A Full Motion Dance Machine

by

Jonathan Stewart Burnham, Christopher Kanoa Hoffman, and Kevin Kar-Leung Miu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for 6.111

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2006

Author . . . .
Department of Electrical Engineering and Computer Science
December 11, 2006

Author . . . .
Department of Aeronautical and Astronautical Engineering
December 11, 2006

Author . . . .
Department of Mechanical Engineering and Department of Nuclear Engineering
December 11, 2006

A Full Motion Dance Machine

by

Jonathan Stewart Burnham, Christopher Kanoa Hoffman, and Kevin Kar-Leung Miu

Submitted to the Department of Electrical Engineering and Computer Science
on December 11, 2006, in partial fulfillment of 6.111

Abstract

A full motion dance machine was designed, tested, and implemented on a Xilinx Virtex 2 FPGA. The dance machine accepts a desired song and processes the real time beats of the music. The dance machine issues 1 of 144 possible dance commands to be followed by the user at given beats. User input is then scored by the dance machine through the recognition of various color swatches on the user's body. Additional user input and controllability are available via the buttons and the switches on the labkit. The buttons enable the user to switch between gameplay and calibration modes. The switches are used to set color thresholds in the calibration setting to allow the game to operate even in different lighting conditions. The game provides feedback to the user through a scoring system dependent on both the accuracy of the user's moves and the velocity at which the user moves. The audio and video modules were separately programmed prior to integration. The finite state machine code was added following the integration of the audio and video components because of their relatively high level of complexity. Afterwards, refinements were made to each of the modules. Timing constraints were carefully monitored during the programming, and their effects were evident in the performance of the system. Due to the fast clock speed necessary for the video display, even mild propagation delays could lead to undesirable results, such as the improper function of the video camera input and glitching in the display. Debugging was mainly done as a function of modeling the logical components necessary to construct the statements in our code and attempting to make optimizations in our algorithms. We were able to complete a challenging project that involved components such as video, audio, game control, and external chips. Expertise in Verilog and digital logic improved greatly over the course of the project.

Thesis Supervisor: Chris J. Terman
Title: Senior Lecturer

Acknowledgments

Jonathan

I would like to first thank Professor Terman for his dedication to this class and the effort he puts into it. It probably been my favorite class so far, and I credit this success to him and his engaging teaching, easy approachibility, and readiness to help.

I would also like to thank my teammates, Christopher Hoffman and Kevin Miu. They were a great team to work with, and I learned a lot from them. Kevin pushed our team to doing the best job that we possibly could, while still keeping the mood a pleasant one. Chris was a great worker and it was nice bouncing ideas off of him. They made this project into a success, and I thank them for it.

Gim Hom made the late nights at 6.111 lab a pleasant place to be. His quickness to help and bright attitude gave us hope through the dark hours of debugging.

I'd like to thank the 6.111 staff, especially Cassie Huang, who were diligent in checking up on us, giving us help and suggestions, and pushing us through the hard times.

Finally, a big thanks to the rest of the 6.111 class. You were a great group of people to work with, and I thoroughly enjoyed watching you create your projects. May we meet again in future labs!

Chris

I would like to thank Professor Terman for teaching and developing a great class. The labs assignments leading up to the final project were really well designed. The labs provided a good introduction to the major components of a digital system. I did not understand why so much time was spent discussing pipelining during lecture, but having seen its importance in my team?s project, I am now very grateful.

Gim Hom is in indispensable part of 6.111. He always seemed to be in lab late at night willing to discuss problems and suggest trouble shooting methods. Despite his lack of experience with the flash memory Jim took an interest in the implementation and provided a lot of assistance loading the data.

I would like to thank Cassie Huang for giving up her Saturdays to keep lab open,

for sharing her knowledge, and helping to guide us through the final project process.

My teammates, Jonathan Burnham and Kevin Miu, were great. Kevin came up with a great project idea and spent many hours in lab getting the video portion of the project to work. Jonathon did a great job of working out all the integration bugs and was very helpful in working out some of the audio difficulties. They were a lot of fun to work with throughout the debugging, testing, and trips to Garment District.

It was exciting to work in a lab with some many talented students. The other groups cheerfully provided assistance with technical problems and stress relieving game demos. Everyone was good about sharing the lab space.

Lastly, I would like to thank my girlfriend, Erin Munsell, for putting up with the long hours spent in lab during the last couple weeks of the project.

Kevin

Firstly, I would like to thank Professor Terman for teaching such a wonderful class. I truly enjoyed my experience, and 6.111 will certainly remain one of my favorite classes at MIT. I learned a great deal both in terms of digital logic and Verilog.

I would also like to thank Gim Hom. I got to know him very well from all the nights in lab, and our discussions from program implementations to Wii consoles provided me great insight. He truly cares for the students, and class would not have been anywhere as enjoyable without him there.

Cassie Huang and Javier Castro were also great TAs. They showed a wealth of understanding for the large class, and took sacrificed a great deal of their schedules to watch over the class.

Of course, I could not have completed the project without my two teammates, Jonathan Burnham and Christopher Hoffman. They produced a great deal of the audio components, game logic, and user interface. They put forth some valiant efforts to get the system integration done.

I must thank Irene Fan, who provided both great ideas and invaluable support. From late night dinners at WILG to endless talks about game implementation, she was a great help. I could not have asked to meet a better friend through the class.

Danny Vo was certainly a character if I've ever met one. His limitless energy, ideas, and dedication were greatly admired. Although we were not partners, we passed many ideas back and forth and helped each other immensely. He was a great help in terms of pipelining and learning about the area constraint settings of the Project Navigator software.

Eddie Fagin was a great motivator in pushing me to improve my game to be the best that it could be. He always set the bar, and I hope that I have met his standards.

Jeff Yuan was also a great help in terms of game logic and graphics used for our implementation. He was a great help with his coding expertise, and he kept the the mood in the lab cheerful.

Stephen Pueblo produced an awesome Donkey Kong game, and we had a lot of fun times encouraging each other with our projects.

And finally, I would like to thank the 6.111 class as a whole. We all had a blast through both the good and the bad. They put up with all my crazy outbursts, and we produced some really cool final projects.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 System Overview

The full motion dance machine presents a splash screen on startup. The splash screen provides three different choices. The user can choose to play the game, calibrate the colored swatches that are used in the game, or view the high scores. To scroll through this menu, the user presses the up and down buttons. The left and right buttons are used to set the difficulty level of the game. When the user is satisfied with his or her selections, the enter button can be pressed.

If the user chooses to calibrate the game, a screen showing the field of view of the camera pops up. The first screen reveals the red calibration. The camera shows a threshold based color filter. The thresholds can be modified by selecting different values with the switches on the labkit. The calibration is done rather intuitively in order of color magnitudes. For example, on the red calibration screen, the top four switches (7-4) are used to set the top four bits of an 8-bit threshold relative to the green value. The other four switches (3-0) are used to set the top four bits of an 8-bit threshold relative to the blue value. One example would be if the user inputs a value of 8'b10000001 on the labkit. On the red calibration screen, this implies that the red must be 8'b10000000 greater than the green value and 8'b00010000 greater than the blue value in order to be considered red. Once an appropriate value is found, the user hits button 1 to set those calibrated values. Pressing button 0 switches to the next calibration screen. After the red, the green, yellow, and teal calibrations follow. The green is compared against the red and the blue. In the yellow calibration, the green magnitude must be a certain threshold greater than the blue and the red magnitude must be a certain threshold greater than the blue. In the teal calibration, the green magnitude must be a certain threshold greater than the red and the blue magnitude must be a certain threshold greater than the red. Once the user is satisfied with all of the calibrations, he or she can press the enter button to return to the splash screen.

If the user moves the selection arrow down to the high score display mode, the high score will be displayed on the hex display on the labkit. The high score is no longer displayed once the user shifts the selection arrow off of the high score menu option.

The final option is to enter the game. When the game is entered, the system will run through a rapid beat detection mode, during which a brief audio tone is emitted. This sound is to confirm that the beat detection has in fact been completed for the song available on the labkit. Immediately following the beat detection, the song begins, and commands have been choreographed to scroll at the selected beats.

When the commands reach the left hand side of the screen, the user should be in the positions indicated by the command. Depending on the user size and position, a stick figure of varying size is drawn on the screen. A gridding, or quadrant, system is also provided based off the size and position of the user. The commands request that the user place his or her hands in specific quadrants at the beats. If the user places his or her hands at the right locations, the position based score will increment. If the user also fulfills the required movement in between beats, a velocity-based score will also increment.

When the song ends, the game returns to the splash screen. The game can be restarted from this point.

Selecting different difficulties provides varying patterns of dance commands. The lowest difficulty only requires two different dance positions, which the higher difficulties require greater variation in the user position and also greater coverage of dance moves in both x and y directions.

1.2 Concept Formulation and Execution

We entered the early stages of the class with various ideas for a final project. The ideas ranged from hardware-intensive projects to projects that were mainly software-based. Our main desire for the final project was to produce a product of which we would all be extremely proud and that we could reasonably finish. Some of the ideas that were considered but cut were a billiards analyzer and shooter and a baseball hitting instructor. After multiple design iterations, we settled on a full motion dance machine.

Dance machines have had some precedence in 6.111. [5, 1] Past projects in this field have had significant shortcomings, such as poor recognition of objects by the camera, poor sound quality and sound integration with the gaming interface, and insufficient storage of audio and video data. We set out to thoroughly answer each of those problem areas, and then to add additional originality through the gaming interface.

Our project contains solutions to many of the problems encountered in the past, and thus offers a much more enjoyable gaming experience. All the memories present in the labkit, such as the BRAM, ZBT, and Flash ROM are used. The Flash ROM has had little precedence in the past, and our incorporation of the Flash allowed us to store large amounts of high-quality audio data without sacrificing storage elsewhere.

Significant effort was put into the swatch detection such that the colors could be easily recognized and that the camera could be advantageous rather than a detriment to the gameplay, as it has been in the past. The colors are recognized and located through a variety of filters, and can be calibrated on location such that the game can be adapted to different lighting conditions. We sought to answer similar problems in other areas and create a dynamic and robust game.

Chapter 2

Deliverables and Final Performance

2.1 Original Checklist

A checklist of deliverables was proposed on November 22, 2006.

Checklist of Deliverables for the Final Project

Jonathan Burnham, Christopher Hoffman, and Kevin Miu

The basic features are marked with just a number.
More advanced features are marked with an additional *. A functioning project could result even without these goals or scaled-down versions of these goals.
The riskiest and less-likely goals are marked with a **. These goals would improve the game, but not in an absolutely necessary manner.

2.1.1 Video

Video Input

1. Modification ZBT video storage to accept 18-bit color. (Modification to Javy's Module by Kevin)

2. Modification ZBT and VRAM addressing to improve image resolution. (Modification to Javy's Module by Kevin)

3. Shift and horizontally reverse video image and clean up spurious pixels. (Modification to Javy's Module by Kevin)

Video Processing

1. Calibration and identification of 4 different colors. The colors are calibrated by setting relative thresholds. (Module written by Kevin)

2. Calculation of the center of mass for each of the 4 colors. (Module written by Kevin)

3.* Velocity determination of each of the center of masses. (Module written by Kevin)

4.* Creating a relation between movement and beat for each of the center of masses and then detecting beats. (Module to be written by Kevin)

Output Module

1. Display of the individual playing the game and the location of the four center of masses. (Module written by Kevin)

2.* Display of the reference quadrants. (Module to be written by Kevin)

3.** Display of swatch positions relative to quadrants. (Module to be written by Kevin)

4.** Display of a virtual "Stickman" emulating the dance of the user. (Module written by Jon)

5.** Make various bode parts of "Stickman" scalable. (Module written by Jon)

6. Creation of reference command images. (Kevin)

7. Script to process images and store them on the ROM. (Kevin)

8.* Display of 144 different reference dance commands. (Module written by Kevin)

9.* Scrolling of reference dance commands timed with beat. (Module to written by Kevin)

2.1.2 Audio

1. Script to convert WAV files into a two's complement data. (Kevin)

2. Beat verification in MATLAB. (Kevin)

3. Demonstrate data storing on Flash Rom and read back with logic analyzer. Data load demo will be either a partial demo with a BRAM or a full load using USB. (Module written by Chris)

4. Demonstrated music playback from Flash memory. (Module written by Chris)

5. Demonstrate that data stored on Flash ROM is persistent by power cycling labkit after loading data and then playing music. (Chris)

6. Demonstrate beat detection at real speed, by blinking a LED to the beat. (Module to be written by Chris)

7. Demonstrate storing detected beat in a BRAM. (Module to be written by Chris)

8.* Perform beat detection at a clock speed higher than the audio's 48Khz, preferably at 27Mhz. (Module to be written by Chris)

9. Demonstrate normal game operation, were music is played back and beat data is read out of BRAM on request from the controller. (Module to be written by Chris)

10.** Audio filtering to improve the sound quality. (Module to be written by Chris)

11.** Beat detection with band pass filtering to improve beat detection accuracy. (Module to be written by Chris)

12.** Demonstrate tone detection by displaying it on screed or using leds. (Module to be written by Chris)

13.** Demonstrate storing performing tone detection preprocessing and storing it in a BRAM. (Module to be written by Chris)

2.1.3 FSM

Stickman Calculations

1.** Averages shoulder y positions to determine the top of the user's torso. (Module to be written by Jon)

2.** Connects hand and shoulder center of mass positions while drawing a neck, head, torso, and legs in real time in a way that resembles a person. (Module to be written by Jon)

Beat Comparison

1. Is able to access the beat memory and compare the real-time user beat with the previously determined audio beat. (Module to be written by Jon)

2. Computes a reasonable score based on the difference in arrival of the beats. (Module to be written by Jon)

User Inputs

1. Is able to process buttons and switches set by the user to control the systems operation, including: scrolling dance pictures, sending calibration data to the video component, doing beat detection, selecting the proper song, stopping/playing back the song, comparing the quadrant position with the instructed position, and calculating a total score. (Module to be written by Jon)

2.1.4 USB Interface

1.** Using a laptop and a DLPDesign USB adapter to send data to labkit. (Module to be written by Kevin)

2.** Storing the USB data on the Flash ROM. (Module to be written by Kevin)

3.** Creating a means of addressing such that 60K (data limit of USB adapter) pieces of data can be incrementally stored on the ROM. (Module to be written by Kevin)

2.1.5 Other

1. Putting together a shirt with 4 reference colors. (Kevin)

2.** Improvement of the HEX display. (Kevin)

2.2 Successfully Completed Items

All items in the basic to advanced features section of the checklist were completed. Most items considered to be extremely risky were actually completed. The only items in the risky section that were not completed were the Karaoke components. The proposed improvements to the hex display were actually performed by Javier Castro, a teaching assistant in the class, before our group had an opportunity to tackle the problem.

The completed items that appeared on the original checklist are as follows:

Delivered Modules for the Final Project

Jonathan Burnham, Christopher Hoffman, and Kevin Miu

The basic features are marked with just a number.
More advanced features are marked with an additional *. A functioning project could result even without these goals or scaled-down versions of these goals.
The riskiest and less-likely goals are marked with a **. These goals would improve the game, but not in an absolutely necessary manner.

2.2.1 Video

Video Input

1. Modification ZBT video storage to accept 18-bit color.

2. Modification ZBT and VRAM addressing to improve image resolution.

3. Shift and horizontally reverse video image and clean up spurious pixels.

Video Processing

1. Calibration and identification of 4 different colors. The colors are calibrated by setting relative thresholds.

2. Calculation of the center of mass for each of the 4 colors.

3.* Velocity determination of each of the center of masses.

4.* Creating a relation between movement and beat for each of the center of masses and then detecting beats.

Output Module

1. Display of the individual playing the game and the location of the four center of masses.

2.* Display of the reference quadrants. (Module to be written by Kevin)

3.** Display of swatch positions relative to quadrants.

4.** Display of a virtual "Stickman" emulating the dance of the user.

5.** Make various bode parts of "Stickman" scalable.

6. Creation of reference command images.

7. Script to process images and store them on the ROM.

8.* Display of 144 different reference dance commands.

9.* Scrolling of reference dance commands timed with beat.

2.2.2 Audio

1. Script to convert WAV files into a two's complement data.

2. Beat verification in MATLAB.

3. Demonstrate data storing on Flash Rom and read back with logic analyzer. Data load demo will be either a partial demo with a BRAM or a full load using USB.

4. Demonstrated music playback from Flash memory.

5. Demonstrate that data stored on Flash ROM is persistent by power cycling labkit after loading data and then playing music.

6. Demonstrate beat detection at real speed, by blinking a LED to the beat.

7. Demonstrate storing detected beat in a BRAM.

8.* Perform beat detection at a clock speed higher than the audio's 48Khz, preferably at 27Mhz.

9. Demonstrate normal game operation, were music is played back and beat data is read out of BRAM on request from the controller.

2.2.3 FSM

Stickman Calculations

1.** Averages shoulder y positions to determine the top of the user's torso.

2.** Connects hand and shoulder center of mass positions while drawing a neck, head, torso, and legs in real time in a way that resembles a person.

Beat Comparison

1. Is able to access the beat memory and compare the real-time user beat with the previously determined audio beat.

2. Computes a reasonable score based on the difference in arrival of the beats.

User Inputs

1. Is able to process buttons and switches set by the user to control the systems operation, including: scrolling dance pictures, sending calibration data to the video component, doing beat detection, selecting the proper song, stopping/playing back the song, comparing the quadrant position with the instructed position, and calculating a total score.

2.2.4 USB Interface

1.** Using a laptop and a DLPDesign USB adapter to send data to labkit.

2.** Storing the USB data on the Flash ROM.

3.** Creating a means of addressing such that 60K (data limit of USB adapter) pieces of data can be incrementally stored on the ROM.

2.2.5 Other

1. Putting together a shirt with 4 reference colors.

Chapter 3

Module Description and Implementation

3.1 Video

The video component was a vital component to our concept of a dance machine, since we desired a system that could function without the mass of sensors and other restrictions commonly seen in dance machines, such as the pads in Konami's Dance Dance Revolution.

The main function of the video component is to take a live feed from a camera and identify the presence and location of predetermined colors. Various filters were applied to improve the recognition of the colors. The color coordinates are then passed to the game logic to be processed.

The source of the video was a camera based on the NTSC-standard. The camera was kindly provided by Arlin and the stockroom staff. A picture of the camera is shown in Figure 3-1.

3.1.1 Video Input

Conveniently, the labkit has a composite video input port. The port is pictured in Figure 3-2. The camera plugs directly into the port. The signal that is received from

Figure 3-1: NTSC Video Camera

the camera is an interlaced NTSC signal. NTSC is a standard in North America and some Asian countries, including Japan. NTSC produces about 30 interlaced video frames per second, which is sufficient for the human eye to appear as uninterrupted motion. Because of the interlaced signal and the alternating even and odd scanlines, an image is produced at 60 Hz. The camera outputs an image that is 720 pixels wide by 480 pixels tall.

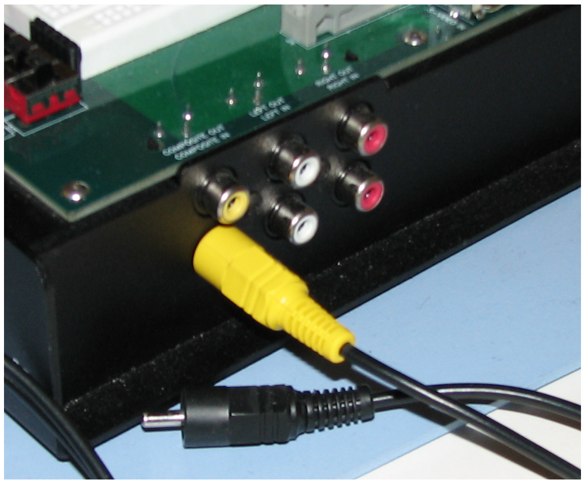


Figure 3-2: Video Input Connection to Labkit

3.1.2 Video Storage - ntsc2zbt.v

The basic skeleton of the ntsc2zbt.v module was prepared by Professor Chuang. Modifications were made to the code in order to shift the location of the video feed and improve the resolution of the video image.

In order to shift the image, the parameter ROW_START was modified. The parameter moves the video feed down on the screen, which was necessary for the display of the dance command images, which were issued at the top of the screen.

The data input was also modified to a 24-bit number, of which 18-bits are stored. Originally, the data input was an 8-bit luminance value that was used to display black and white images. With the 18-bits that are stored, there are 6-bits each of red, green, and blue.

Two sets of these 18-bit values are stored at each ZBT address, which is convenient because of the fact that the ZBT memory is 36-bit. Registers had to be increased in size in order for the image data to be stored.

3.1.3 Video Display - vram_display.v

The vram_display module was modified in order to reflect an improvement in the resolution that was to be displayed. The vram_display was set to have matching bits with the ntsc2zbt module such that the pixels would map correctly.

3.1.4 Color Filtering - cm_calculator.v

The output of the camera is in luminance and chrominance, or YCrCb. Although luminance and chrominance have been shown to be useful in color detection, our objective was to base our calibration on a more intuitive standard such as RGB. [6] A large number of people have had some familiarity with the RGB standard, from photography, graphics programs, to biological/neurological studies.

The first attempts made at color filtering involved looking at colors with high densities in one of the colors of RGB, such as red. An absolute threshold was set for the red value, while the other values were mainly disregarded. A problem with such a filtering method was immediately evident, as the lights, reflections off the white tables, and other noise sources continued to have deep presence in the image. The reason behind such noise was clear: We were simply looking for the presence of a single high value while placing no constraints on the other values. Such a method resulted in any color with a high red value passing. For example, the lights and the white table top have very high values of red, green, and blue, and thus they could still be seen. Orange and yellow objects could also be seen due to their large red component.

A second method was devised where relative thresholds were examined. Two thresholds were used for each color. For example, when looking at red, the only color to be accepted from the camera would need a red component that was a certain threshold above the green value, and then also a red component that was a certain threshold above the blue value. These thresholds were input as 4 bit numbers on the labkit. Since there were two thresholds that had to be determined, 8 total switches had to be set. The use of two relative threshold produced great results.

The final method of filtering was a window filter. A fundamental tradeoff in the threshold filter was the inability to filter out background noise to an acceptable level without greatly reducing the pixels recognized on the target object. A window filter was devised such that the pseudo-random background noise would be disregarded, while the high density color of the target object would be maintained. The window filter looks at a window of 5 points, and reassigns the value at the center point to the lowest value of the 5 point window. In order for a point to return a high value in such a case, the point itself and all the surrounding points would have to be high. Such a method greatly improved noise rejection.

3.1.5 Center of Mass Determination - cm_calculator.v

Few projects have utilized center of mass calculations in the past, and those that did utilize it have had varying degrees of success. [6] Even those projects that have performed center of mass calculations have relied on finding a single center of mass or two center of masses of the same color. Due to the complexity of our project, we required four separate center of mass calculations.

The center of mass determination was aided by the ability to generate pipelined dividers using Core Generator. The dividers were rather large, as both dividers used (one for horizontal calculations and one for vertical calculations) were 28-bit. We performed the sizing of the dividers with the following calculations.

Horizontal Divider

$$(720 \times 480) pixels \times \frac{(1024 - 720) + 1024}{2} pixel^{-1} = 229478400$$

$$2^{28} = 268435456 \geq 229478400$$

Vertical Divider

$$(720 \times 480) pixels \times \frac{(1024 - 720) + 1024}{2} pixel^{-1} = 229478400$$

$$2^{28} = 268435456 \geq 229478400$$

In both cases, a 28 bit divider can easily handle the dividends possible.

The dividers were sufficiently small such that the computations could be completed in the provided by the back porch, sync pulse, and front porch. [4]

After the dividers were generated, the CE signals on the dividers were activated. These signals were used such that the dividers could be turned on when the screen was displaying active pixels.

3.1.6 Quadrant Detection - grid_detect.v

The grid detection takes in the four centers of masses and some precalculated information about them, such as the midpoint of the two shoulder center of masses. The screen is divided into a series of columns and rows based off the distance between shoulders. It then performs a binary search on the two arm center of masses to find out what column and row the arms are in. The row heights were fixed by the height of the camera field of view, or 480 pixels.

3.1.7 Block Diagram

A block diagram of the implementation used in the project is shown in Figure 3-3.

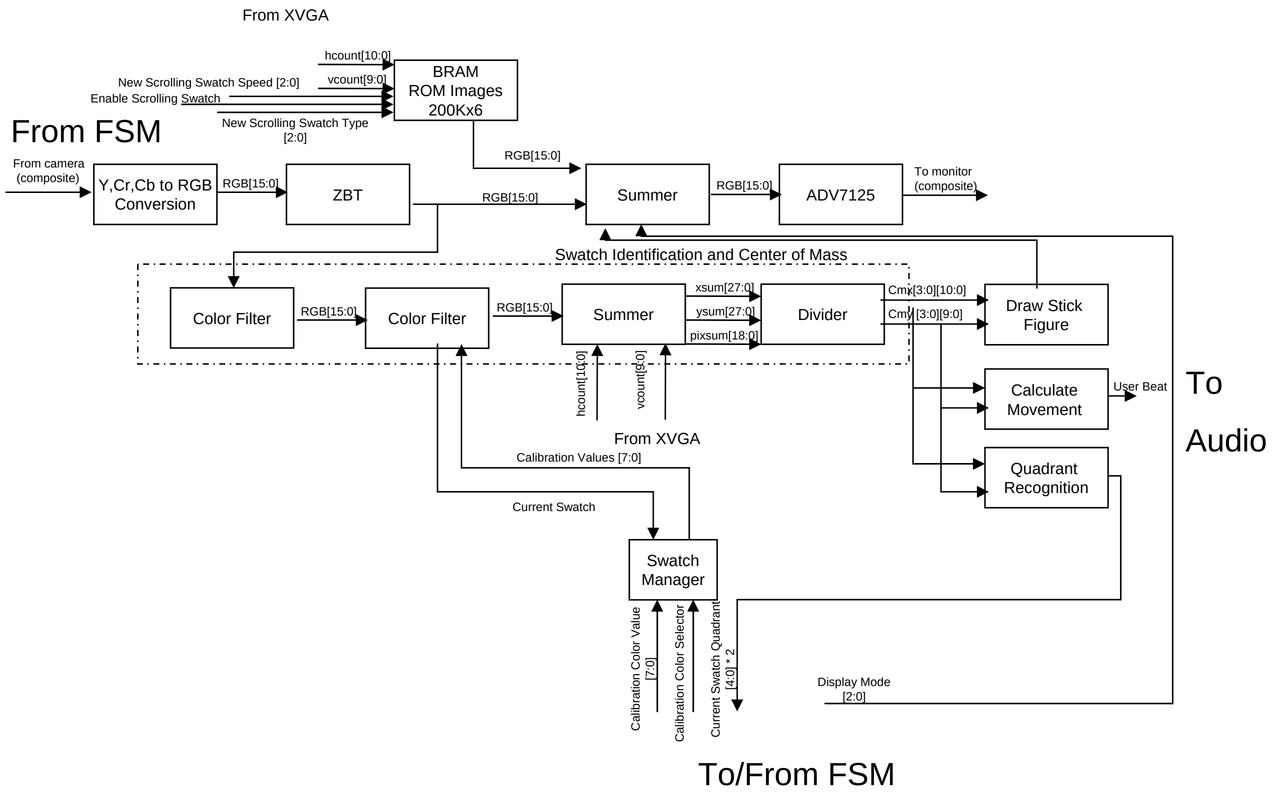


Figure 3-3: Video Block Diagram

3.2 Audio

3.2.1 Clock_16th

The clock 16th module generates a 16Hz enable signal based on an input 48 kHz clock. This module is needed because beat data is stored in the bram every sixteenth of a second. The module functions by incrementing a counter every click of the 48 kHz clock and generating a pulse every three thousand counts. An instance is used for playback and for beat storage. During playback, the module runs off of the 48 kHz ready signal from the AC97 and generates a true 16Hz clock. During high speed detection the input is the new ready from the high speed enable module and the output is a sudo-16Hz clock.

3.2.2 High speed enable

High speed enable generates a clock for the audio modules. Normally the AC97's ready signal serves this purpose, but high speed beat detection requires a faster clock. The situation is complicated by the fact that the flash memory must first be run at a faster speed for detection and then at normal speed for playback. The High speed enable allows the flash to be run at multiple speeds and provides a way of disabling the memory writing module. The module takes as input the 27 MHz clock, AC97 ready, playback start, playback stop, and enable. Enable is used to switch between the two modes and is high during beat detection. The output is the new ready signal. Two instances of this module are uses. An instance of this module is used to control the beat process and memory modules and an instance is used for the flash. During the beat detection mode the two instances should generate a roughly synchronized output, but the output can be expected to be off by a fraction of a clock cycle. Given the nature of audio data and the fact that beat detection involves averages over large samples it should not make a difference. During playback mode the ready signal to the beat processing and detection modules is turned off and the flash is feed AC97 ready.

3.2.3 Beat BRAM

A BRAM is used to store the beat found during high speed detection and play them back during song playback. It is necessary to perform the beat detection in advance so that the user can be told what to expect. A look ahead beat is used to start command images scrolling across the screen a couple seconds before the actually beat. A BRAM with one input port and two output ports is used so that during playback both a current beat and a look ahead beat can be output to the interface. The same address is used for both the input and one of the outputs ports. The two addresses for this one port are generated by different modules so the addresses are added together and then feed to the BRAM. The modules agree to output a zero when not in use.

