DIGITAL DESIGN LABS

Super FPGA Bros

A Novel Approach to Classic Gaming

Douglas Albert & Kevin Marengo

12/10/2009

This project provides an intuitive full body gestural interface for the classic video game, Super Mario Bros. The labkit processes a video feed of the player to determine the location of the arms and head, and uses this information to control the character inside the video game. This creates a more immersive and intuitive interface for the user, creating a brand new experience for a timeless classic.

Table of Contents

List of Tables iii

List of Figures iii

1. Overview 1

1.1 Background 1

1.2 Description of Functionality 1

2 Module Description 2

2.1 Video Capture and Gesture Recognition Modules 2

2.1.1 Overview 2

2.1.2. Camera Capture 3

2.1.3 Gesture Recognition 3

2.1.3.5 *runorwalk* 5

2.1.4 Video Display 6

2.2 Game Logic Modules 7

2.2.1 Game Wrapper 7

2.2.2 Frame Buffer 8

2.2.3 Level Creator 8

2.2.4 Game FSM 10

2.2.5 Sprite RAM 11

2.2.6 Sprite Generator 11

2.2.7 Player Controller 12

3 Insights and Observations 13

3.1 Testing & Debugging 13

3.1.1 Video Logic Module 13

3.1.2 Game Logic Module 15

4 Conclusion 16

Appendix 17

Appendix I – zbt_6111_sample.v 17

Appendix II – Video Module 28

Appendix V – Game Wrapper Module 42

List of Tables

Table 1: Behavioral Pixel Values & Descriptions 12

List of Figures

Figure 1: Figure 1: Top Level Block Diagram 1
Figure 2: Camera Capture & Gesture Recognition Block Diagram 2
Figure 3: Photo of Center of Mass and Line Calculators at work 5
Figure 4: Game Logic block Diagram 7
Figure 5: Tiles Used in Game 10
Figure 6: Photo of game in action. 13

1. Overview

1.1 Background

The original ***Super Mario Bros*** was released by Nintendo in 1985 for the Nintendo Entertainment System. This simple 2D side scroller held the title of best-selling video game of all time until 2009, where it was outsold by Nintendo's own Wii Sports. This project attempts to create a twist on this Nintendo classic. Where the original game was controlled using a simple gamepad containing 4 directional and 2 action buttons, our game allows the player to control the on-screen character using natural movement and gestures. The player's actions in the real world are replicated by the character onscreen, generating an extra level of immersion not found in the original game.

1.2 Description of Functionality

The planned project is divided into two major modules: the camera capture and gesture recognition module, and the game logic module. The figure below gives us a very broad top level overview of our system. By partitioning the design as such, we can work on each module independently allowing both developers to work in parallel, hopefully speeding up the design & implementation process.

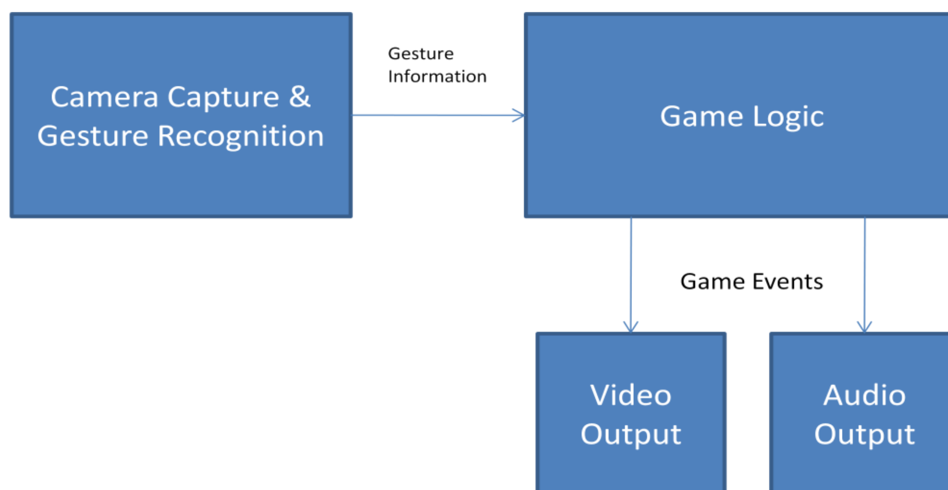

Figure 1: Top Level Block Diagram

The Game Logic module contains circuitry and logic for creating an interactive game world to display on screen, while the Camera Capture module performs the necessary signal processing to convert raw pixel information from an external NTSC camera into something useable by our project. In the game the player is able to navigate in the 2D plane. The player will be faced with many hazards to navigate, both in the form of static environmental obstacles and dynamic enemies that can harm the player and end the game. If the player can successfully navigate to the end of the level, the player can touch a large flag pole to win the game and end the level. Rather than force the user to learn a set of arcane gestures in order to play the game, the gesture modules look for natural movement in the player's body. In order to make the in game character jump, our player simply has to jump in the real

world. In order to make the in game character run forward, the player simply needs to make the motion of running forward.

Operation of the game is a simple two stage process for the user. After the compiled version of the program is loaded onto the labkit a simple calibration sets two thresholds relative to the player's head. If the player's head crosses the upper threshold (as would occur if the player jumps) a jump signal is sent to the game logic. If the player's head crosses the lower threshold, a crouch signal is sent to the game logic. After camera has been calibrated the player can stand in front of the background and commence playing the game. If the game is stopped, as is the case when the player wins or loses the game, pressing the 'Enter' button on the labkit resets the system and allows the user to play again.

2 Module Description

2.1 Video Capture and Gesture Recognition Modules

2.1.1 Overview

The purpose of the video capture and gesture recognition modules is to identify if the user is performing an action like a crouch or jump and if the user is walking left or right or not moving at all. These signals are then sent to the game logic module where they are used to make Mario perform the same actions as the user.

To interact with the system, the player stands in front of the camera wearing a red hat similar to Mario's and a different color marker on the outside of each arm, the default choices being green and yellow. To increase the accuracy of the system, the player stands in front of a black sheet and can wear black clothing and gloves so that the only objects in the video frame with color are the hat and arm markers. The center of mass for each color is calculated and tracked so that they can be used to determine the user's actions. This portion of the project was designed and implemented by Kevin Marengo.

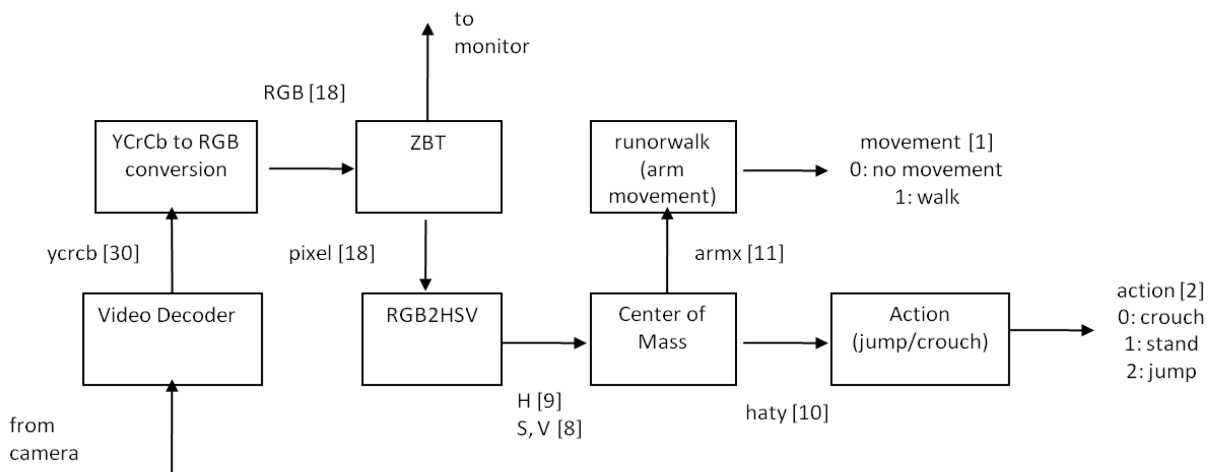

Figure 2: Camera Capture & Gesture Recognition Block Diagram

2.1.2. Camera Capture

2.1.2.1 ADV7185, NTSC Decoder

The ADV7185 module allows the camera to communicate with the FPGA so the data from the video can be used by the other modules. This module was written by Nathan Ickes and was not modified. The NTSC Decoder module takes the NTSC data from the camera as an input and outputs the YCrCb luminance and chrominance values of each pixel as well as the F, V, and H scanning signals of the camera. This module was also written by J. Castro and not modified.

2.1.2.2 YCrCb2RGB

The YCrCb2RGB module takes the 30-bit YCrCb luminance and chrominance values of the pixels from the NTSC Decoder module and calculates the equivalent 24-bit RGB value with 8 bits for R, G, and B separately. The conversion to RGB is necessary because the monitor requires the RGB values to display the video stream from the camera in color. These RGB values will also be used to calculate the HSV values of the pixels to be used in the gesture recognition modules. This conversion is done before the pixel values are stored into the ZBT memory so the RGB values are what are stored. This module was written by Xilinx.

2.1.2.3 NTSC to ZBT

The ntsc_to_zbt module was modified from the original by I. Chuang to use RGB pixel values rather than black and white pixel values. The ZBT word size is 36 bits, so each 36-bit word is made up of two 18-bit pixels, composed of the 6 most significant bits for the R, G, and B values of each full 24-bit pixel. A full 36-bit word is therefore completed on every other clock cycle. This module also synchronizes the 27MHz video clock with the 65MHz system clock, so the data can be used with the other modules that run on the 65 MHz system clock.

2.1.3 Gesture Recognition

2.1.3.1 RGB2HSV

The first thing that occurs in the gesture recognition is the conversion of the RGB values to HSV (Hue, Saturation, and Value) values. Since the functionality of the project requires the tracking of three colors (red, green, and yellow) and the added functionality would require the tracking of 5 colors total, it was suggested that the HSV values be used rather than the RGB values. To track the colors, the Hue (absolute color) value for each color was calculated on the scale of 0-360. Focusing on this 0-360 scale still allowed for a large difference in the values between each color used and was a much easier scale to use when making thresholds for the different colors than the R, G, and B values separately.

Two instances of the RGB2HSV conversion would run at a time because the RGB values were taken from the ZBT RAM 2 pixels at a time since there are two 18-bit pixel values stored at each address. In each instance of the RGB2HSV module, the 8-bit R, G, and B values are used to calculate the 9-bit H value and 8-bit S and V values. The H value was 9 bits to keep the standard 0-360 range of the hue. The S and V values are usually from 0-1, but were scaled up to 0-255 to make full use of the 8 bits. Two pipelined dividers, called divider_hsv, created by the Xilinx Coregen were used for the 16-bit division used in the hue and saturation calculations. Some of the signals had to be delayed to match up with the correct

outputs because of the 18 clock cycles the 16-bit division takes. The NTSC address is delayed in the top-most module so that it matches up with the correct HSV values.

2.1.3.2 Center of Mass

The center of mass module calculates the three centers of mass for the red hat, green arm marker, and yellow arm marker. The module takes in the H, S, and V values of two pixels and the delayed NTSC address corresponding to these two pixels. Both pixels are then checked to see if they are both within the same color threshold for their H values. For example, the H value for both pixels must be within the threshold range set for the same color in order to be counted towards a color's center of mass. Requiring both pixels to be the same color makes the center of mass more robust against noise since noise usually only causes one pixel value to change, and it's not likely to change two adjacent pixels to the same color.

If both pixels fall within the set threshold for a color, then those pixels' x and y coordinates are added to that color's running x and y coordinate sums and the pixel count for that color increase by two. Since the screen refreshes at 60Hz, the new centers of mass are output 60 times a second. The red hat center of mass is always output, but the module only outputs one arm center of mass. The module checks which arm color has a higher pixel count to choose which color center of mass is output. I chose to do this because in theory only one of the arm colors should be present. The arm markers are on the outside of the arms only, so only one of those markers should be showing on the arm facing the camera. Depending on which color arm marker is used, the module outputs a direction signal to designate the direction the player is facing.

Figure 3: Photo of Center of Mass and Line Calculators at work

2.1.3.3 Line Calculator

The line calculator module is used to calculate where the jump line and crouch line thresholds should be set. When the game is started, the player stands in front of the camera and someone presses button0 to set the jump and crouch lines. The module takes the center of mass of the hat and two thresholds are set equidistant above and below this center of mass.

2.1.3.4 Action

In the action module, it is determined if the player is just standing, jumping, or crouching. The module compares the hat center of mass to the jump and crouch line thresholds set using the line calculator module. If the player jumps, the hat should go above the jump line threshold and a jump signal is output. If the play crouches below the crouch line threshold, a crouch signals is output. When the player is standing still or walking, the hat center of mass should be between the two lines and the standing signal is output. The action signal for jump, stand, or crouch is used in the game logic.

2.1.3.5 Runorwalk

The purpose of the runorwalk module is to determine if the player is pumping their arm in order to resemble walking movement. A circular buffer containing the current arm x coordinate and the

previous seven x coordinates is used to calculate an average of the values to use as a reference. Only the x coordinate is used since we are looking for horizontal movement by the arm. If the arm position changes by an amount greater than the threshold for bouncing due to noise, then it is accepted that the player is probably swinging their arm. The direction that the arm is swinging can be tracked, so a count is kept for how many screen refresh cycles the arm has been swinging in the same direction. At the end of an arm swing, the signal "beat" is inverted and the number of screen refresh cycles is stored. If this number of cycles is above a threshold value, then it means the player's arm was probably swinging and probably not noise causing the center of mass to jump around. In this case, the movement signal is asserted meaning the play is walking.

2.1.4 Video Display

There are a few small modules that contribute to the video display. The ZBT RAM is used to store the camera input. The zbt_6111 module writes the RGB data to the ZBT address on every odd clock cycle when write enable is asserted. On every even clock cycle, the data is being read from the ZBT. This module was written by I. Chuang. The XVGA module generates the important display signals like hcount and vcount which are used to determine the pixel coordinates. The VRAM module was modified for color to read from the ZBT every other clock cycle since each ZBT word contains two pixels and on every clock cycle a pixel is output. These modules were written by I. Chuang as well. The video displayed for the video capture and gesture recognition was the color RGB values of the incoming video and some white crosshair lines used to track centers of mass.

2.2 Game Logic Modules

2.2.1 Game Wrapper

For the sake of modularity, each of the major subsystems of our project are abstracted away from one another. The game logic is enclosed by one large game_wrapper module, which contains all of the internal logic of the game's operation. We can consider the game logic a black box that takes in the relevant timing information from the FPGA and labkit as well as gesture information from the camera, and outputs the relevant video data to be parsed by the VGA decoder. This portion of the project was designed and implemented by Douglas Albert.

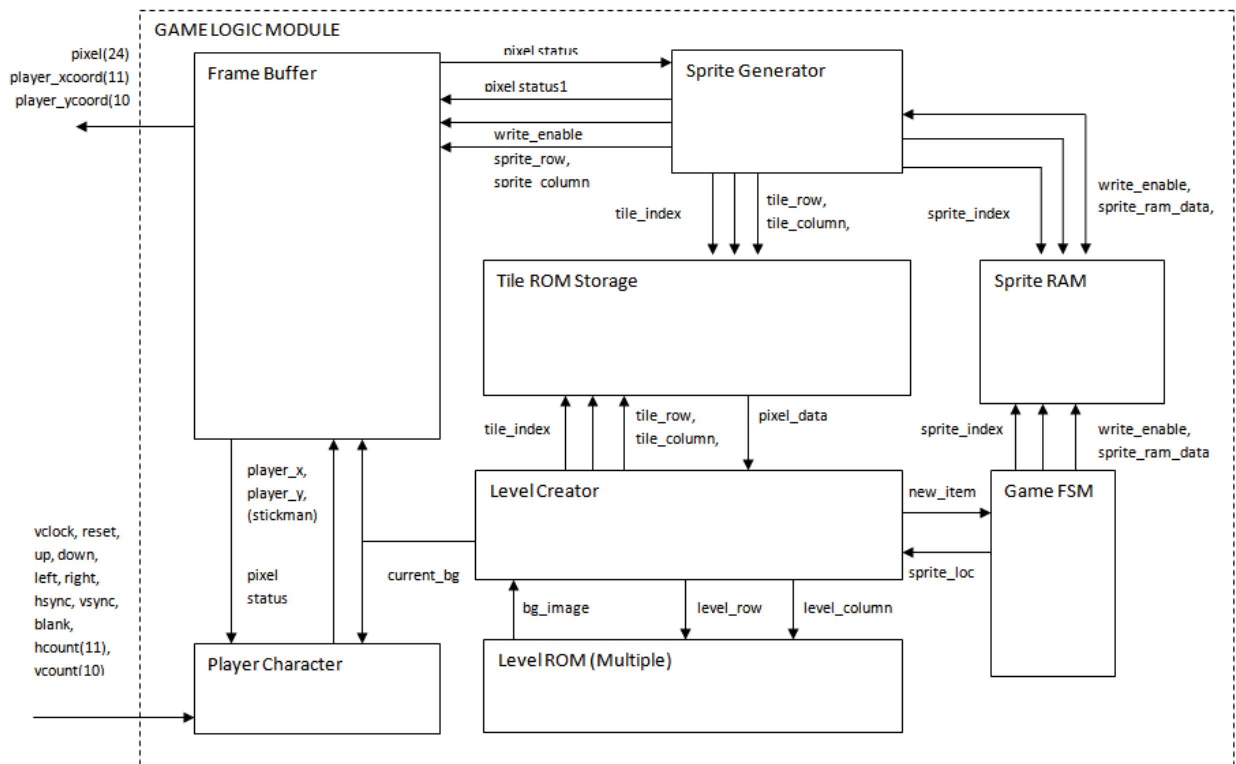

Figure 4: Game Logic Block Diagram

As we can see in the above figure, the game logic takes in video control signals from the XVGA module, as well as arbitrary inputs that control the action inside the game. These inputs were initially wired to the pushbuttons on the labkit, allowing us to work on the game logic and control logic in parallel. In the final version of the project, these signals were wired to the output of Kevin's gesture modules.

