .cloud_debug(cloud_debug1));

      // cloud2

      cloud_controller cloud_controller2(

            // inputs

            .vclock(clock_40mhz), .reset(reset | game_reset),

            .cloud_enable(cloud_enable2), .east(1'b1),
.cloud_speed(cloud_speed2),

            .cloud_initx(cloud_initx2), .cloud_inity(cloud_inity2),

            .hcount(hcount), .vcount(vcount), .hsync(hsync),
.vsync(vsync),

            .blank(blank), .new_frame(new_frame), .halt(halt),

            .cloud_width(cloud_width), .cloud_height(cloud_height),

            .sky_width(sky_width),

            // outputs

            .cloud_pixel(cloud_pixel2),

            .cloud_x(cloud_x2), .cloud_y(cloud_y2),

            .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank),

            .cloud_debug(cloud_debug2));

      // cloud3

      cloud_controller cloud_controller3(

            // inputs
```

```
            .vclock(clock_40mhz), .reset(reset | game_reset),

            .cloud_enable(cloud_enable3), .east(1'b0),
.cloud_speed(cloud_speed3),

            .cloud_initx(cloud_initx3), .cloud_inity(cloud_inity3),

            .hcount(hcount), .vcount(vcount), .hsync(hsync),
.vsync(vsync),

            .blank(blank), .new_frame(new_frame), .halt(halt),

            .cloud_width(cloud_width), .cloud_height(cloud_height),

            .sky_width(sky_width),

            // outputs

            .cloud_pixel(cloud_pixel3),

            .cloud_x(cloud_x3), .cloud_y(cloud_y3),

            .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank),

            .cloud_debug(cloud_debug3));

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


      // instantiate stationary bushes

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

      // bush 1

      bush_controller_tiff bush_controller_tiff1(

            // inputs

            .vclock(clock_40mhz), .reset(reset | game_reset),

            .bush_enable(1'b1), .bush_x(bush_x1), .bush_y(bush_y1),

            .hcount(hcount), .vcount(vcount),

            .hsync(hsync), .vsync(vsync), .blank(blank),

            .bush_height(bush_height), .bush_width(bush_width),

            // outputs

            .bush_pixel(bush_pixel1),
```

```
      .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank));
// bush 2
bush_controller_tiff bush_controller_tiff2(
      // inputs
      .vclock(clock_40mhz), .reset(reset | game_reset),
      .bush_enable(1'b1), .bush_x(bush_x2), .bush_y(bush_y2),
      .hcount(hcount), .vcount(vcount),
      .hsync(hsync), .vsync(vsync), .blank(blank),
      .bush_height(bush_height), .bush_width(bush_width),
      // outputs
      .bush_pixel(bush_pixel2),
      .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank));
// bush 3
bush_controller_tiff bush_controller_tiff3(
      // inputs
      .vclock(clock_40mhz), .reset(reset | game_reset),
      .bush_enable(1'b1), .bush_x(bush_x3), .bush_y(bush_y3),
      .hcount(hcount), .vcount(vcount),
      .hsync(hsync), .vsync(vsync), .blank(blank),
      .bush_height(bush_height), .bush_width(bush_width),
      // outputs
      .bush_pixel(bush_pixel3),
      .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank));
bush_controller_tiff bush_controller_tiff8(
      // inputs
      .vclock(clock_40mhz), .reset(reset | game_reset),
      .bush_enable(1'b1), .bush_x(bush_x8), .bush_y(bush_y8),
      .hcount(hcount), .vcount(vcount),
```

```
      .hsync(hsync), .vsync(vsync), .blank(blank),
      .bush_height(bush_height), .bush_width(bush_width),
      // outputs
      .bush_pixel(bush_pixel8),
      .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank));
bush_controller_tiff bush_controller_tiff9(
      // inputs
      .vclock(clock_40mhz), .reset(reset | game_reset),
      .bush_enable(1'b1), .bush_x(bush_x9), .bush_y(bush_y9),
      .hcount(hcount), .vcount(vcount),
      .hsync(hsync), .vsync(vsync), .blank(blank),
      .bush_height(bush_height), .bush_width(bush_width),
      // outputs
      .bush_pixel(bush_pixel9),
      .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank));
bush_controller_tiff bush_controller_tiff10(
      // inputs
      .vclock(clock_40mhz), .reset(reset | game_reset),
      .bush_enable(1'b1), .bush_x(bush_x10), .bush_y(bush_y10),
      .hcount(hcount), .vcount(vcount),
      .hsync(hsync), .vsync(vsync), .blank(blank),
      .bush_height(bush_height), .bush_width(bush_width),
      // outputs
      .bush_pixel(bush_pixel10),
      .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank));
bush_controller_tiff bush_controller_tiff11(
      // inputs
      .vclock(clock_40mhz), .reset(reset | game_reset),
```

