Super Digital Bros.

The 6.111 Workout Plan

Written in partial fulfillment of the 6.111 CI-M requirements.

Greg Luthman, Akash Shah

12/13/06

Abstract

The game controller implemented in this project combines complex video circuitry with a full-scale adventure game to create an incredibly interactive user experience. Unlike most games that must be played with a controller in hand and show only the persona of the game character, the logic implemented in this project allows for a truly unique package. By immersing the user in the game world and using the user's gestures made on camera to allow him or her to interact with the elements and characters within the game world, the experience is unlike any other.

Table of Contents

List of Tables

List of Figures

Overview

General Overview

Current generations generally grow up with video games as a large part of their entertainment. Perhaps the most popular of these games has been Super Mario Bros., a side-scrolling adventure game for the Nintendo Entertainment System (NES). As children one could become enveloped in the game world and imagine that we were the main character on screen fighting to save the princess from terrible monsters. This project aimed to create a live action, side scrolling adventure game, similar in concept to Super Mario Bros. Instead of playing with a controller and seeing a character move on screen, the game is controlled by the actions a player makes in front of a camera. Not only is video of the player used to determine the proper commands to send to the game, but it is also placed into the game world, giving the player the sense of immersion into the game. The player is able to duck, jump over objects, move forward or backward in the game world.

Gameplay

The game circuitry creates a game world for the user to explore as well as manages interaction between the character and the game world with the help of gesture information sent from the video circuitry. Players are able to kill enemies, jump onto objects, and explore the game world among a variety of other activities. When reaching the end of the level, the character wins the game. However, if an enemy runs into the character, the character loses the game and the game ends.

Gameplay from the user's standpoint is simple and intuitive. The user's game world character mimics the character's actions as seen by the video circuitry. For instance, if the character jumps in the air, the character will also jump inside the game world. Additionally, the background of the video image is filtered out, to allow only the image of the user to appear on the screen.

Thus, the goal of the game for the user is to reach the flagpole area at the end of the level. This will indicate that the user has won the game. If the user is to run into an enemy on the way, the user's turn will end, signaling that the character has died. Along the way, the character should make an effort to rid the game world of as many enemies as possible.

Operation

Once the labkit is programmed with the generated bit file, the first step is for the user to aim the video camera to the blue background screen. Once this is done, the user must hit the '1' button on the labkit. Doing so will allow the video logic to calibrate itself to the background color. Next, the user should place the camera in a position that will allow the camera to capture the user's entire body.

Now, the user can stand in front of the background and begin to play the game. In order to send a jump signal to the game, the user must be on the ground (since people cannot jump again while in the air). In order to move forward, the user should try to move towards the front half of the background area, a move that will allow the character to proceed forwards through the game. Analogously, the user should move towards the back of the window.

If the user wishes to reset the game, the user should press the 'Enter' button on the labkit. This will reset the system and allow the user to run through the game once again.

Design and Implementation

Video Subsystem

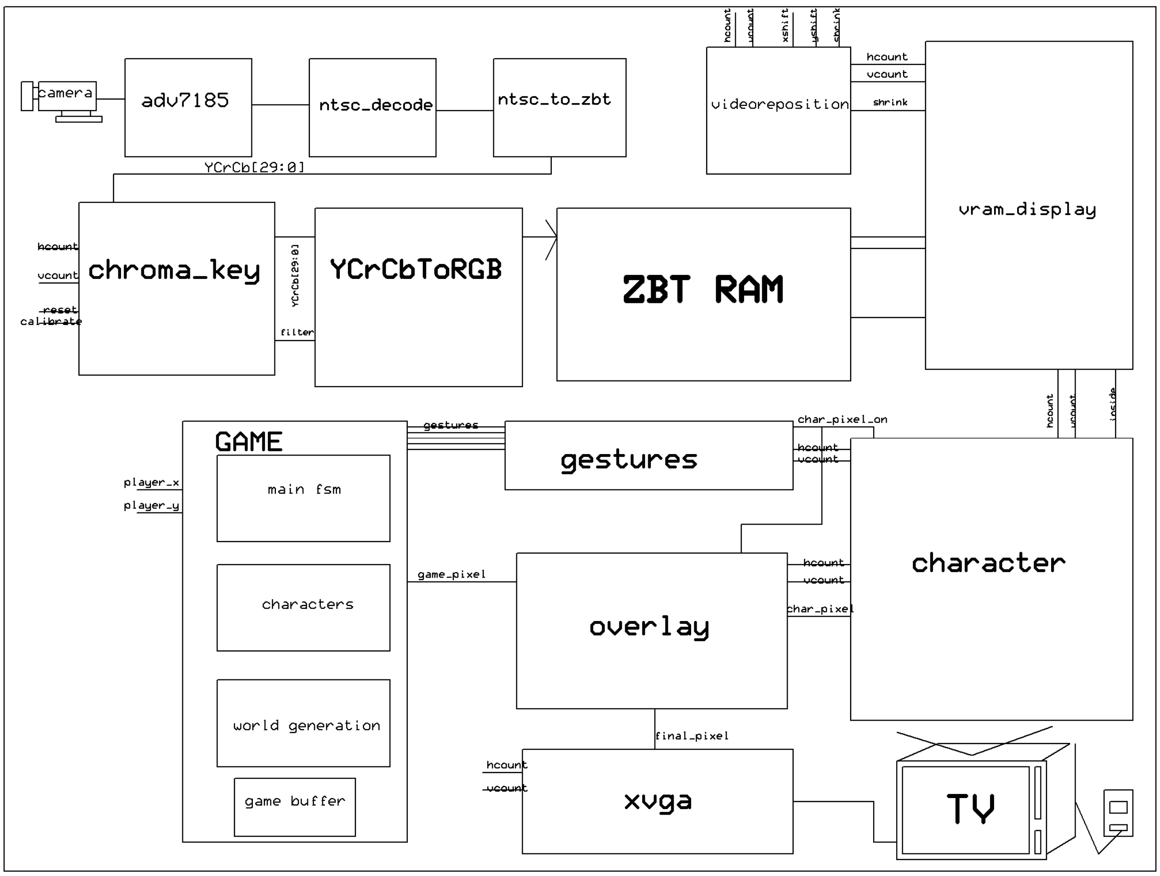


Figure 1 - Video Block Diagram

The video logic handles input from the camera and makes sure it is properly interacting and overlaid with the game world. Incoming video is filtered for background color, converted to RGB space, stored in a buffer, then displayed in a manner that allows translation around the video plane, as well as recognition of the game character as well as any gestures the character makes in an effort to interact with the game world.

Adv7185init

This module initializes the adv7185 interface that decodes the incoming video signal.

NTSC Decode

The NTSC decoding module take the signal coming in from the adv7185 chip on the labkit and converts decodes the information in a way that complies with the NTSC standard that is used for video in North America. The module essentially grabs pixel and YCrCb information for the video from the serial stream of information that is coming in.

NTSC to ZBT

Once the NTSC signal is decoded from the information coming in from the video A/D converter, this information needs to be stored in a frame buffer in order to meet timing requirements of the 65mhz VGA video clock. This then prepares the NTSC information to be loaded into the ZBT ram for video display. Since the ZBT is 36-bits wide, our implementation stores four blocks of information in the two ZBT rams to be able to store a good amount of color information.

Chroma Key

The chroma key module takes in the incoming video feed and tries to key on a certain chroma blue value in order to achieve a ‘blue screen’ effect (like on the local weather). It takes in a calibration signal, and when this signal is set high, the current chroma blue value is stored in a register.

If the calibration signal is not high, the module then checks the data for the incoming pixel. If this pixel has a chroma blue value which falls within a specified tolerance of the value stored in the register, the module will then indicate that this pixel should be filtered out. This information is then used by the integrating module to filter out the pixel data. The luminance and chroma values are turned off before this pixel is stored in the ZBT memory.

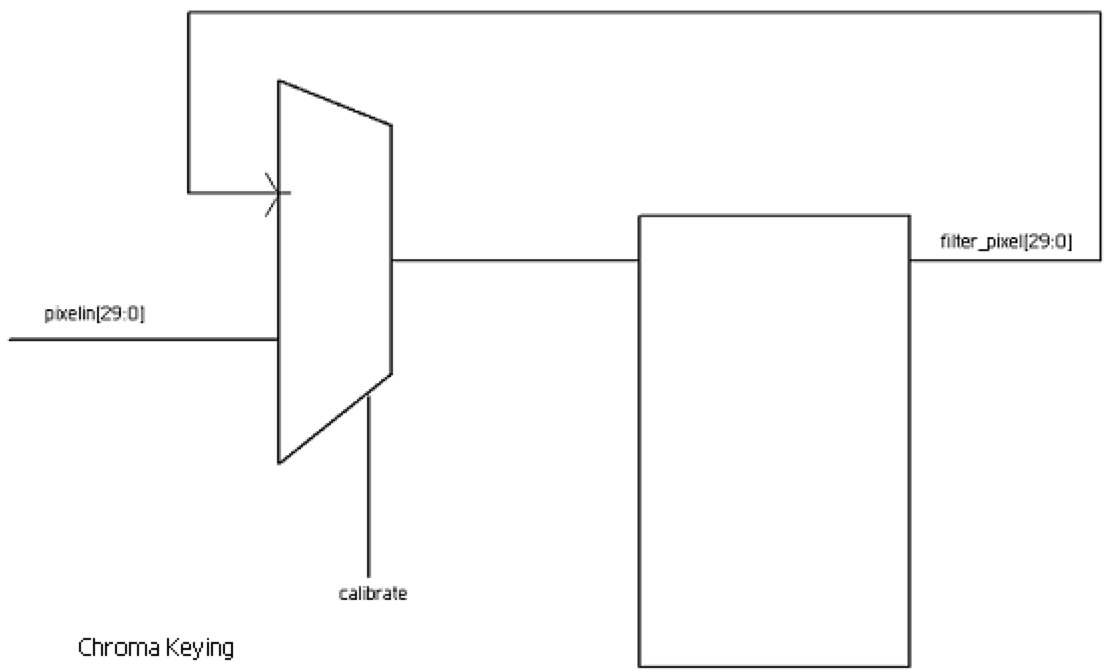


Figure 2 - Chroma key block diagram

The chroma keying logic was originally set up to key off of RGB values once these were read from the ZBT memory. At this point, the parameter allowing for tolerance was adjustable by the user through use of the directional input buttons. A lowpass filter was also used (see below) to filter out noise coming from the video or noise that occurred because of timing or read errors from the ZBT. This allowed for effective keying, but was not entirely reliable as this method was heavily dependent on lighting conditions.

Thus, the chroma keying logic was changed to key on the YCrCb values before they are stored in the ZBT ram. Keying on this set of values allows for cleaner keying since the luminance can simply be ignored, taking care of the issue of having perfect lighting.

YCrCbtoRGB

This module provides color conversion that must be used by the circuitry to change the incoming YCrCb values to a set of RGB values that can be used in the VGA processing. The module uses color conversion formulas to perform the transformation to the RGB space. While the module involves simple arithmetic operations, these operations take time. Therefore, the module requires a three-stage pipeline to meet timing requirements for the video processing.

In addition to the color conversion, the module contains additional logic that checks the incoming YCrCb values to see if they have been filtered out. If the luminance value has been killed to zero, the module will assume this means the video logic desires a pure black color and will then set the outgoing RGB value to all zeros. This then blacks out the pixel.

Multiple sets of registers are used for these calculations since several pipeline stages were necessary. Once values pass through the final stage of the pipeline, they are then sent to a multiplexor that limits the RGB output to a 8-bit unsigned value for each of the three channels.

Video Reposition

This module contains logic that allows the game logic to shift the video window around the display as desired. As part of this shifting logic, the module also creates a bounding box around the video. This bounding box is later used by the gestures, character, and overlay modules to determine when a pixel corresponds to the video feed. The module also determines the grid lines that are used by the gesture recognition module.

The video repositioning module allows for translations of the video window. Accounting for scaling effects as well based on the 'shrink' signal, it creates two bounding boxes, one for the normal video size and one for the state when the video must be shrunk to half its original size. The logic necessary to calculate this box involves simple arithmetic, but it must be pipelined in order to meet timing specifications. In the end, if a given pixel lies within this bounding box, the 'inside' signal will be set high by the module.

Meanwhile, the module also creates a grid that is used by the gestures module. While this involves the same level of calculation as the bounding box, these grid values are only dependent on the shift parameters, and not the current hcount, vcount values as the 'inside' signal is. However, this abstraction layer provides a great deal of convenience when dealing with the video position within other modules.

Calculation of the bounding box is accomplished by taking in the current hcount, vcount values as well as the horizontal and vertical shift constants. This information is then factored in with a set of defined constants that one added or subtracted from will indicate if a current hcount, vcount combination lies within the bounding box. Grid lines are calculated in a similar fashion, using hard-coded parameters along with the horizontal and vertical shift values. Once these values are done with addition or subtraction, values are then fed through a multiplexor that will select which set of calculated values to use based on which state the video window is in, normal or shrunken. As mentioned earlier, these operations are broken down into several pipelined stages to improve efficiency.

Vram Display

This module fetches data from the two ZBT memories in chunks of four pixels (the same way they were stored in the ZBTs). Based on the hcount and vcount values, we can determine the address that data must be read out of the ZBT. Every four cycles, a new set of data is read from the ZBT memories.

This module contains modifications over the sample Vram display module that have a significant impact on what is both read from the ZBT and output to the screen. The module contains logic to account for resizing the video window. Based on the value of the 'shrink' signal, the module uses multiplexors to decide what data to read from the ZBT. If the picture is to be shrunk, the highest-order bit of the hcount values is ignored, and the hcount is shifted left by one, and padded with a '1' on the right instead of the zero's as usual. This padding of 1's is to account for the way the information is read from the ZBT. If the video should not be shrunk, the module proceeds as usual to fetch all of the information from the ZBT.

However, in the case of the shrunken video, there are still some more complications. For instance, the module does not modify the values for the location to which a given chunk of information belongs. Instead, the module essentially interlaces video horizontally now. Every other line is filled in when hcount[9] is low and when it is high, the rest of the data is filled in.

In addition to these modifications for hcount, vcount values are shifted left one bit when passed into the module as well when the video is needs to be shrunk. While only a usual shift in this instance, this change allows us to skip over every line vertically (so video can be shrunk both vertically and horizontally).

Character

This character module checks the incoming pixel for its value. If the value has been blacked out (based on the logic in the chroma key module), the module will then determine that the given pixel corresponds to the background of the video image. If the value has not been blacked out, the module then checks to see if the given pixel lies within the video window. If the pixel does not belong to the background and it lies within the video window, the module will then indicate that the pixel belongs to the player character. Although this module contains fairly simple logic, this abstraction helps a great deal with other modules, notably with collision detection and gesture recognition.

This module contains mostly combinational logic that goes through the process of determining what the given pixel belongs to. However, more complicated logic may be necessary if we were to expand the requirements of a pixel to be classified as belonging to a character (having a flesh tone, for instance). In this case, it is also likely that the logic would need to be pipelined to meet timing constraints.

Gestures

This module contains the logic necessary to recognize player gestures based on the information coming in from the video feed. The driver behind the gesture recognition is the center of mass calculations for the player. Based on the character_pixel_on signal being high, the gestures module will then account for a given pixel as belonging to the game character. All hcount and vcount values are summed up in registers, as well as the number of pixels that were recorded. Finally, when the 'newframe' signal goes high, the divider calculates the new center of mass and the value that is calculated is stored in registers that track the x and y coordinates of the center of mass. All this is done with the help of two instances of the pipelined divider module generated by the Xilinx tools. These dividers, however, cause a problem for the Xilinx tools during compile time since signals must be routed around these, but must still be close enough together to meet the requirements for the 65mHz VGA clock.

Once these x and y coordinates for the center of mass have been calculated, the module has more logic that determines where this center lies. Based on grid lines calculated by the video reposition module, the gestures module will determine if the character is jumping, ducking, moving forward, moving backwards, or standing still. These signals are then fed to the game logic to determine how the character will interact with the game world.

Overlay

The overlay module contains logic that determines which VGA pixel to output to the screen. The combined video and game circuitry contains two sets of pixels as well as two outgoing video feeds. Thus, the overlay module is responsible for doing what its name says and overlaying the two.

