Virtual Juggling

Chris Wilkens and David Rush
6.111
December 14, 2005

Abstract:

The Virtual Juggling Simulator is designed to allow a user to juggle virtual balls by moving their hands in the air. A camera is focused on a user whose hands are found by using a threshold for the input color. The hand's position, velocity and state (based on past and current velocities) are calculated and passed on to a Ball Manager. The Ball Manager determines each ball's new position, velocity and state determined on by their old states and the current inputs. The Ball Manager then passes on to the Output module if and where to display each of the 31 balls on the next frame and the balls are overlaid onto the raw image passed from the camera and displayed on a monitor. The balls' state includes if it is in play, in a hand, in the right hand or left hand, its direction and what its x and y velocity and position are. The hands can be in a throw or catch state. Button and switch inputs are wired from the labkit to reset system, add to or remove balls from the system and to set the gravity for the system. The functionalities of the system are controlled by the major Video Processor, Ball Manager, Output and Controller modules each with a number of submodules.

Table of Contents

List of Figures

Overview

The Juggling Simulator is designed so that a user can juggle virtual balls on a screen by moving their hands in space. The user stands in front of a camera with their hands exposed and they are found by taking the center of mass of the red pixels within the screen. The video assumes that there is one hand on each side of the screen so it finds the center of mass of the red pixels for each side of the screen and displays a small green square where it finds each hand so that their locations can be seen by the user. The pixels that are considered red are found by setting a threshold value to compare to the incoming pixels. Flesh comes up as red as do most warm colors so it is important to have a dark background and all flesh but the hands covered to reduce noise within the system.

As the hands move around their positions are updated each frame which is 60 times per second. From their current position and past positions their velocity and acceleration are calculated and from that throw and catch signals are generated for the hands based on empirically found threshold values. If the hand has accelerated upwards and decelerated beyond a threshold value it is said to be in a throw state for one frame and will release one ball from the hand at its current position and velocity if is holding a ball. If the hand is not moving up it is said to be in a catch state and will catch a ball if the ball is moving down and is in the proximity of the hand. The number of balls a hand can catch is only limited by the number of balls in the system.

Once a ball has been released from the hand it is governed by the laws of virtual physics. The new ball position is calculated each frame by using its current velocity. Each frame the x velocity remains constant unless the ball collides with a side edge of the screen in which case it bounces off with one half of the impact x velocity. If the ball hits the bottom edge of the screen it rebounds with the same y velocity in the opposite direction. If the ball collides the top edge of the screen the new y velocity is set to one pixel per frame in the downward direction. Each frame the y velocity is updated by subtracting the current gravity. This gives the ball acceleration in the downward direction.

The user can modify the juggling experience by pressing buttons and setting switches on the labkit. If the user presses the enter button the system is set to its default setting with three balls in the system- two in the right hand and one in the left. Balls can be added or removed from the system by pressing the up and down buttons. The minimum number of balls is zero and the maximum is 31. When added the balls are placed alternately in the right and left hands. Switches five through zero set the gravity for the system with the high order switches corresponding to the more significant bits. When switch seven is on all the pixels that are determined to be red are displayed as red on the screen otherwise the video output is simply that of the input. With this feature it is easy to identify sources of noise and more easily see where the hand is in relation to the screen. When on, switch six tints each half the screen based on what state each hand is in. If the half of the screen is tinted blue the hand is in a catch state, if gray it is neither in a catch or throw state and when it is tinted green it is in a throw state.

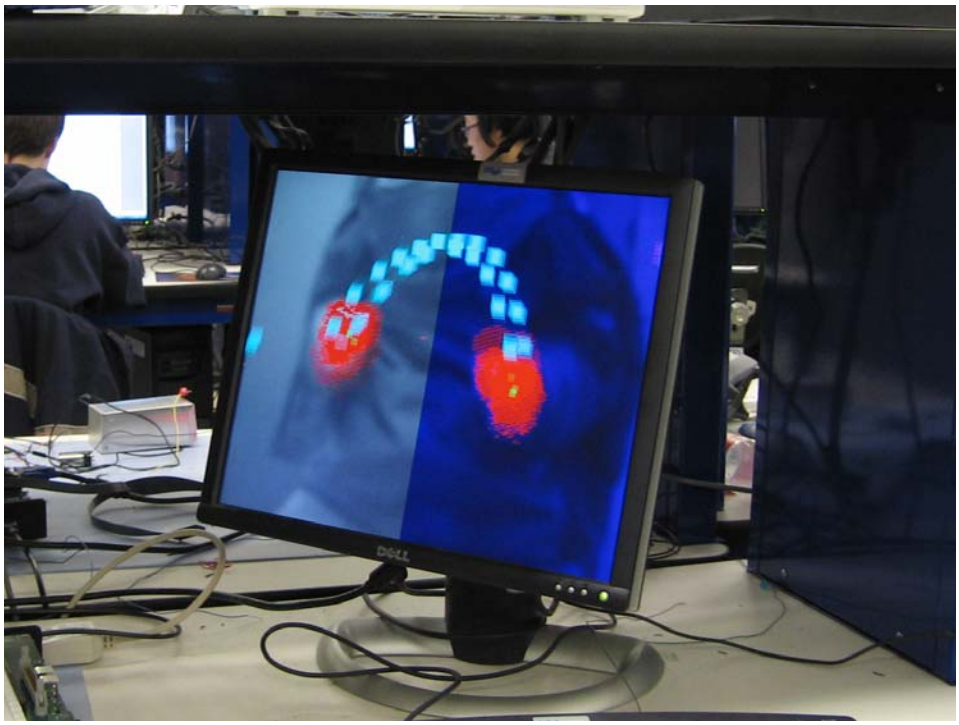

Figure 1: Screenshot of the Simulator

This shot shows David juggling 32 balls in the simulator. One can clearly identify his hands (red) and the 32 cyan balls. Based on the tinting of the screen, one can see that the left hand is not in a catch state, while the right hand is.

Module Structure

The main module of the simulator is the VIRTUAL_JUGGLING module. This model receives the VGA signal of the camera image and performs all necessary processing before the signal is sent to the output. To accomplish this, it contains instantiations of the major blocks of the system.

Within this module are many wires that will henceforth be grouped for simplicity. These wires come in three basic flavors: video signals, 32-bit vectors, and narrow logic signals. The video signals are composed of three 8-bit color channels, four single-bit control signals (sync, blank, hsync, vsync), and 11 and 10 bit *hcount* and *vcount* signals. In general, all position and velocity measurements are carried as 32-bit vectors in which the top 16 bits code the ‘x’ value and the bottom 16 bits code the ‘y’ value. Each of these 16 bit values is encoded as a signed, two’s-complement value with 5 fractional bits. Finally, there are a few miscellaneous signals that represent their own unique codes. The major modules of the system are shown below.

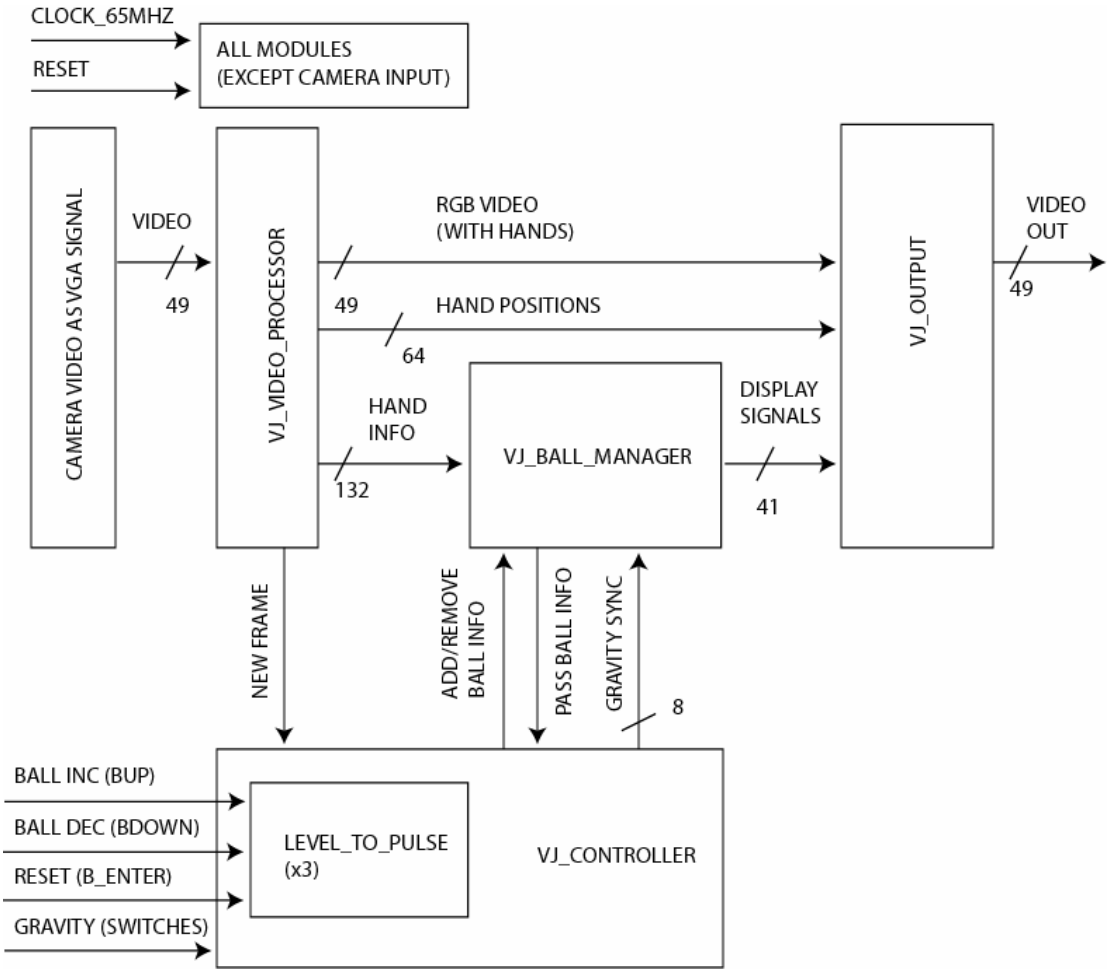

Figure 2: System Block Diagram

This diagram shows the major modules of the system.

Camera Input Module

(by 6.111 staff and Chris Wilkens)

The first block in the video pipeline translates the camera's input signal into a VGA signal. It decodes NTSC video from the camera and buffers it to the ZBT memory, from which it is read out and formed into a VGA signal. This implementation uses the ZBT buffering and output provided by the 6.111 staff with a few small modifications. First, on the output end we downsample the image such that a pixel from the camera occupies four pixels of the 1024x768 XGA display. This allows us to fill the entire output frame with video from the camera. One advantageous result of this fact is that we need not read from the ZBT frame buffer as frequently, which enables the second modification. In order to do color comparisons, we modify the existing implementation to buffer 18 bits of information for each pixel. This gives us 8 bits of the Y and Cr values and 2 bits of the Cb (the Cb is not necessary for our color detection.) Since the Video Processor does color detection based on the Y and Cr signals, we keep these signals intact instead of producing reasonable RGB values at the output of this module. Once this

module has generated the proper video signals, they are subsequently sent to the Video Processor.

Controller

(by David Rush)

The Controller module takes in a number of signals from the labkit and the video processor and outputs signals synchronized to *new_frame_sync* that can be used by the ball manager.

The inputs to the Controller are *clk*, *new_frame_raw*, *reset_raw*, *add_ball_raw*, *dec_ball_raw* and *gravity_raw*. The clock runs at 65MHZ and comes from the labkit. The one bit *new_frame_raw* signal comes from the Video Processing module and signals that a new frame has started. The *reset_raw* is a one bit inverted (so pressed is high and depressed is low) and debounced button from the labkit as are the *add_ball_raw* and *dec_ball_raw* signals. The *gravity_raw* is an eight bit signal from the debounced switches on the labkit.

The outputs of the Controller are *new_fram_sync*, *reset_sync*, *add_ball_sync*, *dec_ball_sync* and *gravity_sync*. They are each one bit except for the eight bit *gravity_sync* signal and are all sent into the Ball Manager to be used by the Physics module and Display module.

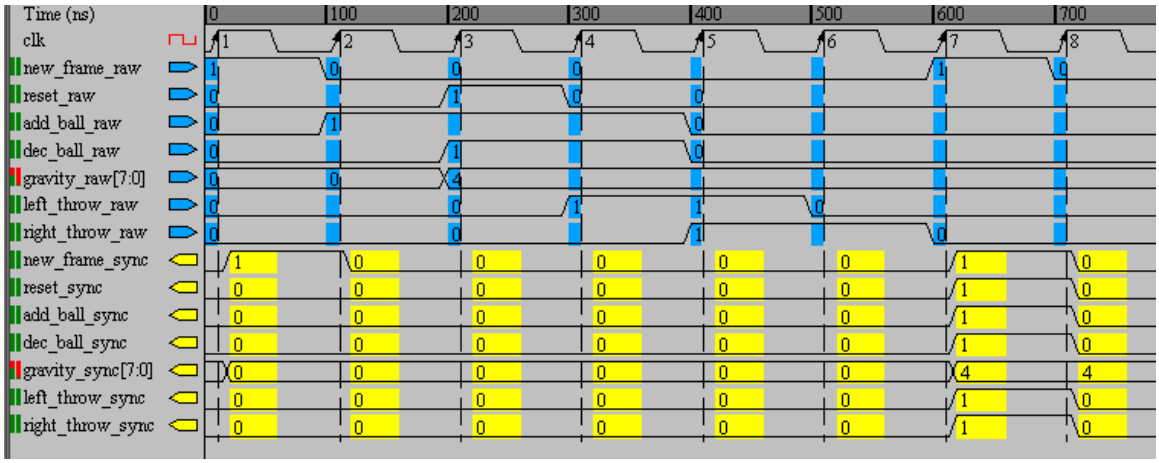

Figure 3: Controller Simulation Results

The controller receives signals at different times during the frame and synchronizes all outputs with *new_frame_sync*

Within the Controller each of the output signals are created using the input signals. Within an always block that triggers at each positive edge of the 65 MHZ clock a number of signals are assigned using non blocking assignments. The *new_frame_sync* is assigned to *new_frame_raw* and *gravity_sync* is assigned to *gravity_raw* if *new_frame_raw* or it remains *gravity_sync*. The *reset_sync* signal is set to one if *new_frame_raw* and *reset_hold* are high otherwise zero. The *reset_hold* signal is assigned to one if *reset_raw* is high otherwise it is set to zero if *new_frame_raw* is high otherwise it is left *reset_hold*. The process creates a *reset_sync* signal that is set to one

for one clock cycle synchronized with *new_frame_sync* if the reset button was pressed any time during the preceding frame. The same process was used to create the *add_ball_sync* and *dec_ball_sync* signals except the _hold signals were replaced with *add_ball_hold* and *dec_ball_hold*. Also for the assignment of *add_ball_hold* and *dec_ball_hold* the analogous *reset_raw* signal was replaced with an *add_ball_pulse* and a *dec_ball_pulse* which came from sending the *add_ball_raw* and *dec_ball_raw* signals into their own Level to Pulse module instances to return the *add_ball_pulse* and *dec_ball_pulse* signals that are high for only 1 clock cycle regardless of how long the buttons were held down. Registers were created to hold values for each of the signals that were assigned within the always block.

Level to Pulse

The level to pulse takes in an input that is high for any number of cycles and outputs a signal that is high for only the first clock edge after the input signal went high.

The inputs are *clk* and *level_in* and the output is *pulse_out*. At each positive edge of the clock *level_hold* is assigned to *level_in* and pulse out is assigned to the combination of *level_in* and not *level_hold*. A register is used to store each *level_hold* and *pulse_out*.

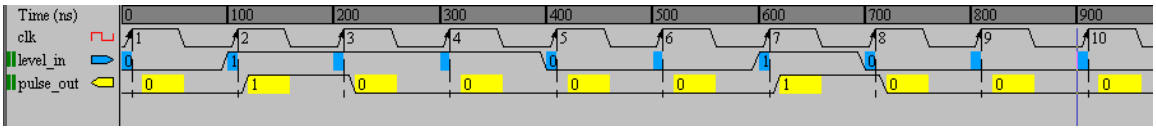

Figure 4: Level to Pulse simulation results

The level to pulse receives a level input and returns a one clock pulse

Video Processor

(by Chris Wilkens)

The VJ_VIDEO_PROCESSOR module processes the VGA signal from the camera modules and determines the location and action of the juggler's hands. It determines the position and velocity information and forwards them to the appropriate modules. When the positions are ready for a given frame, it raises the *new_frame* signal to notify the ball manager that it can begin its physics calculations.

