Running Rate-Controlled Music Player

Sabrina Chang, Lilly Liu

E155 - Fall 2017

Abstract

Our project implements a person's running rate-controlled music player. A song of similar beats-per-minute (BPM) from the runner's initial rate is chosen from a stored database, and depending on the runner's speed during the length of the song, the song speeds up or slows down. The runner's landing rate is obtained through an accelerometer, whose output is processed with an FPGA. The FPGA calculates the period of the runner's landing rate, and sends that data to the Raspberry Pi when it is requested, which does the necessary conversion to BPM and plays a song and controls the speed. The system is able to accurately obtain the user's running BPM and play songs depending on the running speed.

Introduction

Many runners listen to music during their workout to keep up motivation or just to make their runs more interesting. It is more motivating for runners to listen to songs of similar tempos to their running rate than if the running rate greatly mismatched the song's BPM.

This project helps the runner with their music needs by selecting a song initially based off of their running speed during a calibration period. The music is then sped up or slowed down depending on the runner's speed in reference to the song's BPM. The runner's BPM is obtained through an accelerometer that the runner holds in their hand. The runner's landing rate is obtained through the accelerometer held in hand as they run because studies and experimentation have shown that a person's running rate is 1:1 with their arms' swinging rate.

The user interfaces with a webpage that contains a start button to start the calibration and song, and a cancel button that cancels the song. The webpage also displays the runner's speed and the title of the song being played. The website is hosted on the Raspberry Pi, which requests a new running period from the FPGA every two seconds. The FPGA does signal processing on the data from the analog-to-digital converter that converts the output from the accelerometer. The FPGA counts the number of cycles that pass between each swing, and sends this number to the Raspberry Pi upon request. Figure 1 shows the block diagram of our system.

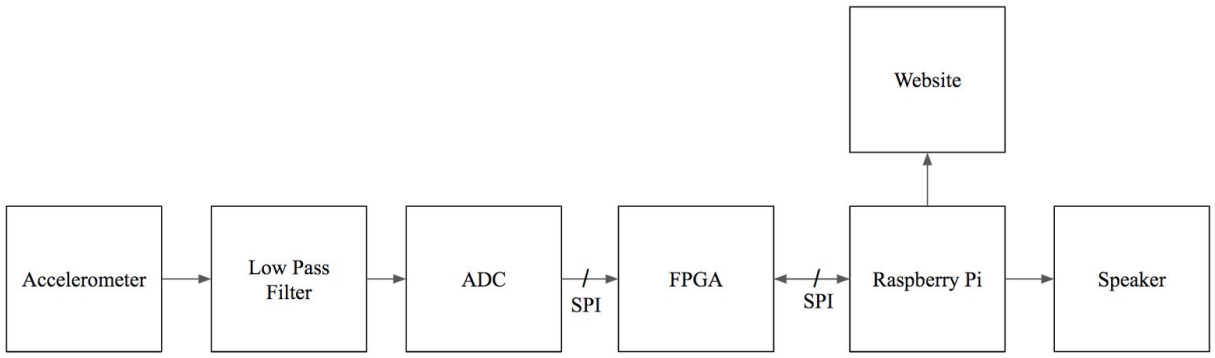

Figure 1: System block diagram

New Hardware

In this project, we used an accelerometer to track the user's hand swings' acceleration. Only the z output pin of the accelerometer is used to gather data, since the user's arm swing accelerates predominantly in the vertical z direction. A low-pass filter is implemented at the output pin. Since the range of data of interest in this project is at most 3 Hz (or 180 BPM, which is the running BPM of advanced athletes), a low-pass filter with cutoff frequency of 1 Hz is implemented with a simple RC filter to improve measurement resolution and prevent aliasing. The datasheet specifies that the bandwidth should be limited to the lowest frequency needed for application to maximize resolution.

The accelerometer's output data is observed to be roughly sinusoidal (see Figure 2), with maximum peaks at the point of maximum acceleration, which is at the lowest point of the user's swing. Our initial data collection revealed that the data is acceptably noise-free and clean to do signal processing on without additional filtering.

