Hand Motion Control of an Audio Player

Diana Cheng and Doris Lin
6.111—Introductory Digital Systems Laboratory
December 14, 2005

Version 1.1

Abstract

Recent technology has taken advantage of motion sensors to accomplish basic commands usually implemented through switches or remote controls. The hand motion controlled audio player created in this project will carry out basic commands, such as volume control, track selection, audio playback and record, stop, and pause, using hand gestures. The audio player will also include a visual component, displaying which functions the user is carrying out, via the VGA. The user will wear a three-axis digital accelerometer on his/her hand through which hand motions will be detected. Upon starting the audio player, the user will be asked to calibrate his/her motion for each command—the motion then being written to an SRAM. After calibration the audio player will be launched, and once the motion is tracked, the controller will map the motion to its corresponding command, and the audio player responds with the correct control of an audio file stored in one of the labkit's ZBT memories. The process of testing and debugging the audio player involved first testing the accelerometer and the audio player individually. The two major components were then interfaced and further analyzed. The end result is an audio player which responds to the user's motion, whether it is turning up the volume or skipping to a new track.

Table of Contents:

1 Overview
  1.1 Functionality 5
  1.2 User Interface 5

2 Accelerometer
  2.1 Accelerometer Specifications 7
  2.2 Accelerometer Interfacing 7
  2.3 RS232 FSM to Read in Data into a Bus 8
  2.4 Implementation of the Tilting Recognition 8
    2.4.1 Calibration State 8
    2.4.2 Decode State 9
  2.5 Motion Processing 9
    2.5.1 Using Matlab to Pre-Process 9
    2.5.2 Filter/Sampling Implementation in Verilog 10
    2.5.3 Thresholding 11
    2.5.4 Edge Detection 11
    2.5.5 Calibration 11
    2.5.6 Decode 11

3 VGA Display
  3.1 VGA Video Format 12
  3.2 VGA Timing 13
  3.3 Calibration Display Module 13
    3.3.1 Divider Module 15
    3.3.2 Character String Display and the Font ROM 16
  3.4 Audio Player Display Module 16
    3.4.1 Button Modules 18
      3.4.1.1 Circle Buttons 18
      3.4.1.2 Square Button 19
      3.4.1.3 Triangle Buttons 19
    3.4.2 Message Module 19

4 Audio Playback
  4.1 Audio Interface 20
    4.1.1 AC97 Module 20
    4.1.2 AC97 Command Module 21
  4.2 Audio Player Module 21
    4.2.1 Audio FSM 21
    4.2.2 Mode FSM 23
    4.2.3 ZBT RAM 24
    4.2.4 Track Memory 24
    4.2.5 Echo Memories 25

5 Interfacing the Accelerometer and Audio Player
5.1 Interface Module …………………………………………………………….. 26

6 Testing and Debugging
6.1 Accelerometer Testing and Debugging ……………………………………….. 27
6.2 VGA Testing and Debugging ……………………………………………….. 28
6.3 Audio Testing and Debugging ……………………………………………….. 29

7 Conclusion ………………………………………………………………………. 30

8 Future Directions ………………………………………………………………. 30

List of Figures

Figure 1: Block Diagram 6
Figure 2: Accelerometer chip LIS3L02DQ 3axis digital output 7
Figure 3: RS232 FSM 8
Figure 4: Tilt Calibrate FSM 9
Figure 5: Original MATLAB Plots Unprocessed 10
Figure 6: Downsampled at 16 and Filtered with a 64 Window Size 10
Figure 7: Thresholded 11
Figure 8: Motion Calidecode FSM 12
Figure 9: VGA Horizontal and Vertical Sync Signals 13
Figure 10: Initial Screen Display 14
Figure 11: Calibration Display in Countdown 15
Figure 12: Audio Player Display 17
Figure 13: AC97 Audio Frame 20
Figure 14: Audio Player Finite State Machine 22
Figure 15: Mode Finite State Machine 24

List of Tables

Table 1: Command Values and Corresponding Function 26

1 Overview

1.1 Functionality

The hand motion controlled audio player simulates a typical audio system with the usual functions like play, pause, stop, record, track selection, volume control, and playback mode options. The unique aspect of this audio player is that it can be controlled through the user's hand gestures rather than a button press. The hand gestures are detected using an accelerometer that will transmit data to the FPGA depending on what motion was done. There will be an initial calibration stage in which the user is instructed to calibrate a gesture for a specific command. After calibrating, the user can then use the audio player by first recording audio. The audio can then be played back with various manipulations, including volume control and track selection. In addition, the different playback options of the audio player include normal playback, echo mode, Alvin the Chipmunk, and Barry White modes.

1.2 User Interface

The user interacts with the audio player via a three-axis digital accelerometer. The accelerometer sits atop a Velcro brace that is worn around the user's hand. The accelerometer is linked to the FPGA labkit with five wires. These wires provide a voltage source and ground to the accelerometer along with lines to transmit and receive data to and from the FPGA. Using this accelerometer, the user can communicate with the audio player.

The audio player then communicates back by implementing the motioned audio command and also via the VGA display on a computer monitor. The VGA display gives the user instructions on how to calibrate the accelerometer and then changes the display to show an audio player visualization after calibration is over. The audio player display allows the user to see which functions are being carried out with simulated button pushes and messages shown on the screen. A block diagram of how the accelerometer, VGA, and audio component interact with one another can be viewed below.

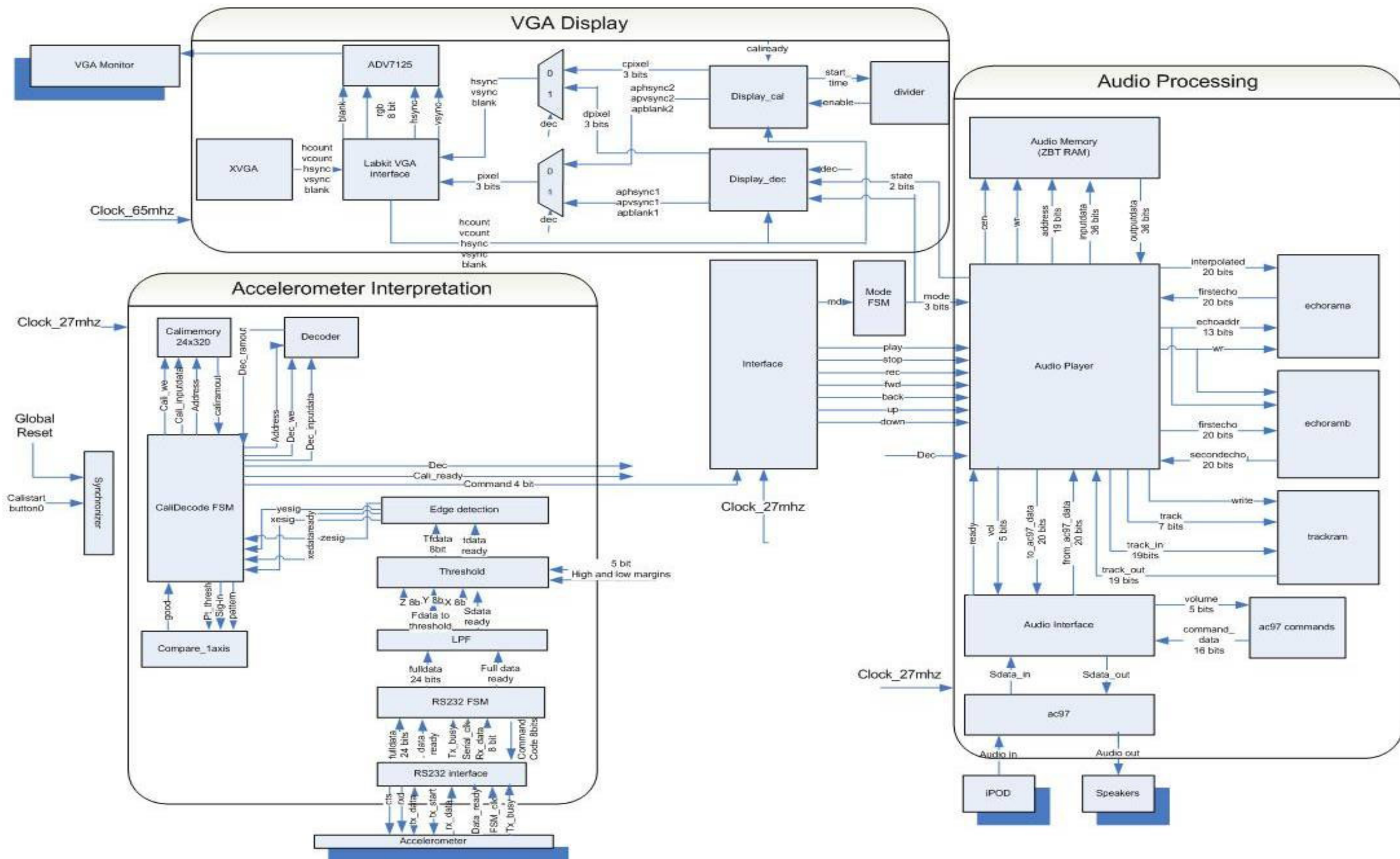

Figure 1: Block diagram

The block diagram shows how the accelerometer, VGA, and audio interact with one another and how the user can affect the audio player through its inputs—iPOD and accelerometer—and see and hear the results via the outputs—speakers and VGA monitor.