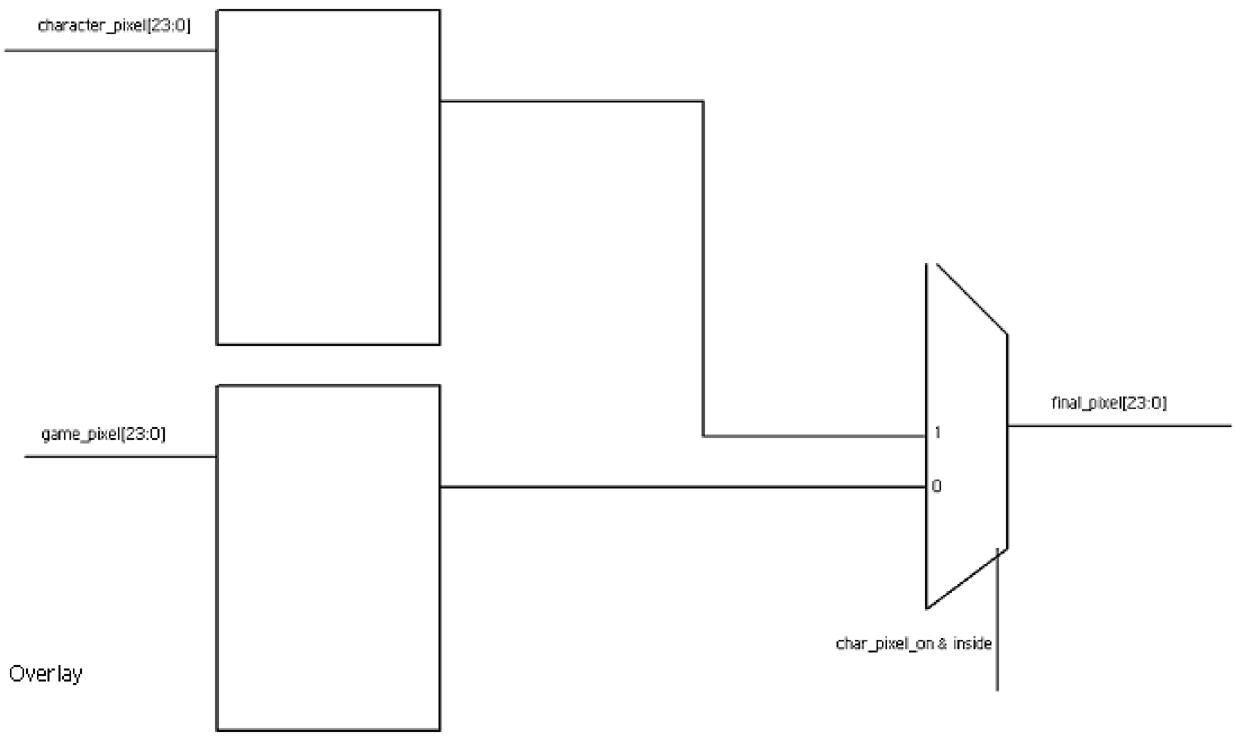


Figure 3 - Overlay Block Diagram

The module will first check if a given pixel falls within the video window. If it does not, the pixel is automatically defaulted to the game world video feed. If the pixel falls within the window, then the module checks to see whether the pixel belongs to the player or not. Since the player image should be on top at all times, the pixels belonging to the player are given priority. If, however, the pixel belongs to the background image, the pixel will be replaced with the game world data (or replaced, rather, since the data for background pixels has already been killed off at this point).

XVGA

The visual output for the project was done using VGA displayed at a 1024 pixel by 768 pixel resolution on an LCD computer monitor. The VGA standard uses interlaced video that is output to the monitor. Each pixel of the image is defined by a set of values for the red, green, and blue channels that are used by the VGA encoder, each channel using 8-bit values. In addition to the pixel values, which compose the active regions of the frame, each frame also includes blanking regions. The blanking intervals occur at the end of every line and at the end of every frame. These blanking regions are actually incredibly useful for our logic considering that this time allows the logic to fill the ZBT and BRAM buffers without having to worry about being able to output a proper image to the screen.

Video wrapper

The module responsible for wrapping all the video modules together also handles integration with the game logic. All instances of the video modules are declared here. First, the video is decoded from the NTSC signal, stored in the ZBT, then read out from the ZBT memory once the time is right. This information is then filtered and processed by the chroma keying, character, and gesture recognition modules. The game wrapper module is also instantiated within this video wrapper. Finally the pixels from the live video world and game world are sent to the overlay module that determines the final pixel that should be displayed by the XVGA module.

This wrapper also contains logic in addition to what is found in the modules it instantiates. User input, for instance is handled within the wrapper. In the case of translations of the video window coming in from the game logic, certain pipelined calculations are performed to get the video shifted in the proper position, regardless of which state (large or small) the video window is set to be in. Additionally, luminance values are killed off within this module after data passes through the chroma keying stage. ZBT memories and video decoders/encoders are also initialized here.

Game Subsystem

The game subsystem is responsible for the generation of the world in which the player interacts. It takes from the video subsystem, a vcount and hcount corresponding to which pixel on the screen is currently being displayed. The video subsystem sends a signal to let the game subsystem know if the player occupied the portion of the screen denoted by the current vcount and hcount. The video subsystem also sends signals relating to various player actions. As of now, the only actions that are used in the game subsystem are: player moving forward, player moving backward, and player jumping. The game modules calculate the position of all objects in the game world based on these action signals and then outputs the x and y coordinates of where the video of the player should be placed on the screen. Working with the video subsystem, the game subsystem creates the effect of the player being inside of the game world.

As with the video subsystem, the game subsystem is implemented using the 6.111 Labkit. More specifically, the game is designed with the Verilog Hardware Description Language and is loaded onto the Xilinx Vertex-2 Field Programmable Gate Array. No other hardware is needed for the game subsystem, as all memories used are part of the Xilinx chip.

The game subsystem is capable of implementing any type of side scrolling adventure game, but because of nostalgic and familiarity reasons, a simplified version of the early Nintendo game, Super Mario Brothers was chosen. The goal of this simplified game is to navigate the game world to the victory flag pole at the end of the level, avoiding both enemies and obstacles. The player may jump onto blocks, pipes, and other objects to avoid enemies, or the player can jump on top of the enemy to defeat the enemy. If the player either touches an enemy other than jumping on it, or falls down a hole to the bottom of the screen, the player loses.

Design Overview

The Game subsystem is designed to be very modular, allowing for easy upgrades or changes to meet a new design requirement. The method of displaying and manipulating images on the screen was also designed to maximize the flexibility of the game. A frame buffer is used to display the images, while two image generator modules provide pixel information to the frame buffer. The information on which the two image generators operate is provided by a number of memories, both read only and random access. Finally a player control module and a sprite behavior finite state machine update the RAM and other values to move things around on the screen. All these modules are then packaged together in a “game wrapper” to abstract the inner workings from the video portions of the project. The only signals that should be available outside this wrapper are those concerned with displaying the game world or signaling what the player does.

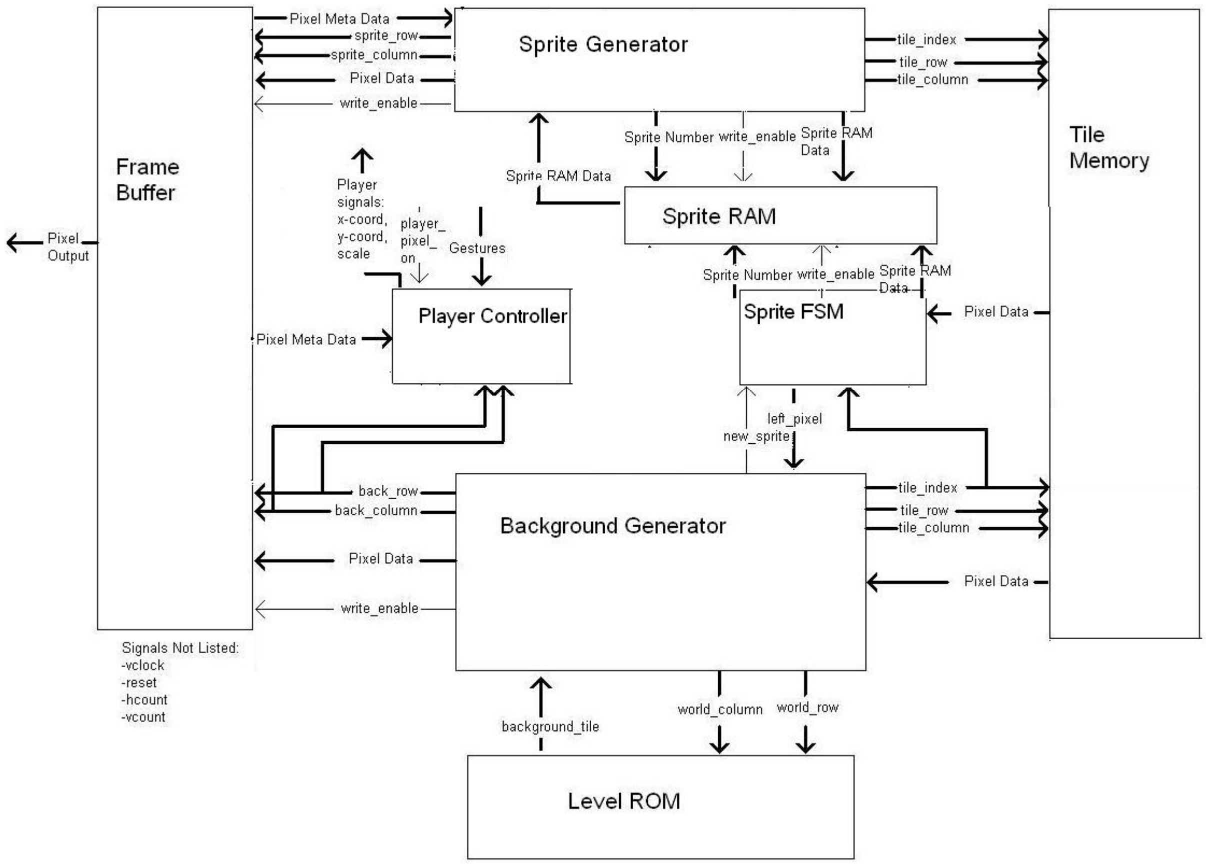


Figure 4 - Game Logic Block Diagram

Frame Buffer

One of the most important, and one of the first, modules to be designed is the Frame Buffer. There are many advantages of using a frame buffer instead of combinational logic to display images. The VGA updates the screen at a frequency of 65 MHz. This gives any combinational driven display logic only 15 nanoseconds to not only determine what color to display based on the vcount and hcount, but also whether a collision has occurred. That is not to mention the signals that must be sent to other modules when a collision has occurred. How a frame buffer gets around the problem of a short update period is to let other modules make decisions about what to display even while the vcount and hcount do not yet correspond to the pixel being calculated.

The main part of the frame buffer is a 256 x 240 x 13 RAM. Each of the 256 x 240 memory locations contains 13 bits of data relating to a corresponding pixel. Because the resolution of the game image is determined by the number of pixels that can be stored, the resolution is 240 pixels high by 256 pixels wide. This decision was based on the available amount of on-chip BRAM, the number of clock cycles available to update the pixels, and the fact that the original Nintendo also displayed in 240 x 256 resolution. A memory location is addressed with the row number as the high order bits, and the column number as the low order bits.

The screen that the game is outputted on is of a higher resolution than the frame buffer resolution. The higher resolution is 1024 x 768. The conversion of stretching low resolution images to fit the high resolution screen is accomplished by having multiple pixels of the high resolution map directly to just one pixel of the low resolution image. Going from 1024 to 256 is not difficult, it is merely a division by four. (Or equivalently shifting by 2 bits.) Going from 768 to 240 is a bit harder as 240 must be multiplied by 3 to maximize the area of display. This means that the lines below 720 must be turned off. The main difficulty is that a simple bit shift cannot perform the desired division. Therefore a count is kept that only increments every three clock periods and reset at every refresh of the screen. The BRAM memory is then addressed using the bit shifted column data and the counted by three row data. The result is that a low resolution pixel is displayed as a 3 pixel high and 4 pixel wide high resolution image.

As there are two image generators writing to the frame buffer, the BRAM was created as a dual port BRAM. Even though the generators don't write at the same time, creating the BRAM as a dual port BRAM saves time, effort, and potential bugs. Even though the BRAM was chosen to have two ports, the sprite generator's port had to be shared with the hcount and vcount used to pick which pixel is currently to be displayed. To accomplish this, the address to be used is chosen based on the current hcount and vcount. If the vcount is above 720 (ie. Displaying the image) then the average created by the two counts takes precedence. Otherwise priority is given to an address sent from the sprite generator. Even though the color is turned off after the vcount is above 720, the output data is not turned off. This data is used by the sprite generator to determine collisions between different objects in the game.

Tile ROM

Before the background generator and the sprite generator can be described, the concept of a tile and a tile read only memory (ROM) must be explained. It is easier to think of objects and enemies as a single entity than a collection of pixels that must move together. With that reasoning, images are restricted to be 16 pixels high and 16 pixels wide and stored in a BRAM within the tile ROM. The tile ROM is also a dual port BRAM so that both background and images can be accessed easily.

The tile ROM stores not only the color, but also the data type of each pixel in a tile. The operation of both the color and data types will be explained in a following section. The two image generators can then access each pixel within a tile image one at a time to send to the Frame Buffer. The addressing scheme of the BRAM has the sprite tile number (0 - 63) as the high order bits, followed by the tile's row, and then the tile's column.

Background ROM

Along the same lines as the tile ROM, the background ROM helps abstract the layout of the game world. The background ROM is only a single port block RAM because only one module (the background generator) needs to read from it. The ROM is indexed by a world row and a world column. The world row can be from 0 to 14, with row 14 being closest to the bottom of the screen. The game world column can range from 0 to 256, with column 0 being at the far left. Each coordinate of the world row and world column stands for a 6 bit tile index. Using the tile index from the tile ROM, a game world can be created using just index numbers. Based on the input row and column, the Background ROM module will return a 6 bit tile index.

Background Generator

The background generator is responsible for writing all the pixel information of the currently displayed screen to the frame buffer. Because the the game world is 256 tiles long and the screen is only 16 tiles long, the Background generator must be able to scroll based on a position number provided by another module in the game subsystem. This number, left_pixel, is a 12 bit number based on the pixel length of the game world. (256 tiles long x 16 pixels per tile) It is the position in the game world of the far left pixels on the screen. Left_pixel can be incremented in units less than 16, so that the screen will scroll smoothly from one tile to another, rather than abruptly scrolling each tile over 16 pixels at a time. To create the smooth scrolling effect, a system had to be created so that only portions of tiles would be displayed at either end of the screen.

To start the background generator, the vcount must be past 720, as not to cause any flickering or glitching in the output caused by updating the frame buffer while it is being read out to the screen. The generator then cycles through each pixel across row 0 (the top of the screen), getting the correct information and writing it into the frame buffer. After completing the row, the count moves to the next row and so forth until all pixels in the frame buffer are updated, at which point the module waits until the next time the vcount is past 720.

Retrieving the pixel color and data to write into the frame buffer is a multi stage effort. The first thing that must be determined is what tile should be displayed at the current pixel. This is read from the Background ROM using the top order bits of the current row as the world row. Determining the world column is more difficult, because the world row depends on how far in the game world the pixel is, while the pixel count inside the background generator only covers the current screen. The world column is calculated by adding the left_pixel count to the current pixel's column, then using only the high order bits.

The resulting tile index number from the background ROM is used with the lower 4 bits of the pixel's row, and the the lower 4 bits of the summation of the left_pixel count and the current pixel's column. This results in the pixel color and pixel data of whichever tile is supposed to be placed at the current pixel's location.

One issue with accessing three memories for each pixel (the background ROM, the tile ROM, and the frame buffer) is pipelining. If the background generator would naively run through each pixel, waiting for it's tile and pixel information to be retrieved, then writing that data to the frame buffer, the process of updating all the pixels would take quite a while. 256 columns x 240 rows x 3 cycles for the memories. That number can be divided by 3 by pipelining all the stages. Placing registers in between the various memory stages can hold the values of the old pixel while the next pixel is starting to be processed. In that way the background generator can run through one pixel every clock cycle as compared to the three clock cycles the generator would take without pipelining. Generating the background turns out to be the longest operation to perform during frame refreshes, and without pipelining, it would be impossible to write everything to the frame buffer in time for the next refresh.

The last operation that the background generator must perform is to inform the sprite generator each time a new sprite has entered the screen. A sprite is a movable object such as an enemy, and a sprite cannot be controlled by the background generator because the background generator has no way of moving a tile withing the game world. A convention that is created and used for this game is that sprites are the top 32 tiles of the tile memory, while the background tiles are the lower 32 tiles of the tile memory. Therefore the background generator checks to see if the top bit of the tile index number retrieved from the background ROM is high, signaling a sprite. The generator also checks to see if the tile row and tile column are at the top left hand corner of the sprite so that a new sprite is only signaled once for every sprite tile in the game world. A register, “last_world_column_updated” must be be consulted as well, before a new sprite is signaled. This register is updated each time a new row has been drawn into frame buffer. The generator must check this register so that if the player stops at a certain place in the game world, the new sprite is only signaled once, and not every time the background cycles through the sprite. (Something that would happen quite often when the screen is refreshed 60 times each second) Finally, if a new sprite is signaled the background generator will just display a generic background tile instead of trying to draw the sprite that has now been passed off to another module.