The entire Game Logic module is designed to be very modular, enabling us to add extra functionality or sub modules while keeping the base functionality intact. There are four major sub modules at work here: A frame buffer for storing and displaying images on the VGA output, a level creator responsible for generating the background image and in-game world, a sprite subsystem for creating and controlling the dynamic items and enemies the player can interact with, and the player

controller for relaying our input signals to the different logics in our system. All of this functionality is abstracted away from the rest of the project, allowing Kevin to focus on implementation of his modules.

2.2.2 Frame Buffer

Although fairly simple in purpose, the frame buffer is perhaps the central module of the entire game logic module. Because our game calls for dynamically updating multiple different images on the screen, a frame buffer provides a more elegant solution as opposed to combinational logic. In addition, once the information is stored in the frame buffer, other modules can poll it to perform their necessary calculations. If this were to be attempted in combinational logic, other modules would be limited to performing their calculations within 15 nanoseconds (corresponding to the 65 MHz clock that drives the vcount and hcount XVGA signals).

Because we already used one of the ZBTs for the video capture module, we decided to implement our frame buffer in BRAM. Normally the amount of available BRAM would limit us, especially when trying to display a full XVGA image. However, we internally perform all graphical operations at the native resolution used by the NES. This allows us to easily store 256x240 pixels worth of data in our buffer. Each pixel stores not only 6 bits of color information, but also 7 bits of "pixel metadata" that are used for calculations such as collisions detection. Using a BRAM also allows us to avoid having to deal with the extra timing constraints imposed by the two clock delay in each ZBT access.

In order to take this low-resolution image and display an XVGA (1024x768) image, we scale each pixel by updating at specific intervals as opposed to each increment of hcount and vcount. To maximize use of screen real estate, each pixel is scaled up by 4 in the horizontal direction. In the vertical direction, we can scale each pixel up by 3 without causing over scan. In effect, each low resolution pixel is represented on screen by 4 x 3 physical pixels. To achieve this, we address our buffer by a bit shifted hcount (every 4 pixels) and every three vcount.

Our frame buffer is currently accessed by both the sprite generator and level generator modules. To avoid potential glitching of signals, our buffer is created as a dual port BRAM. Port A is shared by both the Sprite Generator module as well as the hcount and vcount signals. To avoid glitching we require vcount to be greater than 720 in order for the sprite generator to write to the buffer, as this insures that the image has already been read out to the VGA output. Port B is utilized by the Level Creator module for drawing the in-game world and background.

2.2.3 Level Creator

The level creator is responsible for generating the world the player interacts with, as well as properly scrolling this world according to the player's progress. This is one of the key portions of the game engine. Key game mechanics are handled by these modules of the game engine, while specific levels and tilesets can be swapped out by replacing the relevant level and tile ROMs. The game world is comprised of 16x16 pixel graphical tiles (the details of which are described in the next section), and each screen is comprised of 16x16 of these tiles. Specific combinations of tiles generate the world the player interacts with. Using a tile based method simplifies the creation of levels for our game, as complicated

backgrounds can be created from individual tiles, as opposed to having to code our backgrounds pixel by pixel. The level creator also contains logic to control the scrolling of the game world as the player navigates it. While our level is composed of discreet 16x16 tiles, we need to be able to scroll by increments smaller than 16 in order to maintain smooth video output and gameplay. Internally, we store a 12 bit coordinate called *left_edge,* which contains the current coordinate of the pixels on the left edge of the screen.

The basic operation of the level creator is as follows. Once the vcount is greater than 720 and we know that the last frame has already been displayed on our video output, we can assert *back_we* and write to the frame buffer. The level creator then generates a complete frame by polling both the Level and Tile ROMs. The level ROM specifies which tile is to be at any given row or column coordinate, and the Tile ROM then gives us further information about the pixel content of each tile.

Because the Level Creator accesses three different memories for each pixel, we had to pipeline the circuitry in order to meet timing constraints. By adding registers to store previous outputs, we can poll each of the memories simultaneously. At the cost of a little latency, we are now able to process a pixel on every clock cycle, rather than waiting three clock cycles to retrieve information from our three memories. Had we not implemented this, we would be unable to generate the entire background and output to the frame buffer in between each screen refresh.

In addition to generating both background and foreground elements, our level creator must also notify the sprite subsystem when a new sprite has entered the screen. Because sprites are dynamic objects, they cannot be rendered using our static level creator. As such a ***new_sprite*** signal is sent to the sprite generator, which will then handle the creation and control of the sprite. In such a case, the background generator will specify a generic background tile in place of the sprite. This is necessary, as our Level ROM can only specify one element per coordinate.

2.2.3.1 Tile Rom

To simplify our calculations and graphic generation, it is convenient to consider each image as a distinct entity rather than the individual pixels that compose it. In the original Nintendo Entertainment System, sprites could be one of two sizes, 16x16 or 16x32. In our 6.111 implementation, we went with discreet tiles of 16x16 pixels. Each tile can be treated as a complete entity unto itself.

First, the tile Rom is implemented as a dual port BRAM that is 10 bits wide and 1024 addresses deep. For each pixel in a tile, we have 10 bits of information. The first 6 bits correspond to the color information of the pixel when it is displayed, two bits each for red green and blue. The next three bits store information regarding the pixel's attributes. Currently this information is used primarily for collisions detection and enemy behaviors, but by expanding the number of data bits, we can easily provide for more intricate behaviors and internal game events. Finally the last bit determines whether or not the pixel should be treated as transparent. This allows us to draw more intricate sprites while still retaining visibility of the background.

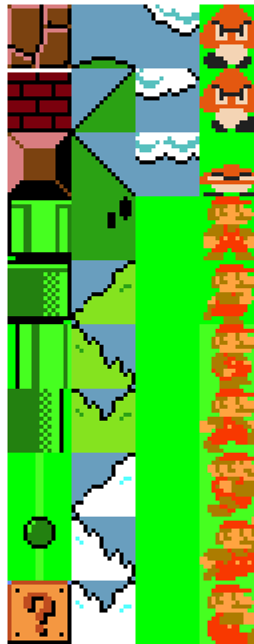

Figure 5: Tiles Used in Game

To create these tiles, a MATLAB script was used to convert the pixel information in the original images to 6-bit RGB values. These were stored in a .coe file that the Xilinx Coregen application can use to preload onto a BRAM.

2.2.3.2 Level Rom

The Level Rom is a 256 x 16 x 16 ROM that indexes all of the possible tiles that make up the game world. Each screen in the world is composed of 16x16 tiles (with 16 tiles visible in the horizontal direction, and 15 tiles in the vertical direction). This ROM is then addressed by a zero indexed row and column starting in the top left corner of the screen. Because this ROM is only utilized by our level creator, we implemented it as a single port block ram.

2.2.4 Game FSM

The Game FSM is responsible for performing necessary calculations in between frame refreshes. Chief among them is the rudimentary gravity that affects all dynamic objects in the game. Between each frame, all sprites (including the player) are lowered by one pixel. This creates a constant gravity inside the game world. If this action causes a sprite to clip into the ground, the sprite generator will quickly correct for this, as it operates at a much faster refresh rate (at the positive edge of each 65MHz clock cycle).

The Game FSM is also responsible for controlling the independent behavior of all of the sprites. Currently logic is only implemented for the basic mushroom enemy, or Goomba. Between each refresh, the FSM will update the x and y coordinates of the sprite, its current state as well as the tile to use. By changing the tile used by the sprite, we can implement simple animations. For example, the Goomba enemy's walking animation simply involves switching between two different sprites back and forth.

If the rest signal is asserted, the FSM enters into RESET, in which it cycles through three different subtates in order to increment the sprite number and initialize all the values for each sprite in the sprite RAM.

2.2.5 Sprite RAM

As sprites are dynamic entities, information about their last known location and state must be stored in some central location. This allows the Sprite Generator module to draw sprites in the correct location, as well as the Game FSM to accurately control the behavior of each sprite. Our Sprite RAM is implemented as a 36 bit wide RAM that is 16 deep. As such we can handle up to 16 sprites onscreen at any given moment. Given that the original Super Mario Bros had an 8 sprite limitation, we felt that 16 sprites provided ample capacity for our side scrolling game.

The Sprite RAM is another dual port BRAM, as information about each sprite is updated by both the Sprite Generator and Game FSM modules. As stated above, the information is stored in 36 bits. The first 20 bits correspond to the x and y coordinates of sprite in the game. The remaining sprites are left to store state information for each sprite. Five bits store the tile index number of the current sprite, allowing us to animate each sprite by modifying this value. Sprites are held in the upper half of our tile ROM, so we can create a proper tile index by adding a 1 to the front of this 5 bit tile index. The next 3 bits store information about the last collision the sprite had with the player. We have 4 bits that are currently unused, allowing us to store extra information about sprites should we expand on the functionality of the game. The final 4 bits are used to store directional information about the sprite's movement.

2.2.6 Sprite Generator

The Sprite Generator is responsible for updating the graphics of the dynamic sprites in the frame buffer. To avoid glitching of the image stored in the frame buffer, we wait until vcount == 780. By this time the Level Creator should have generated the background image for the next frame and we should be able to then overlay our sprites. By writing to the buffer in this manner we can be sure that the background is consistent and that our sprites are always visible, as the last written value to the buffer is what is read by the video output logic for displaying an image.

Once vcount == 780, the sprite generator cycles through each sprite in RAM to obtain it's tile index. Using this tile index the sprite generator can read from the Tile ROM and get the necessary color and behavior data for the current pixel being written. Once this information has been obtained, the sprite generator must check the write location in order to avoid collisions or clipping of the sprites with the game world or other dynamic entities. To perform this check, the Sprite Generator retrieves pixel behavior data from the Frame Buffer and the Tile ROM simultaneously. By comparing the values of the 3 bit behavioral pixel data, we can determine the type of collision that has occurred at any given pixel. The table below lists the currently implemented behavioral types.

Table 1: Behavioral Pixel Values & Descriptions

Pixel Value	Description
000	No collision
001	Stops movement in the left direction
010	Stops movement in the right direction
011	Stops vertical movement
100	Hazard pixel, kills player
101	Weak pixel, player kills sprite
110	Event pixel, will modify player state (unimplemented)
111	Win pixel, player wins and game halts

2.2.7 Player Controller

The Player Controller module takes our video display signals from the XVGA module, as well as the gestural input signals from the Camera Capture & Gesture Recognition module. These gestural input signals currently correspond to the player's movement in the game world, requests to move forward or to jump. Provisions exist to allow backward movement and crouching, but these behaviors were not implemented in the gesture module at time of demonstration. Based on the input signals received, the player controller updates the x and y coordinates of the player.

Time permitting, the Player Controller would also provide additional functionality. The player x and y coordinates are actually output to the top level project module. One proposed functionality was a two player competitive mode, in which two players could race each other by playing on two different labkits. In this scenario, the x and y coordinates on one labkit would be sent to the other labkit, allowing the game to display the location of the opponent on the local screen, providing progress feedback and extra incentive for the player to beat the game.

3 Insights and Observations

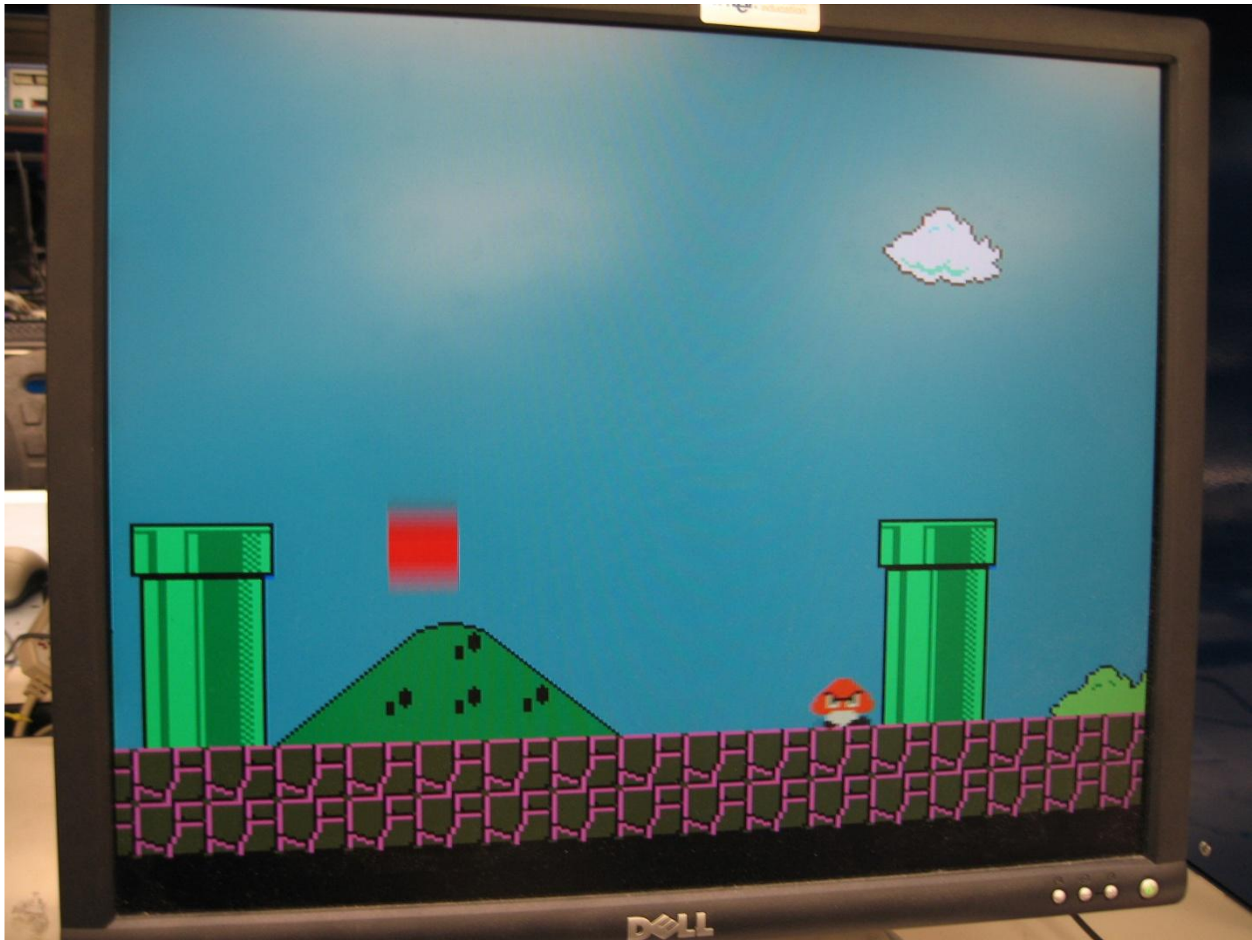

Figure 6: Photo of game in action.