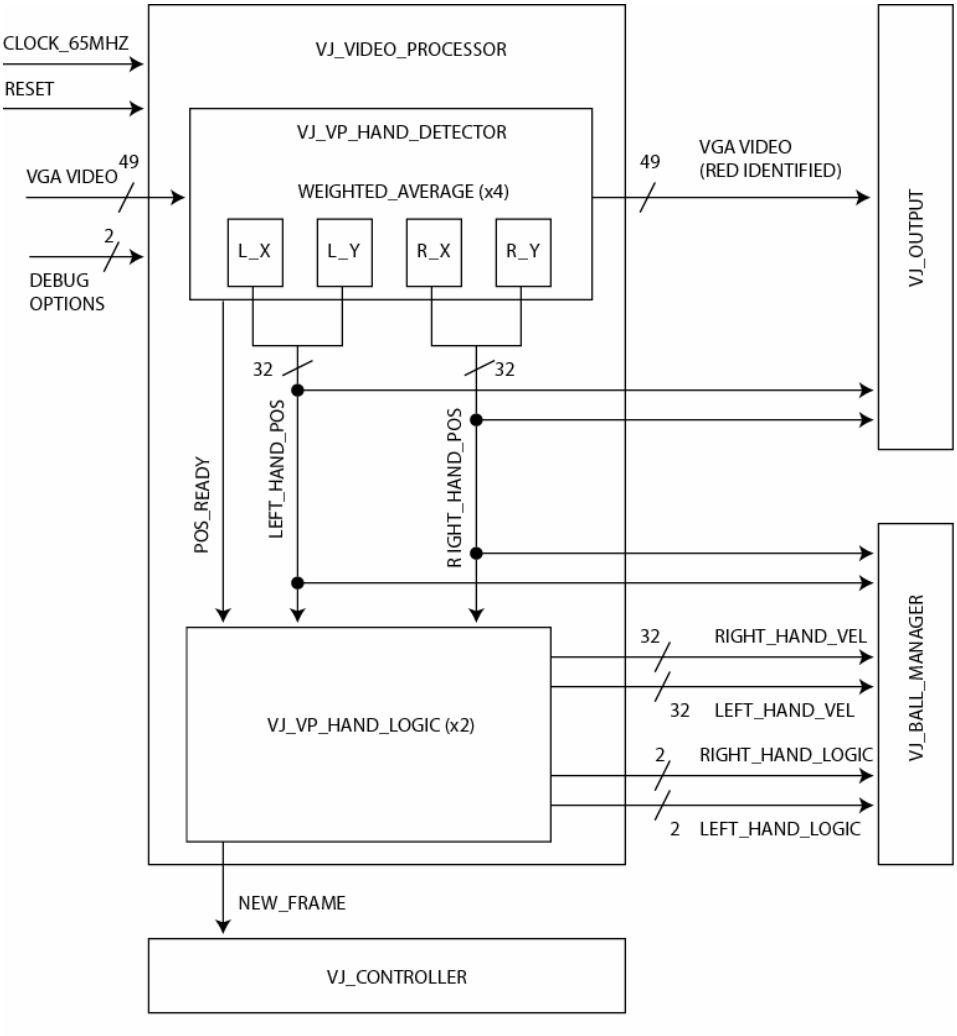

Figure 5: Video Processor Block Diagram
This diagram shows the modules of the Video Processor.

As seen in the block diagram, the specific implementation of the video processor is composed of two major parts: the hand detector, which locates the hands in the image, and the logic parser, which translates a series of positions into action signals that can be sent to the ball manager. In addition to providing the proper signals to these modules, the video processor modifies the output by tinting the flesh-colored pixels red. Finally, the *new_frame* signal is simply the AND of the *logic_ready* signals from the two hand logic modules.

Hand Detector

The hand detector module of the video processor watches the streaming VGA signal to calculate the location of the hands. Abstracted as a black box, this module takes the VGA color signal as input and returns the positions of the hands in the frame when the frame has been completely streamed. For each pixel, it also raises the *hand_pixel*

signal whenever the current pixel belongs to the hand (to allow the video processor to change its color.) In order to signal other components of the video processor, a *pos_ready* pulse is raised for one clock cycle when the hand positions are ready.

For our implementation, we define the position of the left hand to be the center of mass of all flesh-colored pixels in the left half of the screen (or right half for the right hand.) Each coordinate of each hand is calculated by keeping a running sum of the appropriate coordinate values along with a count of the number of elements in the sum. Once the frame has passed, these two numbers in the sums are divided and the result becomes the appropriate coordinate of the hand. This is accomplished with four instances of a weighted average module (one for each coordinate to be calculated.) Due to timing delays through the divide module, it takes 32 cycles after the end of the frame in order to perform the division. Thus, in order to ensure that the process has ample time to complete, we wait many clock cycles until the vertical scan reaches line 770 before raising the *pos_ready* signal. The output position is registered when *pos_ready* is raised, and the internal sums are reset on the vertical sync signal.

Hand Logic

Two instances of the Hand Logic module sit beside the hand detector in the video processor. They watch the hand positions to determine whether or not the hand is throwing or catching. Abstracted, each module takes the new position and *pos_ready* signal and produces the hand velocity and logic signals. The hand logic signals consist of two bits, a catch bit and a throw bit (*hand_logic*[1] is throw, *hand_logic*[0] is catch.) The catch bit will be high whenever the hand is in a state in which the system deems that the hand could be catching, leaving the details of the catch timing up to the ball manager. In contrast, the throw signal is a pulse held high for only a single frame. When these values are computed, the module raises a *logic_ready* signal to notify others that the logic has been calculated.

This is, perhaps, the most interesting part of the video processor because it has the most room for modifications. There are an infinite number of ways one could determine a throw or a catch, but we chose a fairly simple implementation. First, we decided that if the hand was moving up, then it wasn't catching a ball. This seemed reasonable, though we realized that it isn't entirely true in real life. That said, we set a threshold (slightly positive to accommodate noise) for which the hand would be considered catching if its velocity was under that value. Another aspect we noted was that the positions tended to fluctuate due to noise. To compensate for this, we maintained a history of the last four velocities and took the average in order to compute the "actual" velocity of the juggler's hand.

The throw signal was more complicated. If we assumed that the juggler was not holding the balls, then a throw would occur by conservation of momentum whenever the hand decelerated. Thus, we decided to locate throws based on the acceleration of the hand. As with the velocity, we buffered the last four accelerations to get an average and then put a threshold on that average. Our final dilemma was that the throws would tend to "bounce," just like a metal button. For this, we effectively debounced the throw signal by specifying that the hand couldn't throw again until it had been out of the acceleration

region for six consecutive frames. This effectively ensured that we only signaled one throw per movement from the juggler, and the appropriate output signal was raised for a single frame.

Ultimately, these values required one clock cycle to produce, so the hand logic module introduces a one cycle delay after the *pos_ready* signal before all the hand information is ready.

Ball Manager

The Ball Manager takes in signals from the Video Processor and Controller and based on those signals and the balls' current states determines what to do next with the balls and then it sends to the output module if and where to display the balls. The ball manager is composed of three submodules and a dual port 70 bit by 32 address Block RAM (BRAM).

The inputs to the Ball Manager are a 65 MHZ clock from the labkit, a reset, an *add_ball*, a *dec_ball*, and an eight bit gravity signal from the Controller all synchronized to a *new_frame* signal which also comes from the controller. There are also signals from the video processor which are *left_throw, right_throw, l_h_catch, r_h_catch, l_h_y, l_h_x, l_h_y_vel, l_h_x_vel, r_h_y, r_h_x, r_h_y_vel,* and *r_h_x_vel*. The clock determines the cycles per second of the system. The one bit reset signal returns the ball manager to its default state. The one bit *add_ball* and *dec_ball* signals add and remove balls from the system with a minimum of zero and a maximum of 31 balls. The eight bit gravity signal determines how fast to balls accelerate in the downward direction. The one bit *new_frame* signals to the system that a new frame is starting and all the balls' states need to be recalculated. The *left_throw, right_throw, l_h_catch* and *r_h_catch* are each one bit signals indicating that the left and right hands are in a throw or catch state for that frame. The signals *l_h_y, l_h_x, l_h_y_vel, l_h_x_vel, r_h_y, r_h_x, r_h_y_vel*, and *r_h_x_vel* are each 16 bit signed values and give the left and right hand's y position, x position, y velocity and x velocity with new values being supplied each frame.

The outputs of the Ball Manager are all sent to the output module and are *display_enb, ball_pos_out, ball_number,* and *write_request*. The one bit *display_enb* signal indicates that the ball for which information is currently being sent is to be displayed. The *ball_pos_out* is a 32 bit signals with the x position in the upper 16 bits and the y position in the lower 16 bits. The *ball_number* is a 5 bit signal indication which ball's information is currently being sent and the one bit *write_request* indicates that the ball manager wants the information it is sending to be taken by the output module. The output module can be thought of as a memory with the *write_request* signal equivalent to the write enable and the other signals the values that are to be stored in the memory.

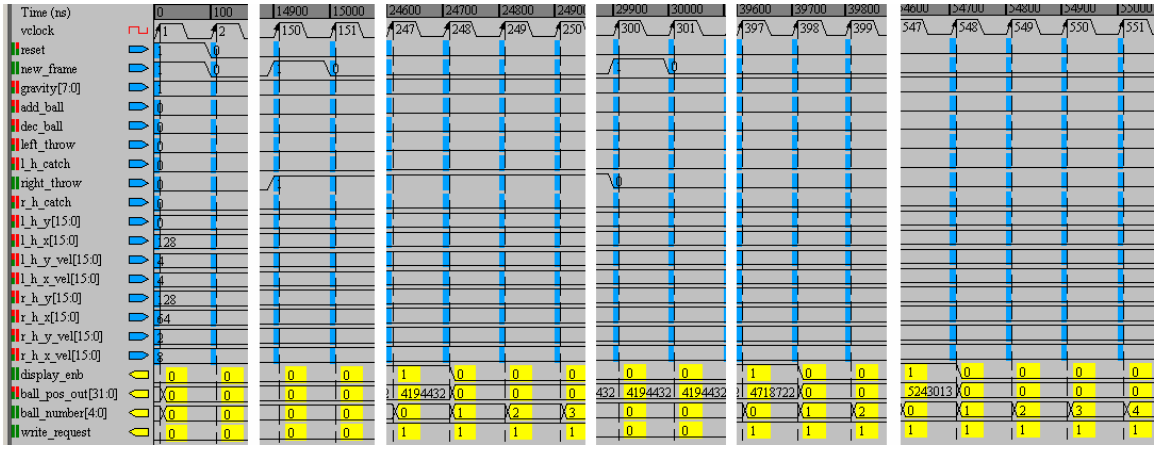

Figure 6: Ball Manager Simulation Results

The ball manager takes in all its inputs and sends values to the output module at the appropriate times. After a throw between clock cycles 150 and 300 the outputs *write_request* as well as display enable and a ball position are given at clock number 397 for ball number zero since it was thrown from the right hand and is in the air and thus needs to be displayed by the display module

Within the Ball Manager the Physics and Display modules are instantiated and wired together. The *clock*, *reset*, *add_ball*, *dec_ball*, *gravity*, *new_frame*, *left_throw*, *right_throw*, *l_h_catch*, *r_h_catch*, *l_h_y*, *l_h_x*, *l_h_y_vel*, *l_h_x_vel*, *r_h_y*, *r_h_x*, *r_h_y_vel*, and *r_h_x_vel* signals are all wire directly to up as inputs to the Physics module. The 70 bit *ball_info_in* from the output of the BRAM is also wired as an input of the Physics module. The outputs of the Physics module are the 70 bit *ball_info_out*, five bit *count_addr* and one bit *wea*, which are wired up to the BRAM, and the one bit *done_calc* which is sent as an input to the Display module.

The inputs of the Display module are wired up to the 65 MHZ clock. The one bit reset and *new_frame* are taken directly from the inputs to the Ball Manager and the one bit *done_calc* is wired from the output of the Physics module and 70 bit *info_from_bram* is wired to the b port of the dual port BRAM. The outputs of the Display module are the five bit *bram_addr_b* which is wired to the "b" port address of the BRAM, the 32 bit *ball_pos_out* which is wire directly to the Ball Manager output as are the one bit *display_enb*, five bit *ball_number*, and one bit *write_request*.

The 70 bit wide by 32 address deep Block RAM (BRAM) is wired within the Ball Manager as well. Both the "a" and "b" clocks are wired to the 65 MHZ clock. The a port address is wired to five bit *count_addr* from the Physics module and the 70 bit data input is wired to the *ball_info_out* (out of the Physics module). The 70 bit "a" output is wired to *ball_info_out* and sent to the Physics module. The five bit "b" address is wired to the *bram_addr_b* from the Display module and the 70 bit "b" output is wired to the *info_from_bram* that is given to the Display module.

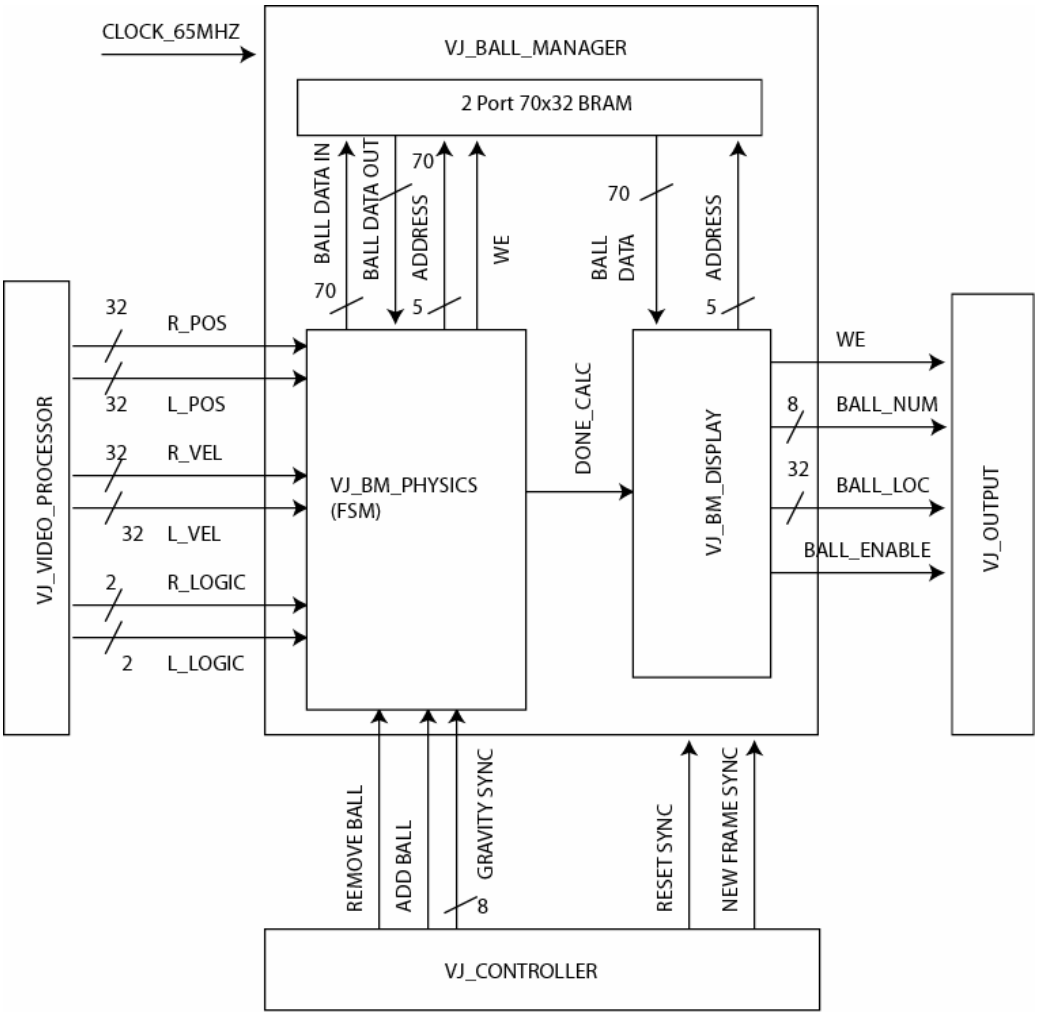

Figure 7: Ball Manager Block Diagram

This diagram shows the smaller modules of the ball manager.