Sprite RAM

To draw sprites on the screen, some information about the sprites must be saved so that the sprite generator can draw the sprite in the right position, and so that the sprite FSM can implement interesting behavior in different sprites. This is accomplished with a 16 deep by 36 bit wide random access memory. Limiting the game to displaying on 16 movable sprites at a time is more than enough for the purposes of a simple side scroll game.

The RAM is dual port so that both the sprite generator and the sprite FSM can update the state and the position of the sprites. The position is stored as a 12 bit game world x coordinate, and an 8 bit y coordinate. Five bits are used to store the sprite's tile index number. Even though the tile memory takes a 6 bit index number, sprite tiles are only in the top 32 tile, so a 1 can be concatenated to the front of the 5 bit saved index number to recall the correct tile. The final 11 bits are used for the sprite's state.

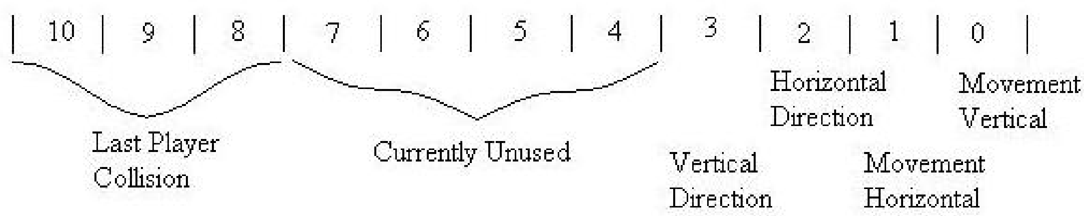


Figure 5 - Sprite State Diagram

Sprite Generator

The sprite generator is the most complicated of all the game subsystem modules. Its basic responsibility is to draw the sprites from the sprite RAM into the frame buffer. The sprite generator will only do this after vcount has reached 780. This gives the background generator enough time to draw all of the background before the sprites are drawn. This must be assured, because if two pixels are drawn to the same address in the frame buffer, then the last one written will be the pixel that is displayed on the screen. Because a background is necessarily behind everything else, it must be drawn before the sprites.

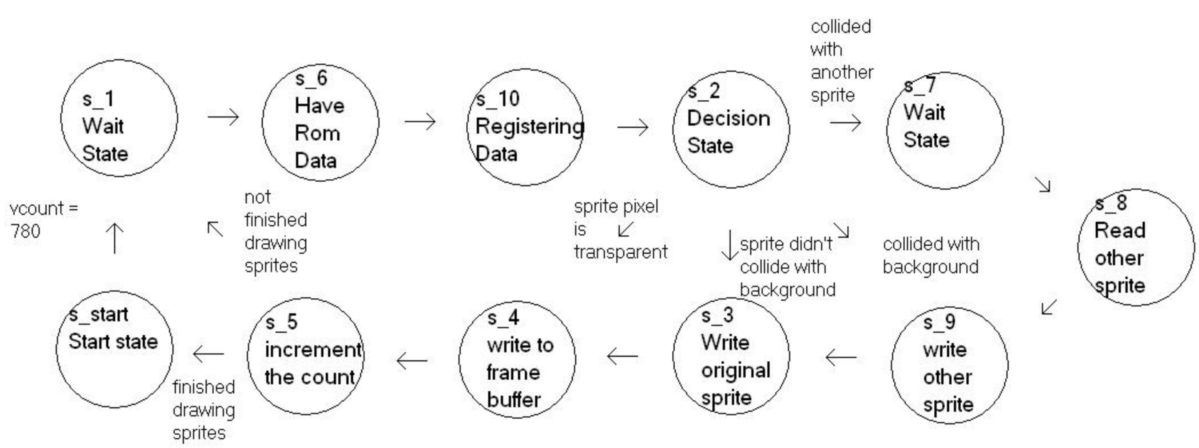


Figure 6 - Sprite Generator FSM Diagram

There is a similarity between how the sprite generator and the background generator draw tiles to the frame buffer. After waiting for the vcount to reach 780, the sprite generator starts with sprite number 0, fetches the sprite tile from the sprite RAM, and retrieves the pixel color and data based on the current pixel being written within the sprite. This is where the similarities end. While the background generator can blindly place the pixel color and data into the frame buffer, the sprite generator must first check the frame buffer at the coordinates that it wishes to write to. Because sprites can move around, they must also be able to interact with the other objects and sprites within the game world. Therefore if the sprite is blindly written into the frame buffer, a collision or other interaction between sprites and objects might be missed, ruining the “physics” of the game world.

The sprite generator retrieves the pixel data from the frame buffer at the same time that the generator is retrieving the pixel data from the tile ROM. Many comparisons are made in combinational logic to determine if the sprite has hit another sprite or object. Determining what to do when a sprite has hit another object is determined by what kind of pixel the sprite hit. This is done by using the pixel data that gets read to and from the frame buffer and the tile memory.

Table 1 - Pixel Type Descriptions

Pixel Type	***Description***
000	Sprite/Player can pass through
001	This pixel stops leftward movement
010	This pixel stops rightward movement
011	This pixel stops vertical movement
100	This pixel will kill the player if touched
101	This pixel will kill the sprite if touched
110	This pixel will modify the player if touched
111	Player wins the game if this pixel is touched

If a sprite hits a wall, the Horizontal Direction state bit of the sprite will be flipped to the opposite direction of what the sprite was going. Colliding with a horizontal object will also push the object away from the wall so that the sprite doesn't get “stuck” in the wall. The same is true of the vertical stop pixel type. If a sprite hits a vertical stop while moving upwards (no sprites jump in the current version), it will stop moving vertical and will be moved down a few pixels. If a sprite hits a vertical stop pixel type while not moving upwards, it must be because it is hitting the ground, so the sprite's y coordinates are pushed up.

More difficult to process is if a sprite hits a pixel type that is of either “Kill the player”, or “Kill the sprite”. These two pixel types denote other sprites. The difficulty arises because, when a sprite hits a vertical pixel while moving downward, the first pixel on the bottom is written into the frame buffer as the sprite generator realizes that the sprite has reached the ground, so the sprite generator pushes the sprite upwards, drawing the rest of the bottom pixels of the sprite on top of itself. When the generator draws one sprite over itself, the generator only registers the collision as another sprite, so it tries to reverse the sprite's direction. This results in a sprite that vibrates back and forth as it keeps running into itself. This is solved by adding 4 more bits to the 6 color bits and the 3 data bits stored in the frame buffer. These 4 new bits are the current sprite's number (0-15) in the sprite RAM. The sprite generator then only registers a collision if the sprite number that is retrieved from the frame buffer isn't the same as the current sprite being written into the frame buffer. The sprite generator also checks collision detection inputs from the player controller. If the player has collided with the current sprite, then whatever type of pixel the player has collided with is saved in the sprite's state's top 3 bits.

The last issue that must be addressed with the generator is updating the sprite with which the current sprite has collided. If this other sprite is not updated, then the earliest drawn sprite in the collision will not register a collision as there were no other sprites on the screen when the earlier sprites were drawn. Therefore more logic is needed to not only store information about the current sprite, but if the current sprite collides with another sprite, the generator needs to look up and update the information of the other sprite.

All of the above computations are performed using a major/minor finite state machine that not only cycles through every sprite, but also every pixel within the sprite. The state diagram doesn't explicitly show the FSM as major/minor, that is only because sprite_number, sprite_column, and sprite_row were concatenated and used as a count that was incremented each cycle around the FSM.

Sprite FSM

The sprite FSM is the brains behind the sprites. While the sprite generator takes care of collisions that move the sprite immediately, (at the next tick of the 65 MHz clock) the sprite FSM takes care of movements that happen more slowly. (at the next frame, or at 60 Hz) The most important of the "slower" actions is that every frame, the sprite is lowered by one pixel. This simulates a rudimentary gravity. If the sprite is lowered into the ground or another pixel type, the fast responding sprite generator will move the sprite back up so that the sprite doesn't continue through the ground or other object.

The other slow moving action that the FSM updates in the sprites is individual sprite movement. Different combinational logic is needed for each different type of sprite, but as of now only one type of sprite is implemented. The combinational logic updates the next x coordinate, y coordinate, sprite state, and sprite tile. For example: Depending on the horizontal direction of the sprite "Goomba" the current x coordinate will be updated to move left or right.

The y coordinate will stay the same because the "Goomba" sprite can't jump, but it will still let gravity affect the sprite. The sprite state for the "Goomba" doesn't change in the sprite FSM, only in the sprite generator when the "Goomba" hits something. The next sprite tile state is interesting, because many things can be done to a sprite from here. If the sprite is off the screen, either in front or behind or fallen down a hole, then the next sprite tile can be set to 00000, which is a completely blank sprite that is recognized as not being a sprite. Another option is to change the sprite tile of the current sprite to achieve animation. If a reset is pressed, the sprite FSM cycles through all 16 sprites and rewrites their sprite tiles to 00000, effectively erasing all the sprites from memory.

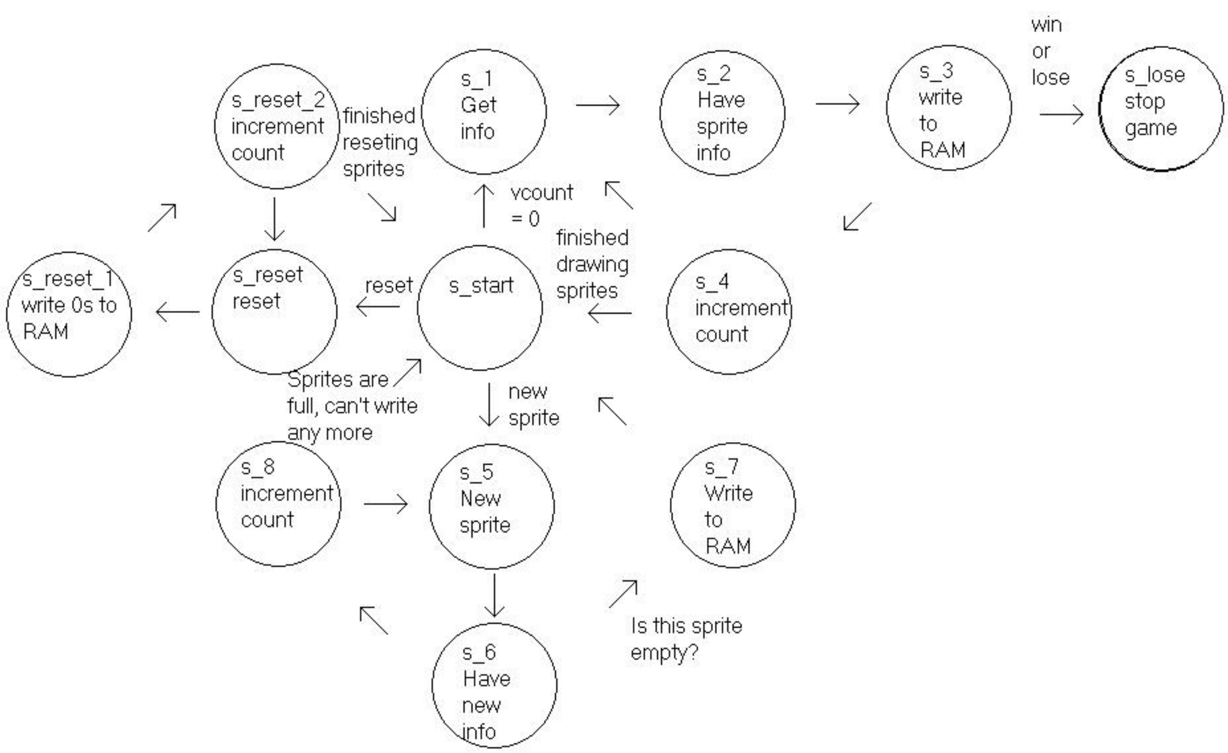


Figure 7 - Sprite FSM Diagram

The sprite FSM also checks the state of every sprite to see if the player has collided with the sprite. If the player has collided, the FSM checks to see what kind of collision it was. If the player defeated the enemy, then the sprite's tile is set to 00000. If the player is defeated by the enemy, a lose_game register is set high, ending the game.

One last issue with the sprite FSM is new sprite signals from the background generator module. If it receives this signal, the sprite generator cycles through the different sprite numbers until it finds a number that is not occupied by another sprite. Then the sprite generator takes the sprite index, the starting y coordinate, (as of now, sprites can only start at the right edge of the screen) and an initializing sprite state, and writes these values to the open sprite number in the sprite RAM.

The FSM for the sprite FSM module is also a major/minor FSM. It is also not explicitly drawn as a major/minor FSM because of the way that sprite_number is being used as part of the count. This FSM is also interesting because there are three different cycles that can be taken all from the same start state.

Player Controller

The player controller's responsibility is to take the signals from the the video subsystem and translate those into interaction with the game world. The signals that the player controller module receives are control signals: move forward, move backward, and jump request. The other input from the video subsystem is a single bit signal that is 1 if the player occupies the pixel denoted by the current hcount and vcount. This poses quite a few difficulties. The first is that while the sprites are updated between frames, the player controller only knows where the player is during the frames. It is important to note that the game subsystem does not draw the player at all. It merely finds out where the player is, reacts, and tells the video subsystem where to place the player's video. This avoids glitching issues of trying to write into the frame buffer while also reading from it.

To detect collisions, the player controller monitors the pixel data that is coming out of the frame buffer as the pixel color is being drawn on the screen. If the player has a pixel in the same place occupied by another object, then a register is set high so that the issue can be resolved between frames by the sprite generator and sprite FSM.

The player controller can move the x and y coordinates where the video subsystem places the video, and so can move the player forward, backward, up and down in the screen. If the player passes a certain point on the screen, left_pixel is incremented, moving the entire game world forward.

The second difficulty that is presented by using a video of a person as the hero of the game is that while the shapes of sprites are known and unchanging, the player video is constantly changing. This is usually not a problem, because while the player's shape changes, if a pixel is touching the ground or another object, the player's coordinates are adjusted accordingly. The biggest danger (bug) of this system is stray pixels. If a stray pixel that is not part of the player is fed into the game subsystem, the game will act just as if a player is at that pixel. For most situations, this is just an inconvenience. ie. The player hits a pipe or wall before s/he is supposed to. In other situations this problem can cause a player to unintentionally hit an enemy. Even worse is if a player is next to a block that stops horizontal movement. If every pixel inside the wall tile pushes the player out, a random pixel that appears inside of the wall tile may push the player off the screen. This is even worse for pixels that stop vertical movement.

Tiles

Tiles are created by loading a coe file into the tile ROM. The data for each pixel is 10 bits. 6 bits of color (2 red, 2 blue, 2 green), 3 bits of pixel data, and one bit that determines if the pixel is transparent. (1 is tranparent). A sample green pixel that blocks leftward movement is as follows :

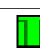
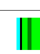
0000110010

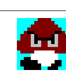
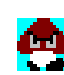
Separated out, it can be seen easier what the bits stand for :

00	00	11	001	0
red	blue	green	data	trans.

Here is a table of all the tiles that are used in the game:

Table 2 - Game Tiles

Tile Name	***Tile Number***	***Picture***
Bottom Brick	000001	
Brick	000010	
Block	000011	
Pipe Top Left	000100	
Pipe Top Right	000101	
Pipe Left	000110	

Tile Name	***Tile Number***	***Picture***
Pipe Right	000111	
Flag Pole	001000	
Flag Pole Top	001001	
Question Box	001010	
Hill Top	001011	
Hill Left	001100	
Hill Right	001101	
Hill Middle	001110	
Bush Left	001111	
Bush Right	010000	
Bush Middle	010001	
Cloud Top Left	010010	
Cloud Top Right	010011	
Cloud Top Middle	010100	
Cloud Bottom Left	010101	
Cloud Bottom Right	010110	
Cloud Bottom Middle	010111	
Goomba 1	100001	
Goomba 2	100010	

Testing and Debugging

Video Circuitry

Testing of the video logic was accomplished primarily through use of programming the labkit. In certain instances, the logic analyzer was utilized to check the state of certain signals. Debugging information was displayed using leds as well as the 16-digit LED display. Test benches could not be readily used since operation often required interfacing with the ZBT memories and required an incoming video feed. While this method of verification was slow and tedious, it still allowed for effective testing.