BRAM space on the labkit is limited and most of the space is being used to store images, so the beats need to be stored in as little space as possible. The beat processing module outputs a new beat hundreds of times per second but a human can't actually detect a hundred of a second It was decided that the best trade off was to store a new beat every sixteenth of a second. That means that at most a beat could be off by a sixteen of a second, which should not really be noticeable and for a four minute song only three thousand eight hundred and forty data points of width one need to be stored. This means that the BRAM will only use one out of one hundred and forty-four BRAM blocks. The beat memory module handles the necessary down sampling and control of the data writing.

Beat Memory The beat memory module is responsible for taking the beats from the beat processing module and storing them in a BRAM. The module is needed because beat processor outputs beats at 187 Hz and beats are stored in the BRAM at only 16 Hz. If a beat is input at anytime during the 16 Hz period a beat is stored in the flash memory. While the beat memory module is active an enable signal is output to the high speed enable modules which tells them to speed up the audio clock so that beat detection can be done in a couple seconds instead of minutes.

3.2.4 Beat Process

Theory of Beat Detection

The goal of a computer beat detection algorithm is to mimic the beat detection performed naturally by humans. In designing a beat detection algorithm the first step to quantify what constitutes a beat. Having no experience with beat detection a Google search was performed to learn more. An algorithm based on gamedev.net's beat detection article was ultimately selected. [3] The theory is that a beat is a loud spike in the song's power (volume). Conceptually you can reason what a beat should look like by assuming it is made by a drum. Striking the drum impulsively increases the energy of the drum and causes it to vibrate generating sound. A drum has some damping so the oscillation is damped resulting in the amplitude decaying. The amplitude of the oscillation is known to be logarithmically related to volume so the volume will jump from zero to a maximum and then slowly decay back to zero. This raises the question of what point in the beat should be detected. For now it will be assumed that the maximum represents the instant detected by a human. This assumption will need to be evaluated once a beat algorithm is developed. A simple beat detection algorithm could be made by recording a beat whenever the power crossed a threshold however this method has several flaws. Variations in volume within a song or between songs require changing the threshold. A simple threshold does not take advantage of the sharp spike in power that is assumed to occur. To solve these problems an algorithm can be created that takes advantage of moving averages. We can define an instantaneous power and a baseline power. The instantaneous power is the average power over a small sampling period and baseline power is the average over a much longer period. The baseline being a historical average will tend to lag behind the instantaneous average. Sharp spikes will result in the instantaneous average initial being much larger than the baseline. Slow rises to high power however don't result in much off a difference between the instantaneous and baseline values. This means that the ratio of the instantaneous to baseline power is a good indication of beats. A threshold can be defined as a ratio of the two powers and a beat recorded when the actually ratio first exceeds the threshold. Intuitively the instantiations sample should be at least as long as the period of the lowest frequency of interests. The source suggests that the baseline average be taken over a time of roughly a second.

Some songs may not respond well to enough to the methods described above. Difficulties can arise from noise in the song, very loud vocals, or multiple beat sources that are not highly synchronized. Problems due to vocals can be remedied because a human is only capable of singing in a limited frequency that does not generally overlap the frequency at which beats are likely to occur. This frequency range can be filtered out of the signal before performing beat detection. Similar methods can be employed to deal with some other sources of noise. Multiple beat sources in a similar frequency range are a more difficult problem to deal with. Selecting a high threshold can cause weaker sources to be ignored but some desired beats may be missed and the ability of a generic algorithm to be used on multiple songs is reduced. If the multiple beats are reasonable synchronized, for example several people clapping a debouncer with a sufficient waiting period should filter our most of the additional beat detections. It should be noted that it is important for the output of the debouncer to go high as soon as the input first goes high to avoid latency. In practice the waiting period can be quite large say half the expected time between beats, which could be a quarter second. Beat debouncing is easily implemented and will be included in the basic beat detection functionality. Frequency filtering requires determining the frequencies to be filtered and the implementation of Fourier transforms to perform the filtering. Fourier transforms require a lot of computation. Performing these calculations on an FPGA would likely require significant pipelining and use a large portion of the available logic gates. Due to the difficulties associated with filtering; it will only be used if satisfactory performance cannot be achieved without filtering.

The beat detection algorithms were verified in MATLAB using the song "Yeah" by Usher. Figures 3-4 and 3-5 show the results of the beat verification.

The figures show that beats are regularly detected by the algorithm. there is some variability to the detection, but that is expected from the music. The variability also shows how such real time analysis differs from methods such as using FFT's to find particular rhythms in a song.

Implementation (of beat process)

The beat algorithm is implemented in the beat process module. The implementation is based on the above described theory with modifications make as needed to fit the constraints of the FPGA. The audio data being used is signed 8bit audio sampled at 48 kHz. Using a 48 kHz sampling rate would require averaging over a large number of samples and using a lot of registers. It is assumed that beats will tend to occur in lower frequencies, below 6 kHz, which means that no needed data should be lost by

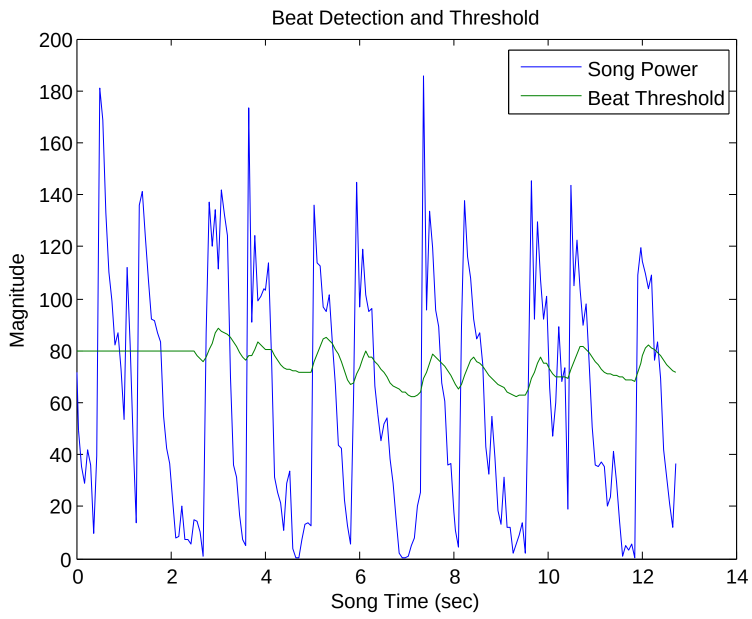


Figure 3-4: Beat Threshold and Local Beat Power of a Selection of ”Yeah” by Usher

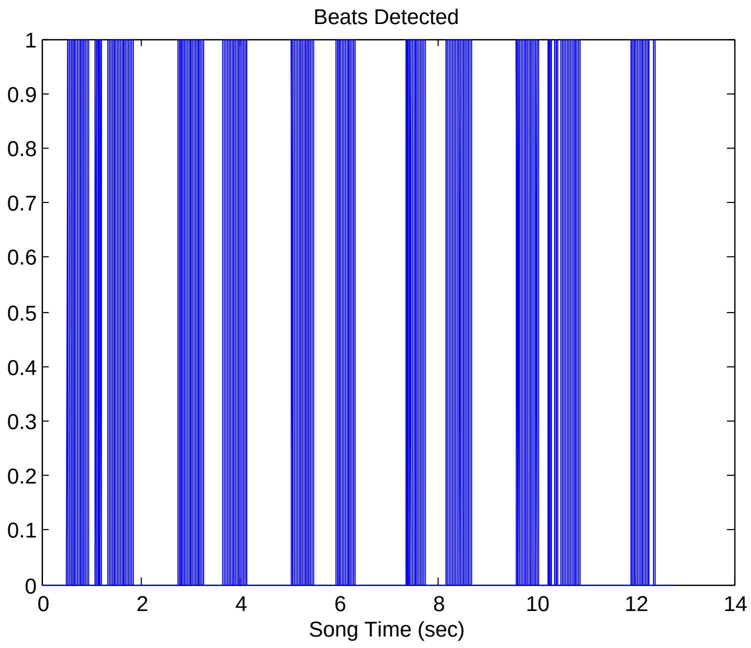


Figure 3-5: Beats Detected in ”Yeah” by Usher

down sampling to 6 kHz. Down sampling should reduce the required processing time and chip space by a factor of 8 or more if everything else is kept the same. The three main things to be calculated are the power of each sample, instantaneous power, and baseline power. The power of a sample is simply its value squared. The instantaneous power is then the average of a small number of the most recent samples. 32 samples are included in the instantaneous average. 32 samples is smaller than recommended by the internet source but was chosen because it is a power of 2 which makes calculations easier, requires smaller registers and seemed like an appropriately sized samples based on some rough calculations. It was initially decided that the baseline should be the average of the last 32 instantaneous samples however that number was ultimately increased to 64 to improve performance. This means that each instantaneous sample is averaged over 0.005 seconds and the baseline is averaged over 0.34 seconds. The modules consist of 64 registers to hold the instantaneous samples that make up the baseline. Each new sample is added to a temporary register and once 32 samples are added to the temporary registers all the registers are shifted and the temporary value is assigned to the first register. The registers are then averaged and compared to the value of the first register. This process would ordinarily require a large number of big registers. The size of the registers is reduced by right shifting the result of each major calculation before storing it to a register. The maximum combinational delay is reduced by using a deep pipeline. However the resulting latency is less that a thousandth of a second which is undetectable by a human. The beat is then determined by multiplying the baseline by a threshold and comparing the value to the current instantaneous sample. When the instantaneous sample is larger a beat is detected. Once a beat is detected the output goes high for a clock cycle and the debouncer is activated to prevent a second beat from be output during a delay period. The threshold was determined by setting it equal to the lab kit's switches and trying different thresholds. Ultimately a threshold of 4.5 and a debouncer waiting period of 0.3 seconds were chosen.

3.2.5 Test FSM

The Test FSM module is modified from a module written by Nathan Ickes. This discussion of test FSM is limited to the version used in the game. A second more complicated version can be found in the flash memory programmer. Test FSM provides a user friendly interface for accessing the flash memory. The inputs are a clock, address reset, and new address. The flash memory only outputs a new sample every two clock cycles. To get the data out at a normal rate will clocking off of ready the address is incremented by two instead of 1. This means that 24 kHz data is being output even though 48 kHz data is stored on the flash. When address reset is high the memory address is set equal to new address. The outputs are sixteen bit data, the current flash address, and control flash control signals.

3.2.6 Storage Tester

In order to test the Beat storage and playback it was necessary to create a small state machine. Storage tester was originally intend only for audio testing purposes, but was ultimately adapted for use in the final program. Storage tester has four states that collectively start high speed beat detection, wait for beat detection to end, start play back, and wait for playback to end. The storage tester is responsible for generating the control signal to set the appropriate flash address at the start of playback and beat detection. The actual addresses are provided by the interface module, which in the final version also provides a start signal to the storage tester.

3.2.7 Audio Out

The AC97 module provided for lab 4 is used unmodified to output audio. The 8 bit data coming out of the flash memory is directly feed to the AC97 module. Due to a lack of buttons on the lab kit the ability to adjust the audio was removed.

3.2.8 Flash Memory

Storing a two minutes song requires around 5 MB of memory which is more than is available in the BRAMs of ZBT. The only memory on the lab kit that can store this much data is the flash memory. Normally the flash memory is not used due to its higher delays and slow data write rate, write once policy. However none of these are really problems for audio usage. The main problem is how to load the flash memory with data. There is no built in way to program it through the Xilinx software. The first method tried was building BRAMs and loading data on to the flash in small blocks. This proved okay for getting a 10 second proof of concept sample onto the flash, but building and loading BRAMs for a two minute song would literally take days. After considering using a serial connection to transfer data, a USB connection was used. Programming the flash with the USB was a little tedious due to the size of the flash memory erase blocks and the maximum file size supported by the PC software, but allowed programming in under an hour.

3.2.9 Block Diagram

A block diagram of the audio components and the data flow between the components is shown in Figure 3-6.

3.3 FSM

The buttons and switches, which are depicted in Figures 3-7 and 3-8 were used as means of interfacing with the user.

The lights on the labkit were also invaluable in debugging. They are shown in Figure 3-9

3.3.1 Scroll Control

The scroll_control module looks ahead in the beat memory to determine when it should start scrolling another dance picture. It looks forward a constant amount in the memory, and when a beat is detected, it commands the begin scrolling module to start scrolling a new picture, unless one is scrolling already, so that it will reach the left side of the screen just as the beat is being played.

3.3.2 Begin Scrolling

The begin_scrolling module starts calculates the offset from the right side of the screen that the dance picture needs to be. It begins the calculations on command from the

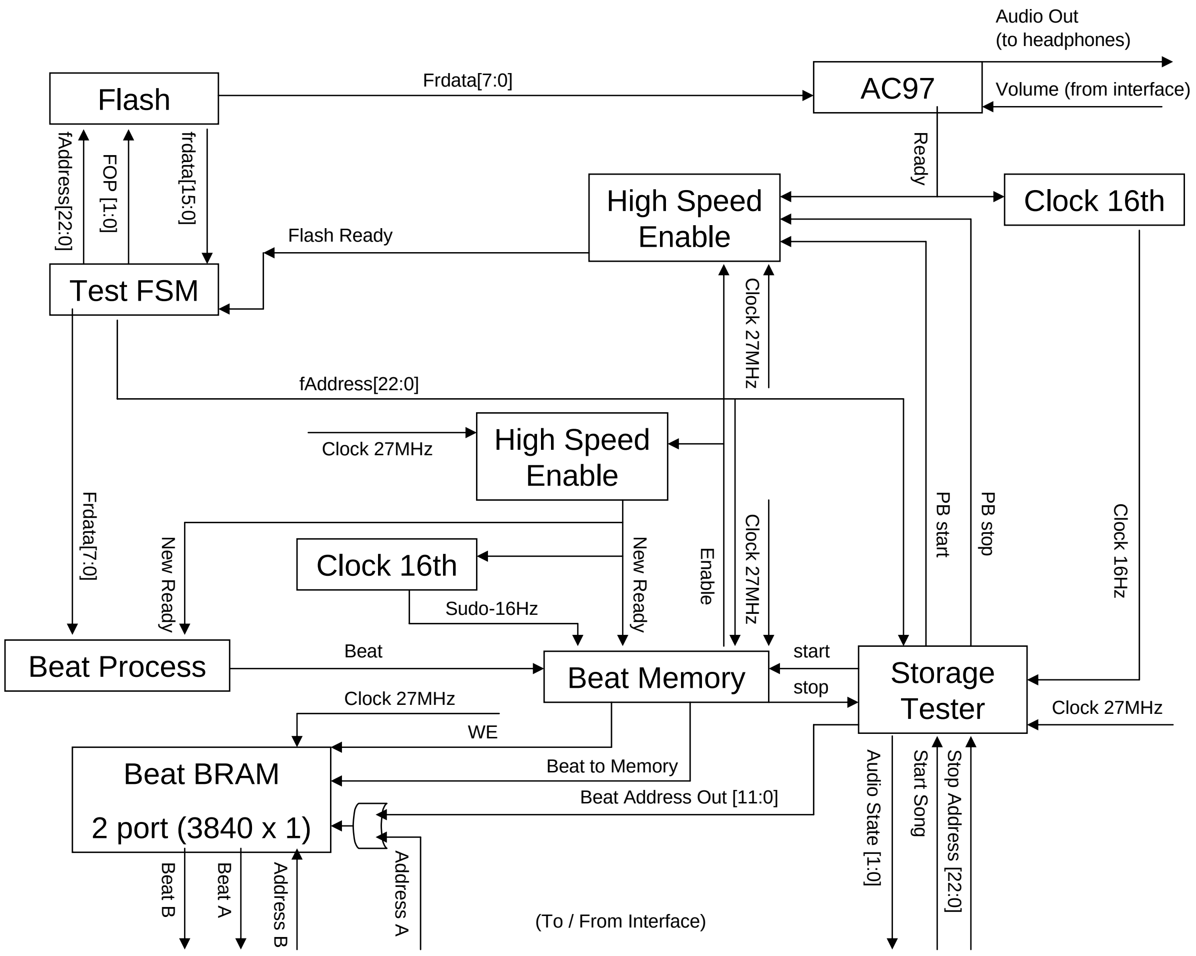


Figure 3-6: Audio Block Diagram

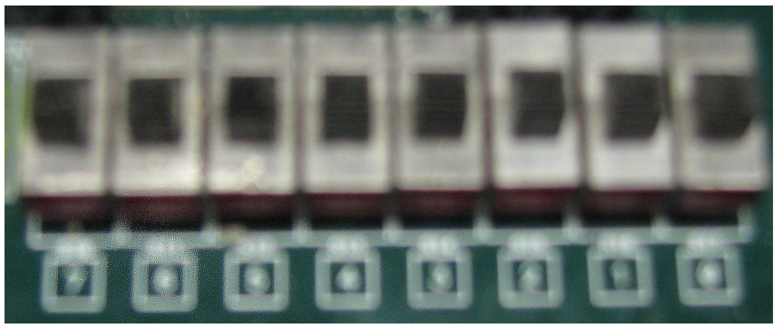


Figure 3-7: Labkit Switches

scroll_control module, and signals the scroll_control module when it has scrolled the picture all the way to the left side of the screen. The speed at which it scrolls is constant, but can easily be changed by adjusting one of its inputs.

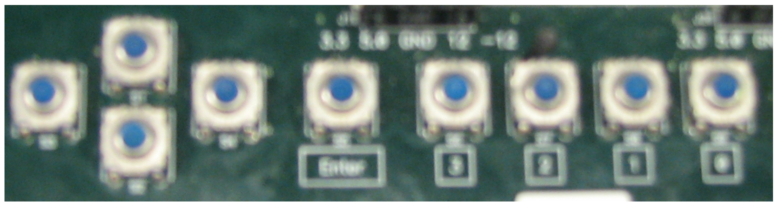


Figure 3-8: Labkit Buttons

Figure 3-9: Labkit Displays

3.3.3 Compare Beats

The compare_beats module checks every new beat to see if the user has moved enough to set the user_beat input from the calculate_cm module. If it has, the score from the beat score is incremented by 50 points. If it gets a reset, the score is set to zero.

3.3.4 Grid Score

Everytime a dance picture reaches the left side of the screen the grid_score module checks to see if the users hand positions match those specified by the dance picture being displayed. Since the grid has a different numbering scheme for the dance pictures than for the hand positions, two large case statements for each hand are used to determine if the hand is in the quadrant listed in the picture. If both hands are in the proper grids, the grid score is incremented by 200. The grid_score module also outputs two lines for the right and left hands which are high whenever its respective hand is in the grid shown in the dance picture. These lines are then displayed on the LED panel on the labkit.

3.3.5 Addypicker

The addypicker module is responsbile for calculating the address in the dance picture BRAM for each hcount and vcount. First, the addypicker module decides what the current dance picture it should display. Everytime the scroll_control module toggles the new_picture line, addypicker picks the next picture in the choreography routine it is currently in, which is determined by the difficulty level decided by the user and supplied from the interface module. It determines the next picture by working its way through a large case statement. The actual address is a function of the picture, hcount, vcount, and the offset from the right side of the screen supplied by the begin_-scrolling routine. The address equation is computed every clock cycle.

3.3.6 Splash Screen

A splash screen was generated in Adobe Photoshop and processed in MATLAB. The splash screen allows navigation between different modes of the game. The image used for the splash screen is shown in Figure 3-10.

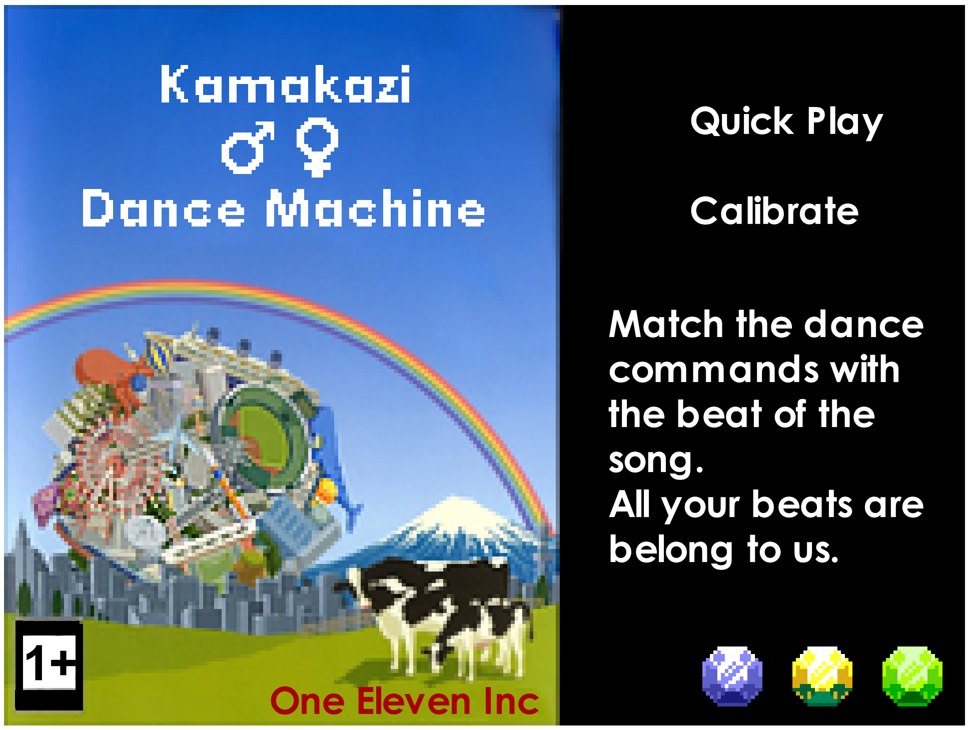

Kamakazi
♂ ♀
Dance Machine
1+
One Eleven Inc
Quick Play
Calibrate
Match the dance commands with the beat of the song.
All your beats are belong to us.

Figure 3-10: Game Splash Screen

3.3.7 Block Diagram

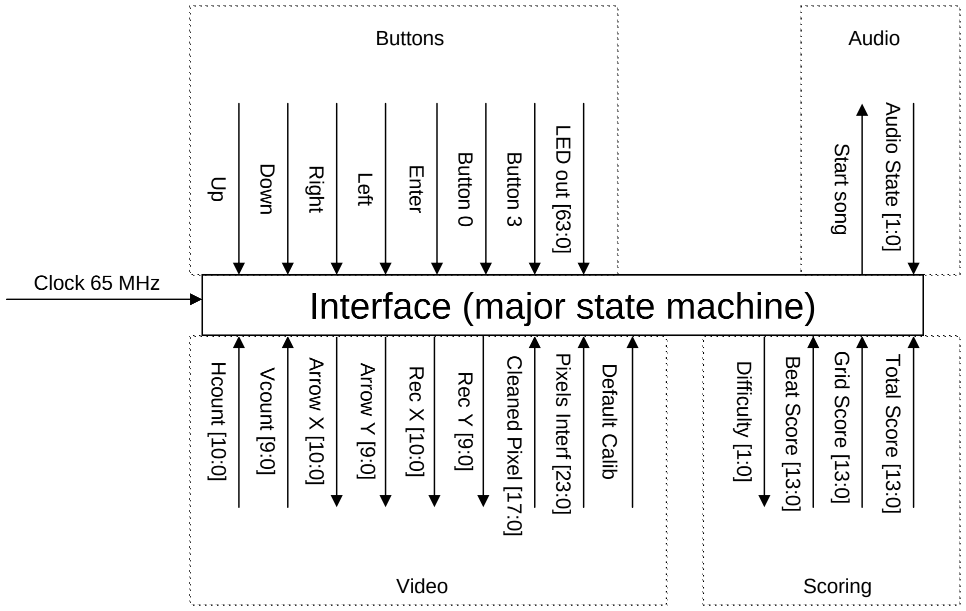


Figure 3-11: Interface Block Diagram

3.4 USB Interface

The motivations for an interface to transfer data from a computer via USB to the labkit resulted from the need to store large amounts of data in the Flash ROM. The building of ROMs using the Core Generator takes an excessive amount of time. Transferring data through the USB is very fast, so we were even able to load the song data on multiple lab kits for parallel testing of our code.

We had examined various sources for sending large amounts of data from a laptop or the lab computers to the labkits. We researched serial interfaces, but then Professor Terman was kind enough to show us some USB adapters that he had obtained.

The USB adapters were fabricated by DLP Design, who specialize in a wide range of USB, RF, and FPGA products. We used the DLP-USB245M model, which has a very basic interface for our purposes. A photograph of the USB adapter is shown in Figure 3-12.

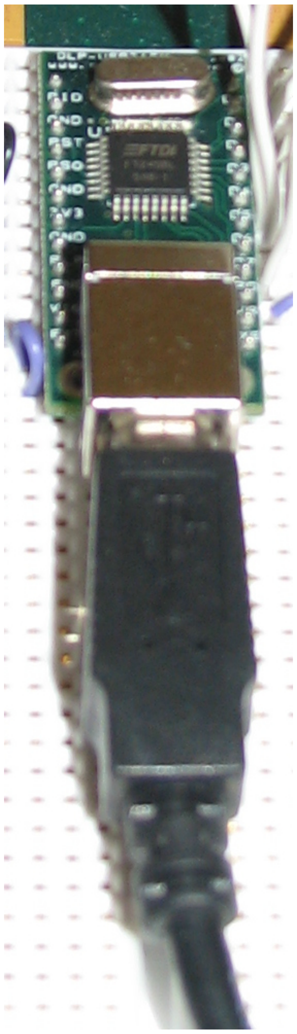


Figure 3-12: DLP Design USB Adapter

3.4.1 Hardware Interface

The DLP Design USB adapter was connected to the labkit mainly through the user pins. The user pins enabled us to have a very simple interface both in terms of inputs and outputs to the USB adapter. A sample connection to the labkit is shown in Figure 3-13.

One can see from the picture that there are a few pins on the left-hand side of the adapter that are wired together. These pins just determine the mode in which the chip operates via a set voltage level from pin 3. The connection used is the basic model presented in the DLP-USB245M datasheet. [2] There are also three pins on the left hand side that are grounded to the ground of the labkit. The USB adapter

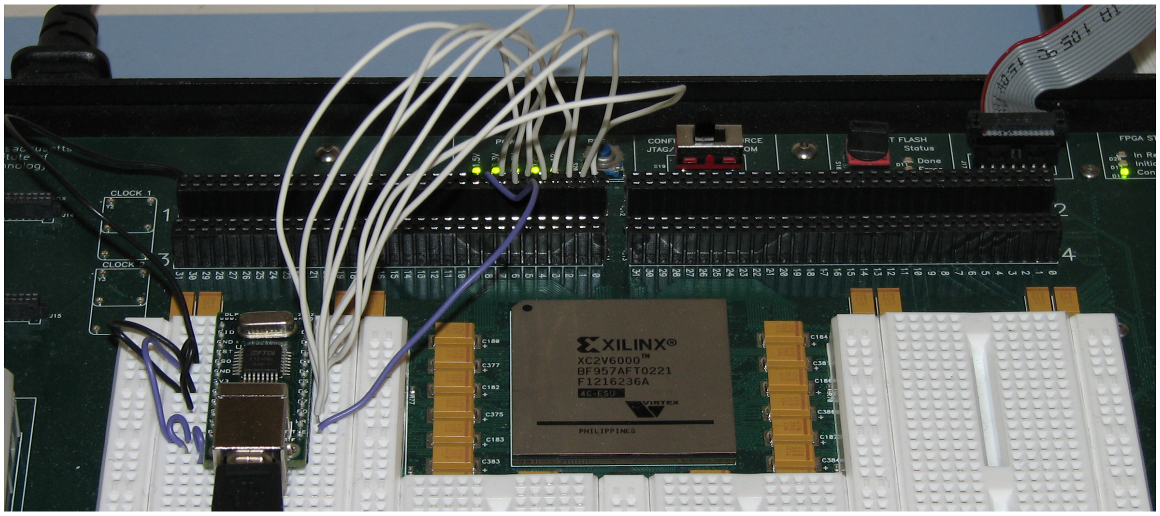


Figure 3-13: DLP Design USB Adapter Connection to Labkit

takes data from the computer and outputs it in a parallel interface, which provides understanding as to the presence of the eight data pins on the right hand side of the adapter. There are nine total wires leading from the right hand side of the adapter to the labkit - eight white and one blue. The blue wire leads to the read pin of the adapter, and the value on this pin is toggled to call on the next data set in the USB adapter FIFO.

The overall schematic of the connection of the USB adapter to the labkit is depicted in Figure 3-14.

3.4.2 Software Interface

A great advantage of using the DLP Design USB adapter was the software that was provided with the adapter. The software allowed the connection of the USB adapter to a USB cable and then to a computer. With the installation of some drivers available with DLP design, the system was ready for data transfer.

The files used in the project source file are shown in Figure 3-16.