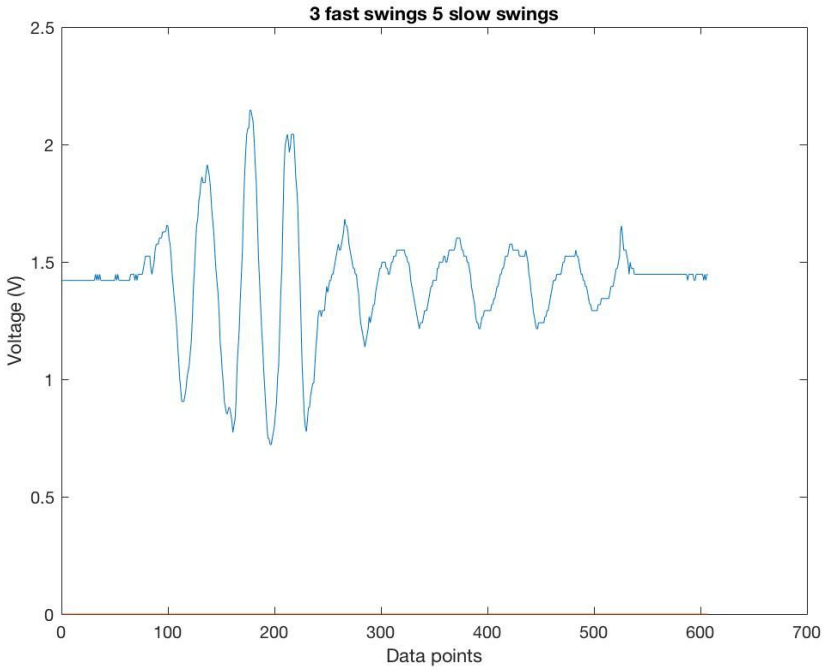

Figure 2: Sample data collected with FPGA from accelerometer

Schematics

As seen in Figure 3, our system's overall circuit schematic is shown. Data is first collected from the z acceleration output pin of the accelerometer ADXL335, then fed into Channel 1 of the analog-to-digital converter MCP3002, where it is converted into 10-bit data for the FPGA to process. The data is transferred from the ADC to FPGA using Serial Peripheral Interface (SPI).

The FPGA receives the data from the accelerometer and calculates the period in real time, then adds it to the stored average of past 8 detected periods to ensure a smoother transition in song speeds for the Raspberry Pi to play out.

The Raspberry Pi requests data from the FPGA over SPI, and converts the period value into BPM, and depending on whether the start button has just been pressed or if it's in the middle of a song, selects a song of similar BPM from a stored database or multiplies the song speed by the ratio of current running BPM over song BPM.

