Christine Chen & Erika Lee
Digital Center of Innovation
6111 Massachusetts Ave.
Cambridge, MA 02139

December 9, 2010

Jane Porsche
2947 Wolfe Rd.
Bypass, MA 42949

Dear Ms. Porsche,

Enclosed is a proposal detailing plans for a virtual board breaking application to be created under our management at the Digital Center of Innovation.

We are requesting funding in the amount of $10,500.00 in total estimated costs for the period of December 9, 2010 through December 8, 2011.

Any questions regarding any aspects of the proposal may be directed to either of us. Thank you for your consideration of our proposal.

Sincerely,

Christine Chen & Erika Lee

Enclosure: Proposal

Virtual Board Breaking

Christine Chen, Erika Lee

December 9, 2010

Abstract

Board breaking is often used as an indicator of technical ability in many martial art curriculums. However, practicing board breaking can often be costly, averaging approximately one to two dollars per board. This document details a virtual board breaking application that can be used as an effective practice tool and presents only a one-time cost for unlimited board breaking. By providing both audio and visual feedback to the user, the application can help the user achieve consistency in board breaking technique. Furthermore, the tool supports board breaking in three different axes of orientation as well as varying degrees of difficulty to simulate varying board thickness. The proposed system design has been fully modeled and is ready for use.

Table of Contents

List of Figures……………………………………………………………………………..ii
1. Overview……………………………………………………………………………1
2. Description………………………………………………………………………….3
3. Testing and Debugging……………………………………………………………..9
4. Conclusion……………………………………………………………………...12
5. References………………………………………………………………………....13
Appendix A………………………………………………………………………...14
Appendix B…………………………………………………………………………21

List of Figures

Figure 1. DE-ACCM3 and AD7824 Chip Diagrams…………………………………………...1
Figure 2. IR beam setup and Accelerometer and Glove Setup…………………………………....1
Figure 3. Board Orientation Possibilities……………………………………………………….....2
Figure 4. Full Block Diagram……………………………………………………………………..4
Figure 5. ADC Read Timing Diagram in Mode 0………………………………………………...3
Figure 6. Acceleration Waveform During a Punch……………………………………………......6
Figure 7. Punch State Transition Diagram………………………………………………………..6
Figure 8. Punch Waveform Force Threshold Determination……………………………………..7

1. Overview

The system is comprised of two main functional components: the hardware component and the user interface component as described below.

The main purpose of the hardware is to gather data from the user's punch. Involved in this process are a three-axis accelerometer (DE-ACCM3) from Dimension Engineering attached to a taekwondo glove, an analog to digital converter chip (AD7824) from Analog Devices, and an IR beam setup, all pictured below. The accelerometer attached to a taekwondo glove is pictured below as well.

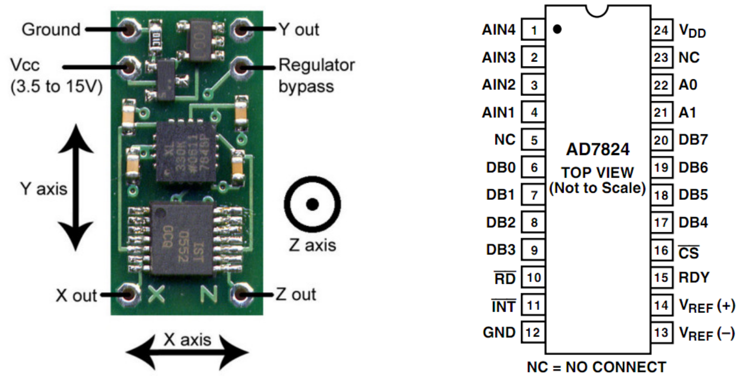

Figure 1. (left to right) DE-ACCM3 and AD7824 chip diagrams

Figure 2. (left to right) IR beam setup and accelerometer chip attached to taekwondo glove

The data gathered by the accelerometer and run through the ADC determines the magnitude of the force of the user's punch, and the IR beam indicates where the virtual board would physically be. Although the system continuously samples data from the accelerometer, it only runs force threshold comparisons once it detects that the IR beam has been broken, indicating a board break attempt. Using switches, the user can also choose a board orientation and difficulty level, which will alter which data channel the ADC will sample from as well as the force magnitude threshold for a board break.

The user interface portion of the application consists of graphical and audio functionality. Coupled with the labkit AC97 codec chip, the audio module plays a recorded sound of a board breaking that is stored in BRAM when the virtual board is broken.

Overall, the graphics module coordinates the display of virtual board images, a force gauge, a difficulty reading, and an instructions page. Three orientation settings and four difficulty levels exist for the board image, with the three orientation possibilities shown in the figure below. In the y and z axes, the number of boards that appear on the screen correspond to the difficulty level. The difficulty reading indicates what difficulty level the system is currently set to (from 1 through 4). The force gague, a rectangular bar which length is proportional to the force magnitude of the punch appears in the lower left-hand corner of the screen at the end of a strike. Toggling switch #6 on the labkit swaps the screen to and from an instruction page that explains the application controls.

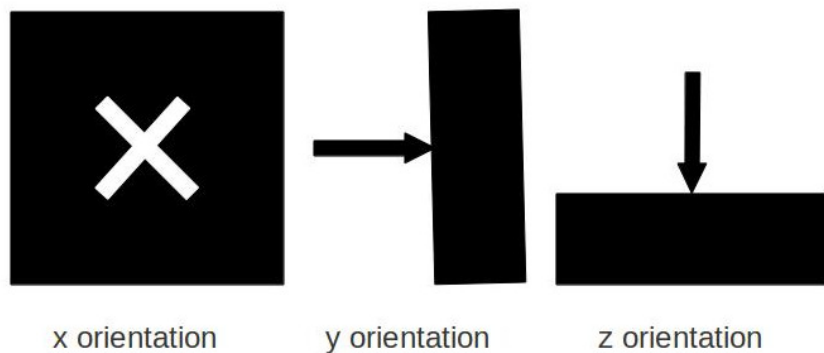

Figure 3. Board Orientation Possibilities

2. Description

Both the hardware and user interface components are comprised of multiple subsystems that are interlinked together to form the full application. The various linkages and modules are displayed in Figure 4 (on next page).

2.1 Hardware Modules (Christine)

2.1.1 ADC Control Module (Christine)

The ADC control module is used to manipulate the control signals for the analog to digital converter (ADC) chip in sampling the data coming from the three-axis accelerometer. To initiate data conversion from analog to digital, the module sets the chip select and read signals to low, as seen in timing diagram below. When the data conversion is complete, the ADC sets the interrupt output signal to low to indicate that the digitized data is valid and ready for use. In addition, the ADC contains four input data channels but can only convert data on one channel at a time.

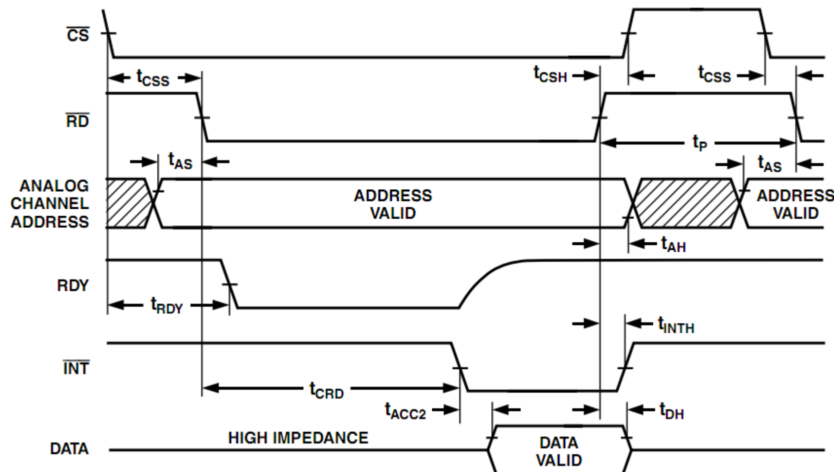

Figure 5. Read Timing Diagram in Mode 0 [1]

Using a 10 kHz clock, the module controls the ADC to continuously sample data from the accelerometer at 3.33 kHz. Other than the 10 kHz clock, inputs to the module include the desired board orientation and the ADC interrupt signal. It uses the desired board orientation to determine which data channel from which the ADC should sample and the interrupt signal to determine when the ADC is ready to sample again. The outputs of the module are the channel address from which the ADC is to sample and the chip select and read signals for sampling initiation.