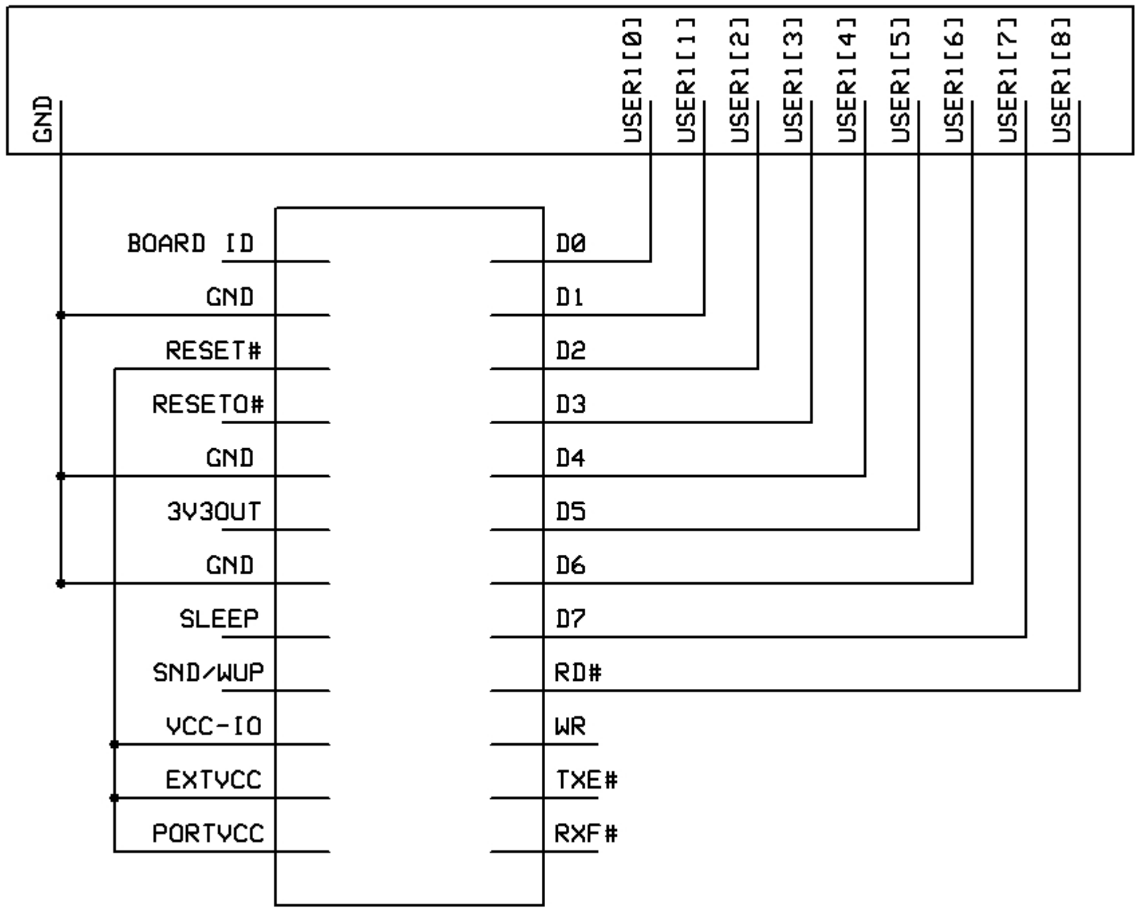


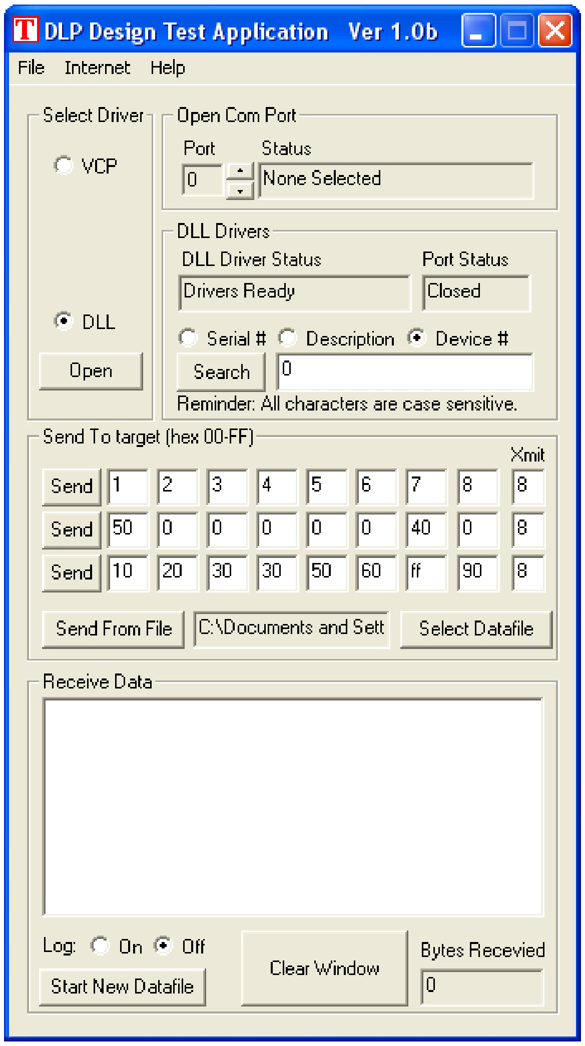
Figure 3-14: DLP Design USB Adapter Schematic

3.4.3 display_16hex.v

The display module was originally written by Nathan Ickes. This display module differs from other display modules because it can display more than just hex characters. Modifications were made to this display module to improve the robustness of the file by moving the disp_rs to a state earlier, as recommended by one of the semester TA's, Javier Castro.

3.4.4 flash_int.v

The flash_int module was also written by Nathan Ickes, and it conveniently performs all the hand-shaking necessary to interface with the Flash ROM.



Select Driver: ( ) VCP [✓ DLL]

Open Com Port — Port: 0 — Status: None Selected

DLL Drivers — DLL Driver Status: Drivers Ready — Port Status: Closed

( ) Serial # ( ) Description [✓ Device #] — 0

Reminder: All characters are case sensitive.

Send To target (hex 00-FF)

								Xmit
1	2	3	4	5	6	7	8	8
50	0	0	0	0	0	40	0	8
10	20	30	30	50	60	ff	90	8

C:\Documents and Sett

Receive Data

Log: ( ) On [✓ Off]

Bytes Recevied: 0

Figure 3-15: DLP Design USB Adapter Computer Interface

3.4.5 flashtest.v

Flashtest was the top level module. It basically connects the signals from the other modules used in the USB to Flash writer.

3.4.6 flashwriter.v

The flashwriter module is the top level module of the USB to Flash writer. The module contains the labkit I/O's. The main file also contains auxiliary modules, such as the debounce module and the audio playback (ac97) modules. In the flashwriter module, Kevin added button interfaces on the labkit that would read in data from

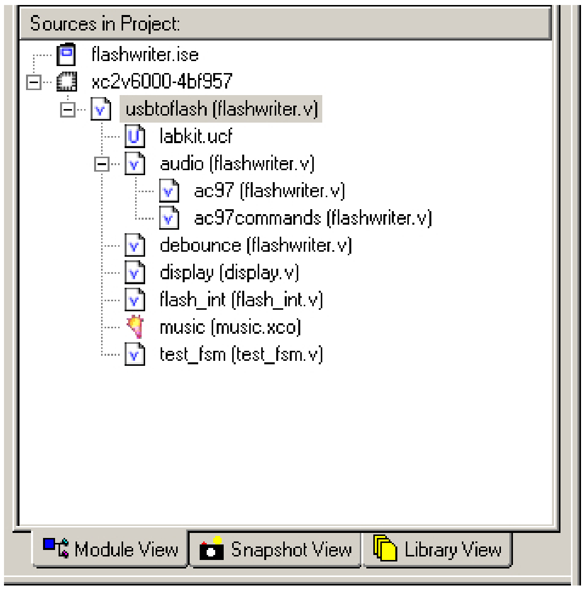


Figure 3-16: Sources in USB to Flash Storage Program

the USB adapter when button 0 was pressed. The data would then be stored on the Flash when button 1 were pressed. The process could be restarted by pressing button 2. Pressing the left button on the labkit enables an erase mode to help clean up errors.

3.4.7 test_fsm.v

The FSM for the Flash read, write, and erase functions was written by Nathan Ickes and revised by Kevin. The main changes in the module include an enable signal that does not let the FSM advance beyond Flash identification steps unless the user directs it to.

The addressing and state flow were also modified such that the FSM could be applied towards the USB to Flash storage methods desired. The addresses are dependent on the switches, and only half a block, or 32768 addresses are written at a time. Only a block is cleared at a time, and the block that is cleared is always the location to be written or one block ahead in the addressing such that previously written data at lower addresses is maintained. The roundabout nature of this code was due to the small memory buffer of the DLP Design USB adapter software and also the necessity of the Flash to erase an entire block at a time.

3.4.8 Block Diagram

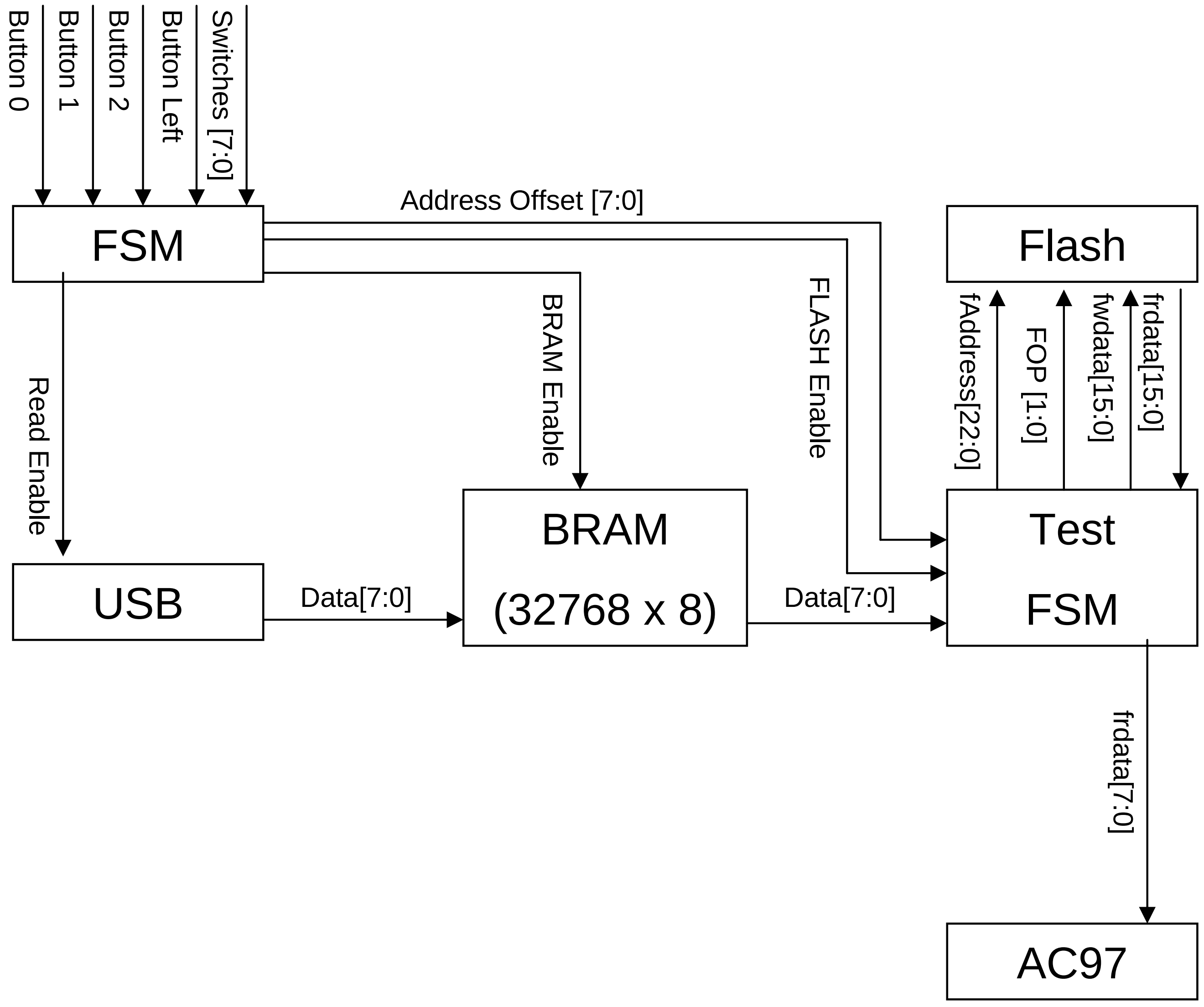


Figure 3-17: USB to Flash Diagram

3.5 System Integration

3.5.1 Interface - interface.v

The interface module provides a menu screen to the user, allowing them to play the song and choose the difficulty level. Since light conditions vary widely depending on the environment, the menu allows the user to enter a calibration mode in order to calibrate the different threshold values of the colors to be detected. There is also a high score mode which displays the current high score on the LED panel when selected.

Interface is responsible for switching between displaying the menu screen or displaying the stickman and calibration screens. It does this by running the menu and stickman pixels through a multiplexer whose output is connected to the video display module. By default, the menu pixels are displayed, but when the user selects to play the game or calibrate the color thresholds the interface module switches to displaying the stickman video data.

In order to display the calibration screens or the stickman screen, interface controls the state lines of the cm_calculator module. During calibration, interface handles incrementing through the different states and during game play it allows only the single stickman state to be output.

The user is able to use the interface module with the up, down, left, right, and enter buttons on the labkit. The up and down buttons move the cursor up and down to select which function to perform. The left and right buttons control the difficulty setting, which is displayed using one to three blocks on the bottom right of the screen, one being the easiest and three being the hardest. The different difficulty settings effect the difficulty of the choreography during game play. The interface module consists of two finite state machines. The first one keeps track of the menu cursor and selections made in the menu. The second controls the difficulty setting. The interface module also has two special states for playing the game and for calibration. Finally, interface also computes the high score and displays it on the LED bank when it is requested.

3.6 Graphical User Interface

3.7 Command Images

A series of IPOD people with Professor Terman's facial features were created in Adobe Photoshop. Each image was 128 pixels wide and 115 pixels high. Each image was divided into a separate left-hand side and right-hand side. The independent sides allowed us to create combinations of the two sides to create a large number of possible complete images. The images are shown in Figure 3-18.

The images were run through a script written in MATLAB. The script separates the left and right hand images and organizes them in two separate COE files. These files can then be loaded on to the labkit and with proper incrementing, all the various images can be accessed.

3.8 Stickman

The "Stickman" module is responsible for drawing a 2-d representation of the user onto the screen using the four center of mass positions given to it by the "cm_-calculator" module. To accomplish this, two sprites were created.

The first was a simple rectangle drawing sprite which took the x-y coordinates of the upper-left and bottom right corners to draw the rectangle. This was easily implemented using simple comparisons.

The more complicated sprite was the line drawing sprite. This module took the x-y positions of the line endpoints and calculated the line connecting them. The line had two parameters for 3'bit color and relative width. The following logic was used in drawing the line. In determining which x-y values are on a line with endpoints $x_1 - y_1$ and $x_2 - y_2$, the following relation is true:

$$\frac{y_1 - y_2}{x_1 - x_2} = \frac{y - y_1}{x - x_1} \tag{3.1}$$

that when rearranged becomes:

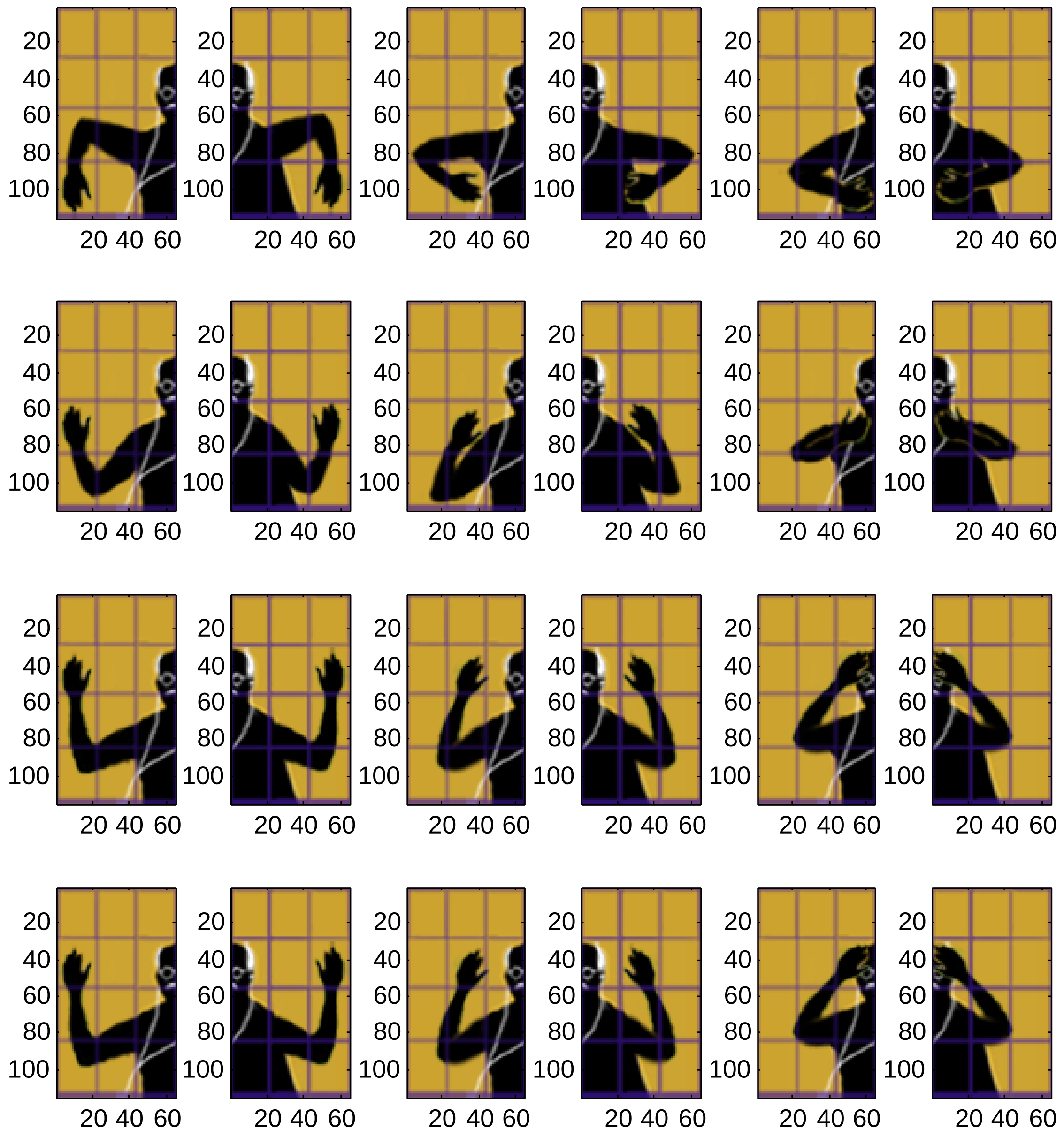


Figure 3-18: Sample Configurations of the IPOD People Abstraction of Professor Terman

$$(x-x_1)\times(y_1-y_2) = (y-y_1)\times(x_1-x_2) \rightarrow (x-x_1)\times(y_1-y_2)-(y-y_1)\times)x_1-x_2) = 0 \tag{3.2}$$

Substituting x and y for 'hcount' and 'vcount' respectively, the above expression can be used for calculating the sprite. However, this only evaluates to true when 'hcount' and 'vcount' are precisely on the line, and when the line is at an angle other than 45, only a few pixels match these conditions. Therefore, a tolerance is necessary in order to display a visible line.

It was noticed during development of the line sprite that when Expression 3.2 was changed into Expression 3.3:

$$(x - x_1) \times (y_1 - y_2) - (y - y_1) \times (x_1 - x_2) \leq tolerance \tag{3.3}$$

a shape in the form of Figure 3-19 was generated.

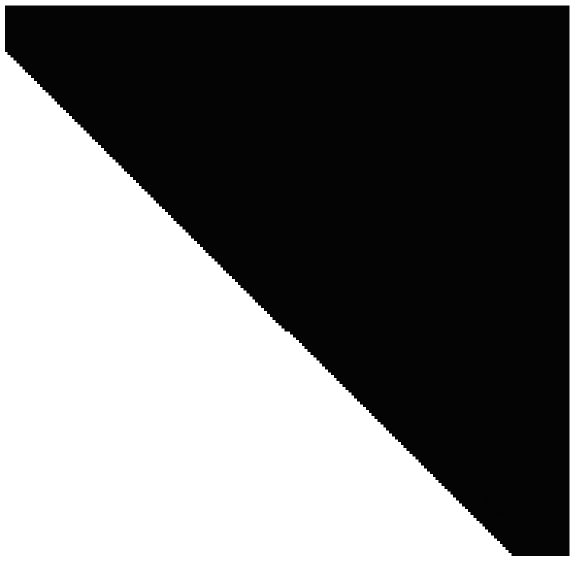


Figure 3-19: Line Sprite Shape

It was then realized that multiplying the left-hand side of expression 3.3 by -1 resulted in a region that was the mirror image of that in figure 3-19. By taking only the intersection of these two regions, a line of a width proportional to 'tolerance' could be created, as illustrated in figure 3-20.

At first this process was done using only unsigned numbers, so four comparisons had to be made to determine which endpoints were greater than the others so that the differences would all be positive. This process worked, but required too much computing time and later caused glitch problems when integrated with other video intensive modules. In order to solve this problem, a signed line module was created

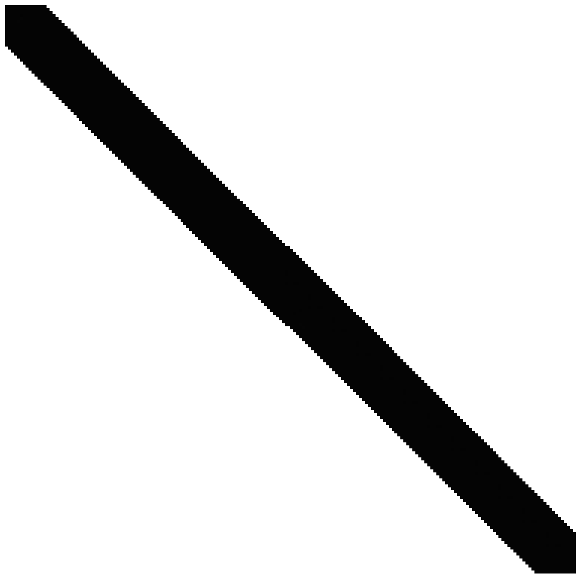


Figure 3-20: Line Sprite

which performed the exact same function with the exact same parameters, but had one equation which covered all conditions, thereby speeding up the sprite considerably.

The stickman module used these two sprites to draw a simple stickperson on the screen representing the user. It used the distance between the two shoulder locations calculated by cm_calculator to draw a scaled representation of a person using torso, neck, and head rectangle sprites and left and right arm line sprites. The line sprites connect the shoulders with the hand center of masses with a line of constant width. The torso, neck, and head rectangles are scaled according to how far apart the shoulder center of mass locations are. This is illustrated in Figure 3-21.

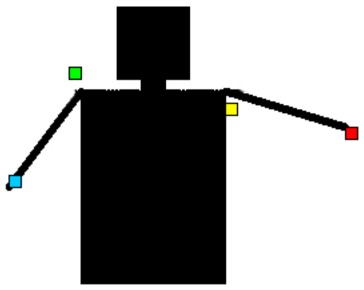


Figure 3-21: Stickman

In order to speed processing to allow all of the other video related modules time to complete, the stickman module which calculates the proportions of the stickperson is fully pipelined to allow for maximum efficiency.

Chapter 4

Testing and Debugging

4.1 Video

4.1.1 Video Input

Video testing was greatly simplified by code provided on the Fall 2005 webpage. There was existing code to take code from the NTSC camera, store it in the ZBT memory, and then to display it on the screen.

Significant improvements were necessary for the existing code. The existing code stored a single pixel for every four pixels. Because of the downsampling, the resolution of the screen was unsatisfactory in our opinion. Each pixel location referred to an independent pixel location in memory. Originally, four pixels were also stored per ZBT address. Such storage was possible because only 8-bits of image information, or the top 8-bits of the luminance, were kept. We stored 18 bits per pixel, or 6 bits each of red, green and blue. This also enabled us to more efficiently use the ZBT RAM, since each address that was accessed was fully-written.

4.1.2 Color Filtering

Originally, the color filtering was performed by simply requiring a single absolute threshold for the colors. For example, when we first attempted single color detection, if we wanted to filter out everything but red, we configured the display such that only pixels with red values greater than 250 were displayed. Such a method was clearly flawed, as any color with a high red value, including yellows and whites were displayed.

As a means of real-time color calibration, the switches on the labkit, as shown in Figure 3-7, were used to input values for the color thresholds.

4.2 Audio

The Audio component was built out of several independently testable modules. This allowed for easier debugging and testing through the development process. The first step was storing and playing back a song. Interfacing with the flash memory turned out to be quite challenging. The flash memory had not been used in previous projects so there we no example programs to reference. The only existing code was a flash memory testing program, written by Nathan Ickes. This program was modified to create a writing program and a reading program. Initially writing was accomplished by building a BRAM using a COE file and then copying the contents of the BRAM to the flash memory. The only way to verify that data was actually written was to read it back. This meant that the reading and writing could not be debugged independently. Displaying data on the hex display was not effective so the logic analyzer was hooked up. The problem then became verifying that the data coming out of the flash was actually the same as what went in. This was hard to do using actually song data. This problem was remedied by replacing the song data with a ramp. Initially between two and three samples would be missing between each sample output by the flash. This problem was ultimately solved by correcting timing issues. It was then determined while all the data was coming out it was coming out at half the expected clock rate. Which meant that audio was being played back at half speed. Unable to determine the cause test FSM was modified to increment address by 2 instead of 1 so that 24 kHz data would come out at the proper speed instead of 48 kHz data at half speed. It was realized on the night before check-off that this problem was caused by the fact that while reading is a one state process the state machine get run every other cycle with an idle period in-between. The next step was to store a full song on the flash memory. Building BRAMs from COE files would take days and was clearly not an elegant solution to the problem. The clear answer was to use a serial interface. A serial interface while require some coding and understanding of hyper terminal protocols would allow writing a full song in less than an hour. However before a serial interface was implemented Professor Terman suggest that we try out a new USB chip he had. We decided to take a risk and try to use USB. The details of the implementation and debugging of the USB interface are included in their own section and will be skipped here for brevity.

Now that song data was loaded on the flash and could be played back and output to the headphones it was time to start implementing a beat detection algorithm. The beat process module was coded up without a debouncer and high order bits of the instantaneous and baseline samples were displayed on the LEDs. This provided enough information to debug the mathematical errors in the algorithms implementation and begin basic fine tuning. Attention was then turned to actually outputting a beat. Initially the beat was displayed on an LED and the threshold was set using the switches. Several hours were wasted due to errors in shifting registers and failing to remember that a '1' turns the LED off. Once this was fixed it because clear that a well functioning beat detection algorithm would result is the LED turning on for such a short time it would not be visible. Initially this was fixed by inverting an LED every time a beat was detected. The beat was not yet debounced so this method only worked some of the time. Next a debouncer was implemented and the LED was kept high for the duration of the debouncer's delay.

With the beat detection algorithm working the next step was to generate a BRAM and store the beats. Two outputs are needed by the interface module, but only one input is needed. Initially there were problems because the core generator kept generating an asymmetrical BRAM when a symmetrical one was requested. This problem was eventually fixed by repeatedly trying to build a symmetrical BRAM and the root cause was never identified. Next the beat memory modules and clock 16th modules had to be written in order to store the beats. Both modules where easily implemented.

Performing beat detection at normal speed would take too long so it was necessary to speed up the process. Up to now things were clocked off of the AC97s ready signal. A sudo-ready new ready was created to be used in place of the AC97 ready during beat detection.

The last task was to play back beats synchronized with the audio. The storage tester module was created. Storage tester is just a simple finite state machine which successfully played back asynchronous audio and beats. The problem was that after beat detection the flash address was not resetting to the first address when told too. This was temporarily fixed by resetting the BRAM address every time the flash looped to the beginning of the song. However this resulted in it being unsynchronized initially and would not work for the actually game. It was eventually worked by luck when the high speed detection was run at approximately 3 MHz but not at any other speed so a timing issue was obviously the cause. It took many days and hours to finally realize that the problem was that the flash only responds every other cycle but commands to it were only being given for one cycle. Having completed all of the need functionality a version with two songs was tried but ultimately abandon due to difficulties storing and playing back beats for two different songs.

4.3 Integration

The integration of the video, audio, and control modules was by far the most frustrating part of the project. The integration took place in three phases. Phase one involved integrating the stickman module with the video and center of mass modules. Phase two involved basic integration and control of the audio and video components, and phase three involved integrating these components under the control of the user interface. The problems that arose during both of these phases will be discussed below.

4.3.1 Stickman and Video Integration

Integrating the stickman module with the video and center of mass modules proved difficult because of the large amount of processing that went on inside both of these modules. The center of mass module was trying to keep track of four different center of masses at once, while the stickman was computing complex line sprites. When first combined, the labkit was unable to handle the amount of computation necessary during each clock cycle, so noisy center of masses occurred in the calculate_cm module and the stickman module became very glitchy. In order to resolve this, both modules were pipelined and trimmed down to allow for processing time. This significantly decreased the processing time, and soon both modules were able to operate simultaneously with each other. Also, the calculate_cm module was not allowed to update the center of masses continuously, but only once every four or so frames. This decreased the noisy and jumpy effects we had, thereby making the game much more playable.

4.3.2 Audio and Video Integration

Integrating the audio and video components of the project together was also challenge. The audio component had a complex control interface, and controlling it took time learn properly. It also required a large degree of computation, particularly during the beat detection phase, and this further exacerbated the timing problems that were encountered during the first phase of integration. Further improvements were made to the video code in both the center of mass and stickman calculations. It was discovered during this phase that line drawing sprite used by the stickman module was very inefficient and that a new signed method of computing the line was needed. This was implemented to a great degree of sucess. Further improvements were made, and soon both modules were able to operate together with each other.

4.3.3 Audio, Video and User Interface Integration

The final phase of integration was controlling integrating the audio and video components with a user interface that would control the system and score the gameplay. Creation of the user interface itself was straightforward, and surprisingly enough, initial integration of the video and audio components into the interface was successful. However, trying to control the components using the interface proved to be more challenging, particularly in controlling beat detection and song playback. These questions were eventually answered after modification of the audio control code. However, adding the control logic began to effect the operation of the video component. These inexplicable effects posed serious challenges since no logical reason for their occurrence could be found. The problems were resolved by trying to simplify and and reduce the computation necessary in all modules.

