Designing an Intuitive Video-Game Controller

Eddie Fagin and Irene Fan

13 December 2006

Abstract

Alternative video-game controllers can directly influence the entertainment value of a game, either positively or negatively. Additionally, some controllers strengthen the player's physical and emotional attachment to the game. This project attempts to explore these relationships further through a simulated video-game console. This system merges a controller interface and informational displays with a fully-playable game. The game is based on Atari's *Super Breakout* (1978), which is played by sliding a paddle horizontally along the bottom edge of the screen. This paddle deflects a ball, which in turn is used to break different colored bricks at the top of the screen. Certain colored bricks will drop items that affect game play, such as bombs that shrink the size of the paddle. The game contains four distinct levels, which must be played in sequence. The player manipulates the game paddle by moving a bright LED within the viewable frame of a color camera, which in turn is connected to a Xilinx Field-Programmable Gate Array (FPGA) system. This LED signal is processed using filter modules designed with extensive testing to determine constraints, and calculation modules to determine the motion of center-point of the LED light clocked at a frequency of 60 Hz. The system processes the speed and direction of user motion and passes this information to the game. The game then uses this data to proportionately control paddle movement. In the design, synchronization and timing details were carefully considered and tested as to avoid metastability and display issues. Most testing was done directly through the FPGA, although ModelSim was used to confirm the operation of smaller auxiliary modules. Other students in the course (as well as staff) were asked to test our project using both the controller and the directional pad on the FPGA, and observations collected from these experiences tends to confirm the hypothesis that the controller does in fact heighten the player's attachment to the game.

Contents

List of Figures

1 Overview

Throughout the somewhat brief history of video games, game designers have been held back by technological restrictions, player preferences, bureaucratic oppression, and deadlines. In the days of arcade ubiquity, consoles were built specifically to house a single game and could be custom-tailored to that game; the number of buttons and joysticks on any arcade console reflected the exact input demands of the game inside. With the move to home consoles, designers have become vulnerable to the constraints imposed by standard game controllers as well. Most console games are now built around these controllers because one of the most decisive factors of a game's success is the ease with which the player can physically interact with the game world. This project explores the concept of an alternative user interface by constructing an new, simple controller for an old game. This design attempts to demonstrate the theory that "intuitive" methods of control will enhance a game's entertainment value.

The controller is a button-activated standard diffuse LED– specifically, a $5 LED flashlight from RadioShack. The project can be driven successfully by any bright LED. A player interacts with the game by pointing the LED controller at a color camera and moving the controller around the frame defined by the camera. This information is the primary input to the game. One of the secondary features of the controller interface is that any simple joystick-powered game from the Atari era could be "plugged in" to the virtual console. However, one game is sufficient to demonstrate the controller's functionality. In this game, the player moves the controller along the horizontal axis to manipulate a paddle. The speed at which the user moves the controller is related to the speed at which the paddle moves.

The controller interface generates velocity information in each frame by filtering and averaging the LED light, as shown by the block diagram in Figure 1. Several modules, originally used to display real-time black-and-white video footage from a

camera, are adjusted in numerous places to generate a real-time color (RGB) camera image. The NTSC decoder converts camera input data to 24 bits of intensity and chrominance (YCrCb) per pixel. One quarter of this information is stored in ZBT memory until needed. Since the camera should be out of focus for better controller operation, a high-quality image is unnecessary.

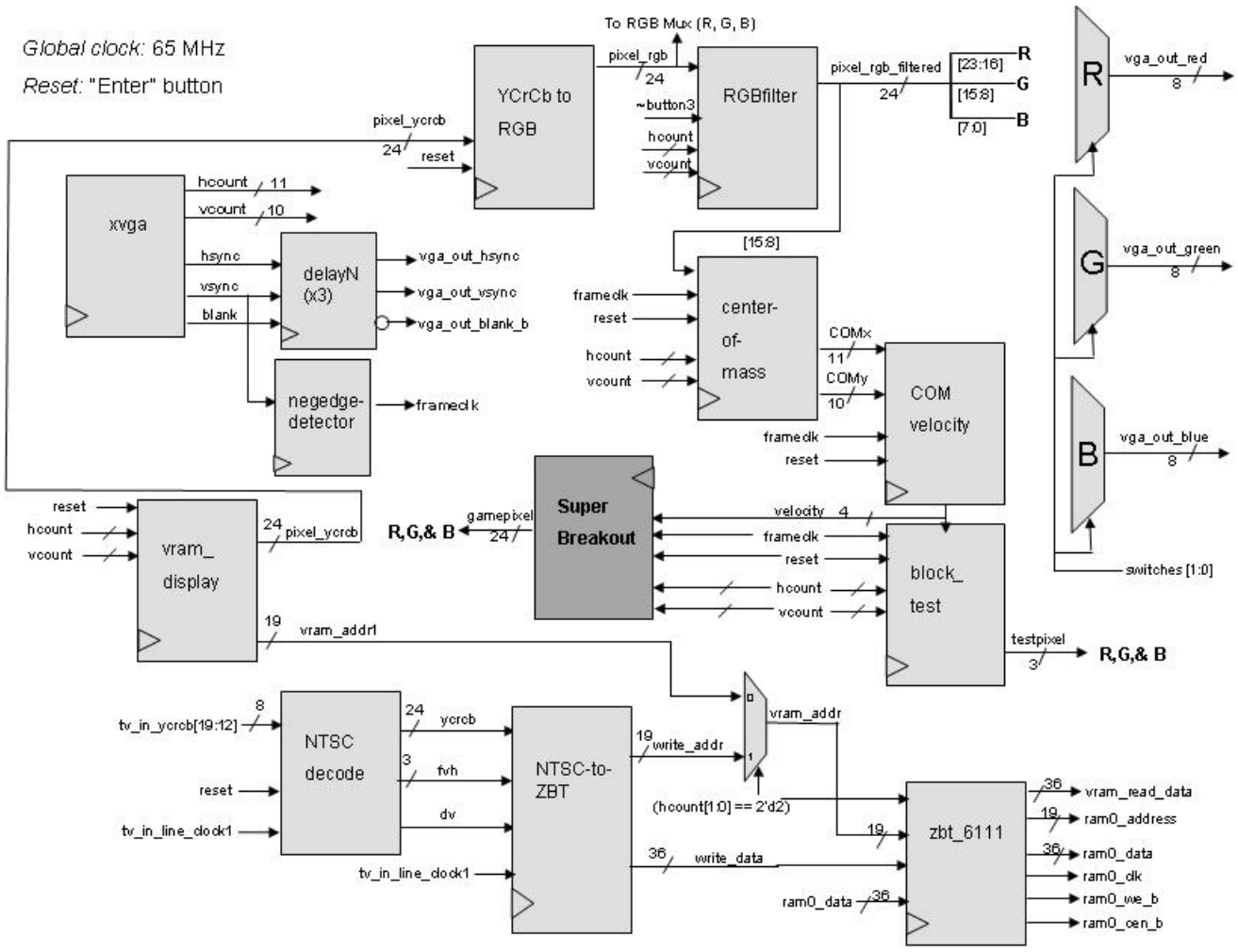

Figure 1: Block diagram for the controller.

The YCrCb2RGB converter from the Xilinx toolkit generates a 24-bit RGB signal for each 24-bit YCrCb read from the ZBT memory. This RGB data is passed through a filter that removes all red, green, and blue values below a certain cutoff point. Of

the data that passes this cutoff, only the strong red, green, or blue values pass through the filter. This process removes most of the image data, allowing only the brightest basic colors to reach the center-of-mass calculations. Only the green pixel data is considered, but the other two colors are readily available if needed

The green pixel data is passed to a module (centerofmass) that calculates the effective center of pixel mass and attempts to remove any anomalies caused by noise or other external factors like the overhead lighting. Once the center of mass in each frame has been calculated, another module (COMvelocity) determines the velocity (speed and direction) of this point, based on what it knows about the previous point's position. The speed can take on one of four values (zero, slow, medium, and fast), and direction can be up, down, left, or right. Therefore, this module has 16 possible outputs (four bits) to the rest of the system.

In this project, an updated version of the popular Atari game *Super Breakout* (1978) was created to demonstrate the controller's functionality. In *Super Breakout*, there are rows of bricks lining the top of the screen and a ball that bounces off the side and top walls, breaking bricks whenever it ricochets against them. A paddle is located at the bottom of the screen, and it is moved left and right to deflect the ball; if the paddle misses the ball and the ball touches the bottom of the screen, the game is over. The user's main motive is to break all the bricks on the screen. In our version, we designed four different levels with various brick formations, implemented multiple lives, and made game-affecting items drop from various bricks as they are hit.

When the controller is being used, the ball's horizontal velocity changes depending on the paddle's speed upon ball-paddle impact. The game simulates the paddle having a large frictional coefficient, which speeds up the ball if both paddle and ball are moving in the same x-direction and slows down the ball if the paddle and ball are travelling opposite ways. The amount by which the ball speeds up or slows

down is proportional to the paddle's speed upon impact.

The game divides the entire screen into an 8 by 24 grid of blocks, each one 128 by 32 pixels. Each block either contains a brick or does not. Bricks' colors reflect the number of remaining hits they need before they break: red bricks need 1 hit, yellow bricks need 2, green bricks need 3, teal bricks need 4, and blue bricks break after 5 hits, as shown in Figure 2. So, a red brick would disappear from the screen once it was hit, a yellow brick would turn red, a green brick would turn yellow, etc. Magenta bricks never break, and whereas the other-colored bricks are solid blocks, the unbreakable bricks are pictured as a thick magenta shell around a hollow core.

Figure 2: The colors of the bricks reflect the number of remaining hits they need to get before they break.

Different levels in the game are designed to gradually introduce the different types of bricks. The first level contains three rows of red, yellow, and green bricks at the top of the screen; the second level brings in the unbreakable magenta bricks; the third level introduces teal and blue bricks in a challenging starburst pattern; and the last level ties everything together in a fun arrangement of bricks spelling out "6.111."

The game has three types of game-affecting items: bombs, expanders, and life capsules, as shown in Figure 3. Bombs look like red balls and are dropped from red

bricks that are broken; they shorten the paddle upon impact. Expanders are pairs of green outward-pointing arrows that fall from green bricks; the paddle is lengthened if it catches one of these as they fall. Bombs can shorten a paddle until it disappears from the screen; expanders lengthen a paddle up to its initial size. Life capsules are the third kind of game-affecting item, and they fall from blue bricks that have been hit. These blue-colored balls, if caught, grant the player an extra life in the game (up to a maximum of 5 extra lives).

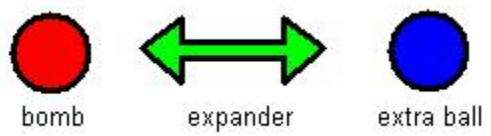

Figure 3: There are three types of game-affecting objects: bombs, which shrink the paddle; expanders, which lengthen the paddle; and life capsules, which grant the player an extra ball.

Besides dropping from red bricks, bombs also start falling from the top of the screen near the end of each level. When there are less than 4 bricks left on the screen, bombs start falling every 128 frames from scattered locations across the top of the screen. This random bombing behavior intensifies when there is only 1 brick left on the screen, increasing the frequency to every 32 frames.

The game's score is incremented by the game's difficulty (speed of 0, 4, 8 or 12) at each collision with a non-magenta block, no matter whether the brick actually breaks. This score is displayed on the lower 32 bits of the hex display on the FPGA labkit during gameplay, and an overall high score for all games played since the labkit was reprogrammed is shown on the top 32 bits.

The game is synchronized to a 65-MHz clock. The main module, `superbreakout.v`, is a big linear finite state machine (FSM) that updates the state of the game at the end of every frame. This is accomplished in several steps. First, given the current ball position and x-y velocities, the module calculates where the ball

would want to move next if no bricks were in its way. The block that contains this anticipated move is checked to see if it contains a brick–in the event of a collision, the appropriate brick is "hit" and the ball "bounces" off of it by changing velocity, simulating a perfect reflection off the brick. The position of the paddle is also updated before the beginning of the next frame, as is the state of any other objects on screen.

Several other submodules work together to create the other major parts of the game. `circle`, `modblob`, and `expander_blob` create the sprites for the ball, bombs, lifecapsules, paddle, and expanders. The display of game-affecting objects and their position is managed under `drop_game_object`. The `display` module draws the bricks from logic and also manages the pixels for all the objects shown on screen. The `speed` module changes the speed of the ball depending on how it hits the paddle, and the `score` module keeps track of the current game score. An intertitle module generates the pixels for intetitles shown at the beginning and end of the game as well as between levels.

The information for the bricks are stored in two BRAMs. One is a dual-port read/write memory, which is used to display the bricks and is modified during the game as bricks are hit. The other is a read-only memory that holds a copy of the original information used to generate the different brick formations in each level. Upon the start of a new level or a reset to the game, the information from the read-only memory is copied over to the dual-port memory, and this generates on screen a fresh copy of the correct brick arrangement.

This document describes how the controller and game were designed, discussing in detail the function and implementation of each submodule. Testing and debugging methodologies are discussed along with thoughts about the overall process of making such a project.

2 Project Modules

2.1 Controller

2.1.1 `rgbfilter.v`

(Eddie)

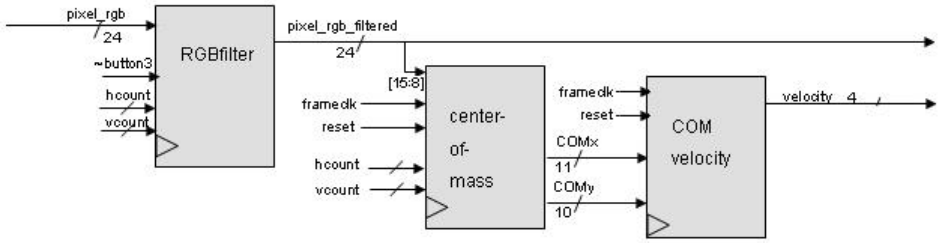

Figure 4: RGB Filter, Center of Mass, and COM Velocity

As shown in Figure 4, the RGB Filter is actually a series of small sub-filters that process the per-frame RGB camera data and remove all low-brightness color values. The middle eight bits (green color values) of the output are then used to calculate the center of green light "mass" for game use. This post-processed green pixel data can be viewed in the project's "Filter" mode of operation.

The purpose of this filter is to isolate any potential inputs (bright lights) and remove any noise or background information. By reducing the possible pixel values from 2^{24} (17 million; 8 bits each of red, green, and blue) to 2^2 (red, green, blue, or black), the center-of-mass calculation requires far less processing power. In addition, the mass calculations only look at the green values, reducing the number of possible inputs to two–green or black.

As mentioned before, our filter has four sub-filters. Each part is designed to remove or amplify a specific feature of the camera image data, outlined here:

1. Remove anything not related to camera input. The NTSC decoder and display

modules generate an image (approximately 700 x 500 pixels) within the larger 1024 x 768 screen. As a result of this process, the pixels outside this window are simply noise that should be discarded when analyzing the image data.

2. Remove all RGB values below a specified cutoff parameter. Each 8-bit value for red, green, and blue indicates the respective brightness of that color. One of the module inputs is an 8-bit constant (0-255) that defines the minimum acceptable brightness, and each third of the RGB input is compared with this value. If the value meets the constraint, it passes unchanged, and otherwise is set to black (zero). For our project, `filter` is fixed at 252, which was determined after extensive testing.

3. Select the brightest color, if it exists, and amplify it. If no color (red, green, or blue) is clearly brighter than the other two, discard that value. For example, if green is brighter than red and blue for this pixel, set the red and blue components to zero and set the green component to 255, the maximum.

4. Determine the output, depending on the user's preferences. One of the module's inputs, `greenfilter`, when high, forces the red and blue components of every pixel to zero. Our project's calculations only require green pixels, so `greenfilter` is always forced to high.

2.1.2 `centerofmass.v`

(Eddie)

The center-of-mass (COM) module dynamically computes the effective center of all bright green pixels in each frame. The module takes as input the stream of filtered pixels from the camera (one pixel / clock), as well as the 65-MHz clock and the frame clock. Since the incoming filtered pixels from RGBFilter are either completely green or completely black, we can treat each pixel as "on" or "off."

For a collection of i objects, each weighing m_i and at position r_i, the center of mass of these objects *(R)* is defined as:

$$\mathbf{R} = \frac{1}{M} \sum m_i \mathbf{r}_i$$

where M is the total mass of the system. In our system, we find the center of mass of green pixels. Each pixel "weighs" the same amount, so we can calculate the center of pixel mass each frame by independently summing the x and y coordinates of each green pixel and dividing those sums by the total number of green pixels in the frame.

Two dividers, created via the CoreGen tools, handle the averaging process. The horizontal and vertical dividers take 22- and 20-bit position sums, respectively, and divide that value by the total pixel weight, a 20-bit value. This pixel weight must pass a certain cutoff (greater than 15) before the COM can be updated; this safety check ensures that the controller does not act unexpectedly due to noise when idle.

2.1.3 `COMvelocity.v`

(Eddie)

The COM module models many pixels each frame as a single point, and passes this coordinate to the COMVelocity module. This module compares the current frames COM to the last frames value, and from this information, determines the speed and direction of the COM.

First, the module figures out if the COM motion is within reasonable limits. If the change in position is much greater than expected, the module understands this anomaly as a glitch or noise, so the controller re-calibrates itself to this new value while setting the COM speed to zero. If the change in COM is very small (two pixels or less in any direction), the module will also ignore this change in order to help stabilize the output.

If the change in COM passes the safety checks, the module then determines the

cardinal direction of point motion, as shown in Figure 5. Essentially, the process models the old COM as the origin of a Cartesian coordinate system and determines the rough direction of the new point (i.e. for QI: "Did the point go more up than left?"). Note, left and right are flipped to account for the camera pointing at the user, creating a mirror image of user motion.

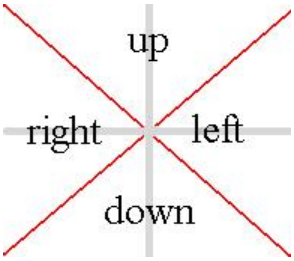

Figure 5: How the COM Velocity module determines COM direction

Finally, the module calculates the approximate speed at which the point travels, categorizing speed as "zero," "slow," "medium," or "fast." Parameters specifying these cutoff values were determined through extensive testing. Finally, the module generates a four-bit velocity output describing one of sixteen possible outputs (four directions, four speeds).

2.1.4 `xvga.v`

The XVGA module, as shown in Figure 6 was provided by the course staff. It generates five signals that are used extensively in the project: hcount and vcount reference the current active pixel; hsync, vsync, and blank are timing signals that allow the FPGA to create an interlaced 60-Hz video output on the screen. The hsync, vsync, and blank signals must be delayed by certain amounts depending on the latency of pixel generators such as the game.

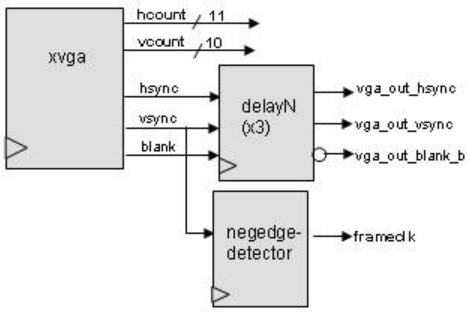

Figure 6: XVGA Module

2.1.5 Modified Video Modules

(Eddie)

A collection of helpful modules is available for student use on the 6.111 web site (`http://web.mit.edu/6.111/www/f2005/handouts.html`), including a package that uses a ZBT RAM (zero-bus turnaround random access memory) buffer to generate a 1024 by 768 black-and-white image from NTSC input. These modules, in their original form, produce an 8-bit intensity value for each pixel. The modules have been modified, as shown in Figure 7, to generate a 24-bit RGB value for each group of four pixels; although image quality is drastically reduced, the amount of information stored in the ZBT does not change and a color image is more useful for the project. The controller processing unit takes as input the RGB data produced by the group of video modules.

2.2 Game Modules

The Super Breakout game is made up of a main module and numerous submodules. The game was built in several stages. First, the pong game from Lab 5 was modified so that the paddle was now moving across the bottom of the screen instead of up and down the left side, and the originally square puck was changed to a circular ball.

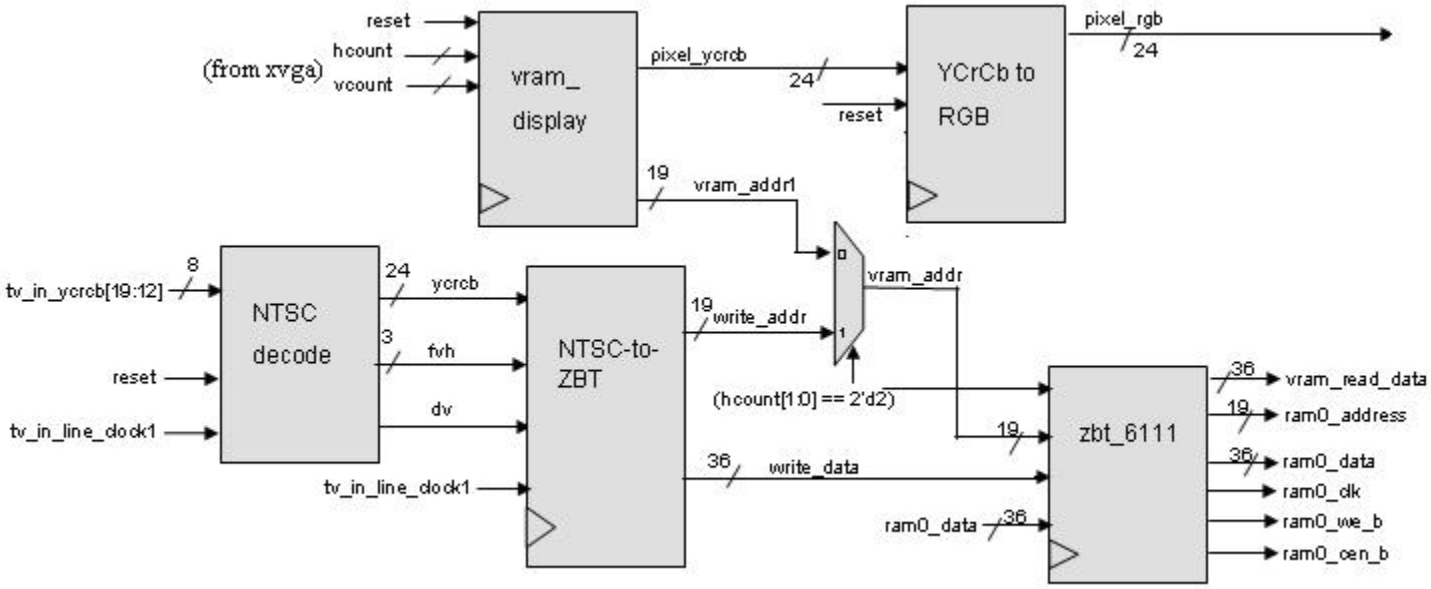

Figure 7: Block Diagram for Camera to RGB Signal Generation

Then, bricks were generated on the 1023 by 768 pixel screen from a 1 by 192 array containing information for the 8 by 24 grid of blocks. Collision detection was implemented to complete the basic game.