Figure 8- Debugging Video Logic

In some cases, however, the VGA video display was also used for testing and debugging purposes. For instance, to test center of mass calculations from the gestures module, a blob, like that from the pong game, was displayed at the x and y coordinates corresponding to the center of mass. This allowed for visual verification of the calculation. Additionally, with the overlay module, a solid color was often used in place of the game world pixels for testing and verification of the video logic.

Some of the most recurring problems encountered with video testing came in the form of small typographical errors or seemingly minor mistakes. Unfortunately, the Xilinx tools try to guess what was intended rather than displaying an error in these circumstances. This resulted in endless hours of sifting through code, only to find the minor error that was being caused.

However, for the video logic, the most nagging problem tended to be issues related to meeting the timing of the 65MHz VGA clock. Since a pixel would need to be displayed once every 15ns or so, the logic and calculations that took place on every clock period had to be quite speedy to finish before the next clock edge. Thus, this resulted in an added requirement of having pipeline several of the modules with several stages. This allowed for increased throughput, letting the clock run at its normal speed, but it adds delay from the time information comes into a module to when it leaves. Considering that it is not a noticeable difference if the video is shifted by a couple pixels in one direction or if the modules indicate that a pixel that is off by one from the character actually belongs to the character, this fortunately did not create the need to worry about this delay of information.

An additional complication that arose was the need to switch from keying on RGB color values to YCrCb values. While the keying on the RGB values worked in a decent manner, this left the issue of dealing with lighting and small variations in color. Thus, the background removal effect was easily thrown off by small changes and the resulting video image was incredibly noisy, a problem that would result in logic detecting false collisions with the player. To reduce the noise, a low-pass filter was created to detect large variations in pixel color in an attempt to smooth out the removal of the background and prevent registering of false collision.

Thus, it was decided to key off values in the YCrCb space. Once changed to key off of YCrCb values, the background removal worked much more reliably and noise was dramatically cut down. Furthermore, this eliminated most of the need for a low-pass filter to reduce the noise, since the chroma values that the module is keying off are insensitive to variations in lighting intensity.

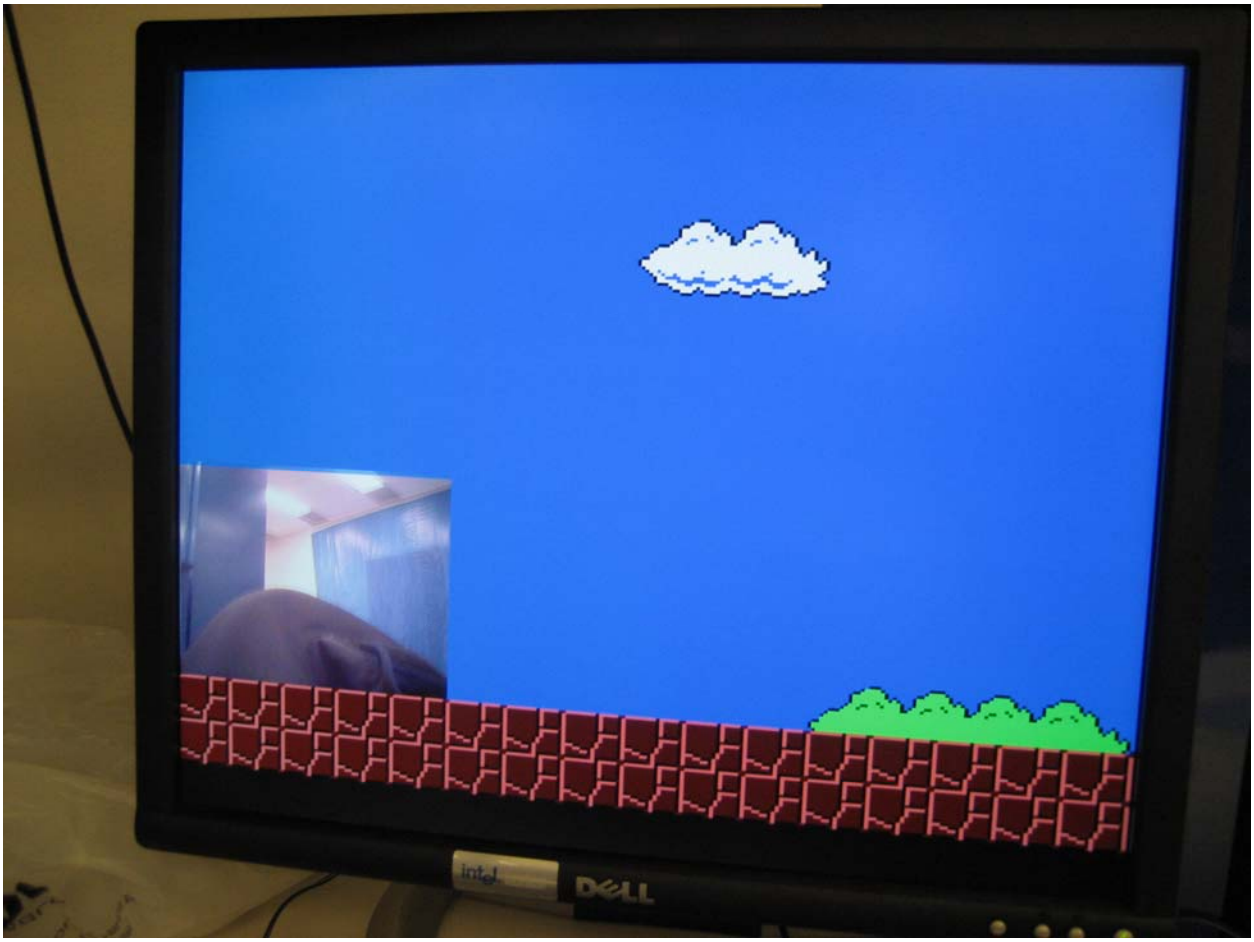


Figure 9 - Video Logic Integrated with Game Logic

Additional complications for the video logic were caused by routing problems with the Xilinx tools. The quality of the generated bit file would vary wildly from one compilation to the next. A lot of the delay the tools cited as being an issue for meeting timing constraints was caused by delay from having to route signals between modules. This delay would, unfortunately, cause glitches in the video display during one of the sub-par runs. Furthermore, abstracting constants away as parameters would cause further delays in the timing of the circuitry. Unable to find a root cause of this issue, some of the constants needed to be left in-line with the calculations.

One complication arose with the video with regards to the grid created for gesture recognition. For an unknown reason , a simple greater than comparison failed for the right side of the grid, while similar calculations worked for the left, top, and bottom. First, a pipelined stage was attempted, but this failed to resolve the problem. Next, adjusting the grid line and offsetting the center was attempted. However, none of these approaches solved the problem. The issue was finally resolved when an unrelated module was changed. Thus, this indicates that the problem may have been due to an issue with the Xilinx router failing to get the signal properly routed to its destination.

Another seemingly simple task ended up creating complications when the video needed to be scaled down by a factor of two. First, when the values were simply shifted left by one, no video would show, but then it was discovered that the logic would only interact with the ZBT ram when the low order bit of hcount is high. Thus, the values needed to be padded with 1’s instead of 0’s. Next, the video would only show every other horizontal pixel that needed to be shown (or every four overall). To fix this when the highest order bit of hcount was high, the rest of the video was filled in. This gave nice, crisp video image that could now be scaled by two. After this was resolved the rest of the logic was accommodated to allow for effortless switching between full-size and reduced-size video, a feature that was desired for game-play.

Finally, although both game and video logic worked efficiently when run separately, when combined, compilation resulted in several delays that slowed the video output signal.

This was because once the game logic finished processing its data, additional processing still needed to be done on the video end. Thus, the video logic would need to wait for the game logic, then perform additional operations. This resulted in heavy glitching of the video signal. To fix this issue, several pipelined stages were used to separate segments of the logic so the video circuitry would not be too dependent on the game logic. This helped fix many of the timing issues that arose.

Game Debugging

Figure 10 - Game Logic working independently of the Video Logic

The process of debugging this subsystem took much longer than the actual design of the system. Often the problems that held the entire subsystem back were nothing more than misspelled wire names or buses of the wrong width. The warnings list inside the Xilinx ISE software is very helpful for finding simple errors, as often a report saying that port sizes do not match lead to the correction of a seemingly impossible problem. Other than staring at the code, the process of debugging the project follows the discussion above, especially the sprite generator module. Fixing one thing in the sprite generator just uncovered a new problem that needed to be fixed.

Conclusions

Thus, this paper documents our design, implementation, and testing of our interactive adventure game. The circuitry includes logic to handle video input, gesture recognition, character recognition, video translation, game background generation, game sprite movement and generation, as well as multiple frame buffers for improve video output quality. In addition, the circuitry contains logic to handle further extensions, such as power-ups and an animated game world.

This project taught us quite a bit about interfacing and working with video as well as on-chip memories. Further, we learned a great deal about major and minor finite state machines, video buffering, BRAM and ZBT memories, sprite generation and animation, chroma keying, gesture recognition using center of mass, and video standards. However, once we began integration of the two halves, meeting timing constraints began to get difficult. Fortunately, through the use of heavy pipelining of the video circuitry and several optimizations across the board.

Our final implementation of the project includes pretty much everything we planned to include from the early stage designs. Although character scaling logic is perfectly functional within the game and video circuitry, we were forced to disable the feature in the final game because of the ratio of the sizes of the video window and the game world. In all, we believe we've been able to provide a complete experience for the user in combining the thrill of a beautiful side-scrolling adventure game with the awe of immersing the user in the game experience with a truly interactive system.

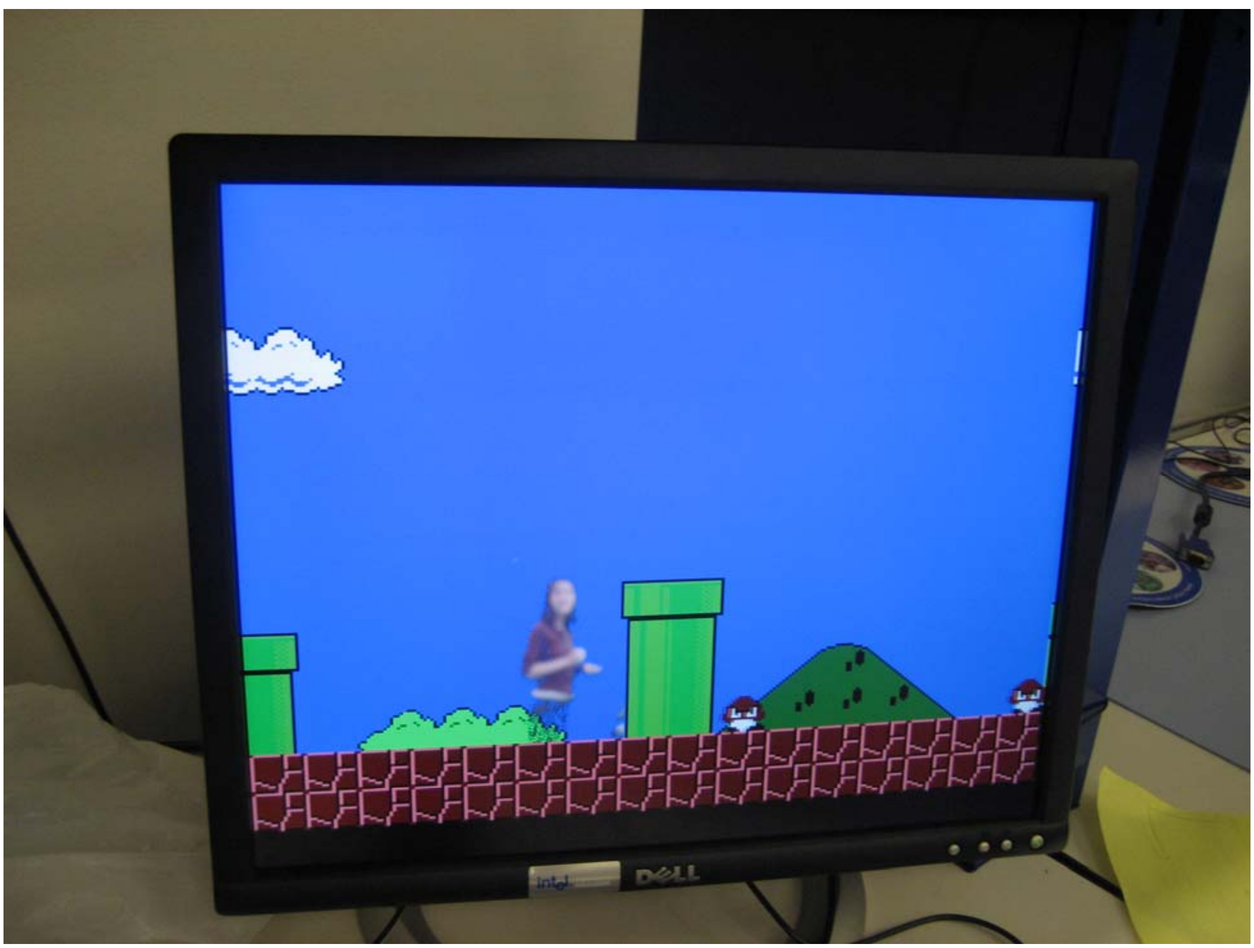


Figure 11 - A working version of the project.

Given time, there are several features that we would have loved to included in the game. We were hoping to have enough time to include game world music to make the game-play more entertaining, especially since music is an integral part to any video game. We would have also liked to include more interactivity in the game world, such as animated and interactive coin boxes, the ability to go into pipes (as in Super Mario Bros.),and power-ups such as power stars and extra lives. Additionally, we would have included a series of game levels, given the extra time. Fortunately, adding another level would be incredibly effortless, since our logic was designed with expandability in mind, and it would simply require mapping out the placement of tiles and sprites.

Thanks to the entire 6.111 staff and all the students.