One bonus goal the team had was to add the ability to play back more than one song. This proved problematic since the flash interface only updates itself once every couple of cycles. Delays had to be added in certain parts of the code in order to allow the flash to load the values being passed to it. However, after this worked, it was discovered that due to the architecture of the audio control, it would require substantive rewriting of the entire audio control code to allow this to happen. This bonus feature was therefore reluctantly dropped, and we stuck with playing back only one song.

The integration of the separate modules was a formidable task. Different coding styles and expectations made the process a challenging one, but by taking the time to understand what was happening beneath the surface of each of the modules and patiently incrementing the degree of integration, all the problems were addressed and glitch issues were overcome.

Chapter 5

Conclusions

The full motion dance machine was a great overview of many features of the labkit. The project integrated challenging and interesting video, audio, and memory access features.

We applied simple principles to accomplish difficult tasks, such as the color and beat detection. We explored areas that had not been examined, such as the use of the USB adapter to write large song files to the labkit Flash ROM. The more we learned about the labkit, the more opportunities we found with the various features offered by it. Even so, we were able to finish a very carefully designed project.

The project was robust enough such that the system could easily be adapted from one lab station to the next, or even one room to the next. The calibration features allowed the system to detect colors even in non-optimal conditions, such as under the fluorescent lights in the Electrical Engineering and Computer Science labs.

One of the greatest achievements of the project was that every single goal that was required or advanced was achieved. Many goals deemed as risky were also completed. These goals included the creation of a virtual stickman who emulates the user's motion on the screen and the USB to Flash storage modules.

There are many improvements that could be made to the project. The Flash ROM currently uses only half of its available memory, as only the lowest byte of each address is written. We could write to each address and use a more efficient means for processing the data such that the entire Flash memory could be utilized.

The video already has safeguards in it to protect from undesired operation when the swatches are obscured. Such safeguards include not accepting new center of mass data unless a certain number of pixels have been identified as the correct color on a screen. These protections were quickly developed and could surely be improved from its current condition.

The graphical user interface could also be improved, as more feedback could be provided to the user during the song, such as words on the screen indicating whether or not a dance move was accepted. Additionally, changes could be made to the scoring such that there were multiple scoring thresholds, so that if a user came close to the right move, they would still get points.

Additional user interfaces would have also been desirable. Storing more songs per labkit would make the game more enjoyable. Saving the high scores into the Flash ROM would have enabled a permanent record of high scores.

The final project was a very challenging and rewarding experience. It demonstrated the importance of abstractions when it came time to integrate the different modules of code. Timing was also a very significant issue, and it required considerations of the project and the processes involved from a basic logic component perspective.

Appendix A

Verilog

A.1 Full Motion Dance Machine

A.1.1 ac97.v

```
//WE DID NOT MODIFY THIS FILE
//used as provided for lab 4
// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

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

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

   reg ready;
   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

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

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

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

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

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

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

```
  ready <= 1'b0;

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

if ((bit_count >= 0) && (bit_count <= 8'd15))
  // Slot 0: Tags
  case (bit_count[3:0])
    4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
    4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
    4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
    4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
4'h4: ac97_sdata_out <= l_right_v; // Right data valid
    default: ac97_sdata_out <= 1'b0;
  endcase

else if ((bit_count >= 8'd16) && (bit_count <= 8'd35))
  // Slot 1: Command address (8-bits, left justified)
  ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

else if ((bit_count >= 8'd36) && (bit_count <= 8'd55))
  // Slot 2: Command data (16-bits, left justified)
  ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

else if ((bit_count >= 8'd56) && (bit_count <= 8'd75))
  begin
     // Slot 3: Left channel
     ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
     l_left_data <= { l_left_data[18:0], l_left_data[19] };
  end
else if ((bit_count >= 8'd76) && (bit_count <= 8'd95))
  // Slot 4: Right channel
     ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
else
  ac97_sdata_out <= 1'b0;
```

```
      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

   always @(negedge ac97_bit_clock) begin
      if ((bit_count >= 8'd57) && (bit_count <= 8'd76))
        // Slot 3: Left channel
        left_in_data <= { left_in_data[18:0], ac97_sdata_in };
      else if ((bit_count >= 8'd77) && (bit_count <= 8'd96))
        // Slot 4: Right channel
        right_in_data <= { right_in_data[18:0], ac97_sdata_in };
   end

endmodule
```

A.1.2 addypicker.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 module - addypicker
//
//
// File:   addypicker.v
// Date:   13-Dec-06
//
// Created: December 13, 2006
// Author: Jonathan Burnham and Kevin Miu
//
// This module determines the choreography of the song and outputs a location
// based command immage from the memory
//
// Inputs:
//
//   clk - clock signal
//   counter - upcoming command beat signal
//   [10:0]hcount - 11-bit horizontal count
//   [9:0]vcount - 10-bit verticla count
//   [10:0]xoffset - picture offset value which effectly moves picture
//   [1:0] difficulty - difficulty level chosen by user to pick choreography
//
// Outputs:
//
//   [17:0] addr - address to access ROMs
//   [7:0] piccount - image index of left image
//   [7:0] secondcount - image index of right image
//
////////////////////////////////////////////////////////////////////////////////

module addypicker(clk,counter,hcount,vcount,addr,xoffset, piccount,
secondcount, difficulty);
    input clk;
    input counter;
    input [10:0]hcount;
    input [9:0]vcount;
    input [10:0]xoffset;
    input [1:0] difficulty;

    output [17:0]addr;
    output [7:0] piccount;
    output [7:0] secondcount;

    reg [17:0]addr1;
    reg [17:0]addr2;
    reg [17:0] addr;
    reg oldcount;
    reg [7:0]piccount = 6;
    reg [7:0]secondcount = 8;


// Choreographs the moves and determines which command images to issue
    always@(posedge clk)
    begin
        oldcount <= counter;
        if(counter && ~oldcount) begin
// Finds difficulty and determines choreography
            case({difficulty,piccount,secondcount})
                // Beginner
                {2'd0,8'd6,8'd8} : begin
                                        piccount <= 8;
                                        secondcount <= 6;
                                    end
                {2'd0,8'd8,8'd6} : begin
                                        piccount <= 6;
                                        secondcount <= 8;
                                    end

                // Intermediate
                {2'd1,8'd6,8'd8} : begin
                                        piccount <= 8;
```

```
                                           secondcount <= 6;
                                        end
                {2'd1,8'd8,8'd6} : begin
                                           piccount <= 1;
                                           secondcount <= 1;
                                        end
                {2'd1,8'd1,8'd1} : begin
                                           piccount <= 6;
                                           secondcount <= 8;
                                        end

                // Advanced
                {2'd2,8'd6,8'd8} : begin
                                           piccount <= 3;
                                           secondcount <= 3;
                                        end
                {2'd2,8'd3,8'd3} : begin
                                           piccount <= 10;
                                           secondcount <= 1;
                                        end
                {2'd2,8'd10,8'd1} : begin
                                           piccount <= 1;
                                           secondcount <= 10;
                                        end
                {2'd2,8'd1,8'd10} : begin
                                           piccount <= 8;
                                           secondcount <= 6;
                                        end
                {2'd2,8'd8,8'd6} : begin
                                           piccount <= 6;
                                           secondcount <= 8;
                                        end
            endcase
        end
// If the command issue is within the desired bounds
// Load a buffer with a partially calculated address
// and determine the true address on the next cycle
        if(vcount<231)
        begin
            if(hcount<256+xoffset&&hcount>xoffset)
            begin
                if(hcount<129+xoffset)
                begin
                    addr1<=(secondcount*115+vcount[9:1]);
                    addr<=addr1*64+(hcount[9:1]-xoffset[9:1]);
                end
                else
                begin
                    addr2<=(piccount*115+vcount[9:1]);
                    addr<=addr2*64+(hcount[9:1]-64-xoffset[9:1]);
                end
            end
        end
    end
endmodule
```

A.1.3 arrow.v

```
//////////////////////////////////////////////////////
//arrow
//Written by Jonathan Burnham
//
//Arrow uses two line sprites to draw a cursor whose
//head is at the x,y position passed to it as an input
//////////////////////////////////////////////////////

module arrow(clock, hcount, vcount, x, y, pixels);
    input clock;        // 65 MHZ
    input [10:0] hcount;    // Current x position of the video
    input [10:0] x;     // x position of the head of the cursor
    input [9:0] vcount; // Current y position of the video
    input [9:0] y;      // y position of the head of the cursor
    output [23:0] pixels;   // sprite pixels for the arrow

    reg [10:0] linex;       // Ending x positions of the arrow
    reg [9:0] line1y, line2y;   // Heights of the arrows
    reg [23:0] pixels = 0;      // Sprite pixels from the arrow
    wire [2:0] pixel1,pixel2;   // pixels from the two line sprites

    // Create two lines and set their color to green and constant width of 4
    line line1(clock,linex,line1y,x,y,hcount,vcount,pixel1);
    line line2(clock,linex,line2y,x,y,hcount,vcount,pixel2);
    defparam line1.WIDTH = 5'd4;
    defparam line2.WIDTH = 5'd4;
    defparam line1.COLOR = 3'b001;
    defparam line2.COLOR = 3'b001;

    // Set end points for the lines and or their pixels together
    always @ (posedge clock) begin
        linex <= x - 25;
        line1y <= y - 25;
        line2y <= y + 25;

        pixels <= {8{pixel1 | pixel2}};
    end
endmodule
```

A.1.4 audio.v

```
//WE DID NOT MODIFY THIS FILE
//this is the file provide for lab 4

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

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

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

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

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

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

   reg audio_reset_b;
   reg [9:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 10'd1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end

   wire ac97_ready;
   ac97 ac97(ac97_ready, command_address, command_data, command_valid,
             left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
             right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock);

   // generate two pulses synchronous with the clock: first capture, then ready
   reg [2:0] ready_sync;
   always @ (posedge clock_27mhz) begin
     ready_sync <= {ready_sync[1:0], ac97_ready};
   end
   assign ready = ready_sync[1] & ~ready_sync[2];

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

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

A.1.5 beat_detect.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 module - beat_detect
//
//
// File:   beat_detect.v
// Date:   13-Dec-06
//
// Created: December 13, 2006
// Author: Kevin Miu
//
// This module takes in the x and y positions of the red and teal swatches
// and from velocity information determines if the user has moved properly
// to satisfy the condition of dancing to a beat.
//
// Inputs:
//   clk - 65 mHz clock
//   [10:0]cmx0,cmx3 - x component center of mass for red and teal, respectively
//   [9:0] cmy0,cmy3 - y component center of mass for red and teal, respectively
//   [1:0] colorcount - current color being analyzed
//   beat_clear - signal that clears the beat enable
// Outputs:
//   beat_enable_red - signal to allow scoring for right hand
//   beat_enable_teal - signal to allow scoring for left hand

module
beat_detect(clk,cmx0,cmy0,cmx3,cmy3,colorcount,beat_clear,beat_enable_red,beat_enable
input clk; input [10:0]cmx0,cmx3; input [9:0]cmy0,cmy3; input [1:0]
colorcount; input beat_clear; output
beat_enable_red,beat_enable_teal;

reg old1_cmx0,old1_cmx3,old2_cmx0,old2_cmx3; reg
old1_cmy0,old1_cmy3,old2_cmy0,old2_cmy3; reg vx0,vx3; reg vy0,vy3;
reg beat_enable_red,beat_enable_teal; reg [4:0]
enable_count_red,enable_count_teal;

always@(posedge clk) begin
    if(beat_clear)
    begin
        beat_enable_red<=0;
        beat_enable_teal<=0;
    end
        case(colorcount)
```

```
                    2'b00: begin //Red count
                        //Save old 6th bit of positions for fair velocity approximation
                        old2_cmx0<=old1_cmx0;
                        old2_cmy0<=old1_cmy0;
                        old1_cmx0<=cmx[0][5];
                        old1_cmy0<=cmy[0][5];
                        old2_cmx3<=old1_cmx3;
                        old2_cmy3<=old1_cmy3;
                        old1_cmx3<=cmx[3][5];
                        old1_cmy3<=cmy[3][5];
                         end
                2'b01: begin
                        //See if the velocity differs from the previous one
                        vx0<=old1_cmx0-old2_cmx0;
                        vy0<=old1_cmy0-old2_cmy0;
                        vx3<=old1_cmx3-old2_cmx3;
                        vy3<=old1_cmy3-old2_cmy3;
                     end
                2'b10: begin
                        //Add any difference in the velocity to the enable counters
                        enable_count_red<=enable_count_red+vx0+vy0;
                        enable_count_teal<=enable_count_teal+vx3+vy3;
                        end
                2'b11: begin
                        //If the 4th bit of the enables go high, allow beat scoring
                        if(enable_count[4]) begin
                            enable_count_red<=0;
                            beat_enable_red<=1;
                            end
                        if(enable_count2[4]) begin
                            enable_count_teal<=0;
                            beat_enable_teal<=1;
                            end

end endmodule
```

A.1.6 beat_memory.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
```

```
// Engineer:Christopher Hoffman
//
// Create Date:    14:40:47 12/01/06
// Design Name:
// Module Name:    beat_memory
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
//this modules takes beats from beat_process and stores them in memory. IT also gener

module beat_memory(clock_27mhz, enable_16, ready, start, beat_in,
beat_out, address, we, stop, enable, offset
                        ,flash_address, last_flash_address);
        input clock_27mhz, enable_16, ready, start;
        input beat_in;
        output beat_out, we;
        output [11:0] address; //to bram input A
        output stop, enable; //stop indicates beat detection is done //enable turns o
        input [3:0] offset; //used to syncronize beat and audio playback.
        input [22:0] flash_address, last_flash_address;

        reg beat_out;
        reg we;
        reg [11:0] temp_address = 0;
        reg temp_beat;
        reg enable = 0;
        reg done = 0;
        reg stop = 0;

        assign address = temp_address + offset;//shifts address to allow syncronizati


        //this always block controls beat detection execution
        // it runs off of the 27 MHz clock to ease inntegration with the state machin
        //the enable signal is controlled by this block and is used to turn on the re
        //needed for high speed beat detection
        //the block also provides a stop pulse to the state machine when beat detecti
        always @ (posedge clock_27mhz) begin
            if (start) begin //fsm says to start beat detection
            enable <= 1;//enable other audio modules to operate in beat detection mod
            end
            else if (done) begin
                stop <= 1;//tell fsm that beat detection is done
                enable <= 0;//return all modules to normal playback mode
            end
            else stop <= 0;
        end

        //note: done is an internal finished signal and stop is the external signal
        always @ (posedge ready) begin
            if (enable) begin
                 if (flash_address >= last_flash_address) begin//set done high
                  done <= 1;
                  temp_address <= 0;
                  we <= 0;
                  end
                 else if (enable_16 && done) done <= 0;//unset done
                 else if (enable_16) begin //write a beat to memory
                    beat_out <= temp_beat;
                    temp_beat<= beat_in;
                    temp_address <= temp_address + 1;
                    we <= 1;
                end
                 else if (!temp_beat) begin//OR together incoming beats
                 temp_beat <= beat_in;
                 we <= 0;
                 end
                 else we <= 0;
            end
            else we <= 0;
        end


endmodule
```

A.1.7 beat_process.v

```
‘timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
```

```
// Engineer: Christopher Hoffman
//
// Create Date:    14:28:43 11/19/06
// Design Name:
// Module Name:    beat_process
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module beat_process(ready, data_in, beat, led, treshold, beat_1cyc,
clock_27mhz);
    input ready, clock_27mhz;
    input signed[7:0] data_in;
    input [7:0] treshold;
    output beat;
    output [5:0] led;
    output beat_1cyc;


    reg old_Ready;
    reg beat;
    reg beat_1cyc;
    reg old_beat;
    reg temp_beat;
    reg[8:0] beat_delay;
    reg[2:0] down_sample = 0;
    reg[4:0] data_index = 0;
    reg[17:0]  temp;
    //instantaneous samples
    reg[10:0]     d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15,d16,
                    d17,d18,d19,d20,d21,d22,d23,d24,d25,d26,d27,d28,d29,d30,d31,d32,
                    d33,d34,d35,d36,d37,d38,d39,d40,d41,d42,d43,d44,d45,d46,d47,d48,d
    reg[15:0]    current;
    reg[31:0] sum; //total of all 64 instantaneous sample
    reg[10:0] base; //sum*threshold/64

        assign led = {d1[8:6], base[8:6]};//diagnostic output of raw data
```

```
//downsamples audio and adds up 32 samples to form instantaneous average
always @ (posedge clock_27mhz) begin
    old_ready <= ready;
    if(!old_ready && ready)begin
    if (down_sample == 0) begin //downsample
    down_sample <= down_sample + 1;
    data_index <= data_index + 1;
    current <= (data_in * data_in); //square values to find power
    if(data_index == 0) temp <= current;
    else temp <= temp + current;
    end
    else down_sample <= down_sample + 1;
    end
    end

    //calculates baseline and determines beat
    always @ (posedge clock_27mhz) begin
    if(!old_ready && ready)begin
    if (data_index == 0) begin //shift all the data registers
    d64 <= d63;
    d63 <= d62;
    d62 <= d61;
    d61 <= d60;
    d60 <= d59;
    d59 <= d58;
    d58 <= d57;
    d57 <= d56;
    d56 <= d55;
    d55 <= d54;
    d54 <= d53;
    d53 <= d52;
    d52 <= d51;
    d51 <= d50;
    d50 <= d49;
    d49 <= d48;
    d48 <= d47;
    d47 <= d46;
    d46 <= d45;
    d45 <= d44;
    d44 <= d43;
    d43 <= d42;
    d42 <= d41;
    d41 <= d40;
    d40 <= d39;
```

```
d39 <= d38;
d38 <= d37;
d37 <= d36;
d36 <= d35;
d35 <= d34;
d34 <= d33;
d33 <= d32;
d32 <= d31;
d31 <= d30;
d30 <= d29;
d29 <= d28;
d28 <= d27;
d27 <= d26;
d26 <= d25;
d25 <= d24;
d24 <= d23;
d23 <= d22;
d22 <= d21;
d21 <= d20;
d20 <= d19;
d19 <= d18;
d18 <= d17;
d17 <= d16;
d16 <= d15;
d15 <= d14;
d14 <= d13;
d13 <= d12;
d12 <= d11;
d11 <= d10;
d10 <= d9;
d9 <= d8;
d8 <= d7;
d7 <= d6;
d6 <= d5;
d5 <= d4;
d4 <= d3;
d3 <= d2;
d2 <= d1;
d1 <= (temp >> 8);//right shift before storing to reduce register size
end
else if (data_index == 1) begin //add up all 64 instantaneous samples
    sum <= d1 + d2 + d3 + d4 + d5 + d6 + d7+d8+d9+d10+d11+d12+d13+d14+d15+d16

end
//devide by 64 to average and multiple by threshold
```

```
        //threshold of 36 converts to an actual value of 4.5
        //becuase it is actually 36 / 8
        else if (data_index == 2) begin
            base <= ((treshold*sum)/512); // 36 looks like a good number for treshold
        end
        //compare instantaeous to baseline
        else if (data_index == 3) begin
            old_beat <= beat;
            if (d1 > base) temp_beat <= 1;
            else temp_beat <= 0;
        end
        //debounce
        else if (data_index == 4) begin
            if(!old_beat && temp_beat) begin
             beat <= 1;
             beat_1cyc <= 1;
             beat_delay <= 0;
             end
            else if (old_beat && (beat_delay < 511)) begin
            beat_delay <= beat_delay +1;
            beat_1cyc <= 0;
            end
            else if (beat_delay >= 511) beat <= 0;
        end
        end
        end




endmodule
```

A.1.8 begin_scrolling.v

```
//////////////////////////////////////////////////////////////
//begin_scrolling
//Written by Jonathan Burnham
//
//Begin_scrolling computes the offset the dance pictures need
//to be starting as soon as the start input goes high.
//////////////////////////////////////////////////////////////
```

```
module begin_scrolling(clock,start,speed,xoffset,finished);
    input clock;        // 65 MHZ
    input start;        // Starts computation when it goes high
    input [18:0] speed; // Speed at which to scroll the picture
    output [10:0] xoffset;  // Calculated offset from right edge of screen
    output finished;    // Goes high when the last scroll completed

    reg finished = 0;       // Contains finished
    reg [10:0] xoffset = 11'd767;   // Starting offset, puts at right edge of screen
    reg [18:0] movement_counter = 0; // Uses speed to gradually move the offset


    // Moves the offset one pixel everytime the counter has incremented to speed
    // Computes when start == 1 or it hasn't reached the end of the screen
    always@(posedge clock) begin
        if (finished == 0 || (finished == 1 && start == 1)) begin
            finished <= 0;
            if (movement_counter == speed) begin
                xoffset <= xoffset - 1;
                movement_counter <= 0;
                if (xoffset == 0) begin
                    finished <= 1;
                    xoffset <= 11'd767;
                    movement_counter <= 0;
                end
            end
            else movement_counter <= movement_counter + 1;
        end
    end
endmodule
```

A.1.9 clock_16th.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Christopher Hoffman
//
// Create Date:    12:12:53 12/02/06
// Design Name:
// Module Name:    clock_16th
// Project Name:
```

```
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
//this module generates a 16 Hz clock
//needed becuase beats are stored in BRAM at 16Hz
//generates 16 Hz clock based on ready signal,
// which allows for easy creation of sudo-16Hz clock for use in high speed beat stori
module clock_16th(ready, enable_16);
     input ready;
     output enable_16;
     parameter delay = 3000; //3000 ready clicks is 1/16 of second   48000/16=3000
     reg[11:0] counter = 0;
     assign enable_16 = !counter;
     always @ (posedge ready) begin
        if(counter >= delay) counter = 0;
        else counter = counter + 1;
     end
endmodule
```

A.1.10 cm_calculator.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 module - cm_calculator
//
//
// File:   cm_calculator.v
// Date:   13-Dec-06
//
// Created: December 13, 2006
// Author: Kevin Miu
//
// This module takes in the video feed, filters out the colors (with or without
// calibration), and determines the center of mass.  The module handles all the image
// processing so that the video signals are not confused elsewhere.
```

```
//
// Inputs:
//   clk - 65 mHz clock
//   [17:0] pixel - input pixels from the camera, 6-bits per color channel
//   [10:0] hcount - 11-bit horizontal count
//   [9:0] vcount - 10-bit vertical count
//   vsync - vertical sync
//   [10:0] quotx - 11-bit x input from divider
//   [9:0] quoty - 10-bit y input from divider
//   [8:0] data - left command image data
//   [8:0] data2 - right command image data
//   new_beat - signal indicating old beat has expired
//   s0,s1,s2,s3,s4,s5,s6,s7,b0,b1,b2,b3 - switch and button inputs
//   [2:0] state - state of calibration FSM
//   sixteenth_count - beat count
//   [10:0] xoffset - command image offset
//
// Outputs:
//   [17:0] cleaned_pixel - processed pixels (filtered, command images added)
//   [28:0] x_sum - horizontal center of mass predivided sum
//   [27:0] y_sum - vertical center of mass predivided sum
//   [18:0] pixel_count - number of pixels counted towards center of mass calculation
//   [1:0] colorcount - color currently being observed (rotates through 4 colors)
//   [10:0] cmx0 - red horizontal center of mass
//   [9:0] cmy0 - red vertical center of mass
//   [10:0] cmx1 - green horizontal center of mass
//   [9:0] cmy1 - green vertical center of mass
//   [10:0] cmx2 - yellow horizontal center of mass
//   [9:0] cmy2 - yellow vertical center of mass
//   [10:0] cmx3 - teal horizontal center of mass
//   [9:0] cmy3 - teal vertical center of mass
//   [10:0] gridcenterx_pos - average x coordinate of shoulders
//   [8:0] hspacing - horizontal spacing between gridlines
//
////////////////////////////////////////////////////////////////////////////////

module
cm_calculator(clk,pixel,hcount,vcount,vsync,cleaned_pixel,x_sum,y_sum,pixel_count,quo
    input clk;
    input [17:0] pixel;
    input [10:0] hcount;
    input [9:0] vcount;
    input vsync;
    input [10:0] quotx;
    input [9:0] quoty;
    input [8:0] data;
    input [8:0] data2;
    input new_beat;
    input s0,s1,s2,s3,s4,s5,s6,s7,b0,b1,b2,b3;
    input [2:0] state;
    input sixteenth_count;
    input [10:0]xoffset;
    input good;

    output [17:0] cleaned_pixel;
    output [28:0] x_sum;
    output [27:0] y_sum;
    output [18:0] pixel_count;
    output [1:0] colorcount;
    output [10:0] cmx0;
    output [9:0] cmy0;
    output [10:0] cmx1;
    output [9:0] cmy1;
    output [10:0] cmx2;
    output [9:0] cmy2;
    output [10:0] cmx3;
    output [9:0] cmy3;
    output [10:0]gridcenterx_pos;
    output[8:0] hspacing;

    reg [8:0] cal1[3:0];
    reg [8:0] cal2[3:0];
    reg [17:0] cleaned_pixel;
    reg [28:0] x_sum;
    reg [27:0] y_sum;
    reg [18:0] pixel_count;
    reg [1:0] colorcount;
    reg [10:0] cmx[3:0];
    reg [9:0] cmy[3:0];
    reg [8:0]gridcenterx;
    reg [8:0]tempgridcenterx;
    reg [3:0] value_1back;
    reg stickman_enable;
    reg [8:0] hspacing;
    reg value_enable;

    wire phsync,pvsync,pblank,reset;
    wire [2:0] pixel_man;

    assign gridcenterx_pos={gridcenterx[8:0],2'b0};
```

```
 assign cmx0=cmx[0];
 assign cmy0=cmy[0];
 assign cmx1=cmx[1];
 assign cmy1=cmy[1];
 assign cmx2=cmx[2];
 assign cmy2=cmy[2];
 assign cmx3=cmx[3];
 assign cmy3=cmy[3];

 stickman man(clk,reset,{cmx[3][10:2],2'b0},{cmy[3][9:2],2'b0},{cmx[1][10:2],2'b0}

always@(posedge clk) begin
     //At each line end, clear the window filter data since only horizontal relati
     if(hcount==1024) value_1back<=0;
     //If a new beat comes in, clear the previous enables
     if(new_beat) begin
         beat_enable1<=0;
         beat_enable2<=0;
         enable_count<=0;
         enable_count2<=0;
     end
     //If signal is high then the user wants default calibration
     if (b3) begin
         cal1[0] <= {8'b01100000};
         cal2[0] <= {8'b00000000};
         cal1[1] <= {8'b00010000};
         cal2[1] <= {8'b00010000};
         cal1[2] <= {8'b00110000};
         cal2[2] <= {8'b01100000};
         cal1[3] <= {8'b00110000};
         cal2[3] <= {8'b00100000};
     end
     //At the end of the frame
    if(hcount==1024&&vcount==768)
     begin
     //Calculate a rough spacing for the gridlines
         hspacing<=cmx[2][10:2]-cmx[1][10:2];
     //Calculate the temporary center of the shoulders
         tempgridcenterx<={1'b0,cmx[1][10:3]}+{1'b0,cmx[2][10:3]};
     //If the temporary center is within a certain range, accept it,
     // otherwise reject it
         gridcenterx<=(tempgridcenterx[8:6]>0)?tempgridcenterx:gridcenterx;
     //Rotate the color observation if in specific states, such as stickman displa
         colorcount<=(state>3)?colorcount+1:state;
     //Clear the x and y counters
```

```
x_sum<=28'b0;
y_sum<=27'b0;
case(colorcount)
    2'b00: begin //Red count
        //Save old 6th bit of positions for fair velocity approximation
        xold2<=xold;
        yold2<=yold;
        xold<=cmx[0][5];
        yold<=cmy[0][5];
        xoldteal2<=xoldteal;
        yoldteal2<=yoldteal;
        xoldteal<=cmx[3][5];
        yoldteal<=cmy[3][5];
        //Enable the green change next clock cycle if over 63 pixels are
        value_enable<=(pixel_count>6'd63)?1:0;
        //Change the red value if the red enable condition is satisfied a
        //specified boundaries
        cmx[colorcount]<=(quotx>9'd309&&quotx<10'd1023&&value_enable)?quo
        cmy[colorcount]<=(quoty>8'd188&&quoty<10'd658&&value_enable)?quot
         end
2'b01: begin
        //See if the velocity differs from the previous one
        xvel<=xold-xold2;
        yvel<=yold-yold2;
        xvelteal<=xoldteal-xoldteal2;
        yvelteal<=yoldteal-yoldteal2;
        //Enable the yellow change next clock cycle if over 255 pixels ar
        value_enable<=(pixel_count>8'd255)?1:0;
        //Change the green value if the green enable condition is satisfi
        //specified boundaries
        cmx[colorcount]<=(quotx>9'd470&&quotx<10'd850&&value_enable)?quot
        cmy[colorcount]<=(quoty>8'd210&&quoty<10'd575&&value_enable)?quot
     end
2'b10: begin
        //Add any difference in the velocity to the enable counters
        enable_count<=enable_count+xvel+yvel;
        enable_count2<=enable_count2+xvelteal+yvelteal;
        //Enable the teal change next clock cycle if over 7 pixels are co
        value_enable<=(pixel_count>3'd7)?1:0;
        //Change the yellow value if the yellow enable condition is satis
        //specified boundaries
        cmx[colorcount]<=(quotx>9'd470&&quotx<10'd850&&value_enable)?quot
        cmy[colorcount]<=(quoty>8'd210&&quoty<10'd575&&value_enable)?quot
         end
2'b11: begin
```