3.1 Testing & Debugging

3.1.1 Video Logic Module

The first simple test of the system was to see if I could display on the monitor the incoming video stream. This was an important milestone early on in the project because it meant that I was instantiating the video camera modules correctly, storing the correct RGB pixel data in the ZBT, reading correctly from the ZBT, and getting correct RGB values for the pixels. With the foundation of the video capture module now working, the RGB2HSV module had to be tested to see if it could correctly calculate the Hue from the RGB values.

To test the RGB2HSV, I first programmed in two simple test cases in Verilog so that on the hex display I could just see the answer I was getting from the module. I was able to hand calculate the answer I should have received since I knew exactly what values for R, G, and B were being input into the module, and my calculations matched those of the module. With the HSV conversion working, I needed a way to determine the Hue value for the different markers and colors I was planning to use, so I created the Crosshairs module and the HSV_Find module.

The Crosshairs module simply creates a large crosshair in the frame by drawing a white line in both the x and y directions to mark where the current x and y coordinates of the crosshairs were with the intersection of the two lines. The lines were drawn using the draw_line_x and draw_line_y modules which output a pixel_enable signal that signals the display to make a white pixel if the hcount is equal to the x coordinate of the crosshairs or if the vcount is equal to the y coordinate of the crosshairs. These crosshairs were movable in all four directions by using the up, down, left, and right directional buttons on the labkit. The HSV_Find module would take in the x and y coordinates of the crosshairs and output the calculated HSV conversion of the RGB pixel at that coordinate. A circular buffer of the current value and previous seven values was used to average the Hue value. The R, G, and B values would each be output using two hex values on the left side of the hex display and the bottom 9 numbers on the hex display were the binary values for each of the 9 bits in the Hue value. With these modules working together, it was possible to scroll around the screen and see what the Hue value was for any pixel as well the R, G, and B values for that same pixel. This became invaluable when it came to making thresholds for the hat and arm markers.

One thing that I noticed right away was that the Hue values for the same object could vary by over ten or so when measured under varying conditions of light, distance from the camera, different orientations of the marker, etc. I first calculated my thresholds with the markers about a foot away from the camera, but noticed that these thresholds would not work for the same markers when there was a lot of light or glare or when the objects were moved further from the camera by even a foot. Every day it was necessary to calculate new thresholds for the different colors because the conditions in the lab would be slightly different. As part of the solution, I picked a spot where the player would be standing on the floor near a wall by my workstation and picked a spot where the camera would be. The distance between the markers and the camera was now set and the lighting did not vary in that spot since it was lit only by the lab lights and not affected by any external light. After doing this, the Hue values seemed to remain more consistent.

With the HSV conversion working and the thresholds set, I tested the Center of Mass module by add 2 more pairs of crosshairs. One pair of crosshairs would identify the center of mass for the red hat and the other the center of mass for the dominant arm marker present. The Center of Mass module did not need any retooling besides changes in the threshold values once in a while.

The instantiation of the Center of Mass module revealed the biggest challenge. Despite getting rid of the effect of noisy pixels in the Center of Mass module, it was revealed that the real world is very noisy itself. For example, if I covered the camera lens with a black item or my hand, the centers of mass would stop jumping around and become static, stuck at the last average center of mass. This showed that the random noisy pixels due to the system itself were not affecting the centers of mass since you could still see on the monitor many random red, green, and blue specs, but the centers of mass were not changing with the static black background. However, upon removing the black item or your hand, the two centers of mass jumped around very erratically due to real world noise. The best solution I could think of to the problem was to put up a large black sheet on the wall used for testing, and a lot of black clothing. To minimize any noise from the world, the background was black, my clothes were black, everything was black save for the arm markers and the red hat. This was not an end all solution though

because there was still glare from the lights and things like that which would cause the centers of mass to jump around in the absence of the hat or arm markers. However, when the red hat was introduced into the environment, the hat center of mass was very accurate. I noticed that many lighter skin tones would add to the center of mass, so I wore a black sweatshirt and gloves, leaving only my face showing. My face added to the center of mass, but it made it more accurate and less jumpy. A new problem was that the green and yellow markers did not work well. The yellow would come up as white (I was still using the crosshairs and hex display on the labkit to check the Hue value) at that distance from the camera which was mostly due to the light reflecting off of it. The green was just a bad color it seemed since it was very hard for the green color to be recognized and used toward the arm center of mass. Other options I tried were orange and purple, but orange was too close to red to be used and the purple was too dark and close to black. In my search for two colors that would work for arm markers, I was only able to find one replacement, a strip of light blue cardboard that I tore off of a box. The blue worked very well actually and I was able to find this suitable replacement in time to test my action and runorwalk modules.

With the centers of mass being detected, the final parts of the gesture detection module were tested. The line calculator worked will with the default values in drawing a jump line and a crouch line above and below the center of mass of the hat, and the action module was able to detect a jump, stand, or crouch. The runorwalk module also worked since the movement signal would be asserted if the play pumped their arm as if they were walking. The action and movement signals were displayed on the hex display for the purposes of testing. The jumping and crouching worked very well. The runorwalk module was a little more sensitive since the player angling their arm could change the Hue value enough for the arm to be unrecognized. Increasing the threshold range was tried, but made the center of mass less accurate. For the purpose of basic functionality, all of the modules worked, many of them very well, after rigorous testing and debugging.

3.1.2 Game Logic Module

To debug the game module, we simply had to play the game and experiment with different test cases and see how the game reacted. Before compilation, I ran into many issues regarding redeclarations and other simple code mistakes. Because of the design of the Game Subsystem, it had to be tested as a whole, as opposed to being able to test individual components separately. While debugging the game, I output the x and y coordinates to the built in fluorescent display on the labkit. This was invaluable during the early stages of the design implementation, as I was unable to draw a player sprite on screen. Even when the graphics for the player were not working, using the coordinate data I was able to somewhat navigate the game levels and determine whether or not the game was behaving as expected.

Luckily, by virtue of performing all of our calculations using the low-resolution image, we were able to save on chip utilization. In consulting with the instructors, we learned that high levels of chip utilization and the fact that the compilation of the code is nondeterministic lead to various weird

glitches and errors. For the most part we were able to avoid these sorts of errors throughout the testing process.

4 Conclusion

Completed, our project demonstrated a motion capture based gesture recognition system combined with a classic 2D side scrolling platform game. Our video capture system processed the incoming video stream to provide center of mass and coordinate data for our colored markers. By determining the speed of the swinging of the arms, or if the player's head crossed calibrated thresholds, we were able to interpret the player's physical actions of walking and jumping and send appropriate control signals. These control signals were then hooked up to the game logic module, which generated the visual representation of the game world and provided feedback for the player's actions. As the player moved in the real world, he could see his correspondingly move in the game world.

Holistically, the project presented an interesting challenge as well as an invaluable learning experience. We were able to meet the majority of the goals we set for ourselves, with the remaining functionality only limited by the amount of time we were able to spend in lab. As things got more complex, the increased compile time limited how quickly we could experiment with different implementations or functionality. Similarly, although our base functionality was met, we could have presented a lot more polish and calibration provided more time. Both modules were being worked on up until the 11th hour, and by some stroke of luck they worked seamlessly together after a single compile.

Ultimately the project working on the project was an intense but fulfilling experience. Late nights in lab staring at hundreds of lines of verilog were apt to cause heightened emotions at times, but it was well worth the payoff of seeing your compiled code running on actual hardware.

Appendix

Appendix I – zbt_6111_sample.v

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

//`include "display_16hex.v"
//`include "debounce.v"
//`include "video_decoder.v"
//`include "zbt_6111.v"
//`include "ntsc2zbt.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 zbt_6111_sample(beep, audio_reset_b,
                       ac97_sdata_out, ac97_sdata_in, ac97_synch,
               ac97_bit_clock,

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

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
               tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

               tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
               tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
               tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
               tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

               ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
               ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

               ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
               ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

               clock_feedback_out, clock_feedback_in,

               flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
               flash_reset_b, flash_sts, flash_byte_b,

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

               disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_in,

               button0, button1, button2, button3, button_enter, button_right,
               button_left, button_down, button_up,

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

               systemace_data, systemace_address, systemace_ce_b,
               systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

               analyzer1_data, analyzer1_clock,
               analyzer2_data, analyzer2_clock,
               analyzer3_data, analyzer3_clock,
```

```
             analyzer4_data, analyzer4_clock);

output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

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

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
        tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
        tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
        tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
        tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

input  button0, button1, button2, button3, button_enter, button_right,
        button_left, button_down, button_up;
input  [7:0] switch;
output [7:0] led;

inout [31:0] user1, user2, user3, user4;

inout [43:0] daughtercard;

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

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
```

```
/*
*/
   // ac97_sdata_in is an input

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

   // Video Input
   //assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b1;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b1;
   //assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = clock_27mhz;//1'b0;
   //assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs

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

/*
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;// clock enable
*/

/* enable RAM pins */

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

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

   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;

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

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

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

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

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

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

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

   // Logic Analyzer
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

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

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

   wire clk = clock_65mhz;

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

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

         //debounce buttons
         wire clean_button_left, clean_button_right, clean_button_up, clean_button_down, clean_button0;
         wire clean_button1, clean_button2, clean_button3;

         debounce debounce1(reset, clk, ~button_left, clean_button_left);
         debounce debounce2(reset, clk, ~button_right, clean_button_right);
         debounce debounce3(reset, clk, ~button_down, clean_button_down);
         debounce debounce4(reset, clk, ~button_up, clean_button_up);
         debounce debounce5(reset, clk, ~button0, clean_button0);
         debounce debounce6(reset, clk, ~button1, clean_button1);
```

```
        debounce debounce7(reset, clk, ~button2, clean_button2);
        debounce debounce8(reset, clk, ~button3, clean_button3);

   // display module for debugging

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

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

   // wire up to ZBT ram

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

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

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

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

   // ADV7185 NTSC decoder interface code
   // adv7185 initialization module
   adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                       .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                       .tv_in_i2c_clock(tv_in_i2c_clock),
                       .tv_in_i2c_data(tv_in_i2c_data));

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

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

   // code to write NTSC data to video memory

   wire [18:0] ntsc_addr;
   wire [35:0] ntsc_data;
   wire        ntsc_we;

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

        YCrCb2RGB ycrcb2rgb(R, G, B, clk, reset, ycrcb[29:20], ycrcb[19:10], ycrcb[9:0]);


   ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, {R[7:2], G[7:2], B[7:2]},
                    ntsc_addr, ntsc_data, ntsc_we, 1'b0);

//      reg [10:0] crosshairs_x_in;
//      reg [9:0] crosshairs_y_in;
//
//      wire [10:0] new_crosshairs_x;
//      wire [9:0] new_crosshairs_y;
//
//      always @(posedge clk) begin
//              if (reset) begin
//                      crosshairs_x_in <= 11'd400;
//                      crosshairs_y_in <= 10'd400;
//              end
//              else begin
//                      crosshairs_x_in <= new_crosshairs_x;
//                      crosshairs_y_in <= new_crosshairs_y;
//              end
```

```
//        end
//
//        // instantiation of the crosshairs which can be moved by pressing up,down,left,right
//
//        Crosshairs crosshairs(hcount, vcount, crosshairs_x_in, crosshairs_y_in, clean_button_up, clean_button_down,
//                                                                             clean_button_left, clean_button_right, clk,
reset,
//                                                                             new_crosshairs_x, new_crosshairs_y);


//
         wire pixel_enable[7:0];

//        draw_line_y crosshair_y(crosshairs_y_in, vcount, pixel_enable0);
//        draw_line_x crosshair_x(crosshairs_x_in, hcount, pixel_enable1);

//
//        wire [7:0] R_out, G_out, B_out;
//
//        RGB_Find rgb(crosshair_x_in, crosshair_y_in, hcount, vcount, R, G, B,
//                                        clk, reset, R_out, G_out, B_out);

// test if the HSV conversion works

//wire [8:0] H_test;
//wire [7:0] S_test, V_test;
//
//wire [7:0] R_test, G_test, B_test;
//
//assign R_test = 8'd255;
//assign G_test = 8'd100;
//assign B_test = 8'd0;

//RGB2HSV rgb2hsv( R_test, G_test, B_test, clk, rst, H_test, S_test, V_test);

// answer should be 23 and it is

// test to find the H S V values of colors to set color thresholds

//        wire [8:0] H_out;
//        wire [7:0] S_out, V_out;
//        wire [7:0] R_out, G_out, B_out;
//
//        HSV_Find hsv(crosshairs_x_in, crosshairs_y_in, hcount, vcount, {vr_pixel[17:12], 2'b0},
//                                        {vr_pixel[11:6], 2'b0}, {vr_pixel[5:0], 2'b0},
//                                        clk, reset, R_out, G_out, B_out, H_out, S_out, V_out);

//         conversion to HSV to be used in center of mass calculations
//
         wire [8:0] H1, H2;
         wire [7:0] S1, S2, V1, V2;

         RGB2HSV RGB2HSV1({ntsc_data[35:30],2'b00}, {ntsc_data[29:24],2'b00}, {ntsc_data[23:18],2'b00},
                                                        clk, reset, H1, S1, V1);
         RGB2HSV RGB2HSV2({ntsc_data[17:12],2'b00}, {ntsc_data[11:6],2'b00}, {ntsc_data[5:0],2'b00},
                                                        clk, reset, H2, S2, V2);

// need to delay the ntsc_addr since the center of mass needs the correct address
// with the HSV values to determine position. the RGB2HSV has a delay 20 cycles

         reg [18:0] ntsc_addr_delay[20:0];

         always @(posedge clk) begin
                  ntsc_addr_delay[0] <= ntsc_addr;
                  ntsc_addr_delay[1] <= ntsc_addr_delay[0];
                  ntsc_addr_delay[2] <= ntsc_addr_delay[1];
                  ntsc_addr_delay[3] <= ntsc_addr_delay[2];
                  ntsc_addr_delay[4] <= ntsc_addr_delay[3];
                  ntsc_addr_delay[5] <= ntsc_addr_delay[4];
                  ntsc_addr_delay[6] <= ntsc_addr_delay[5];
                  ntsc_addr_delay[7] <= ntsc_addr_delay[6];
                  ntsc_addr_delay[8] <= ntsc_addr_delay[7];
                  ntsc_addr_delay[9] <= ntsc_addr_delay[8];
                  ntsc_addr_delay[10] <= ntsc_addr_delay[9];
                  ntsc_addr_delay[11] <= ntsc_addr_delay[10];
                  ntsc_addr_delay[12] <= ntsc_addr_delay[11];
                  ntsc_addr_delay[13] <= ntsc_addr_delay[12];
                  ntsc_addr_delay[14] <= ntsc_addr_delay[13];
                  ntsc_addr_delay[15] <= ntsc_addr_delay[14];
```

```
                ntsc_addr_delay[16] <= ntsc_addr_delay[15];
                ntsc_addr_delay[17] <= ntsc_addr_delay[16];
                ntsc_addr_delay[18] <= ntsc_addr_delay[17];
                ntsc_addr_delay[19] <= ntsc_addr_delay[18];
                ntsc_addr_delay[20] <= ntsc_addr_delay[19];
        end

        wire [8:0]hi_hat_threshold, low_hat_threshold, hi_left_arm_threshold, low_left_arm_threshold,
                  hi_right_arm_threshold, low_right_arm_threshold;

        assign hi_hat_threshold = 9'b101100111; // red 359
        assign low_hat_threshold = 9'b101100011; // red 355

        assign hi_left_arm_threshold = 9'b001110011; // green 115
        assign low_left_arm_threshold = 9'b001101110; // green 110

        assign hi_right_arm_threshold = 9'b010111100; // blue 188
        assign low_right_arm_threshold = 9'b010111000; // blue 184

        wire [10:0] hatx, armx;
        wire [9:0] haty, army;
   wire comvalid;

   Center_of_Mass com(ntsc_addr_delay[20], ntsc_we, H1, S1, V1, H2, S2, V2, hi_hat_threshold,
                                                            low_hat_threshold, hi_left_arm_threshold,
low_left_arm_threshold,
                                                            hi_right_arm_threshold, low_right_arm_threshold, clk,
reset,
                                                            hatx, haty, armx, army, comvalid);

        draw_line_y haty_line(haty, vcount, pixel_enable[2]);
        draw_line_x hatx_line(hatx, hcount, pixel_enable[3]);

        draw_line_y army_line(army, vcount, pixel_enable[4]);
        draw_line_x armx_line(armx, hcount, pixel_enable[5]);

        wire [9:0] jumpliney, crouchliney;

        line_calculator lines(clk, reset, comvalid, clean_button0, haty, jumpliney, crouchliney);

        draw_line_y jump_line(jumpliney, vcount, pixel_enable[6]);
        draw_line_y crouch_line(crouchliney, vcount, pixel_enable[7]);

   reg [1:0] action;
        wire [1:0] action_value;
   wire [1:0] movement;

        always @(posedge clk) begin
                if (reset) action <= 2'b01;
                else action <= action_value;
        end

        action actions(jumpliney, crouchliney, haty, clk, reset, action_value);

   runorwalk walk(hcount, vcount, armx, comvalid, clk, reset, movement);
/////////////////////////////////////////////////////////////////////////////////////////
// Hooking up the game_wrapper module to Kevin's top level labkit.