Appendix A: Verilog Code

```
//          Video Wrapper Module
// File:   video_wrapper.v
// Date:   12/12/06
// Author: Akash Shah
//
// Wrapper code for the video logic for an interactive adventure game
// run on the MIT 6.111 labkit using the ZBT memories for video display.
// Video input from the NTSC digitizer is displayed within an XGA 1024x768 window.
// Two ZBT memories are used as the video buffer.
//
// 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.
//
// In addition to the logic to store and buffer video, this wrapper module
// integrates the logic necessary for background removal, character and gesture
// recognition, game logic, video translation, as well as overlaying of the
// game and live video worlds.
//


///////////////////////////////////////////////////////////////////////////////
//
// 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 video_wrapper(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



//===========================================================
// Akash Shah
//==========================================================
```

```
//
// Modified ZBT module letting us well, do what we need to.
// A lot of pipelining was needed to satisfy timing requirements.
// Basically, everything is put together here.
//============================================================

/* open up both ZBTs to store our video


  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'b0;
    assign ram1_oe_b = 1'b0;
//              assign ram1_we_b = 1'b0;
    assign ram1_bwe_b = 4'h0;

  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;

  // 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);

      wire [35:0] vram_write_data_1;
wire [35:0] vram_read_data_1;
wire [18:0] vram_addr_1;
wire       vram_we_1;

      zbt_6111 zbt2(clk, 1'b1, vram_we_1, vram_addr_1,
              vram_write_data_1, vram_read_data_1,
              ram1_clk, ram1_we_b, ram1_address, ram1_data, ram1_cen_b);

// generate pixel value from reading ZBT memory
wire [23:0]     vr_pixel;
wire [18:0]     vram_addr1;

      wire [10:0] hshift;
      wire [9:0] vshift;

      reg [10:0] xshift;
      reg [9:0] yshift;

      //generate signals for newline and newframe
      wire newframe, newline;
      reg oldv, oldh;

      wire shrink;

      assign shrink = 1'b1;

      wire [10:0] player_xcoord;
      wire [9:0] player_ycoord;

      reg selectorx;

      //registers that store shifts along the plane
      always @ (posedge clk) begin
            //translate video window with player coordinates from game
            xshift <= 11'h62C - player_xcoord;
            yshift <= 10'h32 - player_ycoord;

            //need to select which shifted value to use
            selectorx <= ~shrink | hcount[9];
      end

      reg [10:0] xshiftba, xshiftbb, xshiftc;
```

```
	reg [9:0] yshiftba, yshiftbb, yshiftc;

	reg [10:0] horizontal_shift;
	reg [9:0] vertical_shift, vsa, vsb;

	//pipelineing and further calculations.  different logic for when using small video
	//constants would be stored as parameters, but Xilinx doesn't seem to like this
	always @ (posedge clk) begin

		xshiftba <= xshift;							//normal reading of video
		xshiftbb <= xshift + 10'b10_0000_0000;	//interlace horizontally by shifting video back
		xshiftc <= selectorx ? xshiftba : (xshiftbb + 1'b1);  //need to add one to offset into the odd lines now

		yshiftba <= (yshift + 10'b10_0000_0001 + vcount);	//simlar to xshift, but vertical is easier
		yshiftbb <= (yshift + vcount);
		yshiftc  <= shrink ? ((vcount[9]) ? {yshiftba[8:0], 1'b0} : {yshiftbb[8:0], 1'b0}) : yshiftbb;

		horizontal_shift <= selectorx ?  xshiftba : xshiftbb;	//just ablsolute horizontal shift,

										//without worrying about scaling

		vsa <= yshift + 10'b10_0000_0001;
		vsb <= yshift;

		vertical_shift <= shrink ? ((vcount[9]) ? {vsa[8:0], 1'b0} : {vsb[8:0], 1'b0}) : vsb;

	end

	//reposition our video
	wire inside;

	//defines grid used by the gesture recognition
	wire [10:0]grid_left_x, grid_right_x;
	wire [9:0] grid_top_y, grid_bottom_y;

	videoreposition mover(.clk(clk), .reset(reset), .x(xshiftc), .y(yshiftc),
		.hshift(hshift), .vshift(vshift), .hcount(hcount), .vcount(vcount),
		.shrink(shrink), .inside(inside), .xshift(xshift), .yshift(yshift), .vertical_shift(vertical_shift), .grid_left_x(grid_left_x),
.grid_right_x(grid_right_x),
		.grid_top_y(grid_top_y), .grid_bottom_y(grid_bottom_y));

	//get our video from ZBT
  vram_display vd1(reset,clk,(hcount + hshift),vshift,vr_pixel,
			 vram_addr1,vram_read_data, shrink);

	// generate pixel value from reading ZBT memory
  wire [23:0]		vr_pixel_1;
  wire [18:0]		vram_addr1_1;

  vram_display vd2(reset,clk,(hcount+ hshift),vshift,vr_pixel_1,
			 vram_addr1_1,vram_read_data_1, shrink);

  // 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 [18:0] ntsc_addr_1;
wire [35:0] ntsc_data_1;
wire        ntsc_we_1;

        wire [7:0] rout, gout, bout;

ntsc_to_zbt n2z  (clk, tv_in_line_clock1, fvh, dv, {rout[7:3],gout[7:5]},
                        ntsc_addr, ntsc_data, ntsc_we, 1'b0);

        ntsc_to_zbt n2z2 (clk, tv_in_line_clock1, fvh, dv, {gout[4:2],bout[7:3]},
                        ntsc_addr_1, ntsc_data_1, ntsc_we_1, 1'b0);

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

wire [18:0]         vram_addr2 = count[0+18:0];
        wire [18:0]         vram_addr2_1 = count[0+18:0];
wire [35:0]         vpat = 0;
        wire [35:0]         vpat_1 = 0;

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

wire    sw_ntsc = 1'b1;
wire    my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
        wire    my_we_1 = sw_ntsc ? (hcount[1:0]==2'd2) : blank;

wire [18:0]         write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
        wire [18:0]         write_addr_1 = sw_ntsc ? ntsc_addr_1 : vram_addr2_1;

wire [35:0]         write_data = sw_ntsc ? ntsc_data : vpat;
        wire [35:0]         write_data_1 = sw_ntsc ? ntsc_data_1 : vpat_1;

assign  vram_addr = my_we ? write_addr : vram_addr1;
        assign  vram_addr_1 = my_we_1 ? write_addr_1 : vram_addr1_1;

assign  vram_we = my_we;
        assign  vram_we_1 = my_we_1;

assign  vram_write_data = write_data;
        assign  vram_write_data_1 = write_data_1;

// select output pixel data

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);


        //=============================
        //filter out our background color
```

```
wire filter;
greenscreen gs(.clk(clk), .reset(reset), .calibrate(~button1), .hcount(hcount), .vcount(vcount), .pixelin(ycrcb), .filter(filter));


reg [29:0] postremove;

always @ (posedge clk) begin
        oldv = vsync;
        oldh = hsync;
        postremove <= filter ?  {10'd0, 10'd0, 10'd0} : ycrcb[29:0];
end

//generate newline and newframe
assign newframe = vsync & ~oldv;
assign newline = hsync & ~oldh;

//set to black if must be filtered out
always @ (posedge clk) postremove <= filter ?  {10'd0, 10'd0, 10'd0} : ycrcb[29:0];


//convert to RGB -- this is then stored in the ZBT
YCrCbtoRGB convert(.clk(clock_65mhz), .reset(reset), .Y(postremove[29:20]), .Cr(postremove[19:10]), .Cb(postremove[9:0]),
                                                                                    .R(rout), .G(gout), .B(bout));

wire pixel_out;

//pad the info we got from the ZBT with 1's so we get pure white
always @ (posedge clk) pixel <= {vr_pixel[7:3],3'b111,

vr_pixel[2:0],vr_pixel_1[7:5],2'b11,
                                                                                    vr_pixel_1[4:0],3'b111};

wire char_pixel_on;
wire [23:0] char_pixel_out;

//detect where our character is
character char(.clk(clk), .reset(reset), .hcount(hcount), .vcount(vcount), .newline(newline),
                .newframe(newframe), .scale(1'b0), .tranx(10'b0), .trany(9'b0), .pixelin(pixel), .inside(inside),
                .char_pixel_on(char_pixel_on), .char_pixel_out(char_pixel_out));

wire [10:0] centerx;
wire [9:0] centery;
wire g_stand, g_jump, g_duck, g_left, g_right;

//calculate the center of mass of the player as well as recognize player gestures
gestures gesture_recognition(.clk(clk), .reset(reset), .char_pixel_on(char_pixel_on), .inside(inside),
                .newline(newline), .newframe(newframe), .hcount(hcount), .vcount(vcount), .centerx(centerx),
.centery(centery),
                .g_stand(g_stand), .g_jump(g_jump), .g_duck(g_duck), .g_left(g_left), .g_right(g_right),
.maxy(grid_bottom_y), .maxx(grid_right_x), .minx(grid_left_x), .miny(grid_top_y));


wire[23:0] game_pix;

wire [63:0] disp_bus;
wire a1_clock;
wire [15:0] a1_data;
wire [15:0] a2_data;
wire player_size;

//get the game data
game_wrapper game(.vclock(clk),.reset(reset),.up(g_jump),.down(g_duck),.left(g_left),.right(g_right),
                                                        .hcount(hcount), .vcount(vcount),
.player_pixel_on(char_pixel_on),
```

```
                                                        .pixel(game_pix),
                                                        .player_xcoord(player_xcoord), .player_ycoord(player_ycoord),
.hsync(hs), .vsync(vs), .blank(b));


        //overlay the game data with the video data.  player always on top.
        wire [23:0] final_pixel;
        overlay over (.clk(clk), .reset(reset), .char_pixel_on(char_pixel_on), .char_pixel(pixel),
                                .game_pixel(game_pix), .hcount(shrink ? hcount << 1 : hcount), .vcount(shrink ? vcount << 1 :
vcount), .centerx(centerx),
                                .centery(centery),.pixel_out(final_pixel), .hshift(horizontal_shift), .vshift(vertical_shift),
.shrink(shrink), .inside(inside));

  // VGA Output.  In order to meet the setup and hold times of the
  // AD7125, we send it ~clock_65mhz.
  assign vga_out_red = final_pixel[23:16];
  assign vga_out_green = final_pixel[15:8];
  assign vga_out_blue = final_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 = ~{g_stand, g_jump, g_duck, g_left, g_right, char_pixel_on, reset,switch[0]};


  always @(posedge clk)
        dispdata <= {5'b0, centerx, 6'b0, centery, 5'b0, grid_left_x, 5'b0, grid_right_x};

endmodule

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

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

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

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

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

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

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

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

////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.
///
//==========================================================
//  Modified by Akash Shah
//
//        Now also takes in a signal that indicates whether or not the video needs
//              to be shrunk by a factor of 2.  If so, this alters the way addresses are
//              read out from the ZBT.  Since the pixels are read out groups of 4, this task
//              wasn't as simple as skipping every other pixel.  The module now essentially
//              interlaces pixels horizontally while skipping over some pixels (since the
//              highest-order bit is now ignored).  Meanwhile, the vcount that is passed in
//              must be shifted by one bit to the left.  This will now essentially skip every
//              other line vertically.  This module now allows for efficient scaling of the
//              video feed.
//

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

  input reset, clk;
  input [10:0] hcount;
  input [9:0]        vcount;
  output [7:0] vr_pixel;
  output [18:0] vram_addr;
  input [35:0]  vram_read_data;
        input shrink;

                                              //shift hcount to shrink video
  wire [18:0]        vram_addr = shrink ? {1'b0, vcount, hcount[8:1]} : {1'b0, vcount, hcount[9:2]};
  wire [1:0]         hc4 = shrink ? {hcount[0] , 1'b1} : hcount[1:0];

  reg [7:0]   vr_pixel;
  reg [35:0]         vr_data_latched;
  reg [35:0]         last_vr_data;


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

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

  always @ *                              // each 36-bit word from RAM is decoded to 4 bytes
    case (hc4)
      2'd3: vr_pixel = last_vr_data[7:0];
      2'd2: vr_pixel = last_vr_data[7+8:0+8];
      2'd1: vr_pixel = last_vr_data[7+16:0+16];
      2'd0: vr_pixel = last_vr_data[7+24:0+24];
    endcase

endmodule // vram_display



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

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

  parameter NDELAY = 9;

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

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

endmodule // delayN

//===========================================================
// YCrCbtoRGB - Akash Shah
//==========================================================
//
//  This module converts YCrCb values (30 bit) to a set of
//  24-bit RGB values using the color conversion formulas
//  (found pretty much anywhere online).  However, this module
//  takes additional considerations into account.  For example
//  it checks the luminance values that are coming in.  In order
//          to interface properly with the chroma keying module, this module
//          will return the color 'black' in its calculations whenever Y is
//  set to zero.  This was to allow for efficiency in circuit timing
//  to meet the requirements of the 65mhz video clk.  Finally, this
//  module has a 3-stage pipeline to help yet again with timing specs.
//=============================================================
module YCrCbtoRGB(clk, reset, Y, Cr, Cb, R, G, B);
          input clk, reset;
          input [9:0] Y, Cr, Cb;
          output [7:0] R,G,B;

          reg [20:0] Ra,Ga,Ba,X,A,B1,B2,C;
          reg [9:0] const1,const2,const3,const4,const5;
          reg[9:0] Ya, Cra, Cba;
          reg filter_reg;

//        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

//pipeline like crazy
always @ (posedge clk) begin
  if (reset)
    begin
    Ya <= 0;
        Cra <= 0;
        Cba <= 0;
    end
  else
    begin
      Ya <= Y;
      Cra <= Cr;
      Cba <= Cb;
    end
end

always @ (posedge clk) begin
  if (reset)
    begin
     A <= 0;
        B1 <= 0;
        B2 <= 0;
        C <= 0;
        X <= 0;
        filter_reg <= 0;
    end
  else
   begin
   X <= (const1 * (Ya - 'd64)) ;
   A <= (const2 * (Cra - 'd512));
   B1 <= (const3 * (Cra - 'd512));
   B2 <= (const4 * (Cba - 'd512));
   C <= (const5 * (Cba - 'd512));
      filter_reg <= (Ya == 0);        //filter color to black if Y = 0
   end
end

always @ (posedge clk) begin
  if (reset)
    begin
     Ra <= 0;
        Ga <= 0;
        Ba <= 0;
    end
  else
   begin
   Ra <= X + A;
   Ga <= X - B1 - B2;
   Ba <= X + C;
   end
end

// limit output and kill the data if necessary
assign R = (Ra[20] | filter_reg) ? 0 : (Ra[19:18] == 2'b0) ? Ra[17:10] : 8'b11111111;
assign G = (Ga[20] | filter_reg) ? 0 : (Ga[19:18] == 2'b0) ? Ga[17:10] : 8'b11111111;
assign B = (Ba[20] | filter_reg) ? 0 : (Ba[19:18] == 2'b0) ? Ba[17:10] : 8'b11111111;

endmodule


//==========================================================
```

```
// Videoreposition - Akash Shah
//========================================================
//
//  This module allows for easy repositioning of the video.  In
//  addition to adding shift constants to values passed into the module,
//  this module provides calculations for other modules that rely
//  on video calculations.  For instance, one of the most significant
//              calculations is that which calulates if the current hcount vcount
//              belongs to the video feed window.  This module also calculates
//              x and y values for the grid that is used for calculation by the
//              gestures module.  This all allows for abstraction and simplicity
//              within the other modules.
//============================================================
module videoreposition(clk, reset, x, y, hshift, vshift, shrink, hcount, vcount, xshift, yshift, vertical_shift, inside, grid_left_x, grid_right_x,
grid_top_y, grid_bottom_y);
        input reset, clk, shrink;
  input [10:0] x, hcount, xshift;
  input [9:0] y, vcount, yshift, vertical_shift;

        output [10:0] grid_left_x, grid_right_x;
        output [9:0] grid_top_y, grid_bottom_y;

        output [10:0] hshift;
  output [9:0] vshift;
        output inside;

        assign hshift = x;
        assign vshift = y;

        //initial x and y for small and big video
        parameter small_min_x = 11'd40;
        parameter small_min_y = 10'd100;

        parameter small_max_x = 11'd360;
        parameter small_max_y = 10'd570;

        parameter large_min_x = 11'd40;
        parameter large_min_y = 10'd100;

        parameter large_max_x = 11'd720;
        parameter large_max_y = 10'd570;

        //many registers for several pipeline stages
        reg inside_small_xa, inside_small_xb, inside_small_xc,
                inside_small_ya, inside_small_yb,
                inside_large_xa, inside_large_xb,
                inside_large_ya, inside_large_yb,
                in_small, in_large, inside;

        reg [10:0] glx, grx, glxs, grxs, glxl, grxl, grid_left_x, grid_right_x;
        reg [9:0] gty, gby, gtys, gbys, gtyl, gbyl, grid_top_y, grid_bottom_y;

        reg top_video;

        always @ (posedge clk) begin

                //calculates whether hcount vcount is within video feed or not
                // pipelined to meet timing requirements
                inside_small_xa <= ~hcount[9];
                inside_small_xb <= (hcount + x < 11'd360);
                inside_small_xc <= (hcount + x > 11'd42);

                inside_small_ya <= (vcount + yshift < 10'd278);
                inside_small_yb <= (vcount + yshift > 10'd50);
```

```
			inside_large_xa <=  (hcount + x < 11'd720);
			inside_large_xb <=  (hcount + x > 11'd40);
			inside_large_ya <=  (y > 10'd100);
			inside_large_yb <=  (y < 10'd570);


			//calculates grid lines for gestures module
			//more pipelining fun
			glxs <= 11'd1824 - xshift;
			grxs <= 11'd1748 - xshift;

			gtys <=		10'd176 - yshift;
			gbys <=		10'd220 - yshift;

			glxl <= 11'd290 - xshift;
			grxl <= 11'd490 - xshift;

			gtyl <=		10'd350 - vertical_shift;
			gbyl <=		10'd500 - vertical_shift;


			//stage two of the pipeline
			in_small <= inside_small_xa & inside_small_xb & inside_small_xc & inside_small_ya & inside_small_yb;
			in_large <= inside_large_xa & inside_large_xb & inside_large_ya & inside_large_yb;

			glx <= shrink ? (glxs - 11'h88) : glxl;
			grx <= shrink ? (grxs + 11'hC8) : grxl;

			gty <=		shrink ? gtys : gtyl;
			gby <=		shrink ? gbys : gbyl;

			//output stage of pipeline
			inside <= shrink ? in_small : in_large;

			grid_left_x  <= glx;
			grid_right_x <= grx;

			grid_top_y    <=		gty;
			grid_bottom_y <=		gby;
		end
endmodule

//=========================================================
// Zlowpass - Akash Shah
//========================================================
//
//  This module essentially creates a lowpass filter by checking
//  if there is a large change from one pixel to the next.  It
//		essentially tries to smooth out noise in the video signal.
//  Seems to make little difference now (tried different implementations)
//		since I've changed the way the chroma keying works.
//============================================================
module zlowpass(clk, reset, newline, pixelin, pixelout);
		input clk, reset, newline;

		input [23:0] pixelin;
		output [23:0] pixelout;

		reg [23:0] pixelout;
		reg [23:0] lastpixel;

		wire [23:0] changes;
		wire filter;
```

```
	//one scheme used to filter.  not necessarily the best, but a faster one
	assign changes = pixelin ^ lastpixel;
	assign filter = (changes[23:16] > 8'd32) | (changes[15:8] > 8'd32) | (changes[7:0] > 8'd32);

	always @ (posedge clk) begin
		if(reset || newline)	begin
			pixelout <= pixelin;
			lastpixel <= 23'd0;
		end
		else begin
			pixelout <= filter ? lastpixel : pixelin;
			lastpixel <= pixelin;
		end
	end
endmodule

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

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

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

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

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

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

  parameter		SYNC_1 = 0;
  parameter		SYNC_2 = 1;
  parameter		SYNC_3 = 2;
  parameter		SAV_f1_cb0 = 3;
  parameter		SAV_f1_y0 = 4;
```

```
parameter       SAV_f1_cr1 = 5;
parameter       SAV_f1_y1 = 6;
parameter       EAV_f1 = 7;
parameter       SAV_VBI_f1 = 8;
parameter       EAV_VBI_f1 = 9;
parameter       SAV_f2_cb0 = 10;
parameter       SAV_f2_y0 = 11;
parameter       SAV_f2_cr1 = 12;
parameter       SAV_f2_y1 = 13;
parameter       EAV_f2 = 14;
parameter       SAV_VBI_f2 = 15;
parameter       EAV_VBI_f2 = 16;



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

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

// The data stream looks as follows
// SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | EAV sequence
// There are two things we need to do:
//   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
//   2. Decode the subsequent data

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

assign     state = current_state;

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

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

              SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y0;
              SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cr1;
              SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y1;
              SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cb0;

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

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

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

              endcase
            end
  end // always @ (posedge clk)

  // implement our decoding mechanism

  wire y_enable;
  wire cr_enable;
  wire cb_enable;

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

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

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

  reg        f = 0;

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

endmodule



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