```
            .bush_enable(1'b1), .bush_x(bush_x11), .bush_y(bush_y11),
            .hcount(hcount), .vcount(vcount),
            .hsync(hsync), .vsync(vsync), .blank(blank),
            .bush_height(bush_height), .bush_width(bush_width),
            // outputs
            .bush_pixel(bush_pixel11),
            .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank));
      bush_controller_tiff bush_controller_tiff12(
            // inputs
            .vclock(clock_40mhz), .reset(reset | game_reset),
            .bush_enable(1'b1), .bush_x(bush_x12), .bush_y(bush_y12),
            .hcount(hcount), .vcount(vcount),
            .hsync(hsync), .vsync(vsync), .blank(blank),
            .bush_height(bush_height), .bush_width(bush_width),
            // outputs
            .bush_pixel(bush_pixel12),
            .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank));
      ////////////////////////////////////////////////////////////////
//////////



      // duck_health
      ////////////////////////////////////////////////////////////////
//////////
      duck_health duck_health1 (
            // inputs
            .vclock(clock_40mhz), .reset(reset | game_reset),
.duck_go(duck_enb),
            .disappear_flag(disappear_flag), .kill_duck(kill_duck),
            .duck_x(duck_x), .duck_y(duck_y),
```

```
            .duck_width(duck_width), .duck_height(duck_height),

            // trigger,

            .dot_x(xform_x), .dot_y(xform_y),

            // outputs

            .hit_duck(hit_duck), .duck_die(duck_die),

            .duck_shots(duck_shots));

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


      // appear_counter that controls when ducks fly on/off

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

      appear_counter appear_counter1(

            //inputs

            .clk(clock_40mhz), .reset(reset | game_reset),

            .r_interval(r_interval), .one_hz_enable(one_hz_enable),

            .appear_enable(duck_enb),
.disappear_enable(disappear_enable),

            //outputs

            .appear_count(appear_count),
.disappear_count(disappear_count),

            .appear(appear), .disappear(disappear));

      // debugging

      reg appear_debug;

   reg disappear_debug;

      always @ (posedge clock_40mhz)

      begin

            if (appear) appear_debug <= ~appear_debug;

            if (disappear) disappear_debug <= ~disappear_debug;

      end
```

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


      // velocity that determines duck speed

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

      velocity velocity1(.clk(clock_40mhz), .reset(reset | game_reset),

            .speed_enable(duck_enb), .duck_count(duck_count),
.duck_speed(duck_speed));

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



//    wire appear_pulse_tmp;

//    signal_to_pulse appear_pulse_generator(clock_40mhz, appear,
appear_pulse_tmp);

//

//   reg appear_reg;

//    reg appear_pulse;

//    always @ (posedge clock_40mhz)

//    begin

//          if (reset) begin

//                appear_reg <= 0;

//                appear_pulse <= 0;

//                end // end reset

//          if (appear_pulse_tmp)

//                appear_reg <= 1;

//          else if ((hcount == 11'd0) && (vcount == 10'd0) &&
(appear_reg == 1)) begin

//                appear_pulse <= 1;

//                appear_reg <= 0;

//                end
```

```
//          else appear_pulse <= 0;

//    end


      // instantiate duck_controller

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

      duck_controller duck_controller1 (

            // inputs

            .vclock(clock_40mhz), .reset(reset | game_reset),

            // .trigger(trigger), .dot_x(dot_x), .dot_y(dot_y),

            .hit_duck(hit_duck), .duck_die(duck_die),

            .expire(expire), .appear(appear), .disappear(disappear),

            .bush_select(bush_select), .duck_speed(duck_speed),

            .hcount(hcount), .vcount(vcount), .hsync(hsync),
.vsync(vsync),

            .blank(blank), .new_frame(new_frame), .halt(halt),
.duck_go(duck_enb),

            .duck_width(duck_width), .duck_height(duck_height),

            .sky_height(sky_height), .sky_width(sky_width),

            // outputs

            .appear_enable(appear_enable),
.disappear_enable(disappear_enable),

            .disappear_flag(disappear_flag),

            .duck_pixel(duck_pixel), .duck_x(duck_x), .duck_y(duck_y),
.offscreen(offscreen),

            .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank),

            .state(state), .debug(duck_debug));

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