Block Random Access Memory (BRAM)

The dual port Block Random Access Memory stores the state of each ball in its 70 bit wide by 32 location memory. The "a" port is read/write and is used by the Physics module to retrieve ball data from the BRAM and store the new ball data back into it. The "b" port is read only and is used by the Display module to take the ball data and extract the current position and state and send it onto the output module.

Physics

The Physics module is the biggest and most complex part of the ball controller. The calculations for each ball's position, velocity and state are done within this module and results are stored in a dual port 70 bit by 32 location BRAM. The Physics module takes in inputs from the Controller, the Video Processor and the BRAM and it outputs

data to the BRAM and a signal indicating the calculations for that frame are done. The Controller inputs (wired in through the Ball Manager) control whether the system should be reset, a new frame should be processed whether a ball should be added or removed from the system and what gravity is. Video Processor inputs (also wired in through the Ball Manager) give the left and right hand's x and y positions and velocities as well as the left and right hand throw and catch signals. The input from the BRAM gives each ball's current state, position and velocity. The outputs determine if, what and where in the BRAM the ball data should be stored as well as indicate if the Physics module is done with its calculations and storage for the frame.

The inputs to the Physics module are a 65 MHZ clock from the labkit, a *reset*, an *add_ball*, a *dec_ball*, and an eight bit *gravity* signal from the Controller all synchronized to a *new_frame* signal which also comes from the controller. There are also signals from the video processor which are *left_throw, right_throw, l_h_catch, r_h_catch, l_h_y, l_h_x, l_h_y_vel, l_h_x_vel, r_h_y, r_h_x, r_h_y_vel,* and *r_h_x_vel*. The final input is *ball_info_in* from the BRAM. The clock determines the cycles per second of the system. The one bit *reset* signal returns the Physics module to its default state. The one bit *add_ball* and *dec_ball* signals add and remove balls from the system with a minimum of zero and a maximum of 31 balls. The eight bit *gravity* signal determines how fast to balls accelerate in the downward direction. The one bit *new_frame* signals to the system that a new frame is starting and all the balls' states need to be recalculated. The *left_throw, right_throw, l_h_catch* and *r_h_catch* are each one bit signals indicating that the left and right hands are in a throw or catch state for that frame. The signals *l_h_y, l_h_x, l_h_y_vel, l_h_x_vel, r_h_y, r_h_x, r_h_y_vel,* and *r_h_x_vel* are each 16 bit signed values and give the left and right hand's y position, x position, y velocity and x velocity with new values being supplied each frame. The 70 bit *ball_info_in* from the BRAM is contains each balls' state, position and velocity and is used to determine the new state, position and velocity in combination with the inputs.

The outputs for the Physics module are *ball_info_out, count_addr, done_calc* and *wea*. The 70 bit *ball_info_out* signal is the new position, velocity and state of each ball and the *count_addr* is the current ball that is being updated and written back into memory at the address corresponding to it's ball number. The *done_calc* signal is a one bit signal to indicate to the display module that all the physics for the frame has been calculated. The *wea* is the write enable for the BRAM.

A number of parameters were used to simplify expressions and keep the code from being cluttered with numbers. They are *ball_w, ball_h, hand_h, hand_w, screenw, screenh, ball_right, ball_left, ball_down, ball_up, read_state* and *write_state*. The first six are just the sizes of the balls, hands and screen and can be changed to optimize the system. Bigger hands made it easier to catch. Bigger balls meant they hit walls sooner an different screen sizes would have affected the display. The *ball_* parameters are the states of the balls when moving around when in play and not in the hands. The *_states* signal if the system is in a read state (reading from BRAM) or write state (writing to BRAM).

The first thing the system does is parse the incoming 70 bit *ball_info_in* and assign *in_play_in, in_hand_in, in_left_in, in_right_in, y_dir_in, x_dir_in, ball_y_in, ball_x_in, y_vel_in,* and *x_vel_in* based on those signals. The first six signals are each one bit and are stored in the high six bits of the incoming signal. The remaining signals

are each 16 bits and are stored in the given order within the *ball_info_in.* The Physics module takes these signals as input and creates new values for each *in_play_out, in_hand_out* and so on and appends them together in a 70 bit *ball_info_out* that is stored back into the BRAM for each ball.

On the positive edge of the 65 MHZ clock the always block is evaluated. If there is a *new_frame, done_calc, count_addr, wea* and *read_hold* are set to zero. The *read_hold* signal holds the system in the read state for two cycles so the data is retrieved and assigned to the ball input values before going into the write state. On *new_frame count_divider* is set to three. Count divider is a signal that counts to 2 and each time it reaches two it is set back to zero and the *count_addr* is incremented (each ball takes three clock cycles in the Physics module and this ensures that each ball gets three cycles). Setting the *count_divider* to three here means that when the first time it incremented after a reset it will return to the zero count that it needs to start with after its initial increment. Also on the *new_frame, reset_level* is set to the *reset* input, *left_throw_level* is set to *left_throw*, *right_throw_level* is set to *right_throw*, *add_ball_level* is set to *add_ball,* and *dec_ball_level* is set to *dec_ball,* this ensures that each of those signals values are valid for the entire frame. The *which_hand* is also flipped if the *add_ball* signal is high. This signal determines which hand each new ball is placed into (alternating each time a ball is added). If it's not a *new_frame* then if the *count_addr* signal has reached 31 then the *done_calc* signal is set high and no more physics is calculated that frame. Else if *done_calc* is not high the *count_divider* is incremented or set back to zero if it equals two, the *count_addr* is incremented if *count_divider* is equal to two or it remains unchanged and the *read_or_write* case statement is evauated. If the system is in the *read_state* the *read_hold* bit is flipped, *wea* is set to zero and *read_or_write* is set to *read_hold.* This means that after two cycles in the read state it will enter the write state.

In the write state the state *read_or_write* is set back to the *read_state* and *wea* is set to one since at the rising edge of the next clock there will be new data ready to write into the BRAM.

The rest of the write state within the Physics module is a large if else block that assigns the output values that are put assigned into *ball_info_out* and stored back into the BRAM. The overview is that if the system is being reset put balls zero and one in the right hand and ball two in the left and leave the rest out of the system. If the ball is in play then if the *dec_ball_level* is high take the ball out of play and set *dec_ball_level* back to zero so no more balls get removed. Otherwise if the ball is in a hand if it's in the right hand, the right hand is in a throw state and a fall hasn't already been thrown (*right_thrown* signal is low) then have the ball in play, don't have it in a hand, assign it the hand's current position, velocity and direction and raise the *right_thrown* signal so no more balls are thrown this frame out of the right hand. The analogous case is true for the left hand. Otherwise if the ball is in a hand and no hand is throwing, assign all the outputs to the original inputs since nothing is changing for them this frame.

If the ball is not in a hand but is in play and is not removed leave it in play and check to see if it has been caught. It is caught if the right or left hand is in a catch state, the y direction of the ball is down and the ball is within a given rectangle area above the center of the hand that can be set by the hand width and length. It is placed in the hand that it is caught in if it is caught otherwise the new position velocity and direction of ball are

calculated. In the in play, not removed, not caught state the x and y velocities and positions are calculated separately depending on which direction the ball is moving. If the *x_dir_in* is *ball_right* the balls x postion if it's input position plus it's input velocity. The new x velocity remains the same unless it comes within a distance of two times the velocity of the right wall and then it is halved and the x direction is also switched. If the ball is moving left the new *ball_x_out* is the *ball_x_in* minus the *x_vel_in* and the velocity and direction remain the same unless the ball is within two times the *x_vel_in* in which case the x direction is flipped and the velocity is halved. The y direction and velocity are done similarly except if the ball is moving down the balls position is the old position plus the velocity and the direction is the same unless it comes within two times the y velocity of the bottom of the screen in which case the direction changes and the velocity does not. Otherwise the velocity is updated by adding gravity to it each frame. If the ball is moving up the new position is the input position minus the input velocity. The direction remains the same unless the gravity is greater than the current velocity or the ball comes within two times the velocity of the top of the screen. In the later case the velocity is set to one pixel per frame. The velocity in this state is updated by subtracting gravity from the input velocity.

If the ball is not in play then if a ball is added (*add_ball_level*) is high the *in_play_out* and *in_hand_out* bits are set high and the *in_left_out* and *in_right_out* are set to *which_hand and ~which_hand* respectively The rest of the outputs. The rest of the output values are set to zero as is *add_ball_level* so no more balls are added that frame. If the ball is not in play and no ball is added then all the output signals are set to zero.

Display

When triggered to do so the Display module reads through the BRAM memory and extracts the balls' positions and logic to send onto the output module along with which ball that data is for so it can know if and where to display each of the balls.

The inputs to the display are the clock, reset, *new_frame*, *done_calc* and *info_from_bram* signals. The clock is 65 MHZ from the *labkit*. The reset is synchronized to *new_frame* and both come from the controller. The *done_calc* signal comes from the Physics module and signals that the Physics has been calculated and stored for this frame and that the Display module can begin extracting the needed data to give to the output module. The *info_from_bram* is a 70 bit signal holding all the information about each ball that the Display module parses and sends to the output module.

The outputs are *bram_addr_b*, *ball_pos_out*, *display_enb*, *ball_number* and *write_request*. The five bit *bram_addr_b* is the address the data is going to come from out of the BRAM. The *ball_pos_out* is the 32 bit signal with the x position of the current ball stored in the high 16 bits and the y position in the low 16 bits. The *diplay_enb* is a one bit signal that determines that particular ball is to be displayed. The five bit *ball_number* is the number corresponding to the ball for which the current position and display logic are being outputted. The *write_request* is a one bit signal sent to the Output module which indicates that the data coming in is valid and should be stored.

Registers were created for each of the outputs and three internally used signals since all are assigned in a procedural block. The three internally used signals are a one bit *stop_calc* that indicates that all 31 of the balls that can be used have been processed and the Display module should wait for a *new_frame* to do anything. The five bit *ball_number_hold* is used to store the *bram_addr_b* signal before it is passed onto *ball_number* to pipeline the process since it takes a clock cycle to retrieve the data and another cycle to do the logic on it. A *write_request_hold* register is used for the same reason.

Within the procedural block a number of if statement may be evaluated. If reset it high *stop_calc*, *bram_addr_b*, *write_request* and *ball_number* are all set to zero since the system is going to be reset and old values in the BRAM don't matter so we'll wait until they've been rewritten during the reset frame before giving the Output module anything to display. The system will wait until it receives a *done_calc* from the Physics module before doing anything. Else if the reset isn't high if *stop_calc* is high the system is in a waiting state for new frame. The *stop_calc* signal is set back to zero if *new_frame* is high otherwise it takes on its old value. The same happens for *bram_addr_b* and *ball_number*. The *write_request* is set to zero. Else if neither reset or stop calc then if *done_calc* is high (signaling the Physics module is completed with calculations and writing into memory for the frame) the following happens: *bram_addr_b* is incremented, *stop_calc* is set to one if *ball_number* equals 31 (which means all 31 balls have been processed and their data sent on) otherwise it stays zero, *write_request_hold* is set to 0 if *ball_number* equals 30 (which means all 31 balls will have been processed and their data sent on…not 32 balls due to pipelining) otherwise one, *write_request* is assigned to *write_request_hold*, *ball_pos_out* is assigned to the appended bits 47-32 and 63-48 from *info_from_bram* (where the x and y positions are stored respectively), *display_enb* is set to zero if *ball_number* equals 31 otherwise it is set to the logical value of bits 69 and not 68 or 67 from *info_from_bram* (in play, in left hand and in right hand bits respectively) and *ball_number_hold* is set to *bram_addr_b*.

Output Module

(by Chris Wilkens)

The fourth major block of the implementation is the output module. This module sits in the path of the outgoing VGA signal and overlays the hands and balls. In our implementation, this is accomplished by instantiating a sprite for each object to be displayed. For the two hands, we instantiate two separate directly within the output module and connect them to the position signals from the video processor. However, the situation is slightly more complicated for the 32 balls. From the output module's perspective, these are implemented using a single "aggregate sprite" that can be queried for a pixel location and returns the color based on the location of all 32 balls in the system. This requires a set of connections resembling a memory, which are received from the ball manager's display module and passed directly through to the aggregate ball sprite module.

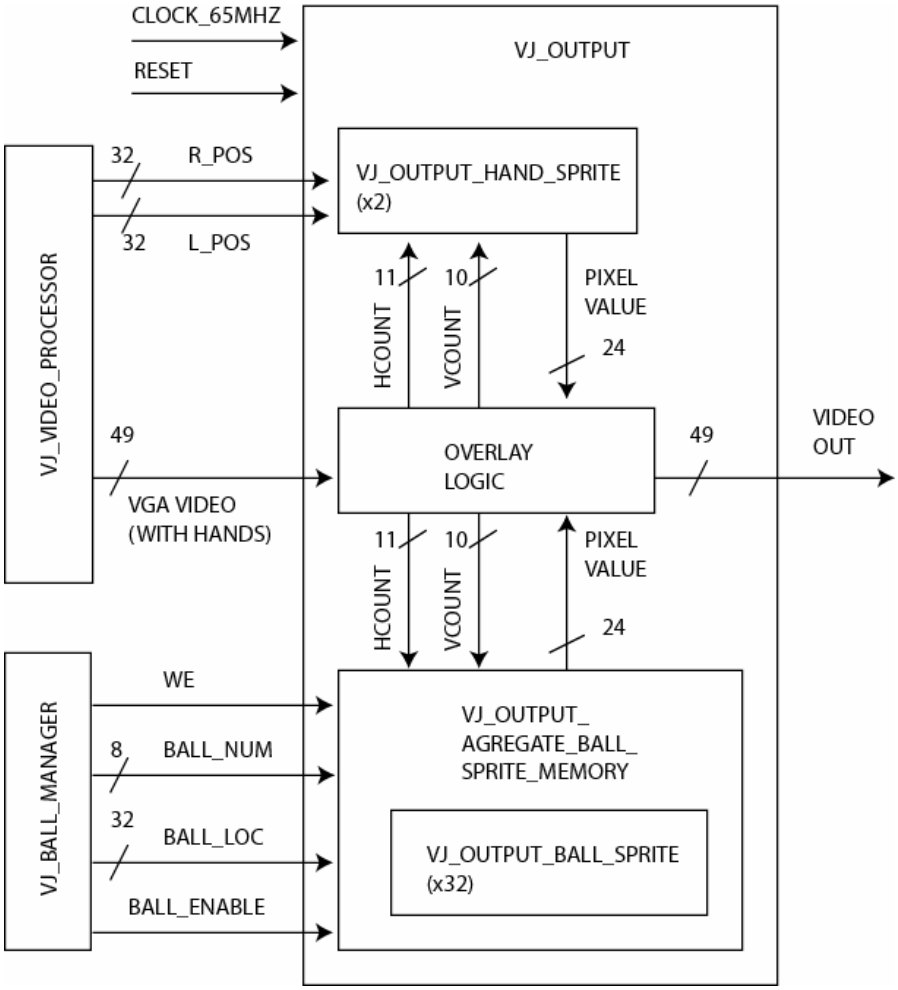

Figure 8: Output Block Diagram

This diagram shows the structure of the output module.

Aggregate Ball Sprite Memory

The ball sprite memory module acts like a memory for the 32 output ball sprites. From the perspective of the output module, it behaves like a single sprite that can be queried for a pixel value. However, it also has a set of “memory” signals that can be used to configure the output. The ball manager uses these signals to sequentially set the positions of the 32 balls. On a given cycle, the ball manager can supply a write enable, a ball number (similar to a memory address,) the position in which the specified ball is to be displayed (similar to memory data,) and whether or not the specified ball is actually to appear on the screen (more memory data.) This information is stored internally such that when the output module queries for a certain location on the screen, the sprite memory can check each ball simultaneously to determine whether or not it claims the given pixel.

We implemented this module as an array of sprite modules. The connections to these sprite modules are large arrays of registers that hold the position and enable signals for the balls. On cycles when the write enable bit is high, this module stores the position and enable signals in the registers for the appropriate ball number. As a consequence, the position of that sprite is updated, such that when a pixel request comes, the module can

simply OR the outputs of each of its 32 sprites, allowing the output module to see it as one sprite.

Testing

Our testing strategy for the project consisted of three facets: independent testing of the video modules in hardware, independent testing of the ball manager in simulation, and combined testing of the modules in hardware. For the video modules, we incrementally tested them in hardware because simulation, in general, would be prohibitively difficult. The first step was to demonstrate that we could buffer and display the proper video. Because we were originally using an older module provided by the staff, this process took time. We did not reach our final design until the staff released a new module that buffered video to the ZBT instead of to the BRAM. At this point, what was supposed to be a simple set of modifications to the existing verilog were confounded by what appeared to be issues computing the address. Once these were fixed, however, the video was ready to go.