```
                //If the 4th bit of the enables go high, allow beat scoring
                if(enable_count[4]) begin
                    enable_count<=0;
                    beat_enable1<=1;
                    end
                if(enable_count2[4]) begin
                    enable_count2<=0;
                    beat_enable2<=1;
                    end
                //Enable the red change next clock cycle if over 7 pixels are cou
                value_enable<=(pixel_count>3'd7)?1:0;
                //Change the teal value if the teal enable condition is satisfied
                //specified boundaries
                cmx[colorcount]<=(quotx>9'd309&&quotx<10'd1023&&value_enable)?quo
                cmy[colorcount]<=(quoty>8'd188&&quoty<10'd658&&value_enable)?quot
                 end
        endcase
        //reset the pixel counts
        pixel_count<=18'b0;
        //if the user presses b1 assign new calibration values
        if(b1==1) begin
            cal1[state]<={1'b0,s7,s6,s5,s4,4'b0};
            cal2[state]<={1'b0,s3,s2,s1,s0,4'b0};
        end
   end
    //if the image is in the expected video window
    if(hcount>9'd309&&hcount<10'd1019&&vcount<10'd730&&vcount>9'd260)
        begin
        case(state)
        // states 0-3 display individual colors for calibration, state 4 shows st
        // 5 is the base camera image
        3'b000:
            begin
                //if the red threshold is satisfied, do red counting/filtering
                if({1'b0,pixel[17:12]}>{1'b0,pixel[11:6]}+{1'b0,s7,s6,s5,s4,2'b0}
                    begin
                        cleaned_pixel<=pixel;
                        x_sum<=x_sum+hcount;
                        y_sum<=y_sum+vcount;
                        pixel_count<=pixel_count+1;
                    end
                else
                    cleaned_pixel<=0;
                end
        3'b001:
```

```
        begin
            //if the green threshold is satisfied, do green counting
            if({1'b0,pixel[11:6]}>{1'b0,pixel[17:12]}+{1'b0,s7,s6,s5,s4,2'b0}
                begin
                    cleaned_pixel<=pixel;
                    x_sum<=x_sum+hcount;
                    y_sum<=y_sum+vcount;
                    pixel_count<=pixel_count+1;
                end
            else
                cleaned_pixel<=0;
            end
    3'b010:
        begin
            //if the yellow threshold is satisfied, do yellow counting
            if({1'b0,pixel[17:12]}>{1'b0,pixel[5:0]}+{1'b0,s7,s6,s5,s4,2'b0}&
                begin
                    cleaned_pixel<=pixel;
                    x_sum<=x_sum+hcount;
                    y_sum<=y_sum+vcount;
                    pixel_count<=pixel_count+1;
                end
            else
                cleaned_pixel<=0;
            end

    3'b011:
        begin
            //if the teal threshold is satisfied, do teal counting
            if({1'b0,pixel[11:6]}>{1'b0,pixel[17:12]}+{1'b0,s7,s6,s5,s4,2'b0}
                begin
                    cleaned_pixel<=pixel;
                    x_sum<=x_sum+hcount;
                    y_sum<=y_sum+vcount;
                    pixel_count<=pixel_count+1;
                end
            else
                cleaned_pixel<=0;
            end
    3'b100:
        //Performs window filter on data so only counts points if 5 consecuti
        // correct threshold which helps filter out noise
        begin
        case(colorcount)
        2'b00:
```

```
if({1'b0,pixel[17:12]}>{1'b0,pixel[11:6]}+cal1[0][8:2]&&{1'b0,pixel[1
        begin
        case(value_1back[2])
        1'b1:
            begin
                x_sum<=x_sum+hcount-2;
                y_sum<=y_sum+vcount;
                pixel_count<=pixel_count+1;
            end
        1'b0:
        value_1back<=value_1back+1;
        endcase
        end
        else
        value_1back<=0;
2'b01:
if({1'b0,pixel[11:6]}>{1'b0,pixel[17:12]}+cal1[1][8:2]&&{1'b0,pixel[1
        begin
        case(value_1back[2])
        1'b1:
            begin
                x_sum<=x_sum+hcount-2;
                y_sum<=y_sum+vcount;
                pixel_count<=pixel_count+1;
            end
        1'b0:
        value_1back<=value_1back+1;
        endcase
        end
        else
        value_1back<=0;
2'b10:
if({1'b0,pixel[17:12]}>{1'b0,pixel[5:0]}+cal1[2][8:2]&&{1'b0,pixel[11
        begin
        case(value_1back[2])
        1'b1:
            begin
                x_sum<=x_sum+hcount-2;
                y_sum<=y_sum+vcount;
                pixel_count<=pixel_count+1;
            end
        1'b0:
        value_1back<=value_1back+1;
        endcase
        end
```

```
                  else
                  value_1back<=0;
          2'b11:
          if({1'b0,pixel[11:6]}>{1'b0,pixel[17:12]}+cal1[3][8:2]&&{1'b0,pixel[5
              begin
                  case(value_1back[2])
                      1'b1:   begin
                                  x_sum<=x_sum+hcount-2;
                                  y_sum<=y_sum+vcount;
                                  pixel_count<=pixel_count+1;
                              end
                      1'b0: value_1back<=value_1back+1;
                  endcase
              end
          else value_1back<=0;
      endcase
      //draw vertical gridlines
      case(hcount[10:2])
          (cmx[1][10:2]-hspacing) : cleaned_pixel<= {6'b100000, good, 11'b00000
          (gridcenterx-hspacing) : cleaned_pixel<= {6'b100000, good, 11'b000000
          (cmx[1][10:2]) : cleaned_pixel<= {6'b100000, good, 11'b00000000000};
          gridcenterx : cleaned_pixel<= {6'b100000, good, 11'b00000000000};
          (cmx[2][10:2]) : cleaned_pixel<= {6'b100000, good, 11'b00000000000};
          (gridcenterx+hspacing) : cleaned_pixel<= {6'b100000, good, 11'b000000
          (cmx[2][10:2]+hspacing) : cleaned_pixel<= {6'b100000, good, 11'b00000
          default : begin //draw horizontal gridlines
                  if(hcount>(cmx[1]-{hspacing,2'b0})&&hcount<(cmx[2]+{hspacing,
                      case(vcount[9:2])
                          8'd182 : cleaned_pixel <= {6'b100000, good, 11'b00000
                          8'd154 : cleaned_pixel <= {6'b100000, good, 11'b00000
                          8'd124 : cleaned_pixel <= {6'b100000, good, 11'b00000
                          8'd93 : cleaned_pixel <= {6'b100000, good, 11'b000000
                          8'd65 : cleaned_pixel <= {6'b100000, good, 11'b000000
                      default : cleaned_pixel <= {pixel_man,pixel_man,pixel_man
                      endcase
                  end
                  //draw stickman
                  else cleaned_pixel <= {pixel_man,pixel_man,pixel_man,pixel_ma
              end
      endcase
  end
 3'b101: //display regular video
    begin
     cleaned_pixel<=pixel;
  end
```

```
         default:
        begin
        end
      endcase
    end
    else
        cleaned_pixel<=0; //display nothing if in wrong range or not proper threshold
    //Center of mass displays
    if(hcount<cmx[0]+3'd5&&hcount>cmx[0]-3'd5&&vcount<cmy[0]+3'd5&&vcount>cmy[0]-3'd5
        cleaned_pixel<={6'b111111,6'b0,6'b0};
    if(hcount<cmx[1]+3'd5&&hcount>cmx[1]-3'd5&&vcount<cmy[1]+3'd5&&vcount>cmy[1]-3'd5
        cleaned_pixel<={6'b0,6'b111111,6'b0};
    if(hcount<cmx[3]+3'd5&&hcount>cmx[3]-3'd5&&vcount<cmy[3]+3'd5&&vcount>cmy[3]-3'd5
        cleaned_pixel<={6'b0,6'b111111,6'b111111};
    if(hcount<cmx[2]+3'd5&&hcount>cmx[2]-3'd5&&vcount<cmy[2]+3'd5&&vcount>cmy[2]-3'd5
        cleaned_pixel<={6'b111111,6'b111111,6'b0};
    //command image display
    if(vcount<8'd231&&hcount>xoffset+2'd2&&hcount<xoffset+9'd257)
        if(hcount<xoffset+9'd129)
            //left image
            cleaned_pixel<={data[8:6],3'b111,data[5:3],3'b111,data[2:0],3'b111};
        else
            //right image
            cleaned_pixel<={data2[8:6],3'b111,data2[5:3],3'b111,data2[2:0],3'b111};
        end
endmodule
```

A.1.11 compare_beats.v

```
//////////////////////////////////////////////////////
//compare_beats
//Written by Jonathan Burnham
//
//Compare_beats computes a score according to how well
//the user is moving with the beat of the song. Resets
//to 0 whenever reset is high
//////////////////////////////////////////////////////


module compare_beats(clock, reset, data, user_beat, score,
reset_beat);
    input clock;        // 27 MHZ
    input reset;        // Set score to zero when high
    input data;     // Current beat data from the beat memory
    input user_beat;    // 1 if the user has moved enough to set the beat
                // 0 otherwise
    output [13:0] score;    // Score from the beat
    output reset_beat;  // Goes high to reset the users beat

    reg [13:0] score;   // Stores the score
    reg reset_beat = 0; // Resets the user beat
    reg old_data;       // Used to look for edge in data

    // If the user_beat is one and data becomes one at the same
    // time, increment the score, otherwise do nothing or set the
    // score to zero if reset is high
    always @ (posedge clock) begin
        old_data <= data;
        if (reset) score <= 0;
        else begin
            if (~old_data && data && user_beat) begin
                score <= score + 14'd50;
                reset_beat <= 1;
            end
            else reset_beat <= 0;
        end
    end
endmodule
```

A.2 flash_int.v

```
//WE DID NOT MODIFY THIS FILE

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash ROM Interface
//
// For Labkit Revision 004
//
//
// Created: January 22, 2005
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
```

```
‘define FLASHOP_IDLE  2’b00 ‘define FLASHOP_READ  2’b01 ‘define
FLASHOP_WRITE 2’b10

module flash_int(reset, clock, op, address, wdata, rdata, busy,
flash_data,
         flash_address, flash_ce_b, flash_oe_b, flash_we_b,
         flash_reset_b, flash_sts, flash_byte_b);

   parameter access_cycles = 5;
   parameter reset_assert_cycles = 1000;
   parameter reset_recovery_cycles = 30;

   input reset, clock; // Reset and clock for the flash interface
   input [1:0] op; // Flash operation select (read, write, idle)
   input [22:0] address;
   input [15:0] wdata;
   output [15:0] rdata;
   output busy;
   inout [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b;
   output flash_reset_b, flash_byte_b;
   input  flash_sts;

   reg [1:0] lop;
   reg [15:0] rdata;
   reg busy;
   reg [15:0] flash_wdata;
   reg flash_ddata;
   reg [23:0] flash_address;
   reg flash_oe_b, flash_we_b, flash_reset_b;

   assign flash_ce_b = flash_oe_b && flash_we_b;
   assign flash_byte_b = 1; // 1 = 16-bit mode (A0 ignored)

   assign flash_data = flash_ddata ? flash_wdata : 16’hZ;

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

   initial
```

```
  flash_reset_b <= 1'b1;

reg [9:0] state;

always @(posedge clock)
  if (reset)
    begin
   state <= 0;
   flash_reset_b <= 0;
   flash_we_b <= 1;
   flash_oe_b <= 1;
   flash_ddata <= 0;
   busy <= 1;
    end
  else if (flash_reset_b == 0)
    if (state == reset_assert_cycles)
  begin
     flash_reset_b <= 1;
     state <= 1023-reset_recovery_cycles;
  end
    else
  state <= state+1;
  else if ((state == 0) && !busy)
    // The flash chip and this state machine are both idle. Latch the user's
    // address and write data inputs. Deassert OE and WE, and stop driving
    // the data buss ourselves. If a flash operation (read or write) is
    // requested, move to the next state.
    begin
   flash_address <= {address, 1'b0};
   flash_we_b <= 1;
   flash_oe_b <= 1;
   flash_ddata <= 0;
   flash_wdata <= wdata;
   lop <= op;
   if (op != `FLASHOP_IDLE)
     begin
        busy <= 1;
        state <= state+1;
     end
   else
     busy <= 0;
    end
  else if ((state==0) && flash_sts)
    busy <= 0;
  else if (state == 1)
```

```
        // The first stage of a flash operation. The address bus is already set,
        // so, if this is a read, we assert OE. For a write, we start driving
        // the user's data onto the flash databus (the value was latched in the
        // previous state.
        begin
       if (lop == `FLASHOP_WRITE)
         flash_ddata <= 1;
       else if (lop == `FLASHOP_READ)
         flash_oe_b <= 0;
       state <= state+1;
        end
      else if (state == 2)
        // The second stage of a flash operation. Nothing to do for a read. For
        // a write, we assert WE.
        begin
       if (lop == `FLASHOP_WRITE)
         flash_we_b <= 0;
       state <= state+1;
        end
      else if (state == access_cycles+1)
        // The third stage of a flash operation. For a read, we latch the data
        // from the flash chip. For a write, we deassert WE.
        begin
       if (lop == `FLASHOP_WRITE)
         flash_we_b <= 1;
       if (lop == `FLASHOP_READ)
         rdata <= flash_data;
       state <= 0;
        end
      else
        begin
       if (!flash_sts)
         busy <= 1;
       state <= state+1;
        end

endmodule
```

A.2.1 gen_model.v

```
/**********************************************************************
```

```
 **
 ** Module: ycrcb2rgb
 ** YCrCb to RGB Converter
 ** The module from Xilinx converts a 30-bit YCrCb value to a 24-bit RGB
 **   value.  Constants are hardcoded into the conversion.
 ** NOT WRITTEN BY KEVIN, JON, OR CHRIS
 **
 **
 ** Generic Equations:
 ***************************************************************************/

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

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

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

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

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

// Stores the YCrCb colors in registers
// Clears the registers on reset
always @ (posedge clk or posedge rst)
   if (rst)
      begin
      Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
      end
   else
      begin
      Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
      end

// Creates intermediate values for RGB calculations
always @ (posedge clk or posedge rst)
   if (rst)
```

```
      begin
       A_int <= 0; B1_int <= 0; B2_int <= 0; C_int <= 0; X_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     A_int <= (const2 * (Cr_reg - 'd512));
     B1_int <= (const3 * (Cr_reg - 'd512));
     B2_int <= (const4 * (Cb_reg - 'd512));
     C_int <= (const5 * (Cb_reg - 'd512));
     end


// Pipelined continuation of the RBG calculation
always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     R_int <= X_int + A_int;
     G_int <= X_int - B1_int - B2_int;
     B_int <= X_int + C_int;
     end


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

endmodule
```

A.2.2 grid_detect.v

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

```
// 6.111 module - grid_cetect
//
//
// File:   grid_detect.v
// Date:   13-Dec-06
//
// Created: December 13, 2006
// Author: Kevin Miu
//
// This module performs an algorithm similar to a binary search in order to
// detect the quadrant that a center of mass is in.  The quadrant positions
// change as the position of two internal center of masses, in this case, cmx1,
// cmy1, cmx2, and cmy2 (4 coordinates for 2 centers of masses) change.
//
// Inputs:
//
//   clk - 65 mhz clock;
//   [10:0] cmx0,cmx1,cmx2,cmx3; // horizontal positions of center of masses
//   [10:0] gridcenterx; //center vertical gridline;
//   [10:0] hspacing; // horizontal spacing between vertical gridlines
//   [9:0] cmy0,cmy1,cmy2,cmy3; // vertical positions of center of masses
//
//
// Outputs:
//
//   [2:0] xlevel0; // column of first center of mass (red)
//   [1:0] ylevel0; // row of first center of mass (red)
//   [2:0] xlevel3; // column of second center of mass (teal)
//   [1:0] ylevel3; // row of second center of mass (teal)
//
////////////////////////////////////////////////////////////////////////////////
module
grid_detect(clk,cmx0,cmy0,cmx1,cmy1,cmx2,cmy2,cmx3,cmy3,gridcenterx,hspacing,xlevel0,
   input clk - 65 mhz clock;
   input [10:0] cmx0,cmx1,cmx2,cmx3; // horizontal positions of center of masses
   input [10:0] gridcenterx; //center vertical gridline;
   input [10:0] hspacing; // horizontal spacing between vertical gridlines
   input [9:0] cmy0,cmy1,cmy2,cmy3; // vertical positions of center of masses
   output [2:0] xlevel0; // column of first center of mass (red)
   output [1:0] ylevel0; // row of first center of mass (red)
   output [2:0] xlevel3; // column of second center of mass (teal)
   output [1:0] ylevel3; // row of second center of mass (teal)
   reg [2:0] xlevel0;
   reg [1:0] ylevel0;
   reg [2:0] xlevel3;
   reg [1:0] ylevel3;
   reg [4:0] state; // state of detection FSM


Columns and Rows
------------------------
|_0_|_1_|_2_|_3_|_4_|_5_| |_1_| |_2_| |_3_|

// This always block contains the entire FSM.  The always block performs
// a binary search given the coordinate of a center of mass and the positions
// of the gridlines.  The grid in which the red center of mass and the teal
// center of mass are calculated.
   always@(posedge clk)
   begin
    case(state)
    5'd0:
        // detect if the x value is greater or above the x
        // centerline
        if(cmx0>=gridcenterx)
            // if yes go to state 1
            state<=1;
        else
            // else skip ahead to check the low half
            state<=3;
    5'd1:
        // if it is greater than the highest gridline, it's in the
        // highest column
        if(cmx0>=cmx2+hspacing)
            begin
                xlevel0<=5;
                state<=5;
            end
        // else it is in one of two lower columns
        else
            state<=2;
    5'd2:
        // is it in the middle column of the three columns above the centerline
        if(cmx0>=cmx2)
            begin
                xlevel0<=4;
                state<=5;
            end
        // or in the column directly above the centerline
        else
            begin
```

```
                state<=5;
                xlevel0<=3;
            end
    5'd3:
        if(cmx0>=cmx1)
            begin
                xlevel0<=2;
                state<=5;
            end
        else
            state<=4;
    5'd4:
        if(cmx0>=cmx1-hspacing)
            begin
                xlevel0<=1;
                state<=5;
            end
        else
            begin
                state<=5;
                xlevel0<=0;
            end
    5'd5:
        if(cmy0>=10'd496)
            state<=6;
        else
            state<=7;
    5'd6:
        if(cmy0>=10'd616)
            begin
                state<=8;
                ylevel0<=3;
            end
        else
            begin
                state<=8;
                ylevel0<=2;
            end
    5'd7:
        if(cmy0>=10'd372)
            begin
                state<=8;
                ylevel0<=1;
            end
        else
```

```
                begin
                    state<=8;
                    ylevel0<=0;
                end
        5'd8:
            state<=9;
        5'd9:
            if(cmx3>=gridcenterx)
                state<=10;
            else
                state<=12;
        5'd10:
            if(cmx3>=cmx2+hspacing)
                begin
                    xlevel3<=5;
                    state<=14;
                end
            else
                state<=11;
        5'd11:
            if(cmx3>=cmx2)
                begin
                    xlevel3<=4;
                    state<=14;
                end
            else
                begin
                    state<=14;
                    xlevel3<=3;
                end
        5'd12:
            if(cmx3>=cmx1)
                begin
                    xlevel3<=2;
                    state<=14;
                end
            else
                state<=13;
        5'd13:
            if(cmx3>=cmx1-hspacing)
                begin
                    xlevel3<=1;
                    state<=14;
                end
            else
                begin
                    state<=14;
                    xlevel3<=0;
                end
        5'd14:
            if(cmy3>=10'd496)
                state<=15;
            else
                state<=16;
        5'd15:
            if(cmy3>=10'd616)
                begin
                    state<=17;
                    ylevel3<=3;
                end
            else
                begin
                    state<=17;
                    ylevel3<=2;
                end
        5'd16:
            if(cmy3>=10'd372)
                begin
                    state<=17;
                    ylevel3<=1;
                end
            else
                begin
                    state<=17;
                    ylevel3<=0;
                end
        5'd17:
            state<=0;

        endcase
       end
    endmodule
```

A.2.3 grid_score.v

```
///////////////////////////////////////////////////////////////
//grid_score
//Written by Jonathan Burnham
```

```
//
//Used to compare the grids of the picture and that of the users
//hands to determine if the user gets a score. Keeps track of the
//users grid score.
////////////////////////////////////////////////////////////

module grid_score(clock, reset, red_x, red_y, teal_x, teal_y,
first_half, second_half,
          finished_scrolling, score, red_right, teal_right, good);
    input clock;        // 64.8 MHZ
    input reset;        // Sets score to 0 when high
    input [2:0] red_x;  // x grid of the users red hand
    input [1:0] red_y;  // y grid of the users red hand
    input [2:0] teal_x; // x grid of the users teal hand
    input [1:0] teal_y; // y grid of the users teal hand
    input [7:0] first_half; // grid of the pictures right hand
    input [7:0] second_half; // grid of the pictures left hand
    input finished_scrolling; // 1 when the picture has reached the end of the screen
    output [13:0] score;      // grid score
    output red_right;         // 1 if the red hand is in the right quadrant
    output teal_right;    // 1 if the teal hand is in the right quadrant
    output good;          // 1 if the user was correct in the last grid, 0 if not


    reg old_finished_scrolling; // Used to detect edge of finished_scrolling
    reg [13:0] score;       // Stores the score
    reg red_right;          // Stores red_right
    reg teal_right;         // Stores teal_right


    // First, decide if the hands are in their proper quadrants, have to use
    // a case statement because the numbering of the quadrants is different
    // between the users grid and the pictures grid
    always @ (posedge clock) begin
        old_finished_scrolling <= finished_scrolling;

        // Decide if the teal hand is in the right position
        case({teal_x,teal_y})
            {3'd0,2'd0} : teal_right <= (second_half == 9) ? 1 : 0;
            {3'd1,2'd0} : teal_right <= (second_half == 10) ? 1 : 0;
            {3'd2,2'd0} : teal_right <= (second_half == 11) ? 1 : 0;
            {3'd0,2'd1} : teal_right <= (second_half == 6) ? 1 : 0;
            {3'd1,2'd1} : teal_right <= (second_half == 7) ? 1 : 0;
            {3'd2,2'd1} : teal_right <= (second_half == 8) ? 1 : 0;
            {3'd0,2'd2} : teal_right <= (second_half == 3) ? 1 : 0;
            {3'd1,2'd2} : teal_right <= (second_half == 4) ? 1 : 0;
            {3'd2,2'd2} : teal_right <= (second_half == 5) ? 1 : 0;
            {3'd0,2'd3} : teal_right <= (second_half == 0) ? 1 : 0;
            {3'd1,2'd3} : teal_right <= (second_half == 1) ? 1 : 0;
            {3'd2,2'd3} : teal_right <= (second_half == 2) ? 1 : 0;
                 default : teal_right <= 0;
        endcase

        // Decide if the red hand is in the right position
        case({red_x,red_y})
            {3'd3,2'd0} : red_right <= (first_half == 11) ? 1 : 0;
            {3'd4,2'd0} : red_right <= (first_half == 10) ? 1 : 0;
            {3'd5,2'd0} : red_right <= (first_half == 9) ? 1 : 0;
            {3'd3,2'd1} : red_right <= (first_half == 8) ? 1 : 0;
            {3'd4,2'd1} : red_right <= (first_half == 7) ? 1 : 0;
            {3'd5,2'd1} : red_right <= (first_half == 6) ? 1 : 0;
            {3'd3,2'd2} : red_right <= (first_half == 5) ? 1 : 0;
            {3'd4,2'd2} : red_right <= (first_half == 4) ? 1 : 0;
            {3'd5,2'd2} : red_right <= (first_half == 3) ? 1 : 0;
            {3'd3,2'd3} : red_right <= (first_half == 2) ? 1 : 0;
            {3'd4,2'd3} : red_right <= (first_half == 1) ? 1 : 0;
            {3'd5,2'd3} : red_right <= (first_half == 0) ? 1 : 0;
                 default : red_right <= 0;
        endcase

        if (reset) score <= 0;      // If reset, set the score to 0
        // If we just reached the end of the screen, if the user has their
        // hands in the correct spot, increment their score and set good
        // to one to display a yellow grid. If they were not correct, set
        // good to zero to display a red grid
        else if (~old_finished_scrolling && finished_scrolling) begin
                if (red_right && teal_right) begin
                    score <= score + 14'd200;
                    good <= 1;
                end
                else good <= 0;
        end
    end
endmodule
```

\subsection{high_speed_enable.v}

```
\begin{verbatim}
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Christopher Hoffman
//
// Create Date:    16:18:01 12/01/06
// Design Name:
// Module Name:    high_speed_enable
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
//Generates a high speed clock and if enable is high output the high speed clock othe
module high_speed_enable(clock_27mhz, ready, enable, new_clock);
        input clock_27mhz, ready, enable;
        output new_clock;

        reg[2:0] delay_counter = 0; //1/8 clock speed ~ 3MHz
        reg quick_clock;

        assign  new_clock = quick_clock | (ready & normal_pb); //select between regul

        always @ (posedge clock_27mhz) begin //this block generates a high speed cloc
            if(enable) begin
            if (delay_counter == 0) begin
                quick_clock <= 1;
                delay_counter <= 1;
            end
            else begin
                delay_counter <= delay_counter + 1;
                quick_clock <= 0;
            end
            end
        end

endmodule
```

A.2.4 interface.v

```
//////////////////////////////////////////////////////
//interface
//Written by Jonathan Burnham
//
//Displays user interface on the screen and controls
//whether the interface or the game play screen is
//displayed. Also controls the difficulty setting,
//menu controls, and high score
//
//////////////////////////////////////////////////////

module interface(clock, reset, hcount, vcount, arrow_x, arrow_y,
rec_x, rec_y, b_up,
         b_down, b_left, b_right, b_enter, button_0, game_pixels, pixels, start_song,
         state_audio, video_state, default_calib, difficulty, score_beat, score_grid,
         score_total, led_out);
    input clock;        // 65 MHZ
    input reset;        // Resets the menu cursor to the starting position
    input [10:0] hcount;    // Current x position being displayed
    input [9:0] vcount; // Current y position being displayed
    output [10:0] arrow_x;  // Arrow cursor x position
    output [9:0] arrow_y;   // Arrow cursor y position
    output [10:0] rec_x;    // Difficulty rectangle x position
    output [9:0] rec_y; // Difficulty rectangle y position
    input b_up;     // Button up
    input b_down;       // Button down
    input b_left;       // Button left
    input b_right;      // Button right
    input b_enter;      // Button enter
    input button_0;     // Button 0
    input [17:0] game_pixels; // Pixels being calculated by cm_calculator
    output [23:0] pixels;     // Pixels to be displayed on the monitor
    output start_song;    // Command line to start playing the song
    input [1:0] state_audio;  // Current state of the audio control module
    output [2:0] video_state; // Current state of the video control module
    output default_calib;     // Load default calibration values
    output [1:0] difficulty;  // Difficulty setting chosen by user
    input [13:0] score_beat;  // Score from getting the beat
    input [13:0] score_grid;  // Score from matching the grids
    input [13:0] score_total; // Total score
    output [63:0] led_out;    // Data to be displayed on the LED panel

    wire [15:0] addr;     // Address to access the menu screen picture
    wire [11:0] data;     // Data from the menu picture BRAM
    wire [23:0] pixelsArrow, pixelsInterf;  // Pixels from the arrow sprite and
                                // the combination of the arrow sprite
                                // and the menu picture
    wire [2:0] pixelsRectangle;         // Pixels from the rectangle sprite
    wire [5:0] state;                   // State of the menu system
    wire up,down,left,right,enter;      // Edge detector wires


    reg [10:0] arrow_x = 11'd715;       // Load default settings for arrow x position
    reg [9:0] arrow_y = 10'd85; // Load default settings for arrow y position
    reg [10:0] rec_x = 11'd820;     // Load default settings for rectangle x position
    reg [9:0] rec_y = 10'd720;      // Load default settings for rectangle y position
    reg pixel_control = 1;          // Controls whether the interface (1) or game pla
    reg start_song = 0;         // Controls whether or not to start the song
    reg done_beat_detection = 0;        // Remembers whether or not the beat has been
    reg [1:0] old_state_audio = 0;  // Used to find the moment of change from one sta
    reg default_calib = 0;          // Controls whether or not to load default calibr
    reg [2:0] video_state;          // Controls video state
    reg [1:0] difficulty = 0;       // Difficuly chosen by the user
    reg disp_hs = 0;                // Controls whether to display the high score
    reg [63:0] led_out;         // Data to be displayed on the LED panel
    reg [2:0] status = 0;           // Status the cursor is at
    reg b_up_,b_down_,b_enter_,b_left_,b_right_,button_0_;  // Old button values used
    reg [3:0] high_counter = 0;     // Used to create a delay for timing differences
    reg calibrated = 1;         // By default, want to load initial calibration value
    reg[13:0] high_score = 0;       // Stores the high score


    interface_pic interface_pic(addr,clock,data);   // Contains the menu picture
    arrow arrow(clock, hcount, vcount, arrow_x, arrow_y, pixelsArrow);  // Sprite to
    rectangle rectangle(clock,rec_x,rec_y,11'd1020,10'd760,hcount,vcount,pixelsRectan
                                                // the rectangle to cover
                                                // up the difficulty boxes


    assign addr = hcount[10:2] + (vcount[9:2] << 8);    // Calculates the address bas
```

```
// Forms the interface pixels by combining the arrow and menu picture pixels. How
// area of the rectangle, we want to blank it out to cover up the unused difficul
assign pixelsInterf = pixelsRectangle[2] == 1 ? {3{4'b0, 4'b1111}} :
                                pixelsArrow | {data[11:8],4'b1111,
                                           data[7:4],4'b1111,
                                           data[3:0],4'b1111};