        wire up, down, left, right;
        assign up = (clean_button_up | (action == 2));
        assign down = (clean_button_down | (action == 0));
        assign right = (clean_button_right | movement);
        assign left = clean_button_left;


   // Game module instantiation
        wire game_over;
        wire [23:0] pixel_game;
        wire [10:0] player_xcoord;
        wire [9:0] player_ycoord;
        game_engine gm(.vclock(clock_65mhz),
                                        .reset(reset),
                                        .up(up),
                                        .down(down),
                                        .left(left),
                                        .right(right),
                                        .hsync(hsync),
                                        .vsync(vsync),
                                        .blank(blank),
```

```
                                        .hcount(hcount),
                                        .vcount(vcount),
                                        .pixel(pixel_game),
                                        .player_xcoord(player_xcoord),
                                        .player_ycoord(player_ycoord),
                                        .game_over(game_over));
   // Are we looking at the player sprite
         wire mario_calc;
         assign mario_calc = (((hcount >= player_xcoord-32) && (hcount < (player_xcoord + 32)))
                                                            && ((vcount >= player_ycoord) && (vcount <
(player_ycoord+48))));

         wire [23:0] pixel_mario;
         blob cheap_mario(.x(player_xcoord),
                                                            .y(player_ycoord),
                                                            .hcount(hcount),
                                                            .vcount(vcount),
                                                            .pixel(pixel_mario));

// End of Douglas's interconnects
/////////////////////////////////////////////////////////////////////////////////////////

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

   wire [18:0]     vram_addr2 = count[0+18:0];
   wire [35:0]     vpat = ( switch[1] ? {4{count[3+3:3],4'b0}}
                          : {4{count[3+4:4],4'b0}} );

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

   wire   sw_ntsc = ~switch[7];
   wire   my_we = sw_ntsc ? (hcount[0]== 1) : blank;
   wire [18:0]     write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
   wire [35:0]     write_data = sw_ntsc ? ntsc_data : vpat;

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

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

   // select output pixel data
//       reg [23:0]        pixel;
//   reg  b,hs,vs;
   reg [23:0]      pixel;
   wire  b,hs,vs;

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

         // select output pixel data
//
//       always @(posedge clock_65mhz) begin
//                if (switch[1:0] == 2'b01) begin
//                         // 1 pixel outline of visible area (white)
//                         hs <= hsync;
//                         vs <= vsync;
//                         b <= blank;
//                         pixel <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7 : 0;
//                end else if (switch[1:0] == 2'b10) begin
//                         // color bars
//                         hs <= hsync;
//                         vs <= vsync;
//                         b <= blank;
//                         pixel <=  ( pixel_enable[2] | pixel_enable[3] |        pixel_enable[4] | pixel_enable[5] |
pixel_enable[6] | pixel_enable[7]) ? 24'b1111_1111_1111_1111_1111_1111 : {vr_pixel[17:12], 2'b0, vr_pixel[11:6], 2'b0,
vr_pixel[5:0], 2'b0};
//      end else begin
//                         // default: Mario
//                         hs <= hsync;
//                         vs <= vsync;
//                         b <= blank;
//                         // pixel <= pixel_game;
//                         pixel <= mario_calc ? pixel_mario : pixel_game ;
```

```
//      end
//   end

   always @(posedge clk)
     begin
          pixel <= switch[0] ? {hcount[8:6],5'b0,hcount[8:6],5'b0,hcount[8:6],5'b0} :
                                    ( //pixel_enable0 | pixel_enable1 |
                                    pixel_enable[2] | pixel_enable[3] |
                                     pixel_enable[4] | pixel_enable[5] |
                                     pixel_enable[6] | pixel_enable[7]) ? 24'b1111_1111_1111_1111_1111_1111 :
                                    switch[3] ? (mario_calc ? pixel_mario : pixel_game ):
                                    {vr_pixel[17:12], 2'b0, vr_pixel[11:6], 2'b0, vr_pixel[5:0], 2'b0};
     end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = pixel[23:16];
   assign vga_out_green = pixel[15:8];
   assign vga_out_blue = pixel[7:0];
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

   // debugging

   assign led = ~{vram_addr[18:13],reset,switch[0]};

   always @(posedge clk)
     // dispdata <= {vram_read_data,9'b0,vram_addr};
     // dispdata <= {2'd0, R_out[7:6], 2'd0, R_out[5:4], 2'd0, R_out[3:2], 2'd0, R_out[1:0],
          //                                          8'd0,
          //                                          2'd0, G_out[7:6], 2'd0, G_out[5:4], 2'd0, G_out[3:2], 2'd0,
G_out[1:0],
          //                                          8'd0,
          //                                          2'd0, B_out[7:6], 2'd0, B_out[5:4], 2'd0, B_out[3:2], 2'd0,
B_out[1:0]};
//                dispdata <= {28'd0, 3'd0, H_out[8],
//                                                     3'd0, H_out[7], 3'd0, H_out[6], 3'd0, H_out[5], 3'd0, H_out[4],
//                                                     3'd0, H_out[3], 3'd0, H_out[2], 3'd0, H_out[1], 3'd0, H_out[0]};

//                dispdata <= {R_out, G_out, B_out,
//                                                     4'b0,
//                                                     3'd0, H_out[8],
//                                                     3'd0, H_out[7], 3'd0, H_out[6], 3'd0, H_out[5], 3'd0, H_out[4],
//                                                     3'd0, H_out[3], 3'd0, H_out[2], 3'd0, H_out[1], 3'd0, H_out[0]};

                  dispdata <= {32'b0,
                                                       2'b0, movement,
                                                       24'b0,
                                                       2'b0, action};

endmodule

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

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

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

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

   // vertical: 806 lines total
```

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

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

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

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

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

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

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

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

   wire                      hc2 = hcount[0];
   reg [17:0]       vr_pixel;
   reg [35:0]       vr_data_latched;
   reg [35:0]       last_vr_data;

   always @(posedge clk)
     last_vr_data <= (hc2) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (!hc2) ? vram_read_data : vr_data_latched;

   always @*                // each 36-bit word from RAM is decoded to 4 bytes
     case (hc2)
       1'b1: vr_pixel = last_vr_data[17:0];
       1'b0: vr_pixel = last_vr_data[35:18];
     endcase

endmodule // vram_display

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

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

   parameter NDELAY = 3;

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

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

```
endmodule // delayN

////////////////////////////////////////////////////////////////////////////////
//
// Cheap Mario Overlay
//
////////////////////////////////////////////////////////////////////////////////
module blob (input [10:0] x,hcount,
                                       input [9:0] y,vcount,
                                       output reg [23:0] pixel);

   always @ (x or y or hcount or vcount) begin
      if ((hcount >= (x-32) && hcount < (x+32)) &&
          (vcount >= (y) && vcount < (y+48)))
         pixel = 24'b111111110000000000000000;
      else pixel = 0;
   end
endmodule
```

Appendix II – Video Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    20:34:51 12/07/2009
// Design Name:
// Module Name:    draw_line_x
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module draw_line_x(line_x, hcount, pixel);

input [10:0] line_x, hcount;
output reg pixel;

always @* begin
         if ((hcount >= (line_x - 2)) & (hcount <= (line_x + 2)))
                  pixel <= 1;
         else pixel <= 0;
end

endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    20:24:45 12/07/2009
// Design Name:
// Module Name:    draw_line_y
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module draw_line_y(line_y, vcount, pixel);
```

```
input [9:0] line_y, vcount;
output reg pixel;

always @* begin
        if ((vcount >= (line_y - 2)) & (vcount <= (line_y + 2)))
                pixel <= 1;
        else pixel <= 0;
end

endmodule
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:25:10 12/07/2009
// Design Name:
// Module Name:    Crosshairs
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module Crosshairs(hcount, vcount, x_in, y_in, button_up, button_down, button_left, button_right,
                                                        clk, reset, x_out, y_out);

input [10:0] x_in, hcount;
input [9:0] y_in, vcount;
input button_up, button_down, button_left, button_right, clk, reset;

output reg [10:0] x_out;
output reg [9:0] y_out;

always @(posedge clk) begin
        if (reset) begin
                x_out <= 11'd400;
                y_out <= 10'd400;
        end
        if ((hcount == 1023) & (vcount ==765)) begin
                if (button_up & (y_in >= 0)) y_out <= y_in - 1;
                else if (button_down & (y_in <= 765)) y_out <= y_in + 1;
                else if (button_left & (x_in >= 0)) x_out <= x_in - 1;
                else if (button_right & (x_in <= 1023)) x_out <= x_in + 1;
        end
end


endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:49:48 12/07/2009
// Design Name:
// Module Name:    HSV_Find
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module HSV_Find(crosshair_x, crosshair_y, hcount, vcount, R_in, G_in, B_in, clk, reset,
                                                R_out, G_out, B_out, H, S, V);
```

```
input [10:0] crosshair_x, hcount;
input [9:0] crosshair_y, vcount;
input [7:0] R_in, G_in, B_in;
input clk, reset;

output reg [7:0] R_out, G_out, B_out;

output reg [8:0] H;
output reg [7:0] S, V;

reg [7:0] R[7:0], G[7:0], B[7:0];
reg [10:0] sum_R, sum_G, sum_B;
reg [7:0] R_average, G_average, B_average;

wire [8:0] H1;
wire [7:0] S1, V1;

RGB2HSV RGB2HSV1(R_average, G_average, B_average, clk, reset, H1, S1, V1);

always @(posedge clk) begin
	if (reset) begin
		H <= 9'd0;
		S <= 8'd0;
		V <= 8'd0;

		R[0] <= 8'd0;
		R[1] <= 8'd0;
		R[2] <= 8'd0;
		R[3] <= 8'd0;
		R[4] <= 8'd0;
		R[5] <= 8'd0;
		R[6] <= 8'd0;
		R[7] <= 8'd0;

		G[0] <= 8'd0;
		G[1] <= 8'd0;
		G[2] <= 8'd0;
		G[3] <= 8'd0;
		G[4] <= 8'd0;
		G[5] <= 8'd0;
		G[6] <= 8'd0;
		G[7] <= 8'd0;

		B[0] <= 8'd0;
		B[1] <= 8'd0;
		B[2] <= 8'd0;
		B[3] <= 8'd0;
		B[4] <= 8'd0;
		B[5] <= 8'd0;
		B[6] <= 8'd0;
		B[7] <= 8'd0;
	end
	else if ((hcount == crosshair_x) & (vcount == crosshair_y)) begin
		R[0] <= R_in;
		R[1] <= R[0];
		R[2] <= R[1];
		R[3] <= R[2];
		R[4] <= R[3];
		R[5] <= R[4];
		R[6] <= R[5];
		R[7] <= R[6];

		G[0] <= G_in;
		G[1] <= G[0];
		G[2] <= G[1];
		G[3] <= G[2];
		G[4] <= G[3];
		G[5] <= G[4];
		G[6] <= G[5];
		G[7] <= G[6];

		B[0] <= B_in;
		B[1] <= B[0];
		B[2] <= B[1];
		B[3] <= B[2];
		B[4] <= B[3];
		B[5] <= B[4];
		B[6] <= B[5];
		B[7] <= B[6];
```

```
                    sum_R <= (R[0] + R[1] + R[2] + R[3] + R[4] + R[5] + R[6] + R[7]) & 11'b111_1111_1111;
                    sum_G <= (G[0] + G[1] + G[2] + G[3] + G[4] + G[5] + G[6] + G[7]) & 11'b111_1111_1111;
                    sum_B <= (B[0] + B[1] + B[2] + B[3] + B[4] + B[5] + B[6] + B[7]) & 11'b111_1111_1111;

                    R_average <= sum_R[10:3];
                    G_average <= sum_G[10:3];
                    B_average <= sum_B[10:3];

                    R_out <= R_in;
                    G_out <= G_in;
                    B_out <= B_in;

                    H <= H1;
                    S <= S1;
                    V <= V1;

          end
end

endmodule
```

Appendix III – RGB Converter

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:48:21 12/01/2009
// Design Name:
// Module Name:    RGB2HSV
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////////////
  //
  //  RGB to HSV conversion
  //
  ////////////////////////////////////////////////////////////////////////////

module RGB2HSV( R, G, B, clk, rst, H, S, V);

input clk, rst;
input [7:0] R, G, B;
output reg [7:0] S, V; // 0 to 255
output reg [8:0] H; // 0 to 360

reg [7:0] max, min, range;

reg [15:0] huedividend, satdividend;
wire [15:0] huequotient, satquotient;

wire [7:0] hueremainder, satremainder;
wire satrfd, huerfd;

reg [7:0] R_delay[19:0], G_delay[19:0], B_delay[19:0];
reg [7:0] min_delay[18:0], max_delay[18:0];

divider_hsv huedivider(clk, huedividend, range, huequotient, hueremainder, huerfd);
divider_hsv satdivider(clk, satdividend, max, satquotient, satremainder, satrfd);

always @(posedge clk)
          begin
                    if ((R >= G) & (R >= B))
                              begin
                                        max <= R;
                                        if (G <= B)
                                                  begin
                                                            min <= G;
```

```
                                        huedividend <= (16'd60 * (B - G)) & 16'b1111_1111_1111_1111;
                                        range <= R - G;
                                        satdividend <= (16'd255 * (R - G)) & 16'b1111_1111_1111_1111;
                                end
                        else if (B <= G)
                                begin
                                        min <= B;
                                        huedividend <= (16'd60 * (G - B)) & 16'b1111_1111_1111_1111;
                                        range <= R - B;
                                        satdividend <= (16'd255 * (R - B)) & 16'b1111_1111_1111_1111;
                                end
                end
        else if ((G >= R) & (G >= B))
                begin
                        max <= G;
                        if (R <= B)
                                begin
                                        min <= R;
                                        huedividend <= (16'd60 * (B - R)) & 16'b1111_1111_1111_1111;
                                        range <= G - R;
                                        satdividend <= (16'd255 * (G - R)) & 16'b1111_1111_1111_1111;
                                end
                        else if (B <= R)
                                begin
                                        min <= B;
                                        huedividend <= (16'd60 * (R - B)) & 16'b1111_1111_1111_1111;
                                        range <= G - B;
                                        satdividend <= (16'd255 * (G - B)) & 16'b1111_1111_1111_1111;
                                end
                end
        else if ((B >= R) & (B >= G))
                begin
                        max <= B;
                        if (R <= G)
                                begin
                                        min <= R;
                                        huedividend <= (16'd60 * (G - R)) & 16'b1111_1111_1111_1111;
                                        range <= B - R;
                                        satdividend <= (16'd255 * (B - R)) & 16'b1111_1111_1111_1111;
                                end
                        else if (G <= R)
                                begin
                                        min <= G;
                                        huedividend <= (16'd60 * (R - G)) & 16'b1111_1111_1111_1111;
                                        range <= B - G;
                                        satdividend <= (16'd255 * (B - G)) & 16'b1111_1111_1111_1111;
                                end
                end
    end

always @(posedge clk) begin
    R_delay[0] <= R;
    R_delay[1] <= R_delay[0];
    R_delay[2] <= R_delay[1];
    R_delay[3] <= R_delay[2];
    R_delay[4] <= R_delay[3];
    R_delay[5] <= R_delay[4];
    R_delay[6] <= R_delay[5];
    R_delay[7] <= R_delay[6];
    R_delay[8] <= R_delay[7];
    R_delay[9] <= R_delay[8];
    R_delay[10] <= R_delay[9];
    R_delay[11] <= R_delay[10];
    R_delay[12] <= R_delay[11];
    R_delay[13] <= R_delay[12];
    R_delay[14] <= R_delay[13];
    R_delay[15] <= R_delay[14];
    R_delay[16] <= R_delay[15];
    R_delay[17] <= R_delay[16];
    R_delay[18] <= R_delay[17];
    R_delay[19] <= R_delay[18];