From there, the other features were added in one by one–different types of bricks, game-affecting objects, score, levels, multiple lives, changing ball speeds, and intertitles for different points in the game.

This section details each of the game modules; all Verilog code is included in the appendix.

2.2.1 `superbreakout.v`

(Irene)

The `superbreakout` module is the main module in which the game comes together. Most gameplay updates happen in a series of less than 15 clock cycles after the vsync signal goes low at the end of a frame.

The game logic begins by observing where the ball would move in the next frame if there were no bricks in its path and it was only bouncing off the walls and paddle; it takes the ball's current coordinates `ball_x` and `ball_y` and looks at the x and y velocity components `ball_xvel` and `ball_yvel` to generate the coordinates of the anticipated move, `check_x` and `check_y`. The ball velocity's x and y components are considered independently. For each component, the game checks for several conditions. If the ball is currently touching the top or side walls, it "bounces" in the other direction by moving the appropriate `xspeed` or `yspeed`. Otherwise, if the ball's next move would send it into or past the wall, the ball only moves as far as the surface of the wall (in which case, if there are no bricks, the first condition is met in the next move and the ball will bounce). If neither of the previous two conditions are met, the ball is sufficiently far enough from the wall and `ball_x` or `ball_y` just shifts by `xspeed` or `yspeed`, keeping the same velocity.

The case where the ball is traveling downward (`ball_yvel` is 1) requires that slightly different conditions be checked since the ball may hit the bottom of the screen or bounce off the paddle. If the ball touches the bottom of the screen, the player loses a life. If the next downward movement of the ball will take it past the top of the paddle, the x-coordinates of the ball are checked to see if they are within the bounds of the paddle, in which case the ball will bounce off the paddle. Otherwise, `check_y` is the current `ball_y` increased by yspeed.

Once established, `check_x` and `check_y` can be used to find the grid coordinates, `check_gridx` and `check_gridy`, of the anticipated move, using the `block_detector` module. In turn, the grid coordinates are used to calculate the address for the memory location containing information for that block. Since the 192 memory elements for the blocks in each level are arranged by rows, the address for a certain `check_gridx` and `check_gridy` would be `check_gridy` $<< 3 +$ `check_gridx`. At the

same time, the paddle speed for this frame is set according to how fast the controller is being moved.

The information for the block of the anticipated move is stored when it is available from the brick memory, in `check_brick_dout[3:0]`. In case this next block is situated diagonally from the current block, the information for the vertically and horizontally adjacent blocks are also retrieved and stored, in `vert_brick_dout[3:0]` and `horiz_brick_dout[3:0]`, respectively. The information for these blocks is used for collision detection.

Collision detection is a key part of the superbreakout module, as it determines whether the anticipated move is feasible. The simplest case is when the block of the anticipated move is the same as the current block containing the ball, in which case there is guaranteed to be no collision. In other cases with no collision occuring in the next frame, the ball can also complete the anticipated move, with `ball_x` and `ball_y` simply taking on the values of `check_x` and `check_y`, respectively. For the cases where the anticipated move is located in a vertically or horizontally adjacent block which contains a brick, the y-velocity is reversed if the current block's y-coordinate is not the same as the y-coordinate of the block containing the anticipated move, and the x-velocity is reversed if the ball travels to a side block. For example, if `ball_gridy` is not the same as `check_gridy`, the ball wants to move vertically and this will make it collide with the vertically adjacent brick; the game logic therefore stops the ball where it is for a frame and reverses its y velocity so it "bounces" off that brick.

The "diagonal case," where the anticipated move is to the block located diagonally from the current one, is more complex–different combinations of bricks located in the diagonally, vertically, and horizontally adjacent blocks warrent different collision behavior, as seen in Figure 8. A collision will occur if either the diagonal block has a brick or if both the horizontally and vertically adjacent blocks contain bricks.

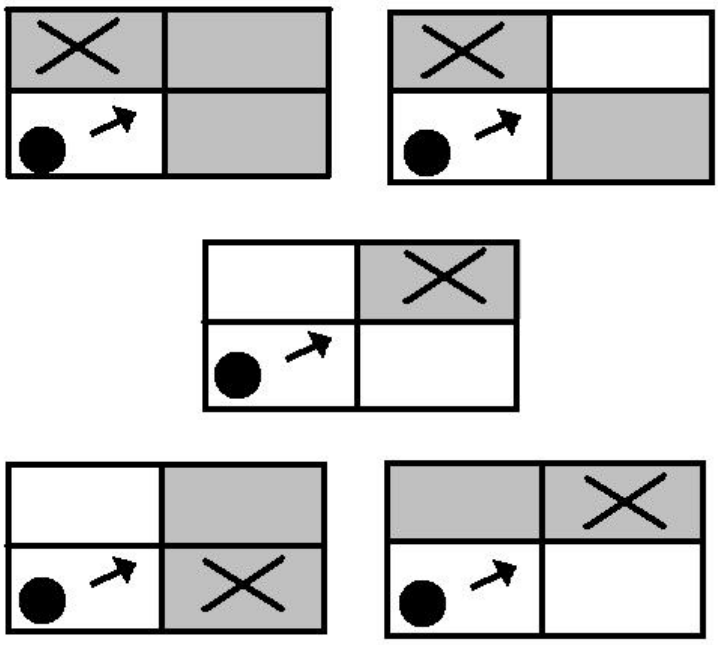

Figure 8: Collision detection is more complex in the case where the ball's anticipated move is to a block located diagonally adjacent from the current one. The 5 different cases and arrangements of adjacent bricks are shown above; gray boxes mark blocks containing bricks, the X denotes the brick that gets broken in the next frame.

1. If both horizontally and vertically adjacent blocks contain bricks, then the expected behavior is for both of these bricks to break; the diagonal brick will not break even if it exists. To achieve the expected behavior, the vertical brick is broken first and the ball's y-velocity is reversed. This way, in the next frame, the horizontally adjacent brick will break and the ball will reflect from the corner into which it bounced.

2. If only the diagonally and vertically adjacent blocks contain bricks, the ball will make the anticipated move to the diagonal block and break the brick that is there. The ball's y-velocity is reversed.

3. If only the diagonal and horizontally adjacent blocks contain bricks, the ball breaks the diagonal one, and the ball's x-velocity is reversed.

4. If out of the 3 adjacent blocks (in the direction of the ball's current x, y

velocities) only the diagonally adjacent block contains a brick, then the ball breaks that brick and reflects off the corner. In other words, both the ball's x and y velocities will change in the next frame.

Only the path traveled by the center of the ball affects the way the ball collides with bricks. If a part of the ball's outer rim overlaps with a the corner of a brick as it is moving, the system will not detect a collision. The black borders around each brick (2 pixels per side) as well as the relatively small size of the ball (4 pixel radius) make this behavior less apparent during gameplay.

Another trick used in this game's collision detection is holding the ball's x and y coordinates at the same value for a frame when the anticipated move generates a collision, instead of shifting the ball all the way to the edge of the brick with which it is colliding. Especially with ball speeds that change when the ball contacts the paddle, shifting the ball to be perfectly tangent to the brick is challenging. At the refresh rate of the game, this shift is unnoticable, so the game is designed such that only the appropriate x and/or y velocities are changed during frames in which collisions occur.

In the event of a collision, the color of the brick that breaks is used to determine what the next color should be using the module `det_brick_din`; this information is written back to the brick BRAM. Also, game-affecting objects are dropped if bricks of the corresponding color are dropped: red bricks drop red bombs, green bricks drop green expanders, and blue bricks drop blue life capsules.

Game-affecting items are also be dropped if fewer than four breakable bricks are left on screen, in which case bombs fall every 128 frames from "random" locations at the top of the screen. When there is only one breakable brick left, bombs drop at a rate of once per 32 frames, or about one every half second. The x-coordinate of the grid from which these bombs are scattered is determined by incrementing a three-bit

variable by 5 after each bomb is dropped. Since the grid is 8 blocks wide, adding 5 to `drop_gridx` prevents the bombs from all falling in a one area. The positions of all currently displayed game-affecting objects are updated when `update_clock` is pulsed high. All game-affecting objects are managed by the `drop_brick_object` module.

If the paddle was hit by a bomb or expander in the previous frame, it is shortened or lengthened accordingly. The paddle width can be shortened until the paddle disappears entirely, but cannot grow to be past the initial paddle width. The amounts the paddle shrinks and grows are specified by the parameters `paddle_shrink_amount` and `paddle_grow_amount`. The paddle also moves left or right according to the input from the left/right buttons on the FPGA or from the controller input.

The last step to updating the state of the game is to grant the player an extra life if a life capsule hit the paddle in the previous frame. The maximum number of extra lives is 5.

Besides updating the regular state of the game at the end of every frame, the superbreakout module also handles game behavior related to system resets and level changes. Upon a reset, the game returns to the first level and the player can start again with 5 lives. When a user forces a level change by pressing the down button on the FPGA, the game cycles to the next level. Since the game has 4 levels, the brick memories are 192*4=768 elements long, each element containing 4 bits (1 bit of display, 3 bits of color). During gameplay, the BRAM is accessed in blocks of 192 elements, with each block representing a level. Moving to the next level involves shifting the address to the BRAM by a multiple of 192 corresponding to the current level so that the game is interacting with the correct part of the memory.

Resets and level changes require the bricks on screen to be regenerated; here is where the read-only brick memory comes into play. The `brick_reset` signal goes high, and information from the read-only BRAM is copied over to the read/write

BRAM one element at a time. Since it takes time for the information from the BRAM to appear after a new address has been set, the brick-reset phase alternates between a write cycle and read cycle. During the write cycle, the next address for the read-only memory is generated, and the data available from the last element accessed in the memory is saved. If the copied element contained the brick information for a non-magenta brick, a variable that keeps track of the initial number of bricks on the screen is incremented. Then, during the read cycle, the address of the element in the read/write memory to be written is set and the write enable signal is pulsed high so the information can be copied over.

The variable that counts the inital number of breakable bricks (i.e., non-magenta bricks) is used to determine the number of bricks that need to be broken to complete the level. Each time a red brick is broken, this number is decremented by 1 and outputted as num_bricks_left; when 0 bricks are left, it is time to move onto the next level.

When the player beats the last level, superbreakout shows the "You Win" intertitle and waits for a reset signal to start the next game; likewise, when a playerloses his last life, the game stops as expected. In other instances where a player fails to deflect the ball with the paddle, losing a life resets the level in terms of moving the paddle back to the middle of the screen, starting the ball at the center of the paddle, and clearing all game-affecting objects from the screen. Already-broken bricks are not regenerated again at this point.

2.2.2 `block_detector.v`

(Irene)

The screen is divided up into a grid of 8 by 24 blocks: given a pixel as denoted by `hcount` and `vcount`, the `block_detector` module returns the coordinates of the block

in which the pixel is located in the form of `grid_x` and `grid_y`. Each block is 128 by 32 pixels and either displays a brick or does not. Performing this change of coordinates allows the game to ignore the individual pixels and work with bricks instead, which is much more intuitive to implement.

The block detector module works by selecting bits out of `hcount` and `vcount`. Since `hcount` needs to be divided by 128, the x-coordinate of the grid is just `hcount[9:7]`. The top bit of hcount is ignored, as only 1024 pixels are displayed on screen. `vcount` needs to be divided by 32, so it is right-shifted by 5 bits and the y-coordinate of the grid is merely `vcount[9:5]`.

Several block detectors are used within the `superbreakout` module: to get the coordintes of the ball's original pixel location, to get the coordinates of the block the ball would be in after its anticipated move, and to get the coordinates of the block the current pixel is in for the `display` module.

2.2.3 `circle.v`

(Irene and Eddie)

The `circle.v` module displays an 8x8 circle from a ROM, and is used for the ball, bombs, and life capsules in the game. The default circle color is white, but bombs are red and life expanders are blue.

The `display` input controls whether or not the circle is displayed on screen. While the ball is always displayed, bombs and lifecapsules are only shown when they are dropped from broken bricks.

To avoid glitches, the circle module loads the data from the ROM into a 64-bit register when `vsync` goes low at the end of each frame, then reads this data in the next frame. At the end of every frame, the 6-bit counter and the ball data in the register are cleared, and the 1st address from which to read the ROM is established.

The module then begins alternating in and out of a read cycle. During the read cycle, the address is set to access the next element of the ROM, and the `data_out` from the ROM for the last element is saved into the register. The clock cycle between each `read_cycle` gives the memory time to fetch the data from the new address. Contents of the register are shifted up one "slot" at this time so that by the time the whole ROM has been read, its data is loaded into the 64-bit register.

When the next frame is displayed, the circle is displayed using the ROM data copied to the register. When `hcount` and `vcount` are within the 8x8 square to the right and below the x-y coordinates of the circle, the last bit of the register is checked. If the last bit is a 1, the `pixel` output is assigned to the circle's color; otherwise, the pixel displays black. After this last bit has been looked at, the register's data is right-shifted so the next bit of information is ready to go.

2.2.4 `det_brick_din.v`

(Irene)

The `det_brick_din` module specifies what the next color of a block should be if a brick of the specified color is hit. Red bricks turn break and turn black, yellow bricks turn red, green bricks turn yellow, teal bricks turn green, blue bricks turn teal, and magenta bricks stay magenta (they never break). This look-up table is accomplished with a case statement that takes `color`, the current brick color, and outputs `next_brick_color`, the 4 bits to be written to the brick memory. The output is 4 bits because the top display bit has been appended to the bottom 3 color bits for convenience.

This module is used in `superbreakout` when it has been determined that there is a collision, and the block's information needs to be updated.

2.2.5 detect_bombs.v

(Irene)

detect_bombs is a "general-purpose" module that detects when the pixels of one specified object overlap with pixels of a second specified object. This module is used in several places to generate signals that indicate when bombs, expanders, and life capsules hit the paddle. It takes as inputs output1_pixel[2:0] and output2_pixel[2:0], two objects' pixel objects, and first ORs the bits of each one together (to determine if either one is black) before AND-ing the results and assigning it to the variable hit. hit is therefore 1 when, at a certain pixel, both objects are overlapping.

In superbreakout.v, the variables shorten_paddle_b1, shorten_paddle_b2, shorten_paddle_b3, shorten_paddle_b4, shorten_paddle_b5, lengthen_paddle, and life_up are used determine whether or not game-affecting objects have collided with the paddle in the last frame; these variables were created because although the output of detect_bombs goes high when there is an overlapping pixel between the two objects, it returns low as soon as hcount and vcount are at at non-overlapping pixels of the colliding sprites. During each frame, these variables are OR-ed with the output of detect_bombs, so once they are high they stay high for the rest of the frame.

2.2.6 display.v

(Irene)

This module creates the bricks from the contents of a dual-port RAM which contains information for every block, and outputs the combined pixel outputs of all objects on screen. The module also handles switching to and from intertitles, as well as the ball-fading feature.

Inputs to display.v include hcount and vcount; grid_x and grid_y, which are

the coordinates of the block hcount and vcount are currently in as determined by an instance of `block_detector`; `brick_data_out`, which is the output from the brick memory containing information about the current block; the pixel outputs for the bombs, expander, life capsule, ball, and paddle; `fade_on`, which controls whether the ball darkens as it moves toward the bottom of the screen; and `title_on`, which determines if an intertitle is supposed to be displaying on screen. The module outputs an address to the brick memory as generated from `grid_x` and `grid_y`, also generates a full 24-bit color game output to the VGA.

The bricks are drawn from logic. Each block has a 2-pixel border around it; these outermost 2 pixels around the block are determined by looking at `hcount[6:1]` and `vcount[4:1]`, because each brick is 128 by 32 pixels wide. If these bits are all 1's or all 0's, then the current pixel is one of the 2 pixels at the right-end/bottom of the brick or at the left-end/top of the brick. The 8-pixel "rim" for the hollow magenta blocks with a similar concept, looking to see if `hcount[6:3]` or `vcount[4:3]` are all 1's or 0's. As `hcount` and `vcount` move across the screen, the address to the brick memory is recalculated every time the coordinates enter a new brick. If the brick data shows that a brick is present, the pixel output is kept at the specified color. There are 2 exceptions: all brick borders and the non-rim part of magenta bricks are always black.

If `title_on` is high, it is time for an intertitle to appear. These fade in over the course of around 4 seconds. If `fade_on` is high, then the ball changes colors as it moves up and down the screen, staying a bright white at the top but fading to a dark gray as it approaches the paddle.

2.2.7 `drop_game_object.v`

(Irene)

`drop_game_object.v` handles the dropping and managing of bombs, expanders, and life capsules in the game. There can be up to 5 bombs, 1 expander, and 1 lifecapsule on the screen at a time. Bombs shorten the paddle when they hit it, expanders lengthen the paddle, and lifecapsules grant the player an extra life.

The module makes dropping objects in the game very easy–in `superbreakout.v`, the signals `drop_bomb`, `drop_expander`, or `drop_lifecapsule` are pulsed high when the respective objects are supposed to be dropped from a brick. `grid_x` and `grid_y` are the only other inputs that need to be set, and these specify the grid from which the game-affecting object is supposed to fall.

This game-affecting objects manager has numerous other inputs and outputs. The `update_clock` signal is pulsed high once a frame from the superbreakout module, when it is time for the all the game-affecting objects currently displayed on screen to update their position for the upcoming frame. The signals `shorten_paddle_b1`, `shorten_paddle_b2`, `shorten_paddle_b3`, `shorten_paddle_b4`, `shorten_paddle_b5`, `lengthen_paddle`, and `life_up` are high if the bomb, expander, or lifecapsule sprites intersected the paddle sprite in the last frame. When objects collide with the paddle, they take effect by changing the length of the paddle or by giving the user an extra life. The bomb, expander, and extra life display signals indicate whether those objects are currently falling down the screen. The `drop_game_object` also generates the x and y position coordinate for the top left corner of each of these objects' sprites, since it controls how these objects fall.

Falling objects disappear when they either hit the paddle or get near the bottom of the screen. Also, on a reset, level change, or when the player loses a life, the screen is cleared of all bombs, expanders, and life capsules.

2.2.8 expander_blob.v

(Eddie)

This module displays the icon for the expander, which lengthens the paddle if caught. `expander_blob.v` is very similar to `circle.v`–at the end of every frame (when vsync goes low), ROM data is loaded into a register and then read from the register during the next frame as to avoid glitches. The difference between `circle.v` and `expander_blob.v` is that they read from different ROMS. The expander is 16x8 pixels, instead of being an 8x8 pixel block. Whereas the bombs and life capsule look like different colored balls, the expander is represented by two outward-pointing arrows. For details on how the contents of the ROM are read into the register before the next frame is displayed, see `circle.v` above.

2.2.9 game_over.v

(Irene)

The game over module raises the `done` signal the end of a level, when there are no more breakable bricks left on the screen. It also outputs the number of breakable bricks left on the screen, `num_bricks_left`, which is used in the "random bombing" in `superbreakout.v`. (When there are 3 or fewer bricks left in the level, bombs start to fall from the top of the screen every 128 frames; and then when there is only 1 brick left, the "acid rain" feature drops a bomb every 32 frames).

The module takes as inputs `init_num_bricks`, the initial number of breakable bricks on the screen; `collision`, which goes high whenever the ball hits a brick; and `color`, which is the color of the brick that was hit.

Whenever `collision` goes high and the brick `color` is red, the number of bricks left is decremented – as soon as this number `num_bricks_left` reaches 0, the level is over and `done` goes high. A `reset` signal sets the number of remaining bricks back

the initial number of breakable bricks in the level as determined after the brick reset phase in `superbreakout`.

2.2.10 `intertitle.v`

(Eddie)

At certain points in the game, gameplay pauses and full-screen image appears on the monitor. These intertitles are useful and entertaining indications of the player's progress beyond the visible LED signals on the FPGA. For example, when the player completes Level 3, a picture materializes announcing that the player has successfully completed the level and that he or she should press a button to continue.

The game can display any of the six possible pre-loaded images, depending on certain game milestones. As these milestones occur, this module loads the appropriate image from BRAM and generates the appropriate pixel output. This image remains loaded until the next milestone; at this point, the previous image BRAM becomes disabled and the new BRAM enable switches on. Each BRAM stores a 256 by 192 pixel 8-color (3-bit) bitmap image, generated in Microsoft Paint and converted to a .coe file using Matlab (the .m file can be found at `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12437&objectType=file`). The six images correspond to the following self-explanatory states: Pre-Game, Level 1 Complete, Level 2 Complete, Level 3 Complete, Game Over, and Game Complete.

Each bitmap pixel is up-sampled to a 4 by 4 pixel block by the intertitle module, generating a 1024 by 768 pixel image for the VGA output. This process is doubly important as it also allows the BRAM time to fetch each image pixel. The games display module contains code that creates a fade-in effect whenever an intertitle becomes visible during normal game operation.

2.2.11 `modblob.v`

(Irene)

This module is modified off the `blob.v` sprite generator given to us for Lab 5; `modblob` makes it easy to change the rectangle's `width[8:0]`, `height[4:0]` and `color[2:0]` because it takes them as inputs instead of setting them as parameters. The paddle grows and shrinks when hit by expanders and bombs, respectively, so it needs to change dimensions in the middle of the game.