// Assign the output pixels to be either the interface or the game play
assign pixels[23:16] = pixel_control ? pixelsInterf[23:16] : {game_pixels[17:12],
assign pixels[15:8] = pixel_control ? pixelsInterf[15:8] : {game_pixels[11:6], 2'
assign pixels[7:0] = pixel_control ? pixelsInterf[7:0] : {game_pixels[5:0], 2'b11

// Create edge detectors
assign up = b_up & ~b_up_;
assign down = b_down & ~b_down_;
assign enter = b_enter & ~b_enter_;
assign left = b_left & ~b_left_;
assign right = b_right & ~b_right_;
assign b0 = button_0 & ~button_0_;

// Assign state for FSM. Note that we use up, down, and enter as values for the F
// This avoids having to create lots of if statements for each case
assign state = {up, down, enter, status};

// Block keeps track of high score, saves values for edge detectors, controls the
// and difficulty, and controls song playback and beat detection
always @ (posedge clock) begin
    if(high_score < score_total) high_score <= score_total;
    led_out <= disp_hs ? {50'b0, high_score} : {18'b0, score_total, 2'b0, score_g
    b_up_ <= b_up;
    b_down_ <= b_down;
    b_enter_ <= b_enter;
    b_left_ <= b_left;
    b_right_ <= b_right;
    button_0_ <= button_0;
    old_state_audio <= state_audio;

    if (reset) begin
        status <= 0;
        arrow_x <= 11'd715;
        arrow_y <= 10'd85;
    end
    case (state)
        6'b010000: begin    // Move down from quick play
                arrow_y <= 10'd185;
```

```
            status <= 1;
             end
    6'b010001: begin    // Move down from calibration
            arrow_y <= 10'd275;
            status <= 2;
             end
    6'b100001: begin    // Move up from calibration
            arrow_y <= 10'd85;
            status <= 0;
             end
    6'b100010: begin    // Move up from high score
            arrow_y <= 10'd185;
            status <= 1;
             end
    6'b001000: begin    // Enter at quick play
            pixel_control <= 0; // Display game play
            start_song <= 1;        // Start the song
            status <= 7;        // Status to play
            video_state <= 3'b100;  // Set video state to stickman
            high_counter <= 1;  // Set delay counter
            default_calib <= calibrated;    // Set whether or not to
                            // calibrate with defaults
            disp_hs <= 0;       // Display users current score
             end
    6'b001001: begin    // Enter at calibrate
            pixel_control <= 0; // Display calibration
            status <= 6;        // Status to calibrate
            video_state <= 3'b0;    // Set video status to red calibrate
            calibrated <= 0;        // set calibrate to 0
           end
    6'b001010: disp_hs <= 1;        // Enter at high score, display high scor
endcase


// Handles start kbeat detection and song playback
if (status == 7) begin
    default_calib <= 0;
    if (high_counter == 0) start_song <= 0;
    else high_counter <= high_counter + 1;
    if (state_audio == 2'b01 && start_song == 0) begin
        if (~done_beat_detection) begin
            start_song <= 1;
            done_beat_detection <= 1;
            high_counter <= 1;
        end
```

```
            else begin
                pixel_control <= 1;
                start_song <= 0;
                status <= 0;
                arrow_x <= 11'd715;
                arrow_y <= 10'd85;
            end
        end
        if (b0) pixel_control <= 1; // Allow the user to force back to the menu s
    end

    // Handles walking through calibration modes and returning
    // to the menu when enter is pressed
    if (status == 6) begin
        if (enter) begin
            pixel_control <= 1;
            status <= 0;
            arrow_x <= 11'd715;
            arrow_y <= 10'd85;
        end
        if (b0) video_state <= (video_state == 3'b101) ? 0 : video_state + 1;
    end

    // Increase or decrease difficulty according to button presses
    if (right && difficulty < 2) difficulty <= difficulty + 1;
    if (left && difficulty > 0) difficulty <= difficulty - 1;

    // Set how many boxes the rectangle covers according to the difficulty
    case (difficulty)
        0: begin
                rec_x <= 11'd820;
                rec_y <= 10'd680;
            end
        1: begin
                rec_x <= 11'd915;
                rec_y <= 10'd680;
            end
        2: begin
                rec_x <= 11'd1020;
                rec_y <= 10'd760;
            end
    endcase

end
```

```
endmodule
```

A.2.5 line.v

```
////////////////////////////////////////////////////////////////////////
//line
//Written by Jonathan Burnham
//
//Line is a sprite that draws a line of variable width and color
//between two x,y coordinates
////////////////////////////////////////////////////////////////////////
module line(clock,x1,y1,x2,y2,hcount,vcount,pixel);
    input clock;        // 64.8 MHZ
    input [10:0] hcount;    // Current video x position
    input [10:0] x1,x2; // Line x endpoints
    input [9:0] vcount; // Current video y position
    input [9:0] y1,y2;  // Line y endpoints
    output [2:0] pixel;

    // Parameters control the thickness of the line as well
    // as its 8'bit color
    parameter COLOR = 3'b111;
    parameter WIDTH = 3'd5;

    reg [2:0] pixel;            // Stores pixel value
    reg signed [11:0] x1_x2;        // Difference between x endpoints
    reg signed [11:0] x1_x2_abs;        // Absolute value of x difference
    reg signed [10:0] y1_y2;        // Difference between y endpoints
    reg signed [10:0] y1_y2_abs;        // Absolute value of y difference
    reg signed [15:0]  avg1;        // Average of differences

    reg signed [11:0] current_dx;       // Difference between current x and x1
    reg signed [10:0] current_dy;       // Difference between current y and y1
    reg signed [11:0] alternate_dx;     // Difference between current x and x2
    reg signed [10:0] alternate_dy;     // Difference between current y and y2
    reg signed [24:0] diffx_dy;     // Stores (x1-x2)*(y1-vcount)
    reg signed [24:0] diffy_dx;     // Stores (y1-y2)*(x1-hcount)


    // Pipelines all the values, and then performs check to see if current hcount and
    // vcount are on the line
    always @ (posedge clock) begin
```

```
        x1_x2 <= x1 - x2;
        y1_y2 <= y1 - y2;
        x1_x2_abs <= x1_x2[11] ? ~x1_x2+1 : x1_x2;
        y1_y2_abs <= y1_y2[10] ? ~y1_y2+1 : y1_y2;
        avg1 <= (x1_x2_abs + y1_y2_abs) << 2;

        current_dx <= x1 - hcount;
        current_dy <= y1 - vcount;
        alternate_dx <= x2 - hcount;
        alternate_dy <= y2 - vcount;

        diffx_dy <= x1_x2 * current_dy;
        diffy_dx <= y1_y2 * current_dx;


        if (x1_x2[11] == current_dx[11] && y1_y2[10] == current_dy[10]  &&    // Test
            x1_x2[11] != alternate_dx[11] && y1_y2[10] != alternate_dy[10]) begin //
                                              // seeing if the signs match up

            // Perform comparison
            if ( ((diffx_dy - diffy_dx) <= avg1) && ((diffy_dx - diffx_dy) <= avg1) )
            else pixel <= 0;
        end
        else pixel <= 0;
    end


endmodule
```

A.2.6 ntsc2zbt.v

```
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// 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.
//
// INPUTS
//   clk - system clock
//   vclk - video clock from camera
//   [2:0] fvh - field, vertical, horizontal - encoding for interlacing, vsync, hsync
//   dv - data valid
//   [23:0] din - input pixel data
//   sw - switch which determines mode (for debugging)
//
// OUTPUTS
//   ntsc_we - write enable for NTSC data
//   [18:0] ntsc_addr - NTSC address
//   [35:0] ntsc_data - data to be written to the ZBT
//
///////////////////////////////////////////////////////////////////////////
// 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 [23:0]      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'd156;

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

   reg [9:0]     col = 0;
   reg [9:0]     row = 0;
   reg [17:0]    vdata = 0;
   reg       vwe;
   reg       old_dv;
   reg       old_frame; // frames are even / odd interlaced
   reg       even_odd;  // decode interlaced frame to this wire
   wire      frame = fvh[2];
   wire      frame_edge = frame & ~old_frame;


// In this always block, the interlacing pattern is determined
   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[23:18],din[15:10],din[7:2]} : vdata;
      end
     end

   // synchronize with system clock

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

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

   // edge detection on write enable signal

   reg old_we;
   wire we_edge = we[1] & ~old_we;
   always @(posedge clk) old_we <= we[1];
   // shift each set of four bytes into a large register for the ZBT

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

   // compute address to store data in

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

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

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

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

// determine which NTSC mode to display in
   always @(posedge clk)
     if ( ntsc_we )
       begin
      ntsc_addr <= sw ? myaddr2 : myaddr;   // normal and expanded modes
      ntsc_data <= sw ? {mydata2} : {mydata};
       end

endmodule // ntsc_to_zbt
```

A.2.7 rectangle.v

```
//////////////////////////////////////////////////////////////////////
//rectangle
//Written by Jonathan Burnham
//
//Computes a rectangle sprite between two x,y coordinates, one being
//the top left corner, the other coordinates the bottom right corner
/////////////////////////////////////////////////////////////////////
```

```
module rectangle(clock,x1,y1,x2,y2,hcount,vcount,pixel);
   input clock;         // 64.8 MHZ
   input [10:0] hcount;     // Current video x position
   input [10:0] x1,x2;      // Top left and bottom right x coords
   input [9:0] vcount;      // Current video y position
   input [9:0] y1,y2;       // Top left and bottom right y coords
   output [2:0] pixel;      // Output pixel for rectangle

   parameter COLOR = 3'b111;    // Default color is white

   reg [2:0] pixel;     // Stores pixel value

   // If the current pixel is within the rectangles boundaries, then set the output
   // pixel to the color
   always @ (posedge clock) begin
    if ((hcount >= x1 && hcount <= x2) && (vcount >= y1 && vcount <= y2)) pixel <= CO
        else pixel <= 0;
   end
endmodule
```

A.2.8 scroll_control.v

```
/////////////////////////////////////////////////////////////
//scroll_control
//Written by Jonathan Burnham
//
//Scroll_control looks ahead in the beat memory and if no
//other dance picture is scrolling, starts scrolling a dance
//picture if it sees a beat to ensure that the picture hits
//the left side of the screen at the same time as the beat
/////////////////////////////////////////////////////////////


module scroll_control(clock, beat_address, done_scrolling, data,
address, start_scrolling, new_pic);
    input clock;            // 27MHZ
    input [11:0] beat_address;  // Current address in the beat memory
    input done_scrolling;       // 1 when we can start the next picture scrolling
    input data;         // Data from beat memory BRAM
    output [11:0] address;      // Look ahead address in beat memory
    output start_scrolling;     // 1, starts scrolling a new picture
    output new_pic;         // Toggles to load a new picture
    assign address = beat_address + 12'd30; // Looks ahead in the beat memory by a co

    reg start_scrolling = 0;        // Stores whether or not to scroll
    reg new_pic = 0;            // Toggles to load a new picture

    // If it has finished scrolling from the previous one and there is a beat ahead i
    // start a new picture scrolling
    always @ (posedge clock) begin
        if (data == 1 && done_scrolling == 1) begin
            start_scrolling <= 1;
            new_pic <= ~new_pic;
        end
        else start_scrolling <= 0;
    end

endmodule
```

A.2.9 stickman.v

```
///////////////////////////////////////////////////////
//stickman
//Written by Jonathan Burnham
//
//This module uses three rectangle sprites and two line
//sprites to draw the upper torso of a stick figure that
//resembles the user. The shoulder and hand positions are
//calculated using the center of mass positions calculated by
//the calculate_cm module. Full pipelining has been done
//in order to minimize the heavy computation it takes to
//generate this model
//////////////////////////////////////////////////////


module stickman
(vclock,reset,leftArmX,leftArmY,leftShoulderX,leftShoulderY,rightShoulderX,rightShoul
         rightArmX,rightArmY,hcount,vcount,hsync,vsync,blank,phsync,pvsync,pblank,pix
    input vclock;       // 65MHz clock
    input reset;            // 1 to initialize module
    input [10:0] leftArmX,leftShoulderX,rightShoulderX,rightArmX;   // Coordinates fo
    input [9:0]  leftArmY,leftShoulderY,rightShoulderY,rightArmY;
    input [10:0] hcount;        // horizontal index of current pixel (0..1023)
    input [9:0]     vcount;     // vertical index of current pixel (0..767)
    input hsync;            // XVGA horizontal sync signal (active low)
    input vsync;            // XVGA vertical sync signal (active low)
    input blank;            // XVGA blanking (1 means output black pixel)
    input enable;
    output phsync;      // stickman's horizontal sync
    output pvsync;      // stickman's vertical sync
    output pblank;      // stickman's blanking
    output [2:0] pixel;     // stickman's pixel

    reg oldVsync;

    wire[2:0] leftArmPixel, rightArmPixel, leftLegPixel, rightLegPixel, torsoPixel, n

    reg [10:0] torsoWidth;
    reg [10:0] torsoHeight;
    reg [10:0] height;
    reg [10:0] torsoCenter;
    reg [10:0] neckWidth;
    reg [10:0] headWidth;
    reg [10:0] rightLegX1, leftLegX1, neckX1, neckX2, headX1, headX2;
    reg [9:0]  legY1, legY2, neckY1, headY1;

    // Creates the different body parts
    line leftArm(vclock,leftShoulderX,height,leftArmX,leftArmY,hcount,vcount,leftArmP
    line rightArm(vclock,rightArmX,rightArmY,rightShoulderX,height,hcount,vcount,righ
    rectangle torso(vclock,leftShoulderX,height,rightShoulderX,legY2,hcount,vcount,to
    rectangle neck(vclock,neckX1,neckY1,neckX2,height,hcount,vcount,neckPixel);
    rectangle head(vclock,headX1,headY1,headX2,neckY1,hcount,vcount,headPixel);
    defparam leftArm.WIDTH = 3'd3;
    defparam rightArm.WIDTH = 3'd3;

    reg [11:0] sum1, sum2, shift1, shift2, shift3, shift4, shift5; // Temporary varia

    // Calculates values to ensure a scaled stickman:
    // Height of Torso: 1.5 * torsoWidth
    // Width of head: 0.5 * torsoWidth
    // Width of neck: 0.25 * torsoWidth
    // Averages the y positions of the shoulders to come up with an average height
    always @ (posedge vclock) begin
        // Calculates body part sizes
        torsoWidth <= rightShoulderX - leftShoulderX;
```

```
        shift1 <= torsoWidth << 1;
        shift2 <= torsoWidth >> 1;
        torsoHeight <= shift1 - shift2;
        sum1 <= rightShoulderY + leftShoulderY;
        height <= sum1 >> 1;
        sum2 <= rightShoulderX + leftShoulderX;
        torsoCenter <= sum2 >> 1;
        headWidth <= torsoWidth >> 2;
        neckWidth <= headWidth >> 2;

        // Calculates where on the body the parts should go
        legY2 <= height + torsoHeight;
        neckX1 <= torsoCenter - neckWidth;
        shift3 <= neckWidth << 1;
        neckY1 <= height - shift3;
        neckX2 <= torsoCenter + neckWidth;
        headX1 <= torsoCenter - headWidth;
        shift4 <= headWidth << 1;
        headY1 <= neckY1 - shift4;
        headX2 <= torsoCenter + headWidth;
    end

    // Use default video signals
    assign phsync = hsync;
    assign pvsync = vsync;
    assign pblank = blank;

    // Output pixels are all of our body part pixels joined together
    assign pixel = leftArmPixel | rightArmPixel | torsoPixel | neckPixel | headPixel;
endmodule
```

A.2.10 storage_tester.v

```
‘timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Christopher Hoffman
//
// Create Date:    12:33:42 12/02/06
// Design Name:
// Module Name:    storage_tester
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module storage_tester(stop, clock_27mhz, clock_16hz, start,
beat_address,state,pb_start,pb_pause,pb_stop, reset_play,
stop_address, faddress, be);
    input stop, clock_27mhz, clock_16hz;
    output start;
    output[11:0] beat_address;
    output[1:0] state;
    output pb_start, pb_pause, pb_stop;
    input[22:0] faddress;
    output reset_play;
    input [22:0] stop_address;
    input be;

    reg [11:0] beat_address = 0;
    reg[1:0] state = 0;
    reg pb_start, pb_pause, pb_stop;
    reg old_clock = 0;
    wire clock;
    reg start = 0;
    reg reset_play = 1;
    reg old_be = 0;
    wire go;

    assign clock = ~old_clock & clock_16hz;

    assign go = be & ~old_be;

    always @ (posedge clock_27mhz) begin
        old_be <= be;
        old_clock <= clock_16hz;
        if (state == 0) begin
            if (go) begin //start beat detection
                start <= 1;
                reset_play <= 0;
            end
            else if (stop == 1) state <= 1; //done with beat detection
            else start <= 0;
        end
        else if (state == 1) begin //start audio playback
            pb_start <= 1;
            reset_play <= 1;
            beat_address <= 0;
            if (go) begin
                reset_play <= 0;
                state <= 2;
            end
        end
        else if (state == 2) begin //play back beats
            if (faddress >= stop_address) state <= 1;
            else if (clock) beat_address <= beat_address + 1;
        end
    end
endmodule
```

A.2.11 test_fsm.v

```
// Comments: we use states 10 -12 for playback. other states are initialization
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash Tester State Machine
//
//
// Created: January 23, 2005
// Author: Nathan Ickes
// Modified: Christopher Hoffman
//
////////////////////////////////////////////////////////////////////////////////

‘define STATUS_RESET             4’h0 ‘define STATUS_READ_ID
4’h1 ‘define STATUS_CLEAR_LOCKS       4’h2 ‘define STATUS_ERASING
4’h3 ‘define STATUS_WRITING           4’h4 ‘define STATUS_READING
4’h5 ‘define STATUS_SUCCESS           4’h6 ‘define
STATUS_BAD_MANUFACTURER  4’h7 ‘define STATUS_BAD_SIZE         4’h8
‘define STATUS_LOCK_BIT_ERROR    4’h9 ‘define
STATUS_ERASE_BLOCK_ERROR 4’hA ‘define STATUS_WRITE_ERROR      4’hB
```

```
‘define STATUS_READ_WRONG_DATA   4’hC

‘define NUM_BLOCKS 128 ‘define BLOCK_SIZE 64*1024 ‘define
LAST_BLOCK_ADDRESS ((‘NUM_BLOCKS-1)*‘BLOCK_SIZE) ‘define
LAST_ADDRESS (‘NUM_BLOCKS*‘BLOCK_SIZE-1)

module test_fsm (reset, clock, fop, faddress, fwdata, frdata, fbusy,
dots, ready, dout, new_address, stop_address, address_reset);

   input reset, clock;
   output [1:0] fop;
   output [22:0] faddress;
   output [15:0] fwdata;
   input [15:0]  frdata;
   input fbusy;
   output [639:0] dots;
    input ready;
    input[7:0] dout;
    input[22:0] new_address;
    input[22:0] stop_address;
    input address_reset;

   reg [1:0] fop;
   reg [22:0] faddress;
   reg [15:0] fwdata;
   reg [639:0] dots;

   ////////////////////////////////////////////////////////////////////////////
   //
   // State Machine
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [7:0] state;
   reg [3:0] status;

   always @(posedge ready)
     if (reset)
       begin
      state <= 0;
      status <= ‘STATUS_RESET;
      faddress <= 0;
      fop <= ‘FLASHOP_IDLE;
       end
     else if (!fbusy && (fop == ‘FLASHOP_IDLE))
```

```
  case (state)
8'h00:
  begin
     // Issue "read id codes" command
     status <= `STATUS_READ_ID;
     faddress <= 0;
     fwdata <= 16'h0090;
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end

8'h01:
  begin
     // Read manufacturer code
     faddress <= 0;
     fop <= `FLASHOP_READ;
     state <= state+1;
  end

8'h02:
  if (frdata != 16'h0089) // 16'h0089 = Intel
    status <= `STATUS_BAD_MANUFACTURER;
  else
    begin
   // Read the device size code
   faddress <= 1;
   fop <= `FLASHOP_READ;
   state <= 8'h10;
    end

////////////////////////////////////////////////////////////////////////
// Read back data
////////////////////////////////////////////////////////////////////////

8'h10: /enter read mode
  begin
     status <= `STATUS_READING;
     fwdata <= 16'hFF; // Issue "read array" command
     fop <= `FLASHOP_WRITE;
     faddress <= 0;
     state <= state+1;
  end
8'h11:
  begin
     fop <= `FLASHOP_READ;
```

```
          state <= state+1;
       end
     8'h12: //read back data
        begin
         if (faddress >= stop_address) //loop back to start of song
           begin
          faddress <= new_address;
             fop <= `FLASHOP_READ;
           end
         else if (address_reset == 1) begin //restart at beginning of song
                faddress <= new_address;
                fop <= `FLASHOP_READ;
            end
          else //playback
           begin

             faddress <= faddress+2; //increment by two as it takes two cycles to com
             fop <= `FLASHOP_READ;

           end
             end
       endcase
     else
       fop <= `FLASHOP_IDLE;

  ////////////////////////////////////////////////////////////////////////////
  //
  // Status display
  //
  ////////////////////////////////////////////////////////////////////////////
  //
  // "Reset     ------" --> During reset
  // "Read ID   000000" --> While reading ID codes
  // "Clr locks 000000" --> While clearing block locks
  // "Erase     000000" --> While erasing
  // "Write     000000" --> While writing
  // "Read      000000" --> While reading
  // " *** PASSED *** " --> If the entire test completes with no errors
  // "Err: Manuf  0000" --> If an incorrect manufacturer code is read
  // "Err: Size   0000" --> If an incorrect size code is read
  // "Err: Locks  0000" --> If an error is detected when clearing the block lock bit
  // "Err: Erase  0000"
  // "Err: 000000 0000"
```

```
// Rd  000000  0000
// Wr  000000  0000
// Id  000000  0000
//

function [39:0] nib2char;
   input [3:0] nib;
   begin
  case (nib)
    4'h0: nib2char = 40'b00111110_01010001_01001001_01000101_00111110;
    4'h1: nib2char = 40'b00000000_01000010_01111111_01000000_00000000;
    4'h2: nib2char = 40'b01100010_01010001_01001001_01001001_01000110;
    4'h3: nib2char = 40'b00100010_01000001_01001001_01001001_00110110;
    4'h4: nib2char = 40'b00011000_00010100_00010010_01111111_00010000;
    4'h5: nib2char = 40'b00100111_01000101_01000101_01000101_00111001;
    4'h6: nib2char = 40'b00111100_01001010_01001001_01001001_00110000;
    4'h7: nib2char = 40'b00000001_01110001_00001001_00000101_00000011;
    4'h8: nib2char = 40'b00110110_01001001_01001001_01001001_00110110;
    4'h9: nib2char = 40'b00000110_01001001_01001001_00101001_00011110;
    4'hA: nib2char = 40'b01111110_00001001_00001001_00001001_01111110;
    4'hB: nib2char = 40'b01111111_01001001_01001001_01001001_00110110;
    4'hC: nib2char = 40'b00111110_01000001_01000001_01000001_00100010;
    4'hD: nib2char = 40'b01111111_01000001_01000001_01000001_00111110;
    4'hE: nib2char = 40'b01111111_01001001_01001001_01001001_01000001;
    4'hF: nib2char = 40'b01111111_00001001_00001001_00001001_00000001;
  endcase
   end
endfunction

wire [159:0] data_dots;
assign data_dots = {nib2char(frdata[15:12]), nib2char(frdata[11:8]),
            nib2char(frdata[7:4]), nib2char(frdata[3:0])};

wire [239:0] address_dots;
assign address_dots = {nib2char({ 1'b0, faddress[22:20]}),
           nib2char(faddress[19:16]),
           nib2char(faddress[15:12]),
           nib2char(faddress[11:8]),
           nib2char(faddress[7:4]),
           nib2char(faddress[3:0])};

always @(status or address_dots or data_dots)
  case (status)
    `STATUS_RESET:
  dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
```

```
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b00100110_01001001_01001001_01001001_00110010, // S
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b00000001_00000001_01111111_00000001_00000001, // T
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000};// -
  `STATUS_READ_ID:
dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b01111110_00001001_00001001_00001001_01111110, // A
     40'b01111111_01000001_01000001_01000001_00111110, // D
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_01000001_01111111_01000001_00000000, // I
     40'b01111111_01000001_01000001_01000001_00111110, // D
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     address_dots};
  `STATUS_CLEAR_LOCKS:
dots <= {40'b00111110_01000001_01000001_01000001_00100010, // C
     40'b01111111_01000000_01000000_01000000_01000000, // L
     40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b01111111_01000000_01000000_01000000_01000000, // L
     40'b00111110_01000001_01000001_01000001_00111110, // O
     40'b00111110_01000001_01000001_01000001_00100010, // C
     40'b01111111_00001000_00010100_00100010_01000001, // K
     40'b00100110_01001001_01001001_01001001_00110010, // S
     40'b00000000_00000000_00000000_00000000_00000000, //
     address_dots};
  `STATUS_ERASING:
dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b01111110_00001001_00001001_00001001_01111110, // A
     40'b00100110_01001001_01001001_01001001_00110010, // S
     40'b00000000_01000001_01111111_01000001_00000000, // I
```

```
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b00111110_01000001_01001001_01001001_00111010, // G
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
    `STATUS_WRITING:
dots <= {40'b01111111_00100000_00011000_00100000_01111111, // W
          40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b00000001_00000001_01111111_00000001_00000001, // T
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b00111110_01000001_01001001_01001001_00111010, // G
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
    `STATUS_READING:
dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b01111110_00001001_00001001_00001001_01111110, // A
          40'b01111111_01000001_01000001_01000001_00111110, // D
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b00111110_01000001_01001001_01001001_00111010, // G
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
    `STATUS_SUCCESS:
dots <= {40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b01111111_00001001_00001001_00001001_00000110, // P
          40'b01111110_00001001_00001001_00001001_01111110, // A
          40'b00100110_01001001_01001001_01001001_00110010, // S
          40'b00100110_01001001_01001001_01001001_00110010, // S
          40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b01111111_01000001_01000001_01000001_00111110, // D
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40'b00101010_00011100_01111111_00011100_00101010, // *
          40’b00101010_00011100_01111111_00011100_00101010, // *
          40’b00000000_00000000_00000000_00000000_00000000};//
      ‘STATUS_BAD_MANUFACTURER:
   dots <= {40’b01111111_01001001_01001001_01001001_01000001, // E
          40’b01111111_00001001_00011001_00101001_01000110, // R
          40’b01111111_00001001_00011001_00101001_01000110, // R
          40’b00000000_00110110_00110110_00000000_00000000, // :
          40’b00000000_00000000_00000000_00000000_00000000, //
          40’b01111111_00000010_00001100_00000010_01111111, // M
          40’b01111110_00001001_00001001_00001001_01111110, // A
          40’b01111111_00000010_00000100_00001000_01111111, // N
          40’b01111111_00001001_00001001_00001001_00000001, // U
          40’b01111111_00001001_00001001_00001001_00000001, // F
          40’b00000000_00000000_00000000_00000000_00000000, //
          40’b00000000_00000000_00000000_00000000_00000000, //
          data_dots};
      ‘STATUS_BAD_SIZE:
   dots <= {40’b01111111_01001001_01001001_01001001_01000001, // E
          40’b01111111_00001001_00011001_00101001_01000110, // R
          40’b01111111_00001001_00011001_00101001_01000110, // R
          40’b00000000_00110110_00110110_00000000_00000000, // :
          40’b00000000_00000000_00000000_00000000_00000000, //
          40’b00100110_01001001_01001001_01001001_00110010, // S
          40’b00000000_01000001_01111111_01000001_00000000, // I
          40’b01100001_01010001_01001001_01000101_01000011, // Z
          40’b01111111_01001001_01001001_01001001_01000001, // E
          40’b00000000_00000000_00000000_00000000_00000000,
          40’b00000000_00000000_00000000_00000000_00000000,
          40’b00000000_00000000_00000000_00000000_00000000,
          data_dots};
      ‘STATUS_LOCK_BIT_ERROR:
   dots <= {40’b01111111_01001001_01001001_01001001_01000001, // E
          40’b01111111_00001001_00011001_00101001_01000110, // R
          40’b01111111_00001001_00011001_00101001_01000110, // R
          40’b00000000_00110110_00110110_00000000_00000000, // :
          40’b00000000_00000000_00000000_00000000_00000000, //
          40’b01111111_01000000_01000000_01000000_01000000, // L
          40’b00111110_01000001_01000001_01000001_00111110, // O
          40’b00111110_01000001_01000001_01000001_00100010, // C
          40’b01111111_00001000_00010100_00100010_01000001, // K
          40’b00100110_01001001_01001001_01001001_00110010, // S
          40’b00000000_00000000_00000000_00000000_00000000,
          40’b00000000_00000000_00000000_00000000_00000000,
          data_dots};
      ‘STATUS_ERASE_BLOCK_ERROR:
```