    G_delay[0] <= G;
    G_delay[1] <= G_delay[0];
    G_delay[2] <= G_delay[1];
    G_delay[3] <= G_delay[2];
    G_delay[4] <= G_delay[3];
    G_delay[5] <= G_delay[4];
    G_delay[6] <= G_delay[5];
```

```
	G_delay[7] <= G_delay[6];
	G_delay[8] <= G_delay[7];
	G_delay[9] <= G_delay[8];
	G_delay[10] <= G_delay[9];
	G_delay[11] <= G_delay[10];
	G_delay[12] <= G_delay[11];
	G_delay[13] <= G_delay[12];
	G_delay[14] <= G_delay[13];
	G_delay[15] <= G_delay[14];
	G_delay[16] <= G_delay[15];
	G_delay[17] <= G_delay[16];
	G_delay[18] <= G_delay[17];
	G_delay[19] <= G_delay[18];

	B_delay[0] <= B;
	B_delay[1] <= B_delay[0];
	B_delay[2] <= B_delay[1];
	B_delay[3] <= B_delay[2];
	B_delay[4] <= B_delay[3];
	B_delay[5] <= B_delay[4];
	B_delay[6] <= B_delay[5];
	B_delay[7] <= B_delay[6];
	B_delay[8] <= B_delay[7];
	B_delay[9] <= B_delay[8];
	B_delay[10] <= B_delay[9];
	B_delay[11] <= B_delay[10];
	B_delay[12] <= B_delay[11];
	B_delay[13] <= B_delay[12];
	B_delay[14] <= B_delay[13];
	B_delay[15] <= B_delay[14];
	B_delay[16] <= B_delay[15];
	B_delay[17] <= B_delay[16];
	B_delay[18] <= B_delay[17];
	B_delay[19] <= B_delay[18];

	min_delay[0] <= min;
	min_delay[1] <= min_delay[0];
	min_delay[2] <= min_delay[1];
	min_delay[3] <= min_delay[2];
	min_delay[4] <= min_delay[3];
	min_delay[5] <= min_delay[4];
	min_delay[6] <= min_delay[5];
	min_delay[7] <= min_delay[6];
	min_delay[8] <= min_delay[7];
	min_delay[9] <= min_delay[8];
	min_delay[10] <= min_delay[9];
	min_delay[11] <= min_delay[10];
	min_delay[12] <= min_delay[11];
	min_delay[13] <= min_delay[12];
	min_delay[14] <= min_delay[13];
	min_delay[15] <= min_delay[14];
	min_delay[16] <= min_delay[15];
	min_delay[17] <= min_delay[16];
	min_delay[18] <= min_delay[17];

	max_delay[0] <= max;
	max_delay[1] <= max_delay[0];
	max_delay[2] <= max_delay[1];
	max_delay[3] <= max_delay[2];
	max_delay[4] <= max_delay[3];
	max_delay[5] <= max_delay[4];
	max_delay[6] <= max_delay[5];
	max_delay[7] <= max_delay[6];
	max_delay[8] <= max_delay[7];
	max_delay[9] <= max_delay[8];
	max_delay[10] <= max_delay[9];
	max_delay[11] <= max_delay[10];
	max_delay[12] <= max_delay[11];
	max_delay[13] <= max_delay[12];
	max_delay[14] <= max_delay[13];
	max_delay[15] <= max_delay[14];
	max_delay[16] <= max_delay[15];
	max_delay[17] <= max_delay[16];
	max_delay[18] <= max_delay[17];
	end

always @(posedge clk)
	begin
		V <= max_delay[18];
```

```
                if (max_delay[18] == 8'b0) S <= 8'b0;
                else  S <= satquotient;
                if (max_delay[18] == min_delay[18]) H <= 9'b0;
                else if (max_delay[18] == R_delay[19])
                        begin
                                if (G_delay[19] <= B_delay[19]) H <= (9'd360 - huequotient) & 9'b111_111_111;
                                else if (B_delay[19] <= G_delay[19]) H <= huequotient & 9'b111_111_111;
                        end
                else if (max_delay[18] == G_delay[19])
                        begin
                                if (R_delay[19] <= B_delay[19]) H <= (huequotient + 9'd120) & 9'b111_111_111;
                                else if (B_delay[19] <= R_delay[19]) H <= (9'd120 - huequotient) & 9'b111_111_111;
                        end
                else if (max_delay[18] == B_delay[19])
                        begin
                                if (R_delay[19] <= G_delay[19]) H <= (9'd240 - huequotient) & 9'b111_111_111;
                                else if (G_delay[19] <= R_delay[19]) H <= (huequotient + 9'd240) & 9'b111_111_111;
                        end
        end

endmodule

/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2006 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

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

`timescale 1ns/1ps

module divider_hsv(
        clk,
        dividend,
        divisor,
        quotient,
        remainder,
        rfd);


input clk;
input [15 : 0] dividend;
input [7 : 0] divisor;
output [15 : 0] quotient;
output [7 : 0] remainder;
output rfd;

// synopsys translate_off
```

```
      DIV_GEN_V1_0 #(
                  1,       // algorithm_type
                  0,       // bias
                  0,       // c_has_aclr
                  0,       // c_has_ce
                  0,       // c_has_sclr
                  0,       // c_sync_enable
                  1,       // divclk_sel
                  16,      // dividend_width
                  8,       // divisor_width
                  8,       // exponent_width
                  0,       // fractional_b
                  8,       // fractional_width
                  1,       // latency
                  8,       // mantissa_width
                  0)       // signed_b
         inst (
                  .CLK(clk),
                  .DIVIDEND(dividend),
                  .DIVISOR(divisor),
                  .QUOTIENT(quotient),
                  .REMAINDER(remainder),
                  .RFD(rfd),
                  .CE(),
                  .ACLR(),
                  .SCLR(),
                  .DIVIDEND_MANTISSA(),
                  .DIVIDEND_SIGN(),
                  .DIVIDEND_EXPONENT(),
                  .DIVISOR_MANTISSA(),
                  .DIVISOR_SIGN(),
                  .DIVISOR_EXPONENT(),
                  .QUOTIENT_MANTISSA(),
                  .QUOTIENT_SIGN(),
                  .QUOTIENT_EXPONENT(),
                  .OVERFLOW(),
                  .UNDERFLOW());


// synopsys translate_on

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

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

endmodule
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:50:09 12/01/2009
// Design Name:
// Module Name:    YCrCb2RGB
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

  ////////////////////////////////////////////////////////////////////////////
  //
  // ycrcb to RGB conversion
  //
  ////////////////////////////////////////////////////////////////////////////

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

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
```

Appendix IV – Gesture Recognition

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:16:27 12/01/2009
// Design Name:
// Module Name:    Center_of_Mass
// Project Name:
// Target Devices:
// Tool versions:
```

```
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////


  ///////////////////////////////////////////////////////////////////////////
  //
  // Center of Mass Calculator
  //
  ///////////////////////////////////////////////////////////////////////////

  module Center_of_Mass(ntsc_addr, ntsc_we, H1, S1, V1, H2, S2, V2, hi_hat_threshold,
                                                                            low_hat_threshold, hi_left_arm_threshold,
low_left_arm_threshold,
                                                                            hi_right_arm_threshold,
low_right_arm_threshold, clk, reset,
                                                                            hatx, haty, armx, army, direction, comvalid);

                   input [18:0] ntsc_addr;
                   input ntsc_we;
                   input [8:0] H1, H2;
                   input [7:0] S1, S2, V1, V2;
                   input [8:0] hi_hat_threshold, low_hat_threshold, hi_left_arm_threshold,
                                                       low_left_arm_threshold, hi_right_arm_threshold,
low_right_arm_threshold;
                   input clk, reset;
                   output reg [10:0] hatx, armx;
                   output reg [9:0] haty, army;
                   output reg comvalid;
                   output reg direction;

                   reg [18:0] last_ntsc_addr;
                   reg [10:0] x1, x2;
                   reg [9:0]  y1, y2;

                   wire [10:0] quotient_x1, quotient_x2;
                   wire [9:0]  quotient_y1, quotient_y2;
                   reg [18:0]  count1, count2, countgreen, countyellow;
                   reg [29:0] sumx1, sumy1, sumx2, sumy2, sumgreenx, sumyellowx, sumgreeny, sumyellowy;

                   always @(posedge clk) begin
                            if (reset) begin
                                     last_ntsc_addr <= 19'd0;
                                     x1 <= 11'd0;
                                     x2 <= 11'd0;
                                     y1 <= 11'd0;
                                     y2 <= 11'd0;
                                     count1 <= 19'd0;
                                     count2 <= 19'd0;
                                     countgreen <= 19'd0;
                                     countyellow <= 19'd0;
                                     sumx1 <= 30'd0;
                                     sumx2 <= 30'd0;
                                     sumy1 <= 30'd0;
                                     sumy2 <= 30'd0;
                                     sumyellowx <= 30'd0;
                                     sumgreenx <= 30'd0;
                                     sumyellowy <= 30'd0;
                                     sumgreeny <= 30'd0;
                                     hatx <= 11'd0;
                                     army <= 11'd0;
                                     haty <= 10'd0;
                                     army <= 10'd0;
                                     direction <= 1;
                                     comvalid <= 0;
                            end
                            else if (ntsc_we) begin
                                     last_ntsc_addr <= ntsc_addr;
                                     if (ntsc_addr != last_ntsc_addr) begin
                                              x1 <= {ntsc_addr[8:0], 1'b0};
                                              y1 <= ntsc_addr[18:9];
                                              x2 <= {ntsc_addr[8:0], 1'b0};
                                              y2 <= ntsc_addr[18:9];
```

```
                                        if ((x1 == 11'd100) & (y1 == 10'd100)) begin
                                                if (count1 > 19'd10) begin
                                                        hatx <= quotient_x1[10:0];
                                                        haty <= quotient_y1[9:0];
                                                        armx <= quotient_x2[10:0];
                                                        army <= quotient_y2[9:0];
                                                end
                                                count1 <= 19'd0;
                                                count2 <= 19'd0;
                                                countgreen <= 19'd0;
                                                countyellow <= 19'd0;
                                                sumx1 <= 30'd0;
                                                sumx2 <= 30'd0;
                                                sumy1 <= 30'd0;
                                                sumy2 <= 30'd0;
                                                sumyellowx <= 30'd0;
                                                sumgreenx <= 30'd0;
                                                sumyellowy <= 30'd0;
                                                sumgreeny <= 30'd0;

                                                comvalid <= 1;
                                        end
                                        else begin
                                                comvalid <= 0;
                                                if ((H1 <= hi_hat_threshold) & (H1 >= low_hat_threshold) &
                                                         (H2 <= hi_hat_threshold) & (H2 >= low_hat_threshold))
begin
                                                        sumx1 <= sumx1 + x1 + x2;
                                                        sumy1 <= sumy1 + y1 + y2;
                                                        count1 <= count1 + 2;
                                                end
                                                else if ((H1 <= hi_left_arm_threshold) & (H1 >=
low_left_arm_threshold) &
                                                                        (H2 <= hi_left_arm_threshold) & (H2
>= low_left_arm_threshold)) begin
                                                        sumgreenx <= sumgreenx + x1 + x2;
                                                        sumgreeny <= sumgreeny + y1 + y2;
                                                        countgreen <= countgreen + 2;
                                                end
                                                else if ((H1 <= hi_right_arm_threshold) & (H1 >=
low_right_arm_threshold) &
                                                                        (H2 <= hi_right_arm_threshold) & (H2
>= low_right_arm_threshold)) begin
                                                        sumyellowx <= sumyellowx + x1 + x2;
                                                        sumyellowy <= sumyellowy + y1 + y2;
                                                        countyellow <= countyellow + 2;
                                                end
                                        end
                                        if (countgreen > countyellow) begin
                                                sumx2 <= sumgreenx;
                                                sumy2 <= sumgreeny;
                                                count2 <= countgreen;
                                                direction <= 1; // going right (forward)
                                        end
                                        else if (countyellow > countgreen) begin
                                                sumx2 <= sumyellowx;
                                                sumy2 <= sumyellowy;
                                                count2 <= countyellow;
                                                direction <= 0; //going left (backward)
                                        end
                                end
                        end
                        else comvalid <= 0;
                end

                divider_com average_x1(.clk(clk), .dividend(sumx1), .divisor(count1), .quotient(quotient_x1),
.remainder(), .rfd());
                divider_com average_y1(.clk(clk), .dividend(sumy1), .divisor(count1), .quotient(quotient_y1),
.remainder(), .rfd());
                divider_com average_x2(.clk(clk), .dividend(sumx2), .divisor(count2), .quotient(quotient_x2),
.remainder(), .rfd());
                divider_com average_y2(.clk(clk), .dividend(sumy2), .divisor(count2), .quotient(quotient_y2),
.remainder(), .rfd());

  endmodule

/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used            *
*     solely for design, simulation, implementation and creation of           *
```

```
*     design files limited to Xilinx devices or technologies. Use            *
*     with non-Xilinx devices or technologies is expressly prohibited        *
*     and immediately terminates your license.                               *
*                                                                            *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"          *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION        *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION            *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS              *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE       *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY               *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR         *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF        *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS        *
*     FOR A PARTICULAR PURPOSE.                                              *
*                                                                            *
*     Xilinx products are not intended for use in life support               *
*     appliances, devices, or systems. Use in such applications are          *
*     expressly prohibited.                                                  *
*                                                                            *
*     (c) Copyright 1995-2006 Xilinx, Inc.                                   *
*     All rights reserved.                                                   *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

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

`timescale 1ns/1ps

module divider_com(
         clk,
         dividend,
         divisor,
         quotient,
         remainder,
         rfd);


input clk;
input [29 : 0] dividend;
input [18 : 0] divisor;
output [29 : 0] quotient;
output [18 : 0] remainder;
output rfd;

// synopsys translate_off

      DIV_GEN_V1_0 #(
                  1,       // algorithm_type
                  0,       // bias
                  0,       // c_has_aclr
                  0,       // c_has_ce
                  0,       // c_has_sclr
                  0,       // c_sync_enable
                  1,       // divclk_sel
                  30,      // dividend_width
                  19,      // divisor_width
                  8,       // exponent_width
                  0,       // fractional_b
                  19,      // fractional_width
                  1,       // latency
                  8,       // mantissa_width
                  0)       // signed_b
         inst (
                  .CLK(clk),
                  .DIVIDEND(dividend),
                  .DIVISOR(divisor),
                  .QUOTIENT(quotient),
                  .REMAINDER(remainder),
                  .RFD(rfd),
                  .CE(),
                  .ACLR(),
```

```
                    .SCLR(),
                    .DIVIDEND_MANTISSA(),
                    .DIVIDEND_SIGN(),
                    .DIVIDEND_EXPONENT(),
                    .DIVISOR_MANTISSA(),
                    .DIVISOR_SIGN(),
                    .DIVISOR_EXPONENT(),
                    .QUOTIENT_MANTISSA(),
                    .QUOTIENT_SIGN(),
                    .QUOTIENT_EXPONENT(),
                    .OVERFLOW(),
                    .UNDERFLOW());


// synopsys translate_on

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

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

endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:37:09 12/01/2009
// Design Name:
// Module Name:    line_calculator
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module line_calculator(clk, reset, comvalid, button0, haty, jumpliney, crouchliney);

input clk, reset, button0, comvalid;
input [9:0] haty;

output reg [9:0] jumpliney, crouchliney;

always @(posedge clk) begin
         if (reset) begin
                  jumpliney <= 9'b000_000_000;
                  crouchliney <= 9'b000_000_000;
         end
         else if (button0 & comvalid) begin
                  jumpliney <= haty - 9'b010_000_000;
                  crouchliney <= haty + 9'b010_000_000;
         end
end

endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:34:49 12/01/2009
// Design Name:
// Module Name:    action
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
```

```
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


  ////////////////////////////////////////////////////////////////////////////
  //
  // Stand, Jump, Crouch
  //
  ////////////////////////////////////////////////////////////////////////////

module action(jumpliney, crouchliney, haty, clk, reset, action);

input clk, reset;
input [9:0] haty, jumpliney, crouchliney;

output reg [1:0] action;

always @(posedge clk)
	begin
		if (reset) action <= 2'b01;
		else if (haty <= jumpliney) action <= 2'b10; // 2 for jump
		else if ((haty > jumpliney) & (haty < crouchliney)) action <= 2'b01; // 1 for stand
		else if (haty >= crouchliney) action <= 2'b00; // 0 for crouch
		else action <= 2'b01;
	end

endmodule
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:35:23 12/01/2009
// Design Name:
// Module Name:    runorwalk
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


  ////////////////////////////////////////////////////////////////////////////
  //
  // Arm Swing Period Calculator (Run or Walk)
  //
  ////////////////////////////////////////////////////////////////////////////

module runorwalk(hcount, vcount, armx, comvalid, clk, reset, movement);

input clk, reset, comvalid;
input [10:0] armx, hcount;
input [9:0] vcount;

output reg [1:0] movement;

reg beat, last_beat;
reg [10:0] x[7:0], x_average;
reg [29:0] count, period;
reg [13:0]sumx;

always @(posedge clk) begin
	if (reset) begin
		movement <= 2'b00; // no movement
		beat <= 0;
		last_beat <= 1;
		x[0] <= 0;
```

```
                    x[1] <= 0;
                    x[2] <= 0;
                    x[3] <= 0;
                    x[4] <= 0;
                    x[5] <= 0;
                    x[6] <= 0;
                    x[7] <= 0;
                    count <= 0;
                    sumx <= 0;
          end
          else if ((hcount == 11'd100) & (vcount == 10'd100)) begin
                    last_beat <= beat;
                    x[0] <= armx;
                    x[1] <= x[0];
                    x[2] <= x[1];
                    x[3] <= x[2];
                    x[4] <= x[3];
                    x[5] <= x[4];
                    x[6] <= x[5];
                    x[7] <= x[6];
                    sumx <= (x[0] + x[1] + x[2] + x[3] + x[4] + x[5] + x[6] + x[7]);
                    x_average <= sumx[13:3];
                    if (beat != last_beat) begin // changed direction
                              count <= 0;
                              period <= count;
                    end
                    else begin
                              if ((armx > (x[0] + 15)) & (armx > x_average)) begin // still moving right
                                        count <= count + 1;
                              end
                              else if ((armx > (x[0] + 15)) & (armx < x_average)) begin // changed direction to right
                                        beat <= ~beat;
                              end
                              else if ((armx < (x[0] - 15)) & (armx > x_average)) begin // changed direction to left
                                        beat <= ~ beat;
                              end
                              else if ((armx < (x[0] - 15)) & (armx < x_average)) begin // still moving left
                                        count <= count + 1;
                              end
                              else begin
                                        movement <= 2'b00;
                              end
                    end
                    if (period > 5) begin
                              movement <= 2'b01;
                    end
                    else begin
                              movement <= 2'b00;
                    end
          end
end


endmodule
```