The next step, though it was started concurrently with testing video input, was to develop the hand detector module that located the hands on the screen. For this, we first tested thresholds for detecting flesh colored pixels. By connecting our thresholds to the switches of the labkit and coloring the selected pixels red, we were able to tweak the parameters and determine good limits for the Y and Cr channels of the input. With the critical pixels selected, it remained to compute the appropriate centers of mass. This was implemented and debugged by connecting the positions of the hands to sprites in order to display them in the video. This process was marginally complicated because the calculations required many clock cycles, but seemed simple enough. However, it proved to be our first major but silly snag. For some inexplicable reason, the positions were not changing, and it wasn't clear why. Simulation testing revealed that the hand detector worked according to specification, so it was unclear why things appeared to break on synthesis. After an exorbitantly large amount of debugging, it was discovered that the clock signal being passed to the hand detector was not in fact a valid signal at all and that the synthesizer had missed it. Once that was fixed, the module worked as promised.

With the position information, it was necessary to compute the hand logic values. Our first attempt was to compute the velocity of the hand by taking the difference with the last data point. This, however, proved to be a torturous task, as early attempts to subtract the two positions simply yielded zero velocities. After days of debugging, it was finally determined that the source of the error was the *pos_ready* signal. This signal was only supposed to be high for one clock cycle per frame in order to notify the hand logic module to calculate the velocities. Instead, the signal was high for many clock cycles, effectively erasing the memory of the logic calculator. Unfortunately, this took days of debugging to discover. Again, once it was discovered, everything ran smoothly. We then proceeded to hook the throw and catch signals up to simple logic for testing purposes. We also modified the video processor to tint the screen whenever a throw or a catch was detected. This was an extremely simple modification that allowed us to clearly visualize and tune the hand logic. Using this method, we iterated on the original, simple logic to produce more complicated throw and catch algorithms.

Concurrent with development of the video processor was development of the output module. The first stage of testing was to simply wire the hand sprites to the positions generated by the video processor. However, the ball sprites remained untested. For this, we created a test jig that cycled through each ball and modified it for each frame. This allowed us to test the sprite "memory" aspect of the module independent of the ball manager. With this testing complete, the video processor and output modules were ready for integration.

While Chris was developing the video input an output, I (David Rush) was developing the ball manager. The biggest thing I learned during the creation of the juggling simulator is that small modules are much better. Building a big system and trying to test it is a disaster because many things are likely to be wrong. Basic functionality should first be established with each small component before more complexity should be added. The basic inputs should be tested first and only once they are working should more complex inputs be added. I made the mistake of writing a huge Physics module and had no way but to enter a combination of inputs into the simulator and check to see that nothing came out. I ended up wiring every signal I had to an output to check that any of them were correct and try to figure out where along the chain there was a problem. I iterated this process dozens of times because there were dozens of mistakes that were made. Velocities were wired to position outputs, mot all the outputs were assigned in the right places, wires were missing or mismatched, some code didn't do what I thought and many times my timing was off by a clock cycle. After many, many iterations of trying to simulate the Physics module and getting a little more correct each time after hours of debugging I learned that small was better and each of the subsequent modules I wrote were much smaller and easier to debug.

For each of the modules within the Ball Manager testing in simulation was the easiest way to initially see if the desired results were had. The level to pulse module was easiest and it can be seen in it's graph that level signals were put in and pulses came out.

The Controller was similar in that it had a limited number of inputs and the expected outputs could easily be checked against the actual outputs. If the outputs weren't all synchronized it wasn't working right.

Within the display module I (David) found in simulation that my ball addresses weren't matching up with the right data and after drawing out the timing diagrams I realized I needed a two clock cycle delay between sending the input address and having the valid data to give to the output module. I pipelined the process by buffering the ball number and write enable and that way each ball only took one clock cycle even if the first ball data didn't come out for 3 cycles.

After testing in simulation I visually tested the Ball Manager since looking at 32 bits worth of position data can't very well tell you if the ball is being displayed at the right position after many seconds since the frame updates 60 times per second. At first it was noticed that all the balls were in the upper left of the screen and moving only a little bit. After checking the position calculations I noticed that I had wired the output position bits to the velocity which was always much smaller. It was later noticed that one ball was always displayed and never moved and that was due to an off by one error in the display module. The balls were also very jittery at one point and it was found to be because one signal wasn't set for an entire frame when it should have been.

After 100 hours of work there were many more errors and bugs that were worked through and in the end though the system worked. This is just a highlight of some of the more interesting errors.

Conclusion

Overall, this project provided an excellent demonstration of the principles of system design, including modularity and simplicity. In the end, the project worked fairly well. Though it couldn't truly simulate the experience of juggling real balls, it could reliably simulate simple juggling moves. Moreover, when configured properly, it was simple enough that even those who couldn't juggle in real life could juggle in the simulator. The net result was a pleasantly functional system.

One of the most prominent design principles in the project was the supremacy of simplicity in system design. For example, our method of hand detection was hardly revolutionary, and calculating the center of mass of the pixels on each side of the screen was very straightforward. However, despite the lack of complexity, this method of hand detection was extremely reliable when the proper precautions were taken. Another simple yet innovative idea was displaying the hand logic by tinting the screen. The implementation was simple – only one line of verilog code was modified – but it allowed the untrained user to gain a much better understanding of the system's interpretation of his or her movements. It also greatly facilitated debugging, as it provided insight into what the system was "thinking." Yet another innovation of simplicity was the idea to allow the user to fix the release velocities of the balls when they are thrown. While this may significantly detract from the usefulness of the simulator for an experienced user, it greatly enhances playability for novices and allows them to become acclimated to the system before enabling the more complex mode.

While the simulator performs fairly well, there are a number of things that could be changed. If one were to rebuild the system from scratch, the system should be better modularized, particularly the ball manager. Also, as with any similar system, there are an infinite number of ways to tweak the video processing that could potentially improve the project's operation. In addition to enhancements to the existing foundation, there are many small additions that could be made. For instance, there were many corner cases that we largely ignored, such as balls on the floor and balls flying above the ceiling. The graphics used in the simulation were also fairly basic and could certainly use improvement. Overall, however, the project was successful. None of its shortcomings dramatically affected its functionality, and it was an enjoyable project to build and use.

Appendix: Verilog Source

This appendix contains the verilog source for our modules. The modules described above are in files as follows:

System Block	Associated Files
General Modules	Labkit.v Debounce.v Display_16hex.v Virtual_juggling.v
Camera Input	Ntsc2zbt.v Video_decoder.v Video_zbt.v 6111zbt.v
Controller	Vj_controller.v Vj_level_to_pulse.v
Video Processor	Vj_video_processor.v Divide.v Yuv_to_rgb.v
Ball Manager	Vj_ball_manager.v Vj_bm_physics.v Vj_bm_display.v Bram70x32.v
Output	Vj_output.v

```
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display.  Video input from the NTSC digitizer is
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used
// as the video frame buffer, with 8 bits used per pixel (black & white).
//
// Since the ZBT is read once for every four pixels, this frees up time for
// data to be stored to the ZBT during other pixel times.  The NTSC decoder
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize
// signals between the two (see ntsc2zbt.v) and let the NTSC data be
// stored to ZBT memory whenever it is available, during cycles when
// pixel reads are not being performed.
//
// We use a very simple ZBT interface, which does not involve any clock
// generation or hiding of the pipelining.  See zbt_6111.v for more info.
//
// switch[7] selects between display of NTSC video and test bars
// switch[6] is used for testing the NTSC decoder
// switch[1] selects between test bar periods; these are stored to ZBT
//           during blanking periods
// switch[0] selects vertical test bars (hardwired; not stored in ZBT)


//////////////////////////////////////////////////////////////////////////
//
// 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;//1'b0;
  //assign tv_in_i2c_data = 1'bZ;
  // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
  // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

  // SRAMs

/* change lines below to enable ZBT RAM bank0 */

/*
  assign ram0_data = 36'hZ;
  assign ram0_address = 19'h0;
  assign ram0_clk = 1'b0;
  assign ram0_we_b = 1'b1;
```

```
   assign ram0_cen_b = 1'b0;    // clock enable
*/

/* enable RAM pins */

   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'h0;

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

   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   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

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

wire clk = clock_65mhz;

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

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

// DEBOUNCE other signals
wire db_button_up, db_button_down,db_button_right,db_button_left;
debounce incdb(power_on_reset, clock_65mhz, ~button_up, db_button_up);
debounce decdb(power_on_reset, clock_65mhz, ~button_down, db_button_down);
debounce butdr(power_on_reset, clock_65mhz, ~button_right, db_button_right);
debounce butdl(power_on_reset, clock_65mhz, ~button_left, db_button_left);
wire [7:0] db_switch;
```

```
debounce switchdb[7:0](power_on_reset, clk, switch, db_switch);
wire db_button0, db_button1, db_button2, db_button3;
debounce b0db(power_on_reset, clock_65mhz, ~button0, db_button0);
debounce b1db(power_on_reset, clock_65mhz, ~button1, db_button1);
debounce b2db(power_on_reset, clock_65mhz, ~button2, db_button2);
debounce b3db(power_on_reset, clock_65mhz, ~button3, db_button3);



// display module for debugging

reg [63:0] dispdata;
display_16hex hexdisp1(reset, clock_65mhz, dispdata,// was clk
                       disp_blank, disp_clock, disp_rs, disp_ce_b,
                       disp_reset_b, disp_data_out);

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

// wire up to ZBT ram

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

zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
              vram_write_data, vram_read_data,
              ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

// generate pixel value from reading ZBT memory
wire [17:0]  vr_pixel;
wire [18:0]  vram_addr1;

vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
                 vram_addr1,vram_read_data);

// ADV7185 NTSC decoder interface code
// adv7185 initialization module
adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                    .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
```

```
                .tv_in_i2c_clock(tv_in_i2c_clock),
                .tv_in_i2c_data(tv_in_i2c_data));

  wire [29:0] ycrcb;   // video data (luminance, chrominance)
  wire [2:0] fvh;      // sync for field, vertical, horizontal
  wire       dv;       // data valid

  ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                .ycrcb(ycrcb), .f(fvh[2]),
                .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

  // code to write NTSC data to video memory

  wire [18:0] ntsc_addr;
  wire [35:0] ntsc_data;
  wire        ntsc_we;
  ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, ycrcb[29:0],
              ntsc_addr, ntsc_data, ntsc_we, db_button2);

  // code to write pattern to ZBT memory
  reg [31:0]   count;
  always @(posedge clk) count <= reset ? 0 : count + 1;

  wire [18:0]  vram_addr2 = count[0+18:0];
  wire [35:0]  vpat = ( db_button0 ? {4{count[3+3:3],4'b0}}
                 : {4{count[3+4:4],4'b0}} );

  // mux selecting read/write to memory based on which write-enable is chosen

  wire         sw_ntsc = ~db_button3;
  wire         my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
  wire [18:0]  write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
  wire [35:0]  write_data = sw_ntsc ? ntsc_data : vpat;

//   wire       write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//   assign     vram_addr = write_enable ? write_addr : vram_addr1;
//   assign     vram_we = write_enable;

  assign       vram_addr = my_we ? write_addr : vram_addr1;
  assign       vram_we = my_we;
  assign       vram_write_data = write_data;
```

```
// select output pixel data

reg [17:0]   pixel; // pixel[8:4] are the top five bits of Y, pixel[3:0] are the top 4 bits of U
wire         b,hs,vs;

delayN dn1(clk,hsync,hs);    // delay by 3 cycles to sync with ZBT read
delayN dn2(clk,vsync,vs);
delayN dn3(clk,blank,b);

always @(posedge clk)
  begin
    pixel <= db_button0 ? {hcount[8:6],5'b0,hcount[8:6],5'b0,hcount[8:7]} : vr_pixel;
  end

// VGA Output.  In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.

wire [63:0] hex_debug;

assign vga_out_pixel_clock = ~clock_65mhz;
virtual_juggling juggler(clock_65mhz,reset,1'b1,~b,
            pixel[17:10],pixel[9:2],{pixel[1:0],6'b0},hs,vs,
            hcount,vcount,
            vga_out_sync_b,vga_out_blank_b,
            vga_out_red,vga_out_green,vga_out_blue,
            vga_out_hsync,vga_out_vsync,
            db_button_up,db_button_down,db_button_left,db_button_right,db_switch,
            hex_debug);



    /*assign vga_out_red = pixel;
assign vga_out_green = pixel;
assign vga_out_blue = pixel;
assign vga_out_sync_b = 1'b1;    // not used
assign vga_out_pixel_clock = ~clock_65mhz;
assign vga_out_blank_b = ~b;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;*/

// debugging

assign led = ~{vram_addr[18:13],reset,db_button0};
```

```
  always @(posedge clock_65mhz)// was clk
    // dispdata <= {vram_read_data,9'b0,vram_addr};
    dispdata <= hex_debug;//{ntsc_data,9'b0,ntsc_addr};




endmodule
```

```
/*

This module represents the entire VirtualJuggling simulator.


*/
module virtual_juggling(clock_65mhz, reset,
                        vga_in_sync_b, vga_in_blank_b,
                        y, u, v,
                        vga_in_hsync, vga_in_vsync,
                        hcount_in, vcount_in,

                        vga_out_sync_b, vga_out_blank_b,
                        vga_out_red, vga_out_green, vga_out_blue,
                        vga_out_hsync, vga_out_vsync,

                        ball_inc,ball_dec,left_throw_button,right_throw_button,db_switch,
                        hex_display);
    input clock_65mhz;
    input reset;
    input vga_in_sync_b, vga_in_blank_b;
    input [7:0] y, u, v;
    input vga_in_hsync, vga_in_vsync;
    input [10:0] hcount_in;
    input [9:0] vcount_in;

    output vga_out_sync_b, vga_out_blank_b;
    output [7:0] vga_out_red, vga_out_green, vga_out_blue;
    output vga_out_hsync, vga_out_vsync;

    input ball_inc,ball_dec;
  input left_throw_button,right_throw_button;
    input [7:0] db_switch;
    output [63:0] hex_display;

    // PASS THROUGH with delay FOR DEBUGGING
    /*reg vga_out_sync_b, vga_out_blank_b;
    reg [7:0] vga_out_red, vga_out_green, vga_out_blue;
    reg vga_out_hsync, vga_out_vsync;

    always @ (posedge clock_65mhz) begin
        vga_out_sync_b <= vga_in_sync_b;
```

```
        vga_out_blank_b <= vga_in_blank_b;
        {vga_out_red,vga_out_green,vga_out_blue} <= {y,u,v};
        vga_out_hsync <= vga_in_hsync;
        vga_out_vsync <= vga_in_vsync;
    end*/

    // Logic wires
    wire new_frame;
    wire [31:0] left_hand_pos, right_hand_pos;
    wire [31:0] left_hand_vel, right_hand_vel;
    wire [1:0] left_hand_logic, right_hand_logic;

    // Intermediate video wires between video processor and output
    wire vga_mid_sync_b, vga_mid_blank_b;
    wire [10:0] hcount_mid;
    wire [9:0] vcount_mid;
    wire [7:0] vga_mid_red, vga_mid_green, vga_mid_blue;
    wire vga_mid_hsync, vga_mid_vsync;

    // Video processor module
    wire [31:0] VP_DEBUG;
    vj_video_processor video_proc(clock_65mhz,reset,
                vga_in_sync_b,vga_in_blank_b,hcount_in,vcount_in,
                vga_in_hsync,vga_in_vsync,
                y,u,v,
     db_switch,
                vga_mid_red,vga_mid_green,vga_mid_blue,
                vga_mid_sync_b,vga_mid_blank_b,
                hcount_mid,vcount_mid,
                vga_mid_hsync,vga_mid_vsync,
                new_frame,
                left_hand_pos,right_hand_pos,
                left_hand_vel,right_hand_vel,
                left_hand_logic,right_hand_logic,
                VP_DEBUG,8'b0);

    //Signals from ball manager to output module
    wire bm_we,bm_be;
    wire [4:0] bm_ball_number;
    wire [31:0] bm_ball_pos;
 wire [7:0] db_switch; //added for color debugging

/*
```

```
    // Test the ball sprites
    wire [31:0] TEST_JIG_DEBUG;
    vj_output_test_jig outputtest(clock_65mhz,reset,
                new_frame,bm_we,bm_ball_number,bm_ball_pos,bm_be,
                TEST_JIG_DEBUG,db_switch[5:1]);
*/

    // Output module
    vj_output outputmodule(clock_65mhz,reset,
                vga_mid_red,vga_mid_green,vga_mid_blue,
                hcount_mid,vcount_mid,
                vga_mid_sync_b,vga_mid_blank_b,vga_mid_hsync,vga_mid_vsync,
                left_hand_pos,right_hand_pos,
                bm_we,bm_ball_number,bm_ball_pos,bm_be,
                vga_out_red,vga_out_green,vga_out_blue,
                vga_out_sync_b,vga_out_blank_b,
                vga_out_hsync,vga_out_vsync);

    // DEBUG OUTPUT
    //assign hex_display = {left_hand_pos,right_hand_pos};
    //assign hex_display = {new_frame,29'b0,left_hand_logic,30'b0,right_hand_logic};
    //assign hex_display = {left_hand_vel,VP_DEBUG};
    //assign hex_display = {left_hand_vel,right_hand_vel};
    //assign hex_display = {left_hand_pos,TEST_JIG_DEBUG};
    //assign hex_display = {20'hABCDE,2'b0,bm_we,bm_be,3'b0,bm_ball_number,bm_ball_pos};

  //Controller modual instantiation

  wire new_frame_sync;
  wire reset_sync;
  wire add_ball_sync;
  wire dec_ball_sync;
  wire [7:0] gravity_sync;
  wire left_throw_sync,right_throw_sync;

  vj_controller  vj_control_instan(.clk(clock_65mhz),.new_frame_raw(new_frame),.reset_raw(reset),.
add_ball_raw(ball_inc),
                .dec_ball_raw(ball_dec),.left_throw_raw(left_throw_button),.right_throw_raw
(right_throw_button),
                .gravity_raw(db_switch),.new_frame_sync(new_frame_sync),
                .reset_sync(reset_sync),.add_ball_sync(add_ball_sync),.dec_ball_sync(dec_ball_sync),
                .left_throw_sync(left_throw_sync),.right_throw_sync(right_throw_sync),
                .gravity_sync(gravity_sync));
```

```
// wire [31:0] left_hand_pos, right_hand_pos;
//      wire [31:0] left_hand_vel, right_hand_vel;
//      wire [1:0] left_hand_logic, right_hand_logic;

   wire [15:0] l_h_x, l_h_y, r_h_x, r_h_y;
//   wire [15:0] l_h_x_vel, l_h_y_vel, r_h_x_vel, r_h_y_vel;
/*   wire left_throw, l_h_catch, right_throw, r_h_catch;
   assign left_throw =left_hand_logic[1];
   assign l_h_catch =left_hand_logic[0];
   assign right_throw =right_hand_logic[1];
   assign r_h_catch =right_hand_logic[0];
 */
   assign l_h_x = left_hand_pos[31:16];
   assign l_h_y = left_hand_pos[15:0];
   assign r_h_x = right_hand_pos[31:16];
   assign r_h_y = right_hand_pos[15:0];
//   assign l_h_x_vel = left_hand_vel[31:16];
//   assign l_h_y_vel = left_hand_vel[15:0];
//   assign r_h_x_vel = right_hand_vel[31:16];
//   assign r_h_y_vel = right_hand_vel[15:0];



   wire signed [16:0] control_left_y_vel;
   wire signed [16:0] control_right_y_vel;
   wire signed [16:0] control_left_x_vel;
   wire signed [16:0] control_right_x_vel;
   assign control_left_y_vel = (17'b11111111100000000 + {1,left_hand_vel[15:0]});
   assign control_right_y_vel = (17'b11111111100000000 + {1,right_hand_vel[15:0]});
   assign control_left_x_vel = (17'b00000000001111111 + {0,left_hand_vel[31:16]});
   assign control_right_x_vel = (17'b11111111110000001 + {1,right_hand_vel[31:16]});

   vj_ball_manager vj_bm_instan(.vclock(clock_65mhz),.reset(reset_sync),.new_frame
(new_frame_sync),.gravity(gravity_sync),
                .add_ball(add_ball_sync),.dec_ball(dec_ball_sync),
       .left_throw(left_hand_logic[1]),//1'b1),//  (left_throw_force|left_hand_logic[1]),
       .l_h_catch(left_hand_logic[0]),//|left_hand_logic[0]),
       .right_throw(right_hand_logic[1]),//1'b1),//   (right_throw_force)|right_hand_logic[1]),
       .r_h_catch(right_hand_logic[0]),//|right_hand_logic[0]),
       .l_h_y(l_h_y),.l_h_x(l_h_x),.l_h_y_vel(left_hand_vel[15:0]),//)16'b1111111100000000,//
control_left_y_vel[16:1]),
       .l_h_x_vel(control_left_x_vel[16:1]),//left_hand_vel[31:16]),16'b0000000001111111
```

```
	.r_h_y(r_h_y),.r_h_x(r_h_x),.r_h_y_vel(right_hand_vel[15:0]),//),16'b1111111100000000//
control_right_y_vel[16:1]),
	.r_h_x_vel(control_right_x_vel[16:1]),//right_hand_vel[31:16]),16'b1111111110000001
	.display_enb(bm_be),.ball_pos_out(bm_ball_pos),.ball_number(bm_ball_number),.write_request
(bm_we));
	assign hex_display = {bm_ball_pos,l_h_x,3'b000,new_frame_sync,3'b000,add_ball_sync,3'b000,
bm_be,3'b000,bm_we};

 /*

  vj_ball_manager vj_bm_instan(.vclock(clock_65mhz),.reset(reset_sync),.new_frame
(new_frame_sync),.gravity(gravity_sync),
		.add_ball(add_ball_sync),.dec_ball(dec_ball_sync),
	.left_throw(left_hand_logic[1]),//1'b1),//  (left_throw_force|left_hand_logic[1]),
	.l_h_catch(left_hand_logic[0]),//|left_hand_logic[0]),
	.right_throw(right_hand_logic[1]),//1'b1),//   (right_throw_force)|right_hand_logic[1]),
	.r_h_catch(right_hand_logic[0]),//|right_hand_logic[0]),
	.l_h_y(l_h_y),.l_h_x(l_h_x),.l_h_y_vel(left_hand_vel[15:0]),//l16'b1111111100000000),
	.l_h_x_vel(left_hand_vel[31:16]),//16'b0000000001111111),
	.r_h_y(r_h_y),.r_h_x(r_h_x),.r_h_y_vel(right_hand_vel[15:0]),//16'b1111111100000000),
	.r_h_x_vel(right_hand_vel[31:16]),//16'b1111111111000000),
	.display_enb(bm_be),.ball_pos_out(bm_ball_pos),.ball_number(bm_ball_number),.write_request
(bm_we));
	assign hex_display = {bm_ball_pos,l_h_x,3'b000,new_frame_sync,3'b000,add_ball_sync,3'b000,
bm_be,3'b000,bm_we};   */
endmodule
```

```
/*
vj_video_processor

This is the main video processing module for the
juggling simulator. It receives the raw camera
image as a VGA stream (with colors in the YUV
space instead of RGB) and locates the hands
in the frame. It also converts the signal to
RGB video so it can be displayed later.

Notes:
1. This module delays the video by one clock cycle.

2. This module displays the camera image in B&W

*/
module vj_video_processor(clock_65mhz,reset,
                          sync_in,blank_in,hcount_in,vcount_in,
                          hsync_in,vsync_in,
                          y,u,v,
              db_switch,
                          vga_red,vga_green,vga_blue,
                          sync_out,blank_out,hcount_out,vcount_out,
                          hsync_out,vsync_out,
                          new_frame,
                          left_pos,right_pos,
                          left_vel,right_vel,
                          left_logic,right_logic,
                          DEBUG_OUT,DEBUG_SETTINGS);

    // VGA signals for incoming video
    input clock_65mhz;
    input reset;
    input sync_in,blank_in;
    input [10:0] hcount_in;
    input [9:0] vcount_in;
    input hsync_in;
    input vsync_in;
    // Incoming color signals are 8-bit YUV values, not RGB
    input [7:0] y,u,v;
  input [7:0] db_switch;
```

```
// The module converts the YUV signal to RGB and outputs
// it to these ports, 8 bits per channel
output [7:0] vga_red,vga_green,vga_blue;
// Since the module must delay the signal, the output
// video signal is sent to the following ports
output sync_out,blank_out;
output [10:0] hcount_out;
output [9:0] vcount_out;
output hsync_out,vsync_out;

// Output signals
// new_frame is raised on the clock cycle when the hand
// information for the current frame has been calculated.
// This signals the beginning of the space between the
// frame and vsync.
output new_frame;
// Hand position/velocity signals
// bits [31:16] are for x, [15:0] are for y
// the low order bit of each value (e.g. 16 or 0)
// corresponds to 2^-5 (i.e. each value has 5 fractional bits)
output [31:0] left_pos,right_pos;
output [31:0] left_vel,right_vel;
//
output [1:0] left_logic,right_logic;

output [31:0] DEBUG_OUT;
input [7:0] DEBUG_SETTINGS;

reg [31:0] DEBUG_OUT;

// We delay the video by one clock cycle in order to calculate hand information.
reg [24:0] video_delay;

reg sync_out,blank_out;
reg [10:0] hcount_out;
reg [9:0] vcount_out;
reg hsync_out,vsync_out;

reg [7:0] vga_red,vga_green,vga_blue;

wire pos_ready,left_ready, right_ready;

assign new_frame = left_ready & right_ready;
```

```
	// Modules for computing the velocity and logical motions of the hands
	vj_vp_hand_logic left_hand(clock_65mhz,reset,pos_ready,left_pos,left_vel,left_logic,left_ready,
DEBUG_SETTINGS);
	vj_vp_hand_logic right_hand(clock_65mhz,reset,pos_ready,right_pos,right_vel,right_logic,
right_ready,DEBUG_SETTINGS);

	wire hand_pixel;	// Receives output from hand detector stating whether or not this pixel
belongs to a hand.
	wire [7:0] red, green, blue; // Outputs from the YUV to RGB converter.

	vj_vp_hand_detector hand_detector(clock_65mhz,reset,hcount_in,vcount_in,vsync_in,y,u,v,
pos_ready,left_pos,right_pos,hand_pixel, DEBUG_SETTINGS);

	yuv_to_rgb video_converter(y, u, v, red, green, blue);

	reg test = 0;

	always @ (posedge clock_65mhz) begin
		{sync_out,blank_out,hcount_out,vcount_out,hsync_out,vsync_out} <=
				{sync_in,blank_in,hcount_in,vcount_in,hsync_in,vsync_in};
		{vga_red, vga_green, vga_blue} <= (hand_pixel & db_switch[7]) ? 24'hFF0000 : (~db_switch
[6]) ? {red, green, blue} :
				hcount_out < 11'd512 ? (left_logic[1] ? {8'h00, green, 8'h00} : left_logic[0] ?
{8'h00, 8'h00, blue} : {red, green, blue}) :
				right_logic[1] ? {8'h00, green, 8'h00} : right_logic[0] ? {8'h00, 8'h00, blue} :
{red, green, blue};
		DEBUG_OUT <= left_logic[1] ? left_vel : DEBUG_OUT;
	end

endmodule

/*
Hand detector module
*/
module vj_vp_hand_detector(clock_65mhz,reset,
		hcount,vcount,vsync,y,u,v,
		pos_ready,left_pos,right_pos,hand_pixel,
		DEBUG_COLOR_TOLERANCE);

	input clock_65mhz;
	input reset;
	input [7:0] y, u, v;
```

```
    input [10:0] hcount;
    input [9:0] vcount;
    input vsync;

    output pos_ready;
    output [31:0] left_pos, right_pos;
    output hand_pixel;

    input [7:0] DEBUG_COLOR_TOLERANCE;

    parameter RED_U_THRESHOLD = 8'b10010000;
    parameter RED_Y_MIN = 8'b00011000;
    parameter RED_Y_MAX = 8'b11111111;
    parameter V_BACK_PORCH_START = 10'd770;// time to get off the screen and compute the
COMs

    reg new_frame;
    reg stale_frame;
    reg pos_ready;

    reg [31:0] left_pos, right_pos;

    reg hand_pixel;

    reg left_hand_pixel, right_hand_pixel;

    wire [15:0] left_x, left_y;
    wire [15:0] right_x, right_y;

    reg average_reset = 0;


    // We need to hold the hcount and vcount values so
    // that the averager uses the correct position
    reg [10:0] x_inc;
    reg [9:0] y_inc;


    always @ (posedge clock_65mhz) begin
        // Intentional use of blocking assignment to effect sequential execution.
        hand_pixel = ((u > RED_U_THRESHOLD) & (RED_Y_MIN < y) & (y < RED_Y_MAX));
        left_hand_pixel <= hand_pixel & (hcount < 11'd512);
        right_hand_pixel <= hand_pixel & (hcount > 11'd511) & (hcount < 11'd1024);
```

```
            // Hold the hcount and vcount values so that they are ready for the divider on the next cycle.
            x_inc <= hcount;
            y_inc <= vcount;

            // CODE FOR TESTING COLOR TOLERANCES...
            //hand_pixel = DEBUG_COLOR_TOLERANCE[1] ?
            //({1'b0,(DEBUG_COLOR_TOLERANCE[0] ? v : u)} > {DEBUG_COLOR_TOLERANCE
[6:2],4'b0}) :
            //({1'b0,(DEBUG_COLOR_TOLERANCE[0] ? v : u)} < {DEBUG_COLOR_TOLERANCE
[6:2],4'b0});

            // If we have reached the end of the displayed frame, then grab the COM's
            if ((vcount == V_BACK_PORCH_START) & ~stale_frame) begin
                pos_ready <= 1;
                left_pos <= {left_x[13:0],2'b0,left_y[13:0],2'b0}; // We calculate the COM with 4
fractional bits, but want 5.
                right_pos <= {right_x[13:0],2'b0,right_y[13:0],2'b0};
                stale_frame <= 1;
            end
            // Make sure that new_frame is only high for one cycle.
            else if (pos_ready) begin
                pos_ready <= 0;
            end
            else if (~vsync) begin
                stale_frame <=0;
            end

            // On vsync, reset the COMs.
            if (~vsync) average_reset <= 1;
            else average_reset <= 0;
        end

        // The parameters are defined to ignore the porches, in case that is an issue.
        weighted_average left_x_com(clock_65mhz,reset,left_hand_pixel,average_reset,{2'b0,x_inc,3'b0},
left_x);
        //defparam left_x_com.IN_MIN = 16'd0;
        //defparam left_x_com.IN_MAX = 16'd1023;
        weighted_average left_y_com(clock_65mhz,reset,left_hand_pixel,average_reset,{3'b0,y_inc,3'b0},
left_y);
        //defparam left_y_com.IN_MIN = 16'd0;
        //defparam left_y_com.IN_MAX = 16'd767;

        weighted_average right_x_com(clock_65mhz,reset,right_hand_pixel,average_reset,{2'b0,
```

```
x_inc,3'b0},right_x);
	//defparam right_x_com.IN_MIN = 16'd0;
	//defparam right_x_com.IN_MAX = 16'd1023;
	weighted_average right_y_com(clock_65mhz,reset,right_hand_pixel,average_reset,{3'b0,
y_inc,3'b0},right_y);
	//defparam right_y_com.IN_MIN = 16'd0;
	//defparam right_y_com.IN_MAX = 16'd767;

endmodule

module vj_vp_hand_logic(clk,reset,
		pos_ready,pos,vel,logic,logic_ready,DEBUG);
	input clk;
	input reset;
	input pos_ready;
	input [31:0] pos;

	output [31:0] vel;
	output [1:0] logic;
	output logic_ready;

	input [7:0] DEBUG;

	parameter CATCH_THRESHOLD = 12'b111111111100; // the y velocity must be greater than this
(negative) value
	parameter THROW_THRESHOLD = 12'b000000000110; // the acceleration must be greater than
this value
	parameter THROW_HISTORY_DEPTH = 6;

	// the output velocity is an average
	reg signed [15:0] x_vel[3:0];
	reg signed [15:0] y_vel[3:0];
	reg signed [15:0] y_accel[3:0];
	reg throw_logic,catch_logic;
	reg [THROW_HISTORY_DEPTH-1:0] throw_history;
	reg [31:0] old_pos;
	reg logic_ready;


   //added this
   parameter X_VEL_CONST = 19'b0; // LINE ADDED
   parameter Y_VEL_CONST = 19'b1111111110000000000; // LINE ADDED
```

```
   wire signed [18:0] x_vel_sum = x_vel[3]+x_vel[2]+x_vel[1]+x_vel[0]+X_VEL_CONST; // LINE
CHANGED

   wire signed [18:0] y_vel_sum = y_vel[3]+y_vel[2]+y_vel[1]+y_vel[0]+Y_VEL_CONST; // LINE
CHANGED

   wire signed [17:0] y_accel_sum = y_accel[3]+y_accel[2]+y_accel[1]+y_accel[0];

   assign vel = {x_vel_sum[17:2],y_vel_sum[18:3]};// LINE CHANGED

 /*
      wire signed [17:0] x_vel_sum = x_vel[3]+x_vel[2]+x_vel[1]+x_vel[0];
      wire signed [17:0] y_vel_sum = y_vel[3]+y_vel[2]+y_vel[1]+y_vel[0];

      wire signed [17:0] y_accel_sum = y_accel[3]+y_accel[2]+y_accel[1]+y_accel[0];

      assign vel = {x_vel_sum[17:2],y_vel_sum[17:2]};//{x_vel_sum[17:2],y_vel[3]};//
*/
      assign logic = {throw_logic, catch_logic & (!throw_logic)};

      always @ (posedge clk) begin
            if (reset) begin
                  {x_vel[3],x_vel[2],x_vel[1],x_vel[0]} <= 64'b0;
                  {y_vel[3],y_vel[2],y_vel[1],y_vel[0]} <= 64'b0;

                  {y_accel[3],y_accel[2],y_accel[1],y_accel[0]} <= 64'b0;

                  throw_history <= 0;

                  old_pos <= 0;
                  catch_logic <= 0;
                  throw_logic <= 0;
            end
            else if (pos_ready) begin
                  old_pos <= pos;
                  {x_vel[3],x_vel[2],x_vel[1],x_vel[0]} <= {x_vel[2],x_vel[1],x_vel[0],pos[31:16]-old_pos
[31:16]};
                  {y_vel[3],y_vel[2],y_vel[1],y_vel[0]} <= {y_vel[2],y_vel[1],y_vel[0],pos[15:0]-old_pos
[15:0]};

                  {y_accel[3],y_accel[2],y_accel[1],y_accel[0]} <= {y_accel[2],y_accel[1],y_accel[0],pos
[15:0]-old_pos[15:0]-y_vel[0]};
```

```
                catch_logic <= (y_vel_sum[17:6] < 12'h7FF) | (y_vel_sum[17:6] >
CATCH_THRESHOLD);
                throw_history[THROW_HISTORY_DEPTH-1:0] <= {throw_history
[THROW_HISTORY_DEPTH-2:0],(y_accel_sum[17:6] < 12'h7FF) & (y_accel_sum[17:6] >
THROW_THRESHOLD) & !catch_logic};//(y_vel[3][15:4] + y_vel[2][15:4])-(y_vel[1][15:4]+y_vel[0]
[15:4]) > {8'hFF,DEBUG[7:4]};
                throw_logic <= throw_history[0] & (!throw_history[THROW_HISTORY_DEPTH-1:1]);
                logic_ready <= 1;
            end
            else if (logic_ready) begin
                logic_ready <= 0;
            end
        end
endmodule

/*
weighted_average
calculates a weighted average of
sequentially supplied values.
*/
module weighted_average(clk,reset,enable,average_reset,inval,average
            /*,sum,count,raw_average,unused_remainder*/);
        input clk;
        input reset;
        input enable;
        input average_reset;

        input [15:0] inval;
        output [15:0] average;

        //output [31:0] sum;
        //output [31:0] count;

        parameter IN_MIN = 16'h0000;
        parameter IN_MAX = 16'hFFFF;

        reg [31:0] sum = 0;
        reg [31:0] count = 0;

        reg [15:0] average;

        wire [31:0] raw_average;        // The top 16 bits of AVERAGE should always be 0
        wire [31:0] unused_remainder;   // We ignore the remainder output.
```

```
    wire unused_rfd;

    always @ (posedge clk) begin
        if (reset | average_reset) begin
            sum <= 0;
            count <= 0;
            //average <= raw_average[15:0];
        end
        else if (enable && (inval > IN_MIN) && (inval < IN_MAX)) begin
            sum <= sum + {16'b0,inval};
            count <= count + 1;
        end
        average <= (|sum) ? raw_average[15:0] : 16'b0; // Force the average to be 0 if the sum is 0
    end

    divide average_divide(sum, count, raw_average, unused_remainder, clk, unused_rfd, 1'b0, 1'b0,
1'b1);

endmodule
```

```
module yuv_to_rgb(y,u,v,vga_r,vga_g,vga_b);
    input [7:0] y;
    input [7:0] u;
    input [7:0] v;
    output [7:0] vga_r;
    output [7:0] vga_g;
    output [7:0] vga_b;

    assign vga_r = y;
    assign vga_g = y;
    assign vga_b = y;

endmodule