```
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
         40'b01111111_00001001_00011001_00101001_01000110, // R
         40'b01111111_00001001_00011001_00101001_01000110, // R
         40'b00000000_00110110_00110110_00000000_00000000, // :
         40'b00000000_00000000_00000000_00000000_00000000, //
         40'b01111111_01001001_01001001_01001001_01000001, // E
         40'b01111111_00001001_00011001_00101001_01000110, // R
         40'b01111110_00001001_00001001_00001001_01111110, // A
         40'b00100110_01001001_01001001_01001001_00110010, // S
         40'b01111111_01001001_01001001_01001001_01000001, // E
         40'b00000000_00000000_00000000_00000000_00000000,
         40'b00000000_00000000_00000000_00000000_00000000,
         data_dots};
      `STATUS_WRITE_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
         40'b01111111_00001001_00011001_00101001_01000110, // R
         40'b01111111_00001001_00011001_00101001_01000110, // R
         40'b00000000_00110110_00110110_00000000_00000000, // :
         40'b00000000_00000000_00000000_00000000_00000000, //
         40'b01111111_00100000_00011000_00100000_01111111, // W
         40'b01111111_00001001_00011001_00101001_01000110, // R
         40'b00000000_01000001_01111111_01000001_00000000, // I
         40'b00000001_00000001_01111111_00000001_00000001, // T
         40'b01111111_01001001_01001001_01001001_01000001, // E
         40'b00000000_00000000_00000000_00000000_00000000,
         40'b00000000_00000000_00000000_00000000_00000000,
         data_dots};
      `STATUS_READ_WRONG_DATA:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
         40'b01111111_00001001_00011001_00101001_01000110, // R
         40'b01111111_00001001_00011001_00101001_01000110, // R
         40'b00000000_00110110_00110110_00000000_00000000, // :
         40'b00000000_00000000_00000000_00000000_00000000,
         address_dots,
         40'b00000000_00000000_00000000_00000000_00000000,
         data_dots};
    default:
    dots <= {16{40'b01010101_00101010_01010101_00101010_01010101}};
    endcase

endmodule
```

A.2.12 top_level.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
// Revised by: Jonathan Burnham, Kevin Miu, and Christopher Hoffman
//         for use in their 6.111 final project. Combines all video,
//         audio, and control modules to implement the Full Motion
//         Dance Machine. This project allows a user to wear a color
//         coded shirt which the machine will recognize and draw a
//         stickperson of the user on the screen. They may then dance
//         to a song by following dance queues which scroll across the
//         screen on synch with the beat of the song. They may also get
//         extra points by moving along with the beat of the song.
//
////////////////////////////////////////////////////////////////////////////////
//
// 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 top_level(beep, audio_reset_b,
               ac97_sdata_out, ac97_sdata_in, ac97_synch,
           ac97_bit_clock,

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

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

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

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

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

           clock_feedback_out, clock_feedback_in,

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

           rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

           mouse_clock, mouse_data, keyboard_clock, keyboard_data,

           clock_27mhz, clock1, clock2,

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

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

           switch,

           led,

           user1, user2, user3, user4,

           daughtercard,

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

           analyzer1_data, analyzer1_clock,
           analyzer2_data, analyzer2_clock,
           analyzer3_data, analyzer3_clock,
           analyzer4_data, analyzer4_clock);

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

// Audio Input and Output
assign beep= 1'b0;
//assign audio_reset_b = 1'b0;
//assign ac97_synch = 1'b0;
//assign ac97_sdata_out = 1'b0;
//ac97_sdata_in is an input

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

// Video Input
//assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b1;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b1;
//assign tv_in_reset_b = 1'b0;
```

```
   assign tv_in_clock = clock_27mhz;//1'b0;
   //assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs

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

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

/* enable RAM pins */

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

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

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

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

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

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

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

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

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs
   // Logic Analyzer
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

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

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

    wire clk = clock_65mhz;


    wire clock_130mhz_unbuf,clock_130mhz;
    DCM vclk3(.CLKIN(clock_27mhz),.CLKFX(clock_130mhz_unbuf));
    // synthesis attribute CLKFX_DIVIDE of vclk3 is 2
    // synthesis attribute CLKFX_MULTIPLY of vclk3 is 24
    // synthesis attribute CLK_FEEDBACK of vclk3 is NONE
    // synthesis attribute CLKIN_PERIOD of vclk3 is 37
    BUFG vclk4(.O(clock_130mhz),.I(clock_130mhz_unbuf));

    // 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,b_enter;
    debounce dbe(power_on_reset, clock_65mhz, ~button_enter, b_enter);
    assign reset = power_on_reset;
```

```
// Debounce all of our buttons and switches for later use
wire s0;
debounce ds0(reset, clock_65mhz, switch[0], s0);
wire s1;
debounce ds1(reset, clock_65mhz, switch[1], s1);
wire s2;
debounce ds2(reset, clock_65mhz, switch[2], s2);
wire s3;
debounce ds3(reset, clock_65mhz, switch[3], s3);
wire s4;
debounce ds4(reset, clock_65mhz, switch[4], s4);
wire s5;
debounce ds5(reset, clock_65mhz, switch[5], s5);
wire s6;
debounce ds6(reset, clock_65mhz, switch[6], s6);
wire s7;
debounce ds7(reset, clock_65mhz, switch[7], s7);

wire button_0;
debounce db0(reset, clock_65mhz, ~button0, button_0);

wire button_1;
debounce db1(reset, clock_65mhz, ~button1, button_1);

wire button_2;
debounce db2(reset, clock_65mhz, ~button2, button_2);

wire button_3;
debounce db3(reset, clock_65mhz, ~button3, button_3);

wire up;
debounce dbu(reset, clock_65mhz, ~button_up, up);

wire left;
debounce dbleft(reset, clock_65mhz, ~button_left, left);

wire right;
debounce dbright(reset, clock_65mhz, ~button_right, right);

wire down;
debounce dbdown(reset, clock_65mhz, ~button_down, down);


// Set up module to display data to the LED display bank on the
// labkit
```

```
    wire [63:0] dispdata;
    display_16hex hexdisp1(reset, clock_65mhz, dispdata,
                   disp_blank, disp_clock, disp_rs, disp_ce_b,
                   disp_reset_b, disp_data_out);


////////////////////////////////////////////////////////////////////////////////
/////
/////           Video Code
/////
////////////////////////////////////////////////////////////////////////////////

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

    // wire up to ZBT ram
    wire [35:0] vram_write_data;
    wire [35:0] vram_read_data;
    wire [18:0] vram_addr;
    wire vram_we;

    zbt_6111 zbt1(clock_65mhz, 1'b1, vram_we, vram_addr,
              vram_write_data, vram_read_data,
              ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

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

    vram_display vd1(reset,clock_65mhz,hcount,vcount,vr_pixel,
                 vram_addr1,vram_read_data);

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

    wire [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[7:0] R,G,B;
YCrCb2RGB YCrCb2RGB1(R, G, B, tv_in_line_clock1, reset, ycrcb[29:20], ycrcb[19:10

wire [18:0] ntsc_addr;
wire [35:0] ntsc_data;
wire        ntsc_we;
ntsc_to_zbt n2z (clock_65mhz, tv_in_line_clock1, fvh, dv, {R,G,B},
             ntsc_addr, ntsc_data, ntsc_we, 0);

// code to write pattern to ZBT memory
reg [31:0]  count;
always @(posedge clock_65mhz) count <= reset ? 0 : count + 1;

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

// mux selecting read/write to memory based on which write-enable is chosen
wire    sw_ntsc = 1;
wire    my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
wire [18:0]     write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
wire [35:0]     write_data = sw_ntsc ? ntsc_data : vpat;

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

// select output pixel data
reg [17:0] pixel;
wire b,hs,vs;

delayN dn1(clock_65mhz,hsync,hs);   // delay by 3 cycles to sync with ZBT read
delayN dn2(clock_65mhz,vsync,vs);
delayN dn3(clock_65mhz,blank,b);

always @(posedge clock_65mhz) pixel <= 0 ? {hcount[8:6],5'b0} : vr_pixel;

// Video Wires
wire [17:0] cleaned_pixel;
```

```
wire[28:0] adder;
wire[27:0] adder2;
wire[18:0] howmany;
wire[18:0] remd;
wire rfd;
wire[18:0] remd2;
wire rfd2;
wire ce=(vcount>768||hcount<1023)?1:0;
wire[28:0] quot;
wire[27:0] quot2;
wire[8:0] dout;
wire[2:0] state;
wire [26:0] counter;
wire sixteenth_count;
wire [10:0]xoffset;
sixteenth sixteenth1(reset,clock_27mhz,sixteenth_count,counter,xoffset_old);
wire xvel;
wire yvel;
wire xold;
wire yold;
wire [8:0] hspacing;
wire [10:0] gridcenterx;
wire [10:0] cmx[3:0];
wire [9:0] cmy[3:0];
wire [2:0]xlevel0;
wire [1:0]ylevel0;
wire [2:0]xlevel3;
wire [1:0]ylevel3;
wire [1:0] colorcount;
wire[4:0]enable_count;
wire beat_reset;
wire [8:0] data2;
wire [2:0] video_state;
wire default_calib;
wire good;
wire beat_enable_red;
wire beat_enable_teal;
wire user_beat;
wire vx0;
wire vy0;
wire [8:0]din;
wire we=0;
wire[17:0]addr;

// Computes center of masses and draws stickman and grid lines
```

```
    cm_calculator cm_calculator(clock_65mhz,pixel,hcount,vcount,vsync,cleaned_pixel,a
             quot2[9:0],dout,s0,s1,s2,s3,s4,s5,s6,s7,button_0,video_state,button_1,si
             button_2,default_calib,cmx[0],cmy[0],cmx[1],cmy[1],cmx[2],cmy[2],cmx[3],
             hspacing,gridcenterx,colorcount,data2,good);

    // Detects which grids the users hands are currently in
    grid_detect grid_detect1(clock_65mhz,vsync,cmx[0],cmy[0],cmx[1],cmy[1],cmx[2],cmy

    // Detects the beats of the user
    beat_detect beat_detect(clk,cmx[0],cmy[0],cmx[3],cmy[3],colorcount,beat_reset,bea

    // Dividers for center of mass calculations
    divider28 divider1(adder,howmany,quot,remd,clock_130mhz,rfd,aclr,sclr,ce);
    divider27 divider2(adder2,howmany,quot2,remd2,clock_130mhz,rfd2,aclr,sclr,ce);


    assign user_beat = beat_enable_red | beat_enable_teal;



////////////////////////////////////////////////////////////////////////////////
/////
/////           End of Video Code
/////
////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////
/////
/////           Audio Code
/////
////////////////////////////////////////////////////////////////////////////////

    // Wires for flash memory
    wire [1:0] fop;
    wire [22:0] faddress;
    wire [15:0] fwdata, frdata;
    wire fbusy;


    // Wires for audio
    wire [7:0] from_ac97_data, to_ac97_data;
    wire ready;
    wire beat_1cyc;
wire enable_16;
wire enable;
wire beat_to_mem;
wire beat_from_mem_a, beat_from_mem_b;
wire[11:0] beat_address_in, beat_address_out;
wire beat_we;
wire start;
wire stop;
wire pb_start, pb_pause, pb_stop, address_reset;
wire [22:0] new_address, stop_address;
wire flash_ready;
wire new_ready;
reg old_pb_start;
wire reset_play;
wire beat_address_control;
wire [11:0] beat_address_a, beat_address_b;
wire [11:0] score_address;
wire beat_out;
wire [13:0] score_beat, score_grid, score_total;
wire done_scrolling;
wire red_right, teal_right;
wire[7:0] first_half, second_half;
wire start_song;
wire clock_16hz;
wire[1:0] state_audio;
wire[63:0] led_out;
wire beat;
wire[5:0] leds;
wire volume;
wire [639:0] dots;

// Assign full volume
assign volume = 5'd31;

// Assign addresses where song resides
assign stop_address = 23'd4320000;
assign new_address = 23'd0;

// This way port a can be used for both storing and reading
// out the beat
assign beat_address_a = beat_address_out | beat_address_in;

// Sum up the scores
assign score_total = score_beat + score_grid;
```

```
// AC97 driver
audio a(clock_27mhz, reset, 5'd24, from_ac97_data, frdata[7:0], ready,
    audio_reset_b, ac97_sdata_out, ac97_sdata_in,
    ac97_synch, ac97_bit_clock);

// Flash Memory
flash_int flashint1 (reset, clock_27mhz, fop, faddress, fwdata, frdata,
             fbusy, flash_data, flash_address, flash_ce_b,
             flash_oe_b, flash_we_b, flash_reset_b, 1'b1,
             flash_byte_b);

// FSM to control flash memory access
test_fsm fsm1 (reset, clock_27mhz, fop, faddress, fwdata, frdata, fbusy,
           dots, flash_ready, dout_audio, new_address, stop_address, reset_play);

// Does the beat detection on the data running through it
beat_process beat1(new_ready, frdata[7:0], beat, leds, 36, beat_1cyc);

// Controls storing the beat in the beat memory during beat detection
beat_memory beat_store(clock_27mhz, enable_16, new_ready, start, beat_1cyc, beat_
               beat_address_in, beat_we, stop, enable, 4'd0, faddress, stop_addre

// Beat memory, dual port for simultaneous reads: one for the current beat to be
// beat comparison, the other by scroll_control so it can look ahead in memory
beat_bram my_beats(beat_address_a, beat_address_b, clock_27mhz, clock_27mhz,
           beat_to_mem, beat_from_mem_a, beat_from_mem_b, beat_we);

// Controls flash clock speed to do high speed beat detection or normal speed pla
high_speed_enable my_ready(clock_27mhz, 0, enable, new_ready);

// Used for timing by beat_memory
clock_16th my_16th_clock(new_ready, enable_16);

// Need another for normal playback speed
clock_16th my_16hz(ready, clock_16hz);

// Another speed control for normal playback
high_speed_enable my_flash_ready(clock_27mhz, ready, enable, flash_ready);

// Controls beat detection and playback
storage_tester my_tester(stop, clock_27mhz, clock_16hz, start, beat_address_out,
        pb_start, pb_pause, pb_stop,reset_play, stop_address, faddress, start_son

// Assign LED banks useful output data
```

```
assign led[7:0] = {~beat_to_mem,~beat_reset,~red_right, ~teal_right,~user_beat,~b
assign dispdata = led_out;
////////////////////////////////////////////////////////////////////////////////
/////
/////           End of Audio Code
/////
////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////
/////
/////           Interface and Scoring Code
/////
////////////////////////////////////////////////////////////////////////////////


wire [23:0] pixelsArrow, pixels_interf, pixelsRectangle;
wire [10:0] arrow_x, rec_x;
wire [9:0] arrow_y, rec_y;
wire [1:0] difficulty;
wire finished;
wire start_scrolling;
wire new_pic;


// Controls the menu interface and directs audio playback and beat detection, sco
// in a meaningful manner
interface interface(clock_65mhz, button_3, hcount, vcount, arrow_x, arrow_y, rec_
              rec_y, up, down, left, right, b_enter, button_0, cleaned_pixel,
              pixels_interf, start_song, state_audio, video_state, default_calib,
              difficulty, score_beat, score_grid, score_total, led_out);


// Looks ahead in the beat memory to determine when to start scrolling the next d
scroll_control scroll_control(clock_27mhz, beat_address_out, done_scrolling,
                    beat_from_mem_b, beat_address_b, start_scrolling, new_pic);

// Tracks where the picture needs to be once scrolling has begun
begin_scrolling begin_scrolling(clock_27mhz,start_scrolling,19'b0010000000000000

// Calculates where in the picture memory (left and right commands)
// the program should be looking in order to get the pixel
// for that particular hcount, vcount, and offset
addypicker addypicker(clock_65mhz,new_pic,hcount,vcount,addr,xoffset, first_half,
    // Compares the user and song beats and calculates a score
    compare_beats compare_beats(clock_27mhz, start_song, beat_from_mem_a, user_beat,

    // Compares the users grid positions with the picture grid
    // positions and computes a score if they are matching the picture correctly
    grid_score grid_score(clock_65mhz, start_song, xlevel0, ylevel0, xlevel3, ylevel3
                    first_half, second_half, done_scrolling, score_grid, red_right, t

    // Memories containing the scrolling dance pictures
    left_commands pic1(addr,clk,din,dout,we);
    right_commands pic2(addr,clk,din,data2,we);


    ///////////////////////////////////////////////////////////////////////////////////
    /////
    /////           End of Interface Code
    /////
    ///////////////////////////////////////////////////////////////////////////////////


    // Send video signals to proper destinations
    assign vga_out_red = pixels_interf[23:16];
    assign vga_out_green = pixels_interf[15:8];
    assign vga_out_blue = pixels_interf[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;

endmodule
```

A.2.13 video_decoder.v

```
//
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
// NOT WRITTEN BY KEVIN, JON, OR CHRIS
//
// 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);
   // INPUTS
   //   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)
   // OUTPUTS
   //   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
   // 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 st
         // whenever the synchronization code is seen, go back to the sync_state befo
         // 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
```

A.2.14 vram_display.v

```
//
// File:   vram_display.v
// Modified by Kevin
//
// vram_display latches the pixels at certain clock cycles to prepare for output
// It generates 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; and we use all 36  bits
// decoded into two bytes of pixel data.
//
// The modifications in the module involve inverting the hcount value such
// that the image is displayed correctly on the screen.  Without the inversion
// moving one way relative to the camera produces a movement in the opposite
// direction on the screen.
//
// Another modification was just practicing the shifting of the image data using
// the vram_addr
//
// hc4 was reduced to a single bit, and the latching was done off this single bit
// value.  Each word is decoded into two sets of 2-bytes each, since 18-bits of
// color content per pixel were stored in the ZBT.
module vram_display(reset,clk,hcount,vcount,vr_pixel,
            vram_addr,vram_read_data);

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

   wire [18:0]   vram_addr = {vcount[9:1]+9'd45,1'b0,~hcount[9:1]+9'd13};
   wire hc4 = hcount[0];
   reg [17:0]    vr_pixel;
   reg [35:0]    vr_data_latched;
   reg [35:0]    last_vr_data;

//move data on opposite cycle
   always @(posedge clk)
     last_vr_data <= (hc4==2'd0) ? vr_data_latched : last_vr_data;

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

//decode words
   always @(*)      // each 36-bit word from RAM is decoded to 2 sets of
                 // 2-bytes each
     case (hc4)
       2'd1: vr_pixel = last_vr_data[17:0];
       2'd0: vr_pixel = last_vr_data[35:18];
     endcase

endmodule // vram_display
```

A.2.15 zbt_6111.v

```
//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
// NOT WRITTEN BY KEVIN, JON OR CHRIS
//
// 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.
// INPUTS
//   clk;           // system clock
//   cen;           // clock enable for gating ZBT cycles
//   we;            // write enable (active HIGH)
//   [18:0] addr;       // memory address
//   [35:0] write_data; // data to write
//
// INPUT/OUTPUT
//   [35:0]  ram_data;  // physical line to ram data
//
// OUTPUTS
//   [35:0] read_data;  // data read from memory
//   ram_clk;   // physical line to ram clock
//   ram_we_b;  // physical line to ram we_b
//   [18:0] ram_address;    // physical line to ram address
//   ram_cen_b; // physical line to ram clock enable

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
```

A.3 USB to Flash Writer

A.3.1 display_16hex.v

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

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

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

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

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

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

```
      end

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

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

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

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

    always @(posedge clock_27mhz)
      if (clock_tick)
        begin
     if (dreset)
       begin
          state <= 0;
          dot_index <= 0;
          control <= 32'h7F7F7F7F;
       end
     else
       casex (state)
         8'h00:
       begin
          // Reset displays
          disp_data_out <= 1'b0;
          disp_rs <= 1'b0; // dot register
          disp_ce_b <= 1'b1;
          disp_reset_b <= 1'b0;
          dot_index <= 0;
          state <= state+1;
       end
```

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

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

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

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

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

```
        data <= data_in;
        state <= state+1;
     end

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

     endcase // casex(state)
    end

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

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

A.3.2 flash_int.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash ROM Interface
//
// For Labkit Revision 004
//
//
// Created: January 22, 2005
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

`define FLASHOP_IDLE  2'b00 `define FLASHOP_READ  2'b01 `define
FLASHOP_WRITE 2'b10

module flash_int(reset, clock, op, address, wdata, rdata, busy,
flash_data,
         flash_address, flash_ce_b, flash_oe_b, flash_we_b,
         flash_reset_b, flash_sts, flash_byte_b,enable);
```

```
parameter access_cycles = 5;
parameter reset_assert_cycles = 1000;
parameter reset_recovery_cycles = 30;

input reset, clock; // Reset and clock for the flash interface
input [1:0] op; // Flash operation select (read, write, idle)
input [22:0] address;
input [15:0] wdata;
output [15:0] rdata;
output busy;
inout [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b;
output flash_reset_b, flash_byte_b;
input  flash_sts;
input enable;
reg [1:0] lop;
reg [15:0] rdata;
reg busy;
reg [15:0] flash_wdata;
reg flash_ddata;
reg [23:0] flash_address;
reg flash_oe_b, flash_we_b, flash_reset_b;

assign flash_ce_b = flash_oe_b && flash_we_b;
assign flash_byte_b = 1; // 1 = 16-bit mode (A0 ignored)

assign flash_data = flash_ddata ? flash_wdata : 16'hZ;

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

initial
  flash_reset_b <= 1'b1;

reg [9:0] state;

always @(posedge clock)
  if (reset)
    begin
   state <= 0;
   flash_reset_b <= 0;
```

```
 flash_we_b <= 1;
 flash_oe_b <= 1;
 flash_ddata <= 0;
 busy <= 1;
  end
else if (flash_reset_b == 0)
  if (state == reset_assert_cycles)
begin
   flash_reset_b <= 1;
   state <= 1023-reset_recovery_cycles;
end
  else
state <= state+1;
else if ((state == 0) && !busy)
  // The flash chip and this state machine are both idle. Latch the user's
  // address and write data inputs. Deassert OE and WE, and stop driving
  // the data buss ourselves. If a flash operation (read or write) is
  // requested, move to the next state.
  begin
 flash_address <= {address, 1'b0};
 flash_we_b <= 1;
 flash_oe_b <= 1;
 flash_ddata <= 0;
 flash_wdata <= wdata;
 lop <= op;
 if (op != `FLASHOP_IDLE)
   begin
      busy <= 1;
      state <= state+1;
   end
 else
   busy <= 0;
  end
else if ((state==0) && flash_sts)
  busy <= 0;
else if (state == 1)
  // The first stage of a flash operation. The address bus is already set,
  // so, if this is a read, we assert OE. For a write, we start driving
  // the user's data onto the flash databus (the value was latched in the
  // previous state.
  begin
 if (lop == `FLASHOP_WRITE)
   flash_ddata <= 1;
 else if (lop == `FLASHOP_READ)
   flash_oe_b <= 0;
```

```
      state <= state+1;
       end
     else if (state == 2)
       // The second stage of a flash operation. Nothing to do for a read. For
       // a write, we assert WE.
       begin
      if (lop == `FLASHOP_WRITE)
        flash_we_b <= 0;
      state <= state+1;
       end
     else if (state == access_cycles+1)
       // The third stage of a flash operation. For a read, we latch the data
       // from the flash chip. For a write, we deassert WE.
       begin
      if (lop == `FLASHOP_WRITE)
        flash_we_b <= 1;
      if (lop == `FLASHOP_READ)
        rdata <= flash_data;
      state <= 0;
       end
     else
       begin
      if (!flash_sts)
        busy <= 1;
      state <= state+1;
       end

endmodule
```

A.3.3 flashwriter.v

```
///////////////////////////////////////////////////////////////////////////////
//
// Switch Debounce Module
//
///////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock, noisy, clean);

   input reset, clock, noisy;
   output clean;

   reg [18:0] count;
   reg new, clean;
   always @(posedge clock)
     if (reset)
       begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
       end
     else if (noisy != new)
       begin
      new <= noisy;
      count <= 0;
       end
     else if (count == 270000)
       clean <= new;
     else
       count <= count+1;

endmodule

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

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

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

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

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

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

reg audio_reset_b;
reg [9:0] reset_count;

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

wire ac97_ready;
ac97 ac97(ac97_ready, command_address, command_data, command_valid,
          left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
          right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
          ac97_bit_clock);

// generate two pulses synchronous with the clock: first capture, then ready
reg [2:0] ready_sync;
always @ (posedge clock_27mhz) begin
  ready_sync <= {ready_sync[1:0], ac97_ready};
end
assign ready = ready_sync[1] & ~ready_sync[2];

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

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

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

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

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

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

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

   initial begin
      ready <= 1'b0;
      // synthesis attribute init of ready is "0";
      ac97_sdata_out <= 1'b0;
      // synthesis attribute init of ac97_sdata_out is "0";
      ac97_synch <= 1'b0;
      // synthesis attribute init of ac97_synch is "0";
      bit_count <= 8'h00;
      // synthesis attribute init of bit_count is "0000";
      l_cmd_v <= 1'b0;
      // synthesis attribute init of l_cmd_v is "0";
      l_left_v <= 1'b0;
      // synthesis attribute init of l_left_v is "0";
      l_right_v <= 1'b0;
      // synthesis attribute init of l_right_v is "0";

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

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

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

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

      if ((bit_count >= 0) && (bit_count <= 15))
        // Slot 0: Tags
        case (bit_count[3:0])
          4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
          4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
          4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
          4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
      4'h4: ac97_sdata_out <= l_right_v; // Right data valid
          default: ac97_sdata_out <= 1'b0;
        endcase

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

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

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

      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

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

endmodule

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

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

reg [23:0] command;
reg command_valid;

reg [3:0] state;

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

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

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

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

   case (state)
     4'h0: // Read ID
       begin
          command <= 24'h80_0000;
          command_valid <= 1'b1;
       end
     4'h1: // Read ID
       command <= 24'h80_0000;
     4'h3: // headphone volume
       command <= { 8'h04, 3'b000, vol, 3'b000, vol };
     4'h5: // PCM volume
       command <= 24'h18_0808;
     4'h6: // Record source select
```

```
          command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
        4'h7: // Record gain = max
      command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
          command <= 24'h0E_8048;
        4'hA: // Set beep volume
          command <= 24'h0A_0000;
        4'hB: // PCM out bypass mix1
          command <= 24'h20_8000;
        default:
          command <= 24'h80_0000;
      endcase // case(state)
   end // always @ (posedge clock)
endmodule // ac97commands


///////////////////////////////////////////////////////////////////////////////
//
// 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 usbtoflash   (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
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Reset Generation
//
// A shift register primitive is used to generate an active-high reset
// signal that remains high for 16 clock cycles after configuration finishes
// and the FPGA's internal clocks begin toggling.
//
////////////////////////////////////////////////////////////////////////////

wire reset;
SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(reset),
        .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// Audio Input and Output
assign beep= 1'b0;
//lab3 assign audio_reset_b = 1'b0;
//lab3 assign ac97_synch = 1'b0;
//lab3 assign ac97_sdata_out = 1'b0;
```

```
// ac97_sdata_in is an input

// VGA Output
assign vga_out_red = 10'h0;
assign vga_out_green = 10'h0;
assign vga_out_blue = 10'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;

// 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'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 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
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
```

```
   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;
   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input
/*
   // Flash ROM
   assign flash_data = 16'hZ;
   assign flash_address = 24'h0;
   assign flash_ce_b = 1'b1;
   assign flash_oe_b = 1'b1;
   assign flash_we_b = 1'b1;
   assign flash_reset_b = 1'b0;
   assign flash_byte_b = 1'b1;
   // flash_sts is an input
*/
   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs
/*
   // LED Displays
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
   // disp_data_in is an input
*/
   // Buttons, Switches, and Individual LEDs
   //lab3 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
```

```
//assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

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

// Logic Analyzer
//lab3 assign analyzer1_data = 16'h0;
//lab3 assign analyzer1_clock = 1'b1;
//lab3 assign analyzer2_data = 16'h0;
//lab3 assign analyzer2_clock = 1'b1;


wire [7:0] from_ac97_data, to_ac97_data;
wire ready;
 reg ready_reg;
 reg ready_regold;
 reg double_ready;
 always @ (posedge clock_27mhz) begin
     ready_reg <= ready;
     ready_regold <= ready_reg;
     end
     always @ (posedge clock_27mhz) begin
         if (ready_reg == !ready_regold) double_ready = 1;
         else double_ready = 0;
         end
     wire ready_27;
     assign ready_27 = double_ready;

// allow user to adjust volume
wire vup,vdown;
reg old_vup,old_vdown;
debounce bup(reset, clock_27mhz, ~button_up, vup);
debounce bdown(reset, clock_27mhz, ~button_down, vdown);
reg [4:0] volume;
```

```
   always @ (posedge clock_27mhz) begin
     if (reset) volume <= 5'd8;
     else begin
    if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
    if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
     end
     old_vup <= vup;
     old_vdown <= vdown;
   end

   // push ENTER button to record, release to playback
   wire playback;
   debounce benter(reset, clock_27mhz, button_enter, playback);

   // light up LEDs when recording, show volume during playback.
   // led is active low

   reg we;                   // write enable
   wire previous_state;       // previous state of recorder (playback or record)
   wire [15:0] addr;          // 16-bit address
   wire interpolate;          // interpolate setting

   assign interpolate=switch[0];  // interpolation is set on by turning switch 0 high
   wire [7:0] dout;               // 8-bit data out from the memory
   wire [7:0] s1;                 // 8-bit previous sample saved data
   wire [7:0] s2;                 // 8-bit saved data
   wire [2:0] count;              // 3-bit counter

////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- USB to Flash Writer
//
//
// Created: December 9, 2006
// Author: Kevin Miu
//
////////////////////////////////////////////////////////////////////////////////

   wire [1:0] fop;
   wire [22:0] faddress;
   wire [15:0] fwdata, frdata;
   wire fbusy;
   wire [639:0] dots;
   reg enable;
```

```
wire b0;
debounce db0(reset, clock_27mhz, ~button0, b0);

wire b1;
debounce db1(reset, clock_27mhz, ~button1, b1);

wire b2;
debounce db2(reset, clock_27mhz, ~button2, b2);

wire bleft;
debounce dbleft(reset, clock_27mhz, ~button_left, bleft);

reg readnotready; //ready to read data from USB
reg usbready;   //write from USB to BRAM signal
reg flashready;//write from BRAM to Flash ROM signal
reg eraseready;//erase 1024x64 block in Flash ROM signal
reg[16:0] address;    //BRAM address
reg[16:0] tempaddress; //counter address for making sure just enough
                                 //samples are read
wire [7:0] din;  //BRAM data in
 reg[3:0]count1; //counter to delay reading from the USB adapter
                         //USB adapter is 6mhz, so 16 counts affords it plenty
                         //of time


assign user1[8]=~readnotready; //user1[8] pin is connect to the read port of the U
 assign din=user1[7:0]; //the data in comes from the user1 lowest 8 bits, which ar
                                 //the 8 data pins of the USB adapter

 always@(posedge clock_27mhz)
 begin
     count1<=count1+1;   //delay counter to time with USB adapter
     if(count1==15)
         begin
             if(b0)
                 usbready<=1;    //the data has been sent from the computer
                                     //and the USB data is ready to be loaded onto
                                     //the BRAM
             if(b1)
                 flashready<=1;  //the USB data has been loaded onto the BRAM
                                     //and it is ready to load onto the Flash ROM
             if(b2)
                 begin
                     eraseready<=0;  //turn off erase signal
                     usbready<=0;    //turn off USB to BRAM signal
```

```
                    flashready<=0;  //turn off BRAM to Flash ROM signal
                    tempaddress<=0;//reset the tempaddress
                    address<=0;     //reset the address
                end
            if(bleft)
                eraseready<=1;      //erase mode
            if(tempaddress<32768)   //tempaddress increments up to 32768, which i
                        //the size of sample blocks loaded
                begin
                    if(usbready)  //if usb ready is high
                        begin
                            readnotready<=~readnotready;    //switch the read sig
                            if(readnotready)
                                begin
                                    tempaddress<=tempaddress+1;  //on the off sig
                                                                        //inc
                                    we<=1;                                //and set
                                                                        //the
                                end
                            else
                                begin
                                    we<=0;      //else don't write to the memory
                                                //is about to change (one sam
                                    address<=address+1; //but increment the USB t
                                end
                        end
                    else
                        readnotready<=1;    //if the user has not pressed the but
                end
        end
    if(flashready)
        begin
            address<=0; //set the USB to BRAM address to 0
            enable<=1;  //enable the Flash ROM
            we<=0;      //don't allow writing of the BRAM
        end
    else
        enable<=0;      //don't enable the Flash ROM
end

// AC97 driver
audio a(clock_27mhz, reset, volume, from_ac97_data, frdata[7:0], ready,
   audio_reset_b, ac97_sdata_out, ac97_sdata_in,
   ac97_synch, ac97_bit_clock);
wire [15:0] ramaddress;
```

```
    //The BRAM address relies on address, which is the USB to BRAM address, and
    //faddress which is the Flash ROM to BRAM address.  When one is nonzero, the othe
    //is zero, thus only one of those sources has access to the BRAM at a time.
    //The switch offset is to offset the fact that the Flash addresses are offset
    //using the switches while the ramaddresses are not.
    assign ramaddress=faddress+address-switch[7:0]*32768;
    music foo(ramaddress,clock_27mhz,din,dout,we);

   flash_int flashint1 (reset, clock_27mhz, fop, faddress, fwdata, frdata,
            fbusy, flash_data, flash_address, flash_ce_b,
            flash_oe_b, flash_we_b, flash_reset_b, 1'b1,
            flash_byte_b,enable);

    wire [3:0] status;
   test_fsm fsm1 (reset, clock_27mhz, fop, faddress, fwdata, frdata, fbusy,
          dots,ready_27,dout,enable,status,switch[7:0],b2,eraseready);


   display disp1 (reset, clock_27mhz, disp_blank, disp_clock, disp_rs,
            disp_ce_b, disp_reset_b, disp_data_out,dots);

    assign analyzer1_clock = clock_27mhz;
    assign analyzer2_clock = clock_27mhz;
    assign analyzer3_clock = clock_27mhz;
    assign analyzer4_clock = clock_27mhz;

   assign analyzer3_data = {9'b000000000,
                flash_address[22:16]};

    wire done;
    assign done=(status==6);
   assign analyzer4_data = flash_address[15:0];
    assign led = ~{enable,3'b000,status};

    assign analyzer1_data = {8'b0,dout};
    assign analyzer2_data = {address};

endmodule
```