```
      // second timer
      ////////////////////////////////////////////////////////////////
//////////
      second_timer second_timer1 (
            //inputs
            .clk(clock_40mhz), .reset(reset | game_reset),
.duck_go(duck_enb),
            //outputs
            .one_hz_enable(one_hz_enable));
      // debugging
   reg second;
   always @ (posedge clock_40mhz)
   begin
      if (one_hz_enable) second <= ~second;
      end
      ////////////////////////////////////////////////////////////////
//////////



      /// SPENCER'S STUFF///
      ////////////////////////////////////////////////////////////////
//////////
  wire [3:0] score_pixel;


  wire [5:0] countdown_game = 6'd45;
  wire [5:0] count;
  timer_spencer timer_thirty (//inputs
                                          reset | game_reset,
clock_40mhz, 0, start_game, win,
                                          countdown_game,
one_hz_enable,
```

```
                                        //outputs
                                        timer_expire, count);

      wire [2*8-1:0] timer_thirty_string_val;
      timer_strings timer_strings_thirty (//inputs

clock_40mhz, hcount, vcount, count,

//outputs

timer_thirty_string_val);

      //Timer Displays
      wire [10:0] hard_timer_thirty_x = 11'd450;
      wire [9:0] hard_timer_thirty_y = 10'd560;
      wire [10:0] hard_timer_thirty_x_length = 11'd256;     //256 =
(15+1)*16

      wire[3:0] hard_timer_thirty_pixel;
      wire [15*8-1:0] hard_timer_thirty_string = "Time Remaining:";
      char_string_display hard_timer_thirty_display
(clock_40mhz,hcount,vcount,
                  hard_timer_thirty_pixel,hard_timer_thirty_string,
                  hard_timer_thirty_x,hard_timer_thirty_y);
   defparam    hard_timer_thirty_display.NCHAR = 15;
   defparam    hard_timer_thirty_display.NCHAR_BITS = 4;

      wire [10:0] timer_thirty_x = hard_timer_thirty_x +
hard_timer_thirty_x_length;
      wire [9:0] timer_thirty_y = hard_timer_thirty_y;
      wire [10:0] timer_thirty_x_length = 11'd32;
```

```
      wire[3:0] timer_thirty_pixel;
      wire [2*8-1:0] timer_thirty_string = timer_thirty_string_val;
      char_string_display timer_thirty_display
(clock_40mhz,hcount,vcount,

timer_thirty_pixel,timer_thirty_string,timer_thirty_x,timer_thirty_y);
   defparam    timer_thirty_display.NCHAR = 2;
   defparam    timer_thirty_display.NCHAR_BITS = 2;

      // Score Displays
      wire [10:0] hard_score_x = 11'd450;
      wire [9:0] hard_score_y = 10'd525;

      wire[3:0] hard_score_pixel;
      wire [6*8-1:0] hard_score_string = "Score:";
      char_string_display hard_score (clock_40mhz,hcount,vcount,

hard_score_pixel,hard_score_string,hard_score_x,hard_score_y);
   defparam    hard_score.NCHAR = 6;
   defparam    hard_score.NCHAR_BITS = 3;

      wire [2*8-1:0] score_string_val;
      wire [3:0] score_val;
      score_strings score_strings1 (.clk(clock_40mhz),
            .hcount(hcount), .vcount(vcount),
            .score_val(score_val),
            .score_string_val(score_string_val));

      wire [10:0] score_x = hard_score_x + 11'd108;
```

```
      wire [9:0] score_y = hard_score_y;

      wire [2*8-1:0] score_string = score_string_val;
      char_string_display score(clock_40mhz,hcount,vcount,
                     score_pixel,score_string,score_x,score_y);
   defparam    score.NCHAR = 2;
   defparam    score.NCHAR_BITS = 2;

//    wire [3:0] score_val;
      scorekeeper scorekeeper1 (//inputs
.vclock(clock_40mhz),
.duck_die(duck_die),
.reset(reset|game_reset),
.timer_expire(timer_expire),
                                                        //output
.gameover(gameover),
                                                        .win(win),
.score_val(score_val)
                                                        );

      wire shudder_time = (gamestate == 2'd0) ? 6'd16 : 6'd1;
      wire [5:0] shudder_count;

      wire bush_shudder_enb = ((gamestate == 2'd1) &&
(offscreen_pulse)) | start_game;
```