Clocking the module at 10 kHz satisfies all the ADC timing conditions since 100µs is larger than the conversion time (t_{CRD}) of 2µs indicated on the datasheet, which is the longest timing constraint the module needs to satisfy.

The module takes three clock cycles to read one data point from the accelerometer. In the first clock cycle, it sets the chip select and read signals to low; by the second clock cycle, the interrupt signal is now low, and the module sets an internal ready signal to a high to indicate

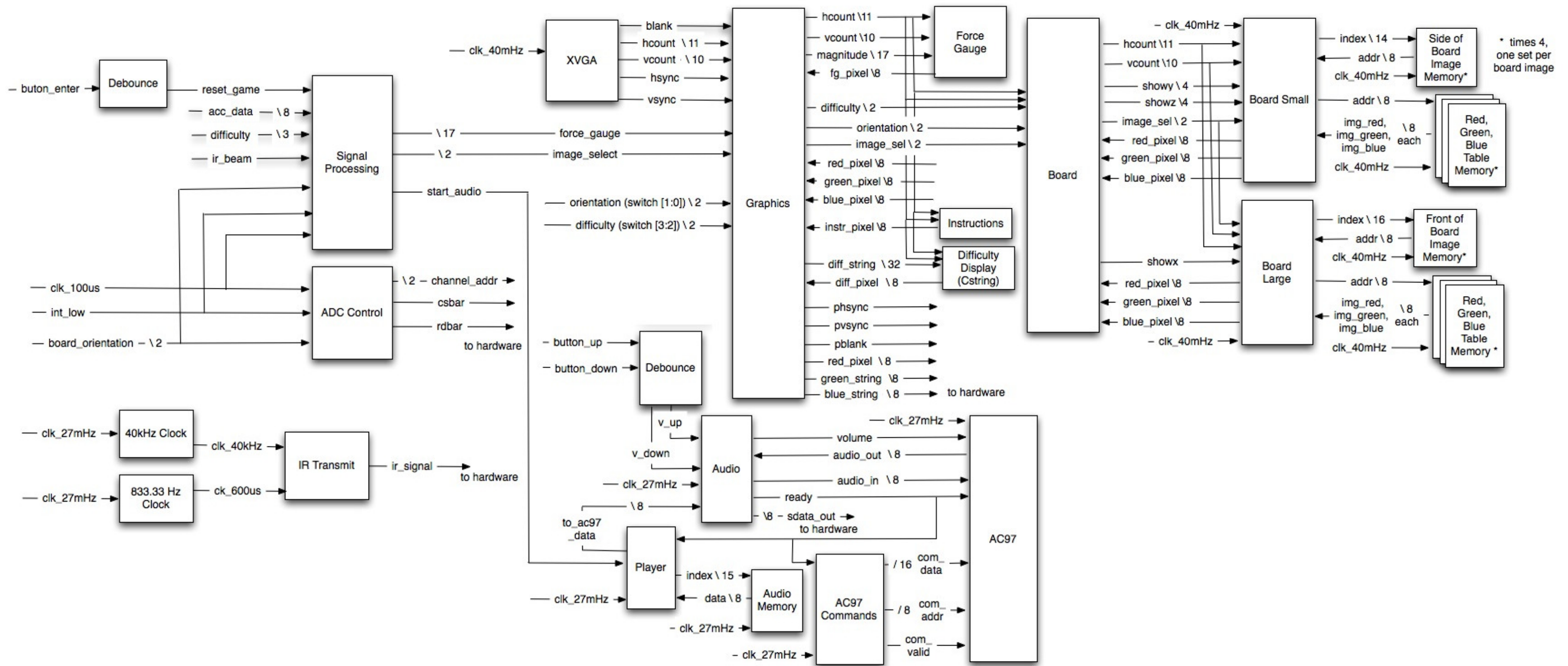

Figure 4. Full System Block Diagram

that a new read cycle can be initiated. During the third clock cycle, the module sees that the ready signal is a high and resets the chip select and read signals to high. On the next clock cycle, the module sets the chip select and read signals to low, re-starting the data conversion cycle.

2.1.2 40kHz, 833Hz, and IR Transmit Modules (Christine)

The 40 kHz and 833 Hz modules are used to slow down the internal 27 MHz clock to generate the necessary square waves for creating the IR transmitter signal. Both clock divider modules take in the 27 MHz clock as their only inputs and output the necessary square wave by inverting the output signal when the internal counters reach 336 and 16199 respectively for 40 kHz and 833 Hz signals.

The IR transmit module takes in both square wave signals outputted by the 40kHz and 833Hz clock modules and produces the transmission signal by and-ing together the two square waves. This is equivalent to a 40 kHz signal being transmitted when the 833 Hz signal is high and a continuous low signal when the 833 Hz signal is low.

2.1.3 100µs Clock Divider Module (Christine)

This clock divider module provides a high pulse every 100µsthat is used by the ADC control and signal processing modules. Dividing down from the internal 27 MHz clock, the module outputs a one every time its internal counter has reached 2699 before resetting.

2.1.4 Signal Processing Module (Christine)

To determine when a punch has been thrown and if the force of the punch is large enough to break the board, the signal processing module analyzes the data coming from the accelerometer via the ADC. Inputs to the module include the 100µs clock signal, the ADC interrupt signal, digitized data from the ADC, the desired board orientation, the desired difficulty level, the reset game signal, and the IR beam signal; outputs are the audio playback signal, the image select signal, and the force magnitude of the punch thrown.

Each time new data is ready from the ADC, the module updates a circular array that represents the acceleration data for the last 150ms as well as the force magnitude variable that represents the sum of all the acceleration data points in the circular array. The circular array is just large enough to encompass the last 150ms because it was empirically determined that 150ms was the average punch length using waveforms generated with an oscilloscope, such as the one below. Acceleration data is continuously sampled and saved regardless of the current punch status.

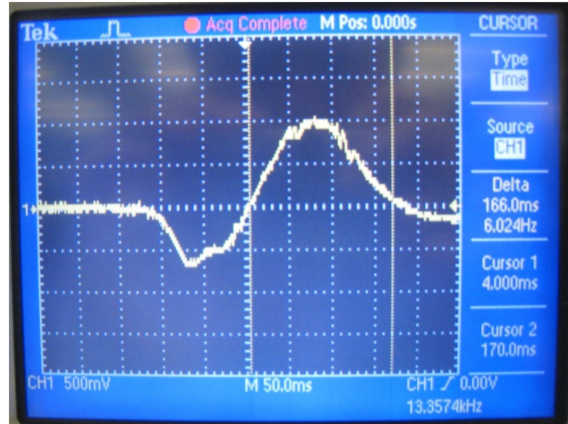

Figure 6. Acceleration Waveform during a Punch

At the heart of the module is the state machine regarding the punch status, which is depicted in the diagram below; the possible states are (1) the system idling waiting for a punch, (2) the system recognizing a punch and processing data, and (3) the system idling after the punch has completed. When the system detects that the IR beam has been broken for more than the standard 600μs envelope, it determines that a board break is being attempted and moves from the first state to the second state. The application concludes that a punch has ended when the acceleration detected has been within a small delta of the resting value of the accelerometer for approximately 1ms. By pressing the reset button or altering the board orientation or difficulty level, the user can reset the state machine back to the first initial waiting state and also return the variables used in the state machine to their default values for that particular board orientation and difficulty level.