2 Accelerometer (Diana Cheng)

2.1 Accelerometer Specifications

The accelerometer used in this project was to provide a means for interpreting and detecting hand gestures from the user. This was then to be passed onto the audio portion where each gesture/motion would correspond to an audio command. The accelerometer used in this lab was the ST Microelectronics 3 axis ±2g digital output MEMS accelerometer, part number LIS3L02DQ. This part came on an evaluation board EK3L02DQ which had already converted the i2c interface from the accelerometer to a standard RS232. It took an external power supply from 3 volts up to 18 and could be immediately tested and hooked up onto the Hyper Terminal. There are two connections between the FPGA and the data chip, a transmission line and a receiver line.

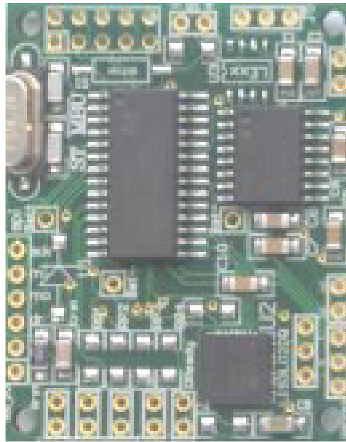

Figure 2: Accelerometer chip LIS3L02DQ 3axis digital output
The evaluation board came with power regulations and standard output pins

2.2 Accelerometer interfacing

To get the FPGA chip to be able to interface with the chip, a RS232 module had to be created. The specifications to talk to the accelerometer included using a standard RS232 interface at an 115200 baud rate, with a data word of a byte, 1 stop bit and no parity. Transmission should occur with least significant bit first. First, a baud clock needed to be generated from the Clock_27mhz signal. This was done using the www.fpga4fun.com website that gave a formula in how to convert standard clock signals into baud rates. Using the

```
BaudGeneratorInc =
    ((Baud<<(BaudGeneratorAccWidth-4))+(ClkFrequency>>5))/(ClkFrequency>>4);
```

formula provided, a serial clock was generated. The serial clock would serve as a data ready to the RS232 FSM signaling every 10th baud clock that a byte was received.

This module was split up into two sections since transmission and receiving could not occur at the same time. Thus ready and enable signals are passed within the module to safeguard against trying to do both at the same time. The FPGA will default to being in receiving mode constantly until a transmission signal interrupts.

2.3 RS232 FSM to read in the data into a bus

Once the basic interface was built, then came to controlling signals that went into and came out of the chip. This consisted of an FSM with 12 states each transitioning on the serial clock. States 1-3 are dedicated to transmitting the single acquisition commands to the chip to query for data. 4F 6F 6E in hex, this will cause the chip to display a 2's compliment string of 8 bytes in the order of S T Xhigh Xlow Yhigh Ylow Zhigh Zlow. Single data acquisition was chosen instead of continual data acquisition because this way, the FSM can completely control and manage the timing issues that would otherwise occur. By allowing the FSM to control the transit and receive, the FSM would be able to sync with the data receive lines more accurately. However, to mimic continual data acquisition, the FSM continually looped around constantly sending it the acquisition commands.

Once the FSM passes the three transmission states, it will loop in a wait state waiting for the chip to respond. However due to the long lag time between transmission and receive, as there was a huge lag time between transmissions and receiving, it was often the case that the chip would miss the beginning bits. Checking for the S byte and the T byte, received signals for the FSM to expect the x y and z data by the next serial clocks. A wait expired signal was created in the event that the S or T byte was missed and never received. This state would wait for roughly .37 seconds before it would reset and retransmit an acquisition signal.

Once S and T received, the rest of the states push the byte into a variable. When all 8 bytes have been received, the variables are concatenated and sent as a 48'bit bus as well as a fulldata_ready signal.

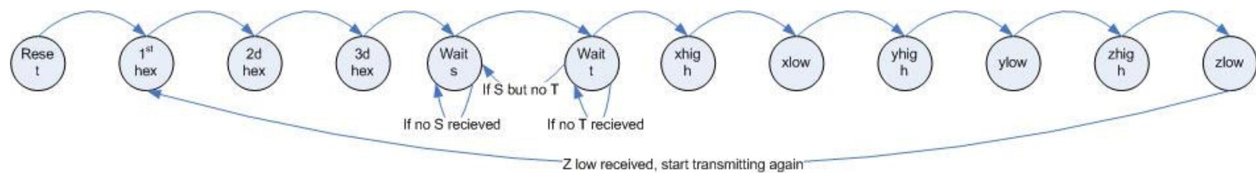

Figure 3: RS232 FSM

The states in how to receive data from the accelerometer

2.4 Implementation of the tilting recognition

Initially, the project handled taking in the tilting gestures of the user. This system required a 2 second timer. During calibration, the user was given a button to start to hold their position for 2 seconds. Once the tilt was recorded, they go back to waiting for the next button push. In this system, the user had to hold their tilt for at least 2 seconds during calibration phase at which time the calidecode module will write the Xhigh, Y high and Zhigh bytes into a 9x24 Sram. Then during decode state, if at any point the user tilted to that position, the command was given. This allowed for the user to customize their own positions that corresponded to the commands. Also in this case left handed users would be compensated for as well. To implement this, the next stage was to go directly to the calidecode module

2.4.1 Calibration state

This module consisted of 3 states. Upon resetting, the FSM started out in the first state where it waited for the user to push the calistart button. Once the button is pushed, a calibrate signal is sent to the audio player as well as a 2 second enable signal to a 2second clock divider. Once the 2second disable is received from the clock divider, the calibrate memory will record the full data position at that instant. If more commands needed to be calibrated, they will return to the Caliwait stage waiting for the next button press.

Once all 9 are written, there is no way to return back to calibration without resetting the system.

2.4.2 Decode state

In decoding stage, it continually parses through the calibrate memory constantly matching the full data xyz information to the memory. No data will be lost in this method because each full data comes at a .0007 seconds however comparing with all 9 address in the calibrate memory occurs every .0000003 seconds so there is plenty of time between each full data received. Thus all positions will be checked and thus won't miss a command.

Once a match is found the module will send out which address the user matched to and that will be distributed to the audio processing.

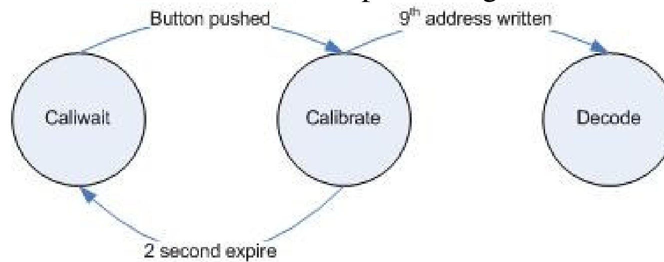

Figure 4: Tilt Calibrate FSM

The states in how to decode and calibrate tilt recognition from the accelerometer

2.5 Motion processing

To further create a more robust system, movement was attempted to be processed. Digital signal processing methods were used. First off, from the RS232 FSM comes in the full data at a 1429 Hz rate. This along with a fulldata_ready signal will first get filtered to eliminate noise and smooth curves. Once filtered, the data will get sampled at a certain rate and then threshold to create very basic shapes with minimal key features such as maxs, mins and x intercepts. Then written to the calibration memory, this will be patterned matched based on a point comparison algorithm.

2.5.1 Using Matlab to pre-processing

In order to realize what shapes the data points correspond to hand movements it was first recorded in secureCRT and written to a text file. In matlab, this was then decoded as 2's compliment numbers and filtered, sampled and threshold. Then in plotting the graphs, thresholds could be determined. Simple matlab codes to process these signals were run through an optimization code that found the threshold values that returned the highest percent of true positives that occurred from the database of hand gestures recorded.

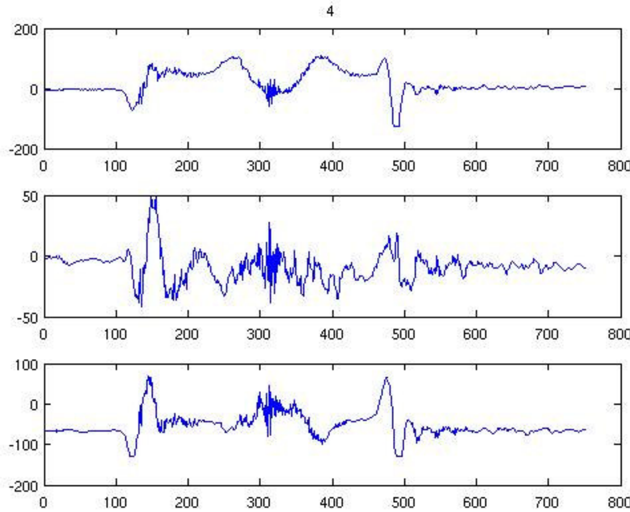

Figure 5: Original matlab plots unprocessed
For one hand gesture x y and z are plotted here

2.5.2 Filter/sampling Implementation in Verilog

Taking only the top 8 bits of each axis, allows for more leniency in the motions. The bottom bits of each axis were too sensitive to be used as valid data. First off the full data ready was sent to the low pass filter module (lpf) that reads in every fulldata and will integrate over a window of 64 every 5th fulldata point. This in effect samples and filters at the same time to avoid unnecessary computation. By saving computation, this will ensure the program enough time to calculate the average of 64 points before the next fulldata signal. The window size and the fulldata number were thresholds found in matlab that served to optimize the smoothest curve that still retained enough key features for a distinct motion. Once completed the sdata and an sdata_ready signal is sent to the thresholding module

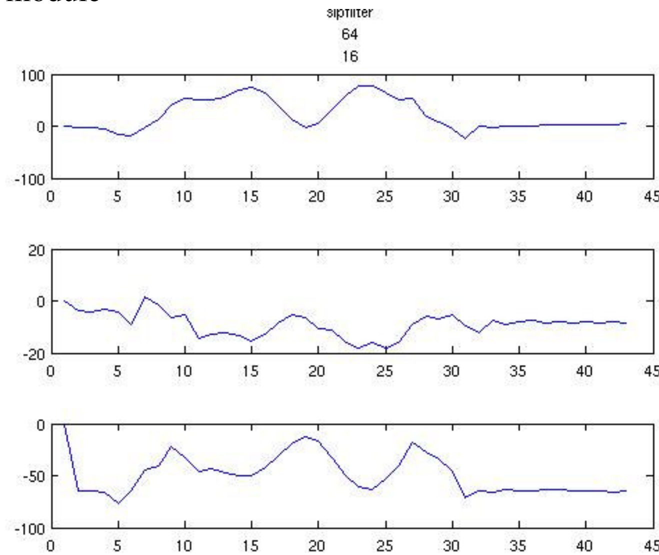

Figure 6: down sampled at 16 and filtered with a 64 window size
For one hand gesture x y and z are plotted here

2.5.3 Thresholding

The theory of the thresholding module is to zero out any noise that could occur from the user's unstable hand. This module will set a high and a low threshold in which if the data point fell within those margins, it will be pushed towards the dc value. For the x and y axis the dc value was zero as it felt no gravity at rest. The z axis dc value was set to -1g since at initial rest (being placed flat on the table) it felt a force of gravity.