Appendix V – Game Wrapper Module

```
`default_nettype none

 ///////////////////////////////////////////////////////////////////////////////
  //
  // GAME WRAPPER AND SUPERVISOR MODULE
  //
  // Should have multiple sub modules
  // COLOR_LUT, BACKGROUND_GEN, BACKROUND_MEMORY, TILE_MEMORY,
  // FRAME_BUFFER, BLOB_GENERATOR, BLOB_RAM, PLAYER_FSM,
  // SPRITE_FSM,
  //
  // The way these modules interface are detailed in the block diagram
  // Eventually introduce a "EVENT_BUS" output
  // Information about game win/loss/death/ gets output through this
  // Information to be sent to the AUDIO_OUTPUT to create sound
  // effects based on the events happening on screen
  ///////////////////////////////////////////////////////////////////////////////
module game_engine(input vclock, reset, up, down, left, right, hsync, vsync, blank,
                                         input [10:0] hcount,
```

```
                                        input [9:0] vcount,
                                        output [23:0] pixel,
                                        output [10:0] player_xcoord,
                                        output [9:0] player_ycoord,
                                              output game_over);

         // Wire up interconnects
                  wire [8:0] back_data;
                  wire [12:0] sprite_data;
                  wire [7:0] back_row, back_column, sprite_row, sprite_column;
                  wire back_we, sprite_we, new_sprite;
                  wire [7:0] world_column;
                  wire [3:0] world_row;
                  wire [5:0] background_tile;
                  wire [5:0] back_tile_pixel,sprite_tile_pixel;
                  wire [2:0] back_pixel_type;
                  wire [2:0] sprite_pixel_type;
                  wire back_pixel_transparent, sprite_pixel_transparent;
                  wire [5:0] back_tile_index,sprite_tile_index;
                  wire [3:0] back_tile_row,sprite_tile_row;
                  wire [3:0] back_tile_column,sprite_tile_column;
                  wire [2:0] o_pixel_data;
                  wire [5:0] pixel_i;
                  wire [11:0] sg_xcoord_i,sg_xcoord_o,fsm_xcoord_i,fsm_xcoord_o;
                  wire [7:0] sg_ycoord_i,sg_ycoord_o,fsm_ycoord_i,fsm_ycoord_o;
                  wire [10:0] sg_sstate_i,sg_sstate_o,fsm_sstate_i,fsm_sstate_o;
                  wire [4:0] sg_sprite_tile_i,sg_sprite_tile_o,fsm_sprite_tile_i,fsm_sprite_tile_o;
                  wire [3:0] sg_sprite_number;
                  wire [3:0] fsm_sprite_number;
                  wire sg_ram_we,fsm_ram_we;
                  wire [63:0] display_bus;
                  wire vertical_collision;                                    // Vertical movement stopped by in
game object
                  wire [3:0] state;                                                             //
                  wire player_collision;
                  wire [11:0] player_collision_xcoord;
                  wire [9:0] player_collision_ycoord;
                  // wire background_vertical_collision;
                  // wire background_horizontal_collision;
                  wire win_game;                                                                         // Win
game signal
                  wire player_fell;                                                             // Fall off the
map
                  wire player_size;                                                             // Big Mario vs
Little Mario. Addt'l functionality future proofing
                  // Movement gesture information
                  wire forward;
                  wire backward;
                  wire jump;
                  wire crouch;
                  wire still;
                  wire [11:0] left_edge;                                                        // Leftmost pixel
onscreen no scrolling back
                  wire [2:0] collision_type;                                          // What collison, based on
pixel metadata
                  wire [4:0] new_sprite_tile;                                         // Which new sprite is being
created?
                  wire [4:0] new_sprite_row;                                          // Where is this sprite
being created?
                  wire [3:0] num_sprites_on_screen;
                  wire [3:0] fb_sprite_number;                                // Sprite identification number
                  wire [3:0] collision_sprite_number;                         // identifies colliding sprites
                  wire [3:0] player_collision_type;                           // Collision using metadata
                  wire [3:0] new_sstate;

                  wire mario_calc;
                  reg transparent_high;

                  // Movement commands
                  assign jump = up;
                  assign crouch = down;
                  assign backward = left;
                  assign forward = right;
                  assign still = ~(|{left,right});
                  // We only use two bits for each color
                  // So we repeat color information in each channel
                  // To send required number of bits to XVGA
                  wire [5:0] pixela;
                  assign pixela =  pixel_i;
```

```
		assign pixel = {pixela[5], pixela[4], pixela[5], pixela[4], pixela[5], pixela[4], pixela[5],
pixela[4],
						pixela[3], pixela[2], pixela[3], pixela[2], pixela[3], pixela[2],
pixela[3], pixela[2],
						pixela[1], pixela[0], pixela[1], pixela[0], pixela[1], pixela[0],
pixela[1], pixela[0]};

	// all of our display data
	assign display_bus = {		24'h000,
							3'b000,player_collision,
							collision_sprite_number,
							num_sprites_on_screen,
							3'b000,sprite_pixel_transparent,

	3'b000,o_pixel_data[2],3'b000,o_pixel_data[1],3'b000,o_pixel_data[0],
							3'b000,transparent_high,

	3'b000,fsm_sstate_i[10],3'b000,fsm_sstate_i[9],3'b000,fsm_sstate_i[8]

	,3'b000,fsm_sstate_i[3],3'b000,fsm_sstate_i[0]};

	// Are we looking at the player sprite
	assign mario_calc = (((hcount > player_xcoord) && (hcount < (player_xcoord + 16))) && ((vcount > player_ycoord)
&& (vcount < (player_ycoord + 16))));

		// Module instantiation

		level_rom				cart(
										.vclock(vclock),

	.world_column(world_column),

	.world_row(world_row),

	.background_tile(background_tile)
										);
		level_creator			smb(
										.vclock(vclock),
										.reset(reset),
										.vcount(vcount),

	.left_edge(left_edge),

	.background_tile(background_tile),

	.world_column(world_column),

	.world_row(world_row),

	.back_tile_pixel(back_tile_pixel),

	.back_pixel_type(back_pixel_type),

	.back_pixel_transparent(back_pixel_transparent),

	.back_tile_index(back_tile_index),

	.back_tile_row(back_tile_row),

	.back_tile_column(back_tile_column),
										.back_we(back_we),

	.back_data(back_data),

	.back_row(back_row),

	.back_column(back_column),

	.new_sprite(new_sprite),

	.new_sprite_tile(new_sprite_tile),

	.new_sprite_row(new_sprite_row)
										);
		tile_memory					tm(
										.vclock(vclock),
										.reset(reset),
```

```
.back_tile_index(back_tile_index),
.back_tile_row(back_tile_row),
.back_tile_column(back_tile_column),
.sprite_tile_index(sprite_tile_index),
.sprite_tile_row(sprite_tile_row),
.sprite_tile_column(sprite_tile_column),
.back_tile_pixel(back_tile_pixel),
.back_pixel_type(back_pixel_type),
.sprite_tile_pixel(sprite_tile_pixel),
.sprite_pixel_type(sprite_pixel_type),
.back_pixel_transparent(back_pixel_transparent),
.sprite_pixel_transparent(sprite_pixel_transparent)
);
frame_buffer fb(
.vclock(vclock),
.reset(reset),
.hcount(hcount),
.vcount(vcount),
.back_row(back_row),
.back_column(back_column),
.back_data(back_data),
.back_we(back_we),
.sprite_row(sprite_row),
.sprite_column(sprite_column),
.sprite_data(sprite_data),
.sprite_we(sprite_we),
.o_pixel_color(pixel_i),
.o_pixel_data(o_pixel_data),
.o_sprite_number(fb_sprite_number),
.bottom_color_on(1'b0)
);
sprite_generator sg(
.vclock(vclock),
.vcount(vcount),
.hcount(hcount),
.reset(reset),
.left_edge(left_edge),
.xcoord_i(sg_xcoord_i),
.ycoord_i(sg_ycoord_i),
.sstate_i(sg_sstate_i),
.sprite_tile_i(sg_sprite_tile_i),
.sprite_tile_pixel(sprite_tile_pixel),
.sprite_pixel_type(sprite_pixel_type),
.sprite_pixel_transparent(sprite_pixel_transparent),
.fb_pixel_data(o_pixel_data),
.player_xcoord(player_xcoord),
```

```
.player_ycoord(player_ycoord),
.player_collision_sprite_number(collision_sprite_number),
.player_collision_i(player_collision),
.player_collision_type(player_collision_type),
.xcoord_o(sg_xcoord_o),
.ycoord_o(sg_ycoord_o),
.sstate_o(sg_sstate_o),
.sprite_tile_o(sg_sprite_tile_o),
.sprite_number_o(sg_sprite_number),
.ram_we(sg_ram_we),
.sprite_tile_index(sprite_tile_index),
.sprite_tile_row(sprite_tile_row),
.sprite_tile_column(sprite_tile_column),
.sprite_row(sprite_row),
.sprite_column(sprite_column),
.sprite_data(sprite_data),
.sprite_we(sprite_we),
.other_sprite_collision(vertical_collision),
.sprite_state(state),
.new_sstate(new_sstate),
.fb_sprite_number(fb_sprite_number)
);

sprite_ram sr(
.vclock(vclock),
.sg_address(sg_sprite_number),
.fsm_address(fsm_sprite_number),
.sg_data_i({sg_xcoord_o,sg_ycoord_o,sg_sprite_tile_o,sg_sstate_o}),
.fsm_data_i({fsm_xcoord_o,fsm_ycoord_o,fsm_sprite_tile_o,fsm_sstate_o}),
.sg_data_o({sg_xcoord_i,sg_ycoord_i,sg_sprite_tile_i,sg_sstate_i}),
.fsm_data_o({fsm_xcoord_i,fsm_ycoord_i,fsm_sprite_tile_i,fsm_sstate_i}),
.sg_we(sg_ram_we),
.fsm_we(fsm_ram_we)
);

player_controller pc(
.vclock(vclock),
.reset(reset),
.vcount(vcount),
.hcount(hcount),
.left_edge(left_edge),
.o_pixel_data(o_pixel_data),
.collision_sprite_number(collision_sprite_number),
.player_collision(player_collision),
.player_collision_type(player_collision_type),
.win_game(win_game),
```

```
        .player_pixel_on(mario_calc),
        .player_xcoord(player_xcoord),
        .player_ycoord(player_ycoord),
        .forward(forward),
        .backward(backward),
        .still(still),
        .crouch(crouch),
        .jump(jump),
        .player_size(player_size),
        .player_fell(player_fell),
        .fb_sprite_number(fb_sprite_number),
        .game_over(game_over)
    );
    game_fsm gf(
        .vclock(vclock),
        .reset(reset),
        .vcount(vcount),
        .hcount(hcount),
        .new_sprite(new_sprite),
        .new_sprite_tile(new_sprite_tile),
        .new_sprite_row(new_sprite_row),
        .xcoord_i(fsm_xcoord_i),
        .ycoord_i(fsm_ycoord_i),
        .player_xcoord(player_xcoord),
        .player_ycoord(player_ycoord),
        .sstate_i(fsm_sstate_i),
        .sprite_tile_i(fsm_sprite_tile_i),
        .ram_we(fsm_ram_we),
        .win_game(win_game),
        .player_fell(player_fell),
        .left_edge(left_edge),
        .xcoord_out(fsm_xcoord_o),
        .ycoord_out(fsm_ycoord_o),
        .sstate_out(fsm_sstate_o),
        .sprite_tile_out(fsm_sprite_tile_o),
        .sprite_number(fsm_sprite_number),
        .player_size(player_size),
        .num_sprites_on_screen(num_sprites_on_screen),
        .game_over(game_over)
    );
endmodule

 ////////////////////////////////////////////////////////////////////////////
 //
 // Level Creator
 // This module is responsible for holding each of the level "cartridges"
 // And will determine the proper level display based on the current player position on screen
 // The frame buffer should be filled up by this and the blob/actor generator
 // Need to be careful of timing considerations, and making sure everything is filled up
 // Before the next redraw of the screen.
```

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


module level_creator(input vclock, reset,
                                              input [9:0] vcount,
                                              input [11:0] left_edge,
                                              input [5:0] background_tile,
                                              input [5:0] back_tile_pixel,
                                              input [2:0] back_pixel_type,
                                              input back_pixel_transparent,
                                              output [7:0] world_column,
                                              output [3:0] world_row,
                                              output [5:0] back_tile_index,
                                              output [3:0] back_tile_row,
                                              output [3:0] back_tile_column,
                                              output back_we,
                                              output [8:0] back_data,
                                              output [7:0] back_row,
                                              output [7:0] back_column,
                                              output new_sprite,
                                              output [4:0] new_sprite_tile,
                                              output [4:0] new_sprite_row);
         reg [7:0] last_updated_column;                                  // holds the last world column checked for new
sprites
         wire [11:0] world_hcount_display;


         // Registers for pipelining
         reg [7:0] back_column_d1,back_column_d2,back_column_d0;
         reg [7:0] back_row_d1,back_row_d2,back_row_d0;
         reg [3:0] back_tile_row_d1,back_tile_column_d1,back_tile_column_d2;
         reg [11:0] world_hcount_display_d1,world_hcount_display_d2;
         reg [7:0] world_column_d1,world_column_d2;

         //to tile memory
         assign back_tile_index = background_tile[5] ? 6'b000000 : background_tile;          // if the object is a
sprite, display blue background
         assign back_tile_row = back_row_d1 [3:0];
         // bottom 4 bits are the row of the tile
         assign back_tile_column = world_hcount_display_d1[3:0];                                   // bottom 4 bits
are the column of the tile