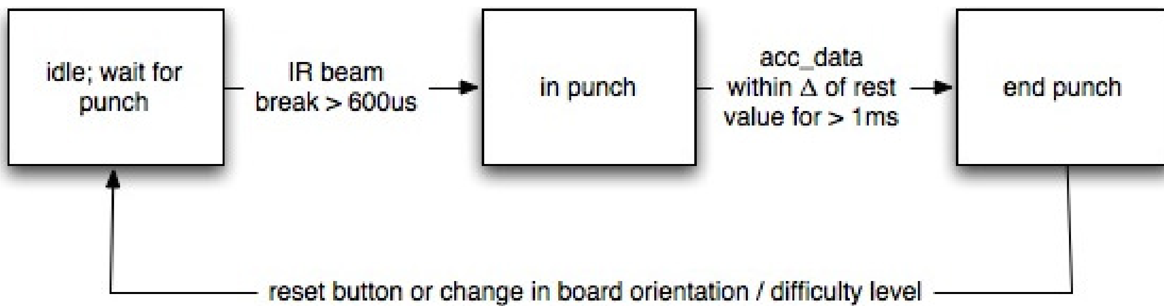

Figure 7. Punch State Transition Diagram

When in the initial idle state waiting for a punch, the system calculates the threshold values for the force magnitude and maximum acceleration reached to break the board for a particular board orientation and difficulty level. These threshold values are dependent on the board orientation since the resting value of an axis on the accelerometer depends on the accelerometer's orientation. Another subtlety of threshold calculations based on the orientation of the accelerometer is how the data values change in the desired direction of motion. In the x axis and z axis board orientations, the acceleration data values actually decrease from the resting value in the desired punch direction whereas in the y axis board orientation, the acceleration data values increase from the resting value in the desired punch direction. These inconsistencies are taken

into account by the system when calculating the current force magnitude sum and maximum acceleration during a punch.

During the punch, the system continually compares the current calculated force magnitude of the punch so far with certain threshold points to determine what board image to display: unbroken, slightly cracked, or more cracked. The slightly cracked board image is displayed if the current maximum calculated force magnitude and maximum acceleration detected is more than half of the minimum force threshold and maximum acceleration calculated in the waiting punch state respectively. The more cracked board image is displayed if those two values are more than three quarters of their respective minimum and maximum board break values. If either of the cracked images is to be shown, the board break sound is triggered for playback.

After the punch has ended, the system checks if the fully broken board image should be shown by comparingthe overall force magnitude of the punch and maximum acceleration achieved with the calculated necessary force and acceleration threshold values calculated in the punch waiting state.

In the background, the module also checks the output of the IR receiver every 100µs to determine how long the output is a high signal during the punchto determine if the beam has been broken by a punch. Furthermore, the module continuously creates the audio playback trigger signal by and-ing together the inverse of the delayed audio state and the current audio state.

The force magnitude passed to the force gauge module is the maximum force magnitude calculated during a punch. Due to the nature of how the force threshold is calculated, as demonstrated in the figure below, the maximum force magnitude calculated is used for display and comparison purposes since that value represents the most centered segment of punch data, regardless of how long the punch lasts.

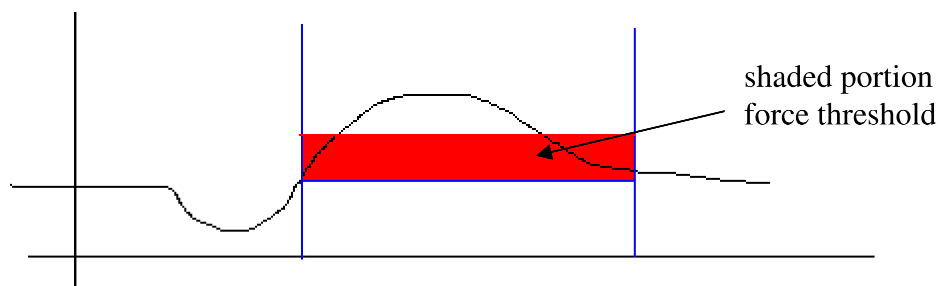

Figure 8. Punch Waveform Force Threshold Determination

2.2 User Interface Modules (Erika)

2.2.1 Button Debounce and Audio Debounce (6.111 staff)

When a button or switch is pressed, the metal contact may open and close, or "bounce," very quickly several times before settling at its new value. In order to eliminate these small

oscillations, the debounce module only allows signals that have been stable for at least approximately 1 millisecond in order to pass through to the rest of the system.

2.2.2 XVGA Module (6.111 staff – with modification)

The XVGA module handles proper generation and synchronization of the signals necessary to specify pixel color values displayed on the monitor. Its output signals, including hcount, vcount, hsync, vsync, and blank, are passed to the graphics module. Several modifications regarding clock speed were made to the original code for the purposes of this project.

2.2.3 Graphics Module (Erika)

The main graphics module of the virtual board break outputs the proper XVGA signals (hsync, vsync, blank, and red, green and blue pixel color values) to display pixel color on screen. To generate a screen resolution of 800x600 pixels, the entire graphics component of the application is clocked at 40MHz. The module is comprised of several major sections described below.

2.2.4 Force Gauge Module (Erika)

The force gauge module displays a bar in the lower left-hand corner of the screen when the punch finishes. The length of the bar that appears is proportional to the input magnitude signal, and the bar is colored on a gradient from gray to white.

Determining the gauge's position, the x and y inputs identify the upper left-hand corner of the force gauge,. The value of the output pixel signal determines the brightness of a particular pixel identified by the hcount and vcount input values.

2.2.5 Cstring Module (6.111 staff – with modification) and Instruction Module (Erika)

The cstring module displays the text used by this application. Using character ASCII encodings and the pixel values in BRAM memory, the module determines the value of the pixel output for a set of hcount and vcount values. One instance of this module is created for every line of text displayed.Several modifications were made to the original Verilog for the purposes of this project.

Comprised of several instances of the cstring module, the instructions module displays the instruction page for application controls.

2.2.6 Board Module (Erika)

The board module is a wrapper module for the board_small and board_large modules. The module takes in an orientation, a difficulty setting, and how cracked the displayed the board image is. These input values determine which board images to display at a given time, and the module outputs the red, green, and blue color values for a given pixel.

Both the board_large and board_small modules store image data in BRAM. Each image requires four BRAM: three for the red, green, and blue color tables, and one containing the table of pixel-

index encoding. The board images use eight different colors, one encoded by each row of the three color tables. Each table index maps to a row in the color table. The red, green, and blue values in that row represent the color of the pixel in that position on the image.

The BRAM modules were created with .coe source files, generated with a MATLAB script provided by the 6.111 staff.

2.2.7 Board_Large Module (Erika)

The board_large module displays the full board image for the board break in the x axis. The left half of the board image is stored in BRAMwhich is then flipped and duplicated on the right in order to display a full board. Duplicating the image in this manner halves the amount of memory necessary to store this image. The dimensions of the half board image are 256 x 128 pixels.

2.2.8 Board_Small Module (Erika)

The board_small module displays the side of the board image for the board break in the y and z axes. The same image is duplicated for both axes, with the image indexes adjusted to flip and rotate the image. For these two axes, increasing the difficulty also adds additional board images to the screen. By adjusting the image indexes, only one instance of the BRAM is necessary, regardless of how many boards are displayed on screen. The dimensions of the image used are 138 x 64 pixels.

2.2.9 Audio, AC97 Command, and AC97 Modules (6.111 staff)

Clocked at 27MHz, the audio module is a wrapper module for the AC97 and AC97 command modules. The ready output signal indicates that a new audio sample is ready. The AC97 command module cycles through a sequence of command values that are output to the AC97 module. Interfacing with the labkit's AC97 audio codec chip, the AC97 module has both input and output ports for audio data.

2.2.10 Player Module (Erika)

The player module outputs the signal for a board break sound when the input playback signal pulse is high. This pulse occurs when a board is broken, thus initiating the playback sound. The output of this module is passed to the audio module to output to the FPGA speaker. This module is clocked at 27MHz.

The BRAM was created with a .coe source file,generated with a MATLAB script provided by the 6.111 staff. Taking in a .wav file, the script output a .coe file with the file's first 10000 audio samples.

3. Testing and Debugging

3.1 Hardware Modules (Christine)

Using the logic analyzer, I verified the correct manipulation of the control signals for the ADC as well as the correctness of the sampled data using the resting values given in the accelerometer datasheet [2].