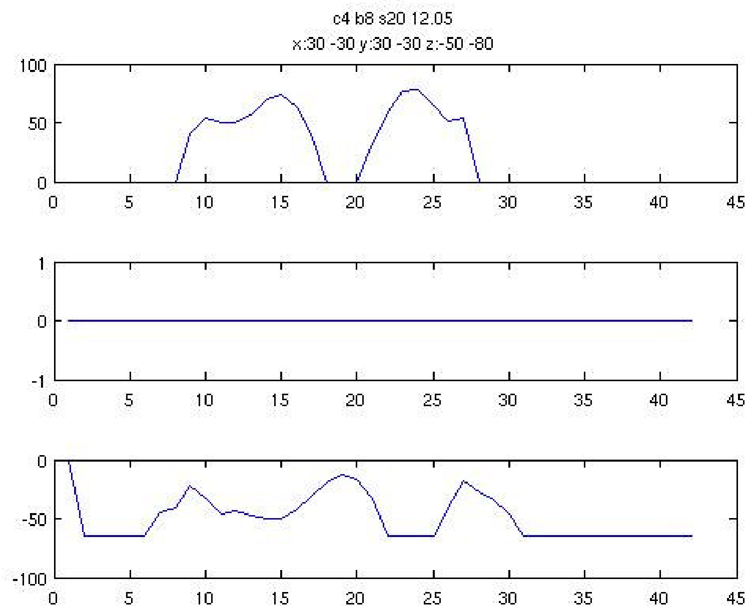

Figure 7: Thresholded
X margins ±20, Y margins ±20, Z margins +15, -20

2.5.4 Edge detection

Once thresholded, the edge detector will send a signal showing non_dc values have been read. This too also thresholds as sometimes noise will be created in the thresholding process. However by making sure that each max and min is larger then 3 non_dc values, it will disregard random spikes in the graph. This will then send a signal to the calibrate state which indicates to start calibrating/decoding

2.5.5 Calibration

Similar to the tilt method this module has a few more states including caliwait, cali trigger, calimemory, decodetrigger, decodemem, and decode. During calibration the signal will wait for the button push from the user, but instead of waiting for 2 seconds, the user has to move their hand in order for the edge detector to see the beginning of the movements. Once edge detected, the next 32 sampled data points will be written to the 24x320calimemory. XYZ bus will be written in each address. The top four bits of the calimemeory address correspond to which motion is being calibrated at that moment.

2.5.6 Decode

Once all 9 motions are calibrated, the FSM is forever in the decode state unless reset. This will act rather like the calibration state in that it will search for an edge and

then once found, write the next 32 xyz data points into the 24x32decode memory. Once written, the decode stage includes sending it into a comparer module that will take each x y and z point and compare it with the calibrated x y and z point. Then setting a point threshold, it will keep a running x y and z score value. For each x point with in the calibration point ±threshold, the x score will add 1 point. Similar for y and z, the max score will be 32. Then setting a total score threshold if the xscore is within 32 minus the totalscore threshold, then it will return a 1 for the xgood variable. Similar for y and z, the decode state will assign a command to the motion only if all xgood, ygood and zgood are all 1's. If none of them matched, it will disregard the values. The timing of this system is sufficient because each sdata_ready comes at a 90hz cycle. The comparing runs off of the 27mhz clock cycle comparing xyz in one cycle. Thus it will take 84375 Hz to complete comparisons for all 9 motions. Thus leaving plenty of time before the next motion occurs.

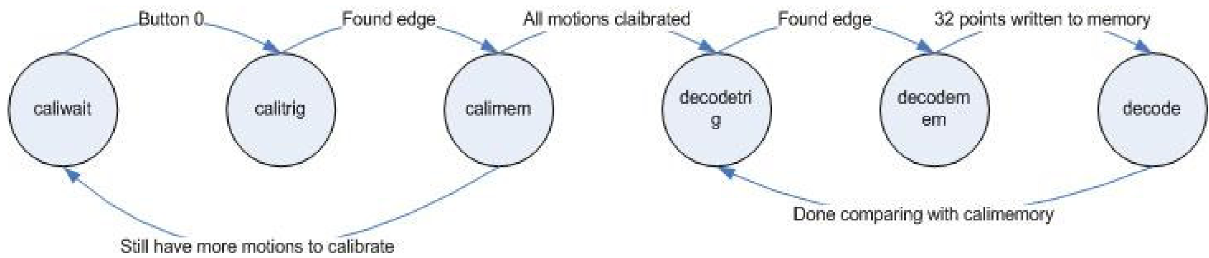

Figure 8: Motion Calidecode FSM
Slightly modified from tilt calidecode FSM

3 VGA Display (Doris Lin)

3.1 VGA Video Format

The visual component of the project was provided using VGA video format displayed on a computer monitor. This format consists of non-interlaced frames composed of horizontal lines of pixels. The frame data are transmitted starting from the top line to the bottom, and the pixels are sent from left to right. The VGA display we created had frame dimensions of 1024x768 pixels. Each pixel of the video includes an R,G, and B component, each 8-bit values. In addition to the pixel values, which compose the active regions of the frame, each frame also includes blanking regions. The blanking intervals occur at the end of every line and at the end of every frame. These blanking regions are controlled by the horizontal sync, vertical sync, and blank signals, all of which can override the pixel value. The timing of these pulse signals can be seen in the diagram below.

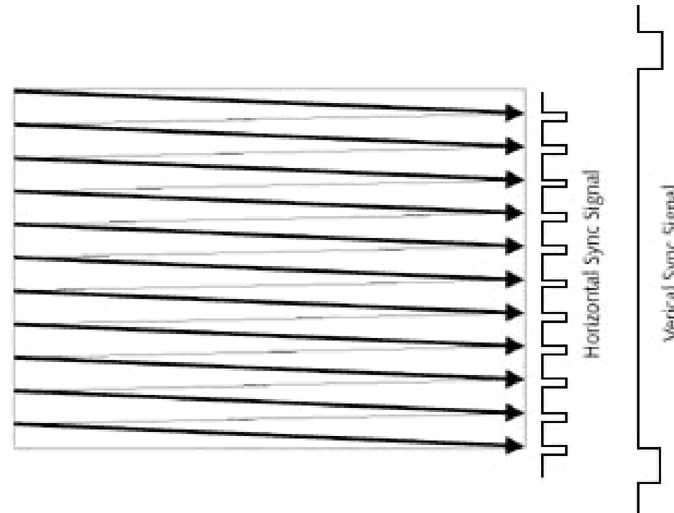

Figure 9: VGA Horizontal and Vertical Sync Signals
The horizontal sync signal pulses at each line end and the vertical sync signal pulses at each frame end (Taken from http://www.tkk.fi/)

During the blanking interval, black pixels are transmitted until the horizontal or vertical sync pulse, when pixels below the normal black level are transmitted. These signals and the pixel values are processed by the ADV7125 video DAC on the Xilinx labkit.

3.2 VGA Timing

The VGA display used for our audio display is a 1024x768 pixel frame with a 60 Hz refresh. According to VGA standards of Xilinx, Inc., this requires a pixel clock of 65 MHz. The Xilinx labkit has a built-in 27 MHz clock. In addition, the labkit includes digital clock managers (DCM) and one can be used with the built-in oscillator to create the 65 MHz pixel clock.

3.3 Calibration Display Module

The calibration display follows a sequence of nineteen screens to relay instructions to the user to calibrate the accelerometer. The first screen displays three strings giving the user an initial message to begin calibration. Each screen thereafter displays two strings. These screens alternate between a calibration countdown screen and a wait screen before the next calibration. The module determines which string to write to the monitor based on several signals. The initial screen of the calibration display can be seen in the following figure.