A.3.4 test_fsm.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash Tester State Machine
//
//
// Created: January 23, 2005
// Author: Nathan Ickes
// Modified: December 2006 by Kevin Miu
// The new module takes in a starting address and clears the block
// (64x1024 piece of memory) associated with that block if the
// address is an even multiple of 32768.  If the block is an odd
// multiple of 32768, the following block is cleared.  The code no
// longer performs an additional erase after the write, allowing the
// Flash to be permanently written.  After the writing is done, it
// plays all the audio that has been written to the Flash to confirm
// that the Flash data was loaded correctly.
//
// INPUTS
//  reset - module reset signal
//  clock - 27 mHz clock
//  [15:0]  frdata - 16 bits of data read from the Flash
//  fbusy - Flash operation is busy
//  ready - Flash is ready
//  [7:0] bram_data - 8 bit sound data from BRAM
//  enable - enable writing to the Flash via user input
//  [7:0] piece_number - address selection
//  b2 - button 2 press
//  eraseready - signal to bypass normal writing operations and just
//                  erase the Flash address desired
//
// OUTPUTS
//  [1:0] fop - current Flash operation
//  [22:0] faddress - 23-bit Flash address
//  [15:0] fwdata - 16-bit data to write to the Flash
//  [639:0] dots - hex display dots
//  [3:0] status - current status for debugging error conditions
//
///////////////////////////////////////////////////////////////////////////////

‘define STATUS_RESET            4’h0 ‘define STATUS_READ_ID
4’h1 ‘define STATUS_CLEAR_LOCKS      4’h2 ‘define STATUS_ERASING
4’h3 ‘define STATUS_WRITING          4’h4 ‘define STATUS_READING
4’h5 ‘define STATUS_SUCCESS          4’h6 ‘define
```

```
STATUS_BAD_MANUFACTURER  4'h7 ‘define STATUS_BAD_SIZE          4'h8
‘define STATUS_LOCK_BIT_ERROR    4'h9 ‘define
STATUS_ERASE_BLOCK_ERROR 4'hA ‘define STATUS_WRITE_ERROR       4'hB
‘define STATUS_READ_WRONG_DATA   4'hC

‘define NUM_BLOCKS 128
//‘define BLOCK_SIZE 64*1024
‘define BLOCK_SIZE 59495 ‘define LAST_BLOCK_ADDRESS
((‘NUM_BLOCKS-1)*‘BLOCK_SIZE) ‘define LAST_ADDRESS
(‘NUM_BLOCKS*‘BLOCK_SIZE-1)

module test_fsm (reset, clock, fop, faddress, fwdata, frdata, fbusy,
dots, ready,bram_data,enable,status,piece_number,b2,eraseready);

   input reset, clock;
   output [1:0] fop;
   output [22:0] faddress;
   output [15:0] fwdata;
   input [15:0]  frdata;
   input fbusy;
   input ready;
   input [7:0] bram_data;
   input enable;
   input [7:0] piece_number;
   input b2, eraseready;

   output [639:0] dots;
   output [3:0] status;

   reg [1:0] fop; //Flash operation
   reg [22:0] faddress; //Flash address
   reg [15:0] fwdata; //data to be written to Flash
   reg [639:0] dots; //dots to be displayed on the hex display
   reg [7:0] state; //state of the FSM
   reg [3:0] status; //status of the Flash erasing/writing/reading
   reg [7:0] piece_offset; //address offset
   reg [22:0] start_erase; //erase address to start at

   ////////////////////////////////////////////////////////////////////////////
   //
   // State Machine
   //
   ////////////////////////////////////////////////////////////////////////////

   always @(posedge clock)
```

```
if (b2)
  begin
 state <= 0;
 status <= `STATUS_RESET;
 faddress <= piece_number*32768;
 fop <= `FLASHOP_IDLE;
  end
else if (!fbusy && (fop == `FLASHOP_IDLE))
  case (state)
8'h00:
  begin
       piece_offset<=piece_number-1;
     // Issue "read id codes" command
     status <= `STATUS_READ_ID;
     faddress <= piece_number*32768;
     fwdata <= 16'h0090;
     fop <= `FLASHOP_WRITE;
     state <= (enable)?state+1:state;
  end

8'h01:
  begin
     // Read manufacturer code
     faddress <= 0;
     fop <= `FLASHOP_READ;
     state <= state+1;
  end

8'h02:
  if (frdata != 16'h0089) // 16'h0089 = Intel
    status <= `STATUS_BAD_MANUFACTURER;
  else
    begin
   // Read the device size code
   faddress <= 1;
   fop <= `FLASHOP_READ;
   state <= state+1;
    end

8'h03:
  if (frdata != 16'h0018) // 16'h0018 = 128Mbit
    status <= `STATUS_BAD_SIZE;
  else
    begin
   faddress <= piece_number*32768;
```

```
     fwdata <= 16'hFF;
     fop <= `FLASHOP_WRITE;
     state <= state+1;
      end
  8'h04:
    begin
       // Issue "clear lock bits" command
       status <= `STATUS_CLEAR_LOCKS;
       faddress <= piece_number*32768;
       fwdata <= 16'h60;
       fop <= `FLASHOP_WRITE;
       state <= state+1;
    end
  8'h05:
    begin
       // Issue "confirm clear lock bits" command
       faddress <= piece_number*32768;
       fwdata <= 16'hD0;
       fop <= `FLASHOP_WRITE;
       state <= state+1;
    end
  8'h06:
    begin
       // Read status
       faddress <= piece_number*32768;
       fop <= `FLASHOP_READ;
       state <= state+1;
    end
  8'h07:
    if (frdata[7] == 1) // Done clearing lock bits
      if (frdata[6:1] == 0) // No errors
        begin
       faddress <= (piece_number==0)?0:(piece_offset[7:1]+1)*64*1024;
       start_erase<=(piece_number==0)?0:(piece_offset[7:1]+1)*64*1024;
       fop <= `FLASHOP_IDLE;
       state <= state+1;
        end
      else
        status <= `STATUS_LOCK_BIT_ERROR;
    else // Still busy, reread status register
      begin
     faddress <= piece_number*32768;
     fop <= `FLASHOP_READ;
      end
```

```
////////////////////////////////////////////////////////////////////////
// Block Erase Sequence
////////////////////////////////////////////////////////////////////////

8'h08:
  begin
     status <= `STATUS_ERASING;
     fwdata <= 16'h20; // Issue "erase block" command
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end
8'h09:
  begin
     fwdata <= 16'hD0; // Issue "confirm erase" command
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end
8'h0A:
  begin
     fop <= `FLASHOP_READ;
     state <= state+1;
  end
8'h0B:
  if (frdata[7] == 1) // Done erasing block
    if (frdata[6:1] == 0) // No errors
      if (faddress >= start_erase+64*1024-1) // `LAST_BLOCK_ADDRESS)
    begin
        faddress <= piece_number*32768;
       fop <= `FLASHOP_IDLE;
        if(eraseready)
        state<=8'h10;
        else
       state <= state+1;
    end
      else
    begin
       faddress <= faddress+`BLOCK_SIZE;
       fop <= `FLASHOP_IDLE;
       state <= state-3;
    end
    else // Erase error detected
      status <= `STATUS_ERASE_BLOCK_ERROR;
  else // Still busy
   begin
    fop <= `FLASHOP_READ;
```

```
     faddress <= faddress;
     end

//////////////////////////////////////////////////////////////////////
// Write Addresses to All Locations
//////////////////////////////////////////////////////////////////////

8'h0C:
  begin
     status <= `STATUS_WRITING;
     fwdata <= 16'h40; // Issue "setup write" command
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end
8'h0D:
  begin
     fwdata <= {8'hFF,dout}; // Finish write
     fop <= `FLASHOP_WRITE;
     state <= state+1;
  end
8'h0E:
  begin
     // Read status register
     fop <= `FLASHOP_READ;
     state <= state+1;
  end
8'h0F:
  if (frdata[7] == 1) // Done writing
    if (frdata[6:1] == 0) // No errors
      if (faddress != (piece_number+1)*32768) // `LAST_ADDRESS)
    begin
       faddress <= (faddress>=piece_number*32768)?faddress+1:piece_number*32768;
       fop <= `FLASHOP_IDLE;
       state <= state-3;
    end
      else
    begin
       faddress <= piece_number*32768;
       fop <= `FLASHOP_IDLE;
       state <= 8'h10;
    end
    else // Write error detected
      status <= `STATUS_WRITE_ERROR;
  else // Still busy
    fop <= `FLASHOP_READ;
  ///////////////////////////////////////////////////////////////////////
  // Read back data
  ///////////////////////////////////////////////////////////////////////

  8'h10:
    begin
       status <= `STATUS_READING;
       fwdata <= 16'hFF; // Issue "read array" command
       fop <= `FLASHOP_WRITE;
       faddress <= 0;
       state <= state+1;
    end
  8'h11:
    begin
       fop <= `FLASHOP_READ;
       state <= state+1;
    end
  8'h12:
     begin
     if (ready)
      if (faddress == (piece_number+1)*32768)
        begin
       fop <= `FLASHOP_IDLE;
       faddress <= 0;
        end
      else
        begin

          faddress <= faddress+1;
          fop <= `FLASHOP_READ;

        end
         end

    endcase
  else
    fop <= `FLASHOP_IDLE;
////////////////////////////////////////////////////////////////////////////
//
// Status display
//
////////////////////////////////////////////////////////////////////////////
//
// "Reset     ------" --> During reset
// "Read ID   000000" --> While reading ID codes
// "Clr locks 000000" --> While clearing block locks
// "Erase     000000" --> While erasing
// "Write     000000" --> While writing
// "Read      000000" --> While reading
// " *** PASSED *** " --> If the entire test completes with no errors
// "Err: Manuf  0000" --> If an incorrect manufacturer code is read
// "Err: Size   0000" --> If an incorrect size code is read
// "Err: Locks  0000" --> If an error is detected when clearing the block lock bit
// "Err: Erase  0000"
// "Err: 000000 0000"


// Rd  000000  0000
// Wr  000000  0000
// Id  000000  0000
//

function [39:0] nib2char;
   input [3:0] nib;
   begin
  case (nib)
    4'h0: nib2char = 40'b00111110_01010001_01001001_01000101_00111110;
    4'h1: nib2char = 40'b00000000_01000010_01111111_01000000_00000000;
    4'h2: nib2char = 40'b01100010_01010001_01001001_01001001_01000110;
    4'h3: nib2char = 40'b00100010_01000001_01001001_01001001_00110110;
    4'h4: nib2char = 40'b00011000_00010100_00010010_01111111_00010000;
    4'h5: nib2char = 40'b00100111_01000101_01000101_01000101_00111001;
    4'h6: nib2char = 40'b00111100_01001010_01001001_01001001_00110000;
    4'h7: nib2char = 40'b00000001_01110001_00001001_00000101_00000011;
    4'h8: nib2char = 40'b00110110_01001001_01001001_01001001_00110110;
    4'h9: nib2char = 40'b00000110_01001001_01001001_00101001_00011110;
    4'hA: nib2char = 40'b01111110_00001001_00001001_00001001_01111110;
    4'hB: nib2char = 40'b01111111_01001001_01001001_01001001_00110110;
    4'hC: nib2char = 40'b00111110_01000001_01000001_01000001_00100010;
    4'hD: nib2char = 40'b01111111_01000001_01000001_01000001_00111110;
    4'hE: nib2char = 40'b01111111_01001001_01001001_01001001_01000001;
    4'hF: nib2char = 40'b01111111_00001001_00001001_00001001_00000001;
  endcase
   end
endfunction

wire [159:0] data_dots;
assign data_dots = {nib2char(frdata[15:12]), nib2char(frdata[11:8]),
            nib2char(frdata[7:4]), nib2char(frdata[3:0])};
```

```
wire [239:0] address_dots;
assign address_dots = {nib2char({ 1'b0, faddress[22:20]}),
          nib2char(faddress[19:16]),
          nib2char(faddress[15:12]),
          nib2char(faddress[11:8]),
          nib2char(faddress[7:4]),
          nib2char(faddress[3:0])};

always @(status or address_dots or data_dots)
  case (status)
    `STATUS_RESET:
  dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b00100110_01001001_01001001_01001001_00110010, // S
      40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b00000001_00000001_01111111_00000001_00000001, // T
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b00001000_00001000_00001000_00001000_00001000, // -
      40'b00001000_00001000_00001000_00001000_00001000, // -
      40'b00001000_00001000_00001000_00001000_00001000, // -
      40'b00001000_00001000_00001000_00001000_00001000, // -
      40'b00001000_00001000_00001000_00001000_00001000, // -
      40'b00001000_00001000_00001000_00001000_00001000};// -
    `STATUS_READ_ID:
  dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b01111110_00001001_00001001_00001001_01111110, // A
      40'b01111111_01000001_01000001_01000001_00111110, // D
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b00000000_01000001_01111111_01000001_00000000, // I
      40'b01111111_01000001_01000001_01000001_00111110, // D
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b00000000_00000000_00000000_00000000_00000000, //
      address_dots};
    `STATUS_CLEAR_LOCKS:
  dots <= {40'b00111110_01000001_01000001_01000001_00100010, // C
      40'b01111111_01000000_01000000_01000000_01000000, // L
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b00000000_00000000_00000000_00000000_00000000, //
```

```
          40'b01111111_01000000_01000000_01000000_01000000, // L
          40'b00111110_01000001_01000001_01000001_00111110, // O
          40'b00111110_01000001_01000001_01000001_00100010, // C
          40'b01111111_00001000_00010100_00100010_01000001, // K
          40'b00100110_01001001_01001001_01001001_00110010, // S
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
    `STATUS_ERASING:
dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b01111110_00001001_00001001_00001001_01111110, // A
          40'b00100110_01001001_01001001_01001001_00110010, // S
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b00111110_01000001_01001001_01001001_00111010, // G
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
    `STATUS_WRITING:
dots <= {40'b01111111_00100000_00011000_00100000_01111111, // W
          40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b00000001_00000001_01111111_00000001_00000001, // T
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b00111110_01000001_01001001_01001001_00111010, // G
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
    `STATUS_READING:
dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
          40'b01111111_01001001_01001001_01001001_01000001, // E
          40'b01111110_00001001_00001001_00001001_01111110, // A
          40'b01111111_01000001_01000001_01000001_00111110, // D
          40'b00000000_01000001_01111111_01000001_00000000, // I
          40'b01111111_00000010_00000100_00001000_01111111, // N
          40'b00111110_01000001_01001001_01001001_00111010, // G
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          40'b00000000_00000000_00000000_00000000_00000000, //
          address_dots};
    `STATUS_SUCCESS:
dots <= {40'b00000000_00000000_00000000_00000000_00000000, //
```

```
     40'b00101010_00011100_01111111_00011100_00101010, // *
     40'b00101010_00011100_01111111_00011100_00101010, // *
     40'b00101010_00011100_01111111_00011100_00101010, // *
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b01111111_00001001_00001001_00001001_00000110, // P
     40'b01111110_00001001_00001001_00001001_01111110, // A
     40'b00100110_01001001_01001001_01001001_00110010, // S
     40'b00100110_01001001_01001001_01001001_00110010, // S
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b01111111_01000001_01000001_01000001_00111110, // D
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00101010_00011100_01111111_00011100_00101010, // *
     40'b00101010_00011100_01111111_00011100_00101010, // *
     40'b00101010_00011100_01111111_00011100_00101010, // *
     40'b00000000_00000000_00000000_00000000_00000000};//
  `STATUS_BAD_MANUFACTURER:
dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b00000000_00110110_00110110_00000000_00000000, // :
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b01111111_00000010_00001100_00000010_01111111, // M
     40'b01111110_00001001_00001001_00001001_01111110, // A
     40'b01111111_00000010_00000100_00001000_01111111, // N
     40'b01111111_00001001_00001001_00001001_00000001, // U
     40'b01111111_00001001_00001001_00001001_00000001, // F
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     data_dots};
  `STATUS_BAD_SIZE:
dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b00000000_00110110_00110110_00000000_00000000, // :
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00100110_01001001_01001001_01001001_00110010, // S
     40'b00000000_01000001_01111111_01000001_00000000, // I
     40'b01100001_01010001_01001001_01000101_01000011, // Z
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b00000000_00000000_00000000_00000000_00000000,
     40'b00000000_00000000_00000000_00000000_00000000,
     40'b00000000_00000000_00000000_00000000_00000000,
     data_dots};
  `STATUS_LOCK_BIT_ERROR:
dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
```

```
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b00000000_00110110_00110110_00000000_00000000, // :
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b01111111_01000000_01000000_01000000_01000000, // L
      40'b00111110_01000001_01000001_01000001_00111110, // O
      40'b00111110_01000001_01000001_01000001_00100010, // C
      40'b01111111_00001000_00010100_00100010_01000001, // K
      40'b00100110_01001001_01001001_01001001_00110010, // S
      40'b00000000_00000000_00000000_00000000_00000000,
      40'b00000000_00000000_00000000_00000000_00000000,
      data_dots};
  `STATUS_ERASE_BLOCK_ERROR:
dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b00000000_00110110_00110110_00000000_00000000, // :
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b01111110_00001001_00001001_00001001_01111110, // A
      40'b00100110_01001001_01001001_01001001_00110010, // S
      40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b00000000_00000000_00000000_00000000_00000000,
      40'b00000000_00000000_00000000_00000000_00000000,
      data_dots};
  `STATUS_WRITE_ERROR:
dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b00000000_00110110_00110110_00000000_00000000, // :
      40'b00000000_00000000_00000000_00000000_00000000, //
      40'b01111111_00100000_00011000_00100000_01111111, // W
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b00000000_01000001_01111111_01000001_00000000, // I
      40'b00000001_00000001_01111111_00000001_00000001, // T
      40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b00000000_00000000_00000000_00000000_00000000,
      40'b00000000_00000000_00000000_00000000_00000000,
      data_dots};
  `STATUS_READ_WRONG_DATA:
dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b01111111_00001001_00011001_00101001_01000110, // R
      40'b00000000_00110110_00110110_00000000_00000000, // :
          40’b00000000_00000000_00000000_00000000_00000000,
          address_dots,
          40’b00000000_00000000_00000000_00000000_00000000,
          data_dots};
     default:
     dots <= {16{40’b01010101_00101010_01010101_00101010_01010101}};
     endcase

endmodule
```

A.4 MATLAB Scripts

A.4.1 Terman IPOD Person Image Processor

```
close all clear all
%FOR LOOP to read each of 12 images
for j=1:12
% LEFT COMMANDS
% The left commands were saved as 24-bit bitmaps that were
% 64x115.  There are twelve images on this side.
%     switch(j)
%         case(1)
%             image_name=’a1.bmp’;
%         case(2)
%             image_name=’a3.bmp’;
%         case(3)
%             image_name=’a5.bmp’;
%         case(4)
%             image_name=’b1.bmp’;
%         case(5)
%             image_name=’b3.bmp’;
%         case(6)
%             image_name=’b5.bmp’;
%         case(7)
%             image_name=’c1.bmp’;
%         case(8)
%             image_name=’c3.bmp’;
%         case(9)
%             image_name=’c5.bmp’;
%         case(10)
%             image_name=’d1.bmp’;
%         case(11)
%             image_name=’d3.bmp’;
%         case(12)
%             image_name='d5.bmp';
%     end
% RIGHT COMMANDS
% The right commands were saved as 24-bit bitmaps that were
% 64x115.  There are twelve images on this side.
    switch(j)
        case(1)
            image_name='a2.bmp';
        case(2)
            image_name='a4.bmp';
        case(3)
            image_name='a6.bmp';
        case(4)
            image_name='b2.bmp';
        case(5)
            image_name='b4.bmp';
        case(6)
            image_name='b6.bmp';
        case(7)
            image_name='c2.bmp';
        case(8)
            image_name='c4.bmp';
        case(9)
            image_name='c6.bmp';
        case(10)
            image_name='d2.bmp';
        case(11)
            image_name='d4.bmp';
        case(12)
            image_name='d6.bmp';
    end
    imdata=imread(image_name); %Pull image data from file
    image(imdata); % Draw the image for personal gratification
    r=transpose(double(imdata(:,:,1))); % Transposed red data
    g=transpose(double(imdata(:,:,2))); % Transposed green data
    b=transpose(double(imdata(:,:,3))); % Transposed blue data
    % Convert each color channel to binary to cut off some bits
    a=dec2bin(r(:,:,1));
    b=dec2bin(g(:,:,1));
    c=dec2bin(b(:,:,1));
    % Save the most significant 9-bits
    d=strcat(a(:,1:3),b(:,1:3),c(:,1:3));
    % Convert the strings back into an array
    e=bin2dec(d);
    % Append the end of an array holding all the image info with the
    %    latest image processed
    f((j-1)*length(e)+1:j*length(e))=e;
end

%Write the image data to file
%Uncomment following line to write to left_commands.coe
%csvwrite('left_commands.coe',f)
%Uncomment following line to write to right_commands.coe
csvwrite('right_commands.coe',f)
```

A.4.2 General Image Converter

```
close all;
clear all;
s=imread('cover copy','bmp'); % Read image
% Convert each color channel to a 4 bit number
r=dec2hex(s(:,:,1)); g=dec2hex(s(:,:,2)); b=dec2hex(s(:,:,3));
d=strcat(r(:,1),g(:,1),b(:,1)); % Take the top 4 bits of each color channel
e=hex2dec(d(:,1:3)); % Convert back to an array
for i=1:size(s,2)
    f(:,i)=e((i-1)*size(s,1)+1:i*size(s,1)); % Recreate a matrix with the array
end figure
dlmwrite('cover.coe',f,'delimiter',',') % Write the matrix with comma delimited value
```

A.4.3 WAV Converter and Processor

```
clear all; close all;

format long
for i=0:300 %Groups of 32768 samples to write
    num=i;
    %Read the data at the desired interval
    [y,fs,nbits]=wavread('yeah',[num*32768+1 (num+1)*32768]);
    % Rename the current sample
    music=y;
    % Convert the numbers to 8 bit numbers
    music3=round((music(:,1)+music(:,2))/2*2^7);
    % Two's complement conversion
    for i=1:length(music3)
        if(music3(i)>=0)
            music4(i)=music3(i);
        else
```

```
            music4(i)=255+music3(i)+1;
        end
    end
    % Add character 'd' to the end of each number to indicate
    %   that the numbering system is decimal
    data4=strcat(num2str(music4'),'d');
    % Indicate information about the file name based on the
    %   interval that the file covers
    fid = fopen(strcat(num2str(num),'.dat'), 'wt');
    % Print to the file
    fprintf(fid, '%1id\n', music4);
    % Cloes the file so it can be accessed elsewhere
    fclose(fid)
end
```

Appendix B

Additional Photos

B.1 Lab Group

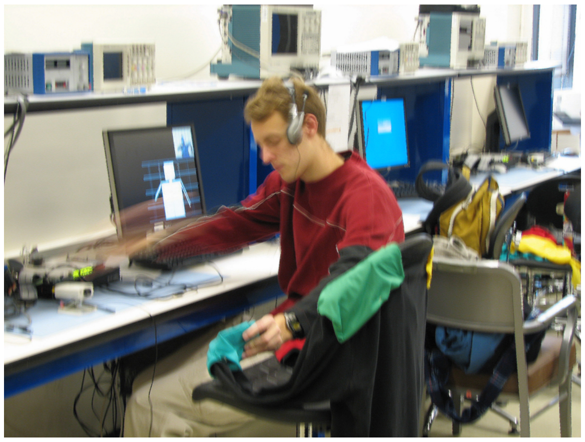


Figure B-1: Jonathan Burnham working away at 9 AM on a Saturday morning

Figure B-2: By himself

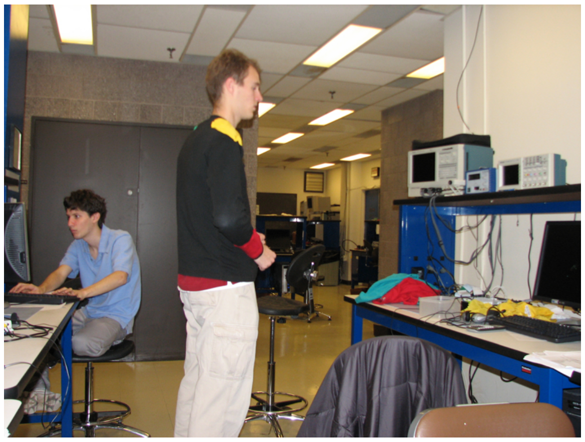


Figure B-3: The boys working away

Bibliography

[1] Annamaria Ayuso and Sharmeen Browarek. Mit dance dance revolution. http://web.mit.edu/6.111/www/s2006/PROJECT/18/main.html, May 2006.

[2] DLP Design. Dlp-usb245m-g usb to fifo parallel interface module. http://www.dlpdesign.com/usb/dlp-usb245m13.pdf, 2002.

[3] Gamedev.net. Beat detection algorithms. http://www.gamedev.net/reference/programming/feature 2005.

[4] Nathan Ickes. Vga video output. http://web.mit.edu/6.111/www/f2006/index.html, 2005.

[5] Helen Liang, Wendi Li, David Meyer, and Lucia Tian. Piano dance revolution. http://web.mit.edu/6.111/www/s2006/PROJECT/14/main.html, May 2006.

[6] Christopher Wilkens and David Rush. Virtual juggling. http://web.mit.edu/6.111/www/f2005/projects/wilkensc_Project_Final_Report.pdf, 2005.