```
      reg bush_shudder_enb_delayed;

      always @ (posedge clock_40mhz)

      bush_shudder_enb_delayed <= bush_shudder_enb;


      timer_spencer shudder_timer1 (//inputs

                                          .reset(reset|game_reset),
.clk(clock_40mhz), .start_timer(bush_shudder_enb),

                                          .start_game(1'b0),
.halt(halt), .value(shudder_time),


.one_hz_enable(one_hz_enable),

                                          // outputs


.expired(shudder_time_expire),

                                          .count(shudder_count));

      reg shudder_timer_expire_debug;

      always @ (posedge clock_40mhz)

      if (duck_enb_tmp) shudder_timer_expire_debug <=
~shudder_timer_expire_debug;


      wire bush_select1;

      wire bush_select2;

      wire bush_select3;

      wire bush_select4;


      bush_selector bush_selector2 (//inputs


      clock_40mhz, reset|game_reset, one_hz_enable,


      bush_select, bush_shudder_enb_delayed,


      shudder_time_expire,
```

```
                                                                //outputs
        bush_select1, bush_select2, bush_select3, bush_select4,
        start_shudder_timer
                                                                );

        parameter BUSH_HEIGHT = 10'd50;
        parameter BUSH_WIDTH = 11'd133;
        wire [3:0] bush_pixel4;
        wire [10:0] bush_x4;
        wire [9:0] bush_y4;
        bush_animator2 bush_animator1 (//inputs
        clock_40mhz, reset|game_reset,
        bush_select1, //
                                                                hcount,
vcount, hsync, vsync, blank,
                                                                new_frame,
                                                                halt,
        bush_pixel4, bush_x4, bush_y4, dhsync, dvsync, dblank);

        wire [3:0] bush_pixel5;
        wire [10:0] bush_x5;
        wire [9:0] bush_y5;
        bush_animator2 bush_animator2 (//inputs
        clock_40mhz, reset|game_reset,
        bush_select2, // temporarily the up button
```

```
                                                                hcount,
vcount, hsync, vsync, blank,
                                                                new_frame,
                                                                halt,

      bush_pixel5, bush_x5, bush_y5, dhsync, dvsync, dblank);


      wire [3:0] bush_pixel6;
      wire [10:0] bush_x6;
      wire [9:0] bush_y6;
      bush_animator2 bush_animator3 (//inputs

      clock_40mhz, reset|game_reset,

      bush_select3, // temporarily the up button
                                                                hcount,
vcount, hsync, vsync, blank,
                                                                new_frame,
                                                                halt,

      bush_pixel6, bush_x6, bush_y6, dhsync, dvsync, dblank);


      wire [3:0] bush_pixel7;
      wire [10:0] bush_x7;
      wire [9:0] bush_y7;
      bush_animator2 bush_animator4 (//inputs

      clock_40mhz, reset|game_reset,

      bush_select4, // temporarily the up button
                                                                hcount,
vcount, hsync, vsync, blank,
                                                                new_frame,
```

```
                                                        halt,
      bush_pixel7, bush_x7, bush_y7, dhsync, dvsync, dblank);


      defparam    bush_animator4.BUSH_X_START = 11'd450;

      defparam    bush_animator3.BUSH_X_START = 11'd350;

      defparam    bush_animator2.BUSH_X_START = 11'd200;

      defparam    bush_animator1.BUSH_X_START = 11'd50;

////////////////
// Name Text      //
////////////////

      parameter PROJECT_TITLE_X = 11'd325;
      parameter PROJECT_TITLE_Y = 10'd430;
      // parameter PROJECT_TITLE_Y = 10'd350;

   wire [10:0] title_x1 = PROJECT_TITLE_X;
      wire [9:0] title_y1 = PROJECT_TITLE_Y;
      wire [10:0] title_x_length1 = 11'd144;    //256 = (15+1)*16
      wire[3:0] title_pixel1;
      wire [9*8-1:0] title_string1 = "Lasershot";
      char_string_display title_display1 (clock_40mhz,hcount,vcount,
                  title_pixel1,title_string1,
                  title_x1,title_y1);
```