```

```
module vj_controller(clk,new_frame_raw,reset_raw,add_ball_raw,dec_ball_raw,left_throw_raw,
right_throw_raw,gravity_raw,
                new_frame_sync,reset_sync,add_ball_sync,dec_ball_sync,left_throw_sync,right_throw_sync,
gravity_sync);
   input clk;
   input new_frame_raw;
   input reset_raw;
   input add_ball_raw;
   input dec_ball_raw;
   input left_throw_raw;
   input right_throw_raw;
   input [7:0] gravity_raw;

   output new_frame_sync;
   output reset_sync;
   output add_ball_sync;
   output dec_ball_sync;
   output left_throw_sync;
   output right_throw_sync;
   output [7:0] gravity_sync;

   reg [7:0] gravity_sync = 7'b0;
   reg reset_sync=0;
   reg reset_hold=0;
   reg right_throw_sync=0;
   reg right_throw_hold=0;
   reg left_throw_sync=0;
   reg left_throw_hold=0;
   reg new_frame_sync=0;
   reg add_ball_hold=0;
   reg add_ball_sync=0;
   reg dec_ball_hold=0;
   reg dec_ball_sync=0;

   wire dec_ball_pulse;
   wire add_ball_pulse;
   wire right_throw_force_pulse;
   wire left_throw_force_pulse;
//   reg right_throw_force_hold=0;  //to hold right throw that is a pulse until new frame
//   reg left_throw_force_hold=0;
//   reg right_throw_force=0;
//   reg left_throw_force=0;
```

```
    vj_level_to_pulse foradd(clk,add_ball_raw,add_ball_pulse);
    vj_level_to_pulse fordec(clk,dec_ball_raw,dec_ball_pulse);
    vj_level_to_pulse forleft(clk,left_throw_raw,left_throw_force_pulse);
    vj_level_to_pulse forright(clk,right_throw_raw,right_throw_force_pulse);

  always @ (posedge clk) begin
    new_frame_sync <= new_frame_raw;
    gravity_sync <= new_frame_raw ? gravity_raw : gravity_sync;

    reset_sync <= (new_frame_raw && reset_hold) ? 1 : 0;
    reset_hold <= reset_raw ? 1 : (new_frame_raw ? 0 : reset_hold);

    right_throw_sync <= (new_frame_raw && right_throw_hold) ? 1 : 0;
    right_throw_hold <= right_throw_force_pulse ? 1 : (new_frame_raw ? 0 : right_throw_hold);

    left_throw_sync <= (new_frame_raw && left_throw_hold) ? 1 : 0;
    left_throw_hold <= left_throw_force_pulse ? 1 : (new_frame_raw ? 0 : left_throw_hold);

    add_ball_sync <= (new_frame_raw && add_ball_hold) ? 1 : 0;
    add_ball_hold <= add_ball_pulse ? 1 : (new_frame_raw ? 0 : add_ball_hold);

    dec_ball_sync <= (new_frame_raw && dec_ball_hold) ? 1 : 0;
    dec_ball_hold <= dec_ball_pulse ? 1 : (new_frame_raw ? 0 : dec_ball_hold);
/*
    right_throw_force <= new_frame_raw ? right_throw_force_hold : right_throw_force;
    right_throw_force_hold <= new_frame_raw ? 0 : (right_throw_force_pulse ? 1 :
right_throw_force_hold);

    left_throw_force <= new_frame_raw ? left_throw_force_hold : left_throw_force;
    left_throw_force_hold <= new_frame_raw ? 0 : (left_throw_force_pulse ? 1 :
left_throw_force_hold);
*/
  end


endmodule
```

```
module vj_level_to_pulse(clk,level_in,pulse_out);
  input clk;
  input level_in;

  output pulse_out;

  reg level_hold=0;
  reg pulse_out=0;

  always @ (posedge clk) begin
    level_hold <= level_in;
    pulse_out <= (level_in && ~level_hold);
  end

  endmodule
```

```
module vj_ball_manager(vclock,reset,new_frame,gravity,
            add_ball,dec_ball,
      left_throw,l_h_catch,right_throw,r_h_catch,
      l_h_y,l_h_x,l_h_y_vel,l_h_x_vel,
      r_h_y,r_h_x,r_h_y_vel,r_h_x_vel,

      display_enb,ball_pos_out,ball_number,write_request);


  input vclock;       // 65MHz clock
  input reset;        // 1 to initialize module  -- IS COMING in syncronized to new_frame
  input new_frame;  //from Controller
  input [7:0] gravity;  // gravity from switches debounced and synchronized to new_frame
  input add_ball;   //adds a ball when button 1 is pressed
  input dec_ball;   //removes a ball when button 0 is pressed
  input left_throw;  //pulse when left hand is in throwing position
  input l_h_catch;  //signal when left hand is in catch position
  input right_throw; //when right hand throws
  input r_h_catch;  //when right hand catches
  input [15:0] l_h_y;      //left hand y position
  input [15:0] l_h_x;     // left hand x position
  input [15:0] l_h_y_vel;   //left hand y velocity
  input [15:0] l_h_x_vel;   // left hand x velocity
  input [15:0] r_h_y;      //right hand y position
  input [15:0] r_h_x;     // right hand x position
  input [15:0] r_h_y_vel;   //right hand y velocity
  input [15:0] r_h_x_vel;   // right hand x velocity

  output display_enb;  //says the ball is to be displayed
  output [31:0] ball_pos_out;
  output [4:0] ball_number;
  output write_request;

  wire display_enb;
  wire [31:0] ball_pos_out;
  wire [4:0] ball_number;
  wire write_request;
  wire [69:0] info_from_bram;
  wire [4:0] bram_addr_b;
  wire wea;
  wire done_calc;
  wire [69:0] ball_info_in;// = 70'b0;
```

```
  wire [69:0] ball_info_out;// = 70'b0;
  wire [4:0] count_addr;// = 5'b0;




  vj_bm_physics instan_ball_m(vclock,reset,new_frame,gravity,
            add_ball,dec_ball,
      left_throw,l_h_catch,right_throw,r_h_catch,
      l_h_y,l_h_x,l_h_y_vel,l_h_x_vel,
      r_h_y,r_h_x,r_h_y_vel,r_h_x_vel,
            ball_info_in,ball_info_out,count_addr,done_calc,wea);



  vj_bm_display instan_bm_disp(.vclock(vclock),.reset(reset),.new_frame(new_frame),.done_calc
(done_calc),
                        .info_from_bram(info_from_bram),.bram_addr_b(bram_addr_b),
                        .ball_pos_out(ball_pos_out),.display_enb(display_enb),
                        .ball_number(ball_number),.write_request(write_request));



 bram70x32 instan(
   .addra(count_addr),
       .addrb(bram_addr_b),
       .clka(vclock),
       .clkb(vclock),
       .dina(ball_info_out),
       .douta(ball_info_in),
       .doutb(info_from_bram),
       .wea(wea));    // synthesis black_box



endmodule
```

```
module vj_bm_physics (vclock,reset,new_frame,gravity,
            add_ball,dec_ball,
      left_throw,l_h_catch,right_throw,r_h_catch,
      l_h_y,l_h_x,l_h_y_vel,l_h_x_vel,
      r_h_y,r_h_x,r_h_y_vel,r_h_x_vel,
            ball_info_in,ball_info_out,count_addr,done_calc,wea);



  input vclock;       // 65MHz clock
  input reset;        // 1 to initialize module  -- IS COMING in syncronized to new_frame
  input new_frame;  //from Controller
  input [7:0] gravity;  // gravity from switches
  input add_ball;   //is 1 for whole frame synced to new_frame
  input dec_ball;   //is 1 for whole frame and synced to new_frame
  input left_throw;  //pulse when left hand is in throwing position
  input l_h_catch;  //signal when left hand is in catch position
  input right_throw; //when right hand throws
  input r_h_catch;  //when right hand catches
  input [15:0] l_h_y;      //left hand y position
  input [15:0] l_h_x;     // left hand x position
  input [15:0] l_h_y_vel;   //left hand y velocity
  input [15:0] l_h_x_vel;   // left hand x velocity
  input [15:0] r_h_y;      //right hand y position
  input [15:0] r_h_x;     // right hand x position
  input [15:0] r_h_y_vel;   //right hand y velocity
  input [15:0] r_h_x_vel;   // right hand x velocity
  input [69:0] ball_info_in;  //ball data from BRAM

  output [69:0] ball_info_out; //  //ball data to BRAM
  output [4:0] count_addr;     //address to bram
  output done_calc;
  output wea;  //write enable to BRAM



  parameter ball_r = {11'd40,5'b00000};

  parameter ball_w = {11'd33,5'b00000};
  parameter ball_h = {11'd33,5'b00000};

  parameter hand_h = {11'd50,5'b00000};
```

```
parameter hand_w = {11'd140,5'b00000};

parameter ball_right=0;   //to keep track of states within case statement
parameter ball_left=1;
parameter ball_down=0;
parameter ball_up=1;

parameter screenw = {11'd1023,5'b00000};
parameter screenh = {11'd767,5'b00000};

   wire in_play_in;
   wire in_hand_in;
   wire in_left_in;
   wire in_right_in;
   wire y_dir_in;
   wire x_dir_in;
   wire [15:0] ball_y_in;
   wire [15:0] ball_x_in;
   wire [15:0] y_vel_in;
   wire [15:0] x_vel_in;

      assign in_play_in = ball_info_in[69];
      assign in_hand_in = ball_info_in[68];
      assign in_left_in = ball_info_in[67];
      assign in_right_in = ball_info_in[66];
      assign y_dir_in = ball_info_in[65];
      assign x_dir_in = ball_info_in[64];
      assign ball_y_in = ball_info_in[63:48];
      assign ball_x_in = ball_info_in[47:32];
      assign y_vel_in = ball_info_in[31:16];
      assign x_vel_in = ball_info_in[15:0];

reg [15:0] ball_x_out = 16'h0000;
reg [15:0] ball_y_out = 16'd0;
//keep track of x and y velocities
reg [15:0] x_vel_out = 16'd0;
reg [15:0] y_vel_out = 16'd0;


reg x_dir_out = 0;    //controls movement in x dir left or right
reg y_dir_out = 0;    //controls movement in y dir up or down

wire [15:0] two_x_vel_in;
```

```
  assign two_x_vel_in = (2 * x_vel_in);

  wire [15:0] two_y_vel_in;
  assign two_y_vel_in = (2 * y_vel_in);

  //so i have a comparative value to know if the ball is too close to the edge
  wire [15:0] far_right;     //
  assign far_right = ball_x_in + ball_w; // how far toward right the ball is
  wire [15:0] far_bot;
  assign far_bot = ball_y_in + ball_h; //how far toward botton the ball is



 //ball movement control with everything else around it

parameter read_state = 0;
parameter write_state = 1;
reg read_hold=0;  //so the program stays in the read_state two cycles per round.
reg read_or_write=0;  // to determine whether we're reading or writing to the BRAM alternating the we signal
reg [6:0]balls= 7'b0;      //keeps track of how many balls are currenlty in the system
reg [4:0]count_addr= 5'b0;  //keeps track of which BRAM address the ball controller is at
reg done_calc=0;      //all the balls have been read and written during a frame when high
reg in_play_out=0;        //ball is in play when high
reg in_hand_out=0;        //ball is in a hand when high
reg in_left_out=0;        //ball in in left hand when high
reg in_right_out=0;       //ball is in right hand when high
reg wea=0;            // write enable on BRAM
reg which_hand=0;  //switches which hand the added ball goes into
reg reset_level=0;  // leveled reset to be high for one entire frame to reset each value
reg add_ball_level=0; //high when add ball goes high and stays until a ball is added   (add_ball synced to new_frame)
reg dec_ball_level=0;  //high when dec_ball goes high until ball is deleted  (dec_ball synced to new_frame)
//reg ball_added=0;  //high when ball has been added and set low every new frame
//reg ball_deced=0;  //high when ball has been removed and set low every new frame
reg [1:0] count_divider= 2'b0;  //so count_addr increments only once per 3 clock cycles
reg left_thrown = 0;  //signals within one frame that a ball has been thrown from the left hand so no more that frame
reg right_thrown = 0;  //signals within one frame that a ball has been thrown from right hand and no more that frame
reg left_throw_level=0;
reg right_throw_level=0;
```

```
//display_16hex hexInstan(reset, vclock, {ball_x,x_vel,two_x_vel,neg_two_x_vel}, disp_blank,
disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out);



always @ (posedge vclock) begin


//   reset_level <= reset ? 1 : (new_frame ? 0 : reset_level); //makes a reset level signal for one frame
//   add_ball_level <= add_ball ? 1 : (new_frame ? 0 : add_ball_level); //,makes add ball level for one
frame
//   dec_ball_level <= dec_ball ? 1 : (new_frame ? 0 : dec_ball_level);
//   right_thrown <= new_frame ? 0 : right_thrown;
//   left_thrown <= new_frame ? 0 : left_thrown;

//   right_throw_level <= ?

/*   if(~add_ball_level) begin
     add_ball_level <= add_ball ?  1 : add_ball_level;
   end

   if(~dec_ball_level) begin
     dec_ball_level <= dec_ball ? 1 : dec_ball_level;
   end
*/
   if (new_frame) begin
         done_calc <= 0;
     count_addr <= 0;
     count_divider <= 3;
     wea <= 0;  ///added for debugging
     read_hold <= 0;
     reset_level <= reset;
     left_throw_level <= left_throw;
     right_throw_level <= right_throw;
     which_hand <= add_ball ? ~which_hand : which_hand;  //switches which hand ball goes into each
time you add one
     add_ball_level <= add_ball;
     dec_ball_level <= dec_ball;
```

```
   right_thrown <= 0;
   left_thrown <= 0;
end
     else if (count_addr == 31) begin
       done_calc <= 1;
     end

else if (~done_calc) begin  //includes case statement
   count_divider <= (count_divider == 2) ? 0 : (count_divider + 1);
       count_addr <= (count_divider == 2) ? (count_addr + 1) : count_addr;



   case (read_or_write)
     read_state: begin
           read_hold <=  ~read_hold;
           wea <= 0;
           read_or_write <= read_hold; //sent it to write_state after 2nd cycle in read_state
         end
     write_state: begin
       read_or_write <= read_state;  //send it back to read state after 1 cycle in write state
       wea <= 1;
       if (reset || reset_level) begin
         if (count_addr < 2) begin
           in_play_out <= 1;
           in_hand_out <= 1;
           in_left_out <= 0;
           in_right_out <= 1;
           y_dir_out <= 1'b0;
           x_dir_out <= 1'b0;
           ball_y_out <= 16'b0;
               ball_x_out <= 16'b0;
               y_vel_out <= 16'b0;
               x_vel_out <= 16'b0;
         end
         else if (count_addr == 2)begin
           in_play_out <= 1;
           in_hand_out <= 1;
           in_left_out <= 1;
           in_right_out <= 0;
           y_dir_out <= 1'b0;
               x_dir_out <= 1'b0;
               ball_y_out <= 16'b0;
               ball_x_out <= 16'b0;
```

```
            y_vel_out <= 16'b0;
            x_vel_out <= 16'b0;
        end
        else begin
          in_play_out <= 0;
          in_hand_out <= 0;
          in_left_out <= 0;
          in_right_out <= 0;
          y_dir_out <= 1'b0;
            x_dir_out <= 1'b0;
            ball_y_out <= 16'b0;
            ball_x_out <= 16'b0;
            y_vel_out <= 16'b0;
            x_vel_out <= 16'b0;
        end
      end
      else if (in_play_in) begin  //if it's in play do all this
        if (dec_ball_level) begin
          in_play_out <= 0;
          dec_ball_level <= 0;
          end

           else if (in_hand_in) begin   // if it is in a hand do this
                if (in_right_in && right_throw_level  && ~right_thrown) begin  // if it's in the right
hand and right hand is throwing do this                    ball_x <= r_h_x;   // set ball position and velocity
and direction
                        in_play_out <= 1;
          in_hand_out <= 0;
          in_left_out <= 0;
          in_right_out <= 0;
          ball_x_out <= r_h_x;
          ball_y_out <= r_h_y;
                        x_vel_out <= r_h_x_vel[15] ? -r_h_x_vel : r_h_x_vel;
                        y_vel_out <= r_h_y_vel[15] ? -r_h_y_vel : r_h_y_vel;
                        y_dir_out <= r_h_y_vel[15] ? ball_up : ball_down;
                        x_dir_out <= r_h_x_vel[15] ? ball_left : ball_right;
          right_thrown <= 1;
                    end
               else if (in_left_in && left_throw_level && ~left_thrown) begin
                        in_play_out <= 1;
          in_hand_out <= 0;
          in_left_out <= 0;
          in_right_out <= 0;
```

```
            ball_x_out <= l_h_x;
                        ball_y_out <= l_h_y;
                        x_vel_out <= l_h_x_vel[15] ? -l_h_x_vel : l_h_x_vel;
                        y_vel_out <= l_h_y_vel[15] ? -l_h_y_vel : l_h_y_vel;
                        y_dir_out <= l_h_y_vel[15] ? ball_up : ball_down;
                        x_dir_out <= l_h_x_vel[15] ? ball_left : ball_right;
            left_thrown <= 1;
                      end
          else begin
            in_play_out <= in_play_in;
            in_hand_out <= in_hand_in;
            in_left_out <= in_left_in;
            in_right_out <= in_right_in;
            ball_x_out <= ball_x_in;
                        ball_y_out <= ball_y_in;
                        x_vel_out <= x_vel_in;
                        y_vel_out <= y_vel_in;
                        y_dir_out <= y_dir_in;
                        x_dir_out <= x_dir_in;
          end
           end
              else begin  //if not in hand do all this
          in_play_out <= 1;
          if (l_h_catch && ~y_dir_in &&
            (ball_x_in < (l_h_x + (hand_w / 2))) &&
            (((hand_w / 2) + ball_x_in) > l_h_x) &&
            (ball_y_in < l_h_y) &&
            ((ball_y_in + ball_h + (hand_h / 2)) > l_h_y)) begin
                      //((((ball_y_in - l_h_y) * (ball_y_in - l_h_y)) +
                      //((ball_x_in - l_h_x) * (ball_x_in - l_h_x))) < (ball_r * ball_r))) begin
                      // if it the left hand is in a catch position and the hand
            // is in the radius of the ball then catch it and set new states
                          in_hand_out <= 1;
                          in_right_out <= 0;
                          in_left_out <= 1;
                        end
                    else if (r_h_catch &&  ~y_dir_in &&
                  (ball_x_in < (r_h_x + (hand_w / 2))) &&
                  (((hand_w / 2) + ball_x_in) > r_h_x) &&
                  (ball_y_in < r_h_y) &&
                  ((ball_y_in + ball_h + (hand_h / 2)) > r_h_y)) begin
            //((((ball_y_in - r_h_y) * (ball_y_in - r_h_y)) + ((ball_x_in - r_h_x) * (ball_x_in - r_h_x)))
< (ball_r * ball_r))) begin
```

```
                        in_hand_out <= 1;
                        in_right_out <= 1;
                        in_left_out <= 0;
                    end
                else begin
            //x_vel_out <= x_vel_in;
            in_left_out <= in_left_in;
            in_right_out <= in_right_in;
            in_hand_out <= 0;       //needs to be set otherwise if a ball is added to the hand in_hand is
then one
            in_right_out <= 0;
            in_left_out <= 0;
            if (x_dir_in == ball_right) begin
              ball_x_out <= ball_x_in + x_vel_in;
              x_dir_out <= (screenw < (two_x_vel_in + far_right)) ? ball_left : ball_right;
              x_vel_out <= (screenw < (two_x_vel_in + far_right)) ? (x_vel_in / 2) : x_vel_in;
            end
            else begin
              ball_x_out <= ball_x_in - x_vel_in;
              x_dir_out <= (ball_x_in < (two_x_vel_in/* + ball_w*/)) ? ball_right : ball_left;
              x_vel_out <= (ball_x_in < (two_x_vel_in/* + ball_w*/)) ? (x_vel_in / 2) : x_vel_in;
            end
            if (y_dir_in == ball_down) begin
              ball_y_out <= ball_y_in + y_vel_in;
              y_dir_out <= (screenh < (far_bot + (two_y_vel_in + gravity))) ? ball_up : ball_down;
              y_vel_out <= (y_vel_in + gravity);
            end
            else begin
              ball_y_out <= ball_y_in - y_vel_in;
              y_dir_out <= ((ball_y_in < two_y_vel_in) || (y_vel_in < gravity)) ? ball_down : ball_up;
              y_vel_out <= (ball_y_in < two_y_vel_in) ? 16'd32 : ((y_vel_in < gravity) ? (gravity -
y_vel_in) : (y_vel_in - gravity));
            end
          end
        end
      end
      else begin //when not in play do this
        if (add_ball_level) begin
          in_play_out <= 1;
          in_hand_out <= 1;
          in_left_out <= which_hand;
          in_right_out <= ~which_hand;
              y_dir_out <= 1'b0;
```

```
                x_dir_out <= 1'b0;
                ball_y_out <= 16'b0;
                ball_x_out <= 16'b0;
                y_vel_out <= 16'b0;
                x_vel_out <= 16'b0;
              add_ball_level <= 0;
               end
            else begin
              in_play_out <= 1'b0;
              in_hand_out <= 1'b0;
              in_left_out <= 1'b0;
              in_right_out <= 1'b0;
                y_dir_out <= 1'b0;
                x_dir_out <= 1'b0;
                ball_y_out <= 16'b0;
                ball_x_out <= 16'b0;
                y_vel_out <= 16'b0;
                x_vel_out <= 16'b0;
            end
          end
        end
      endcase
    end
end


assign ball_info_out = {in_play_out,in_hand_out,in_left_out,in_right_out,y_dir_out,x_dir_out,
ball_y_out,ball_x_out,y_vel_out,x_vel_out};




endmodule
```

```
module vj_bm_display (vclock,reset,new_frame,done_calc,info_from_bram,
                bram_addr_b,ball_pos_out,display_enb,ball_number,write_request);

  input vclock;       // 65MHz clock
  input reset;        // 1 to initialize module  -- IS COMING in syncronized to new_frame
  input new_frame;  //from Controller
  input done_calc;
  input [69:0] info_from_bram;

  output [4:0] bram_addr_b;
  output [31:0] ball_pos_out;
  output display_enb;
  output [4:0] ball_number;
  output write_request;
 // output stop_calc;

  reg [4:0]bram_addr_b=0;
  reg [31:0] ball_pos_out=0;
  reg display_enb=0;                  //need to set equal to 0
  reg [4:0] ball_number=0;
  reg write_request=0;

  reg stop_calc=0;
  reg [4:0]ball_number_hold=0;
  reg write_request_hold=0;



  always @ (posedge vclock) begin
    if (reset) begin
      stop_calc <= 0;
      bram_addr_b <= 0;
      write_request <= 0;
      ball_number <= 0;
    end
    else if (stop_calc) begin
      stop_calc <= new_frame ? 0 : stop_calc;
      bram_addr_b <= new_frame ? 0 : bram_addr_b;
      write_request <= 0;
      ball_number <= new_frame ? 0 : ball_number;
    end
    else if (done_calc) begin
```

```
        bram_addr_b <= (bram_addr_b + 1);      //inc bram_addr until is reaches the last value then reset
        stop_calc <= (ball_number == 31) ? 1 : stop_calc;     //stop_calculations when you hit the last bram
address
        write_request_hold <= (ball_number == 30) ? 0 : 1;   //stop at 30 because of delay in pipeline
        write_request <= write_request_hold;
        ball_pos_out <= {info_from_bram[47:32],info_from_bram[63:48]};   //x now in high and y in low
16 bits
        display_enb <= (ball_number == 31) ? 0 : (info_from_bram[69] && ~(info_from_bram[68] ||
info_from_bram[67]));
        ball_number <= ball_number_hold;      //to delay ball_number
        ball_number_hold <= bram_addr_b;
     end
  end


endmodule
```

```
/*
vj_output

This is the output module for the juggling simulator.
It takes a video feed and overlays the hand and ball
sprites.

Notes:

This module delays the video feed by TWO clock cycles.
The first cycle allows the system to

*/
module vj_output(clock_65mhz,reset,
                 // Video inputs
                 r,g,b,
                 hcount,vcount,
                 sync_in,blank_in,hsync_in,vsync_in,

                 // Sprite position inputs
                 left_hand_pos,right_hand_pos,
                 we,ball_num,new_pos,be,

                 // Video outputs
                 vga_out_red,vga_out_green,vga_out_blue,
                 vga_out_sync_b,vga_out_blank_b,
                 vga_out_hsync,vga_out_vsync);
    input clock_65mhz;
    input reset;

    // Incoming XGA video signals
    input [7:0] r, g, b;
    input [10:0] hcount;
    input [9:0] vcount;
    input sync_in, blank_in, hsync_in, vsync_in;

    // Incoming hand locations
    input [31:0] left_hand_pos, right_hand_pos;

    // Incoming ball-sprite memory signals
    input we;
    input [4:0] ball_num;
```

```
    input [31:0] new_pos;
    input be;

    // Output XGA signals
    output [7:0] vga_out_red, vga_out_green, vga_out_blue;
    output vga_out_sync_b, vga_out_blank_b;
    output vga_out_hsync, vga_out_vsync;

    reg vga_out_sync_b, vga_out_blank_b;
    reg vga_out_hsync, vga_out_vsync;
    reg vga_out_red, vga_out_green, vga_out_blue;

    wire [23:0] overlay_pixel, left_hand_pixel, right_hand_pixel, ball_pixel;

    vj_output_hand_sprite left_hand_sprite(left_hand_pos[31:21],left_hand_pos[14:5],
        hcount,vcount,left_hand_pixel);
    vj_output_hand_sprite right_hand_sprite(right_hand_pos[31:21],right_hand_pos[14:5],
        hcount,vcount,right_hand_pixel);

    vj_output_agregate_ball_sprite_memory ball_sprite(clock_65mhz,reset,hcount,vcount,
        ball_pixel,we,ball_num,new_pos,be);

    // We OR the output of the left hand, right hand, and ball pixels
    // to determine the overlay pixel. A black overlay pixel is not displayed.
    assign overlay_pixel = left_hand_pixel | right_hand_pixel | ball_pixel;

    // Registers for pipelining the output
    reg [3:0] vga_signal_pipeline;
    reg [23:0] vga_color_pipeline;

  always @ (posedge clock_65mhz) begin
        // Delay the video by TWO clock cycles
        // one cycle is for computing output,
        // the other is to guarantee good setup/hold times
        vga_signal_pipeline <= {sync_in,blank_in,hsync_in,vsync_in};
        {vga_out_sync_b,vga_out_blank_b,vga_out_hsync,vga_out_vsync} <= vga_signal_pipeline;

        vga_color_pipeline <= (|overlay_pixel) ? overlay_pixel : {r,g,b};
        {vga_out_red,vga_out_green,vga_out_blue} <= vga_color_pipeline;
  end

endmodule
```

```
/*
This module acts as a single sprite for displaying
all the balls on the screen. The output module provides
hcount and vcount, while the module returns an output
pixel color. As with other sprites, this logic is purely
combinational.

This module also acts as a form of memory. By providing
synchronous inputs, other modules may modify the location
of a given ball on the screen.
*/
module vj_output_agregate_ball_sprite_memory(clk,reset,hcount,vcount,pixel_out,
                        we,ball_num,new_pos,be/*debug ,x_out,y_out,ball_enable_out end debug*/);
    input clk; // This is the clock used for synchronous functions
    input reset;
    input [10:0] hcount; // The current hcount being queried
    input [9:0] vcount; // The current vcount being queried
    output [23:0]pixel_out; // The color value of the pixel at (hcount,vcount)

    input we; // Write enable signal (syncronous)
    input [4:0] ball_num; // Number of ball to modify (synchronous)
    input [31:0] new_pos; // New position of ball (synchronous)
    input be;

    //DEBUG
    //output [11*4-1:0] x_out;
    //output [10*4-1:0] y_out;
    //output [4-1:0] ball_enable_out;

    parameter NUM_BALLS = 32;


    reg [11*32-1:0] x = 0;
    reg [10*32-1:0] y = 0;
    reg [32-1:0] ball_enable = 0;
    wire [32-1:0] ball_pixels;

    //DEBUG
    //assign x_out = x[11*4-1:0];
    //assign y_out = y[10*4-1:0];
    //assign ball_enable_out = ball_enable[4-1:0];

    vj_output_ball_sprite ball_sprites[32-1:0](x,y,ball_enable,hcount,vcount,ball_pixels);
```

```
    assign pixel_out = (|ball_pixels) ? 24'h00FFFF : 24'h0;

    always @ (posedge clk) begin
        if (reset) begin
            x <= 0;
            y <= 0;
            ball_enable <= 0;
        end
        else if (we) begin
            //x[ball_num+10:ball_num] <= new_pos[26:16];
            //y[ball_num+9:ball_num] <= new_pos[9:0];
            x[11*ball_num+10] <= new_pos[31];
            x[11*ball_num+9] <= new_pos[30];
            x[11*ball_num+8] <= new_pos[29];
            x[11*ball_num+7] <= new_pos[28];
            x[11*ball_num+6] <= new_pos[27];
            x[11*ball_num+5] <= new_pos[26];
            x[11*ball_num+4] <= new_pos[25];
            x[11*ball_num+3] <= new_pos[24];
            x[11*ball_num+2] <= new_pos[23];
            x[11*ball_num+1] <= new_pos[22];
            x[11*ball_num] <= new_pos[21];

            y[10*ball_num+9] <= new_pos[14];
            y[10*ball_num+8] <= new_pos[13];
            y[10*ball_num+7] <= new_pos[12];
            y[10*ball_num+6] <= new_pos[11];
            y[10*ball_num+5] <= new_pos[10];
            y[10*ball_num+4] <= new_pos[9];
            y[10*ball_num+3] <= new_pos[8];
            y[10*ball_num+2] <= new_pos[7];
            y[10*ball_num+1] <= new_pos[6];
            y[10*ball_num] <= new_pos[5];

            ball_enable[ball_num] <= be;
        end
    end
endmodule


/*
This module is a sprite for displaying a single hand on the screen.
```