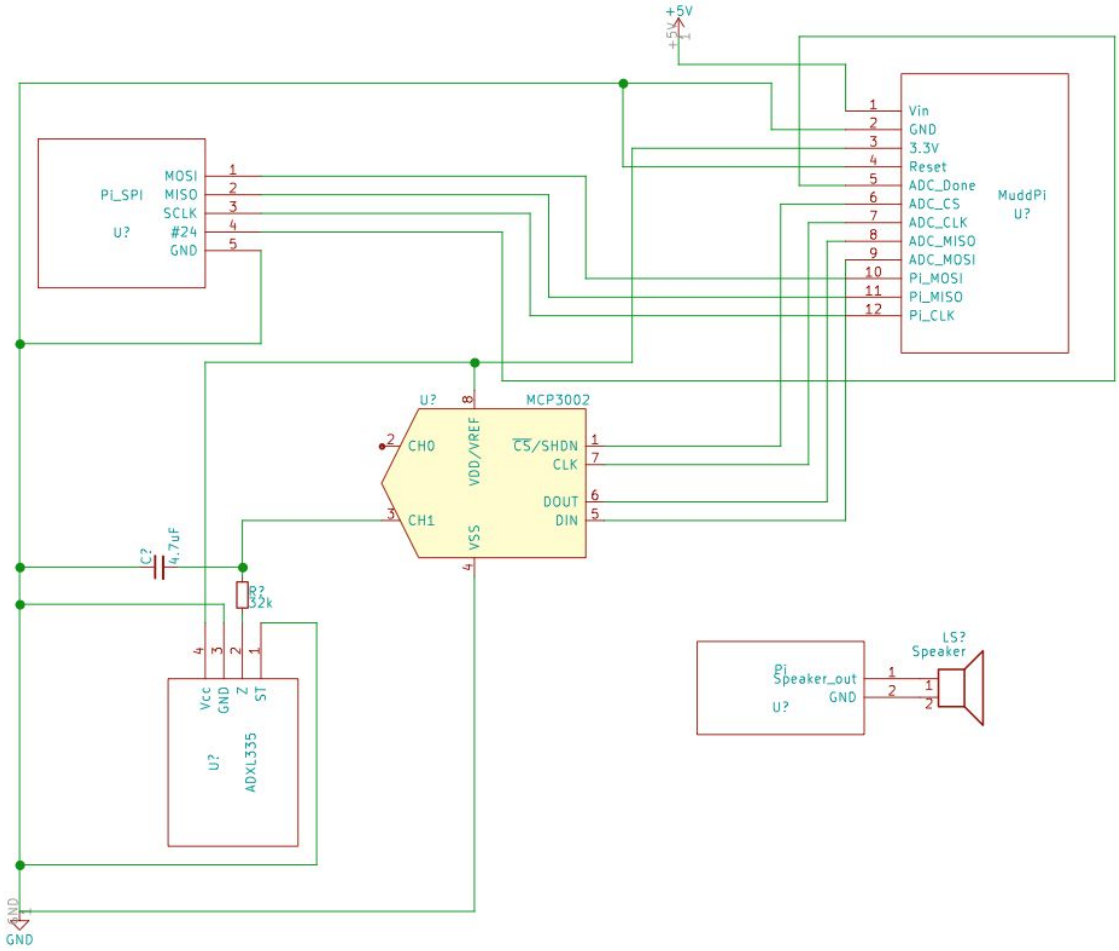

Figure 3: Overall Circuit Schematic

FPGA Design

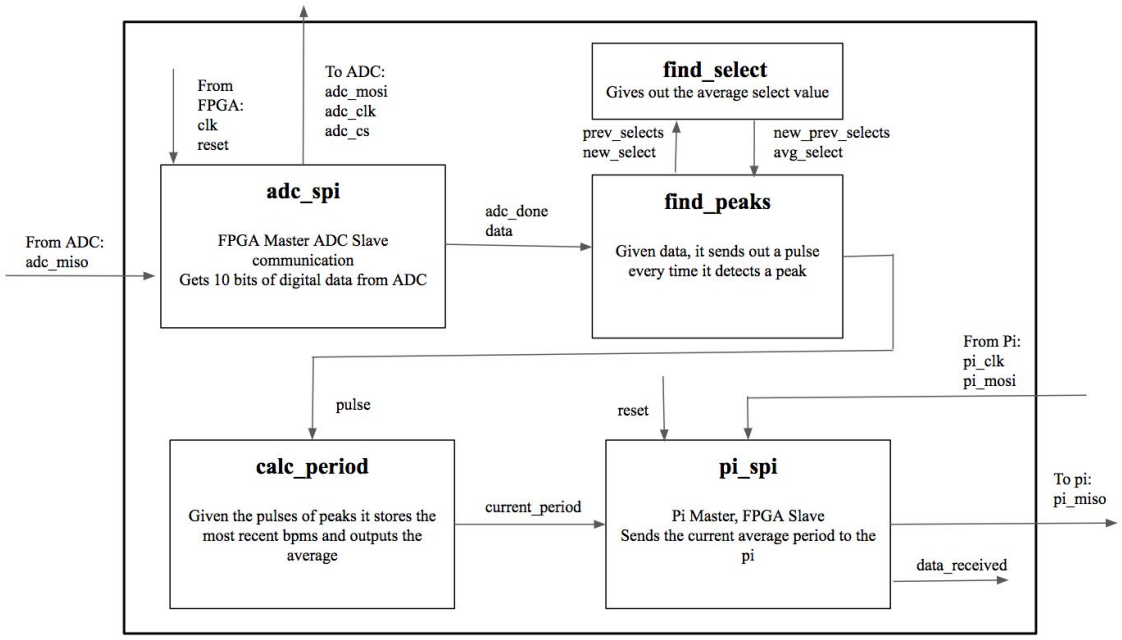

Figure 4: Block Diagram of FPGA

Summary

Upon receiving data from the accelerometer with SPI communication, the FPGA calculates the period between each arm swing using peak detection and sends it to the Raspberry Pi upon request with SPI communication.