         // to frame buffer
         assign back_data = back_pixel_transparent ? 9'b000111000 : {back_tile_pixel,back_pixel_type};
         assign back_we = ((vcount > 720) && (vcount <= 775));                                                // 55
lines to store the next screen
         assign back_column = back_column_d2;
         assign back_row = back_row_d2;
         // to background memory
         assign world_column = world_hcount_display[11:4];
         assign world_row = back_row_d0 >> 4;
         assign world_hcount_display = (left_edge + back_column_d0);

         always @ (posedge vclock) begin
                  //pipeline delays
                  back_column_d1 <= back_column_d0;
                  back_row_d1 <= back_row_d0;
                  back_column_d2 <= back_column_d1;
                  back_row_d2 <= back_row_d1;
                  back_tile_row_d1 <= back_tile_row;
                  back_tile_column_d1 <= back_tile_column;
                  world_hcount_display_d1 <= world_hcount_display;
                  world_hcount_display_d2 <= world_hcount_display_d1;
                  world_column_d1 <= world_column;
                  world_column_d2 <= world_column_d1;
                  if (reset) begin
                           back_column_d0 <= 0;
                           back_row_d0 <= 0;
                  end
                  else if ((vcount >= 720) && (vcount < 780)) begin                                   // if vcount is
off of the low-res screen
                           back_column_d0 <= back_column_d0 + 1;
         // increment the back_column every clock cycle
                           if (back_column == 8'b11111111)
         // if hcount is at the end of the low-res screen
                                    back_row_d0 <= back_row_d0 + 1;
                           if ((back_row_d2 == 239) && (back_column_d2 == 255))
```

```
                                    last_updated_column <= world_column_d2;                                    //is
updated at the end of every write cycle
                  end
         end // always @ vclock
         assign new_sprite = ((back_column_d2 == 8'b11111111) && (background_tile[5] == 1) &&(back_tile_row_d1 ==
4'b0000) &&(back_tile_column_d2 == 4'b0000) &&~(last_updated_column == world_column_d2)); // only add a new sprite when
the top left corner of it just touches the screen
         assign new_sprite_tile = background_tile[4:0];
         assign new_sprite_row = world_row;
endmodule



//////////////////////////////////////////////////////////////////////
//
// Level ROM
// Attempt to debug memory issues by dynamically generating maps in logic
//////////////////////////////////////////////////////////////////////

module level_rom(input vclock,
                                                      input [7:0] world_column,
                                                      input [3:0] world_row,
                                                      output [5:0] background_tile);
backrom                                      br(.addr({world_column[7:4],world_row,world_column[3:0]}),
                                                                         .clk(vclock),
                                                                         .dout(background_tile)
                                                                         );
endmodule

  ///////////////////////////////////////////////////////////////////////////////
  //
  // Frame Buffer
  // This allows us to make a much nicer looking display,
  // and avoids the wonderful issue of muxing all of the potential sprites
  // In original NES implementation, there was a 256x240 frame buffer.
  // Determine timing such that the frame buffer is only written to after a certain amount of time.
  // Avoid writing to the buffer during a current draw operation.
  // If scaled up with a 4:3 pixel ratio, wait until vcount >720 to draw.
  // Alternatively, with only 2x2 scaling and static background frame,
  // Give yourself more time to redraw the screen.
  //
  // 256x240 resolution.
  //
  //
  // Changelog:
  //     12/1/2009 : To avoid timing issues caused by reading and writing to a single ZBT
  //                                        the frame buffer is instead implemented in a BRAM. There is ample space
available
  //                                        on the FPGA to hold all of the buffers and  memories we desire
  ///////////////////////////////////////////////////////////////////////////////
module frame_buffer(input vclock,
                                             input reset,
                                             input [10:0] hcount,
                                             input [9:0] vcount,
                                             input [7:0] back_row,
                                             input [7:0] back_column,
                                             input [8:0] back_data,
                                             input back_we,
                                             input [7:0] sprite_row,
                                             input [7:0] sprite_column,
                                             input [12:0] sprite_data,
                                             input sprite_we,
                                             input bottom_color_on,
                                             output [5:0] o_pixel_color,          // output pixel color data;
                                             output [2:0] o_pixel_data,           // pixel characteristics
                                             output [3:0] o_sprite_number);       //which sprite in the sprite ram is
this? (for collision detection)
         wire [12:0] o_data;
         wire [15:0] sprite_output_choose;                               // will be the hcount,vcount address if
outputting to screen, otherwise will write sprite data in
         wire choose_we;                                                                   // choose what to use for
write enable for wea
         wire [12:0] internal_output_data;

         reg [7:0] vcount_by_three; // running count of hcount divided by three
         reg [1:0] count;

         always @ (posedge vclock) begin
         // We can change scaling factors by changing the parameters here
```

```
	// We just change how long we spend drawing each pixel
		if (vcount == 0) begin
			vcount_by_three <= 8'b00000000;
			count <= 2'b00;
		end
		else if (hcount == 1340) begin
			count <= count + 1;
		end
		else begin
			vcount_by_three <= vcount_by_three;
			count <= count;
		end
		if(count == 3) begin
			vcount_by_three <= vcount_by_three + 1;
			count <= 2'b00;
		end
	end

	// To avoid glitching, make sure we are either reading or writing to the frame buffer
	// never both
	assign sprite_output_choose = (vcount > 720)   ? {sprite_row,sprite_column} : {vcount_by_three,hcount[9:2]};
	assign choose_we = (vcount > 720) ? sprite_we : 0;

	// The output data stream
	// The frame buffer contains information relevant not only to the display,
	// But also to internal game logic
	assign o_data = internal_output_data;
	assign o_pixel_color = (vcount <= 720) ? o_data[8:3] : bottom_color_on ? 9'b100010001 : 0;
	assign o_pixel_data = o_data[2:0];                                              // data always stays on
	assign o_sprite_number = o_data[12:9];                                          // so does the sprite number

	// instantiating the 256x240x9 frame buffer memory
	buffermem                   framebuf(          .addra(sprite_output_choose),

	.addrb({back_row,back_column}),
								.clka(vclock),
								.clkb(vclock),
								.dina(sprite_data),
								.dinb({4'b000,back_data}),

	.douta(internal_output_data),
								.wea(choose_we),
								.web(back_we));
endmodule


  ///////////////////////////////////////////////////////////////////////////
  //
  // SPRITE_FSM
  // This module is the brains of the entire game. Enemy and item logic is
  // determined here. Simple "gravity" and collision detection is based on events.
  // Eventually items, scoring, timing, and win conditions are implemented here.
  //
  // Remember, we use a coordinate system that is zero indexed
  // With increasing values from the top left to the bottom right
  ///////////////////////////////////////////////////////////////////////////


  module game_fsm(  input vclock,
						input reset,
						input [9:0] vcount,
						input [10:0] hcount,
						input new_sprite,
						input [4:0] new_sprite_tile,                //which
sprite is new on the board
						input [4:0] new_sprite_row,                 //how
high does the sprite start off? comes from the world row output
						input [11:0] xcoord_i,
	//sprite x coordinates
						input [7:0] ycoord_i,
	//sprite y coordinates
						input [10:0] sstate_i,
	//sprite state
						input [4:0] sprite_tile_i,
	//which tile is the sprite?
						input win_game,
	//did the player just win the game?
```

```
                                                    input player_fell,
         //did the player just fall down a hole?
                                                    input wire [11:0] left_edge,
         //leftmost border of the screen, after this clear memory
                                                    input [10:0] player_xcoord,                       // Draw
Player Sprite
                                                    input [9:0] player_ycoord,                        // At the
correct coordinates
                                                    //sprite ram
                                                    output [11:0] xcoord_out,
                                                    output [7:0] ycoord_out,
                                                    output [10:0] sstate_out,
                                                    output [4:0] sprite_tile_out,
                                                    output reg [3:0] sprite_number,
                                                    output wire ram_we,
                                                    // Player information
                                                    output reg player_size,
         //1 means big
                                                    output reg [3:0] num_sprites_on_screen, // will we have room to
add another sprite?
                                                    output game_over);
         //sprite ram data
         reg [11:0] xcoord_o;
         reg[7:0] ycoord_o;
         reg [10:0] sstate_o;
         reg [4:0] sprite_tile_o;
         //player data
         reg [3:0] state;
         reg [7:0] ycoord_hold;
         reg [11:0] xcoord_hold;
         reg [10:0] sstate_hold;
         reg [4:0] sprite_tile_hold;

         wire lose_game;
         reg new_player_size;
         reg [5:0] new_sprite_tile_hold;
         reg [4:0] new_sprite_row_hold;
         reg lose_game_i;
         reg [2:0] game_clock; // We could map this to switches to change speed, like Lab5



         /////////////////////////////////////////////////////////////////////////////////////////
         // State transitions
         parameter s_start = 0;
         parameter s_1 = 1;
         parameter s_2 = 2;
         parameter s_3 = 3;
         parameter s_4 = 4;
         parameter s_5 = 5;
         parameter s_6 = 6;
         parameter s_7 = 7;
         parameter s_8 = 8;
         parameter s_lose = 9;
         parameter s_reset = 10;
         parameter s_reset_1 = 11;
         parameter s_reset_2 = 12;

         always @ (posedge vclock) begin // state transitions
                  if (reset) begin
                           state <= s_reset;
                           sprite_number <= 0;
                           player_size <= 0;
                           num_sprites_on_screen <= 0;
                           game_clock <= 0;
                  end

                  else case (state)
                           s_start : state <= ((vcount == 0) && (hcount == 0)) ? s_1 : new_sprite ? s_5 : state;
                           s_1 : state <= s_2; //wait state
                           s_2 : state <= (win_game) ? s_lose : (sprite_tile_i == 0) ? s_4 : s_3;              //win or
lose, just pause the game send audio signals
                           s_3 : state <= (lose_game) ? s_lose : s_4;
                           s_4 : state <= (sprite_number == 15) ? s_start : s_1;
                           s_5 : state <= s_6;
                           s_6 : state <= (sprite_tile_i == 0) ? s_7 : s_8;
                           s_7 : state <= s_start;
                           s_8 : state <= (sprite_number == 15) ? s_start : s_5;
                           s_lose : state <= state;
```

```
				s_reset : state <= s_reset_1;
				s_reset_1 : state <= s_reset_2;
				s_reset_2 : state <= (sprite_number == 15) ? s_start : s_reset;
				default : state <= s_start;
			endcase

			if (state == s_4) sprite_number <= sprite_number + 1;
		// we just updated a sprite
			if (state == s_8) sprite_number <= sprite_number + 1;
		// we just made a new sprite
			if (state == s_reset_2) sprite_number <= sprite_number + 1;
		// Initialize all the sprites
			if (state == s_start) sprite_number <= 0;
				// Sprite RAM should be empty/ we can overwrite

			// latching
			if (state == s_2) begin
				ycoord_hold <= ycoord_i;
				xcoord_hold <= xcoord_i;
				sprite_tile_hold <= sprite_tile_i;
				sstate_hold <= sstate_i;
			end

			// if a new sprite is sent
			if ((state == s_start) && new_sprite) begin
		// will only be sent while FSM is in s_start
				new_sprite_tile_hold <= new_sprite_tile;
			// put coordinates in registers
				new_sprite_row_hold <= new_sprite_row;
			end

			//test signal, how many sprites are we displaying?
			if (state == s_3) num_sprites_on_screen <= sprite_number;

			// player size
			if (state == s_3) player_size <= new_player_size;
			if ((hcount == 0) && (vcount == 0))
			game_clock <= game_clock + 1;
		end // state transitions


		//// combinational logic
		assign game_over = (state == s_lose);
				// halt because we've lost
		assign ram_we = ((state == s_3) || (state == s_7) || (state == s_reset_1));

		// Information to be stored in RAM
		assign xcoord_out = (state == s_7) ? left_edge + 256 : (state == s_reset_1) ? 0 : xcoord_o;
		assign ycoord_out = (state == s_7) ? ({new_sprite_row_hold,4'b0000}): (state == s_reset_1) ? 0 : ycoord_o;
		assign sstate_out = (state == s_7) ? 11'b00000000010 : (state == s_reset_1) ? 0 : sstate_o;
		assign sprite_tile_out = (state == s_7) ? new_sprite_tile_hold : (state == s_reset_1) ? 0 : sprite_tile_o;
		assign lose_game = (lose_game_i || player_fell); //two ways to lose

		always @ (xcoord_hold,ycoord_hold,sprite_tile_hold,sstate_hold,player_size,left_edge,game_clock) begin

			//player size and lose game
			if(sstate_hold[4] && (sstate_hold[10:8] == 3'b100) && player_size) begin		// if player is
big
				new_player_size = 0;
				lose_game_i = 0;
			end
			else if(sstate_hold[4] && (sstate_hold[10:8] == 3'b100) && ~player_size) begin	// if player is
little
				new_player_size = 0;
				lose_game_i = 1;
			end
			else begin
				new_player_size = player_size;
				lose_game_i = 0;
			end

		// Enemy Logic for Goomba
			if (sprite_tile_hold == 5'd0) //default tile ? do nothing
				{xcoord_o,ycoord_o,sprite_tile_o,sstate_o} =
{xcoord_hold,ycoord_hold,sprite_tile_hold,sstate_hold};
			else if ((sprite_tile_hold == 5'b00001) || (sprite_tile_hold == 5'b00010)) begin // Goomba Enemy Logic
				if (game_clock[1:0] == 0) begin
					if (sstate_hold[1] && sstate_hold[2]) //moving right
						xcoord_o = xcoord_hold + 1;
```

```
					else if (sstate_hold[1] && ~sstate_hold[2]) //moving left
						xcoord_o = xcoord_hold - 1;
					else xcoord_o = xcoord_hold; //otherwise stay stationary
			end
			else xcoord_o = xcoord_hold;
				//y coordinates
				ycoord_o = ycoord_hold + 1; //gravity
			//next sprite tile
			if (((xcoord_hold + 16) <= left_edge) || (ycoord_hold >= 240) || (xcoord_hold >= (left_edge + 260)))
// if the tile is off the screen
				sprite_tile_o = 0; // remove the sprite
			else if (sstate_hold[4] && (sstate_hold[10:8] == 3'b101)) // if the player killed the sprite
				sprite_tile_o = 0; // remove the sprite
			else if (xcoord_hold[2])
				sprite_tile_o = 5'b00001;
			else sprite_tile_o = 5'b00010; //otherwise keep the old tile

			//sstate
			sstate_o = sstate_hold; //do nothing to the state
		end // Goomba Enemy Logic
		//otherwise do nothing to the sprite
		else {xcoord_o,ycoord_o,sprite_tile_o,sstate_o} = {xcoord_hold,ycoord_hold,sprite_tile_hold,sstate_hold};
		end //enemy logic
endmodule

/////////////////////////////////////////////////////////////////////////////////////////
//
// player_controller
// Description: controls all the player's movements. It also detects collisions, and sets
// the appropriate flags high.
//
// player control was originally abstracted away from the game logic
// such that additional functionality could be added to this module
// e.g. A stickman dynamic representation of the player would take the x and y coordinates
// of the player's arms and mimic the player's configuration.
// xcoord and ycoord would tell this drawing module where to insert this dynamic representation
// dynamic representation could be performed as an overlay on top of the frame buffer image
/////////////////////////////////////////////////////////////////////////////////////////

module player_controller(
				input vclock,								//system
clock
				input reset,
				input [9:0] vcount,
				input [10:0] hcount,
				// frame buffer
				input [2:0] o_pixel_data,				// pixel metadata for collisions /
death
				input [3:0] fb_sprite_number,				// which of our 16 sprites are we
dealing with
				// Input control signals.
				// hold a one while each "action" is being performed
				input forward,
				input backward,
				input still,
				input crouch,
				input jump,
				// from sprite_fsm
				input player_pixel_on,					// are we dealing with the
player sprite
				input player_size,					// big/little mario
				input game_over,						// game has ended,
stop movement
				// to sprite_fsm
				output reg win_game,						// game has won,
tell audio to play song
				output reg player_fell,					// player has lost, halt
game
				// to various
				output reg [11:0] left_edge,			// what is our leftmost pixel?
				// to sprite generator
				output reg [10:0] player_xcoord,		// where is the player
				output reg [9:0] player_ycoord,		// where is the player
				output reg [4:0] sprite_tile_out,	// draw one of these sprites
				output reg [3:0] sprite_number,		// which sprite from RAM
				output wire ram_we,						// write to the
RAM
				//to sprite_generator
				output reg [3:0] collision_sprite_number, // which sprite did we collide with
```

```
                output reg player_collision,                                    // has the player collided
                output reg [2:0] player_collision_type);        // does the player live or die?