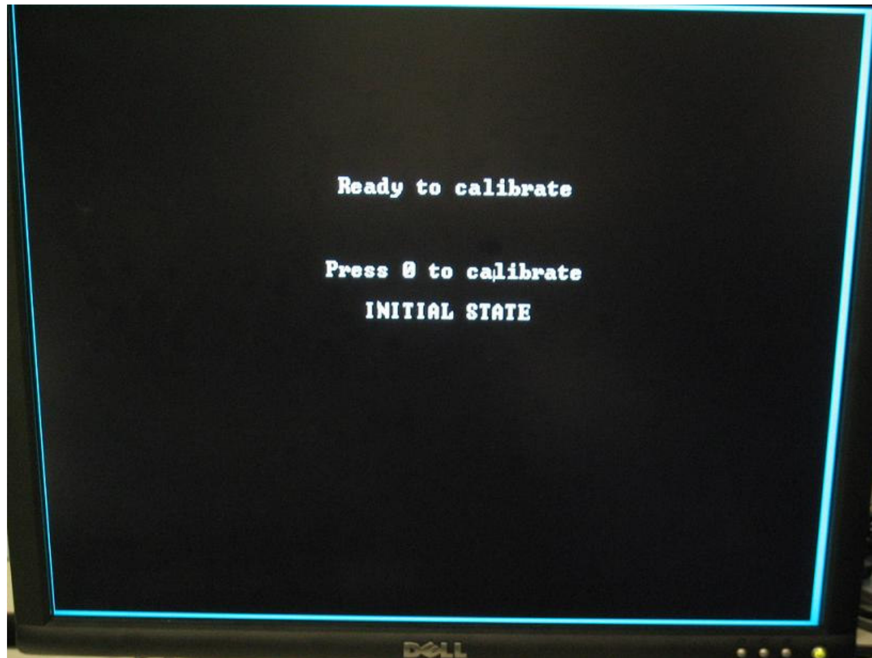

Figure 10: Initial Screen Display
The initial screen upon starting or resetting the system, tells the user to how to begin calibrating.

One of the important signals received by the calibration display module is next, sent by one of the accelerometer modules. The accelerometer responds to a button push that will trigger the calibration of the next function. The accelerometer in turn sends a signal to this VGA module to indicate that calibration of the next command has begun. When this signal is received, a nextcal pulse is created out of it and a calibration screen is shown. The screen displays a message notifying the user that he/she is calibrating a motion along with a five second countdown. When the five seconds is up, another screen is presented to the user. This screen informs the user to press the calibration button and informs him/her which motion will be calibrated next. There are nine calibrated motions in total. The figure below shows the display module counting down at two seconds.

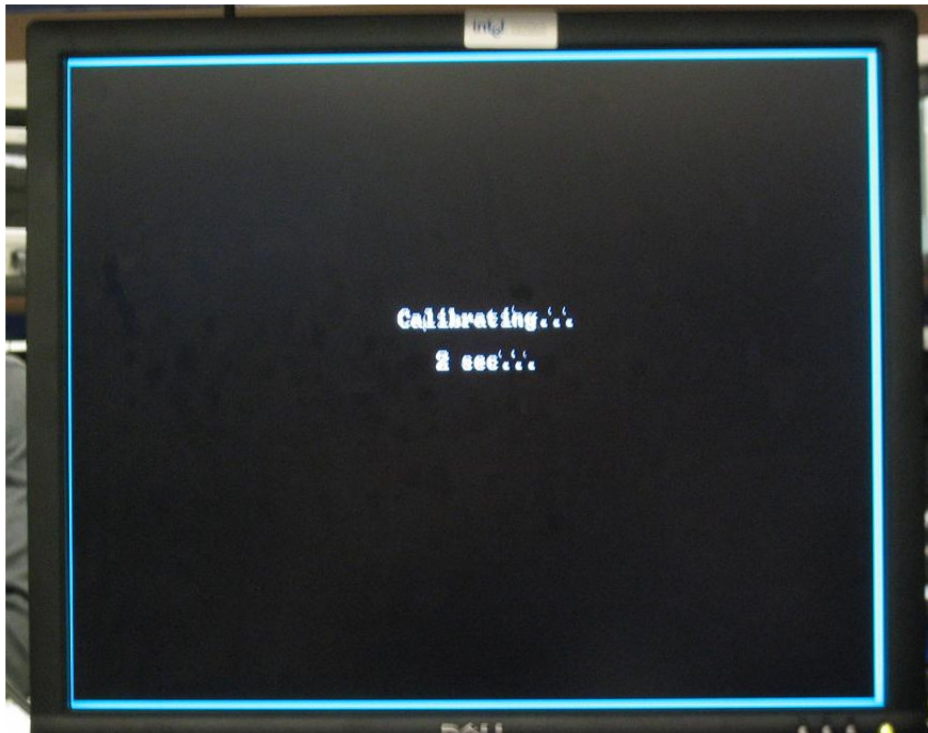

Figure 11: Calibration Display in Countdown
The calibration display counts down during calibration from five seconds to zero.

This module also contains a four-bit counter register. After each calibration, the counter is incremented. The counter is used to keep track of which motion is being calibrated and is used to determine which string to display. For instance, when the counter equals zero, its initial value, the string displaying the next function to calibrate is chosen to be "PLAY/PAUSE." Then when the counter reaches seven, the module knows only one calibration is left and the chosen string to display is "MODE."

The strings to be displayed are stored in the cstring registers, whose values are sent to instantiations of the char_string_display module provided by the 6.111 staff. Each instantiation of the char_string_display returns a three-bit pixel value, which are then latched and or-ed together to get one final three-bit value to return and transmit to the ADV7125 chip.

3.3.1 Divider Module

As mentioned previously, some of the screens in the calibration sequence display a five-second countdown. This countdown depends upon a one hertz enable signal created by the divider module. The divider takes the 65MHz clock, and using a 26-bit counter, counts up to 65 million. When it reaches this value, it assigns a high value to the enable signal for one clock cycle. Each time the enable pulses is thus one second and signals the calibration display module to count down by one on the monitor.

While the divider transmits a signal to the calibration display, the display module also transmits a signal to the divider. This input to the divider module is a start signal and tells the divider to reset the counter to zero and continue counting to 65 million. The start_time signal in the display module goes high when ever the nextcal pulse goes high and the time_ct equals five,

the initial countdown value. When these conditions are true, it means the motion calibration has just begun and the countdown should also begin.

3.3.2 Character String Display and the Font ROM

The other module called by the calibration display is char_string_display, which accesses a font ROM. The character string display module takes a string as an input and returns a 3-bit RGB pixel value. The module takes each character from the string and looks up the character in the font ROM that has an initial file loaded into it. This file includes 8x12 bit characters which are doubled in size on the VGA display, so each character is actually 16x24 pixels.

The delay in getting these characters from the font ROM and displaying them on the screen is rather large, nearly 20 ns, which creates glitches in the display since each pixel clock cycle is only about 15 ns long. To allay this large delay, the assigned strings in the display_cal module were split up and sent to the character string display in parts. In addition, the returned pixels from the character string display module were latched and pipelined.

3.4 Audio Player Display Module

Once the calibration stage is over, the accelerometer transmits a signal indicating that it is beginning to decode the user's motions. This signal switches the monitor to displaying the audio player screen. The display simulates what a typical audio player would look like with the function buttons in a toolbar at the bottom of the screen. Included buttons are play, pause, stop, record, skip forward, volume up, and volume down. In the top left corner is a message informing the user as to which function is currently being carried out, and in the top right corner are four bars for each of the four playback modes—normal, echo, Alvin, and Barry. All the buttons begin as blue and are highlighted lime green when they are "pushed." The mode bars work similarly. The current playback mode is highlighted in lime green while all the other bars remain blue. The buttons and bars are created using sprites, and the messages are displayed once again by calling the character string display module. A picture of the audio player display can be seen in the figure below.

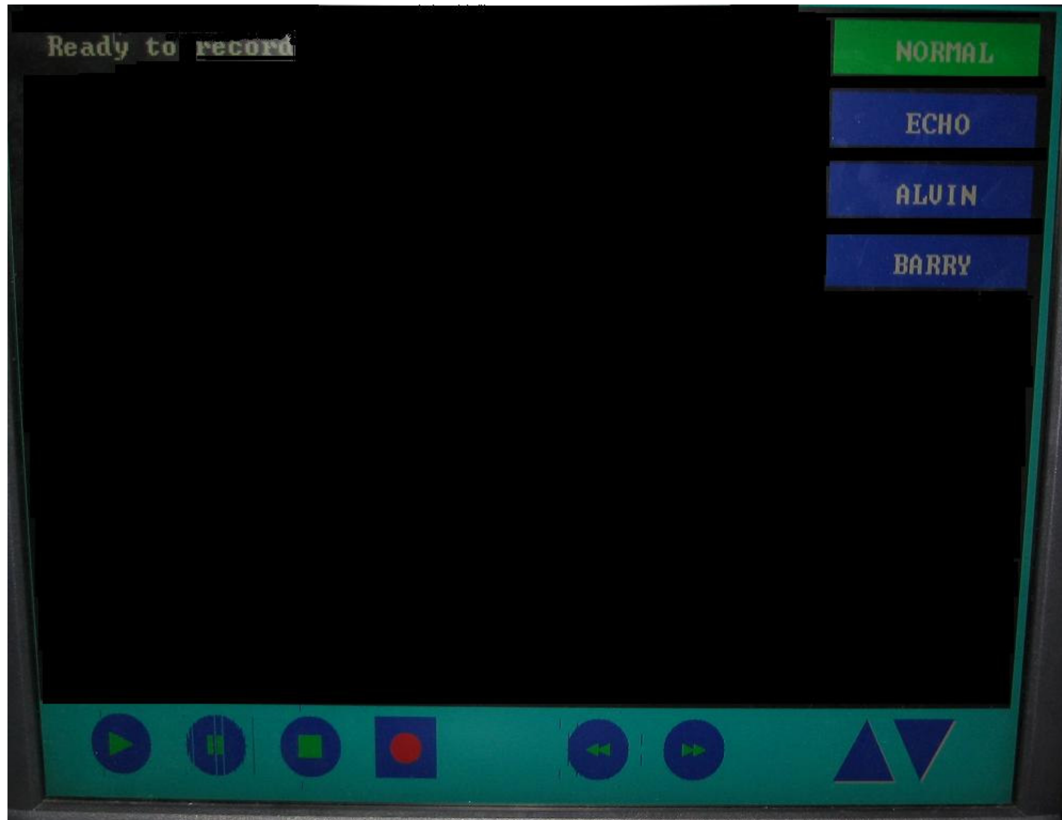

Figure 12: Audio Player Display

The audio player display initially displays the message "Ready to record" in the top left but later displays whatever function is being carried out. The button bar is at the bottom of the screen and the mode bars at the top right of the screen.