```
defparam    title_display1.NCHAR = 9;
defparam    title_display1.NCHAR_BITS = 4;

   wire [10:0] title_x2 = PROJECT_TITLE_X;
   wire [9:0] title_y2 = PROJECT_TITLE_Y + 10'd30;
   wire [10:0] title_x_length2 = 11'd336;    //256 = (15+1)*16
   wire[3:0] title_pixel2;
   wire [21*8-1:0] title_string2 = "A 6.111 Final Project";
   char_string_display title_display2 (clock_40mhz,hcount,vcount,
                title_pixel2,title_string2,
                title_x2,title_y2);
defparam    title_display2.NCHAR = 21;
defparam    title_display2.NCHAR_BITS = 5;

   wire [10:0] title_x3 = PROJECT_TITLE_X;
   wire [9:0] title_y3 = PROJECT_TITLE_Y + 10'd60;
   wire [10:0] title_x_length3 = 11'd256;    //256 = (15+1)*16
   wire[3:0] title_pixel3;
   wire [16*8-1:0] title_string3 = "By Tiffany Chen,";
   char_string_display title_display3 (clock_40mhz,hcount,vcount,
                title_pixel3,title_string3,
                title_x3,title_y3);
defparam    title_display3.NCHAR = 16;
defparam    title_display3.NCHAR_BITS = 4;

   wire [10:0] title_x4 = PROJECT_TITLE_X +11'd50;
   wire [9:0] title_y4 = PROJECT_TITLE_Y + 10'd90;
   wire [10:0] title_x_length4 = 11'd272;    //256 = (15+1)*16
```

```
      wire[3:0] title_pixel4;
      wire [17*8-1:0] title_string4 = "Spencer Sugimoto,";
      char_string_display title_display4 (clock_40mhz,hcount,vcount,
                  title_pixel4,title_string4,
                  title_x4,title_y4);
   defparam    title_display4.NCHAR = 17;
   defparam    title_display4.NCHAR_BITS = 5;

      wire [10:0] title_x5 = PROJECT_TITLE_X +11'd130;
      wire [9:0] title_y5 = PROJECT_TITLE_Y + 10'd120;
      wire [10:0] title_x_length5 = 11'd208;    //256 = (15+1)*16
      wire[3:0] title_pixel5;
      wire [13*8-1:0] title_string5 = "and Paul Yang";
      char_string_display title_display5 (clock_40mhz,hcount,vcount,
                  title_pixel5,title_string5,
                  title_x5,title_y5);
   defparam    title_display5.NCHAR = 13;
   defparam    title_display5.NCHAR_BITS = 4;


      parameter volume = 5'd20;
   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;
      // AC97 driver
   lab4audio a(clock_27mhz_buf, reset|game_reset, volume,
from_ac97_data, to_ac97_data, ready,
            audio_reset_b, ac97_sdata_out, ac97_sdata_in,
            ac97_synch, ac97_bit_clock);
```

```
   // light up LEDs when recording, show volume during playback.
   // led is active low
   //assign led = playback ? ~{3'b000, volume} : 8'h00;

   wire [7:0] to_ac97_data1;
      wire [11:0] laser_addr;
      wire [7:0] s1;
      wire [7:0] s2;
      wire [3:0] laser_count;
      wire [2:0] laser_state;

//    wire [7:0] to_ac97_data2;
//    wire [11:0] laser_addra;
//    wire [7:0] s1a;
//    wire [7:0] s2a;
//    wire [3:0] laser_counta;
//    wire [2:0] laser_statea;
//
wire [7:0] to_ac97_data3;
      wire [1:0] gameover_addr;
      wire [7:0] gameover_s1;
      wire [7:0] gameover_s2;
      wire [3:0] gameover_count;
      wire [2:0] gameover_state;


      // Debounce interpolate switch
//    wire interpolate;
```

```
//    debounce interpolate_switch (reset, clock_27mhz_buf, switch[4],
interpolate);

//    // push button3 button to record, release to playback

//   wire playback;

//   debounce benter(reset, clock_27mhz, ~button3, playback);

//

      // instantiate laser sound

//    sound_shotgun sound_shotgun1(clock_27mhz, reset, playback, ready,
interpolate, to_ac97_data1, laser_addr, s1, s2, laser_count,
laser_state);

//   sound_laser sound_laser1(clock_40mhz, reset, playback, ready,
interpolate, to_ac97_data2, laser_addra, s1a, s2a, laser_counta,
laser_statea);

//



   // Make the kill duck signal 2 cycles wide

//    reg kill_duck_enable;

//   reg kill_duck_count;

//   always @ (posedge clock_40mhz)

//   begin

//      if(reset)

//      begin

//         kill_duck_enable <= 0;