```
///////////////////////////////////////////////////////////////////////////////
// Register 0
///////////////////////////////////////////////////////////////////////////////

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

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

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

`define VIDEO_QUALITY               2'h0
 // 0: Broadcast quality
 // 1: TV quality
 // 2: VCR quality
 // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE        1'b0
 // 0: Normal mode
 // 1: Square pixel mode
`define DIFFERENTIAL_INPUT          1'b0
 // 0: Single-ended inputs
 // 1: Differential inputs
`define FOUR_TIMES_SAMPLING         1'b0
 // 0: Standard sampling rate
 // 1: 4x sampling rate (NTSC only)
`define BETACAM                     1'b0
 // 0: Standard video input
 // 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE    1'b1
 // 0: Change of input triggers reacquire
 // 1: Change of input does not trigger reacquire

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

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

`define Y_PEAKING_FILTER            3'h4
 // 0: Composite =  4.5dB,  s-video =  9.25dB
 // 1: Composite =  4.5dB,  s-video =  9.25dB
 // 2: Composite =  4.5dB,  s-video =  5.75dB
 // 3: Composite =  1.25dB, s-video =  3.3dB
```

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

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

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

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

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

///////////////////////////////////////////////////////////////////////////////
// Register 4
///////////////////////////////////////////////////////////////////////////////

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

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

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


`define GENERAL_PURPOSE_OUTPUTS         4'b0000
`define GPO_0_1_ENABLE                  1'b0
 // 0: General purpose outputs 0 and 1 tristated
 // 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE                  1'b0
 // 0: General purpose outputs 2 and 3 tristated
 // 1: General purpose outputs 2 and 3 enabled
```

```
`define BLANK_CHROMA_IN_VBI                 1'b1
 // 0: Chroma decoded and output during vertical blanking
 // 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE                        1'b0
 // 0: GPO 0 is a general purpose output
 // 1: GPO 0 shows HLOCK status

`define ADV7185_REGISTER_5 {`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI, `GPO_2_3_ENABLE, `GPO_0_1_ENABLE,
`GENERAL_PURPOSE_OUTPUTS}

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

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

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

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

`define INPUT_CONTRAST_ADJUST               8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

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

`define INPUT_SATURATION_ADJUST             8'h8C

`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register A
///////////////////////////////////////////////////////////////////////////////

`define INPUT_BRIGHTNESS_ADJUST             8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

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

`define INPUT_HUE_ADJUST                    8'h00

`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register C
///////////////////////////////////////////////////////////////////////////////
```

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

`define ADV7185_REGISTER_C {`DEFAULT_Y_VALUE, `DEFAULT_VALUE_AUTOMATIC_ENABLE, `DEFAULT_VALUE_ENABLE}

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

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

`define ADV7185_REGISTER_D {`DEFAULT_CB_VALUE, `DEFAULT_CR_VALUE}

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

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

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE, `TEMPORAL_DECIMATION_CONTROL,
`TEMPORAL_DECIMATION_ENABLE}

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

`define POWER_SAVE_CONTROL                  2'h0
 // 0: Full operation
 // 1: CVBS only
 // 2: Digital only
 // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY          1'b0
 // 0: Power-down pin has priority
 // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                1'b0
 // 0: Reference is functional
 // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR            1'b0
 // 0: LLC generator is functional
 // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                     1'b0
 // 0: Chip is functional
 // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                    1'b0
 // 0: Normal operation
 // 1: Reacquire video signal (bit will automatically reset)
```

```
`define RESET_CHIP                1'b0
 // 0: Normal operation
 // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP, `POWER_DOWN_LLC_GENERATOR,
`POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY, `POWER_SAVE_CONTROL}

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

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

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

`define ADV7185_REGISTER_10 8'h00
`define ADV7185_REGISTER_11 8'h00
`define ADV7185_REGISTER_12 8'h00
`define ADV7185_REGISTER_13 8'h45
`define ADV7185_REGISTER_14 8'h18
`define ADV7185_REGISTER_15 8'h60
`define ADV7185_REGISTER_16 8'h00
`define ADV7185_REGISTER_17 8'h01
`define ADV7185_REGISTER_18 8'h00
`define ADV7185_REGISTER_19 8'h10
`define ADV7185_REGISTER_1A 8'h10
`define ADV7185_REGISTER_1B 8'hF0
`define ADV7185_REGISTER_1C 8'h16
`define ADV7185_REGISTER_1D 8'h01
`define ADV7185_REGISTER_1E 8'h00
`define ADV7185_REGISTER_1F 8'h3D
`define ADV7185_REGISTER_20 8'hD0
`define ADV7185_REGISTER_21 8'h09
`define ADV7185_REGISTER_22 8'h8C
`define ADV7185_REGISTER_23 8'hE2
`define ADV7185_REGISTER_24 8'h1F
`define ADV7185_REGISTER_25 8'h07
`define ADV7185_REGISTER_26 8'hC2
`define ADV7185_REGISTER_27 8'h58
`define ADV7185_REGISTER_28 8'h3C
`define ADV7185_REGISTER_29 8'h00
`define ADV7185_REGISTER_2A 8'h00
`define ADV7185_REGISTER_2B 8'hA0
`define ADV7185_REGISTER_2C 8'hCE
`define ADV7185_REGISTER_2D 8'hF0
`define ADV7185_REGISTER_2E 8'h00
`define ADV7185_REGISTER_2F 8'hF0
`define ADV7185_REGISTER_30 8'h00
```

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

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

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


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

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

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

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

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

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

```
always @(posedge clock_27mhz)
  if (clk_div_count == 26)
    begin
          clock_slow <= ~clock_slow;
          clk_div_count <= 0;
    end
  else
    clk_div_count <= clk_div_count+1;

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign reset_slow = reset_count != 0;

//
// I2C driver
//

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

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

//
// State machine
//

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

always @(posedge clock_slow)
   if (reset_slow)
        begin
          state <= 0;
          load <= 0;
          tv_in_reset_b <= 0;
          old_source <= 0;
        end
   else
        case (state)
         8'h00:
          begin
            // Assert reset
            load <= 1'b0;
            tv_in_reset_b <= 1'b0;
            if (!ack)
                  state <= state+1;
          end
         8'h01:
          state <= state+1;
         8'h02:
          begin
            // Release reset
            tv_in_reset_b <= 1'b1;
            state <= state+1;
                  end
         8'h03:
```

```
  begin
    // Send ADV7185 address
    data <= 8'h8A;
    load <= 1'b1;
    if (ack)
              state <= state+1;
  end
8'h04:
  begin
    // Send subaddress of first register
    data <= 8'h00;
    if (ack)
              state <= state+1;
  end
8'h05:
  begin
    // Write to register 0
    data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
    if (ack)
              state <= state+1;
  end
8'h06:
  begin
    // Write to register 1
    data <= `ADV7185_REGISTER_1;
    if (ack)
              state <= state+1;
  end
8'h07:
  begin
    // Write to register 2
    data <= `ADV7185_REGISTER_2;
    if (ack)
              state <= state+1;
  end
8'h08:
  begin
    // Write to register 3
    data <= `ADV7185_REGISTER_3;
    if (ack)
              state <= state+1;
  end
8'h09:
  begin
    // Write to register 4
    data <= `ADV7185_REGISTER_4;
    if (ack)
              state <= state+1;
  end
8'h0A:
  begin
    // Write to register 5
    data <= `ADV7185_REGISTER_5;
    if (ack)
              state <= state+1;
  end
8'h0B:
  begin
    // Write to register 6
    data <= 8'h00; // Reserved register, write all zeros
    if (ack)
              state <= state+1;
  end
8'h0C:
```

```
  begin
    // Write to register 7
    data <= `ADV7185_REGISTER_7;
    if (ack)
           state <= state+1;
  end
8'h0D:
  begin
    // Write to register 8
    data <= `ADV7185_REGISTER_8;
    if (ack)
           state <= state+1;
  end
8'h0E:
  begin
    // Write to register 9
    data <= `ADV7185_REGISTER_9;
    if (ack)
           state <= state+1;
  end
8'h0F: begin
   // Write to register A
   data <= `ADV7185_REGISTER_A;
 if (ack)
   state <= state+1;
end
8'h10:
  begin
    // Write to register B
    data <= `ADV7185_REGISTER_B;
    if (ack)
           state <= state+1;
  end
8'h11:
  begin
    // Write to register C
    data <= `ADV7185_REGISTER_C;
    if (ack)
           state <= state+1;
  end
8'h12:
  begin
    // Write to register D
    data <= `ADV7185_REGISTER_D;
    if (ack)
           state <= state+1;
  end
8'h13:
  begin
    // Write to register E
    data <= `ADV7185_REGISTER_E;
    if (ack)
           state <= state+1;
  end
8'h14:
  begin
    // Write to register F
    data <= `ADV7185_REGISTER_F;
    if (ack)
           state <= state+1;
  end
8'h15:
  begin
    // Wait for I2C transmitter to finish
```

```
    load <= 1'b0;
    if (idle)
        state <= state+1;
  end
8'h16:
  begin
    // Write address
    data <= 8'h8A;
    load <= 1'b1;
    if (ack)
        state <= state+1;
  end
8'h17:
  begin
    data <= 8'h33;
    if (ack)
        state <= state+1;
  end
8'h18:
  begin
    data <= `ADV7185_REGISTER_33;
    if (ack)
        state <= state+1;
  end
8'h19:
  begin
    load <= 1'b0;
    if (idle)
        state <= state+1;
  end