Other inputs to `modblob` are display, which determines if the sprite is shown on screen (the paddle is always displayed); `x` and `y`, the x-y coordinates of the sprite's top left corner; and `hcount` and `vcount`, the current displayed pixel.

As in `blob.v`, when `hcount` and `vcount` are in the rectangle of size `width` and `height` to the bottom right of the sprite's top left corner, the outputted `pixel` has the specified color. Otherwise, the displayed pixel for the sprite is black.

2.2.12 `recorder.v`

(Eddie)

The Recorder module interfaces with the AC97 audio chip on the FPGA and maintains a BRAM that allows the user to record, play back, and clear a brief sound clip. While the user holds down the record button (Button Up), he or she can speak into a microphone and record a very short audio waveform. Whenever the playback signal is asserted high, the entire sound clip plays. This playback cannot be interrupted by another playback request. This module is used to generate and store user-defined collision sound effects in Game Mode.

2.2.13 `score.v`

(Irene)

`score.v` calculates the score – every time the ball hits a non-magenta brick, the score increments by the current speed. Inputs are `reset`, which resets the score back to 0; `collision`, which is 1 whenever the ball hits a brick; `pspeed`, the initial speed of the ball as set by `switch[3:2]`; and `color`, the color of the brick that was hit when collision when high.

Whenever there is a collision, if the color of the brick is not magenta, then the score in increased by pspeed. In the *Super Breakout* game, the current score is displayed on the lower 32 bits of the LED hex display, and the high score is shown on the upper 32 bits (see `LEDdisplay.v`).

2.2.14 `speed.v`

(Eddie)

This module changes the ball's velocity in the horizontal direction (x-velocity) depending on the paddle's speed on ball-paddle impact. Essentially, this module simulates a frictional coefficient on the paddle: if the paddle and ball are moving in the same direction, the ball's x-velocity increases; if the two are moving in different directions, the x-velocity decreases. The amount of change is directly proportional to the paddle's speed. The most important function of the speed module is to provide a varied gameplay experience.

2.3 System

2.3.1 Hex2Dec and Tencounter

(Eddie)

The game generates a 32-bit score output on the FPGA that the player can view at any time. However, the provided numerical LED display module only outputs a 32-bit input as an 8-digit hexadecimal number. Most users cannot quickly convert a

hexadecimal number to its integer equivalent, so the Hex2Dec module converts the 32-bit hex input to an 8-digit integer, encoded such that each digit is represented as a 4-bit value. As an example, the Hex2Dec module would convert the 8-bit hexadecimal number 8h5A to 8h90, because 90 is the decimal equivalent of 5A.

Each tencounter stores a single decimal digit. When a tencounter's input is asserted high on a rising clock edge, the tencounter increments its internal 4-bit output register by one. If cin is high and the internal register value is 4'h9, the register resets to zero and cout gets asserted high for one clock cycle. When several of these blocks are connected in series, each cout acts as a counter that pulses at one-tenth the rate of its own input.

The Hex2Dec module connects eight tencounters in series and pulses one input into the system every clock cycle for x cycles, where x is the size of the 8-bit input. The eight 4-bit tencounter outputs are concatenated to create a 32-bit output as shown in Figure 9. Therefore, Hex2Dec requires (x / 27,000,000) seconds to generate a correct output, using a 27-MHz clock.

2.3.2 LED Display

(Eddie)

During gameplay, the user should be able to view his or her score, while in other modes of operation the user might want to view the current mode or check the current high score. The LEDdisplay module controls the FPGA LED display unit, allowing the project and user to switch between hexadecimal and character output at any time. The 6.111 handouts include modules that display each type of output, but no method to switch between the two. In addition, the character output depends on the current mode of operation and game state.

LEDdisplay's primary function is to act as a two-input multiplexer, controlled by

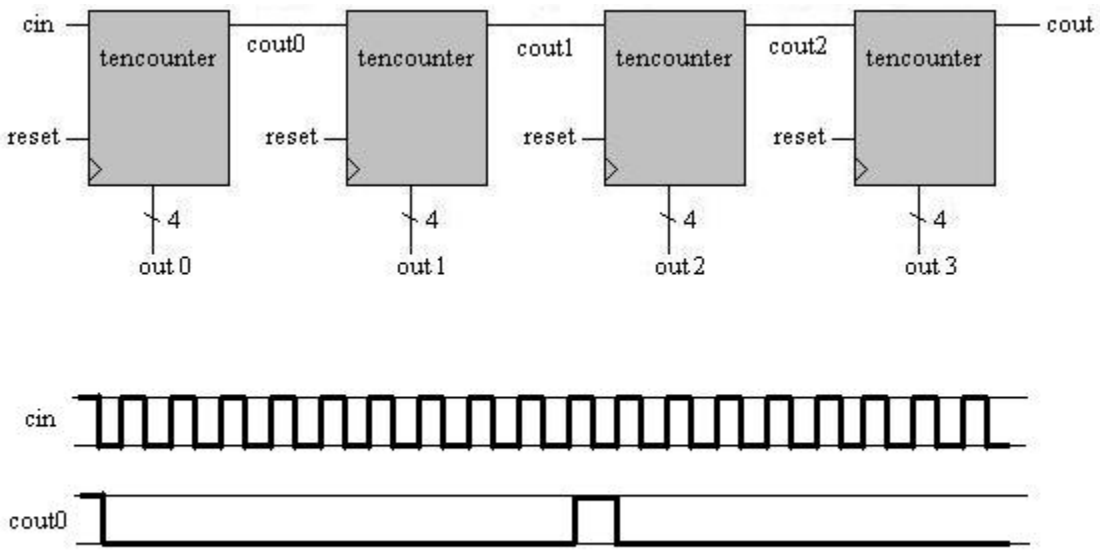

Figure 9: Timing and Structure of Hex2Dec

the "menu_display" input bit. If menu_display is high, characters are visible on the display; otherwise, a hexadecimal number appears. In the default case, if "Game Mode" is active and the game has not ended, the current score and high score appear as the right and left eight digits, respectively. Otherwise, the current mode of operation or game-ending state (win or loss) appears as ASCII character output. The user may press Button 0 at any time to switch between the two types of output.

The module also determines which character string to output at any point, depending on the mode of operation: "Filter Mode," "Paint Mode," "Camera Mode," and "Game Mode." Note that "Game Mode" is only visible when the user presses the display-toggle button during game operation. In addition, if the player beats the game or loses, "You Win!!!" or "Game Over :-(" are visible, respectively, in addition to the on-screen intertitles.

2.3.3 NegEdgeDetector

(Eddie)

When this simple module detects a negative edge, it generates a quick pulse. For example, NegEdgeDetector pulses the frame clock whenever the vsync signal from the XVGA module transitions from high to low.

2.4 Paint Mode

2.4.1 Block Test

(Eddie)

This module generates the output for Paint Mode, which is also used in Camera Mode. In Paint Mode, the user moves a white-square cursor around a black background. By setting three of the FPGA switches, the user can define a 3-bit drawing color. The cursor will then leave a trail in the defined color. (This process is detailed in the TrackerBlob module description.) The canvas can be cleared at any time by pressing Button 3.

Two sub-modes of operation are possible in Paint Mode: Position Mode, which simply places the cursor at the point calculated by the centerofmass module; and Velocity Mode, which moves the cursor around the screen according to the direction and speed determined by the COMVelocity module. These sub-modes creatively demonstrate the functionality of the controller modules, and allow the user to test the controller before playing a game. In addition, Camera Mode combines the RGB camera output with the paint canvas for a fun and entertaining mini-game.

2.4.2 HighForFrame

(Eddie)

This module holds an input high for one full frame-clock cycle, as shown in Figure 10. This function is generally used to hold an asynchronous user input stable for an extended period of time.

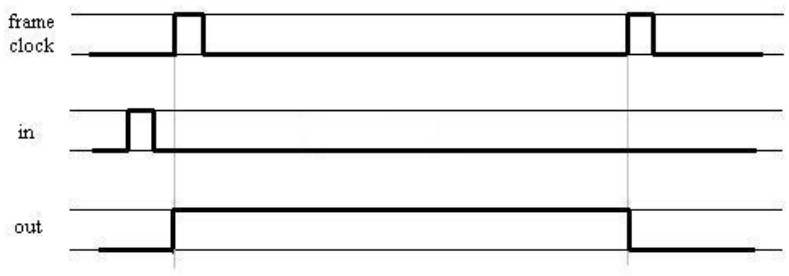

Figure 10: HighForFrame timing diagram

2.4.3 TrackerBlob

(Eddie)

TrackerBlob is a specialized sprite controller that remembers the path of the cursor sprite in Paint Mode and Camera Mode. The module updates a BRAM that stores 3-bit color information for each 16 by 16 pixel block on the 1024 by 768 display.

3 Testing and Debugging

3.1 Eddie

One of the biggest problems I faced during the controller implementation step was modifying the pre-existing camera-to-ZBT code, because I had to quickly learn the inner-most details of over ten modules so that I knew exactly what to modify to convert our YCrCb camera output to RGB. Once I got the RGB information, I spent days playing with the FPGA switches in order to determine appropriate cut-off parameters for the filter module. By accident, I discovered that the analog filter

provided by tuning the camera out-of-focus significantly removed the amount of noise we had to process. I shared this finding with other members of the course, who I believe were all able to utilize this camera feature.

My center-of-mass module turned out to be far less time-consuming than I had anticipated, although the dividers took some time to generate. Unfortunately, these two dividers (one for each axis) quadrupled the compile-time of our project, which made further testing in other areas extremely irritating. The dividers also experienced occasional "Place and Route" technical difficulties, depending on how the Xilinx tools decided to wire the project. These difficulties usually worked themselves out in later compilations. Due to the linear nature of my design, I could not test any module without completing all of the previous modules, and errors in supposedly flawless designs became immediately apparent when I tried to use those outputs in later modules. As the number of modules in my chain grew, I found myself in a negative feedback loop that led to surprising improvements in my original designs. I believe that the linearity of my project was invaluable to the design process, but only because my initial plans were admittedly inefficient.

After finishing the controller, I created some auxiliary features for the project, including "Paint Mode," extra levels, the LED displays, intertitles, and sound effects. By far the most difficult module to implement was the intertitle capability. I eventually used a MATLAB script to convert a Microsoft Paint bitmap file to a COE memory initialization file, but the process of discovering the MATLAB script, creating the images in Paint, and figuring out the display timing delays took me approximately thirty hours to complete. In retrospect, the reason this process took so long was because I eventually realized that in order to create a clear but space-efficient image, I had to start with a 256 x 192 pixel image (1/16 screen area) and expand each pixel to a 4 x 4 pixel block. Once I overcame this hurdle, the rest

was relatively easy.

My favorite auxiliary module is the hexadecimal-to-base-10 conversion unit. While this module is surprisingly simple to understand, discovering the algorithm and perfecting the process became an arduous ordeal. At one point, I calculated that a 64-bit hex number would require several millenia to finish conversion, and shortly thereafter settled with a 32-bit conversion system. I used ModelSim to test the functionality before integrating the code into the rest of the project, and found that I had no problems with the integration step.

Finally, one of my larger tasks was to combine the controller, game, and auxiliary modules into a single final project. I was amazed that this step worked almost perfectly on the first try, except for minor glitches that I found without difficulty the second time around. I have no doubt that this ease was entirely due to the fact that Irene and I had decided early on to modularize our code as much as possible; in other words, we strictly limited the amount with which the game and controller depended on each other. In terms of the overall structure of our project, I believe that modularization was our best design decision.

3.2 Irene

First and foremost, my largest problem was pipelining. Besides that, the only other serious issue I really had was collision detection.

Initially, I tried collision detection by shifting the ball to the edge of the brick it hit, in the case of a collision. This method proved glitchy and unreliable, especially when the ball moved diagonally and, although it appeared to hit two bricks, basically skipped right through the obstacles. I managed to get standard collision detection (clearly horizontal or vertical collisions) working almost immediately, but almost a month passed before I was able to implement diagonal collision detection. I

experimented with several methods, including pixel overlapping. However, these methods all let to strange and unanticipated behavior, so I decided to scrap that idea and start over once I had cleared my mind. I realized that the best way to implement diagonal collisions was to go case-by-case through all the different situations and instantiate a priority system, whereby the ball could never collide with more than one brick at a time. This system worked because if the ball hit one brick in the diagonal case, it would inevitably bounce off that brick and hit an adjacent brick if it existed. (See the figure in the Super Breakout module description.)

Another issue I had was with sprite display. For weeks, my sprites were glitchy and my bricks did not display properly. I eventually discovered that the problem lay with the sprite BRAMs; I was not allowing enough time for the BRAMs to access and write information, and as such messed up the whole program. I tested several different combinations of latency cycles across the board, and eventually solved the sprite display problem.

Pipelining fixed my display issues but created serious problems when I tried to modularize the Super Breakout module. The problem with this module is that it is extremely difficult to modularize due to the linear nature of the design as well as because many modules share the same registers and inputs. I decided to implement a large linear FSM, with several empty states in between to create the necessary latency cycles to solve my display problems. While the code for superbreakout.v is lengthy, I found that this method worked best both for debugging and for clarity, since the counters clearly indicated what would happen on the next clock cycle.

4 Conclusion

By far, the best idea we had was to modularize our code. As we mentioned in the "Testing & Debugging" section, isolating the controller and the game allowed the two of us to work completely independently for more than a month, but we needed less than an hour to combine the final versions of each part. The modularization was not exactly as clean as we had hoped, because the FPGA inputs and outputs, as well as the various modes of operation, strongly interacted with many of the project modules. However, we believe that modularization was a very important part of our successful group collaboration, and we recommend this strategy to future teams.

We found ourselves with slightly more free time at the end of the semester than we had originally anticipated. This freedom caused several problems, however. We reached an addictive point where we could not stop adding extra features to the game. We eventually stepped back and declared a moratorium on additions, although perhaps by then it was already too late; for the last few days of the project, we decided to focus only on fixing the small glitches and timing issues that plagued the whole system, and we believe that we did so successfully.

Pipelining turned out to be our biggest problem, especially when a lack thereof severely hampered our efforts to display sprites on the screen. We spent weeks trying to figure out why the ball did not display correctly in our game. Interestingly, once we found the solution to our problem – pipelining – we only needed a few hours to fix all the bugs. We believe that this would not have been possible without the in-class lectures on timing and pipelining, and the real-world exposure to a theoretical problem we discussed in class helped to solidify the concept in our minds. Instead of understanding the problem and being able to draw pipelined diagrams (a la 6.004), we were placed in a situation where we actually had to implement the material we had learned. For both of us, this realization was one of the most interesting parts of

the project, because we were finally able to look past the video game and see the engineering and educational implications of our project.

Although we learned a great deal during the course of the project, we had a lot of fun implementing our design. Even though the two of us essentially cohabited in the laboratory (along with Gim Hom, Cassie Huang, Javier Castro, and Kevin Miu, our unflagging sources of moral and technical support), we did so only because we really enjoyed what we were creating. Of course, when the final product is a video game, testing becomes far more interesting! In the end, the two of us emerged from this experience looking forward to our next design projects, whatever they might be.

A Verilog Code

The following appended code is the Verilog that we wrote or modified for the entire project, in alphabetical order. Each of these modules is described in further detail in Section 2 of this report.

A.1 block_detector.v

```
// block_detector.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan

/*
   This module divides the screen up into a grid of 8 by 24 blocks:
   given a pixel, the module returns the coordinates of the block in
   which the pixel is located. (Each block is 128 by 32 pixels and
   either displays a brick or doesn't.) Performing this change of
   coordinates let's you ignore the individual pixels and work with
   bricks instead, which is much more intuitive.

   Inputs:
      hcount, vcount (from the xvga module; which pixel are we
         currently working with)

   Outputs:
      grid_x, grid_y: the column and row of the block on the screen in
         which hcount and vcount are located
*/

module block_detector(hcount, vcount, grid_x, grid_y);
   input [10:0] hcount;
   input [9:0] vcount;
   output [2:0] grid_x;                // 0 through 7, for the 8 bricks across
   output [4:0] grid_y;                // 0 through 23

   assign grid_x = hcount[9:7];        // divide hcount by 128 and the top bit
                                       //  isn't used
   assign grid_y = vcount[9:5];        // divide vcount by 32
endmodule
```

A.2 block_test.v

```
// block_test.v
// 6.111 Fall 2006 Final Project
// Module created by Eddie Fagin

/* This module generates the output for "Paint Mode." The user moves a white
   square around a black background to paint a picture in seven possible
   non-black colors. You can paint in two possible modes: (1) position - the
   pointer moves to the current frame's center of light mass from the
   camera output; (2) velocity - the pointer moves relative to light motion,
   much like pushing a block around a room. In vel-mode, the square moves
   differently for each of the 16 possible velocity inputs from the controller,
   and allows the user to get a feel for the controller before playing a game.
   This is also useful when determining how far away the user should stand from
   the camera.

   Inputs
      vclock: 65-MHz system clock
      frameclk: goes high at the end of each frame (negative edge of vsync)
      reset: system reset
      velocity: 4-bit speed and direction representation.
                See the code for more details.
      hcount, vcount: current VGA output position
      color: the three-bit draw color set by switch[6:4]
      mode: draw mode - if on, position ("mouse pointer") based motion.
            if off, velocity-based motion ("pushing a block")
      comX, comY: the current center of pixel "mass" for this frame,
                  used in position mode

   Ouputs
      pixel: determines the color of the current coordinates;
             eight possible values.
*/

module block_test (vclock,frameclk,reset,velocity,
        hcount,vcount,color,mode,comX,comY,pixel);
   input vclock;
   input frameclk
   input reset, mode;
   input [3:0] velocity;
   input [10:0] hcount,comX;
   input [9:0] vcount,comY;
   input [2:0] color;
   output [2:0] pixel;

   // Set up constants for resolution, block size and speeds.
   parameter HMAX = 10'd1023;
   parameter VMAX = 10'd767;
   parameter ball_size = 4'h8;
   parameter slow_speed = 4'h5;
   parameter normal_speed = 4'hA;
   parameter fast_speed   = 4'hF;

   reg [10:0] ball_x;
   reg [9:0] ball_y;
   wire [2:0] pixel;

   // This block creates a white "ball" sprite. It's actually a 16 x 16 block,
   //  calibrated from the top left corner.
   // Adjustments to ball_x and ball_y will move the block in two dimensions.
   trackerblob ball0 (ball_x, ball_y, hcount, vcount, vclock, reset,
                      color, ~(velocity[3:2] == 0), pixel);
```

```
      defparam ball0.WIDTH = ball_size;
      defparam ball0.HEIGHT = ball_size;
      defparam ball0.COLOR = 3'b111; // white

   always @ (posedge vclock) begin
      // On a reset, center the ball
      if (reset) begin
         ball_x <= (HMAX - ball_size) >> 1'b1;
         ball_y <= (VMAX - ball_size) >> 1'b1;
      end

      // Each frame, update block position. Checks boundary conditions so that
      //  block stays on screen.
      else if (frameclk) begin
         // position-based motion
         if (mode) begin
            ball_x <= 10'd745 - comX;
            ball_y <= comY;
         end

         // velocity-based motion
         // case blocks are in the order slow, medium, fast for each direction
         else
          case (velocity) // velocity = {speed, direction}
                     // speed:     (00) zero   (01) slow  (10) medium (11) fast
                     // direction: (00) up     (01) right (10) down   (11) left

            // move up
            4'b0100: ball_y <= (ball_y < slow_speed) ?
                               ball_y : ball_y - slow_speed;
            4'b1000: ball_y <= (ball_y < normal_speed) ?
                               ball_y : ball_y - normal_speed;
            4'b1100: ball_y <= (ball_y < fast_speed) ?
                               ball_y : ball_y - fast_speed;
             // move down
             4'b0110: ball_y <= (ball_y > VMAX - slow_speed - ball_size) ?
                                ball_y : ball_y + slow_speed;
             4'b1010: ball_y <= (ball_y > VMAX - normal_speed - ball_size) ?
                                ball_y : ball_y + normal_speed;
             4'b1110: ball_y <= (ball_y > VMAX - fast_speed - ball_size) ?
                                ball_y : ball_y + fast_speed;
            // move left
            4'b0111: ball_x <= (ball_x < slow_speed) ?
                               ball_x : ball_x - slow_speed;
            4'b1011: ball_x <= (ball_x < normal_speed) ?
                               ball_x : ball_x - normal_speed;
            4'b1111: ball_x <= (ball_x < fast_speed) ?
                               ball_x : ball_x - fast_speed;
            // move right
            4'b0101: ball_x <= (ball_x > HMAX - slow_speed - ball_size) ?
                               ball_x : ball_x + slow_speed;
            4'b1001: ball_x <= (ball_x > HMAX - normal_speed - ball_size) ?
                               ball_x : ball_x + normal_speed;
            4'b1101: ball_x <= (ball_x > HMAX - fast_speed - ball_size) ?
                               ball_x : ball_x + fast_speed;

            // default: no change (speed = 0)
           endcase
      end
   end
endmodule
```