For the signal processing module, correct state machine transitions were confirmed using the logic analyzer as well as the hex display on the labkit. The hex display showed the force magnitude calculated for a user's punch as well as the maximum acceleration detected. These values were used to determine that the correct image transitions were being made as well. Because the audio signal trigger is a pulse lasting only 100µs, the logic analyzer was utilized to affirm correct audio signal playback.

The punch detection process initially did not include the use of the IR beam to indicate board impact. Rather, the state machine was used to detect that the acceleration data deviated at least a certain delta from the resting value for a pre-determined period of time. This pre-determined period of time was to be longer than a well-placed tap to the accelerometer or a flick of the glove. Nonetheless, the challenge of truly determining when a punch was thrown was alleviated using an IR beam to denote the plane of the board, which also had the added benefit of giving something for the user to aim at.

Another error that took a while to figure out was how the accelerometer data changed in the plane of motion with the accelerometer orientation on the glove. Initially, I had been operating under the assumption that the accelerometer data values would increase from the resting value when the punch was attempted. However, for the x and z axes board orientations, the punching motion produced accelerometer data values that decreased from the resting values. Thus, the necessary modifications were made to the Verilog code to accommodate this unexpected behavior.

3.2 User Interface Modules (Erika)

An issue that came up with the board images was insufficient memory on the FPGA. At first, the board module was structured such that there were a total of nine board submodules (one for the x axis, four for the y axis, and four for the z axis). Each of these submodules contained an instance of the read-only BRAM with the image data. I had not realized that declaring a new instance in each module would write eight copies of the BRAM with data for the image of the side of board, when only one was necessary. Compiling this code revealed that the labkit's memory was insufficient to store all of these BRAM instances.

This problem was fixed by restructuring the board image modules. In the new design, the eight board submodules for the y and z axes were substituted with one Small Board module, which determined which board images to show with only one copy of the BRAM with stored image data. The new implementation compiled without memory issues.

The audio module's output originally remained unchanging when the virtual board was broken. This bug was a result of the audio player handling the play signal incorrectly. Instead of playing the entire sound data on a high pulse playback input, the module would output the sound data only while the playback signal was high. The sound was only being played for one clock cycle and was thus completely inaudible. A play_on_off flag, which toggled on when the pulse arrived and off when the sound data reached its end, was added to the player module. After this modification, the player module handled the playback pulse properly.

4. Conclusion

The main purpose of the system proposed is this document is for use as a board breaking practice tool for various martial arts curriculums. Allowing for unlimited breaks, the application provides feedback to the user about the force magnitude of the punch to help build consistency in board breaking technique. The application also presents flexibility to the user in the board orientation and the difficulty level, which can be used to simulate board thickness.

To improve upon the system prototype described here, modifications would include upgrading the hardware for wireless functionality, displaying a force versus time graph as part of the user interface, and being able to save individual user data for comparison purposes. Wireless functionality could be implemented with an RF system and would allow for greater range of motion and increased flexibility for portability. Providing increased user feedback, the force versus time graph gives the user the ability to track the acceleration of their punch in the temporal dimension. Being able to save user data gives users the opportunities to track their long-term progress and improvement.

Currently, even without these modifications, a working prototype has been modeled and is ready to be implemented as is.

5. References

[1] Analog Devices, "LC^2 MOS High Speed 4- and 8-Channel 8-Bit ADCs", AD7824/AD7828 datasheet

[2] Dimension Engineering, "DE-ACCM3D Buffered ±3g Tri-axis Accelerometer" DE-ACCM3D datasheet