8'h1A: begin
   data <= 8'h8A;
   load <= 1'b1;
   if (ack)
    state <= state+1;
end
8'h1B:
  begin
    data <= 8'h33;
    if (ack)
        state <= state+1;
  end
8'h1C:
  begin
    load <= 1'b0;
    if (idle)
        state <= state+1;
  end
8'h1D:
  begin
    load <= 1'b1;
    data <= 8'h8B;
    if (ack)
        state <= state+1;
  end
8'h1E:
  begin
    data <= 8'hFF;
    if (ack)
        state <= state+1;
  end
8'h1F:
  begin
```

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

endmodule

// i2c module for use with the ADV7185

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

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

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

  reg [7:0] state;

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

  always @(posedge clock4x)
    if (reset)
      begin
            state <= 0;
            ack <= 0;
      end
    else
```

```
case (state)
        8'h00: // idle
          begin
            scl <= 1'b1;
            sdai <= 1'b1;
            ack <= 1'b0;
            idle <= 1'b1;
            if (load)
                    begin
                      ldata <= data;
                      ack <= 1'b1;
                      state <= state+1;
                    end
          end
        8'h01: // Start
          begin
            ack <= 1'b0;
            idle <= 1'b0;
            sdai <= 1'b0;
            state <= state+1;
          end
        8'h02:
          begin
            scl <= 1'b0;
            state <= state+1;
          end
        8'h03: // Send bit 7
          begin
            ack <= 1'b0;
            sdai <= ldata[7];
            state <= state+1;
          end
        8'h04:
          begin
            scl <= 1'b1;
            state <= state+1;
          end
        8'h05:
          begin
            state <= state+1;
          end
        8'h06:
          begin
            scl <= 1'b0;
            state <= state+1;
          end
        8'h07:
          begin
            sdai <= ldata[6];
            state <= state+1;
          end
        8'h08:
          begin
            scl <= 1'b1;
            state <= state+1;
          end
        8'h09:
          begin
            state <= state+1;
          end
        8'h0A:
          begin
            scl <= 1'b0;
            state <= state+1;
```

```
  end
8'h0B:
  begin
    sdai <= ldata[5];
    state <= state+1;
  end
8'h0C:
  begin
    scl <= 1'b1;
    state <= state+1;
  end
8'h0D:
  begin
    state <= state+1;
  end
8'h0E:
  begin
    scl <= 1'b0;
    state <= state+1;
  end
8'h0F:
  begin
    sdai <= ldata[4];
    state <= state+1;
  end
8'h10:
  begin
    scl <= 1'b1;
    state <= state+1;
  end
8'h11:
  begin
    state <= state+1;
  end
8'h12:
  begin
    scl <= 1'b0;
    state <= state+1;
  end
8'h13:
  begin
    sdai <= ldata[3];
    state <= state+1;
  end
8'h14:
  begin
    scl <= 1'b1;
    state <= state+1;
  end
8'h15:
  begin
    state <= state+1;
  end
8'h16:
  begin
    scl <= 1'b0;
    state <= state+1;
  end
8'h17:
  begin
    sdai <= ldata[2];
    state <= state+1;
  end
8'h18:
```

```
  begin
    scl <= 1'b1;
    state <= state+1;
  end
8'h19:
  begin
    state <= state+1;
  end
8'h1A:
  begin
    scl <= 1'b0;
    state <= state+1;
  end
8'h1B:
  begin
    sdai <= ldata[1];
    state <= state+1;
  end
8'h1C:
  begin
    scl <= 1'b1;
    state <= state+1;
  end
8'h1D:
  begin
    state <= state+1;
  end
8'h1E:
  begin
    scl <= 1'b0;
    state <= state+1;
  end
8'h1F:
  begin
    sdai <= ldata[0];
    state <= state+1;
  end
8'h20:
  begin
    scl <= 1'b1;
    state <= state+1;
  end
8'h21:
  begin
    state <= state+1;
  end
8'h22:
  begin
    scl <= 1'b0;
    state <= state+1;
  end
8'h23: // Acknowledge bit
  begin
    state <= state+1;
  end
8'h24:
  begin
    scl <= 1'b1;
    state <= state+1;
  end
8'h25:
  begin
    state <= state+1;
  end
```

```
            8'h26:
              begin
                scl <= 1'b0;
                if (load)
                        begin
                          ldata <= data;
                          ack <= 1'b1;
                          state <= 3;
                        end
                else
                        state <= state+1;
              end
            8'h27:
              begin
                sdai <= 1'b0;
                state <= state+1;
              end
            8'h28:
              begin
                scl <= 1'b1;
                state <= state+1;
              end
            8'h29:
              begin
                sdai <= 1'b1;
                state <= 0;
              end
    endcase

endmodule

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:37:44 11/10/06
// Design Name:
// Module Name:    character
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//==========================================================
//Akash Shah
//==========================================================
//
//
//                      The character module is responsible for detecting if a given pixel belongs to
//              the game character.  Using pixel color information, it then decides if the
//                      pixel should belong to the character or not.  If it should belong to the character,
//                      it sets the char_pixel_on signal high, indicating that this is the case.  Additionally,
//                      the module takes a signal from the repositioning module that indicates whether
//                      or not the current pixel is from the video window or part of the game world or
//                      background and should not be considered as a player pixel.
//
////////////////////////////////////////////////////////////////////////////////
```

```
module character(clk, reset, hcount, vcount, newline, newframe, scale, tranx, trany, pixelin,
                                                                char_pixel_on, char_pixel_out, inside);
input clk, reset, inside;
input [10:0] hcount, tranx;
input [9:0] vcount, trany;

input newline, newframe;

input scale;

input [23:0] pixelin;

output char_pixel_on;

output [23:0] char_pixel_out;

//since we've taken out color identified as 'backgroud' and set
//these to black, we just check to see what the value of the
//incoming pixel is.  Not all zeros because RGB values are
//padded with ones when coming out of the ZBT.
assign char_pixel_on = inside & ~((pixelin[23:16] < 8'b00001000)
                                                    & (pixelin[15:8] < 8'b00001000)
                                                              &(pixelin[7:0] < 8'b00001000));


//doesn't really do anything but it's a layer of abstraction
assign char_pixel_out = pixelin;

endmodule


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

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

  parameter NDELAY = 650000;
  parameter NBITS = 20;

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

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

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
```

```
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

  reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

  reg [4:0] count;
  reg [7:0] reset_count;
  reg clock;
  wire dreset;

  always @(posedge clock_27mhz)
    begin
        if (reset)
          begin
            count = 0;
            clock = 0;
          end
        else if (count == 26)
          begin
            clock = ~clock;
            count = 5'h00;
          end
        else
          count = count+1;
    end

  always @(posedge clock_27mhz)
    if (reset)
      reset_count <= 100;
    else
      reset_count <= (reset_count==0) ? 0 : reset_count-1;

  assign dreset = (reset_count != 0);

  assign disp_clock = ~clock;

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

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

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

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

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

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

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

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

        8'h04:
         begin
           // Setup the control register
           disp_ce_b <= 1'b0;
           disp_data_out <= control[31];
```

```
            control <= {control[30:0], 1'b0};         // shift left
            if (dot_index == 0)
                  state <= state+1;
            else
                  dot_index <= dot_index-1;
          end

        8'h05:
          begin
            // Latch the control register data / dot data
            disp_ce_b <= 1'b1;
            dot_index <= 39;              // init for single char
            char_index <= 15;             // start with MS char
            state <= state+1;
            disp_rs <= 1'b0;              // Select the dot register
          end

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

  endcase

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

always @(nibble)
 case (nibble)
  4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
  4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
  4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
  4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
  4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
  4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
```

```
    4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
    4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
    4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
    4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
    4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
    4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
    4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
    4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
    4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
    4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
   endcase

endmodule

//////////////////////////////////////////////
//
//                      game_wrapper
//                      Description: This module provides a nice and tidy
//                              module for Akash to send and receive signals
//                              from. It contains all the wires to connect all the
//                              compentents of the game subsystem.
//                              For a better view of what is going where,
//                              look at the Block Diagram
//
////////////////////////////////////////////////



module game_wrapper(vclock,reset,up,down,left,right,hcount,vcount,player_pixel_on,pixel,
                                player_xcoord,player_ycoord, hsync, vsync, blank);

        input vclock;
        input reset, hsync, vsync, blank;
        input up;
        input down;
        input left;
        input right;
        input [10:0] hcount;
        input [9:0] vcount;
        input player_pixel_on;

        output [5:0] pixel;
        output [10:0] player_xcoord;
        output [9:0] player_ycoord;

        wire [15:0] analyzer1_data,analyzer2_data;

  assign phsync = hsync;
  assign pvsync = vsync;
  assign pblank = blank;

        wire [23:0] pixel;



        //bunches of bunches of wires

        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;
```

```
	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;
	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;
	wire player_fell;
	wire [10:0] player_xcoord;
	wire [9:0] player_ycoord;
	wire player_size;
	wire g_foward;
	wire g_backward;
	wire g_jump;
	wire g_crouch;
	wire g_stationary;
	wire player_pixel_on;
	wire [11:0] left_pixel;
	wire [2:0] collision_type;
	wire [4:0] new_sprite_tile;
	wire [4:0] new_sprite_row;
	wire [3:0] num_sprites_on_screen;
	wire [3:0] fb_sprite_number;
	wire [3:0] collision_sprite_number;
	wire [3:0] player_collision_type;
	wire [3:0] new_sstate;
	wire game_over;

	reg transparent_high;


//	assign player_pixel_on = (((hcount > player_xcoord) && (hcount < (player_xcoord + 100)))
//														&& ((vcount > player_ycoord)
&& (vcount < (player_ycoord + 100))));



	assign g_jump = up;
	assign g_foward = right;
	assign g_backward = left;
	assign g_crouch = down;
	assign g_stationary = ~(|{left,right});
```

```
// changing from 6 bit color to 24 bit color

wire [5:0] pixela;
assign pixela = player_pixel_on ? left_pixel[7:2] : 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]};

//
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]};

assign analyzer1_clock = vclock;


assign analyzer1_data = {collision_sprite_number[3:0],player_collision,player_collision_type,

sg_sprite_number[3:0],sg_sstate_o[10:8],sg_sstate_o[4]};


assign analyzer2_data = {o_pixel_data[2:0],fb_sprite_number[3:0],state[3:0],new_sstate[3:0],1'b0};



// hook those modules up!!

background_memory bm(
                .vclock(vclock),
                .world_column(world_column),
                .world_row(world_row),
                .background_tile(background_tile)
                );

background_generator bg(
                .vclock(vclock),
                .reset(reset),
                .vcount(vcount),
                .left_pixel(left_pixel),
                .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_pixel(left_pixel),
                                .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_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)
                                        );
//      assign sg_ram_we = 1;

        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_pixel(left_pixel),
                                    .o_pixel_data(o_pixel_data),
                                    .player_pixel_on(player_pixel_on),
                                    .collision_sprite_number(collision_sprite_number),
                                    .player_collision(player_collision),
                                    .player_collision_type(player_collision_type),
                                    .win_game(win_game),
                                    .player_xcoord(player_xcoord),
                                    .player_ycoord(player_ycoord),
                                    .g_foward(g_foward),
                                    .g_backward(g_backward),
                                    .g_stationary(g_stationary),
                                    .g_crouch(g_crouch),
                                    .g_jump(g_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),
                                    .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_pixel(left_pixel),
                                    .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

///////////////////////////////////////////////////////////////////
//
//  background_generator
//  description: holds the level memory, and based on the
//          center pixel of the screen, displays the
//          appropriate tiles.      Will update frame
//          buffer      after vcount > 720, it should update
//          the entire buffer by the time vcount > 780
//          letting the sprite generator overwrite
//          the background.
//
////////////////////////////////////////////////////////////////////
module background_generator(vclock,reset,vcount,left_pixel,background_tile,world_column,world_row,

        back_tile_pixel,back_pixel_type,back_pixel_transparent,back_tile_index,

        back_tile_row,back_tile_column,back_we,back_data,back_row,back_column,

        new_sprite,new_sprite_tile,new_sprite_row);

        input vclock; // 65MHz clock
        input reset;  // global reset
        input [9:0] vcount; // video verticle count
        input [11:0] left_pixel; // how far along in the game world is the left side of the screen?
        //background memory
        input [5:0] background_tile; // which tile to display as requested by world_column and world_row
        //tile memory
        input [5:0] back_tile_pixel; // RGB value of the pixel requested
        input [2:0] back_pixel_type; // character type of pixel requested
        input back_pixel_transparent; // high if the pixel is transparent

        //background memory
        output [7:0] world_column; //which column in our "world" do we want to choose?
        output [3:0] world_row;    // which row in our "world" do we want to choose?
        //tile memory
        output [5:0] back_tile_index;      // which tile to select
        output [3:0] back_tile_row; // which row of the tile
        output [3:0] back_tile_column; // which column of the tile
        //frame buffer
        output back_we; // background write enable
        output [8:0] back_data; // pixel data for background
        output [7:0] back_row; // low-res pixel row
        output [7:0] back_column; // low-res pixel column
        //sprite fsm
```

```
output new_sprite; // high if a new sprite just entered the screen
output [4:0] new_sprite_tile;
output [4:0] new_sprite_row;

wire [5:0] back_tile_index;
wire [5:0] back_tile_pixel;
wire [2:0] back_pixel_type;
wire back_pixel_transparent;
wire [8:0] back_data;
wire back_we;
wire [11:0] world_hcount_display;
wire [7:0] world_column;
wire [3:0] world_row;
wire [3:0] back_tile_row;
wire [3:0] back_tile_column;
wire new_sprite;

wire [7:0] back_column; // horizontal      screen count
wire [7:0] back_row; // vertical screen count
reg [7:0] last_updated_column; //holds the last world column checked for new sprites
reg [7:0] back_column_d1,back_column_d2,back_column_d0; //pipeline
reg [7:0] back_row_d1,back_row_d2,back_row_d0;//pipeline registers
reg [3:0] back_tile_row_d1,back_tile_column_d1,back_tile_column_d2; //pipeline
reg [11:0] world_hcount_display_d1,world_hcount_display_d2;    //pipeline
reg [7:0] world_column_d1,world_column_d2;         //pipeline


//to tile memory
assign back_tile_index = background_tile[5] ? 6'b000000 : background_tile; // if the object is a sprite, display the generic background
image
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)); // background has 55 lines to store new background
assign back_column = back_column_d2;
assign back_row = back_row_d2;


// to background memory
assign world_column = world_hcount_display[11:4]; // top 8 bits are which column to use
assign world_row = back_row_d0 >> 4;      // top 4 bits are which row to use
assign world_hcount_display = (left_pixel + back_column_d0); //which pixel column of the game world to work with


//incrementing the back_row and back_column across the screen
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


		// updated right before frame buffer
  assign new_sprite = ((back_column_d2 == 8'b11111111)		// at the right edge of the screen
									&& (background_tile[5] == 1)	// the tile is a sprite
(upper 32 tiles)
									&&(back_tile_row_d1 == 4'b0000)		// top of
the tile
									&&(back_tile_column_d2 == 4'b0000)	// left
edge of the tile
									&&~(last_updated_column == world_column_d2)
		// only send the signal once for each new sprite
									); // 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

//////////////////////////////////////////////////////////////////////
//
//  background_memory
//  Description: returns the tile number based on which tile in
//		the 15x256 game world grid is requested. Is implemented
//		in combinational logic now, but will be in a memory
//
//////////////////////////////////////////////////////////////////////
module background_memory(vclock,world_column,world_row,background_tile);

			input vclock; // 65 MHz clock
			input [7:0] world_column; // which column in the world is used 0..225
			input [3:0] world_row;	// which row in the world is used 0..15

			output [5:0] background_tile;	// 64 different tiles

			wire [5:0] background_tile;
//			reg [5:0] background_tile;
			//test combinational logic until the BRAM is working

//			assign background_tile = world_column[0] ? 6'b000001 : 6'b000000;		// for testing purposes, vertical bars
/*			assign background_tile_i = ((world_row == 11) && (world_column[2:0] == 0)) ? 6'b100001 : //an enemy every so often
											((world_row == 14) ||
((world_row >= 12) && (world_column[3]))) ? 6'b000001 : //ground
											6'b000000; // generic
background

			always @ (posedge vclock)
```

```
                    background_tile <= background_tile_i; //simulate the time it takes for the memory
*/



        backrom br(
                    .addr({world_column[7:4],world_row,world_column[3:0]}),
                    .clk(vclock),
                    .dout(background_tile)
                    );

endmodule

///////////////////////////////////////////////
//
//  FRAME_BUFFER
//  Description: holds every pixel value to
//               be outputted to the screen.
//                                      Low resolution 256x240
//
//                                      Takes in data and addresses from
//                                      sprites and background and saves them to memory
//                                      sprites and backgrounds should only start writing
//                                      after vcount > 720, so that when the screen is
//                                      displayed, it won't interfere with anything
///////////////////////////////////////////////

module frame_buffer(vclock,reset,hcount,vcount,back_row,back_column,back_data,back_we,sprite_row,

    sprite_column,sprite_data,sprite_we,o_pixel_color,o_pixel_data,o_sprite_number,
                                                    bottom_color_on);

    input vclock; // global clock
    input reset;  // global reset
    input [10:0] hcount; // horizontal screen count
    input [9:0] vcount; // vertical screen count
    input [7:0] back_row; // horizontal row to write background data (0-255)
    input [7:0] back_column; // verticle column to write background data(0-239)
    input [8:0] back_data; // background pixel data to be saved to the memory
    input back_we; // background write enable
    input [7:0] sprite_row; // horizontal row to write sprite data (0-255)
    input [7:0] sprite_column; // verticle column to write sprite data(0-239)
    input [12:0]        sprite_data; // sprite pixel data to be saved to the memory
    input sprite_we; // sprite write enable
    input bottom_color_on; // if high, colors the undisplayable portion of the screen

    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 [5:0] o_pixel_color;
    wire [2:0] o_pixel_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;
    wire [3:0] o_sprite_number;

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

    always @ (posedge vclock) begin
    // decimating the count for scaling
        if (vcount == 0) begin
```

```
                        vcount_by_three <= 8'b00000000;
                        count <= 2'b00;
                end
                else if (hcount == 1340)        begin                           // increment at 1340 instead of 1343 b/c
we don't know how long vcount is 1343
                        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


// choosing between drawing sprites and outputing data
        assign sprite_output_choose = (vcount > 720)            ? {sprite_row,sprite_column} : {vcount_by_three,hcount[9:2]}; // only assign
sprites after frame is displayed, also stretch the screen resolution to the desired resolution
        assign choose_we = (vcount > 720) ? sprite_we : 0; // if the screen is displaying, we aren't writing sprites to it

// output logic
//      assign o_data = (vcount <= 720) ? internal_output_data : bottom_color_on ? 9'b100010001 : 0; // display black if past 240 * 3,
character data 001 is non-lateral moves
        assign o_data = internal_output_data;
        assign o_pixel_color = (vcount <= 720) ? o_data[8:3] : bottom_color_on ? 9'b100010001 : 0;//  turn the color off if below 720
        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}), // 256 columns per row ((back_row * 256) + 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

/////////////////////////////////////////////////////////////////
//
//                      game_fsm
//                      Description : this module is responsible for the enemy logic,
//                                              sprite gravity, animation, and any other sprite
//                                              modifier that runs at 60 Hz
//
//////////////////////////////////////////////////////////////////


module game_fsm(vclock,reset,vcount,hcount,new_sprite,new_sprite_tile,new_sprite_row,
                                                xcoord_i,ycoord_i,sstate_i,sprite_tile_i,win_game,player_fell,
                                                left_pixel,xcoord_out,ycoord_out,sstate_out,sprite_tile_out,sprite_number,
                                                ram_we,player_size,num_sprites_on_screen,game_over);

        input vclock; //65 MHz clock
        input reset;  // global reset
        input [9:0] vcount;
```

```
input [10:0] hcount;
input new_sprite; // high if there is a new sprite on the board
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 [11:0] left_pixel;


//sprite ram
output [11:0] xcoord_out;
output [7:0] ycoord_out;
output [10:0] sstate_out;
output [4:0] sprite_tile_out;
output [3:0] sprite_number;
output ram_we;
// player stuff
output player_size; //1 means big
output [3:0] num_sprites_on_screen;
output game_over;


wire [11:0] left_pixel;
//sprite ram data
reg [11:0] xcoord_o;
reg[7:0] ycoord_o;
reg [10:0] sstate_o;
reg [4:0] sprite_tile_o;
reg [3:0] sprite_number;
wire ram_we;
//player data
reg player_size;

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;
reg [3:0] num_sprites_on_screen;

wire lose_game;
wire game_over;
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; //move off the top bit for slower moving enemies


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

// next state logic

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 machine
                        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
                                    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; // incrementing the sprite
                        if (state == s_8) sprite_number <= sprite_number + 1;
                        if (state == s_reset_2) sprite_number <= sprite_number + 1;

                        if (state == s_start) sprite_number <= 0;

//placing data in registers for the combinational logic
                        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 //always


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


          assign game_over = (state == s_lose); // send it to the player controller so it knows to stop working

          assign ram_we = ((state == s_3) || (state == s_7) || (state == s_reset_1)); // when do we right to the RAM?


// what to write to the sprite ram

          assign xcoord_out = (state == s_7) ? left_pixel + 256 : (state == s_reset_1) ? 0 : xcoord_o; //new sprite or old sprite?
          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'b0000000010 : (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_pixel,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

                    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          //generic enemy
                              // 3 tile types, change!!!!


                              //x coordinates
                              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_pixel) || (ycoord_hold >= 240) || (xcoord_hold >= (left_pixel + 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 //generic enemy


          //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.
//
///////////////////////////////////////////////////////////////////////////////////



module player_controller(vclock,reset,vcount,hcount,left_pixel,o_pixel_data,player_pixel_on,

     collision_sprite_number,player_collision,player_collision_type,win_game,fb_sprite_number,

     player_xcoord,player_ycoord,g_foward,g_backward,

     g_stationary,g_crouch,g_jump,player_size,player_fell,game_over);

     input vclock; //system clock
     input reset;
     input [9:0] vcount;
     input [10:0] hcount;
     // frame buffer
     input [2:0] o_pixel_data; //requested          pixel data
     input [3:0] fb_sprite_number;
     // from video stuff
     input player_pixel_on;
     input g_foward;
     input g_backward;
     input g_stationary;
     input g_crouch;
     input g_jump;
     // from sprite_fsm
     input player_size;
     input game_over;
```

```
// to sprite_fsm
output win_game;
output player_fell;
// to various
output [11:0] left_pixel;
// to video
output [10:0] player_xcoord;
output [9:0] player_ycoord;
//to sprite_generator
output [3:0] collision_sprite_number;
output player_collision;
output [2:0] player_collision_type;


reg win_game;
reg [3:0] collision_sprite_number;
reg player_collision;
reg [10:0] player_xcoord;
reg [9:0] player_ycoord;
wire jump_hold;
reg [5:0] jump_length;
reg player_fell;
reg [11:0] left_pixel;
reg [2:0] jump_state;
wire new_jump;
reg [2:0] player_collision_type;
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



	if (reset) begin
		collision_sprite_number <= 0;
		player_collision <= 0;
		win_game <= 0;
		player_xcoord <= 0;
		player_ycoord <= 0;
		jump_length <= 0;
		player_fell <= 0;
		left_pixel <= 0;
		jump_state <= 0;
	end

	else if (game_over) begin
		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				// player book keeping

		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 && g_foward) begin      //if going foward and hit something
                        player_xcoord <= player_xcoord - 2;
        end

        else if (o_pixel_data == 3'b001 && g_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;
                in_the_air <= 0;
        end

        else if (player_ycoord < 4)
                player_ycoord <= player_ycoord + 1;

        else if (player_size && (player_xcoord > 0)) // if the left corner of the player video is past 0
        begin
                player_xcoord <= 0;
                left_pixel <= left_pixel + 1;                           // move the screen over
        end

        else if (~player_size && (player_xcoord >= 151)) // if the corner of the small video is past half
        begin
                player_xcoord <= player_xcoord - 4;
                left_pixel <= left_pixel + 1;
        end

        else if (o_pixel_data == 3'b111)
                win_game <= 1;

        else if (vcount == 750)
                player_fell <= 1;       // if the player falls off the bottom of the screen


end //player book keeping

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

//small FSM for level to pulse conversion
        case(jump_state)
                2'b00 : jump_state <= g_jump ? 2'b01 : jump_state;
                2'b01 : jump_state <= g_jump ? 2'b11 : 2'b00;
                2'b11 : jump_state <= g_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(g_foward) // if the player has requested to go foward
                    player_xcoord <= player_xcoord + 4;

                if(g_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(vclock,vcount,hcount,reset,left_pixel,xcoord_i,ycoord_i,sstate_i,sprite_tile_i,

        sprite_tile_pixel,sprite_pixel_type,sprite_pixel_transparent,

        fb_pixel_data,player_collision_sprite_number,player_collision_i,player_collision_type,

        xcoord_o,ycoord_o,sstate_o,sprite_tile_o,sprite_number_o,ram_we,
                                        sprite_tile_index,sprite_tile_row,sprite_tile_column,

        sprite_row,sprite_column,sprite_data,sprite_we,other_sprite_collision,sprite_state,
                                        fb_sprite_number,new_sstate);

        input vclock; //65mhz clock
        input [9:0] vcount; //vertical count
        input [10:0] hcount; //horizontal count
        input reset; //global reset
        input [11:0] left_pixel;
//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 [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 ram_we;
//tile memory
		output [5:0] sprite_tile_index;
		output [3:0] sprite_tile_row;
		output [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 [3:0] sprite_state;
		output [2:0] new_sstate;

		//declaring the type of each output
		wire sprite_we;
		reg ram_we;
		reg [3:0] sprite_number;
		wire [3:0] sprite_number_o;
		wire [11:0] xcoord_o;
		wire [7:0] ycoord_o;
		wire [10:0] sstate_o;
		wire [5:0] sprite_tile_index;
		reg [3:0] sprite_tile_row;
		reg [3:0] sprite_tile_column;
		wire [7:0] sprite_row;
		wire [7:0] sprite_column;
		wire [12:0] sprite_data;
		wire [4:0] sprite_tile_o;

		//internal signals
		wire [11:0] sprite_column_i;
		reg [3:0] sprite_state; //high if finished with updating the sprites
		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;
		wire [2:0] new_sstate;
		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;
		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_9 = 9;
		parameter s_10 = 10;
		parameter s_11 = 11;

 // output logic

		// 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_pixel + 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}; //will only change when
ram_we is high (ie. state = s_3)


	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; // otherwise use what the FSM wants



	assign sprite_tile_o = (sprite_state == s_9) ? colliding_sprite_tile :
									(xcoord_hold > (272 + left_pixel)) ? 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;


//	assign ram_we = ((sprite_state == s_3) || (sprite_state == s_9));

 // next state logic
	always @ (posedge vclock) begin
		if (reset) begin							// what to do on a reset
			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;         // only start when vcount = 780
and hcount =1
                  s_1               : sprite_state <= s_6; //merely a waiting state to retrieve the data from the tile_memory
and frame buffer
                  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; // gives time to write to the sprite_RAM
                  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 //
finished with the run through the memory
                                                            : s_1;
                  default  : sprite_state <= s_start;
            endcase

            // increment the {sprite_number,sprite_tile_row,sprite_tile_column} count
            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
//      Description: Just holds the BRAM in a nice little package
//
/////////////////////////////////////////////

module sprite_ram(vclock,sg_address,fsm_address,sg_data_i,fsm_data_i,sg_data_o,fsm_data_o,sg_we,fsm_we);

input vclock;
input [3:0] sg_address,fsm_address;
input [35:0] sg_data_i,fsm_data_i;
input sg_we,fsm_we;

output [35:0] sg_data_o,fsm_data_o;


 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 tile_memory(vclock,reset,back_tile_index,back_tile_row,back_tile_column,

        sprite_tile_index,sprite_tile_row,sprite_tile_column,back_tile_pixel,back_pixel_type,
                                        sprite_tile_pixel,sprite_pixel_type,back_pixel_transparent,
                                        sprite_pixel_transparent);

        input vclock; // 65 MHz clock
        input reset;  // global reset
        input [5:0] back_tile_index;    // tile selector input for the background module
        input [3:0] back_tile_row;      // what row of the selected tile
        input [3:0] back_tile_column;   // what column of the selected tile
        input [5:0] sprite_tile_index;  // tile selector input for the sprite module
        input [3:0] sprite_tile_row;    // what row of the selected tile
        input [3:0] sprite_tile_column; // what column of the selected tile

        output [5:0] back_tile_pixel;   // output 6-bit RGB  value to background module
        output [2:0] back_pixel_type;       // 3 bit pixel type
```

```
	output back_pixel_transparent;  // high if the pixel is transparent
	output [5:0] sprite_tile_pixel; // output 6-bit RGB data value to sprite module
	output [2:0] sprite_pixel_type; // 3 bit pixel type
	output sprite_pixel_transparent; // high if the pixel is transparent

	wire [5:0] back_tile_pixel, sprite_tile_pixel;
	wire [2:0] back_pixel_type, sprite_pixel_type;
	wire sprite_pixel_transparent, back_pixel_transparent;

	// initializing the BRAM
	tilbram tilemem(
	.addra({back_tile_index,back_tile_row,back_tile_column}),		// index, then tile_row, then 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}),  // pixel RGB value, transparent?, type
	.doutb({sprite_tile_pixel,sprite_pixel_type,sprite_pixel_transparent})
	);


/*	// test output
	assign back_tile_pixel = back_tile_index[0] ? 6'b001100 : 0;
	assign back_pixel_type = 3'b000;
	assign back_pixel_transparent = back_tile_index[0] ? 0 : 1;
 */

	//test output
/*	assign back_tile_pixel = (back_tile_index == 0) ? 6'b001100 :
								(back_tile_index == 1) ? 6'b000011 :
								(back_tile_index == 2) ? 6'b010000 :
								(back_tile_index == 3) ? 6'b111100 :
								(back_tile_index == 4) ? 6'b110011 :
								 6'b100100;

	assign back_pixel_type = (back_tile_index == 2) ? 3'b011 :
								(back_tile_index == 3) ? 3'b010 : // stop
right
								(back_tile_index == 4) ? 3'b001 : // stop left
								 3'b000;
	assign back_pixel_transparent = 0;

	assign sprite_tile_pixel = (sprite_tile_index == 6'b100000) ? 6'b000000 :
									(sprite_tile_index == 6'b100001)
? 6'b110000 :
									(sprite_tile_index == 6'b100010)
? 6'b001000 :
									(sprite_tile_index == 6'b100011)
? 6'b000010 :
									0;
	assign sprite_pixel_type = (sprite_tile_index == 6'b100000) ? 3'b000 :
									(sprite_tile_index == 6'b100001)
? 3'b101 :
									(sprite_tile_index == 6'b100010)
? 3'b100 :
									(sprite_tile_index == 6'b100011)
? 3'b101 :
									3'b000;

	assign sprite_pixel_transparent = ~((sprite_tile_index == 6'b100001) ||

	(sprite_tile_index == 6'b100010) ||

	(sprite_tile_index == 6'b100011));
*/
```

```
//		assign sprite_pixel_transparent = 0;
endmodule


`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    13:40:26 11/19/06
// Design Name:
// Module Name:    gestures
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//============================================================
// Gestures - Akash Shah
//============================================================
//
//  This module handles gesture recognition for the character
//  playing the game.  It takes in the incoming pixels along
//  with their hcounts and vcounts and sums these up in registers.
//  Meanwhile, it sums up the number of pixels that are taken into
//  account (only when char_pixel_on is high is a pixel averaged.)
//		Next, it takes these sums and sends them to two dividers to
//  calculate the centers of mass along the vertical and
//  horizontal axes.  This center of mass value is then used
//  along with the incoming infomation for the repositioned grid
//  to determine where the center of mass lies in the plane.
//  Based on where the center falls in the grid, certain control
//  signals indicating gestures are set high (i.e. when center goes
//  high enough, this indicates a jump and g_jump goes high).
//============================================================
module gestures(clk, reset, char_pixel_on, inside, newline, newframe, hcount, vcount, minx, miny, maxx, maxy,
								centerx, centery, g_stand, g_jump, g_duck, g_left, g_right);

	input clk, reset, char_pixel_on, inside, newline, newframe;
	input [10:0] minx, maxx;
	input [9:0]  miny, maxy;
	input [10:0] hcount;
	input [9:0] vcount;

	output [10:0] centerx;
	output [9:0] centery;

	//gesture control signals
	output g_stand, g_jump, g_duck, g_left, g_right;

	reg [10:0] centerx;
	reg [9:0] centery;

	reg g_left, g_right;

	wire [10:0] new_center_x;
	wire [9:0] new_center_y;
```

```
    //registers storing sums of coordinates and num pixels
    reg [31:0] accumulator_x;
    reg [31:0] accumulator_y;
    reg [21:0] accumulator_num_pixels;

    wire [21:0] remx, remy;
    wire rfdx, rfdy;

    //divide to find center of mass
    pipelined_divider dividex(.clk(clk), .dividend(accumulator_x), .divisor(accumulator_num_pixels),
            .quot(new_center_x), .remd(remx), .rfd(rfdx), .ce(1'b1));
    pipelined_divider dividey(.clk(clk), .dividend(accumulator_y), .divisor(accumulator_num_pixels),
            .quot(new_center_y), .remd(remy), .rfd(rfdy), .ce(1'b1));


    always @ (posedge clk) begin
        if(reset) begin
            centerx <= 11'd0;
            centery <= 10'd0;
            accumulator_x <= 30'b0;
            accumulator_y <= 30'b0;
            accumulator_num_pixels <= 0;
        end

        //clear registers every frame
        else if(newframe) begin
            accumulator_x <= 30'b0;
            accumulator_y <= 30'b0;
            accumulator_num_pixels <= 0;
            centerx <= new_center_x;
            centery <= new_center_y;

            //was having problems getting this working properly in combinational logic
            //the < minx would work fine, along with the vertical limits, but the > maxx
            //would fail.  so...i just changed it, and this seems to work.
            g_left <= new_center_x < minx;
            g_right <= new_center_x > (minx + 11'h75);
        end

        //add up x and y values only for the character
        if(char_pixel_on && inside) begin
             accumulator_x <= accumulator_x + hcount;
             accumulator_y <= accumulator_y + vcount;
             accumulator_num_pixels <= accumulator_num_pixels + 1;
        end
    end

    //I left this in here because this is how it *should* be.  Xilinx can't seem
    //to get itself to wire the maxx wire properly...

    //assign g_left = centerx < minx;
    //assign g_right = ~(centerx < maxx);


    assign g_stand = ~(g_left | g_right);

    //analogous calculations for jump and duck work perfectly
    assign g_jump = centery < miny;
    assign g_duck = centery > maxy;


endmodule
```