The sprites and messages are created by instantiating other modules that all return a 3-bit pixel value. These pixel values are latched and then or-ed together to form one RGB value. The final 3-bit RGB value is returned to the higher level and transmitted to the ADV7125 chip.

The display includes FSMs that serve to determine the color of the sprites on the screen or to create signals to pass to the button modules. The display module takes as a 3-bit mode input, which is used to set color values for the mode bars. For instance, when mode equals zero, this means the audio player is in normal playback mode, so the first mode bar—the block sprite for the normal mode message—is set to be a color of lime green while all the others are set to blue. The display module also takes another 3-bit input called func. This input indicates which state the audio player is in, such as play, pause, stop, or record state, and is used to create push signals to be sent to the button modules. These signals simulate a button being pushed and held down, as will be seen on the display when the "pushed" button turns bright green. For instance, when func is zero, this signifies that the audio player is in play state. The display module sets the signal pplay to one and all other signals—pstop, ppause, and prec—to zero. These signals are each sent to their respective button instantiation. When they are high, this tells the button module to set the button color to green instead of blue.

3.4.1 Button Modules

There are three different shapes for the audio player's buttons. Play, pause, stop, skip forward, and skip backward are all circular while record is square and volume up and down are triangular. There are thus three different button modules—one for each shape. The buttons are instantiated in the audio player display module, display_dec, and within each button module, other modules are called to create the button shape and symbol to be placed on the button.

3.4.1.1 Circle Buttons

The module that draws circular buttons takes as input a 3-bit command and a pushed signal and outputs a pixel value. The 3-bit command is sent by the audio player display module and tells the c_button module which symbol pixel—play, pause, stop, skip forward, or skip back—to use. The push signal tells the module if the instantiated button has been pushed and is used to set the color of the button and the symbol.

Within this module, the circle module is called to create the overall shape of the button. This module takes a 3-bit color input to specify the circle's color. It also takes the x- and y-coordinates of the circle's center. The radius of the drawn circle is a parameter and can be changed to the programmer's liking. Using mathematical comparisons to check if the pixel and line count are within the circle's radius, the pixel value of the circle is set and returned to the c_button module.

In addition, the block and tri_RL modules are instantiated to create the button symbols. The block module takes in a 3-bit color input as well that determines the rectangle's color. The width and height of the rectangle are parameters that can be adjusted using defparam. Other inputs to the module are the x- and y-coordinate of the top left corner of the block. The block module uses conditional statements to determine if the current pixel and line count are within the block's area and assigns a pixel value accordingly, returning this 3-bit value to the c_button module.

The tri_RL module works similarly to the circle and block modules. It too takes a 3-bit color input and x- and y-coordinates to specify the triangle's location. The module is designed to be able to draw isosceles triangles pointing left or right, so another input is given—a direction signal. The dir signal tells the module to draw a triangle pointing to the right when it is low and to draw one pointing left when it is high. The x- and y- coordinates are the pixel locations of the center of the base and of the base, respectively. The base and height of the triangle are parameters within the tri_RL module and can be altered using defparam. Once again, mathematical conditions are used to check if the pixel and line count are within the triangle's area. If so, the appropriate pixel color value is returned to c_button. Otherwise, a three-bit zero is returned as the pixel value.

At the end of the c_button module, the appropriate symbol pixel is chosen based on the 3-bit command input and the button and symbol pixels. The pixel values are conditioned so that the symbol pixel takes precedence over the button pixel. Using this condition, one 3-bit RGB value is created to return to the audio display module.

3.4.1.2 Square Buttons

There is only one square button displayed on the monitor, and this is the record button. The square button module is then tailored toward the record button and called rec_button. Rec_button takes a push signal that, like the c_button module, indicates whether the button has been pushed. If so, the button's color is set to lime green, and otherwise is left blue. The button shape is created by instantiating the block module and drawing a rectangle with equal height and width. The record symbol is created by instantiating the circle module with the radius adjusted to be smaller. The block and circle modules both return pixel values to the rec_button module which ors the two values together for one 3-bit pixel value to send back to the audio display module.

3.4.1.3 Triangle Buttons

The triangle button module is used for the volume up and down buttons. Like the other buttons, this module takes a push signal that determines the color of the button and returns a 3-bit pixel value to the audio player display module. However, these buttons are unique from the rest in that they have no symbol component but have a shadow. The offset of the shadow is determined by whether the push signal is high or low. If high, the shadow is to the upper left, and if low, the shadow is to the lower right. The t_button module also takes an additional symbol called up_down, which determines whether it should draw a triangular button pointing up for volume up or one pointing down for volume down. This signal is passed into the instantiations of the tri_UD module, which draws isosceles triangles either pointing up or down. When up_down is high, the downward pointing triangle is drawn and when low, the upward pointing triangle is drawn. This module has width and height parameters that can be altered with defparam and uses mathematical comparisons to check if the pixel and line count are within the triangle's area. If so, the pixel is assigned the input color value. Otherwise, a black pixel, 3-bit zero, is returned to the t_button module. The t_button module then gives precedence to the actual button pixel over the shadow pixel to create one RGB value to transmit back to the audio player display.

3.4.2 Message Module

In addition to the buttons, strings are displayed on the screen to give information to the user. As in the calibration display, the character string display module and font ROM are used to show these strings. The message module determines which strings to pass to the char_string_display module using an FSM. It takes a 3-bit input called func passed from the audio player to the player display module and then to the message module. This 3-bit input specifies whether the audio player is in play, pause, stop, or record state, and depending on the state, the appropriate string is assigned to the cstring1 register. For instance, when func is zero, the message to display is "Play." For func equals one or two, the string register is assigned to "Paused," and so forth.

The message module also constantly prints four strings to display the choices for playback mode. These strings are parameters within the module and are passed as inputs into four separate instantiations of the character string display. In total, the character string display

module is instantiated five times, returning five different 3-bit pixel values. These pixels are or-ed together and passed back to the main display module.

4 Audio Playback (Doris Lin)

4.1 Audio Interface

Our audio system both records tracks and plays back the recorded data using the Intel AC97 audio codec chip. The chip transmits one bit on each of its clock cycles on its SDATA_IN wire and it receives one bit each clock cycle from the FPGA on its SDATA_OUT wire. The transmitted bits are divided into frames of 256 bits each. This frame begins with a 16 bit tag followed by twelve 20-bit samples. From codec to FPGA, there is one 20-bit sample from each ADC of the codec, and in the opposite direction, there is one 20-bit sample sent to each of the DACs in the codec. The AC97 bit clock runs at 12.288 MHz, so each new frame is ready to be sent at a rate of 48 kHz. The structure of the frame can be seen more clearly in the given figure below.

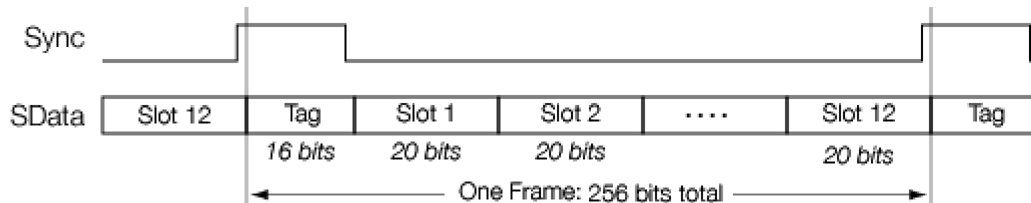

Figure 13: AC97 Audio Frame
Frames in the SDATA lines begin with a 16 bit header tag and are followed by twelve slots of 20-bit samples. The sync signal is high when a new frame is ready (Taken from the 6.111 Labkit Documentation)

The audio module that interfaces the AC97 chip receives the SDATA_IN information and outputs the SDATA_OUT bits. In addition it also outputs a ready signal whenever each new frame is ready to be sent. It also passes through a 20-bit audio sample to the audio player module, which is recorded in record state, and in turn receives a 20-bit audio sample from the audio player when in playback state along with a 5-bit volume value. These inputs are processed and the outputs created with two modules—the ac97 and ac97command modules.

4.1.1 AC97 Module

As previously mentioned, the AC97 chip and FPGA communicate using 256-bit frames headed with a tag. The ac97 module assembles the frame together when transmitting audio from FPGA to AC97 chip and disassembles the frames when transmitting the other direction.

The ready signal that indicates a new frame is set to be sent is generated within this module. It goes high whenever the bit count equals 128 and low again when the bit count equals two. This signal is synchronized with the rising edge of the FPGA's 27 MHz clock, which is the clock that the audio modules run off of.

In addition, the module takes the audio-in data from the AC97 and outputs a 20-bit audio sample, the from_ac97_data signal, to the audio player. In turn, it takes the audio-out data, to_ac97_data, from the audio player and sends 20-bit slots to the AC97 chip.