Appendix A. Hardware Modules Verilog Code

```
//this module creates a 40kHz square wave for the IR beam
//by dividing down the internal 27mHz clock
module IR_40khz (input clk_27mhz, output reg clk_40khz);

        reg [9:0] count, next_count;
        wire signal_40khz;

        always @(posedge clk_27mhz) begin
                //every 12.5us, the value of clk_40khz is flipped to create the square wave
                if (signal_40khz) clk_40khz<=~clk_40khz;
                count<=next_count;
        end

        always @* begin
                //counts to 336, which is ~12.5 us then resets
                next_count<=(count==336) ? 0: count+1;
        end

        //goes high every 12.5 us
        assign signal_40khz=(count==336);

endmodule

//this module creates a 833Hz square wave for the IR beam
//by dividing down the internal 27mHz clock
module IR_600us (input clk_27mhz, output reg clk_600us);

        reg [14:0] count, next_count;
        wire pulse_600us;

        always @(posedge clk_27mhz) begin
                //every 600us, clk_600us switches value to create the square wave
                if (pulse_600us) clk_600us<=~clk_600us;
                count<=next_count;
        end

        always @* begin
                //counts to 16199 which is 600us before resetting
                next_count<=(count==16199) ? 0: count+1;
        end

        //goes high eveyr 600us
        assign pulse_600us=(count==16199);

endmodule

//this module creates the IR transmission signal
module IR_transmit (input clk_40khz, input clk_600us, output signal);

        //signal just and-ed version of 40kHz and 833Hz square waves
        assign signal = clk_600us & clk_40khz;

endmodule
```

```
//this module goes high every 100us
module clock_100us (input clk_27mhz, output clk_100us);
		reg [12:0] count, next_count;

		always @(posedge clk_27mhz) begin
				count<=next_count;
		end

		always @* begin
				//counts to 2699 which is 100us before resetting
				next_count=(count==2699) ? 0:count+1;
		end

		//goes high every 100us
		assign clk_100us=(count==2699);

endmodule

//this module manipulates the ADC sampling of the accelerometer data
module adc_control (input clk_10us, input [1:0] board_orientation, input int_low,
		output reg [1:0] channel_address, output reg cs_bar, output reg rd_bar);

		reg ready=0;
		//local constants for the channel address of the three axes
		localparam X_AXIS=2'b00;
		localparam Y_AXIS=2'b01;
		localparam Z_AXIS=2'b10;
		localparam NA=2'b11;
		//local constants for the board orientation
		localparam HOR_THICK=2'b00;
		localparam VERT_THIN=2'b10;
		localparam HOR_THIN=2'b01;

		//occurs every 100us
		always @(posedge clk_10us) begin
				//if ready==0 then set chip select and read signals active low
				if (!ready) begin
						cs_bar<=0;
						rd_bar<=0;
				end
				//if ready==1 then data conversion done, set chip select and read signals back to high
				else begin
						cs_bar<=1;
						rd_bar<=1;
				end
				//if data is ready, then flip ready
				//***NOTE*** it flips ready signal twice every read cycle
				//first flip of ready to high when data initally complete
				//second flip of ready back to low when chip select and read signals set back to high
				if (!int_low) begin
						ready<=~ready;
				end
		end

		//chooses the data channel to look at based on the board orientation
		always @* begin
```

```
                case (board_orientation)
                        HOR_THICK: channel_address<=X_AXIS;
                        VERT_THIN: channel_address<=Y_AXIS;
                        HOR_THIN: channel_address<=Y_AXIS;
                        default: channel_address<=NA;
                endcase
        end

endmodule

module signal_process (input clk_100us, input int_low, input [7:0] acc_data, input [1:0] board_orientation,
    input [2:0] difficulty, input reset_game,  input ir_beam, output reg [2:0] image_select,
    output reg [16:0] force_gauge, output start_audio, output reg [7:0] max_acc,
    output reg [1:0] punch_state, output reg [7:0] rest_value, output reg [3:0] wait_count,
    output reg [15:0] max_delta, output reg [23:0] min_force, output reg [15:0] current_delta,
    output reg which);

    reg [7:0] save_data[511:0];     //circular data array for accelerometer data
    reg [1:0] board_old;            //delayed version of board orientation
    reg [2:0] diff_old;             //delayed version of the difficulty level
    reg [8:0] index;                //which index of circular array changing
    reg [4:0] done_count;           //how long has it been since punch complete?
    reg [16:0] force_gauge_inter;   //maximum force sum seen for the punch so far
    reg [16:0] force_gauge_sum;     //sum of all data points in array
    //audio signal variables
    localparam ON=1'b1;
    localparam OFF=1'b0;
    reg old_audio;
    reg new_audio;
    //local constants for punch state machine
    localparam GET_READY=2'b00;
    localparam PUNCH_WAIT=2'b01;
    localparam IN_PUNCH=2'b10;
    localparam PUNCH_DONE=2'b11;
    reg [7:0] MAX_ACC;
    //local constants for board orientation
    localparam HOR_THICK=2'b00;
    localparam VERT_THIN=2'b10;
    localparam HOR_THIN=2'b01;
    //local constants for image select
    localparam UNBROKEN=2'b00;
    localparam CRACKED=2'b01;
    localparam MORE_CRACKED=2'b10;
    localparam BROKEN=2'b11;
    //other local useful constants
    localparam TOTAL_DATA=512;
    localparam NUM_LEVELS=4;
    localparam REST_VALUE_x=85;
    localparam REST_VALUE_y=95;

    //only occurs when int_low goes low b/c that means new and valid data
    always @(negedge int_low) begin

        //saving delayed version of signals
        old_audio<=new_audio;
        diff_old<=difficulty;
```

```
board_old<=board_orientation;
//updating circular array and time continuous variables
save_data[index]<=acc_data;
index<=index+1;
force_gauge_sum<=force_gauge_sum+acc_data-save_data[index];

//restores default values for variables when game is reset
//due to press of reset_game button or changes in board orientation/difficulty
if (reset_game || board_old!=board_orientation || diff_old!=difficulty) begin
    punch_state<=PUNCH_WAIT;
    image_select<=UNBROKEN;
    force_gauge_inter<=0;
    force_gauge<=0;
    current_delta<=0;
    new_audio<=OFF;

    case (board_orientation)
        VERT_THIN:      begin
                            MAX_ACC<=0;
                            which<=1;
                            max_acc<=255;
                            rest_value<=REST_VALUE_x;
                        end
        HOR_THIN:       begin
                            which<=0;
                            max_acc<=0;
                            rest_value<=REST_VALUE_y;
                            MAX_ACC<=255;
                        end
        HOR_THICK:      begin
                            which<=1;
                            max_acc<=255;
                            rest_value<=REST_VALUE_x;
                            MAX_ACC<=0;
                        end
        default:        begin
                            rest_value<=0;
                            MAX_ACC<=0;
                        end
    endcase

end
else begin
    //state machine for punch data
    case (punch_state)
        PUNCH_WAIT: begin //idling state when waiting for punch to occur
                            //calculates threshold force magnitude and max accelerations for board
                            if (!which) begin
                                min_force<=difficulty*(MAX_ACC-rest_value)*64;
                                max_delta<=difficulty*(MAX_ACC-rest_value)/NUM_LEVELS;
                            end
                            else begin
                                min_force<=difficulty*(rest_value-MAX_ACC)*64;
                                max_delta<=difficulty*(rest_value-MAX_ACC)/NUM_LEVELS;
                            end
                            //if ir beam broken, then change state
```

```
                if (wait_count>=10) begin
                    punch_state<=IN_PUNCH;
                end
            end
IN_PUNCH: begin
            //updates maximum acceleration value seen and other force gauge sums
            //if max acceleration and force gauge sums meet threshold, then change image
            //and turn on audio
                if (!which) begin
                    if (acc_data>max_acc) begin
                        max_acc<=acc_data;
                    end
                    current_delta<=(max_acc>=rest_value) ? (max_acc-rest_value):0;
                    force_gauge <= (force_gauge_inter>force_gauge) ?
                        force_gauge_inter:force_gauge;
                    force_gauge_inter<=force_gauge_sum>(TOTAL_DATA*rest_value) ?
                        force_gauge_sum-TOTAL_DATA*rest_value:0;
                    if (current_delta>(max_delta/4*3) &&
                        force_gauge_inter>=(min_force/4*3)) begin
                        image_select<=MORE_CRACKED;
                        new_audio<=ON;
                    end
                    else if (current_delta>(max_delta/2) &&
                        force_gauge_inter>=(min_force/2)) begin
                        image_select<=CRACKED;
                        new_audio<=ON;
                    end
                    else image_select<=UNBROKEN;
                end
                else begin
                    if (acc_data<max_acc) begin
                        max_acc<=acc_data;
                    end
                    current_delta<=(max_acc>=rest_value) ? (max_acc-
                        rest_value):rest_value-max_acc;
                    force_gauge <= (force_gauge_inter>force_gauge) ?
                        force_gauge_inter:force_gauge;
                    force_gauge_inter<=force_gauge_sum>(TOTAL_DATA*rest_value) ?
                        force_gauge_sum-TOTAL_DATA*rest_value
                        :TOTAL_DATA*rest_value-force_gauge_sum;
                    if (current_delta>(max_delta/4*3) &&
                        force_gauge_inter>=(min_force/4*3)) begin
                        image_select<=MORE_CRACKED;
                        new_audio<=ON;
                    end
                    else if (current_delta>(max_delta/2) &&
                        force_gauge_inter>=(min_force/2)) begin
                        image_select<=CRACKED;
                        new_audio<=ON;
                    end
                    else image_select<=UNBROKEN;
                end
                //if near resting value then increment done count
                if (acc_data>=(rest_value-15) && acc_data<=(rest_value+15)) begin
                    done_count<=done_count+1;
                end
```

```
                                    else begin
                                        done_count<=0;
                                    end
                                    //if near resting value for more than 1ms then punch must be done so move
                                    //into next state
                                    if (done_count>10) begin
                                        punch_state<=PUNCH_DONE;
                                        done_count<=0;
                                    end
                                end
                PUNCH_DONE: begin
                                //final max acceleration and force gauge check for final image reveal
                                    if (current_delta>=max_delta && force_gauge_inter>=min_force) begin
                                        image_select<=BROKEN;
                                    end
                                end
                default:        begin
                                end
            endcase
        end
    end

    //determine ir beam break
    always @(posedge clk_100us) begin
        //normal ir beam operation indicates that maximum wait_count reached is only 6
        //counts how long a "1" has been read by the receiver --> continuous "1" if ir beam broken
        if (ir_beam) begin
            wait_count<=wait_count+1;
        end
        else wait_count<=0;
    end

    //start_audio pulse created by and-ing together inverse of delayed audio and current audio state
    assign start_audio=(~old_audio&(new_audio));

endmodule
```