A.3 centerofmass.v

```
// centerofmass.v
// 6.111 Fall 2006 Final Project
// Module created by Eddie Fagin

/* This module calculates the center of "mass" of
   certain non-black pixels on-screen each frame.
   The pixel colors fed to this module have already
   been processed by an intensity filter, so all
   all inputs are solid colors (red, green, or blue).
   For the purposes of our project, the input pixels
   are green. Therefore, this module finds the center
   of all green pixels. I used two divider modules
   (generated by CoreGen) that divides two 32-bit numbers.
   Every pixel has the same "weight", so I just summed up
   (separately) the  x- and y- coordinates of every qualified
   pixel and divided by the total number of such pixels.
   Inputs
      pixel: the 8-bit filtered and processed pixel from the camera
      x, y: the coordinates of the aforementioned pixel
             (on an imaginary 1024x768 screen)
      clk: 65-MHz system clock
      frameclk: goes high at the end of each frame
                (negative edge of vsync)
      reset: system reset
   Ouputs
      comX, comY: the current center of pixel "mass," updated every frame.
*/

module centerofmass(pixel, x, y, clk, frameclk, reset, comX, comY);
   input[7:0] pixel;
   input[10:0] x;
   input[9:0] y;
   input clk, frameclk, reset;
   output [10:0] comX;
   output [9:0] comY;

   reg[10:0] comX;
   reg[9:0]  comY;
   // accumulating registers
   // x: 11 bit values * at most 2^10 pixels.
   reg[21:0] comXinc;
   reg[19:0] comYinc;

   // Weight is just a count of the current number of active pixels.
   // center of mass = sum of values / weight
   reg[19:0] weight;

   // The two dividers, generated via CoreGen,
   //  continuously calculating a quotient.
   // Note: these things are HUGE! They take a long time to compile.
   wire [19:0] xremainder_not_used, yremainder_not_used;
   wire xrfd, yrfd;
   wire [21:0] xquotient;
   wire [19:0] yquotient;
   averager divX(comXinc,weight,xquotient,xremainder_not_used,clk,xrfd);
   averager2 divY(comYinc,weight,yquotient,yremainder_not_used,clk,yrfd);

   always @ (posedge clk) begin
```

```
      if (reset) begin
         comX <= 11'd0;
         comY <= 10'd0;
        {comXinc, comYinc} <= 42'd0;
        weight <= 20'd0;
      end

      // Each frame, update the center of pixel mass and reset the counters.
      // The (weight <= 4'hf) is simply a filter that leaves the COM
      //  unchanged if the number of considered pixels is less than 16.
      // (We assume that it's just noise.)
   else if (frameclk) begin
comXinc <= 22'd0;
comYinc <= 20'd0;
comX <= (weight <= 4'hf) ? comX : xquotient[10:0];
comY <= (weight <= 4'hf) ? comY : yquotient[9:0];
weight <= 20'd0;
end

// Each pixel is replicated four times
                //  (a.k.a my quick-and-dirty camera processing strategy)
//  so we can just consider every fourth pixel in our calculation.
else if (pixel && x[1:0] == 2'b00) begin
weight <= weight + 1'b1;
comXinc <= comXinc + x;
comYinc <= comYinc + y;
end
end
endmodule
```

A.4 circle.v

```
// circle.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan and Eddie Fagin

/*
   This module displays a 8x8 circle from a ROM. To avoid glitches, it
   loads the data from the ROM into a 64 bit register when vsync is
   low, then reads the data in the next frame. Default circle color is
   white.

   Inputs:
      display -- if 1, circle is shown on screen
      vclock -- 65 Mhz system clock
      reset -- reset signal
      frameclock -- pulses high at the end of every frame
      x, y -- top left coordinates of the 8x8 box around the circle
      hcount, vcount -- current x, y pixel

   Outputs:
      pixel -- pixel output for the ball
*/

module circle(display, vclock, reset, frameclk, x, y, hcount, vcount, pixel);
   input display;                          // should the circle be shown on screen?
   input vclock;                           // 65Mhz clock
   input reset;                            // on reset, go back to the start of the ROM
   input frameclk;                         // indicates new frame
   input [10:0] x;                         // x-coordinate of the circle
   input [9:0] y;                          // y-coordinate of the circle
   input [10:0] hcount;                    // current x-pixel
   input [9:0] vcount;                     // current y-pixel
   output [2:0] pixel;                     // color output for the circle at this
        //    (hcount, vcount) coordinate

   parameter COLOR = 3'b111;               // default ball color is white
   parameter start_address = 6'd0;

   reg [2:0] pixel;
   reg [5:0] address;                      // address for the circle's ROM
   wire data_out;                          // data coming from the ROM
   reg [63:0] ball_data;                   // register gets loaded with circle's data
   reg [5:0] counter;                      // counter for loading data into register
   reg read_cycle;                         // switch on and off to add a delay for
                                           //    reading from memory

   // ROM for the 8x8 pixel circle
   circle8 ball2(.addr(address), .clk(vclock), .dout(data_out));

   always @ (posedge vclock) begin
      if(reset) begin                      // Reset
         address <= start_address;
         ball_data <= 64'd0;
         counter <= 6'd0;
         read_cycle <= 1'b0;
         pixel <= 3'b000;
      end

      // Every frame, reset the counter and ball data, and set the 1st
      // address from which to read
      else if (frameclk) begin
         counter <= 6'd0;
         address <= 6'd0;
```

```
            read_cycle <= 1'b1;
            ball_data <= 64'd0;
        end

        // Alternate between read_cycle and the next statment -- during
        // the read cycle, set the address for the ROM and copy data_out
        // into the register
        else if (read_cycle) begin
            read_cycle <= 1'b0;
            address <= (address == 63) ? 8'd0 : address + 1'b1;
            ball_data[0] <= data_out;
        end

        // Between read cycles, give the memory time to fetch the data
        // from the new address; shift the contents of the register up 1
        // "slot" so the data from the ROM is slowly loaded into the
        // register
        else if (counter < 8'd63) begin
            counter <= counter + 1'b1;
            read_cycle <= 1'b1;
            ball_data <= ball_data << 1'b1;
        end

        // When hcount, vcount, are within the 8x8 square to the right
        // and below the x,y coordinates of the circle, look at the
        // register to see if there is a pixel in that location --
        // if so, set pixel to the right color
        if(hcount >= x && hcount < x + 4'h8 && vcount >= y
              && vcount < y + 4'h8 && display) begin
            pixel <= ball_data[0] ? COLOR : 3'd0;
            ball_data <= ball_data >> 1'b1;             // discard the pixel data
                                                        // once it's been looked at
        end
        else pixel <= 1'b0;
    end
endmodule
```

A.5 debounce.v

```
//////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module
// provided by 6.111 staff for various lab assignments
// NOT CHANGED BY EDDIE AND IRENE!
//
//////////////////////////////////////////////////////////////////////

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
```

A.6 delayN.v

```
// The delayN module delays the "in" signal by four clock cycles.
// We use this for delaying hsync, vsync, and blank in our project.
// This module was included with the ZBT sample package from the
// 6.111 Fall 2005 handouts and we didn't change it.

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
```

A.7 det_brick_din.v

```
// det_brick_din.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan

/*
   This module specifies what the next color of a block should be if a
   brick of the specified color is hit.

   Red bricks turn black (the brick "breaks" and is no longer
   displayed), yellow bricks turn red, green bricks turn yellow, teal
   bricks turn green, blue bricks turn teal, and magenta bricks stay
   magenta (they never break).

   Inputs:
      vclock -- system clock
      color -- current brick color

   Outputs:
      next_brick_color -- what color this block should turn if the
         brick is hit
*/

module det_brick_din(vclock, color, next_brick_color);
   input vclock;                          // 65 Mhz clock
   input [2:0] color;                     // current brick color
   output [3:0] next_brick_color;         // what color the brick should become

   reg [3:0] next_brick_color;

   // case statement takes the current brick color and returns next one
   always @ (posedge vclock)
      case(color)
         3'b100:   //red
            next_brick_color <= 4'b0000;  // red turns black
         3'b110:   //yellow
            next_brick_color <= 4'b1100;  // yellow turns red
         3'b010:   //green
            next_brick_color <= 4'b1110;  // green turns yellow
         3'b011:   //teal
            next_brick_color <= 4'b1010;  // teal turns green
         3'b001:   //blue
            next_brick_color <= 4'b1011;  // blue turns teal
         3'b101:   //magenta
            next_brick_color <= 4'b1101;  // magenta stays magenta
         default:
            next_brick_color <= 4'b0000;
      endcase
endmodule
```

A.8 detect_bombs.v

```
// detect_bombs.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan

/*
   This module detects when the pixels of one specified object overlap
   with pixels of a second specified object.

   The module is used for generating a signal that indicates when
   bombs, expanders, and life capsules hit the paddle.

   Inputs:
      object1_pixel -- pixels from 1st object
      object2_pixel -- pixels from 2nd object

   Outputs:
      hit -- goes high when 2 pixels overlap
*/

module detect_bombs(object1_pixel, object2_pixel, hit);
input [2:0] object1_pixel;      // 1st object's pixels
input [2:0] object2_pixel;      // 2nd object's pixels
output hit;                     // 1 if both objects pixels overlap

// OR each object's pixels to see if the color is black.
       // If neither color is black, then there's been an overlap!
assign hit = ((|object1_pixel) & (|object2_pixel));
endmodule
```

A.9 display.v

```
// display.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan

/* This module creates the bricks from the contents of a dual-port RAM
   which contains information for every block; it also outputs the
   pixels of all other objects on screen. The module also handles
   switching to and from intertitles to generate a full 24-bit color
   game output to the VGA.
   Inputs:
      vclock -- system clock
      reset -- reset signal
      frameclk -- pulses high when vsync goes low
      hcount, vcount -- current pixel
      grid_x, grid_y -- current block to look at
      brick_data_out -- information about the particular block and is
         4 bits -- 1 bit to toggle the display and 3 bits of color
      paddle_pixel -- pixel for the paddle
      ball_pixel -- pixel for the ball
      title_pixel -- pixel for intertitles
      bomb1_pixel, bomb2_pixel, bomb3_pixel, bomb4_pixel,
         bomb5_pixel -- pixel for the bombs
      expander_pixel -- pixel for the expander
      extralife_pixel -- pixel for the life capsule
      fade_on -- 1 if the ball fades (gradient) as it falls down
         the screen
      title_on -- 1 if an intertitle should be displayed on screen
   Outputs:
      brick_address -- address for accessing the brick memory
      pixel -- combined pixel output
*/

module display(vclock, reset, frameclk, hcount, vcount, grid_x,
               grid_y, brick_data_out, paddle_pixel, ball_pixel,
               title_pixel, bomb1_pixel, bomb2_pixel, bomb3_pixel,
               bomb4_pixel, bomb5_pixel, expander_pixel,
               extralife_pixel, fade_on, title_on, brick_address,
               pixel);
   input vclock;
   input reset;
   input frameclk;
   input [10:0] hcount;
   input [9:0] vcount;
   input [2:0] grid_x;
   input [4:0] grid_y;
   input [3:0] brick_data_out;
   input [2:0] paddle_pixel;
   input [2:0] ball_pixel;
   input [2:0] title_pixel;
   input [2:0] bomb1_pixel, bomb2_pixel, bomb3_pixel, bomb4_pixel,
               bomb5_pixel;
   input [2:0] expander_pixel, extralife_pixel;
   input fade_on;
   input title_on;

   output [9:0] brick_address;
   output [23:0] pixel; // 24-bit VGA output
```

```
   reg [23:0] pixel;

   wire [2:0] pixel_3; // 3-bit game pixel
   wire [23:0] pixel_24; // 24-bit version of the 3-bit game pixel
   wire [23:0] fullball_pixel; // 24-bit version of the ball pixel
   reg [7:0] counter; // fade-in counter for intertitles

   reg [2:0] last_grid_x;       // keeps track of the grid the pixel is in
   reg [4:0] last_grid_y;
   reg [2:0] brick_pixel;
   reg [9:0] brick_address;

   // Each 128 by 32 block has a 2 pixel border around it
   assign border = ((hcount[6:1] == 6'b111111) || (hcount[6:1] == 6'd0)
                    || (vcount[4:1] == 4'b1111) || (vcount[4:1] == 4'd0))?
                    1'b1: 1'b0;

   // Besides the border, there is a 8 pixel rim (used for magenta blocks
   //    which have a "hollow" center)
   assign rim = ((hcount[6:3] == 4'b1111) || (hcount[6:3] == 4'd0)
                 || (vcount[4:3] == 2'b11) || (vcount[4:3] == 2'd0))?
                 1'b1 : 1'b0;


   always @ (posedge vclock) begin
      if (reset) begin
         last_grid_x <= 3'd0;
         last_grid_y <= 5'd0;
         brick_address <= 10'd0;
         counter <= 8'd0;
      end

      // If grid_x or grid_y changes, recalculate address
      else if((grid_x != last_grid_x) || (grid_y != last_grid_y)) begin
         last_grid_x <= grid_x;
         last_grid_y <= grid_y;
         brick_address <= grid_y << 3 + grid_x;
      end
      else brick_address <= brick_address;

      // brick_data_out[3] is 1 if that block has a brick to be displayed
      // If a brick is there, draw black for the border and black for the
      //    insides of magenta bricks, otherwise fill the rest of the block
      //    in with the color specified by brick_data_out[2:0]
      brick_pixel <= brick_data_out[3]?
                     (border? 3'd0 : ((~rim && brick_data_out[2:0] == 3'b101)?
                         3'd0 : brick_data_out[2:0]))
                     : 3'd0;

      // Eddie's additions:
      // If intertitles are supposed to be displayed, fade them in.
      // The fade process takes approximately four seconds.
      if (title_on) begin
         counter <= (frameclk & (counter != 8'hff)) ? counter + 1 : counter;
         pixel <= pixel_24 & {counter,counter,counter};
      end
// otherwise, display all the in-game objects, merging in the ball.
      else begin
         pixel <= (pixel_24) ? pixel_24 : fullball_pixel;
         counter <= 8'd0;
      end
   end

   // convert the ball pixel to a 24-bit value, and if necessary,
   // fade the ball depending on its vertical location.
```

```
   assign fullball_pixel = fade_on ? {~vcount[9:2]&{8{ball_pixel[2]}},
                                      ~vcount[9:2]&{8{ball_pixel[1]}},
                                      ~vcount[9:2]&{8{ball_pixel[0]}}} :
                                      {{8{ball_pixel[2]}},
                                       {8{ball_pixel[1]}},
                                       {8{ball_pixel[0]}}};

   // if a title screen should be on, show title_pixel; otherwise, the pixel
   //    output is the combination of all pixel outputs
   assign pixel_3 = title_on ? title_pixel :
                               (paddle_pixel | brick_pixel | bomb1_pixel |
                                bomb2_pixel | bomb3_pixel | bomb4_pixel |
                                bomb5_pixel | expander_pixel | extralife_pixel);

   // extend the three-bit game pixel to a 24-bit value without changing colors
   assign pixel_24 = {{8{pixel_3[2]}}, {8{pixel_3[1]}}, {8{pixel_3[0]}}};
endmodule
```

A.10 drop_game_object.v

```
// drop_game_object.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan

/*
   This module handles dropping and managing bombs/expanders/life
   capsules in the game. There can be up to 5 bombs, 1 expander, and 1
   lifecapsule on the screen at a time. Bombs shorten the paddle when
   they hit it, expanders lengthen the paddle, and lifecapsules grant
   the player an extra life.

   The module makes dropping objects in the game very easy -- in
   superbreakout.v, I just pulse either drop_bomb, drop_expander, or
   drop_lifecapsule high and then set the grid_x and grid_y it from
   which the object is supposed to fall.

   Inputs:
      vclock -- system clock
      reset -- reset signal
      update_clock -- when this pulses high, it's time to update
         the position of all the bombs/expanders/life capsules;
         happens once a frame
      has_game_started -- is the game in play
      level_complete -- 1 if player reaches the end of the level
      force_next_level -- 1 when player skips to the next level
         without finishing the current one
      loselife -- 1 if player loses a life
      drop_bomb, drop_expander, drop_lifecapsule -- 1 to drop a
         bomb, expander, or lifecapsule, respectively
      drop_gridx, drop_gridy -- x and y coordinates of the block
         that the object should drop from
      shorten_paddle_b1, shorten_paddle_b2, shorten_paddle_b3,
         shorten_paddle_b4, shorten_paddle_b5 -- 1 when a bomb1,
         bomb2, bomb3, bomb4, bomb5 hits the paddle, respectively
      lengthen_paddle -- 1 when expander hits the paddle
      life_up -- 1 when lifecapsule hits the paddle

   Outputs:
      bomb1_display, bomb2_display, bomb3_display, bomb4_display,
         bomb5_display -- controls whether each bomb is displayed
      expander_display, extralife_display -- 1 if expander,
         lifecapsule should be displayed on screen
      bomb1_x, bomb2_x, bomb3_x, bomb4_x, bomb5_x, expander_x,
         extralife_x -- x-coordinates for the top left corner
         of these sprites
      bomb1_y, bomb2_y, bomb3_y, bomb4_y, bomb5_y, expander_y,
         extralife_y -- y-coordinates for the top left corner
         of these sprites
*/

module drop_game_object(vclock, reset, update_clock, has_game_started,
                        level_complete, force_next_level, loselife,
                        drop_bomb, drop_expander, drop_lifecapsule,
                        drop_gridx, drop_gridy, shorten_paddle_b1,
                        shorten_paddle_b2, shorten_paddle_b3,
                        shorten_paddle_b4, shorten_paddle_b5,
                        lengthen_paddle, life_up, bomb1_display,
                        bomb2_display, bomb3_display, bomb4_display,
                        bomb5_display, expander_display,
                        extralife_display, bomb1_x, bomb2_x, bomb3_x,
                        bomb4_x, bomb5_x, expander_x, extralife_x,
```

```
                    bomb1_y, bomb2_y, bomb3_y, bomb4_y, bomb5_y,
                    expander_y, extralife_y);
input vclock, reset, update_clock, has_game_started,
      level_complete, force_next_level, loselife;
input drop_bomb, drop_expander, drop_lifecapsule;
input [2:0] drop_gridx;
input [4:0] drop_gridy;
input shorten_paddle_b1, shorten_paddle_b2, shorten_paddle_b3,
      shorten_paddle_b4, shorten_paddle_b5, lengthen_paddle,
      life_up;
output bomb1_display, bomb2_display, bomb3_display, bomb4_display,
       bomb5_display, expander_display, extralife_display;
output [10:0] bomb1_x, bomb2_x, bomb3_x, bomb4_x, bomb5_x,
              expander_x, extralife_x;
output [9:0] bomb1_y, bomb2_y, bomb3_y, bomb4_y, bomb5_y, expander_y,
             extralife_y;

// Parameters
parameter screen_height = 10'd768;       // screen is 1024 x 768 pixels
parameter brick_width = 8'd128;          // brick is 128 pixels wide
parameter bomb_speed = 4'd12;            // bombs fall 12 pixels/frame
parameter bomb_dimension = 4'd8;         // bombs are 8x8 pixels
parameter expander_speed = 4'd12;        // expanders fall 12 pixels/frame
parameter expander_dimension = 4'd8;     // expanders are 8x8 pixels
parameter extralife_speed = 4'd12;       // lifecapsules fall 12 pixels/frame
parameter extralife_dimension = 4'd8;    // lifecapsules are 8x8 pixels
reg bomb1_display, bomb2_display, bomb3_display, bomb4_display,
    bomb5_display, expander_display, extralife_display;
reg [10:0] bomb1_x, bomb2_x, bomb3_x, bomb4_x, bomb5_x, expander_x,
           extralife_x;
reg [9:0] bomb1_y, bomb2_y, bomb3_y, bomb4_y, bomb5_y, expander_y,
          extralife_y;

always @ (posedge vclock) begin
   // On a reset or level change or when the player loses a life,
   // clear the screen of bombs/expanders/life capsules
   if(reset | (has_game_started & level_complete) |
      force_next_level | loselife) begin
      bomb1_display <= 0;
      bomb2_display <= 0;
      bomb3_display <= 0;
      bomb4_display <= 0;
      bomb5_display <= 0;
      expander_display <= 0;
      extralife_display <= 0;
   end
   // Display an object when it has been signaled to be dropped
   // from a brick
   else begin
      // If a bomb is dropped, display the next bomb that is not
      // yet on the screen (at most 5 can be falling down the
      // screen at a time) from the top center of the brick
      // specified by drop_gridx, drop_gridy
      if(drop_bomb) begin
         if(~bomb1_display) begin
            bomb1_display <= 1'b1;
            bomb1_x <= (drop_gridx << 7) +
```

```
                ((brick_width - bomb_dimension) >> 1);
      bomb1_y <= drop_gridy << 5;
   end
   else if(~bomb2_display) begin
      bomb2_display <= 1'b1;
      bomb2_x <= (drop_gridx << 7) +
                ((brick_width - bomb_dimension) >> 1);
      bomb2_y <= drop_gridy << 5;
   end
   else if(~bomb3_display) begin
      bomb3_display <= 1'b1;
      bomb3_x <= (drop_gridx << 7) +
                ((brick_width - bomb_dimension) >> 1);
      bomb3_y <= drop_gridy << 5;
   end
   else if(~bomb4_display) begin
      bomb4_display <= 1'b1;
      bomb4_x <= (drop_gridx << 7) +
                ((brick_width - bomb_dimension) >> 1);
      bomb4_y <= drop_gridy << 5;
   end
   else if(~bomb5_display) begin
      bomb5_display <= 1'b1;
      bomb5_x <= (drop_gridx << 7) +
                ((brick_width - bomb_dimension) >> 1);
      bomb5_y <= drop_gridy << 5;
   end
end

// If an expander is supposed to be dropped and there isn't
// already one on the screen, drop it! This happens when a green
// brick is hit
else if(drop_expander && ~expander_display) begin
   expander_display <= 1'b1;
   expander_x <= (drop_gridx << 7) +
                ((brick_width - expander_dimension) >> 1);
   expander_y <= drop_gridy << 5;
end

// If a life capsule is supposed to be dropped, drop one if
// there isn't already one on screen -- happens when a blue
// brick is hit
else if(drop_lifecapsule && ~extralife_display) begin
   extralife_display <= 1'b1;
   extralife_x <= (drop_gridx << 7) +
                 ((brick_width - extralife_dimension) >> 1);
   extralife_y <= drop_gridy << 5;
end

// Move all the bombs/expanders/lifecapsules that are
// currently being displayed by updating their y-coordinate
// every frame (when update_clock is specified in
// superbreakout.v). Objects disappear when they either hit the
// paddle or get near the bottom of the screen
if(update_clock) begin
   if(bomb1_display) begin
      bomb1_y <= (bomb1_y + bomb_speed > screen_height)?
                 screen_height - bomb_dimension : bomb1_y + bomb_speed;
      bomb1_display <= ~((bomb1_y >= screen_height - bomb_dimension)
                      || shorten_paddle_b1);
   end
   if(bomb2_display) begin
```

```
            bomb2_y <= (bomb2_y + bomb_speed > screen_height)?
                       screen_height - bomb_dimension : bomb2_y + bomb_speed;
            bomb2_display <= ~((bomb2_y >= screen_height - bomb_dimension)
                            || shorten_paddle_b2);
         end
         if(bomb3_display) begin
            bomb3_y <= (bomb3_y + bomb_speed > screen_height)?
                       screen_height - bomb_dimension : bomb3_y + bomb_speed;
            bomb3_display <= ~((bomb3_y >= screen_height - bomb_dimension)
                            || shorten_paddle_b3);
            end
         if(bomb4_display) begin
            bomb4_y <= (bomb4_y + bomb_speed > screen_height)?
                       screen_height - bomb_dimension : bomb4_y + bomb_speed;
            bomb4_display <= ~((bomb4_y >= screen_height - bomb_dimension)
                            || shorten_paddle_b4);
         end
         if(bomb5_display) begin
            bomb5_y <= (bomb5_y + bomb_speed > screen_height)?
                       screen_height - bomb_dimension : bomb5_y + bomb_speed;
            bomb5_display <= ~((bomb5_y >= screen_height - bomb_dimension)
                            || shorten_paddle_b5);
         end
         if(expander_display) begin
            expander_y <= (expander_y + expander_speed > screen_height)?
                          screen_height - expander_dimension :
                          expander_y + expander_speed;
            expander_display <=
                          ~((expander_y >= screen_height - expander_dimension)
                               || lengthen_paddle);
         end
         if(extralife_display) begin
            extralife_y <= (extralife_y + extralife_speed > screen_height)?
                           screen_height - extralife_dimension :
                           extralife_y + extralife_speed;
            extralife_display <=
                          ~((extralife_y >= screen_height - extralife_dimension)
                               || life_up);
         end
      end
   end
endmodule
```