//         kill_duck_count <= 0;

//      end

//      else if(kill_duck)

//      begin

//         kill_duck_enable <= 1;

//         kill_duck_count <= 1;

//      end

//      else if(kill_duck_count == 0)
```

```
//         kill_duck_enable <= 0;

//      else

//         kill_duck_count <= 0;

//   end



   // Then synchronize the signal to the 27MHz clock

   wire kill_duck_doubled, sync_kill_duck_doubled;

   pulse_doubler pd1(.clk(clock_40mhz), .reset(reset),

                     .input_pulse(kill_duck),
.doubled_pulse(kill_duck_doubled));

   synchronize sync_kd(clock_27mhz_buf, kill_duck_doubled,
sync_kill_duck_doubled);

   defparam sync_kd.WIDTH = 1;



   sound_laser sound_laser1(clock_27mhz_buf, reset|game_reset,
sync_kill_duck_doubled, ready, 1'b1, to_ac97_data1, laser_addr, s1, s2,
laser_count, laser_state);

   //sound_shotgun sound_shotgun1 (clock_27mhz, reset, kill_duck,
ready, interpolate, to_ac97_data2, laser_addra, s1a, s2a, laser_counta,
laser_statea);



   wire gameover_pulse, gameover_pulse_doubled;

   signal_to_pulse spgameover(clock_40mhz, gameover, gameover_pulse);

   pulse_doubler pd2(.clk(clock_40mhz), .reset(reset),

                     .input_pulse(gameover_pulse),
.doubled_pulse(gameover_pulse_doubled));



   wire sync_gameover;

   synchronize sync_go(clock_27mhz_buf, gameover_pulse_doubled,
sync_gameover);

   defparam sync_go.WIDTH = 1;

      sound_gameover sound_gameover1(clock_27mhz_buf, reset|game_reset,
sync_gameover, ready, 1'b1, to_ac97_data3, gameover_addr, gameover_s1,
```

```
gameover_s2, gameover_count, gameover_state);


      assign to_ac97_data = (to_ac97_data1 + to_ac97_data3);

      //instantiate laser overlay


      // END SPENCER'S STUFF //

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


      // video controller for game

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

   video_controller video_controller1(

               // inputs

               .vclock(clock_40mhz),.reset(reset | game_reset),


.hcount(hcount),.vcount(vcount),.hsync(hsync),.vsync(vsync),.blank(blan
k),

               .start_game(start_game), .halt(halt), .win(win),
.gameover(gameover),

               .duck_pixel(duck_pixel), .duck_x(duck_x),
.duck_y(duck_y),

               .cloud_pixel1(cloud_pixel1), .cloud_x1(cloud_x1),
.cloud_y1(cloud_y1),

               .cloud_pixel2(cloud_pixel2), .cloud_x2(cloud_x2),
.cloud_y2(cloud_y2),

               .cloud_pixel3(cloud_pixel3), .cloud_x3(cloud_x3),
.cloud_y3(cloud_y3),

               .bush_pixel1(bush_pixel1), .bush_x1(bush_x1),
.bush_y1(bush_y1),

               .bush_pixel2(bush_pixel2), .bush_x2(bush_x2),
.bush_y2(bush_y2),

               .bush_pixel3(bush_pixel3), .bush_x3(bush_x3),
.bush_y3(bush_y3),
```

```
                .bush_pixel8(bush_pixel8), .bush_x8(bush_x8),
.bush_y8(bush_y8),