Appendix B: User Interface Modules (Erika)

```
//////////////////////////////////////////////////////////////////////////////
//
// graphics:

// contains graphical elements of the board rbeak application
//
//////////////////////////////////////////////////////////////////////////////

module graphics (
  input vclock,                          // 40MHz clock
  input reset,                           // 1 to initialize module
  input [1:0] image_sel,                 //select image to display
  input instr,
  input [1:0] orient,
  input [1:0] difficulty,
  input [16:0] magnitude,                // length of force gauge
  input [10:0] hcount,                   // horizontal index of current pixel
  input [9:0]     vcount,                // vertical index of current pixel
  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)
  output reg phsync,                     // graphic's module's horizontal sync
  output reg pvsync,                     // graphic's module's vertical sync
  output reg pblank,                     // graphic's module's blanking
  output reg [7:0] red_pixel,
  output reg [7:0] green_pixel,
  output reg [7:0] blue_pixel);

        wire [7:0] force_pixel, pixelr, pixelg, pixelb, cdpixel, instr_pixel;

        wire [103:0] cstring = "difficulty: ";

        reg [7:0] diff_value_str;



        //module to display force gauge
         force_gauge #(.HEIGHT(64))  gauge(.x(10),.y(500),.hcount(hcount),.vcount(vcount),
        .pixel(force_pixel), .magnitude(magnitude));



        //module to display board images

        board_images bi(.hcount(hcount),.vcount(vcount), .clock(vclock), orient(orient), .difficulty(difficulty),
        .image_select(image_sel), .pixel_red(pixelr), .pixel_green(pixelg), .pixel_blue(pixelb));

        //module to display difficulty
        char_string_display cd(.vclock(vclock),.hcount(hcount),.vcount(vcount), .pixel(cdpixel),.cstring({cstring,
        diff_value_str, 144'b0}),.cx(11'd400),.cy(10'd50));


        //module to display instructions
```

```
	instructions i(.vclock(vclock),.hcount(hcount),.vcount(vcount), .pixel(instr_pixel));



	always @(posedge vclock) begin

		//set output values
		phsync <= hsync;
		pvsync <= vsync;
		pblank <= blank;
		red_pixel <= (instr)? instr_pixel : pixelr | force_pixel | cdpixel;
		green_pixel <= (instr)? instr_pixel : pixelg | force_pixel | cdpixel;
		blue_pixel <= (instr)? instr_pixel : pixelb | force_pixel | cdpixel;

		//ascii encodings for numbers 1, 2, 3, 4
		case(difficulty)
		0: diff_value_str <= 8'b00110001;
		1: diff_value_str <= 8'b00110010;
		2: diff_value_str <= 8'b00110011;
		3: diff_value_str <= 8'b00110100;
		endcase
	end
endmodule

////////////////////////////////////////////////////////////////////////
//
// force gauge:
//	produces a gradient-colored rectangle that varies
// in length, depending on the magnitude input
//
////////////////////////////////////////////////////////////////////////
module force_gauge
   #(parameter HEIGHT = 64)     // default height: 64 pixels
   (input [10:0] x,hcount,
    input [9:0] y,vcount,
    input [16:0] magnitude,
    output reg [7:0] pixel);

	reg [16:0] scaled_x;

	 always @ * begin

	scaled_x <= (x + (magnitude /64));
	if ((hcount >= x && hcount <  scaled_x * 17 / 32) &&  (vcount >= y && vcount < (y + HEIGHT)))
		pixel = 128;
	else if ((hcount >= x && hcount < scaled_x * 18 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
		pixel = 136;
	else if ((hcount >= x && hcount < scaled_x * 19 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
		pixel = 154;
	else if ((hcount >= x && hcount < scaled_x * 20 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
		pixel = 162;
	else if ((hcount >= x && hcount < scaled_x * 21 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
		pixel = 170;
	else if ((hcount >= x && hcount < scaled_x * 22 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
		pixel = 178;
	else if ((hcount >= x && hcount < scaled_x * 23 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
```

```
                pixel = 186;
            else if ((hcount >= x && hcount < scaled_x * 24 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 192;
            else if ((hcount >= x && hcount < scaled_x * 25 / 32)  && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 200;
            else if ((hcount >= x && hcount < scaled_x * 26 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 208;
            else if ((hcount >= x && hcount < scaled_x * 27 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 216;
            else if ((hcount >= x && hcount < scaled_x * 28 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 224;
            else if ((hcount >= x && hcount < scaled_x * 29 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 232;
            else if ((hcount >= x && hcount < scaled_x * 30 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 240;
            else if ((hcount >= x && hcount < scaled_x * 31 / 32) && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 248;
            else if ((hcount >= x && hcount < scaled_x ) && (vcount >= y && vcount < (y + HEIGHT)))
                pixel = 256;
            else pixel = 0;
   end
endmodule

////////////////////////////////////////////////////////////////////////////////
//
// Instructions Module:
// Displays instructions on how to use this application.
//
////////////////////////////////////////////////////////////////////////////////

module instructions(input vclock, input [10:0] hcount, input [9:0] vcount, output wire [7:0] pixel );

            //Instructions to display on screen.
            wire [247:0] cstring1 = "Welcome to Virtual Board Break!";
             wire [247:0] cstring2 = "Board Orientation - switch[1:0]";
            wire [247:0] cstring3 = "          [00] x axis          ";
            wire [247:0] cstring4 = "          [01] y axis          ";
            wire [247:0] cstring5 = "          [10] z axis          ";
            wire [247:0] cstring6 = "   Difficulty - switch[3:2]    ";
            wire [247:0] cstring7 = "          [00] Level 1         ";
            wire [247:0] cstring8 = "          [01] Level 2         ";
            wire [247:0] cstring9 = "          [10] Level 3         ";
            wire [247:0] cstring10 = "          [11] Level 4         ";
            wire [247:0] cstring11 = "      Reset: enter button      ";

            //Pixel values output to screen.
            wire [7:0] pixel1, pixel2, pixel3, pixel4, pixel5, pixel6, pixel7, pixel8, pixel9, pixel10, pixel11;
            assign pixel = pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | pixel9 | pixel10 | pixel11;

            //char_string_display modules that produce the instructions on screen.
            char_string_display cd1(.vclock(vclock),.hcount(hcount),.vcount(vcount), .pixel(pixel1),.cstring(cstring1),
                .cx(11'd176),.cy(10'd100));
             char_string_display cd2(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                 .pixel(pixel2),.cstring(cstring2),.cx(11'd176),.cy(10'd150));
             char_string_display cd3(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                 .pixel(pixel3),.cstring(cstring3),.cx(11'd176),.cy(10'd180));
```

```
	char_string_display cd4(.vclock(vclock),.hcount(hcount),.vcount(vcount),
			.pixel(pixel4),.cstring(cstring4),.cx(11'd176),.cy(10'd210));
	char_string_display cd5(.vclock(vclock),.hcount(hcount),.vcount(vcount),
			.pixel(pixel5),.cstring(cstring5),.cx(11'd176),.cy(10'd240));
	char_string_display cd6(.vclock(vclock),.hcount(hcount),.vcount(vcount),
			.pixel(pixel6),.cstring(cstring6),.cx(11'd176),.cy(10'd290));
	char_string_display cd7(.vclock(vclock),.hcount(hcount),.vcount(vcount),
			.pixel(pixel7),.cstring(cstring7),.cx(11'd176),.cy(10'd320));
	char_string_display cd8(.vclock(vclock),.hcount(hcount),.vcount(vcount),
			.pixel(pixel8),.cstring(cstring8),.cx(11'd176),.cy(10'd350));
	char_string_display cd9(.vclock(vclock),.hcount(hcount),.vcount(vcount),
			.pixel(pixel9),.cstring(cstring9),.cx(11'd176),.cy(10'd380));
	char_string_display cd10(.vclock(vclock),.hcount(hcount),.vcount(vcount),