A.11 expander_blob.v

```
// expander_blob.v
// 6.111 Fall 2006 Final Project
// Module written by Eddie Fagin

/*
   This module displays the icon for the expander, which lengthens the
   paddle if caught. expander_blob.v is very similar to circle.v, but
   uses a different ROM. At the end of every frame (when vsync goes low),
   ROM data is loaded into a register and then read from the register
   during the next frame as to avoid glitches.

   Inputs:
      display -- whether or not the expander should be displayed on screen
         (becomes 1 when the expander is dropped from a brick)
      vclock -- system clock
      reset -- reset signal
      frameclk -- pulses high when vsync goes low at the end of a frame
      x, y -- x-y coordinates of top left corner or the 16x8 rom
      hcount, vcount -- current pixel

   Outputs:
      pixel -- pixel color output for the current hcount, vcount
*/

module expander_blob(display, vclock, reset, frameclk, x, y, hcount, vcount, pixel);
   input display;         // 1 if expander should be displayed
   input vclock;          // 65Mhz clock
   input reset;           // reset signal -- go back to beginning of ROM
   input frameclk;        // 1 when vsync goes low at the end of a frame
   input [10:0] x;        // x, y coordinates of expander
   input [9:0] y;
   input [10:0] hcount;   // current pixel coordinates
   input [9:0] vcount;
   output [2:0] pixel;    // pixel output for expander

   parameter COLOR = 3'b010;          // default expander color is green
   parameter start_address = 6'd0;

   reg [2:0] pixel;
   reg [6:0] address;                 // address for expander ROM
   wire data_out;                     // data from ROM
   reg [127:0] ball_data;             // load ROM data into this register
   reg [6:0] counter;                 // counter for ROM
   reg read_cycle;                    // toggles on and off so to allot extra time
                                      //    for reading from ROM

   // 16x8 ROM for expander
   expander exp(.addr(address), .clk(vclock), .dout(data_out));

   always @ (posedge vclock) begin
      if(reset) begin                 // Reset
         address <= start_address;
         ball_data <= 128'd0;
         counter <= 7'd0;
         read_cycle <= 1'b0;
         pixel <= 3'b000;
      end

      // At the end of every frame, start loading ROM contents into
      // the register
      else if (frameclk) begin
         counter <= 7'd0;
```

```
            address <= 7'd0;
            read_cycle <= 1'b1;
            ball_data <= 128'd0;
        end

        // Alternate between this case and the next one while loading data
        // from the ROM to the register -- here, set the address and load
        // the data_out into the register
        else if (read_cycle) begin
            read_cycle <= 1'b0;
            address <= (address == 127) ? 7'd0 : address + 1'b1;
            ball_data[0] <= data_out;
        end

        // In between read cycles while data, give the memory time to fetch
        // the data and update the counter
        else if (counter < 63) begin
            counter <= counter + 1'b1;
            read_cycle <= 1'b1;
            ball_data <= ball_data << 1'b1;
        end

        // If the expander is supposed to be displayed, look at register
        // for pixel data when hcount and vcount are within the 16x8 "box"
        // to the left of x,y; otherwise display black
        if (hcount >= x && hcount <= x + 4'hf &&
            vcount >= y  && vcount <= y + 3'b111 & display) begin
            pixel <= ball_data[0] ? COLOR : 3'd0;
            ball_data <= ball_data >> 1'b1;
        end
        else pixel <= 1'b0;
    end
endmodule
```

A.12 finalproject.v

```
//////////////////////////////////////////////////////////////////////
//
// 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 finalproject(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
   //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;

   // 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, clock_65mhz, ~button_enter, user_reset);
   assign reset = user_reset | power_on_reset;

/////////////////////////////////////////
The following group of modules was modified
from an example in the 6.111 handouts from
Fall 2005. The example generates a black-and-
white camera image by storing the incoming
data stream in the ZBT memory and then
retrieving it for VGA display purposes. Our
modifications essentially change this code to
store a color image (24-bit/pixel) instead of
the 8-bit B&W image. This modification creates
a blockier image, but our project requires an
out-of-focus camera for optimum performance
so the grainy image is of little importance.

The modules are:
xvga
zbt_6111
vram_display *
adv7185init
ntsc_decode  *
ntsc_to_zbt  *
delayN
YCrCb2RGB

*: Modified from original source. All
modifications are noted in the module
code.
/////////////////////////////////////////

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

    // wire 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(clock_65mhz, 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 [23:0]  vr_pixel;
  wire [18:0]  vram_addr1;
  vram_display vd1(reset,clock_65mhz,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 [23:0] 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[19:12]),
       .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] write_addr;
    wire [35:0] write_data;
   wire        ntsc_we;
   ntsc_to_zbt n2z (clock_65mhz, tv_in_line_clock1, fvh, dv, ycrcb[23:0],
    write_addr, write_data, ntsc_we);

// set up ZBT-writing connections
    wire  my_we = (hcount[1:0]==2'd2);
    assign  vram_addr = my_we ? write_addr : vram_addr1;
   assign  vram_we = my_we;
   assign  vram_write_data = write_data;

    // latch output pixel data
    reg [23:0]  pixel_ycrcb;
always @(posedge clock_65mhz) pixel_ycrcb <= vr_pixel;

// YCrCb2RGB module provided by Xilinx
wire [23:0] pixel_rgb;
wire [23:0] pixel_rgb_filtered;
YCrCb2RGB colors (.R(pixel_rgb[23:16]), .G(pixel_rgb[15:8]),
 .B(pixel_rgb[7:0]), .clk(clock_65mhz), .rst(reset),
 .Y({pixel_ycrcb[23:16], 2'b00}), .Cr({pixel_ycrcb[15:8], 2'b00}),
 .Cb({pixel_ycrcb[7:0], 2'b00}));


/***********************************
 Now, for our original modules!
***********************************/

// Debounced switch and button wires;
//  see bottom of this file for implementation modules
wire switch0, switch1, switch2, switch3, switch4,
 switch5, switch6, switch7;
wire buttonleft, buttonright, buttondown;

// Only strong red, green, or blue colors will pass through the filter.
// This is used to remove background noise and intensify bright spots
//  (like LEDs).
rgbfilter colorfilter(.RGBin(pixel_rgb), .clk(clock_65mhz),
 .filter(8'd252), .greenfilter(1'b1), .hcount(hcount),
 .vcount(vcount), .RGBout(pixel_rgb_filtered));

// Generates the frame clock (on negative edge of vsync).
wire frameclk;
negedgedetector ed(vsync, clock_65mhz, reset, frameclk);

// Returns the approximate center of all green pixels.
// This module is influenced by noise such as overhead
//  fluorescent lighting, but works fine in most cases.
wire [10:0] comX;
wire [9:0] comY;
   centerofmass COM(.pixel(pixel_rgb_filtered[15:8]), .x(hcount),
 .y(vcount), .clk(clock_65mhz), .frameclk(frameclk),
 .reset(reset), .comX(comX), .comY(comY));

// Takes in the pixel center determined by the above
// module and calculates an approximate velocity
// through a two-bit output representing the four
// cardinal directions and four possible speeds.
```

```
// velocity = {speed, direction}
//  speed:     (00) zero  (01) slow  (10) medium (11) fast
   // direction: (00) up    (01) right (10) down   (11) left
wire [3:0] velocity;
COMvelocity COMv(.x(comX), .y(comY), .clk(clock_65mhz),
 .frameclk(frameclk), .reset(reset), .velocity(velocity));

// Debugging module for testing controller sensitivity.
// The user manipulates a white square on the screen,
//  within the 2-D plane determined by the screen size.
wire[2:0] testpixel;
wire draw_reset;
block_test block (.vclock(clock_65mhz), .frameclk(frameclk),
 .reset(draw_reset), .velocity(velocity), .hcount(hcount),
 .vcount(vcount), .color({switch6, switch5, switch4}),
    .mode(switch7),.comX(comX), .comY(comY), .pixel(testpixel));

// When the user switches operation modes, trigger a
// system-wide reset and hold the reset signal high for
// one frame using our "highforframe" module.
reg mode_change;
wire mode_reset;
reg [1:0] oldswitches;
always @ (posedge clock_65mhz) begin
if (reset)
mode_change <= 1'b0;
else if (oldswitches != {switch1, switch0})
mode_change <= 1'b1;
else mode_change <= 1'b0;
oldswitches <= {switch1, switch0};
end
highforframe modehold(reset, clock_65mhz, frameclk,
 mode_change, mode_reset);

// **************
// Super Breakout
// **************

wire[23:0] gamepixel;    // output pixel for VGA
wire game_start;       // button 1
wire start = game_start     // game starts if we're in game mode
 && {switch1, switch0} == 2'b00;
wire gamereset = reset|mode_reset; // reset whenever operation mode changes
wire collision;  // high for frame when ball hits a brick
wire [1:0] level         // current game level (-1), 0-3

// Four possible game states:
// (00) - game hasn't started yet
      // (01) - game is running
      // (10) - game over (lose)
      // (11) - game complete (win)
wire [1:0] game_over;

// Hexadecimal game score, converted to base 10 in LEDdisplay
wire [31:0] game_score;

// Game difficulty, defined as 4 * switch[3:2]
wire [3:0] game_speed = {2'b00, switch3, switch2} << 2'b10;

// Number of extra balls, pre-formatted for LEDs.
// In other words, there's only one '1' in this value,
```

```
//  that lights up the LED of the # of extra balls left.
// So 6'b000100 --> 2 extra balls ('1' at position 2).
// To increase/decrease the # of lives, we just shift the value.
wire [5:0] lives;

// Indicator variable that goes high when an intertitle
//  is displaying in the game.
wire title_on;

// Demo cheat: allow user to skip levels by pressing Button Down
wire next_level;
negedgedetector forcer(buttondown, clock_65mhz, reset, next_level);

// Allow user to turn ball fading on and off by pressing Button 2.
// As soon as the user presses the button, the ball fade is activated,
      //  and the user can hold down the button as long as he/she wishes
//  without effect.
reg fade = 1'b0;
wire fade_toggle;
negedgedetector fader(game_toggle, clock_65mhz, reset, fade_toggle);
always @ (posedge clock_65mhz)
fade <= (fade_toggle) ? ~fade : fade;

// Hold a collision pulse high for one frame so that the sound-effects
//  module has time to notice it (sfx is running 27-MHz clock, collision
//  is on 65-MHz).
wire collision_frame;
highforframe collisionhold(reset, clock_65mhz, frameclk,
 collision, collision_frame);

// Latch the displayed score from the game, and allow it to reset.
reg [31:0] score;
always @ (posedge clock_65mhz) score <= (gamereset) ? 32'd0 : game_score;

// The main game module
superbreakout game(.vclock(clk), .reset(gamereset), .frameclk(frameclk),
 .start(start), .pspeed(game_speed), .paddle_velocity(velocity),
 .force_next_level(next_level), .inleft(~buttonleft), .inright(~buttonright),
 .hcount(hcount), .vcount(vcount), .fade_on(fade), .collision(collision),
 .pixel(gamepixel), .game_over(game_over), .score(game_score), .level(level),
 .lives(lives), .title_on(title_on));

// Delay most signals by 3 cycles to sync with ZBT read
//  and pipelined game output
wire  b,hs,vs;
    delayN dn1(clock_65mhz,hsync,hs);
    delayN dn2(clock_65mhz,vsync,vs);
    delayN dn3(clock_65mhz,blank,b);

// We need to delay the intertitle VGA signals by five clock cycles
//  because pixel (0,0) isn't ready until (4,0).
wire b_title, hs_title, vs_title;
delayN dn1b(clock_65mhz,hsync,hs_title);
 defparam dn1b.NDELAY = 3'd5;
delayN dn2b(clock_65mhz,vsync,vs_title);
 defparam dn2b.NDELAY = 3'd5;
delayN dn3b(clock_65mhz,blank,b_title);
 defparam dn3b.NDELAY = 3'd5;

/* VGA Output.  In order to meet the setup and hold times of the
      AD7125, we send it ~clock_65mhz.
Four output modes, determined by debounced switch[1:0]:
  00: Game Mode (Super Breakout)
```

```
01: Draw Mode (Drawing applet)
10: Camera Mode (What the camera sees)
11: Filter Mode (Filtered camera view)

The following RGB assignments are just a large mux that
 passes the correct outputs to the VGA depending on
 the operation mode set by switch[1:0].   */
    assign vga_out_red   =  switch1 ?
(switch0 ? pixel_rgb_filtered[23:16] :
{8{testpixel[2]}} | pixel_rgb[23:16]) :
(switch0 ? {8{testpixel[2]}} : gamepixel[23:16]);
assign vga_out_green   = switch1 ?
(switch0 ? pixel_rgb_filtered[15:8] :
{8{testpixel[1]}} | pixel_rgb[15:8]) :
(switch0 ? {8{testpixel[1]}} : gamepixel[15:8]);
    assign vga_out_blue   = switch1 ?
(switch0 ? pixel_rgb_filtered[7:0] :
{8{testpixel[0]}} | pixel_rgb[7:0]) :
(switch0 ? {8{testpixel[0]}} : gamepixel[7:0]);

assign vga_out_sync_b = 1'b1;    // not used
    assign vga_out_pixel_clock = ~clock_65mhz;

// This makes sure the correct delays are implemented at the
      //  correct time. The conditional statement is literally
//  translated as "If the intertitle is showing on screen..."
assign vga_out_blank_b =
(title_on & {switch1,switch0} == 2'b00) ? ~b_title : ~b;
    assign vga_out_hsync =
(title_on & {switch1,switch0} == 2'b00) ? hs_title : hs;
    assign vga_out_vsync =
(title_on & {switch1,switch0} == 2'b00) ? vs_title : vs;

// LED menu display
// If we're in "active game mode," - game is visible and running -
//  show score by default, otherwise show character display.
//  The user can force the alternate display by holding Button 0.
wire toggle_disp;
reg menu_display;
always @ (posedge clock_27mhz)
menu_display <= ({switch1, switch0} != 2'b00) ?
 ~toggle_disp : (game_over[1] ^ toggle_disp);
LEDdisplay myLEDs(gamereset, clock_27mhz, {switch1, switch0},
 menu_display, game_over, score, disp_blank, disp_clock,
 disp_rs, disp_ce_b, disp_reset_b, disp_data_out);

// User-defined sound effects setup
// The user can press Button Up to record a short
//  sound clip, which is played back in full whenever
//  the ball collides with a brick.
// This process uses most of Eddie's code from Lab 4,
//  but it's modified to play everything when
//  "playback" is pulsed high.
wire [7:0] from_ac97_data, to_ac97_data;
    wire recording, ready;
reg playback;
always @ (posedge clock_27mhz) begin
if (reset) playback <= 1'b0;
else if (collision_frame) playback <= 1'b1;
else playback <= 1'b0;
```

```
end
recorder r(clock_27mhz, reset, playback, recording,
 ready, from_ac97_data, to_ac97_data);

    // AC97 driver
    audio a(clock_27mhz, power_on_reset, 5'd16, from_ac97_data,
 to_ac97_data, ready, audio_reset_b, ac97_sdata_out,
 ac97_sdata_in, ac97_synch, ac97_bit_clock);

// Set up LED light output
assign led = ~{level, lives};

// Debounce all buttons and switches
debounce bup(reset, clock_27mhz, ~button_up, recording);
debounce bdown(reset, clock_65mhz, button_down, buttondown);
debounce bleft(reset, clock_65mhz, button_left, buttonleft);
debounce bright(reset, clock_65mhz, button_right, buttonright);
debounce b0(reset, clock_65mhz, ~button0, toggle_disp);
debounce b1(reset, clock_65mhz, ~button1, game_start);
debounce b2(reset, clock_65mhz, button2, game_toggle);
debounce b3(reset, clock_65mhz, ~button3, draw_reset);
debounce s0(reset, clock_65mhz, switch[0], switch0);
debounce s1(reset, clock_65mhz, switch[1], switch1);
debounce s2(reset, clock_65mhz, switch[2], switch2);
debounce s3(reset, clock_65mhz, switch[3], switch3);
debounce s4(reset, clock_65mhz, switch[4], switch4);
debounce s5(reset, clock_65mhz, switch[5], switch5);
debounce s6(reset, clock_65mhz, switch[6], switch6);
debounce s7(reset, clock_65mhz, switch[7], switch7);
endmodule
```

A.13 game_over.v

```
// game_over.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan

/*
   This module raises the "done" signal when there are no more
   breakable bricks left on the screen. It also outputs the number of
   breakable bricks left on the screen (so in superbreakout.v, when
   there are 3 or fewer bricks left on the screen, some random bombs
   start to fall, and then when there is only 1 brick left we have an
   "acid rain" of bombs).

   Inputs:
      vclock -- system clock
      reset -- reset signal
      init_num_bricks -- initial number of breakable bricks on screen
      collision -- 1 when ball hits a brick
      color -- color of the current brick

   Outputs:
      done -- 1 when no breakable bricks are left on the screen
      num_bricks_left -- number of breakable bricks left on screen
*/

module game_over(vclock, reset, init_num_bricks, collision, color,
                 done, num_bricks_left);
   input vclock;
   input reset;
   input [7:0] init_num_bricks;
   input collision;
   input [2:0] color;

   output done;
   output [7:0] num_bricks_left;

   reg [7:0] num_bricks_left;
   reg done;

   always @ (posedge vclock) begin
      // On a reset, start counting down from the initial number of
      // bricks
      if(reset) begin
         num_bricks_left <= init_num_bricks;
         done <= 1'b0;
      end

      // Raise "done" if there are no breakable bricks left
      else if(num_bricks_left == 8'd0)
         done <= 1'b1;

      // Done should only pulse for one clock cycle.
      // We also set num_bricks_left to an arbitrary non-zero value
      //  so that the game does not accidentally restart.
      else if (done) begin
         done <= 1'b0;
         num_bricks_left <= 1;
      end

      // Every a red brick is hit, decrement the number of bricks left
      // (red bricks are the only ones that break)
      else if(collision)
         num_bricks_left <= (color == 3'b100)? num_bricks_left - 1 : num_bricks_left;
      else
         num_bricks_left <= num_bricks_left;
   end
```

```
endmodule
```

A.14 gen_model.v

```
/*************************************************************************
 **
 ** Module: ycrcb2rgb
 ** NOTE: EDDIE AND IRENE USED THIS, BUT DID NOT CHANGE IT FROM
 ** ITS ORIGINAL FORM!
 **
 ** Generic Equations:
 *************************************************************************/

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

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

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

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

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

always @ (posedge clk or posedge rst)
   if (rst)
      begin
      Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
      end
   else
      begin
  Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
      end

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

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

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

A.15 hex2dec.v

```
// hex2dec.v: contains two modules - hex2dec and tencounter
// Created by Eddie Fagin
// 6.111 Final Project, Fall 2006
//
// This is my favorite unnecessary feature. These modules convert a 32-bit
// hexadecimal number to its base-10 representation, also stored as a 32-bit
// hex number. We use this to display the game score as an integer.
// The implementation is a little tricky, but uses basic concepts from
// elementary school.