4.1.2 AC97 Command Module

In order to assemble and disassemble the audio frames, the ac97 module needs command data, which is provided by the ac97command module. This module takes as input the ready signal provided by the ac97 module, the volume sent by the audio interface, and the source signal sent by the audio interface as well. In this case, source is always one to signify that the audio from the microphone port of the FPGA is the sound source. The returned command data and address are dependent on the state of the ac97command module. The state register is four bits, incrementing by one each 27MHz clock cycle. In state three, the volume value is used in determining the 16-bit command data which will control the headphone volume. This command is important for volume control of the audio player. The command address and data are then used in the ac97 module to create the first and second slots of the audio frame.

4.2 Audio Player Module

From the audio interface, a 20-bit from_ac97_data signal is sent to the audio player module, which also sends a to_ac97_data 20-bit audio sample to the audio interface. The audio player module is essentially a large finite state machine that carries out specific functions depending on its current state.

4.2.1 Audio FSM

The states and actions of the audio player module are dependent on its input play/pause, stop, record, skip forward, skip backward, volume up/down, and change mode signals that are generated by the accelerometer/audio player interface. All these signals behave like button presses. The play/pause, stop, and record signals determine the functional state of the audio player. The different states of the system and how they transition to and from one another can be seen in the figures below.

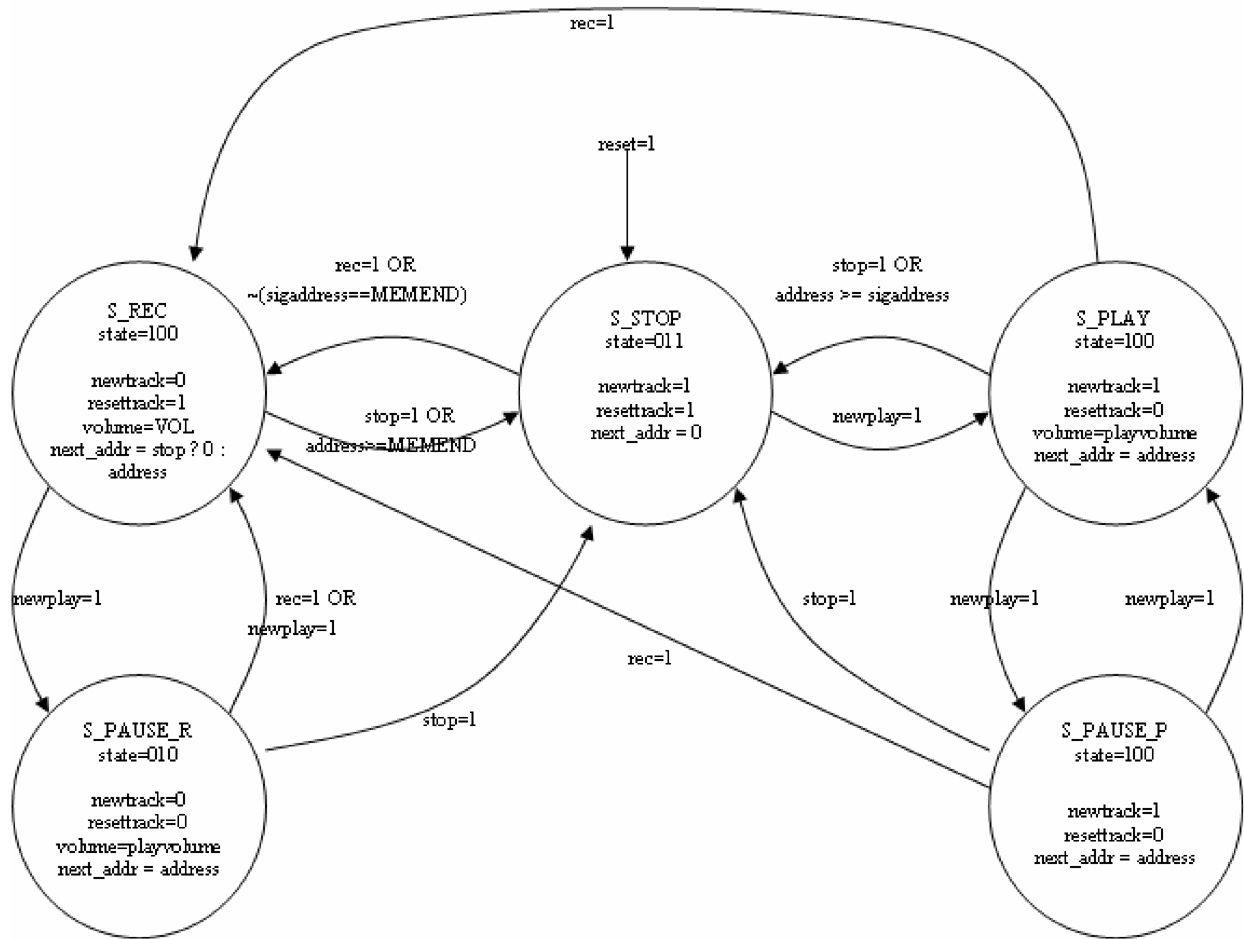

Figure 14: Audio Player Finite State Machine
The audio FSM determines which functional state the audio player is in.

Upon starting the audio player after the calibration stage, the audio memory is blank, so the only valid function to be implemented is record. The audio player thus remains in a reset stop state until rec is high. When the record button is "pushed," the audio player begins recording one track to the labkit's ZBT RAM and is in record state, S_REC. In record state, the audio fed into the FPGA is directly fed back out, so the user can hear what he/she is recording. From record state, the audio player can transition into S_PAUSE_R or S_STOP state. If the play/pause command is issued, the audio player pauses the recording and is in S_PAUSE_R state. From this state, the player can transition back to S_REC or to S_STOP. If play/pause is "pressed" again, the audio player begins recording in record state again without writing a new track. The audio player will remain in S_REC state if no signals are high until the audio memory is full. The player will then transition to the S_STOP state and be unable to record again. Otherwise, if the stop signal goes high in any state, the state also transitions to the S_STOP state. From here, the module can enter S_PLAY or S_REC state. If it enters record state again, the audio player begins recording after the last address to be written to and records a new track. If it enters the playback state, the audio player begins from the beginning of the recorded audio and plays back the stored audio data.

In playback state, there are several functions that can be carried out. The state transitions from S_PLAY are either to S_PAUSE_P, S_REC, or S_STOP. It enters S_PAUSE_P state when the play/pause signal is high and then goes back to S_PLAY from S_PAUSE_P when the play/pause signal goes high again. From S_PAUSE_P or S_PLAY, the audio player can transition to the S_REC or S_STOP states as well if the rec or stop signals, respectively, go high. However, record state can only be entered if the memory is not full yet. If the record state is entered, the audio player begins recording again from the last address recorded to and writes a new track. If the user issues no command, the audio player will remain in S_PLAY state until it reaches the memory's end, in which case, it will automatically transition to stop state.

The skip forward and back and volume up/down signals are actions that can be carried out when the audio player is in playback state. When in S_PAUSE_P or S_PLAY, whenever the skip forward or back signals are high, the track memory's address is incremented or decremented to the appropriate track. From this address of the track memory, a 19-bit value is read out, which corresponds to an address to read from the ZBT RAM, where the audio data is stored.

When volume up is signaled, the playvolume register is incremented so the volume is approximately 1.25 times as loud as it was before until it reaches a max volume represented by a 5-bit value of 31. If the volume down instead is signaled, the playvolume register decrements by about 0.75 until it reaches a minimum value of one. This register value is assigned to a volume output that is sent to the audio interface. When not in playback mode, the volume output is either not changed, as in S_STOP or S_PAUSE state, or returned to a default value of 16 in S_REC state.

Another function of that can be implemented in play state is the playback mode. The audio player module takes a two-bit mode value that is the lower two bits of the mode output from the mode FSM. This input tells the audio player module which playback mode the user has chosen, which then affects playback factors. For instance, when Alvin mode has been chosen, the sampling rate doubles by halving the value of pb_sample, and in Barry mode, the sampling rate decreases by 1.5 times by increasing pb_sample by 1.5 times. In addition, a shift register is assigned accordingly to allow for appropriate audio data interpolation as the sampling rate changes. In normal and echo modes, the sampling rate is one-fourth the ready signal rate, so data is sampled at 12 kHz. The mode value not only affects sampling rate but also affects what to output to the audio interface. When mode equals one, this means the audio player should be in echo mode, so the to_ac97_data is the superposition of the original interpolated signal added to two echo signals. Otherwise, in playback, the audio player feeds back only the one original interpolated audio signal.

4.2.2 Mode FSM

The mode finite state machine is a simple one. It takes the change mode signal, **md**, from the accelerometer/audio player interface module and uses this input to change state. The state machine has five states—a reset, normal, echo, Alvin, and Barry state—and so a 3-bit output signifying which mode has been chosen is given to the higher level to distribute to other modules, such as the audio player and its display module.

The FSM begins in reset state but once the change mode signal is high, the FSM transitions to the normal state. It then cycles through the states—normal to echo to Alvin to Barry and then back to normal—transitioning at every high change mode signal. The state transitions can be seen in the figure below.