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

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

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

   input        clk;           // system clock
   input        vclk;          // video clock from camera
   input [2:0]                 fvh;
   input        dv;
   input [7:0]                 din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output       ntsc_we;  // write enable for NTSC data
   input        sw;                         // switch which determines mode (for debugging)

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

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

   reg [9:0]   col = 0;
   reg [9:0]   row = 0;
   reg [7:0]   vdata = 0;
   reg                    vwe;
   reg                    old_dv;
   reg                    old_frame;              // frames are even / odd interlaced
   reg                    even_odd;               // decode interlaced frame to this wire

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

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

             if (!fvh[2])
               begin
                  col <= fvh[0] ? COL_START :
                           (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
                  row <= fvh[1] ? ROW_START :
                           (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
                  vdata <= (dv && !fvh[2]) ? din : vdata;
               end
     end

   // synchronize with system clock
```

```
  reg [9:0] x[1:0],y[1:0];
  reg [7:0] data[1:0];
  reg       we[1:0];
  reg             eo[1:0];

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

  // edge detection on write enable signal

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

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

  reg [31:0] mydata;
  always @(posedge clk)
   if (we_edge)
     mydata <= { mydata[23:0], data[1] };

  // compute address to store data in

  wire [18:0] myaddr = {1'b0, y[1][8:0], eo[1], x[1][9:2]};

  // alternate (256x192) image data and address
  wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
  wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};

  // update the output address and data only when four bytes ready

  reg [18:0] ntsc_addr;
  reg [35:0] ntsc_data;
  wire       ntsc_we = sw ? we_edge : (we_edge & (x[1][1:0]==2'b00));

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

endmodule // ntsc_to_zbt

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:09:41 11/16/06
// Design Name:
// Module Name:    overlay
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
```

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//============================================================
// Akash Shah
//============================================================
//			The overlay module is responsible for taking the game world and the true-life
//			video character and overlaying the two into a single image.  Based on information
//			it receives about the given pixel, if it determines that the pixel belongs to
//			the character's image, this pixel will be displayed.  Background pixels from the
//			video feed are removed and replaced with game world pixels.  Finally, any pixels
//		that lie outisde the area of the video feed default to being game world pixels.
//			Additionally, the overlay module accounts for scaling effects and handles the cases
//			where the video feed is at it's normal size, or the feed is decimated to its
//			half size.
//
////////////////////////////////////////////////////////////////////////////////
module overlay(clk, reset, char_pixel_on, char_pixel, game_pixel, shrink, hcount, vcount, centerx, centery, hshift, vshift, pixel_out, inside);
	input clk, reset, char_pixel_on, shrink, inside;
	input [23:0] char_pixel, game_pixel;

	input [10:0] hcount, centerx;
	input [9:0] vcount, centery;
	input [10:0] hshift;
	input [9:0] vshift;

	output [23:0] pixel_out;

	//registers for pipelining fun
	reg [23:0] pixel_a, pixel_b, pixel_c, pixel_d, pixel_e;
	reg select, select2;
	reg [23:0] pixel_out;

	always @ (posedge clk) begin
		pixel_a <= char_pixel;
		pixel_b <= game_pixel;

		pixel_c <= pixel_a;
		pixel_d <= pixel_b;

		select <= (char_pixel_on & inside);
		select2 <= select;


		pixel_e <= select2 ? pixel_c : pixel_d;
		pixel_out <= pixel_e;
	end
endmodule


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

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

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

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

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

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

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

   reg [1:0]   we_delay;

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

   // create two-stage pipeline for write data

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

   // wire to ZBT RAM signals

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

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

endmodule // zbt_6111
```