			.pixel(pixel10),.cstring(cstring10),.cx(11'd176),.cy(10'd410));
	char_string_display cd11(.vclock(vclock),.hcount(hcount),.vcount(vcount),
			.pixel(pixel11),.cstring(cstring11),.cx(11'd176),.cy(10'd460));
endmodule

////////////////////////////////////////////////////////////////////////
//
// board:
// This module recieves the orientation and difficulty settings
// and determines which boards to show on the screen.
//
////////////////////////////////////////////////////////////////////////
module board_images
   (input [10:0] hcount, input [9:0] vcount, input [2:0] orient,  input [2:0] difficulty,  input [2:0] image_select, input
clock,  output reg [7:0] pixel_red, output reg [7:0] pixel_green, output reg [7:0] pixel_blue);

	//declare orientation values
	parameter ORIENT_X = 0;
	parameter ORIENT_Y = 1;
	parameter ORIENT_Z = 2;

	reg showx;
	reg [3:0] showy;
	reg [3:0] showz;

	wire [7:0] pixel_red_f, pixel_red_s;
	wire [7:0] pixel_green_f, pixel_green_s;
	wire [7:0] pixel_blue_f, pixel_blue_s;

	//module to show full board image
	board_large bl(.showx(showx), .hcount(hcount), .vcount(vcount), .clock(clock),  .img_sel(image_select),
		.pixel_red(pixel_red_f), .pixel_green(pixel_green_f), .pixel_blue(pixel_blue_f));

	//module to show side of board image
	board_small bs(.showy(showy), .showz(showz),  .hcount(hcount), .vcount(vcount), .clock(clock),
		.img_sel(image_select), .pixel_red(pixel_red_s), .pixel_green(pixel_green_s),
		pixel_blue(pixel_blue_s));


	always @(posedge clock) begin

		//Each show value corresponds to a board.
		//A show value is high when its corresponding board should be shown on the screen.
```

```
        //Whether that board should be shown at a given time
        //depends on the orietnation and difficulty user inputs.

        showx <=(orient == ORIENT_X);

        showy[3] <= (difficulty >= 3) && (orient == ORIENT_Y);
        showy[2] <= (difficulty >= 2) && (orient == ORIENT_Y);
        showy[1] <= (difficulty >= 1) && (orient == ORIENT_Y);
        showy[0] <= (difficulty >= 0) && (orient == ORIENT_Y);

        showz[3] <= (difficulty >= 3) && (orient == ORIENT_Z);
        showz[2] <= (difficulty >= 2) && (orient == ORIENT_Z);
        showz[1] <= (difficulty >= 1) && (orient == ORIENT_Z);
        showz[0] <= (difficulty >= 0) && (orient == ORIENT_Z);

        //Pixel outputs for red, green, and blue.
        //Or operation of the full board module output and the side of board module output.
        pixel_red <= pixel_red_f | pixel_red_s;
        pixel_green <= pixel_green_f | pixel_green_s;
        pixel_blue <= pixel_blue_f | pixel_blue_s;

  end
endmodule

///////////////////////////////////////////////////////////////////////
//
// board_large:
// This module outputs the rgb pixel values taken from the generated
// memory files with the full board images.
//
///////////////////////////////////////////////////////////////////////
module board_large
    (input showx,   input [10:0] hcount,  input [9:0] vcount,  input clock,  input [1:0] img_sel, output reg [7:0]
    pixel_red,  output reg [7:0] pixel_green, output reg [7:0] pixel_blue);

    //parameters for the board in the x orientation
    parameter HEIGHT = 256;
    parameter WIDTH = 128;
    parameter XLOC = 272;
    parameter YLOC = 122;

    //declare image memory look up values
    reg [15:0] index;
    reg [7:0] table_addr;
    reg [7:0] img_red, img_green, img_blue;
    wire [7:0] image_out_full1, image_out_full2, image_out_full3, image_out_full4;
    wire [7:0] fboard_red1, fboard_green1, fboard_blue1;
    wire [7:0] fboard_red2, fboard_green2, fboard_blue2;
    wire [7:0] fboard_red3, fboard_green3, fboard_blue3;
    wire [7:0] fboard_red4, fboard_green4, fboard_blue4;

    //images for front of board
    f1 full1(.addr(index), .clk(clock), .dout(image_out_full1));
    f2 full2(.addr(index), .clk(clock), .dout(image_out_full2));
    f3 full3(.addr(index), .clk(clock), .dout(image_out_full3));
    f4 full4(.addr(index), .clk(clock), .dout(image_out_full4));
```

```
//color tables for front of board
red_f1 redf1 (.addr(table_addr), .clk(clock), .dout(fboard_red1));
red_f2 redf2 (.addr(table_addr), .clk(clock), .dout(fboard_red2));
red_f3 redf3 (.addr(table_addr), .clk(clock), .dout(fboard_red3));
red_f4 redf4 (.addr(table_addr), .clk(clock), .dout(fboard_red4));
green_f1 greenf1 (.addr(table_addr), .clk(clock), .dout(fboard_green1));
green_f2 greenf2 (.addr(table_addr), .clk(clock), .dout(fboard_green2));
green_f3 greenf3 (.addr(table_addr), .clk(clock), .dout(fboard_green3));
green_f4 greenf4 (.addr(table_addr), .clk(clock), .dout(fboard_green4));
blue_f1 bluef1 (.addr(table_addr), .clk(clock), .dout(fboard_blue1));
blue_f2 bluef2 (.addr(table_addr), .clk(clock), .dout(fboard_blue2));
blue_f3 bluef3 (.addr(table_addr), .clk(clock), .dout(fboard_blue3));
blue_f4 bluef4 (.addr(table_addr), .clk(clock), .dout(fboard_blue4));

always @(posedge clock) begin
	//If showx is high, display the full board image.
	if (showx) begin
		//Depending on image_select, pass in a different image value for rgb table look up.
		//Images differ in the degree of how "broken" they are.
		case (img_sel)
			0: begin
				table_addr <= image_out_full1;
				img_red <= fboard_red1;
				img_green <= fboard_green1;
				img_blue <= fboard_blue1;
			end
			1: begin
				table_addr <= image_out_full2;
				img_red <= fboard_red2;
				img_green <= fboard_green2;
				img_blue <= fboard_blue2;
			end
			2: begin
				table_addr <= image_out_full3;
				img_red <= fboard_red3;
				img_green <= fboard_green3;
				img_blue <= fboard_blue3;
			end
			3: begin
				table_addr <= image_out_full4;
				img_red <= fboard_red4;
				img_green <= fboard_green4;
				img_blue <= fboard_blue4;
			end
			default: begin
				img_red <= 0;
				img_green <= 0;
				img_blue <= 0;
			end
		endcase

		//check image boundary conditions for the left side of the board
		if ((hcount >= XLOC && (hcount < XLOC + WIDTH)) &&(vcount >= YLOC &&
			vcount < (YLOC + HEIGHT))) begin
```

```
					//memory index for left half of board
					index <= WIDTH * (vcount - YLOC) + (hcount - XLOC);

					//pull image data from memory, and set as output
					pixel_red <= img_red;
					pixel_green <= img_green;
					pixel_blue <= img_blue;

					//check image boundary conditions for the right side of the board
				end else if ((hcount >= (XLOC + WIDTH) && (hcount < (XLOC + WIDTH +
				WIDTH))) && (vcount >= YLOC && vcount < (YLOC + HEIGHT))) begin

					//memory index for left half of board
					index <= WIDTH * (vcount - YLOC + 1) - (hcount - WIDTH - XLOC + 2);

					//pull image data from memory, and set as output
					pixel_red <= img_red;
					pixel_green <= img_green;
					pixel_blue <= img_blue;

				//if not within image boundaries, output 0 values for that pixel
				end else begin
					pixel_red <= 0;
					pixel_green <= 0;
					pixel_blue <= 0;
				end

				//if the small board is not visible, output only 0 values from this module
			end else begin
				pixel_red <= 0;
				pixel_green <= 0;
				pixel_blue <= 0;
			end
		end
endmodule

///////////////////////////////////////////////////////////////////////
//
// board_small:
// This module outputs the rgb pixel values taken from the generated
// memory files with the side of board images.
//
///////////////////////////////////////////////////////////////////////
module board_small
	(input [3:0] showy, input [3:0] showz, input [10:0] hcount,  input [9:0] vcount,  input clock,  input [1:0]
	img_sel, output reg [7:0] pixel_red, output reg [7:0] pixel_green, output reg [7:0] pixel_blue);

	//parameters for the board in the y orientation
	parameter Y_HEIGHT = 64;
	parameter Y_WIDTH = 138;
	parameter Y_XLOC = 272;
	parameter Y_YLOC1 = 150;
	parameter Y_YLOC2 = 225;
	parameter Y_YLOC3 = 300;
	parameter Y_YLOC4 = 375;
```

```
//parameters for the board in the z orientation
parameter Z_HEIGHT = 138;
parameter Z_WIDTH = 64;
parameter Z_YLOC = 122;
parameter Z_XLOC1 = 275;
parameter Z_XLOC2 = 350;
parameter Z_XLOC3 = 425;
parameter Z_XLOC4 = 500;

//declare image boundary values
reg [10:0] XLOC;
reg [9:0] YLOC;
reg [9:0] HEIGHT;
reg [10:0] WIDTH;