// hex2dec module

/* hex2dec takes a 32-bit hex number and "converts"
   it to a 32-bit base-10 number. For example, if the
   hex input was 8'h0F, the output would be 8'h15.
   (Of course, a 64-bit number would take a very long
   time to convert if the high-order bits were non-zero.
   With a 27-MHz clock, 64 bits maxed out would take
   approximately 22,000 years, which seems slightly
   impractical for game purposes. So we're using only
   the lower 32 bits of the LED display.) This module
   links up eight four-bit conversion submodules,
   much like a 32-bit adder connects multiple single-bit
   sub-adders.  Like a 32-bit adder, I'm sure there's a
   better way to implement this functionality, but at the
   moment this isn't a major component of our project so
   I'm not going to lose sleep over it. :-)
   Inputs
      clk: 27-MHz system clock, which the display modules need.
      reset: system reset
      hex: up to 32 bits in hexadecimal form, score from game.
   Ouputs
      dec: up to 32 bits, score from game converted to base 10.
*/

module hex2dec(clk, reset, hex, dec);
   input clk, reset;
   input [31:0] hex;

   output [31:0] dec;
   reg [31:0] counter, oldHex;

   wire cout0, cout1, cout2, cout3, cout4, cout5, cout6, cout7;
   wire [3:0] out0, out1, out2, out3, out4, out5, out6, out7;

   // Define a custom reset that takes into account the sequential
   // nature of the counter. (i.e. It doesn't reset if the new
   // value is greater than the old one, because the counter
   // automatically tries to get up to this new value from the old one. This
   // drastically reduces the run-time.)
   reg myReset;
   always @ (posedge clk) myReset <= ((oldHex > hex) || reset);

   // Think of this as clicker on a mechanical counter.
   // When high, it forces the value stored by the counters
   // to increase by one (each clock cycle).
   reg enable;

   // The chain of counters. Each counter represents a power of ten
   // as a four-bit number, 0-9 (so ten6 represents the millions digit).
   // The cout bits represent a digit's overflow (9 -> 10), while the
```

```
   // overflowing digit resets to zero. You probably did this in 2nd grade.
   tencounter ten0 (clk, myReset, enable, cout0,  out0);
   tencounter ten1 (clk, myReset, cout0,  cout1,  out1);
   tencounter ten2 (clk, myReset, cout1,  cout2,  out2);
   tencounter ten3 (clk, myReset, cout2,  cout3,  out3);
   tencounter ten4 (clk, myReset, cout3,  cout4,  out4);
   tencounter ten5 (clk, myReset, cout4,  cout5,  out5);
   tencounter ten6 (clk, myReset, cout5,  cout6,  out6);
   tencounter ten7 (clk, myReset, cout6,  cout7,  out7);

   // Paste each of the 4-bit digits together into a 32-bit number
   assign dec = {out7, out6, out5, out4, out3, out2, out1, out0};

   always @ (posedge clk) begin
      if (myReset) begin
         enable <= 1'b0;    // We're not clicking yet!
         counter <= 32'd1;  // This way, counter initially loops right to zero.
         oldHex <= hex;     // Remember the last input for efficiency purposes.
      end
      else begin
         // If the counter hasn't caught up to the input, keep clicking.
         enable <= (counter < hex);
         if (enable) begin
            counter <= counter + 1'b1;
         end
      end
   end
endmodule

// tencounter module

/* tencounter is my basic conversion tool for changing a hex
   number into base 10. It works the same way you'd count a base-10
   number -- every time you reach a multiple of ten, add one to the
   next-highest digit. The next-highest digit should do the same.
   Therefore, to get an N-digit base-10 number, you'd need N of
   these modules, linked up linearly, and you should feed input
   into the lowest digit's cin.

   Inputs
      clk: 27-MHz system clock, which the display modules need.
      reset: system reset
      cin: the "enable" bit. Counter increments when this goes high.

   Ouputs
      cout: the carry-out bit, goes high when the counter reaches 10.
      out: the four-bit "digit" (0-9) for this particular power of 10.
*/
module tencounter(clk, reset, cin, cout, out);
  input clk, cin, reset;
  output cout;
  output [3:0] out;

  reg [3:0] out;
  reg cout;

  always @ (posedge clk) begin
     if (reset) begin
      out <= 4'h0;
      cout <= 1'b0;
   end

   // Every enable (cin), increment the internal counter (out)
   //  until it overruns (9 -> 10), then reset everything and
   //  briefly set cout high to add one to the power of ten
```

```
   //  immediately following the current digit.
   else if (cin) begin
      if (out == 4'h9) begin
         cout <= 1'b1;
         out <= 4'h0;
      end
      else begin
         cout <= 1'b0;
         out <= out + 1'b1;
      end
   end
   else cout <= 1'b0;
  end
endmodule
```

A.16 highforframe.v

```
// highforframe.v
// 6.111 Fall 2006 Final Project
// Module created by Eddie Fagin

/* This module holds an input high for one full frame-clock cycle,
   from rising edge to rising edge. This is mostly used to simulate
   the user holding down a button for an extended period of time.

   Inputs
      reset: system/user reset
      clk: system clock, 65-MHz for our project
      frameclk: goes high for one clock cycle after the negative
                edge of each vsync
      in: the signal pulse we'd like to hold

   Outputs
      out: in, held for one frame
*/

module highforframe(reset, clk, frameclk, in, out);
   input reset, clk, frameclk, in;
   output out;

   reg hold_in, out;

   always @ (posedge clk) begin
      if (reset) begin
         out <= 1'b0;
         hold_in <= 1'b0;
      end

      // Latch the incoming signal
      else if (in && ~hold_in && ~out) hold_in <= 1'b1;

      // At the next frameclk after latching, set the output high
      else if (frameclk && hold_in) begin
         hold_in <= 1'b0;
         out <= 1'b1;
      end

      // After one frame has elapsed, release the signal hold
      else if (frameclk && out)
         out <= 1'b0;
   end
endmodule
```

A.17 intertitle.v

```
// intertitle.v
// 6.111 Fall 2006 Final Project
// Module created by Eddie Fagin

/* This module controls the various images shown
   before, during, and after SuperBreakout.
   There are six images, mostly created from
   NES game screenshots (thanks, Nintendo):
      1. Instructions: before the game starts
      2. Mario Bros.: after completion of Level 1
      3. Sonic: after completion of Level 2
      4. Mega Man: after completion of Level 3
      5. Prof. Terman: he scolds you if you lose the game
      6. Final: when (more relevantly, if) you beat the game
   These images are each represented as 256 x 192 pixel
   screenshots in BRAM, and blown up so that each BRAM pixel
   expands to a 4 x 4 pixel block on screen. 3-bit color.
   The intertitles take five clock cycles to set up, so we
   delayed these hsync, vsync and blank values by 5 to compensate.

   Inputs
      reset: game reset signal
      vclock: 65-MHz clock
      hcount, vcount: current xvga display location
      game_level: current game level (for our project, ranges from 0-3)
      game_over: two-bit game status representation
         (00) - game hasn't started yet
         (01) - game's running
         (10) - game over (you lose)
         (11) - game complete (you win)

   Ouputs
      title_pixel: the 3-bit pixel that should be output to VGA
                   if we're in title-display mode
*/

module intertitle (reset, vclock, hcount, vcount,
                   game_level, game_over, title_pixel);
   input reset, vclock;
   input [10:0] hcount;
   input [9:0] vcount;
   input [1:0] game_over, game_level;

   output [2:0] title_pixel;
   reg [2:0] title_pixel;

   wire has_game_started = (game_over != 2'b00);

   // The BRAM connectors
   reg win_enable, win;
   wire [3:0] win_dout;
   reg lose_enable, lose;
   wire [3:0] lose_dout;
   reg level1_enable, level;
   wire [3:0] level1_dout;
   reg level2_enable;
   wire [3:0] level2_dout;
   reg level3_enable;
   wire [3:0] level3_dout;
   reg intro_enable, intro;
   wire [3:0] intro_dout;

   // The various 256 x 192 read-only image BRAMs,
```

```
//  with enable bits
youwin winram({vcount[9:2],hcount[9:2]},vclock,win_dout,win_enable);
youlose loseram({vcount[9:2],hcount[9:2]},vclock,lose_dout,lose_enable);
level1complete level1ram({vcount[9:2],hcount[9:2]},
                         vclock,level1_dout,level1_enable);
level2complete level2ram({vcount[9:2],hcount[9:2]},
                         vclock,level2_dout,level2_enable);
level3complete level3ram({vcount[9:2],hcount[9:2]},
                         vclock,level3_dout,level3_enable);
intro introram({vcount[9:2],hcount[9:2]},
                         vclock,intro_dout,intro_enable);

// The way this works is an indicator will go high when the
//  respective title should be displayed (provided title display
//  is globally active). Then, one clock cycle later, we start
//  reading the BRAM (to give it time to set up). The appropriate BRAM
//  gets enabled when hcount[1:0] == 1, and then read when
//  hcount[1:0] == 3. This gives the BRAM enough time to fetch the data.
always @ (posedge vclock) begin
   if (reset) begin               // reset
      title_pixel <= 3'b000;
      intro_enable <= 1'b0;
      win_enable <= 1'b0;
      lose_enable <= 1'b0;
      level1_enable <= 1'b0;
      level2_enable <= 1'b0;
      level3_enable <= 1'b0;
      intro <= 1'b1;
      win <= 1'b0;
      lose <= 1'b0;
      level <= 1'b0;
   end

   // If we're at the end of the game
   else if (win) begin
      case(hcount[1:0])
         2'b01: win_enable <= 1'b1;
         2'b11: title_pixel <= {win_dout[0],win_dout[1],win_dout[2]};
         default: win_enable <= 1'b0;
      endcase
   end

   // Otherwise, if you've lost...
   else if (lose) begin
      case(hcount[1:0])
         2'b01: lose_enable <= 1'b1;
         2'b11: title_pixel <= {lose_dout[0],lose_dout[1],lose_dout[2]};
         default: lose_enable <= 1'b0;
      endcase
   end

   // You haven't won or lost yet? Let's make sure.
   else if (game_over[1]) begin  // game is over
      if (game_over[0]) begin    // win
         win <= 1'b1;
      end
      else begin                   // lose
         lose <= 1'b1;
      end
   end

   // OK, so you're still playing the game. What level are you on?
   else if (level) begin
      case (game_level)
```

```
            2'b01:
               case(hcount[1:0])
                  2'b01: level1_enable <= 1'b1;
                  2'b11: title_pixel <=
         {level1_dout[0],level1_dout[1],level1_dout[2]};
                  default: level1_enable <= 1'b0;
               endcase
            2'b10:
               case(hcount[1:0])
                  2'b01: level2_enable <= 1'b1;
                  2'b11: title_pixel <=
                           {level2_dout[0],level2_dout[1],level2_dout[2]};
                  default: level2_enable <= 1'b0;
               endcase
            2'b11:
               case(hcount[1:0])
                  2'b01: level3_enable <= 1'b1;
                  2'b11: title_pixel <=
                           {level3_dout[0],level3_dout[1],level3_dout[2]};
                  default: level3_enable <= 1'b0;
               endcase
            default:
               case(hcount[1:0])
                  2'b01: win_enable <= 1'b1;
                  2'b11: title_pixel <=
                           {win_dout[0],win_dout[1],win_dout[2]};
                  default: win_enable <= 1'b0;
               endcase
         endcase
      end

      // Have you even started playing yet?? Yeesh!
      else if (has_game_started) begin // level complete
         level <= 1'b1;
      end

      // I guess not. Let's display the default introduction screen.
      else if (intro) begin
         case(hcount[1:0])
            2'b01: intro_enable <= 1'b1;
            2'b11: title_pixel <=
                     {intro_dout[0],intro_dout[1],intro_dout[2]};
            default: intro_enable <= 1'b0;
         endcase
      end
   end
endmodule
```

A.18 modblob.v

```
// modblob.v
// 6.111 Fall 2006 Final Project
// Module written by Irene Fan

/*
   This module is modified off the blob.v sprite generator given to us
   for Lab 5; modblob makes it easy to change the rectangle's width,
   height and color. This is used for the paddle which grows and
   shrinks when hit by expanders and bombs, respectively.

   Inputs:
      display -- whether or not the sprite should be displayed on screen
      x, y -- x-y coordinates for the top left corner of the sprite
      hcount, vcount -- current pixel
      width, height -- rectangle dimensions
      color -- color of rectangle

   Outputs:
      pixel -- pixel output for the sprite at the current pixel
*/

module modblob(display,x,y,width,height,color,hcount,vcount,pixel);
   input display;            // whether or not the sprite should be shown
   input [10:0] x;           // x-y coordinates of sprite's top left corner
   input [9:0] y;
   input [10:0] hcount;      // x-y coordinates of displayed pixel
   input [9:0] vcount;
   input [8:0] width;        // dimensions of sprite
   input [4:0] height;
   input [2:0] color;        // color of sprite
   output [2:0] pixel;       // pixel output

   reg [2:0] pixel;

   always @ (x or y or hcount or vcount or width or height
             or display or color) begin

      // if hcount and vcount are currently in the rectangle,
      // color that pixel; otherwise, display black

      if ((hcount >= x && hcount < (x+width)) &&
          (vcount >= y && vcount < (y+height)) && display)
         pixel = color;
      else
         pixel = 3'b000;
   end
endmodule
```

A.19 negedgedetector.v

```
// negedgedetector.v
// 6.111 Fall 2006 Final Project
// Module created by Eddie Fagin

// This is a simple module that just pulses "out" for one clock
//  cycle at the falling edge of "in." I made a decent timing
//  diagram in our report, if you're interested.

module negedgedetector(in, clk, reset, out);
input in, clk, reset;
output out;

reg old, out;
always @ (posedge clk) begin
old <= reset ? 1'b0 : in;
out <= old & ~in;
end
endmodule
```

A.20 ntsc2zbt.v

```
//
// 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.
//
// Original: 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.
//
// EDDIE'S CHANGE: We're only using 24 bits now per line, to store
//  one 24-bit YCrCb value from the NTSC video input. We also only store
//  one quarter of the pixels, because we really don't need quality.
//////////////////////////////////////////////////////////////////////////
// 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);

   input   clk; // system clock
   input   vclk; // video clock from camera
   input [2:0]   fvh;
   input   dv;
   input [23:0]   din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output   ntsc_we; // write enable for NTSC data

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

   // here put the 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 [23:0]   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 : vdata;
  end
     end
```

```
   // synchronize with system clock
   reg [9:0] x[1:0],y[1:0];
   reg [23:0] 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 [23:0] mydata; // Eddie: changed from [35:0]
   always @(posedge clk)
     if (we_edge)
       //Eddie: remove [mydata <= { mydata[23:0], data[1] };]
 mydata <= data[1];
   // compute address to store data in
   wire [18:0] myaddr = {1'b0, y[1][8:0], eo[1], x[1][9:2]};
   // update the output address and data only when four bytes ready
   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   wire       ntsc_we = (we_edge & (x[1][1:0]==2'b00));
   always @(posedge clk)
     if ( ntsc_we )
       begin
  ntsc_addr <= myaddr;
  ntsc_data <= {12'h000, mydata};
       end
endmodule
```

A.21 recorder.v

```
//////////////////////////////////////////////////////////////////////////////////
// Eddie's sound effects module
// Record/playback
// -skeleton provided by 6.111 staff for lab 4
// -modified for lab 4
// -modified again for final project
// -now, a quick playback pulse sets off a full uninterrupted playback.
//
//////////////////////////////////////////////////////////////////////////////////

module recorder(clock_27mhz, reset, playback, recording,
                ready, from_ac97_data, to_ac97_data);
   input clock_27mhz;     // 27mhz system clock
   input reset;           // 1 to reset to initial state
   input playback;        // 1 to initiate full playback
   input recording;       // 1 to write new data to BRAM
   input ready;           // 1 when AC97 data is available
   input [7:0] from_ac97_data;     // 8-bit PCM data from mic

   output [7:0] to_ac97_data;      // 8-bit PCM data to headphone

   reg [10:0] index;        // 11-bit BRAM pointer
   reg [10:0] max;          // 11-bit pointer to end of recording
   reg [2:0] i;             // counter variable (0-7)
   reg notactive;           // indicator; allows module to detect a mode change
   reg we;                  // write enable indicator for BRAM
   reg playback_is_on;      // indicator; on if playback is active
   reg [7:0] to_ac97_data;  // 8-bit PCM data to headphone

   wire [7:0] dout;  // 8-bit PCM data; post-processed version goes to headphone
   wire [7:0] din;   // 8-bit PCM data from microphone

   // from_ac97_data gets directly stored into BRAM if we is active
   assign din = from_ac97_data;

   mem mymem (index, clock_27mhz, din, dout, we);

   always @ (posedge clock_27mhz) begin
      // on a reset, initialize all of the variables
      //  (except to_ac97_data, which gets initialized
      //  on the first assertion of ready)
      if (reset) begin
         max <= 11'b0;
         index <= 11'b0;
         we <= 1'b0;
         i <= 3'd0;
         notactive <= 1'b1;
         playback_is_on <= 1'b0;
      end

      // whenever the ac97 is ready to go...
      else if (ready && (playback | recording | playback_is_on)) begin
         i <= (i == 4'h7) ? 3'b0 : i + 1'b1;  // i <= (i+1) % 8
         // sample data every 8 cycles
         we <= (i == 4'h0 && index < 11'd2047 && recording);

         to_ac97_data <= dout;

         // recording mode
         if (recording)   begin
             // if we just switched to recording mode, reset memory pointer
             // and hang for one "ready" cycle as a safety precaution
```

```
            playback_is_on <= 1'b0;
            if (notactive) begin
                notactive <= 1'b0;
               index <= 13'b0;
            end
            // otherwise, every eight cycles,
            //  index <= (index + 1) % (size of memory - 1)
            else if (i == 4'h0 & index < 11'd2047) index <= index + 1'b1;
         end
         else if (playback_is_on) begin
            // if we just switched from recording mode, reset memory pointer
            // and hang for one "ready" cycle as a safety precaution
            if (notactive) begin
               notactive <= 1'b0;
               max <= index;
               index <= 13'b0;
            end
            // otherwise, on the eighth cycle,
            //  index <= (index + 1) % (size of sound clip - 1)
            else begin
               if (i == 4'h0) begin
                  if (index == max) begin
                     playback_is_on <= 1'b0;
                     notactive <= 1'b1;
                  end
                  else index <= index + 1'b1;
               end
            end
         end
         // If user initiates playback, start up the playback mechanism
         else if (playback) begin
            playback_is_on <= 1'b1;
            notactive <= 1'b1;
         end
      end
      else if (ready) notactive <= 1'b1;
   end
endmodule
```

A.22 rgbfilter.v

```
// rgbfilter.v
// 6.111 Fall 2006 Final Project
// Module created by Eddie Fagin

/*    This module takes processes raw RGB data through
   a multi-step filtering mechanism. All filtering was created
   after lots of experimentation and guesswork. Since we're
   looking for high-intensity LEDs, I wanted this filter to
   remove as much of the environment as possible, allowing
   only bright lights (or reflections of those lights).

   Inputs
      RGBin: Current 24-bit RGB value.
          (8-bit red, green, blue, in that order.)
      clk: 65-MHz system clock
      filter: 8-bit "high-pass filter"
           (actually just a minimum brightness specification)
      greenfilter: if on, removes all red and blue points.
      hcount, vcount: the current pixel coordinates

   Ouputs
      RGBout: the 24-bit filtered output
*/

module rgbfilter(RGBin, clk, filter, greenfilter, hcount, vcount, RGBout);
   input [23:0] RGBin;
   input clk;
   input [7:0] filter;
   input greenfilter;
   input [10:0] hcount;
   input [9:0] vcount;
   output [23:0] RGBout;

   // Camera "frame" coordinates
   parameter [10:0] max_x = 11'd745;
   parameter [10:0] min_x = 11'd45;
   parameter [9:0]  max_y = 10'd564;
   parameter [9:0]  min_y = 10'd66;

   reg [7:0] red0, green0, blue0;
   reg [7:0] red1, green1, blue1;
   reg [7:0] red2, green2, blue2;

   // Stage 1: Remove anything not related to the camera input.
   // (Outside the "box.")
   always @ (posedge clk) begin
       if (hcount >= min_x && hcount <= max_x &&
        vcount >= min_y && vcount <= max_y)
       begin
          red0 <= RGBin[23:16];
          green0 <= RGBin[15:8];
          blue0 <= RGBin[7:0];
       end
       else {red0, green0, blue0} <= 24'h000000;
   end

   // Stage 2: Remove all weak red, green, and blue values.
      always @ (posedge clk) begin
       red1 <= (red0 < filter) ? 8'h00 : red0;
       green1 <= (green0 < filter) ? 8'h00 : green0;
       blue1 <= (blue0 < filter) ? 8'h00 : blue0;
   end
```

```
   // Stage 3: Three colors enter, one color leaves!
   // Only let the strongest of the three primary colors through,
   //  if it exists (it doesn't in white light).
   always @ (posedge clk) begin
      // Take out strong whites, if they exist
      if (red1 >= filter && blue1 >= filter && green1 >= filter) begin
         red2 <= 8'h00;
         green2 <= 8'h00;
         blue2 <= 8'h00;
      end
      // Only allow the strongest of the three colors
      //  (R, G, or B) to pass.
      else begin
         red2 <= (red1 > blue1 && red1 > green1) ? 8'hFF : 8'h00;
         green2 <= (green1 > blue1 && green1 > red1) ? 8'hFF : 8'h00;
         blue2 <= (blue1 > green1 && blue1 > red1) ? 8'hFF : 8'h00;
      end
   end
   // Stage 4: Allow only the green values to pass if "greenfilter" is on.
   // This is used as the output for "Filter Mode."
   assign RGBout = greenfilter ? {8'h00, green2, 8'h00} : {red2, green2, blue2};
endmodule
```

A.23 score.v

```
// score.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan

/*
   This module calculates the score -- every time the ball hits a
   non-magenta brick, the score increments by the current speed.

   Inputs:
      vclock -- system clock
      reset -- reset signal
      collision -- 1 when the ball hits a brick
      pspeed -- ball speed
      color -- color of the brick that was hit

   Outputs:
      score -- current game score
*/

module score(vclock, reset, collision, pspeed, color, score);
   input vclock;
   input reset;
   input collision;
   input [3:0] pspeed;
   input [2:0] color;

   output [31:0] score;

   reg [31:0] score;

   always @ (posedge vclock) begin
      if(reset)                  // on reset, reset the score
         score <= 32'd0;

      // When the ball hits a non-magenta brick, increment the score
      // by the current speed
      else if(collision)
         score <= (color == 3'b101)? score : score + pspeed;

      else
         score <= score;
   end
endmodule
```

A.24 speed.v

```
// speed.v
// 6.111 Fall 2006 Final Project
// Module created by Eddie Fagin

/* This module changes the ball velocity (in the
   horizontal direction) depending on the paddle's
   speed on ball-paddle impact. Basically, think of
   the paddle as having some sort of frictional coefficient.
   This module could be instantiated in order to vary the
   vertical direction as well, but we chose not to implement
   that feature.