Note that this sprite doesn't contain its own color information.

CODE BASE TAKEN FROM LAB 4
*/

```
module vj_output_hand_sprite(x,y,hcount,vcount,pixel);
	parameter WIDTH = 16;
	parameter HEIGHT = 16;
	parameter COLOR = 24'h00FF00;
	input [10:0] x;
	input [9:0] y;
	input [10:0] hcount;
	input [9:0] vcount;
	output [23:0] pixel;
	reg [23:0] pixel;
	always @ (x or y or hcount or vcount) begin
		if ((hcount >= x && hcount < (x+WIDTH)) &&
				(vcount >= y && vcount < (y+HEIGHT)))
			pixel = COLOR;
		else pixel = 0;
	end
endmodule

/*
This module is a sprite for displaying a single ball on the screen.

CODE BASE TAKEN FROM LAB 4
*/
module vj_output_ball_sprite(x,y,enable,hcount,vcount,pixel);
	parameter RADIUS = 16;
	input [10:0] x;
	input [9:0] y;
	input enable;
	input [10:0] hcount;
	input [9:0] vcount;
	output pixel;
	reg pixel;

	// Compute the position deltas before we use them
	wire [10:0] delta_x = x-hcount;
	wire [9:0] delta_y = y-vcount;
```

```
    always @ (enable or x or y or hcount or vcount) begin
        if (enable &&
                (hcount >= (x-RADIUS) && hcount <= (x+RADIUS)) &&
                (vcount >= (y-RADIUS) && vcount <= (y+RADIUS))) begin
            //pixel = 1;
            // In order to avoid delaying the video signal, we hard-wire the hands
            /*if ((delta_x <= 11'b00000000010) || (delta_x >= 11'b11111111101))
                pixel = 1;
            else if ((delta_y <= 10'b0000000010) || (delta_y >= 10'b1111111101))
                pixel = 1;
            else if ((((delta_x <= 11'b00000000100) || (delta_x >= 11'b11111111011)))
                    && (((delta_y <= 10'b000001000) || (delta_y >= 10'b1111110111))))
                pixel = 1;
            else if ((((delta_x <= 11'b0000001000) || (delta_x >= 11'b11111110111)))
                    && (((delta_y <= 10'b000000100) || (delta_y >= 10'b1111111011))))
                pixel = 1;
            else if ((delta_x+delta_y <= 11'b00000001010) || (delta_x+delta_y >= 11'b11111110101))
                pixel = 1;*/
            if (((delta_x+delta_y <= 2*RADIUS) || (delta_x+delta_y >= 2*RADIUS))&&
                    ((delta_x-delta_y <= 2*RADIUS) || (delta_x-delta_y >= 2*RADIUS)))
                pixel = 1;
            else
                pixel = 0;
        end
        else pixel = 0;
    end
endmodule

/*
This module is used for testing the output module. It displays
the hand sprites and all 32 balls on the screen.

*/
module vj_output_test_jig(clock_65mhz,reset,new_frame,we,ball_number,new_pos,be,DEBUG_DATA,
SWITCHES);

    input clock_65mhz;
    input reset;
    input new_frame;

    output we;
    output [4:0] ball_number;
    output [31:0] new_pos;
```

```
	output be;

	input [4:0] SWITCHES;
	output [31:0] DEBUG_DATA;

	reg counting = 0;

	reg we,be;
	reg [4:0] ball_number;
	reg [31:0] new_pos;

	reg [31:0] inner_counter = 32'b0;
	assign DEBUG_DATA = inner_counter;

	always @ (posedge clock_65mhz) begin
		if (reset) begin
			inner_counter <= 0;
			counting <= 0;
			ball_number <= 0;
		end
		if (new_frame) begin
			counting <= 1;
			ball_number <= 0;
			inner_counter <= inner_counter + 1;
		end
		else   if (counting) begin
			if (ball_number == 5'b11111) counting <= 0;//stop counting.
			ball_number <= ball_number+1;
			we <= 1;
			be <= inner_counter[4+ball_number[0]];
			new_pos <= {({1'b0,ball_number[2:0],inner_counter[10:4],5'b0})+16'd64,
					({2'b0,ball_number[4:3],inner_counter[10:4],5'b0})+16'd64};
		end
		else   begin
			we <= 0;
			be <= 0;
		end
	end

endmodule
```