        // potentially simplify this logic
        // Currently written for a jump gesture that could be held
        // Allowing for finer control of jump length and height
        wire jump_hold;
        reg [5:0] jump_length;
        reg [2:0] jump_state;
        wire new_jump;
        reg in_the_air;
        assign jump_hold = |jump_length;                                // is the player still jumping?
        assign new_jump = (jump_state == 2'b01);

        always @ (posedge vclock) begin                                         // general game logic
                if (reset) begin
        // initialize the game
                        collision_sprite_number <= 0;
                        player_collision <= 0;
                        win_game <= 0;
                        player_xcoord <= 128;
                        player_ycoord <= 114;
                        jump_length <= 0;
                        player_fell <= 0;
                        left_edge <= 0;
                        jump_state <= 0;
                end
                else if (game_over) begin                                               // player does not
move, the game halts
                        player_xcoord <= player_xcoord;
                        player_ycoord <= player_ycoord;
                end
                else if ((vcount == 1) && (hcount == 1)) begin                          //restart at
vcount = 1 and hcount = 1
                        collision_sprite_number <= 0;
                        player_collision <= 0;
                        player_collision_type <= 0;
                end
                else if (player_pixel_on) begin
        // game events
                        if ((o_pixel_data == 3'b100) ||     (o_pixel_data == 3'b101) || (o_pixel_data == 3'b110))
begin
                                collision_sprite_number <= fb_sprite_number;            //what is the last thing the
player hit?
                                player_collision <= 1;
                                player_collision_type <= o_pixel_data;
                        end
                        else if (o_pixel_data == 3'b010 && forward) begin               //if going foward and hit
something
                                player_xcoord <= player_xcoord - 2;
                        end
                        else if (o_pixel_data == 3'b001 && backward)                    //if going backward and hit
something
                                player_xcoord <= player_xcoord + 2;
                        else if (o_pixel_data == 3'b011 && jump_hold) begin     //if jumping and hit something
                                player_ycoord <= player_ycoord + 5;
                                jump_length <= 0;
                        end
                        else if (o_pixel_data == 3'b011) begin                                  // if
landed on the ground
                                player_ycoord <= player_ycoord - 1;
        // popped back up
                                in_the_air <= 0;
                                // the player is back on solid ground again
                        end
                        else if (player_ycoord < 4)
        // gravitational pull
                                player_ycoord <= player_ycoord + 1;
                        else if (player_xcoord > 512) begin  // if the player is more than half way
                                player_xcoord <= player_xcoord - 4;
                                left_edge <= left_edge + 1;
        // move the screen over
                        end
                        else if (o_pixel_data == 3'b111)
        // grab the flag at the end of the game
                                win_game <= 1;
                        else if (vcount == 750)
                                player_fell <= 1;
                        // if the player falls off the bottom of the screen
```

```
		end // game events
		// change the player's state every frame
		else if((vcount == 720) && (hcount == 1)) begin

			case(jump_state)
				2'b00 : jump_state <= jump ? 2'b01 : jump_state;
				2'b01 : jump_state <= jump ? 2'b11 : 2'b00;
				2'b11 : jump_state <= jump ? 2'b11 : 2'b00;
				default : jump_state <= 2'b00;
			endcase

			if(~jump_hold)
	// if the player is not jumping
				player_ycoord <= player_ycoord + 4;		// gravity
			if(jump_hold) begin
	// if the player is jumping
				player_ycoord <= player_ycoord - 4;
				jump_length <= jump_length - 1;
				in_the_air <= 1;
			end
			if(new_jump && ~in_the_air) begin				// if the player has
requested a jump
				jump_length <= 6'b111000;
				player_ycoord <= player_ycoord - 1;		//get the player started off the
ground
			end
			if(forward)
		// if the player has requested to go foward
				player_xcoord <= player_xcoord + 4;
			if(backward && ~(player_xcoord < 4))		// if the player requests to go backward and
is not at the edge of the screen
				player_xcoord <= player_xcoord - 4;
			end // player controls
	end //always
endmodule


//////////////////////////////////////////////////////////////
//
// sprite_generator
// description: draws the sprites from the sprite RAM, determines
// collisions, and updates the collision state
//
//////////////////////////////////////////////////////////////


module sprite_generator(
	input vclock,					//65mhz clock
	input [9:0] vcount,				//vertical count
	input [10:0] hcount,			//horizontal count
	input reset,					//global reset
	input [11:0] left_edge,
	//sprite RAM
	input [11:0] xcoord_i,			// changing the x coordinate of the sprite
	input [7:0] ycoord_i,			// changing the y coordinate of the sprite
	input [10:0] sstate_i,			// changing the sprite's state
	input [4:0] sprite_tile_i, // 32 different sprite tiles
	//tile memory
	input [5:0] sprite_tile_pixel,		// RGB value of the pixel requested
	input [2:0] sprite_pixel_type,		// character type of pixel requested
	input sprite_pixel_transparent,		// high if the pixel is transparent
	//Frame Buffer
	input [2:0] fb_pixel_data,		// pixel characteristics
	input [3:0] fb_sprite_number,
	//player collision detection
	input [10:0] player_xcoord,		// Draw Player Sprite
	input [9:0] player_ycoord,		// At the correct coordinates
	input [3:0] player_collision_sprite_number,
	input player_collision_i,			//high if the player collided with a sprite
	input [2:0] player_collision_type,
	//sprite RAM
	output [11:0] xcoord_o,
	output [7:0] ycoord_o,
	output [10:0] sstate_o,
	output [4:0] sprite_tile_o,
	output [3:0] sprite_number_o,
	output reg ram_we,
	//tile memory
	output [5:0] sprite_tile_index,
```

```
        output reg [3:0] sprite_tile_row,
        output reg [3:0] sprite_tile_column,
        //Frame Buffer
        output [7:0] sprite_row,                        // horizontal row to write sprite data (0-255)
        output [7:0] sprite_column,          // verticle column to write sprite data(0-239)
        output [12:0] sprite_data,           // sprite pixel data to be saved to the memory
        output sprite_we,                                      // sprite write enable
        output other_sprite_collision,
        output reg [3:0] sprite_state,
        output [2:0] new_sstate);

        reg [3:0] sprite_number;

        //internal signals
        wire [11:0] sprite_column_i;
        wire next_horizontal_direction;      // which way the current sprite should go next
        wire next_move_vertical;                        // high if moving vertical
        wire vertical_collision;
        reg [7:0] ycoord_hold;
        reg [11:0] xcoord_hold;
        reg [10:0] sstate_hold;
        reg [2:0] pixel_data_hold;
        reg [2:0] sprite_pixel_type_hold;
        reg [4:0] sprite_tile_hold;
        wire hor_dir;
        reg [3:0] fb_sprite_number_hold;
        reg [11:0] colliding_sprite_xcoord;
        reg [7:0] colliding_sprite_ycoord;
        reg [10:0] colliding_sprite_sstate;
        reg [4:0] colliding_sprite_tile;
        reg [5:0] sprite_tile_pixel_hold;
        reg sprite_collision_hold;
        wire player_collision;

        // parameters for the state machine
        parameter s_start = 0;
        parameter s_1 = 1; // Query sprite info
        parameter s_2 = 2; // Info obtained
        parameter s_3 = 3; // Write Information to the RAM
        parameter s_4 = 4; // Increment the counter
        parameter s_5 = 5; // New Sprite is created
        parameter s_6 = 6; // Information about sprite held
        parameter s_7 = 7; // Write this info to RAM
        parameter s_8 = 8; // If we have no RAM space, return to start
        parameter s_9 = 9; // Collision Calculations
        parameter s_10 = 10;
        parameter s_11 = 11;
        // output logic

        // Event signals based on pixel metadata
        // detect if collided with an object
        assign hor_dir = sstate_hold[2];
        // detect if collided with another sprite
        assign other_sprite_collision = (~sprite_pixel_transparent && ((pixel_data_hold == 3'b101) || (pixel_data_hold
== 3'b100)) && ~(sprite_number == fb_sprite_number_hold));
        // detect if collided with an object
        assign next_horizontal_direction = sprite_collision_hold ? ~hor_dir :(~sprite_pixel_transparent &&
(pixel_data_hold == 3'b010)) ? 0 : (~sprite_pixel_transparent && (pixel_data_hold == 3'b001)) ? 1 : hor_dir;
        // change direction if the sprite hits something
        assign vertical_collision = (~(sprite_pixel_transparent) &&      ((pixel_data_hold == 3'b011) ||
        ((pixel_data_hold == 3'b101) && ~(fb_sprite_number == sprite_number))));
        //which way do we move next?
        assign next_move_vertical = vertical_collision ? 0 : sstate_hold[0];
        //determine the pixel type of the sprite pixel that hit the player last
        assign new_sstate = player_collision ? player_collision_type : sstate_hold[10:8];
        // has the player collided with this sprite?
        assign player_collision = (player_collision_i && (player_collision_sprite_number == sprite_number));


        // output to frame_buffer
        assign sprite_column_i = (xcoord_hold - left_edge + sprite_tile_column);   //* normalize to the screen
        assign sprite_column = sprite_column_i[7:0];
                //add in the tile column
        assign sprite_row = ycoord_hold + sprite_tile_row;
        assign sprite_data = {sprite_number,sprite_tile_pixel_hold,sprite_pixel_type_hold}; //will only change the frame
buffer when not    transparent (taken care of in the FSM)
        assign sprite_we = (sprite_column_i > 256) ? 0 : (sprite_state == s_4);

        // output to tile_memory
```

```
	assign sprite_tile_index = {1'b1,sprite_tile_hold};
	//append a 1, sprites are only in the top half of the tile memory

	// output to sprite_RAM
	// if in state 9, use the data for the sprite we are colliding with, otherwise use the data
	// for the sprite we are currently working with
	assign sstate_o =  (sprite_state == s_9) ?

	{colliding_sprite_sstate[10:3],~colliding_sprite_sstate[2],colliding_sprite_sstate[1:0]} :

	{new_sstate,sstate_hold[7:5],player_collision,sstate_hold[3],next_horizontal_direction,
						sstate_hold[1],next_move_vertical};
	assign xcoord_o =  ((sprite_state == s_9) && (colliding_sprite_sstate[2])) ? (colliding_sprite_xcoord - 1) :
						((sprite_state == s_9) && ~(colliding_sprite_sstate[2])) ?
	(colliding_sprite_xcoord + 1) :
						(sprite_collision_hold && next_horizontal_direction) ?
	(xcoord_hold + 1) :
						(sprite_collision_hold && next_horizontal_direction) ?
	(xcoord_hold - 1) :
						xcoord_hold;
	assign ycoord_o =  (sprite_state == s_9) ? colliding_sprite_ycoord :
						(vertical_collision && sstate_hold[3]) ? (ycoord_hold + 1) :
	// if hit something going up, move down
						(vertical_collision && ~sstate_hold[3]) ? (ycoord_hold - 1) :
	// if hit the bottom, move up
						ycoord_hold;
							// don't chnge
	assign sprite_tile_o =      (sprite_state == s_9) ? colliding_sprite_tile :
							(xcoord_hold > (272 + left_edge)) ? 0 :
							sprite_tile_hold;
	assign sprite_number_o = ( (sprite_state == s_9) ||(sprite_state == s_7) || (sprite_state == s_8))
								? fb_sprite_number_hold     : sprite_number;

	// next state logic
	always @ (posedge vclock) begin
		if (reset) begin
			sprite_state <= s_start;
			{sprite_number,sprite_tile_row,sprite_tile_column} <= 0;
			ycoord_hold <= 0;
			pixel_data_hold <= 0;
			ram_we <= 0;
			sprite_collision_hold <= 0;
			colliding_sprite_xcoord <= 0;
			colliding_sprite_ycoord <= 0;
			colliding_sprite_sstate <= 0;
			colliding_sprite_tile <= 0;
		end
		else case (sprite_state)
			s_start : sprite_state <= ((vcount == 780) && (hcount == 1)) ? s_1        : sprite_state;
			s_1 : sprite_state <= s_6;  //fetch data
			s_6 : sprite_state <= s_10;
			s_10 : sprite_state <= 2;
			s_2 : sprite_state <= other_sprite_collision ? s_7 : (~(fb_pixel_data == 3'b000) &&
	~(sprite_pixel_transparent)) ? s_3 : ((fb_pixel_data == 3'b000) &&        ~(sprite_pixel_transparent)) ? s_4
	: (sprite_pixel_transparent) ? s_5 : sprite_state;
			s_7 : sprite_state <= s_8;
			s_8 : sprite_state <= s_9; //reading colliding sprite data
			s_9 : sprite_state <= s_3; //writing colliding sprite data
			s_3 : sprite_state <= (~sprite_pixel_transparent) ? s_4 : sprite_state; // write data
			s_4 : sprite_state <= s_5; //gives time to write to the frame buffer
			s_5 : sprite_state <= (&{sprite_number,sprite_tile_row,sprite_tile_column}) ? s_start : s_1;
			default : sprite_state <= s_start;
		endcase


		if (sprite_state == s_5) begin
			{sprite_number,sprite_tile_row,sprite_tile_column} <=
{sprite_number,sprite_tile_row,sprite_tile_column} + 1;
		end
		// registering a lot of wires for comparison
		if (sprite_state == s_8) begin
			colliding_sprite_xcoord <= xcoord_i;
			colliding_sprite_ycoord <= ycoord_i;
			colliding_sprite_sstate <= sstate_i;
			colliding_sprite_tile <= sprite_tile_i;
			ram_we <= 1;
		end
		if (sprite_state == s_6) begin
			ycoord_hold <= ycoord_i;
```

```
                        xcoord_hold <= xcoord_i;
                        sprite_tile_hold <= sprite_tile_i;
                        sstate_hold <= sstate_i;
                end
                if (sprite_state == s_2) begin
                        if(~(fb_pixel_data == 3'b000) && ~(sprite_pixel_transparent) && ~other_sprite_collision)
                                ram_we <= 1;
                        pixel_data_hold <= fb_pixel_data;                                       //from frame
buffer
                        fb_sprite_number_hold <= fb_sprite_number;                      //from frame buffer
                        sprite_pixel_type_hold <= sprite_pixel_type;            //from tile mem
                        sprite_tile_pixel_hold <= sprite_tile_pixel;            //from tile mem
                        sprite_collision_hold <= other_sprite_collision;        //uses fb_pixel_data and
fb_sprite_number
                end
                if (sprite_state == s_3) begin
                        ram_we <= 0;
                end
        end //always
endmodule

//////////////////////////////////////////
//
// sprite_ram
// instantiate the BRAM
//
///////////////////////////////////////////
module sprite_ram(input vclock,
                                        input [3:0] sg_address,
                                        input [35:0] fsm_address,
                                        input [35:0] sg_data_i,
                                        input [35:0] fsm_data_i,
                                        output [35:0] sg_data_o,
                                        output [35:0] fsm_data_o,
                                        input sg_we,fsm_we);
spriteram                               SRAM(    .addra(sg_address),
                                                                        .addrb(fsm_address),
                                                                        .clka(vclock),
                                                                        .clkb(vclock),
                                                                        .dina(sg_data_i),
                                                                        .dinb(fsm_data_i),
                                                                        .douta(sg_data_o),
                                                                        .doutb(fsm_data_o),
                                                                        .wea(sg_we),
                                                                        .web(fsm_we)
                                                                        );
endmodule

//////////////////////////////////////////////////////////
//
// tile_memory
// description: This module stores 64 tiles of 16 x 16
// pixels, it has two ports, one for the
// background module, and one for the sprite
// module
//
// Module works. We can grab arbitrary tiles from ROM
// And display them on the screen.
//////////////////////////////////////////////////////////


module tile_memory(         input vclock, // 65 MHz clock
                                        input reset, // global reset
                                        input [5:0] back_tile_index,
                                        input [3:0] back_tile_row,
                                        input [3:0] back_tile_column,
                                        input [5:0] sprite_tile_index,
                                        input [3:0] sprite_tile_row,
                                        input [3:0] sprite_tile_column,
                                        output [5:0] back_tile_pixel,
                                        output [2:0] back_pixel_type,
                                        output back_pixel_transparent,
                                        output [5:0] sprite_tile_pixel,
                                        output [2:0] sprite_pixel_type,
                                        output sprite_pixel_transparent);
// initializing the BRAM
tile_rom                    tilemem(

        .addra({back_tile_index,back_tile_row,back_tile_column}),
```

```
          .addrb({sprite_tile_index,sprite_tile_row,sprite_tile_column}),
                                                                  .clka(vclock),
                                                                  .clkb(vclock),

          .douta({back_tile_pixel,back_pixel_type,back_pixel_transparent}),

          .doutb({sprite_tile_pixel,sprite_pixel_type,sprite_pixel_transparent})
                                                                  );
endmodule
```