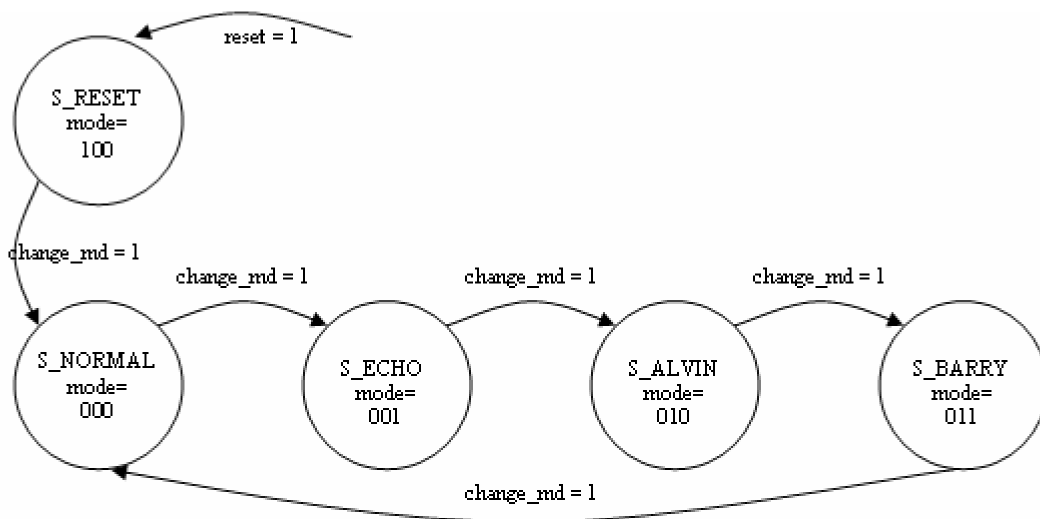

Figure 15: Mode Finite State Machine
The mode FSM transitions states every time the change_md input goes high.

4.2.3 ZBT RAM

To store the recorded audio data, the FPGA's ZBT RAM was used. This memory is 512Kx32, but only 20-bits were stored at each of the addresses. The 20-bit audio data was sampled at a rate of 12 kHz, so the memory filled up after around 48 seconds of recording.

During recording state, each time a counter counted from zero to SAMPLE-1, which is at a default of three, the address of the ZBT as well as the sigaddress register keeping track of the highest written address were both incremented by one. The counter was then reset to zero and the process was repeated again. The write enable goes high before the address is incremented to write the last sampled from_ac97_data. Since the from_ac97_data is 20-bits, but each address is 36-bits wide, the actual data written to the memory is the 20-bit sample with 16 zeros concatenated to the front.

During playback, the stored data is read at the set sampling rate. Whenever the counter reaches pb_sample, the address is incremented until it reaches the high address. The counter is then reset to zero and the count up begins again. The lower 20-bits of the data out from the RAM is stored as the outputdata, and the data stored at the last sampling is transferred to the oldoutputdata register. This allows for linear interpolation between the two samples, which rids of a lot of high frequency noise.

The ZBT RAM has a two-clock cycle delay when it reads and writes. However, since we feed it the 27 MHz clock and only write to the ZBT at 12 kHz and read from the ZBT at fastest, 24 kHz, a two 27 MHz clock-cycle delay is not an issue.

4.2.4 Track Memory

Along with the ZBT RAM, the audio player module interacts with several other smaller memory elements. One of these is a small 128x19 RAM. This RAM is used to store the beginning address of each recorded track to allow for the skip forward and skip back functions.

The RAM allows for 128 tracks to be recorded as it is 128 deep, and with only 48 seconds recording time, this track RAM should be sufficient. At each of these addresses, a 19-bit value can be stored. This 19-bit value is the address from the ZBT RAM which is written whenever a new track is written. Hence, the start address of each track is stored in this memory.

On each transition to record, the audio player checks if the value in the newtrack register is one or zero. If it is one, the track address, labeled track, and the highest track, labeled lasttrack, are incremented by one. In addition, the write enable signal wr is goes high to signal to the RAM to start writing the current ZBT address. The newtrack register only holds a high value if the audio player is transitioning from S_PLAY, S_PAUSE_P, or S_STOP state. The register is zero when transitioning from S_PAUSE_R so no new track is recorded when you stop pausing while recording.

When transitioning into play state from S_STOP or S_REC, the resettrack register holds a high value of one. This resets the track to track one, ensuring the audio player to start playing from the beginning of the recorded audio data. If the user pauses the system in during playback and enters S_PAUSE_P, though, the track is not reset when entering play state again.

In playback state, the current address being read from the ZBT RAM is compared to the data out from the track memory, stored in register track_out. Each time the address is greater than track_out + 18000, the track is incremented until it is past the last track. By incrementing in this manner, it allows for a more complex skip back system. If the current track is more than 18,000 addresses into the song, the skip back command brings the audio player back to the beginning of the track. Otherwise, it takes the user to the previous track. In normal playback mode, since the audio player samples at 12 kHz, 18000 addresses into the track is equivalent to 1.5 seconds into the current song.

4.2.5 Echo Memories

There are two other memories that also interact with the audio player. These are the two echo RAMs—echo RAM A and echo RAM B. Each RAM is 4096x20. The depth of the RAM determines how delayed the echo will be and the width of the RAM is the same as the bit-size of the audio data. The echo RAMs are both dual port RAMs so they one address can be written to while the next is read.

Both echo RAMs are written to and read from during playback state at every new audio frame. The data input to the first echo RAM is the interpolated signal of the original audio data halved in magnitude. Since the from_ac97_data is a PCM signal, the 20-bit represents the magnitude of the sample, so to create the echo, the interpolated input to the memory was divided by two. The second echo RAM took as its input the output from the first echo RAM. When the playback mode was chosen to be echo, the original interpolated signal, first echo signal, and second echo were superimposed by adding them together and sending the sum to the audio interface.

The echoaddr was incremented at every new frame during audio playback so the address given by the value of echoaddr was read from each echo RAM while the previous address was written to. By writing the current audio data to the previous address, a delay is created for the echo since that data will not be read until the 4096 counts later when the address counter wraps around. Since the address was incremented at a 48 kHz rate, each echo has a delay of 4096/48 kHz, which is about 100 ms. Hence, the first echo is about 0.1 ms after the original signal and

the second echo is about 0.1 ms after the first one. Using a small delay, the echo mode gives the sense of being in a large hall with the audio reverberating.

5 Interfacing the Accelerometer and Audio Player

5.1 Interface Module (Doris Lin)

Once the modules for the accelerometer and audio player have been created individually, the two parts must be interfaced so that the accelerometer controls the audio player. The interface module takes a 4-bit value from the accelerometer interface and transforms it into eight different one-bit signals to output to the audio player modules.

The interface was implemented using a case statement that looks at the 4-bit command input and then appropriately assigns one or zero to each of the eight outputs. The 4-bit command is the address of a RAM that stores the hand motion pattern during the calibration stage. When no longer calibrating, the accelerometer's transmissions are decoded and matched to the data stored in this RAM. Since there are nine positions calibrated, there are nine positions stored in the RAM. The command is the address of this RAM, and since the functions are calibrated in a fixed order, the address tells the interface which function should be implemented. The 4-bit command value and its corresponding function can be seen in Table.

Table 1: Command Values and Corresponding Function

Command Value	*Function*
4'b0000	Initial State
4'b0001	Play/Pause
4'b0010	Stop
4'b0011	Record
4'b0100	Skip Forward
4'b0101	Skip Back
4'b0110	Volume Up
4'b0111	Volume Down
4'b1000	Change Mode
Default case	Not Valid Command

The eight outputs of the interface are the signals play, stop, record, fwd, back, up, down, and md. The case statement assigns the command's corresponding function signal to a high one and all the other output signals to zero. For instance, if the command value is one, the play signal is assigned to one. If the command value is zero, all signals are set to zero, and if the command value is not in the range of 0-9, the signals are kept at their previous values. The interface assigns these signals to simulate a button being pushed, since most audio players respond to buttons being pressed to implement a function. A command value of 1-9 simulates a button being pushed while a command value of zero signifies a button being released. All other values of command lead the interface module and audio player to act as if no button has been either pushed or released.

The interface outputs play, stop, record, fwd, back, up, and down to the audio player module, which these signals to determine its state and which functions to carry out. The md signal is transmitted to the mode FSM and tells the module when to change states.

6 Testing and Debugging

6.1 Accelerometer Testing and Debugging (Diana Cheng)

In testing the accelerometer, it was rather difficult in it was hard to view since most enable signals were a pulse and also I didn't have an expected return value and so it was hard to see if it was correctly reading accelerations. Mostly the logic analyzer was used and only a few times were the test benches in Xilinx used because often it would work in the test bench but errors would occur in real life and so much of the debugging occurred in the real interaction with the chip. The first task was to get the accelerometer to respond to the transmitted commands. This was tested using the logic analyzer looking at transmission and receive lines and seeing what hex values were being transmitted. Also other ways to tests the communication was using HyperTerminal and secureCRT to compare values for things such as chip name acquisitions. These could then be compared to see if indeed the right data was being sent back. To test the FSM the states were displayed on the logic analyzer with the full data for one axis to see if the right values were being pushed into the right variables.

Testing the tilt calibrations required extensive use of the led lights as well as the hex display. Using the led's in the tilt, one light would blink every 2 seconds, one would show which state the FSM was in, and one would show which address they were writing. Then in decode state, the values read from the chip were displayed in the hex as well as the command that was being sent to the audio modules. This would let me know accurately if the module was working. When it wasn't working, extensive use of the logic analyzer allowed me to trace which signals were going wrong. At one point the memory would write one address off and thus decode the wrong command back to the audio player. This was a simple fix as looking at ramout and inputdata variables were very helpful in seeing what information was actually being read and written into memory.