   Inputs
      clk: 65-MHz system clock
      ball_vel: if on, ball's going right; otherwise, left
      paddle_vel: the velocity input from the controller - {speed, direction}
               speed:    (00) zero  (01) slow  (10) medium (11) fast
              direction: (00) up    (01) right (10) down   (11) left
      oldspeed: the current ball's x-speed

   Ouputs
      newspeed: the modified ball x-speed, in case the game wants to use it
*/

module speed(clk, ball_vel, paddle_vel, oldspeed, newspeed);
  input clk, ball_vel;
  input [3:0] paddle_vel;
  input [3:0] oldspeed;

  output [3:0] newspeed;
  reg [3:0] newspeed;

  // Hard-wire signals to velocity input for clarity
  wire paddle_right = (paddle_vel[1:0] == 2'b01);
  wire paddle_left = (paddle_vel[1:0] == 2'b11);
  wire [1:0] paddle_speed = paddle_vel[3:2];

  always @ (posedge clk) begin
      // If the paddle's not going left or right,
      //  don't change anything.
      if (!paddle_right && !paddle_left)
       newspeed <= oldspeed;

      else
         // If the ball and paddle are going the same direction,
         //  increase the speed.
         if ((ball_vel && paddle_right) || (~ball_vel && paddle_left))
            newspeed <= (oldspeed + paddle_speed <= 4'hf) ?
               oldspeed + paddle_speed : oldspeed;

         // If they're going in opposite directions, friction should
         //  slow the ball down.
         else newspeed <= (oldspeed > paddle_speed) ?
            oldspeed - paddle_speed : oldspeed;
  end
endmodule
```

A.25 superbreakout.v

```
// superbreakout.v
// 6.111 Fall 2006 Final Project
// Module created by Irene Fan

/*
   This game is our version of Atari's "Super Breakout" (1978). In
   Super Breakout, there are rows of bricks lining the top of the
   screen and a ball that bounces off the side and top walls, breaking
   bricks whenever it ricochets against them. A paddle is located at
   the bottom of the screen, and it can be moved left and right to
   deflect the ball; if the paddle misses the ball and the ball
   touches the bottom of the screen, the game is over. The users
   main motive is to break all the bricks on the screen, and the
   bricks will be a variety of colors to indicate how many times they
   must be hit before they break and disappear from the screen. Our
   version includes levels, multiple lives, and game-affecting items
   dropping from various bricks as they are hit.

   superbreakout.v is the main game module, a big linear FSM that
   updates the state of the game at the end of each frame. First,
   given the current ball position and x-y velocities, we figure out
   where the ball would want to move next if no bricks were in its
   way. We check this anticipated move to see if there will be a
   collision -- in the event of one, we break the appropriate brick
   and change the ball's velocity, simulating a perfect reflection off
   the brick. We also update the position of the paddle and the state
   of any other objects on screen before the beginning of the next
   frame.
      - Bricks are generated from a .coe file -- each level is an
        array of 1x192 (one address per each of the 8*24 = 192 blocks
        filling the screen). Each memory cell holds 4 bits of
        information about whether that block holds a brick (1 bit) and
        what that brick's current color is (3 bits).
      - Bricks' colors reflect the number of remaining hits they need
        before they break: Red = 1 hit; Yellow = 2 hits, Green = 3
        hits, Teal = 4 hits, Blue = 5 hits (Magenta bricks never break)
      - Red bricks drop a bomb when they break; the bomb shrinks the
        paddle if it hits it
      - Green bricks drop expanders, which lengthen the paddle
      - Blue bricks drop life capsules, which grant an extra ball
      - When there are less than 4 bricks on the screen, bombs drop
        randomly every 128 frames; when there is only 1 brick left,
        bombs drop every 32 frames

   Modules instantiated in superbreakout.v:
   - circle -- for the ball, bombs, lifecapsule
   - modblob -- for the paddle
   - drop_game_object -- drops bombs/expanders/lifecapsules and
        updates their position every frame
   - expander_blob -- for the expander
   - block_detector -- given hcount, vcount coordinates for a pixel,
        returns what block the pixel is in; used for displaying
        bricks, finding the block of the current ball coordinates, and
        finding the block of the anticipated ball coordinates
   - det_brick_din -- determines the next color of the brick that was hit
   - brick3rdual, brick3rdualro -- BRAMs with information about the
        bricks; the first is a read/write memory used when displaying
        the bricks and edited upon collisions, and the second is a
        read-only memory that holds the original brick information to
        be used when regenerating levels
```

```
   - display -- display the bricks, ball, paddle, and objects on the screen
   - detect_bombs -- detect when objects hit the paddle
   - speed -- changes the speed of the ball depending on how it hits
        the paddle
   - score -- keeps track of current score
   - negedgedetector -- indicates when brick reset phase is over
   - game_over -- keeps track of when the level is complete and how
        many breakable bricks are left
   - intertitle -- generates intertitle pixels

   Inputs:
      vclock -- system clock (65 Mhz)
      reset -- reset signal (1 to initialize output)
      frameclk -- triggers on negedge of vsync (at the end of every
         frame)
      start -- game start command
      pspeed -- initial x-speed and y-speed (in our game, x-speed
         changes when you are playing with the controller but y-speed
         never changes)
      paddle_velocity -- how many pixels the paddle can move per frame
      force_next_level -- when 1, game skips to next level (_not_ for
         regular playing mode)
      inleft, inright -- left and right button inputs that override
         the controller
      hcount -- horizontal index of current pixel (0..1023)
      vcount -- vertical index of current pixel (0..767)
      fade_on -- 1 to enable the ball fading feature

   Outputs:
      collision -- 1 when ball collides with a brick
      pixel -- 24-bit pixel output
      game_over -- two-bit value indicating current game status --
         (00) - game hasn't started yet
         (01) - game is running
         (10) - game over (lose)
         (11) - game complete (win)
      score -- score of the current game in hex, up to 32 bits
      level -- one of 4 possible levels
      lives -- number of lives left (max 6 lives total)
      title_on -- 1 if an intertitle should be displayed
*/

module superbreakout (vclock, reset, frameclk, start, pspeed,
                      paddle_velocity, force_next_level, inleft,
                      inright, hcount, vcount, fade_on, collision,
                      pixel, game_over, score, level, lives, title_on);
   input vclock;
   input reset;
   input frameclk;
   input start;
   input [3:0] pspeed;
   input [3:0] paddle_velocity;
   input force_next_level;
   input inleft;
   input inright;
   input [10:0] hcount;
   input [9:0] vcount;
   input fade_on;

   output collision;
   output [23:0] pixel;
   output [1:0] game_over;
```

```
output [31:0] score;
output [1:0] level;
output [5:0] lives;
output title_on;

// Parameters
parameter screen_width = 11'd1024;        // screen is 1024 by 768 pixels
parameter screen_height = 10'd768;
parameter ball_radius = 4'd4;             // ball radius is 4 pixels
parameter init_paddle_width = 9'd256;     // paddle starts at 256 pixels
parameter paddle_height = 5'd16;          // paddle height is 16 pixels
parameter brick_width = 8'd128;           // bricks are 128 by 32 pixels
parameter brick_height = 6'd32;
parameter init_copy_size = 10'd192;       // number of blocks per level
                                          //    (there are 8 by 24
                                          //    blocks/screen)
parameter paddle_shrink_amount = 6'd32;   // how much the paddle shrinks
                                          //    when a bomb hits it
parameter paddle_grow_amount = 6'd32;     // how much the paddle grows
                                          //    when an expander hits it
parameter paddle_slow = 5'd5;             // paddle speeds
parameter paddle_normal = 5'd15;
parameter paddle_fast = 5'd30;

// ball_x and ball_y are the coordinates for the center of the circle
reg [10:0] ball_x;
reg [9:0] ball_y;

// check_x and check_y are the coordinates of the next anticipated move
reg [10:0] check_x;
reg [9:0] check_y;

// paddle_x is the x-coordinate of the top left corner of the paddle
reg [10:0] paddle_x;

reg [4:0] paddle_speed;                   // speed of the paddle
reg [8:0] paddle_width;                   // width of the paddle

// Ball starts off traveling toward the top right of the screen
reg ball_xvel, ball_yvel;

// How many horizontal and vertical pixels the ball travels per frame
reg [3:0] xspeed, yspeed;
// modified ball speed assuming it hits the paddle
wire [3:0] newXspeed;

// Pixel outputs for circle and paddle;
//    they are combined with the brick pixel output
//    to produce pixel[23:0] which is outputted to vga
wire [2:0] paddle_pixel, circle_pixel;

reg title_on;
wire [2:0] title_pixel;

// From the block detectors (x and y coordinates of the blocks the
// ball is currently in and wants to move to)
wire [2:0] ball_gridx, check_gridx;
wire [4:0] ball_gridy, check_gridy;

// Registers for the memories
wire [9:0] display_address;
reg [9:0] addr_offset;
reg [9:0] check_address;
```

```
reg [3:0] change_brick_din;
wire [3:0] display_data_out, brick_dout;
reg change_brick_we;
wire [2:0] display_grid_x;
wire [4:0] display_grid_y;
reg [9:0] copy_address;
wire [3:0] regen_data_out;

// Check memory, see if there is a brick where the ball wants to move
// Save brick_dout to these registers
reg [3:0] check_brick_dout, vert_brick_dout, horiz_brick_dout;

// Initial number of breakable bricks in the level
reg [7:0] init_num_bricks;
// Number of breakable bricks left on screen
wire [7:0] num_bricks_left;
// Used in brick reset
reg write_cycle;
reg brick_reset;

// For det_brick_din (stands for determine brick data_in)
wire [3:0] next_brick_color;
reg [2:0] color;               // current block color

// For drop_game_object
   // Pulses high when dropped objects update position
reg update_clock;
   // Pulses high when the object should be dropped
reg drop_bomb, drop_expander, drop_lifecapsule;
   // x, y coordinates of the block from which object is dropped
reg [2:0] drop_gridx;
reg [4:0] drop_gridy;
   // Toggles whether each of these objects is displayed
wire bomb1_display, bomb2_display, bomb3_display, bomb4_display,
     bomb5_display, expander_display, extralife_display;
   // x-coordinates of the objects
wire [10:0] bomb1_x, bomb2_x, bomb3_x, bomb4_x, bomb5_x,
            expander_x, extralife_x;
   // y-coordinates of the objects
wire [9:0] bomb1_y, bomb2_y, bomb3_y, bomb4_y, bomb5_y, expander_y,
           extralife_y;
   // Pixel outputs for the objects
wire [2:0] bomb1_pixel, bomb2_pixel, bomb3_pixel, bomb4_pixel,
           bomb5_pixel, expander_pixel, extralife_pixel;
   // Goes high when the object pixels overlap paddle pixels
wire bomb1_hit_paddle, bomb2_hit_paddle, bomb3_hit_paddle,
     bomb4_hit_paddle, bomb5_hit_paddle, expander_hit_paddle,
     extralife_hit_paddle;
   // 1 if object hit the paddle in the past frame
reg shorten_paddle, shorten_paddle_b1, shorten_paddle_b2,
    shorten_paddle_b3, shorten_paddle_b4, shorten_paddle_b5,
    lengthen_paddle, life_up;

reg [3:0] counter;                  // for the linear fsm
// Bombs drop every 32 frames when there is 1 bricks left
reg [4:0] last_brick_counter;
// Bombs drop every 128 frames when there are < 4 bricks left
reg [6:0] frame_counter;
// Determines from which grid the "random" bombs should fall
reg [2:0] bomb_gridx;
```

```
reg collision;              // 1 when ball collides w/ brick
reg had_collision;          // collision marker

// Is the game halted?
reg stop;                   // game over
reg has_game_started;
reg done;                   // victory
reg[1:0] level;             // 4 levels total
parameter MAX_LEVEL = 3;
assign game_over = has_game_started ?
                    (done ? 2'b11 : stop ? 2'b10 : 2'b01) : 2'b00;
      // game_over shows current game status
      // (00) - game hasn't started yet
      // (01) - game is running
      // (10) - game over (lose)
      // (11) - game complete (win)

reg brick_reset_start;
wire game_ready;
wire level_complete;
reg loselife;
reg [5:0] lives;

// Determine user input
wire left = (paddle_velocity[1:0] == 2'b11) | inleft;
wire right = (paddle_velocity[1:0] == 2'b01) | inright;

// Instantiate a circle for the ball -- always displayed on screen
// Using the center coordinates of the ball here, but display
// it using the top left corner's coordinates in circle.v
circle ball1(.display(1'b1), .vclock(vclock), .reset(reset),
             .frameclk(frameclk), .x(ball_x - ball_radius),
             .y(ball_y - ball_radius), .hcount(hcount),
             .vcount(vcount), .pixel(circle_pixel));

// Paddle -- always displayed, always at bottom of screen
modblob my_paddle(.display(1'b1), .x(paddle_x), .y(10'd752),
                  .width(paddle_width), .height(paddle_height),
                  .color(3'b010), .hcount(hcount), .vcount(vcount),
                  .pixel(paddle_pixel));

// Drops bombs/expanders/lifecapsules and updates their position if
// they are displayed
drop_game_object bomb_manager(.vclock(vclock), .reset(reset),
      .update_clock(update_clock),
      .has_game_started(has_game_started),
      .level_complete(level_complete),
      force_next_level(force_next_level), .loselife(loselife),
      .drop_bomb(drop_bomb), .drop_expander(drop_expander),
      .drop_lifecapsule(drop_lifecapsule), .drop_gridx(drop_gridx),
      .drop_gridy(drop_gridy),
      .shorten_paddle_b1(shorten_paddle_b1),
      .shorten_paddle_b2(shorten_paddle_b2),
      .shorten_paddle_b3(shorten_paddle_b3),
      .shorten_paddle_b4(shorten_paddle_b4),
      .shorten_paddle_b5(shorten_paddle_b5),
      .lengthen_paddle(lengthen_paddle), .life_up(life_up),
      .bomb1_display(bomb1_display), .bomb2_display(bomb2_display),
      .bomb3_display(bomb3_display), .bomb4_display(bomb4_display),
```

```
      .bomb5_display(bomb5_display),
      .expander_display(expander_display),
      .extralife_display(extralife_display), .bomb1_x(bomb1_x),
      .bomb2_x(bomb2_x), .bomb3_x(bomb3_x), .bomb4_x(bomb4_x),
      .bomb5_x(bomb5_x), .expander_x(expander_x),
      .extralife_x(extralife_x), .bomb1_y(bomb1_y),
      .bomb2_y(bomb2_y), .bomb3_y(bomb3_y), .bomb4_y(bomb4_y),
      .bomb5_y(bomb5_y), .expander_y(expander_y),
      .extralife_y(extralife_y));

// Instantiate bombs (red)
// These shorten the paddle
circle bomb1(bomb1_display, vclock, reset, frameclk, bomb1_x,
             bomb1_y, hcount, vcount, bomb1_pixel);
   defparam bomb1.COLOR = 3'b100;
circle bomb2(bomb2_display, vclock, reset, frameclk, bomb2_x,
             bomb2_y, hcount, vcount, bomb2_pixel);
   defparam bomb2.COLOR = 3'b100;
circle bomb3(bomb3_display, vclock, reset, frameclk, bomb3_x,
             bomb3_y, hcount, vcount, bomb3_pixel);
   defparam bomb3.COLOR = 3'b100;
circle bomb4(bomb4_display, vclock, reset, frameclk, bomb4_x,
             bomb4_y, hcount, vcount, bomb4_pixel);
   defparam bomb4.COLOR = 3'b100;
circle bomb5(bomb5_display, vclock, reset, frameclk, bomb5_x,
             bomb5_y, hcount, vcount, bomb5_pixel);
   defparam bomb5.COLOR = 3'b100;

// Instantiate expander -- this one is a double arrow (<---->)
// These lengthen the paddle
expander_blob expander(expander_display, vclock,reset, frameclk,
               expander_x, expander_y, hcount, vcount,
               expander_pixel);
   defparam expander.COLOR = 3'b010;

// Instantiate life capsule
// This gives the player and extra life when it hits the paddle
circle extralife(extralife_display, vclock,reset,
                 frameclk,extralife_x, extralife_y, hcount, vcount,
                 extralife_pixel);
   defparam extralife.COLOR = 3'b001;

// Function takes hcount, vcount and outputs the block they're in
block_detector display_detector(hcount, vcount, display_grid_x,
                                display_grid_y);

// What the next color the brick that was hit should be
det_brick_din nextcolor(vclock, color, next_brick_color);

// Data for each block is 4 bits:
//    1 bit for display (is brick there), 3 for color

// This is the memory for displaying & checking for bricks
// All the levels are in the same .coe file -- addr_offset lets
// you cycle through the levels in the bram.
brick3rdual brickmemory(.addra(display_address + addr_offset),
                        .addrb(check_address + addr_offset),
                        .clka(vclock), .clkb(vclock),
                        .dinb(change_brick_din),
                        .douta(display_data_out),
                        .doutb(brick_dout), .web(change_brick_we));
```

```
// Read-only memory for regenerating bricks
brick3rdualro brickmemory_ro(.addr(copy_address + addr_offset),
                             .clk(vclock), .dout(regen_data_out));

// Display the bricks, ball, paddle, and objects on screen
module display(.vclock(vclock), .reset(game_ready),
               .frameclk(frameclk), .hcount(hcount),
               .vcount(vcount), .grid_x(display_grid_x),
               .grid_y(display_grid_y),
               .brick_data_out(display_data_out),
               .paddle_pixel(paddle_pixel),
               .ball_pixel(circle_pixel),
               .title_pixel(title_pixel),
               .bomb1_pixel(bomb1_pixel),
               .bomb2_pixel(bomb2_pixel),
               .bomb3_pixel(bomb3_pixel),
               .bomb4_pixel(bomb4_pixel),
               .bomb5_pixel(bomb5_pixel),
               .expander_pixel(expander_pixel),
               .extralife_pixel(extralife_pixel),
               .fade_on(fade_on), .title_on(title_on),
               .brick_address(display_address), .pixel(pixel));

// Find the block the current coordinates of the ball is in
block_detector orig_pos_detector(ball_x, ball_y, ball_gridx,
                                 ball_gridy);

// Find the block the coordinates of the ball's anticipated move is
// in
block_detector new_pos_detector(check_x, check_y, check_gridx,
                                check_gridy);

// Detect when objects hit the paddle
detect_bombs detector1(bomb1_pixel, paddle_pixel, bomb1_hit_paddle);
detect_bombs detector2(bomb2_pixel, paddle_pixel, bomb2_hit_paddle);
detect_bombs detector3(bomb3_pixel, paddle_pixel, bomb3_hit_paddle);
detect_bombs detector4(bomb4_pixel, paddle_pixel, bomb4_hit_paddle);
detect_bombs detector5(bomb5_pixel, paddle_pixel, bomb5_hit_paddle);
detect_bombs detector6(expander_pixel, paddle_pixel, expander_hit_paddle);
detect_bombs detector7(extralife_pixel, paddle_pixel, extralife_hit_paddle);

// Speed of the ball can change depending on how it hits the paddle
speed bouncer(.clk(vclock), .ball_vel(ball_xvel),
              .paddle_vel(paddle_velocity),.oldspeed(xspeed),
              .newspeed(newXspeed));

// Keeps track of current game score
score scorebox(.vclock(vclock), .reset(reset),
               .collision(collision), .pspeed(pspeed),
               .color(check_brick_dout[2:0]), .score(score));

// Indicates when brick reset phase is over
negedgedetector brickreset(brick_reset, vclock, reset, game_ready);

// Keeps track of when the level is complete and how many breakable
// bricks are left
game_over brick_watchdog(.vclock(vclock), .reset(game_ready),
                         .init_num_bricks(init_num_bricks),
                         .collision(collision), .color(color),
                         .done(level_complete),
                         .num_bricks_left(num_bricks_left));
```

```
// Generates intertitle pixels
intertitle titles(reset, vclock, hcount, vcount, level, game_over,
                  title_pixel);

/*************** SUPERBREAKOUT STARTS HERE!!!
always @ (posedge vclock) begin

   // On a reset or new level, initialize variables
   if(reset | (has_game_started & level_complete) |
      force_next_level) begin
      ball_x <= screen_width >> 1;
      ball_y <= screen_height - paddle_height - ball_radius;
      ball_xvel <= 1'b1;
      ball_yvel <= 1'b0;
      paddle_x <= (screen_width - init_paddle_width) >> 1'b1;
      stop <= 1'b0;
      counter <= 4'd0;
      collision <= 1'b0;
      brick_reset <= 1'b1;
      brick_reset_start <= 1'b1;
      copy_address <= 10'd0;
      check_address <= 10'd0;
      write_cycle <= 1'b1;
      init_num_bricks <= 8'd0;
      paddle_speed <= 5'd0;
      xspeed <= 4'h0;
      yspeed <= 4'h0;
      title_on <= 1'b1;
      loselife <= 1'b0;
      bomb_gridx <= 3'b000;
      paddle_width <= init_paddle_width;
      shorten_paddle <= 1'b0;
      last_brick_counter <= 5'd0;
      drop_bomb <= 1'b0;
      drop_expander <= 1'b0;
      drop_lifecapsule <= 1'b0;
      update_clock <= 1'b0;
      had_collision <= 1'b0;

      // Reset-specific initialization
      if (reset) begin
         level <= 2'd0;
         addr_offset <= 10'd0;
         done <= 1'b0;
         has_game_started <= 1'b0;
         lives <= 6'b010000;        //start with five lives
      end

      // If the last level played was the final level, then stop the game
      // Otherwise, the user forced a level change, so loop back to
      //   the first level
      else if (level == MAX_LEVEL) begin
         level <= (force_next_level) ? 2'b00 : level;
         addr_offset <= (force_next_level) ? 10'd0 : addr_offset;
         done <= has_game_started ? 1'b1 : 1'b0;
         has_game_started <= ~force_next_level;
      end

      // If the last level played was NOT the final level, then advance
      //    to the next level unconditionally
      else begin
         level <= level + 1'b1;
         addr_offset <= addr_offset + init_copy_size;
         done <= 1'b0;
```

```
      has_game_started <= 1'b0;
   end
end

// If the level hasn't started yet and the user presses
//    the start button, begin game
else if (~has_game_started & start) begin
   xspeed <= pspeed;
   yspeed <= pspeed;
   has_game_started <= 1'b1;
   title_on <= 1'b0;
end

// If the player just died, reset the level
// If this was the player's last life, then end the game
else if (loselife) begin
   if (lives == 6'd1) begin // if this is your last life...
      lives <= 6'd0;
      stop <= 1'b1;         // game over = true
   end
   else begin
      ball_x <= screen_width >> 1'b1;
      ball_y <= screen_height - paddle_height - ball_radius;
      check_x <= screen_width >> 1'b1;
      check_y <= screen_height - paddle_height - ball_radius;
      ball_xvel <= 1'b1;
      ball_yvel <= 1'b0;
      paddle_x <= (screen_width - init_paddle_width) >> 1'b1;
      lives <= lives >> 1'b1;
      loselife <= 1'b0;
      has_game_started <= 1'b0;
      paddle_width <= init_paddle_width;
      shorten_paddle <= 1'b0;
      last_brick_counter <= 5'd0;
      drop_bomb <= 1'b0;
      drop_expander <= 1'b0;
      drop_lifecapsule <= 1'b0;
   end
end