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

```
module divide (
      dividend,
      divisor,
      quot,
      remd,
```

```
        clk,
        rfd,
        aclr,
        sclr,
        ce);    // synthesis black_box

input [31 : 0] dividend;
input [31 : 0] divisor;
output [31 : 0] quot;
output [31 : 0] remd;
input clk;
output rfd;
input aclr;
input sclr;
input ce;

// synopsys translate_off

        SDIVIDER_V3_0 #(
                0,      // c_has_aclr
                0,      // c_has_ce
                0,      // c_has_sclr
                1,      // c_sync_enable
                1,      // divclk_sel
                32,     // dividend_width
                32,     // divisor_width
                0,      // fractional_b
                32,     // fractional_width
                0)      // signed_b
        inst (
                .DIVIDEND(dividend),
                .DIVISOR(divisor),
                .QUOT(quot),
                .REMD(remd),
                .CLK(clk),
                .RFD(rfd),
                .ACLR(aclr),
                .SCLR(sclr),
                .CE(ce));


// synopsys translate_on
```

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

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

endmodule
```

// 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 bram70x32.v when simulating
// the core, bram70x32. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Guide".

```
module bram70x32 (
    addra,
    addrb,
    clka,
    clkb,
```

```
        dina,
        douta,
        doutb,
        wea);    // synthesis black_box

input [4 : 0] addra;
input [4 : 0] addrb;
input clka;
input clkb;
input [69 : 0] dina;
output [69 : 0] douta;
output [69 : 0] doutb;
input wea;

// synopsys translate_off

        BLKMEMDP_V6_1 #(
                5,      // c_addra_width
                5,      // c_addrb_width
                "0",    // c_default_data
                32,     // c_depth_a
                32,     // c_depth_b
                0,      // c_enable_rlocs
                1,      // c_has_default_data
                1,      // c_has_dina
                0,      // c_has_dinb
                1,      // c_has_douta
                1,      // c_has_doutb
                0,      // c_has_ena
                0,      // c_has_enb
                0,      // c_has_limit_data_pitch
                0,      // c_has_nda
                0,      // c_has_ndb
                0,      // c_has_rdya
                0,      // c_has_rdyb
                0,      // c_has_rfda
                0,      // c_has_rfdb
                0,      // c_has_sinita
                0,      // c_has_sinitb
                1,      // c_has_wea
                0,      // c_has_web
                18,     // c_limit_data_pitch
                "mif_file_16_1",        // c_mem_init_file
```

```
        0,      // c_pipe_stages_a
        0,      // c_pipe_stages_b
        0,      // c_reg_inputsa
        0,      // c_reg_inputsb
        "0",    // c_sinita_value
        "0",    // c_sinitb_value
        70,     // c_width_a
        70,     // c_width_b
        0,      // c_write_modea
        0,      // c_write_modeb
        "0",    // c_ybottom_addr
        1,      // c_yclka_is_rising
        1,      // c_yclkb_is_rising
        1,      // c_yena_is_high
        1,      // c_yenb_is_high
        "hierarchy1",   // c_yhierarchy
        0,      // c_ymake_bmm
        "16kx1",        // c_yprimitive_type
        1,      // c_ysinita_is_high
        1,      // c_ysinitb_is_high
        "1024", // c_ytop_addr
        0,      // c_yuse_single_primitive
        1,      // c_ywea_is_high
        1,      // c_yweb_is_high
        1)      // c_yydisable_warnings
    inst (
        .ADDRA(addra),
        .ADDRB(addrb),
        .CLKA(clka),
        .CLKB(clkb),
        .DINA(dina),
        .DOUTA(douta),
        .DOUTB(doutb),
        .WEA(wea),
        .DINB(),
        .ENA(),
        .ENB(),
        .NDA(),
        .NDB(),
        .RFDA(),
        .RFDB(),
        .RDYA(),
        .RDYB(),
```

```
        .SINITA(),
        .SINITB(),
        .WEB());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of bram70x32 is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of bram70x32 is "black_box"

endmodule
```

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

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

  parameter NDELAY = 650000;
  parameter NBITS = 20;

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

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

endmodule
```

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

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

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

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

  reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

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

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

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

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

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

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

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

  always @(posedge clock_27mhz)
    if (clock_tick)
```

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

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

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

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

```
      state <= state+1;
    end

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

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

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

```
        end

      endcase // casex(state)
    end

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

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

```
    endcase

endmodule
```

```
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.

////////////////////////////////////////////////////////////////////////
// Prepare data and address values to fill ZBT memory with NTSC data

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);

  input        clk;   // system clock
  input        vclk;  // video clock from camera
  input [2:0]   fvh;
  input        dv;
  input [29:0]         din;
  output [18:0] ntsc_addr;
  output [35:0] ntsc_data;
  output       ntsc_we;       // write enable for NTSC data
  input        sw;           // switch which determines mode (for debugging)

  parameter     COL_START = 10'd30;
  parameter     ROW_START = 10'd30;

  // here put the luminance data from the ntsc decoder into the ram
  // this is for 1024 x 768 XGA display

  reg [9:0]     col = 0;
  reg [9:0]     row = 0;
  reg [17:0]    vdata = 0;
  reg           vwe;
  reg           old_dv;
  reg           old_frame;     // frames are even / odd interlaced
  reg           even_odd;      // decode interlaced frame to this wire
```

```
wire        frame = fvh[2];
wire        frame_edge = frame & ~old_frame;

always @ (posedge vclk) //LLC1 is reference
  begin
    old_dv <= dv;
    vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
    old_frame <= frame;
    even_odd = frame_edge ? ~even_odd : even_odd;

    if (!fvh[2])
      begin
        col <= fvh[0] ? COL_START :
              (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
        row <= fvh[1] ? ROW_START :
              (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
        vdata <= (dv && !fvh[2]) ? {din[29:22],din[19:12],din[9:8]} : vdata;
      end
  end

// synchronize with system clock

reg [9:0] x[1:0],y[1:0];
reg [17:0] data[1:0];
reg       we[1:0];
reg       eo[1:0];

always @(posedge clk)
  begin
    {x[1],x[0]} <= {x[0],col};
    {y[1],y[0]} <= {y[0],row};
    {data[1],data[0]} <= {data[0],vdata};
    {we[1],we[0]} <= {we[0],vwe};
    {eo[1],eo[0]} <= {eo[0],even_odd};
  end

// edge detection on write enable signal

reg old_we;
wire we_edge = we[1] & ~old_we;
always @(posedge clk) old_we <= we[1];

// shift each set of four bytes into a large register for the ZBT
```

```
  reg [35:0] mydata;
  always @(posedge clk)
    if (we_edge)
      mydata <= { mydata[17:0], data[1] };

  // compute address to store data in

  wire [18:0] myaddr = {y[1][8:0], eo[1], x[1][9:1]};

  // alternate (256x192) image data and address
  wire [35:0] mydata2 = {data[1],data[1],data[1],data[1]};
  wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};

  // update the output address and data only when four bytes ready // WRITE EVERY 2!!!

  reg [18:0] ntsc_addr;
  reg [35:0] ntsc_data;
  wire       ntsc_we = sw ? we_edge : (we_edge & (x[1][0]==1'b0));

  always @(posedge clk)
    if ( ntsc_we )
      begin
        ntsc_addr <= sw ? myaddr2 : myaddr;   // normal and expanded modes
        ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};
      end

endmodule // ntsc_to_zbt
```

```
//
// 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
```

```
//////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
  input vclock;
  output [10:0] hcount;
  output [9:0] vcount;
  output       vsync;
  output       hsync;
  output       blank;

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

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

  // vertical: 806 lines total
  // display 768 lines
  wire      vsyncon,vsyncoff,vreset,vblankon;
  assign    vblankon = hreset & (vcount == 767);
  assign    vsyncon = hreset & (vcount == 776);
  assign    vsyncoff = hreset & (vcount == 782);
  assign    vreset = hreset & (vcount == 805);

  // sync and blanking
  wire      next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
    hcount <= hreset ? 0 : hcount + 1;
    hblank <= next_hblank;
    hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

    vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
    vblank <= next_vblank;
```

```
    vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

    blank <= next_vblank | (next_hblank & ~hreset);
  end
endmodule

//////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.

module vram_display(reset,clk,hcount,vcount,vr_pixel,
                    vram_addr,vram_read_data);

  input reset, clk;
  input [10:0] hcount;
  input [9:0]  vcount;
  output [17:0] vr_pixel;
  output [18:0] vram_addr;
  input [35:0]  vram_read_data;

  parameter HMID = 9'd367; // The horizontal center of the image in MEMORY
  parameter HSTART = HMID-9'd256; // The horizontal counter decrements!!!
  parameter VMID = 9'd287; // The vertical center of the image in MEMORY
  parameter VSTART = VMID-9'd192;

  wire [18:0]  vram_addr = {1'b0,vcount[9:1]+VSTART, ~hcount[10:2]-9'd180};

  wire [1:0]    hc4 = hcount[1:0];
  reg [17:0]    vr_pixel;
  reg [35:0]    vr_data_latched;
  reg [35:0]    last_vr_data;

  always @(posedge clk)
    last_vr_data <= (hc4==2'd3) ? vr_data_latched : last_vr_data;

  always @(posedge clk)
    vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched;
```

```
  always @(*)        // each 36-bit word from RAM is decoded to 4 bytes
    case (hc4)
      2'd3: vr_pixel = last_vr_data[17:0];//last_vr_data[8:0];
      2'd2: vr_pixel = last_vr_data[17:0];//last_vr_data[8+9:0+9];
      2'd1: vr_pixel = last_vr_data[35:18];//last_vr_data[8+18:0+18];
      2'd0: vr_pixel = last_vr_data[35:18];//last_vr_data[8+27:0+27];
    endcase

endmodule // vram_display

//////////////////////////////////////////////////////////////////////////
// parameterized delay line

module delayN(clk,in,out);
  input clk;
  input in;
  output out;

  parameter NDELAY = 3;

  reg [NDELAY-1:0] shiftreg;
  wire             out = shiftreg[NDELAY-1];

  always @(posedge clk)
    shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule // delayN
```

```
//
// 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.
//

////////////////////////////////////////////////////////////////////////////////
// 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;

endmodule // zbt_6111
```