                .bush_pixel9(bush_pixel9), .bush_x9(bush_x9),
.bush_y9(bush_y9),

                .bush_pixel10(bush_pixel10), .bush_x10(bush_x10),
.bush_y10(bush_y10),

                .bush_pixel11(bush_pixel11), .bush_x11(bush_x11),
.bush_y11(bush_y11),

                .bush_pixel12(bush_pixel12), .bush_x12(bush_x12),
.bush_y12(bush_y12),



                // inputs from spencer

                .sk_pixel(sk_pixel), .score_pixel(score_pixel),

                .hard_score_pixel(hard_score_pixel), .hs_x(hard_score_x),

                .hs_y(hard_score_y),

                .score_x(score_x), .score_y(score_y),

                .hard_timer_thirty_pixel(hard_timer_thirty_pixel),

                .hard_timer_thirty_x(hard_timer_thirty_x),

                .hard_timer_thirty_y(hard_timer_thirty_y),

                .hard_timer_thirty_x_length(hard_timer_thirty_x_length),

                .timer_thirty_pixel(timer_thirty_pixel),
.timer_thirty_x(timer_thirty_x),

                .timer_thirty_y(timer_thirty_y),

                .timer_thirty_x_length(timer_thirty_x_length),

                .bush_pixel4(bush_pixel4), .bush_x4(bush_x4),
.bush_y4(bush_y4),

                .bush_pixel5(bush_pixel5), .bush_x5(bush_x5),
.bush_y5(bush_y5),

                .bush_pixel6(bush_pixel6), .bush_x6(bush_x6),
.bush_y6(bush_y6),

                .bush_pixel7(bush_pixel7), .bush_x7(bush_x7),
.bush_y7(bush_y7),

        .title_pixel1(title_pixel1), .title_x1(title_x1),
.title_y1(title_y1),
```

```
              .title_x_length1(title_x_length1),

              .title_pixel2(title_pixel2), .title_x2(title_x2),
.title_y2(title_y2),

              .title_x_length2(title_x_length2),

              .title_pixel3(title_pixel3), .title_x3(title_x3),
.title_y3(title_y3),

              .title_x_length3(title_x_length3),

              .title_pixel4(title_pixel4), .title_x4(title_x4),
.title_y4(title_y4),

              .title_x_length4(title_x_length4),

              .title_pixel5(title_pixel5), .title_x5(title_x5),
.title_y5(title_y5),

              .title_x_length5(title_x_length5),


              // blob for paul's x & y coordinates

              .blob_enb(switch[2]),

        .xform_blob_data(xform_blob_data),


              // outputs

              .bush_height(bush_height), .bush_width(bush_width),

              .cloud_height(cloud_height), .cloud_width(cloud_width),

              .sky_height(sky_height), .sky_width(sky_width),

              .duck_height(duck_height), .duck_width(duck_width),

              .cloud_speed1(cloud_speed1), .cloud_speed2(cloud_speed2),
.cloud_speed3(cloud_speed3),

              .cloud_enable1(cloud_enable1),
.cloud_enable2(cloud_enable2), .cloud_enable3(cloud_enable3),


.dhsync(dhsync),.dvsync(dvsync),.dblank(dblank),.game_pixel(game_pixel)
,

              .gamestate(gamestate),

              .blob_x(blob_x), .blob_y(blob_y),
```

```
              .up(up),.down(down),.left(left),.right(right));


      // instantiate colormap for 16-color

   // With modifications to display a blob at the location of the last
shot

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

      wire [23:0] rgb;

   wire [23:0] colormap_output;

      colormap colormap1 (.clk(clock_40mhz), .hsync(hsync),
.vsync(vsync), .blank(blank),

            .hcount(hcount), .vcount(vcount), .switch(switch[1:0]),

            .dhsync(dhsync), .dvsync(dvsync), .dblank(dblank),

            .game_pixel(game_pixel), .b(b), .hs(hs), .vs(vs),

            .rgb(colormap_output));

   // Don't display transform dot on the title screen

   assign rgb = colormap_output;

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



   // duck hunt system display data and led

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

      assign duck_hunt_disp_data =

      {12'd0,state, 2'd0,gamestate, score_val, 1'b0,

      duck_count, 2'b00, duck_shots,2'b0, bush_select,

            1'b0,disappear_count,2'd0,appear_count,2'd0,r_interval};