// Game over...
else if (stop) begin
   title_on <= 1'b1;         // turn the "game over" intertitle on
   has_game_started <= 1'b1; // keep game running so that it doesn't
                             //    accidentally reset
end

// Brick-reset phase
// Regenerate all bricks on screen, or a new set of bricks for a
//    new level
else if(brick_reset) begin
   // One latency cycle to give the BRAM some time
   if (brick_reset_start) brick_reset_start <= 0;

   // If we're at the end of the reset cycle, then exit this loop
   else if(check_address == init_copy_size - 1'b1) begin
      brick_reset <= 1'b0;
      change_brick_we <= 1'b0;
      change_brick_din <= 4'b1111;
   end

   else begin
      // Write cycle: turn on the write-enable so that the read-only
      //     memory contents can be copied over to the dynamic BRAM
```

```
      if(write_cycle) begin
         change_brick_din <= regen_data_out;
         init_num_bricks <= (~regen_data_out[3] ||
                             (regen_data_out[2:0] == 3'b101) ||
                             (regen_data_out[2:0] == 3'b000))?
                             init_num_bricks : init_num_bricks + 1'b1;
         copy_address <= copy_address + 1'b1;
         change_brick_we <= 1'b1;
         write_cycle <= 1'b0;
      end

      // Read cycle: get next regen_data_out from the read-only
      // memory to copy over to the BRAM in the next write cycle
      else begin
         check_address <= check_address + 1'b1;
         write_cycle <= 1'b1;
      end
   end
end

// THE CORE OF NORMAL SUPERBREAKOUT MODE
else if(has_game_started && ~stop && ~done) begin
   // Keeps track of whether or not objects have hit the paddle
   //    in the last frame
   shorten_paddle <= shorten_paddle || bomb1_hit_paddle ||
                     bomb2_hit_paddle || bomb3_hit_paddle ||
                     bomb4_hit_paddle || bomb5_hit_paddle;
   shorten_paddle_b1 <= shorten_paddle_b1 || bomb1_hit_paddle;
   shorten_paddle_b2 <= shorten_paddle_b2 || bomb2_hit_paddle;
   shorten_paddle_b3 <= shorten_paddle_b3 || bomb3_hit_paddle;
   shorten_paddle_b4 <= shorten_paddle_b4 || bomb4_hit_paddle;
   shorten_paddle_b5 <= shorten_paddle_b5 || bomb5_hit_paddle;
   lengthen_paddle <= lengthen_paddle || expander_hit_paddle;
   life_up <= life_up || extralife_hit_paddle;

   // At the end of every frame, move the ball and update the
   //    position of everything

   if(frameclk) begin
      // Update counter for our massive FSM
      counter <= counter + 1'b1;
      // These 2 counters are for end-game bomb-dropping
      last_brick_counter <= last_brick_counter + 1'b1;
      frame_counter <= frame_counter + 1'b1;

      // STEP 1: See where would the ball move in the next frame
      // if there were no bricks
      // note: working with coordinates of the ball's center

      // Ball changing x-direction
      if(ball_xvel) begin                        // moving right
         // if the ball is touching the right wall, bounce
         // toward the left
         if((ball_x + ball_radius) == screen_width) begin
            check_x <= ball_x - xspeed;
            ball_xvel <= 1'b0;
         end
         // keep the ball from going through the wall
         else if((ball_x + ball_radius + xspeed) > screen_width)
            check_x <= screen_width - ball_radius;
         // otherwise move right
         else
            check_x <= ball_x + xspeed;
```

```
   end
   else begin                                    // moving left
      // if the ball is touching the left wall, bounce
      // toward the right
      if(ball_x == ball_radius) begin
         check_x <= ball_x + xspeed;
         ball_xvel <= 1'b1;
      end
      // keep ball from going through the wall
      else if(ball_x < (xspeed + ball_radius))
         check_x <= ball_radius;
      // otherwise move left normally
      else
         check_x <= ball_x - xspeed;
   end

   // moving vertically
   if(ball_yvel) begin                           // moving down
      // if the ball touches the bottom of the screen, you
      // lose a life
      if((ball_y + ball_radius) == screen_height)
         loselife <= 1'b1;
      // check to see if the ball hits the paddle
      else if((ball_y + ball_radius + yspeed)
              >= (screen_height - paddle_height)) begin
         // outside of the bounds of the paddle
         if(((ball_x + ball_radius) < paddle_x)
            || (ball_x > (paddle_x + paddle_width)))
            check_y <= ((ball_y + ball_radius + yspeed) >= screen_height)?
                       screen_height - ball_radius : ball_y + yspeed;
         else begin
            // ball already missed paddle earlier
            if((ball_y + ball_radius) >= (screen_height - paddle_height))
               check_y <= screen_height - ball_radius;
            // ball hits paddle!!
            else begin
               xspeed <= newXspeed;
               check_y <= screen_height - paddle_height - ball_radius;
               ball_yvel <= 1'b0;
            end
         end
      end
      // otherwise move down the screen normally
      else
         check_y <= ball_y + yspeed;
   end
   else begin                                    // moving up
      // hit the top of the screen, bounce down
      if(ball_y == ball_radius) begin
         check_y <= ball_y + yspeed;
         ball_yvel <= 1'b1;
      end
      // ball can't go through top of screen
      else if(ball_y < (yspeed + ball_radius))
         check_y <= ball_radius;
      // otherwise, ball moves upwards normally
      else
         check_y <= ball_y - yspeed;
   end
end

// STEP 2: 1 clock cycle later, take the grid coordinates of
// the anticipated move and use those to set the address so the
```

```
// corresponding block can be checked in memory
else if(counter == 4'd1) begin
   counter <= counter + 1'b1;
   change_brick_we <= 1'b0;

   // set this address to get information about the block
   //    enclosing the new coordinates
   check_address <= (check_gridy << 3) + check_gridx;

   // paddle speed doesn't change if you're playing the game
   // with the FPGA buttons
   if (inleft | inright) paddle_speed <= paddle_normal;
   else
      case (paddle_velocity[3:2])
         2'b00: paddle_speed <= 5'd0;
         2'b01: paddle_speed <= paddle_slow;
         2'b10: paddle_speed <= paddle_normal;
         2'b11: paddle_speed <= paddle_fast;
      endcase
end

// Extra clock cycle to account for read and write to BRAM
else if (counter == 4'd2) counter <= counter + 1;

// STEP 3: 3 clock cycles later, store the info for block
// we're checking.  If the ball is moving diagonally across
// blocks, need the data for the 2 adjacent blocks as well
else if(counter == 4'd3) begin
   counter <= counter + 1'b1;
   check_brick_dout <= brick_dout;

   // Set address for the vertically adjacent (in the
   // ball_yvel direction) block
   check_address <= (check_gridy << 2'b11) + ball_gridx;
end

// Extra clock cycle for the BRAM
else if(counter == 4'd4)
   counter <= counter + 1'b1;

// Store the information for the vertical block; set the
// address for the horizontally adjacent block
else if(counter == 4'd5) begin
   counter <= counter + 1'b1;
   vert_brick_dout <= brick_dout;

   // Check horizontal brick
   check_address <= (ball_gridy << 2'b11) + check_gridx;
end

// Extra clock cycle for the BRAM
else if(counter == 4'd6) counter <= counter + 1'b1;

// Store the information for the horizontal block
else if(counter == 4'd7) begin
   counter <= counter + 1'b1;
   horiz_brick_dout <= brick_dout;
end

// STEP 4: Update everything on screen
else if(counter == 4'd8) begin
   counter <= counter + 1'b1;

   // STEP 4a: Collision Detection -- if there is a
   // collision, leave the ball in the same location for a frame
   // and just change the x and/or y velocity
```

```
   // A) Diagonal case: the ball wants to move to a block
   //    diagonal from its current one
   if((check_gridx != ball_gridx) && (check_gridy != ball_gridy)) begin
      // There will be a collision if there is a brick in
      // the diagonal block, or if there are bricks both
      // vertically and horizontally adjacent to the current block
      if(check_brick_dout[3]
         || (vert_brick_dout[3] && horiz_brick_dout[3]))
         had_collision <= 1'b1;

      // Both adjacent blocks have bricks; break vertically
      // adjacent one first
      if(vert_brick_dout[3] && horiz_brick_dout[3]) begin
         // break the vertical one first
         check_address <= (check_gridy << 2'b11) + ball_gridx;
         ball_yvel <= ~ball_yvel;         // change the y velocity
         color <= vert_brick_dout[2:0];
      end

      // Diagonal and vertically adjacent blocks have bricks;
      // break diagonal one
      else if(check_brick_dout[3] && vert_brick_dout[3]) begin
         // break the diagonal block
         check_address <= (check_gridy << 2'b11) + check_gridx;
         ball_yvel <= ~ball_yvel;         // change the y velocity
         color <= check_brick_dout[2:0];
      end

      // If only the diagonal and horizontally adjacent
      // blocks have bricks, break the diagonal one
      else if(check_brick_dout[3] && horiz_brick_dout[3]) begin
         // break the diagonal brick
         check_address <= (check_gridy << 2'b11) + check_gridx;
         ball_xvel <= ~ball_xvel;         // change the x velocity
         color <= check_brick_dout[2:0];
      end

      // Only diagonal block has a brick; break it and
      // reflect back off the corner
      else if(check_brick_dout[3]) begin
         check_address <= (check_gridy << 2'b11) + check_gridx;
         ball_xvel <= ~ball_xvel;         // change both x,y velocities
         ball_yvel <= ~ball_yvel;
         color <= check_brick_dout[2:0];
      end
   end

   // B) Horizontal/Vertical/Same Block Case
   else if(check_brick_dout[3]) begin               // hit a brick
      had_collision <= 1'b1;
      color <= check_brick_dout[2:0];
      check_address <= (check_gridy << 2'b11) + check_gridx;

      // change the appropriate velocity
      ball_yvel <= (check_gridy == ball_gridy) ? ball_yvel : ~ball_yvel;
      ball_xvel <= (check_gridx == ball_gridx) ? ball_xvel : ~ball_xvel;
   end
end

// Extra cycle for det_brick_din to figure out what the next
// color of the brick that was hit should be
else if(counter == 4'd9)
   counter <= counter + 1'b1;
```

```
// STEP 4b: Set the next color to be the data_in to the brick RAM
else if(counter == 4'd10) begin
   counter <= counter + 1'b1;
   change_brick_din <= next_brick_color;
end

else if(counter == 4'd11) begin
   counter <= counter + 1'b1;

   // STEP 4c: If there is a collision (had_collision = 1),
   // pulse collision signal high and write the next brick
   // information to that block
   if(had_collision)   begin
      change_brick_we <= 1'b1;
      collision <= 1'b1;

      // STEP 4d: If the brick is red/green/blue, drop a
      // bomb/expander/lifecapsule from the top center of that block
      if((color == 3'b100) || (color == 3'b010)
          || (color == 3'b001)) begin
         drop_gridx <= (vert_brick_dout[3] && horiz_brick_dout[3])?
                       ball_gridx : check_gridx;
         drop_gridy <= check_gridy;
         case(check_brick_dout[2:0])
            3'b100: drop_bomb <= 1'b1;
            3'b010: drop_expander <= 1'b1;
            3'b001: drop_lifecapsule <= 1'b1;
            default: begin
               drop_bomb <= 1'b0;
               drop_expander <= 1'b0;
               drop_lifecapsule <= 1'b0;
            end
         endcase
      end
   end

   // STEP 4e: If there wasn't a collision but there are < 4
   // breakable bricks left on screen, drop some bombs
   else begin
      if(((num_bricks_left == 1) && (last_brick_counter == 5'd31)) ||
         ((num_bricks_left < 3'd4) && (frame_counter == 7'd127))) begin
         drop_gridx <= bomb_gridx;
         drop_gridy <= 1'b0;

         // Update where to next randomly drop a bomb
         bomb_gridx <= bomb_gridx + 4'h5;
         drop_bomb <= 1'b1;
      end

      // Complete the anticipated move
      ball_x <= check_x;
      ball_y <= check_y;
   end

   // STEP 4f: Update all object positions via drop_game_objects.v
   update_clock <= 1'b1;

   // STEP 4g: Shorten/lengthen paddle if hit by bomb/expander
   if(shorten_paddle || lengthen_paddle) begin
      if(shorten_paddle) begin
         if(paddle_width <= paddle_shrink_amount)
            paddle_width <= 9'd0;
         else begin
            paddle_x <= paddle_x + (paddle_shrink_amount >> 1'b1);
```

```
                  paddle_width <= paddle_width - paddle_shrink_amount;
               end
            end
            else begin
               if((paddle_width + paddle_grow_amount) > init_paddle_width) begin
                  paddle_x <=
                   (paddle_x<((init_paddle_width-paddle_width)>>1'b1))?
                              11'd0 :
                              paddle_x-((init_paddle_width-paddle_width)>>1'b1);
                  paddle_width <= init_paddle_width;
               end
               else begin
                  paddle_x <= (paddle_x < (paddle_grow_amount >> 1'b1))?
                              11'd0 : paddle_x - (paddle_grow_amount >> 1'b1);
                  paddle_width <= paddle_width + paddle_grow_amount;
               end
            end
         end

         // STEP 4h: Update paddle position
         else begin
            if(left)
               paddle_x <=
                (paddle_x < paddle_speed)? 11'd0 : (paddle_x - paddle_speed);
            else if(right)
               paddle_x <=
                ((paddle_x + paddle_width + paddle_speed) > screen_width)?
                   (screen_width - paddle_width) : (paddle_x + paddle_speed);
            else paddle_x <= paddle_x;
         end

         // STEP 4i: Update # of lives if lifecapsule was caught
         if(life_up)
            lives <= (lives == 6'b100000) ? lives : lives << 1'b1;
      end

      // Clear variables when done using them for the frame
      else if (counter == 4'd12) begin
         counter <= 4'h0;
         had_collision <= 1'b0;
         collision <= 1'b0;
         change_brick_we <= 1'b0;
         shorten_paddle_b1 <= 1'b0;
         shorten_paddle_b2 <= 1'b0;
         shorten_paddle_b3 <= 1'b0;
         shorten_paddle_b4 <= 1'b0;
         shorten_paddle_b5 <= 1'b0;
         shorten_paddle <= 1'b0;
         lengthen_paddle <= 1'b0;
         life_up <= 1'b0;
         drop_bomb <= 1'b0;
         drop_expander <= 1'b0;
         drop_lifecapsule <= 1'b0;
         update_clock <= 1'b0;
      end

      // Just update game once/frame
      else begin
         paddle_x <= paddle_x;
         ball_x <= ball_x;
         ball_y <= ball_y;
      end
   end
end
endmodule
```

A.26 trackerblob.v

```
// trackerblob.v
// 6.111 Fall 2006 Final Project
// Module created by Eddie Fagin

/* Trackerblob is a specialized sprite controller that remembers the path of the
   pointer-block sprite. The module does this by updating a BRAM storing the
   3-bit color value of each 16 x 16 pixel block on the 1024 x 768 screen.
   Inputs
      (x,y): the current pointer position, provided by the block_test module
      (hcount, vcount): current VGA output position
      vclock: 65-MHz system clock
      frameclk: goes high at the end of each frame (negative edge of vsync)
      reset: while high, this signal clears the BRAM canvas.
             The reset signal sent to this module is guaranteed to be high
             for at least one frame-clock cycle.
      velocity: 4-bit speed and direction representation.
                See the code for more details.
      color: the current three-bit draw color set by switch[6:4]
      draw_on: this module's "enable" bit
   Ouputs
      pixel: determines the color of the current (hcount, vcount);
             eight possible values.
*/

module trackerblob(x,y,hcount,vcount,vclock, reset,color,draw_on,pixel);
   parameter WIDTH = 8'd64;      // default width: 64 pixels
   parameter HEIGHT = 8'd64;     // default height: 64 pixels
   parameter COLOR = 3'b111;  // default color: white

   input [10:0] x,hcount;
   input [9:0] y,vcount;
   input vclock, reset;
   input [2:0] color;
   input draw_on;
   output [2:0] pixel;
   reg [2:0] pixel;

   // BRAM i/o connections, not including vclock
   reg [2:0] drawcolor;         // din
   wire [2:0] tracepixel;       // dout
   reg [12:0] addr;           // addr
   reg we;               // we

   // This 8,192 x 3 BRAM stores the "picture" we've painted so far
   // as a collection of 16x16 pixel blocks.
   tracersingle t(addr,vclock,drawcolor,tracepixel,we);

   always@(posedge vclock) begin
      // While reset is held down, cycle through BRAM
   //  and clear it. The reset signal is guaranteed
   //  to be held down sufficiently long.
   if (reset) begin
         addr <= addr + 1'b1;
         we <= 1'b1;
         drawcolor <= 3'b000;
      end

   // Latch the color and set write-enable high at the cursor.
   // Otherwise, model each BRAM location as a 16 x 16 block.
   // We are always drawing the cursor as a 16 x 16 white sprite,
```

```
   //  using the top left corner as the reference point.
      else begin
         drawcolor <= color;
         addr <= {vcount[9:4], hcount[10:4]};
         we <= draw_on ? (hcount == x && vcount == y) : 1'b0;

      if ((hcount >= x && hcount < (x+WIDTH)) &&
          (vcount >= y && vcount < (y+HEIGHT)))
                 pixel   <= COLOR;  // draw cursor
         else pixel <= tracepixel;    // or draw rest of canvas
   end
   end
endmodule
```

A.27 video_decoder.v

```
// Eddie's note: I modified this code so that YCRCB was manipulated
// as a 24-bit value instead of a 30-bit value. (8 bits each, as opposed
// to 10 bits each.) Javier told me that the standard had changed to 24
// bits since his original module, and gave me a personal go-ahead to change
// this code. My changes were really minor, but necessary. They're also
// somewhat spread out and Javier sort of told me exactly what to do, so
// I don't really feel like I should take much credit for this.
//
// We also did not include the adv7185 modules to save space.
// These modules can be found on the 6.111 Fall 2005 web page.
//
//
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
//
// This file contains the ntsc_decode and adv7185 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 - 8-bit input from chip. should map to pins [19:12]
   // 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 [7:0] tv_in_ycrcb; // modified for 8 bit input - should be P[19:12]
   output [23:0] ycrcb;
   output  f;
   output  v;
   output  h;
   output  data_valid;

   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 [7:0]  y = 8'h00;  // luminance
reg [7:0]  cr = 8'h00; // chrominance
reg [7:0]  cb = 8'h00; // 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 == 8'h00) ? SYNC_2 : SYNC_1;
       SYNC_2: current_state <= (tv_in_ycrcb == 8'h00) ? SYNC_3 : SYNC_1;
       SYNC_3: current_state <= (tv_in_ycrcb == 8'h80) ? SAV_f1_cb0 :
            (tv_in_ycrcb == 8'h9d) ? EAV_f1 :
            (tv_in_ycrcb == 8'hab) ? SAV_VBI_f1 :
            (tv_in_ycrcb == 8'hb6) ? EAV_VBI_f1 :
            (tv_in_ycrcb == 8'hc7) ? SAV_f2_cb0 :
            (tv_in_ycrcb == 8'hda) ? EAV_f2 :
            (tv_in_ycrcb == 8'hec) ? SAV_VBI_f2 :
            (tv_in_ycrcb == 8'hf1) ? EAV_VBI_f2 : SYNC_1;

       SAV_f1_cb0: current_state <= (tv_in_ycrcb == 8'hff) ? SYNC_1 : SAV_f1_y0;
       SAV_f1_y0: current_state <= (tv_in_ycrcb == 8'hff) ? SYNC_1 : SAV_f1_cr1;
       SAV_f1_cr1: current_state <= (tv_in_ycrcb == 8'hff) ? SYNC_1 : SAV_f1_y1;
       SAV_f1_y1: current_state <= (tv_in_ycrcb == 8'hff) ? SYNC_1 : SAV_f1_cb0;

       SAV_f2_cb0: current_state <= (tv_in_ycrcb == 8'hff) ? SYNC_1 : SAV_f2_y0;
       SAV_f2_y0: current_state <= (tv_in_ycrcb == 8'hff) ? SYNC_1 : SAV_f2_cr1;
       SAV_f2_cr1: current_state <= (tv_in_ycrcb == 8'hff) ? SYNC_1 : SAV_f2_y1;
       SAV_f2_y1: current_state <= (tv_in_ycrcb == 8'hff) ? 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[5:4] : 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[6] : f;
     end
endmodule
```

A.28 vram_display.v

```
//////////////////////////////////////////////////////////////////////////
// I. Chuang's code, grabbed and modified from the Fall '05 6.111 website.
//    -EDDIE'S MODIFICATION:
//      8 bits x 4 pixels / line --> 24 bits x 1 pixel / line
//
// 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 24 bits here.
// The original module stored four 8-bit values per line - we changed that to
// only store one 24-bit value per line and read accordingly.

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 [23:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

   wire [18:0]   vram_addr = {1'b0, vcount, hcount[9:2]};

   wire [1:0]   hc4 = hcount[1:0];
   reg [23:0]   vr_pixel;
   reg [35:0]   vr_data_latched;
   reg [35:0]   last_vr_data;

// This part was modified by Eddie to read a 24-bit RGB value.
   always @(posedge clk) begin
     last_vr_data <= (hc4==2'd3) ? vr_data_latched : last_vr_data;
     vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched;
  vr_pixel <= last_vr_data[23:0];
end
endmodule
```

A.29 xvga.v

```
// NOTE: EDDIE AND IRENE DID NOT CHANGE THIS FILE, WE JUST USED IT!

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
```

A.30 zbt_6111.v

```
// NOTE: EDDIE AND IRENE DID NOT CHANGE THIS FILE!! WE ONLY USED IT.

// 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
```