To test the motion calibration, this was much more difficult as many of the ready signals were pulses and since many of the stages ran off of different divisions of the 27 MHz clock, it was hard to view over the logic analyzer. To debug this, the hex display was a huge help as it was able to expose 64 bits of information. Similar setup to the tilting debugging was used, focusing mostly on the led and hex display. Debugging in matlab was helpful as well for it would allow me to visualize the graphs and thresholding mistakes. The biggest problem was discovered just a few minutes before the presentation in that there was a large unexpected delay between the fulldata readies. This meant that the matlab values that were meant to optimize the signal processing were not valid in real life as the chip seemed to take longer then expected. It is hypothesized that this extra lag time comes from the long connection wires as well as the variable noise from the environment. Thus all the testing that occurred using the actual chip was down sampling too much and thus would never be able to decode due to lack of key features.

The logic analyzer was a big help in finding this error because I was able to toggle between different clocks using all 4 probe groups. Once this error was found, the led and hex lights were used to see if trial and error could find a combination of thresholds that would show something being decoded. I was able to show some motions being decoded in the end however

the amount of false positives and false negatives were horrendous. In matlab the optimal thresholds showed a 98.6% positive rate with 0 false positives however this was not the same for the actual system. Unfortunately, modeling the noise in matlab was not within my time limit and further testing was difficult beyond this point.

The overall biggest debugging error came from the timing of each signal as the hardware specifications were not accurately representing the entire system. I had introduced more delays using long wires and cables as well as using a 3.3 voltage instead of a higher voltage. Ways to fix this were to put more constraints on the user in how fast and slow they were allowed to make motions as well as how frequently they tried to send audio commands. Also the user was required to best mimic the time and motion they performed during calibration. However even with best efforts, decoding motion proved to be rather difficult.

6.2 VGA Testing and Debugging (Doris Lin)

The VGA display was straightforward in testing. Merely viewing the display on the screen, it was simple to tell if the VGA was displaying what it was supposed to. When creating the audio player display, I knew which shapes needed to be displayed, so testing began by generating smaller modules that drew those desired shapes, such as rectangles, circles, and triangles of various sizes. Each module was then synthesized and a bit file generated to display the shape on the VGA. Most of the modules worked as hoped, except the triangle modules at first drew pixels on unexpected lines of the frame. These discrepancies were all due to mathematical errors used in finding the boundaries of the triangle. By viewing the VGA display, it was easy to see which edge was incorrectly found, and after the math was double-checked for that edge, the triangle modules worked as desired.

Once the smaller modules were working, I instantiated them in larger modules to create the different buttons, which were then displayed onto the screen. I started by displaying one button, and when one button worked, I would instantiate another button to display to the screen. However, as more and more buttons were added to the display, more glitches appeared on the screen. I then took away some buttons to display only one button at a time to test if it was one particular sprite causing random pixels to be displayed across the screen. However, each individual button displayed clearly without glitches.

The problem with the sprites was that some of the calculations within the sprite modules created a long delay. For instance, the circle module multiplies several values together to ensure that the hcount and vcount values are within the area of the circle to assign that pixel a color value. This means checking $hcount^2 + vcount^2 <= RADIUS^2$ for each pixel in the frame for each circular sprite. In addition, there were many sprites instantiated in the audio display module so or-ing together all the pixels of the sprites just added to the combinational delay.

The button sprites were not the only elements adding to a large combinational delay. For both the calibration and audio player displays, the strings shown on the screen were especially prone to glitches. These glitches were likely caused by the delay of looking up characters in the font ROM and also due to multiple instantiations of the char_string_disp module, thereby instantiating several font ROMs.

Viewing the synthesis results, the combination delays for these modules were in the range of 18 to 20 ns, but the 65 MHz clock period is only around 15-16 ns. The long combinational delays were alleviated by latching the returned pixel values in the display modules and by splitting up the strings into smaller parts so fewer characters had to be looked up in the font

ROM in each char_string_disp instantiation. By latching the returned pixel values on the rising edge of the pixel clock and or-ing them together in groups, I was able to pipeline these values so they would be valid at the next rising clock. By doing so, the glitches in the displays were significantly reduced.

6.3 Audio Testing and Debugging (Doris Lin)

The audio portion was also rather straightforward to test, by listening, but less straightforward to debug. One of the major bugs when implementing the audio player was, although the audio player seemed to be reading and writing to the ZBT memory correctly, the audio quality in playback was terrible and filled with loud static. At first, the audio player was sampling at 6,000 kHz and was only storing the higher 8-bits of the from_ac97_data. I experimented by storing the full 20-bit from_ac97_data, thinking the lower eight bits mattered with the audio I was sending into the player. However, the audio quality improved little if at all. Next, sampling rates were increased so that I was writing and playing back data at 12,000 kHz, then 24,000 kHz, and even 48,000 kHz. At 48,000 kHz, though, the audio playback should have sounded just as it did in record state since this sampling rate equaled the rate of the ready signal generated by the AC97 chip, and I was essentially just feeding back exact audio input data as in record state. This was not the case; the audio was still filled with noise.

The next step was then to check to make sure the ZBT RAM was being written to properly. In order to test this, a ZBT test module was created in which during record state, the address currently being written to was writing the 19-bit address as its memory input data. In playback state then, the logic analyzer should have shown the address incrementing regularly and the output data from the RAM matching the address that was just read from. The logic analyzer, instead, showed the address incrementing at irregular intervals. Using the ready signal as the clock and a sampling rate of 12 kHz, the address should have incremented one every four ready signals. However, the address would increment sometimes every four and sometimes every five or six. The data out from the RAM was even more sporadic and would sometimes change even when the address had not. This irregular pattern suggested that the audio player was occasionally missing the ready signal on a rising clock edge.

Upon closer examination, it was spotted that the ready signal generated by the ac97 module was not synchronized with the 27 MHz clock. It was instead synchronized with the ac97 bit clock which runs at 12.288 MHz. By editing the ac97 module, the ready signal goes high at the rising edge of the 27 MHz clock instead of at the positive edge of the ac97 bit clock, thus synchronizing ready with the 27 MHz clock. By making this simple change, running the ZBT test module again, the addresses and data out all change at correct regular intervals and the data out corresponds to the address just read from. Testing the real audio player then, the sound out during playback was incredibly cleaner.

The basic functions of the audio player like play, pause, stop, record, and volume control were fairly simple to implement, but the echo mode and the track selection were more difficult to program because both of these involved memory elements. Once again, when these functions did not work, the logic analyzer was useful in debugging the audio player module, especially with the track selection. Using the logic analyzer, it was possible to see what addresses were being written to or read from, what the highest track was, etc. By viewing the behavior of the memories on the logic analyzer, I was able to appropriately adjust the address incrementing or decrementing.

As for the echo mode, the first implementation of the echo returned a very loud noisy audio signal. When I played back the echo alone without decreasing the magnitude, it sounded like the original audio except with a small delay. However, when the echo was halved in magnitude, either by dividing the ZBT output data by two or shifting right by one, loud static was heard. Several things were attempted to resolve this issue, but the solution was to use the interpolated audio rather than the audio outputted straight from the ZBT. The interpolated audio is essentially the output data from the ZBT without the high frequency noise, and using the cleaner audio signal cleaned up the echo sound as well.

7 Conclusion (Doris Lin and Diana Cheng)

The basic goals of our audio system were met. The accelerometer tilts are successfully recorded during calibration state and appropriately matched to a command after calibration. The audio player and accelerometers are then effectively integrated so that the audio and visual components respond correctly to the accelerometer's tilt. The audio player then successfully carries out the desired functions. It records, stops, pauses, and plays, and during playback, the volume can be increased or decreased and tracks skipped by using the appropriate hand gesture. However, there are several extensions that could be added to our audio player to improve the system and make it more robust.

8 Further Directions (Doris Lin and Diana Cheng)

On the accelerometer side, matching tilts were very successful, but the number of commands was limited by the number of different possible tilts. Pattern matching the waveforms of motions proved to be a hefty challenge in handling the tradeoff between robustness and accuracy. Using the accelerometer had a large learning curve because getting the chip to interface took half of the time allotted. It was difficult when the company didn't give accurate data sheets that would provide not enough, or sometimes false information. Given the time frame, this challenge of interfacing with hardware was a large part of the problem.

Beyond that, given the remaining timeframe, implementing the tilt was a more feasible plan as it had less variability. Once the data came from the chip, it was a matter of manipulating write enable signals and ramouts that could be matched later on. It was more a stretch to create the motion detection within the remaining timeframe however given maybe another week I believe optimal thresholds could have been set. As much of the signal processing back work required a lot of optimization and trial and error in motion calibrating.

Given the timeframe, I believe a more realistic project to mimic motion would have been to implement a lock tilt mechanism where a series of tilts that can almost be seen as movement, would correspond to a command. This lock-mechanism model is often times used in DSP in signal and pattern recognition and due to the inconsistent timing issues found with the lag time from the wires, this would be less dependent on the time and ability of the user to mimic their motions.

On the audio and visual components of the audio player, there are complexities that could be added to increase the attractiveness of the audio system. For one, altering the number of calls to the font ROM could decrease the amount of glitches in the display. As for future extensions, the visual display of the audio player could be more interactive and include a visualizer that

responded to the music being played. Another suggestion could be to display a visualizer in the middle of the display showing the motion of the user based on the x, y, and z data transmitted from the accelerometer.

For the audio, the second ZBT RAM could be added to work with the system to double the memory of the audio player and double the amount of audio that could be stored. The implementation of the track memory will have to be adjusted as well if this is carried out to also know which ZBT's address is being stored, the first ZBT RAM or second. Other further ideas could be to add more playback modes with more complex filtering such as phase shift, which would involve using the Fast Fourier Transform.

Our created audio system was a good starting point that lays down the ground work that can lead to many interesting future paths.