   assign duck_hunt_led =  ~{3'd0,shudder_timer_expire_debug,

                                 offscreen,second,switch[1:0]};
```

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



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


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


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


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

   // Hex display stuff

   display_16hex hexdisp1(reset, clock_27mhz_buf, hex_data,
                  disp_blank, disp_clock, disp_rs, disp_ce_b,
                  disp_reset_b, disp_data_out);



   // VGA stuff
   wire     hsync_delayed7, vsync_delayed7, blank_delayed7;
   wire  hsycn_delayed1, vsync_delayed1, blank_delayed1;


   // Delay signals by 2 (zbt delay) + 5 (video priority encoder delay)
   delayN dn1(clock_40mhz, hsync, hsync_delayed7);
   delayN dn2(clock_40mhz, vsync, vsync_delayed7);
   delayN dn3(clock_40mhz, blank, blank_delayed7);

   // Delay signals by 1 (BRAM delay)
```

```
   delay dn4(clock_40mhz, hsync, hsync_delayed1);

   defparam dn4.WIDTH = 1;
   delay dn5(clock_40mhz, vsync, vsync_delayed1);

   defparam dn5.WIDTH = 1;
   delay dn6(clock_40mhz, blank, blank_delayed1);
   defparam dn6.WIDTH = 1;


   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.

   assign vga_out_red = duck_hunt_enable ? rgb[23:16] : vga_out[23:16];
   assign vga_out_green = duck_hunt_enable ? rgb[15:8] : vga_out[15:8];
   assign vga_out_blue = duck_hunt_enable ? rgb[7:0] : vga_out[7:0];

//   assign vga_out_red = vram_read_data[23:16] ;
//   assign vga_out_green = vram_read_data[15:8] ;
//   assign vga_out_blue = vram_read_data[7:0] ;


   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = duck_hunt_enable ? ~blank_delayed1 :
~blank_delayed7;
   assign vga_out_hsync = duck_hunt_enable ? hsync_delayed1 :
hsync_delayed7;
   assign vga_out_vsync = duck_hunt_enable ? vsync_delayed1 :
vsync_delayed7;
   assign vga_out_pixel_clock = ~clock_40mhz;

//   assign vga_out_blank_b = ~b;
//   assign vga_out_hsync = hs;
//   assign vga_out_vsync = vs;



endmodule

////////////////////////////////////////////////////////////////////////
////////
// svga: Generate SVGA display signals (800x600 @ 60Hz)

module svga(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); // 799
//   assign    hsyncon = (hcount == 1047);      // 839
//   assign    hsyncoff = (hcount == 1183);     // 967
//   assign    hreset = (hcount == 1343); // 1055
//
//   // vertical: 806 lines total
//   // display 768 lines
//   wire      vsyncon,vsyncoff,vreset,vblankon;
//   assign    vblankon = hreset & (vcount == 767);   // 599
//   assign    vsyncon = hreset & (vcount == 776);    // 604
//   assign    vsyncoff = hreset & (vcount == 782);  // 608
//   assign    vreset = hreset & (vcount == 805);           // 631

   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 799); // 799
   assign    hsyncon = (hcount == 839);   // 839
   assign    hsyncoff = (hcount == 967);  // 967
   assign    hreset = (hcount == 1055);   // 1055

   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 599);     // 599
   assign    vsyncon = hreset & (vcount == 600);      // 604
   assign    vsyncoff = hreset & (vcount == 604);  // 608
   assign    vreset = hreset & (vcount == 627);       // 631

   // 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
//////////////////////////////////////////////////////////////////////
//////
// parameterized delay line

module delayN(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 2+5;

   reg [NDELAY-1:0] shiftreg;
   wire         out = shiftreg[NDELAY-1];

   always @(posedge clk)
```

```
     shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule // delayN


//   always@(posedge clk2)
//   begin
//      case(cswitch[4:2])
//         3'b000:
//         begin
//            vram_write_addr <= box_gen_addr;
//            vram_write_data <= box_gen_data;
//            vram_we <= box_gen_we;
//         end
//         3'b001:
//         begin
//            vram_write_addr <= sync_pc_ram_addr;
//            vram_write_data <= {12'h0, sync_pc_ycrcb[29:22],
sync_pc_ycrcb[29:22], sync_pc_ycrcb[29:22]};
//            vram_we <= sync_pc_dv;
//         end
//         3'b010:
//         begin
//            vram_write_addr <= sync_cc_ram_addr;
//            vram_write_data <= {12'h0, sync_cc_r, sync_cc_g,
sync_cc_b};
//            vram_we <= sync_cc_dv;
//         end
//         3'b011:
//         begin
//            vram_write_addr <= sync_cc_ram_addr;
//            vram_write_data <= {12'h0, sync_cc_r, 16'h0};
//            vram_we <= sync_cc_dv;
//         end
////         3'b100:
////         begin
////            data_to_write <= {20'h0, sync_cc_g, 8'h0};
////            write_x <= sync_cc_x;
////            write_y <= sync_cc_y;
////            we <= sync_cc_x < 11'd725 && sync_cc_y < 10'd525 &&
sync_cc_dv;
////
////         end
//         3'b111:
//         begin
//            vram_write_addr <= sync_threshold_ram_addr;
//            vram_write_data <= {28'h0, sync_threshold_output};
//            vram_we <= sync_threshold_dv;
//         end
//         default:
//         begin
//            vram_write_addr <= 19'h0;
//            vram_write_data <= 36'h0;
//            vram_we <= 0;
//         end
//      endcase
//   end
```