reg xcond, yupcond, ydowncond;

//decalare image memory values
reg [15:0] index;
reg [7:0] table_addr;
reg [7:0] img_red, img_green, img_blue;
wire [7:0] image_out_side1, image_out_side2, image_out_side3, image_out_side4;
wire [7:0] sboard_red1, sboard_green1, sboard_blue1;
wire [7:0] sboard_red2, sboard_green2, sboard_blue2;
wire [7:0] sboard_red3, sboard_green3, sboard_blue3;
wire [7:0] sboard_red4, sboard_green4, sboard_blue4;

//images of side of board
s1 side1(.addr(index), .clk(clock), .dout(image_out_side1));
s2 side2(.addr(index), .clk(clock), .dout(image_out_side2));
s3 side3(.addr(index), .clk(clock), .dout(image_out_side3));
s4 side4(.addr(index), .clk(clock), .dout(image_out_side4));

//color tables for side of board
red_s1 reds1 (.addr(table_addr), .clk(clock), .dout(sboard_red1));
red_s2 reds2 (.addr(table_addr), .clk(clock), .dout(sboard_red2));
red_s3 reds3 (.addr(table_addr), .clk(clock), .dout(sboard_red3));
red_s4 reds4 (.addr(table_addr), .clk(clock), .dout(sboard_red4));
green_s1 greens1 (.addr(table_addr), .clk(clock), .dout(sboard_green1));
green_s2 greens2 (.addr(table_addr), .clk(clock), .dout(sboard_green2));
green_s3 greens3 (.addr(table_addr), .clk(clock), .dout(sboard_green3));
green_s4 greens4 (.addr(table_addr), .clk(clock), .dout(sboard_green4));
blue_s1 blues1 (.addr(table_addr), .clk(clock), .dout(sboard_blue1));
blue_s2 blues2 (.addr(table_addr), .clk(clock), .dout(sboard_blue2));
blue_s3 blues3 (.addr(table_addr), .clk(clock), .dout(sboard_blue3));
blue_s4 blues4 (.addr(table_addr), .clk(clock), .dout(sboard_blue4));

always @(posedge clock) begin
        //side of the board images are visible for y and z axes
        if (showy[0] || showz[0]) begin

                //Depending on image_select, pass in a different image value for rgb table look up.
                //Images differ in the degree of how "broken" they are.
                case (img_sel)
                0: begin
                        table_addr <= image_out_side1;
                        img_red <= sboard_red1;
```

```
					img_green <= sboard_green1;
					img_blue <= sboard_blue1;
				end
				1: begin
					table_addr <= image_out_side2;
					img_red <= sboard_red2;
					img_green <= sboard_green2;
					img_blue <= sboard_blue2;
				end
				2: begin
					table_addr <= image_out_side3;
					img_red <= sboard_red3;
					img_green <= sboard_green3;
					img_blue <= sboard_blue3;
				end
				3: begin
					table_addr <= image_out_side4;
					img_red <= sboard_red4;
					img_green <= sboard_green4;
					img_blue <= sboard_blue4;
				end
				default: begin
					img_red <= 0;
					img_green <= 0;
					img_blue <= 0;
				end
			endcase

		//if we are displaying images on the z axis, set y parameters
		if (showz[0]) begin
			YLOC <= Z_YLOC;
			HEIGHT <= Z_HEIGHT;
			WIDTH <= Z_WIDTH;

			//adjust x parameter accordingly (when drawing multiple boards on the screen)
			if (hcount <= Z_XLOC1 + Z_WIDTH) XLOC <= Z_XLOC1;
			else if (showz[1] && hcount <= Z_XLOC2 + Z_WIDTH) XLOC <= Z_XLOC2;
			else if (showz[2] && hcount <= Z_XLOC3 + Z_WIDTH) XLOC <= Z_XLOC3;
			else if (showz[3] && hcount <= Z_XLOC4 + Z_WIDTH) XLOC <= Z_XLOC4;

			//image boundary conditions
			xcond <=(hcount >= XLOC && (hcount < XLOC + WIDTH));
			yupcond <= (vcount >= YLOC && vcount < (YLOC + HEIGHT));
			ydowncond <=(vcount >= (YLOC + HEIGHT) && vcount < (YLOC + (HEIGHT + HEIGHT)));

			//vertical image
			if (xcond && yupcond) begin
				//memory index for upper half of board
				index <= HEIGHT * (hcount - XLOC) + (vcount - YLOC);
				//pull image data from memory, and set as output
				pixel_red <= img_red;
				pixel_green <= img_green;
				pixel_blue <= img_blue;
			end else if (xcond && ydowncond) begin
				//memory index for lower half of board
				index <= HEIGHT * (hcount - XLOC + 1) - (vcount - HEIGHT - YLOC + 1);
```

```
				//pull image data from memory, and set as output
				pixel_red <= img_red;
				pixel_green <= img_green;
				pixel_blue <= img_blue;
			//if not within image boundaries, output 0 values for that pixel
			end else begin
				pixel_red <= 0;
				pixel_green <= 0;
				pixel_blue <= 0;
			end
			//if we are displaying images on the y axis, set x parameters
			end else if (showy[0]) begin
				XLOC <= Y_XLOC;
				HEIGHT <= Y_HEIGHT;
				WIDTH <= Y_WIDTH;

				//adjust y parameter accordingly (when drawing multiple boards on the screen)
				if (vcount <= Y_YLOC1 + Y_HEIGHT) YLOC <= Y_YLOC1;
				else if (showy[1] && vcount <= Y_YLOC2 + Y_HEIGHT) YLOC <= Y_YLOC2;
				else if (showy[2] && vcount <= Y_YLOC3 + Y_HEIGHT) YLOC <= Y_YLOC3;
				else if (showy[3] && vcount <= Y_YLOC4 + Y_HEIGHT) YLOC <= Y_YLOC4;

				//draw horizontal image
				//boundary conditions for left half of the board
				if ((hcount >= XLOC && (hcount < XLOC + WIDTH)) &&
					(vcount >= YLOC && vcount < (YLOC + HEIGHT))) begin

					//memory index for the left half of board
					index <= WIDTH * (vcount - YLOC) + (hcount - XLOC);

					//pull image data from memory, and set as output
					pixel_red <= img_red;
					pixel_green <= img_green;
					pixel_blue <= img_blue;

					//boundary conditions for right half of the board
				end else if ((hcount >= (XLOC + WIDTH) && (hcount < (XLOC + WIDTH +
				WIDTH))) &&(vcount >= YLOC && vcount < (YLOC + HEIGHT))) begin

					//memory index for the right half of board
					index <= WIDTH * (vcount - YLOC + 1) - (hcount - WIDTH - XLOC + 1);

					//pull image data from memory, and set as output
					pixel_red <= img_red;
					pixel_green <= img_green;
					pixel_blue <= img_blue;

					//if not within image boundaries, output 0 values for that pixel
				end else begin
					pixel_red <= 0;
					pixel_green <= 0;
					pixel_blue <= 0;
				end
			end
		//if the small board is not visible, output only 0 values from this module
		end else begin
```

```
                    pixel_red <= 0;
                    pixel_green <= 0;
                    pixel_blue <= 0;
            end
            end
endmodule

////////////////////////////////////////////////////////////////////////////////
//
// player:
// Plays the sound of a board breaking when it recieves a high
// pulse for the playback signal.
//
////////////////////////////////////////////////////////////////////////////////
module player(
  input wire clock,                         // 27mhz system clock
  input wire reset,                 // 1 to reset to initial state
  input wire playback,             // 1 pulse for playback
  input wire ready,                 // 1 when AC97 data is available
  output reg [7:0] to_ac97_data    // 8-bit PCM data to headphone
);
            wire [7:0] break;
            reg [15:0] index = 0;
            reg break_on_off;

            audioram ar(.addr(index), .clk(clock), .dout(break));

             always @ (posedge clock) begin

                    //When ready high, audio data is ready
                     if (ready) begin
                            //When recieving the playback pulse, start from teh beginning of the file,
                            //and toggle play break on.
                            if (playback) begin
                                    index<=0;
                                    break_on_off<=1;
                            end else begin
                                    //When the end of the sound file is reached, go back to the beginning.
                                    //Toggle play break off if its on.
                                    if (index >= 10000) begin
                                            index <= 0;
                                            break_on_off<=0;
                                    //Increment index to progress through the sound file.
                                    end else
                                            index <= index + 1;
                            end

                            //When play break is toggled on, play the break sound data.
                            //Otherwise, play nothing.
                            if (break_on_off) to_ac97_data <= break[7:0];
                            else to_ac97_data<=8'b0;

     end
   end
endmodule
```