ADC slave FPGA master SPI Communication

The ADC converts the analog voltage output from the accelerometer into 10-bit data that is sent to the FPGA. The FPGA requests data from the ADC on the adc_clk by sending out adc_mosi and adc_cs which contains information on CS, channel selection mode, and data transfer order. The slow clock (sclk) is chosen to be a factor of 2^{15} slower than the utility board's clock (40 MHz) because our counter for the period is based off of the communication, and faster clocks would cause overflow in our period counter since we are only sending eight bits.

Finding Period

Once all ten bits of data are shifted in from the ADC, the find_peaks module takes the new data in and determines whether there is a peak. The model for peak detection was first simulated in MATLAB to verify the accuracy of detection. On the FPGA, it accomplishes peak

detection by first identifying all peaks by computing the slopes between the current and previous data points and multiplying the newly found slope with the stored previous slope, then identifying the previous data point as a potential peak if the multiplied result is less than or equal to zero.

Another criteria the potential peak must pass to be identified as an actual peak is that it must be a certain margin above the local minimum, or the valley before the peak. This margin is defined as ‘new_select’ in the find_peaks module, and it is the difference between the values of a confirmed peak and valley divided by four. The actual margin used to determine peaks is the moving average of the past four ‘select’ values.

This peak detection function performs more accurately as time goes on. As seen in Figure 5, it is able to detect peaks with sufficient accuracy, but the initial detection and transitional periods are less accurate because the ‘select’ values and periods need to settle to a steady-state value to consistently be able to pick the correct peaks.

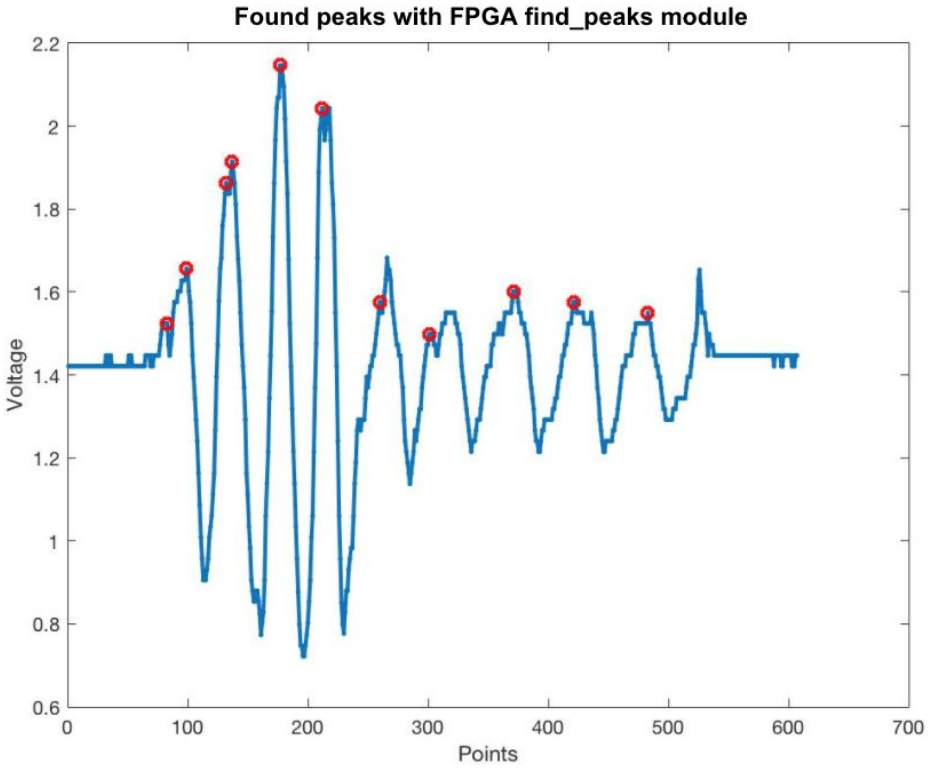

Figure 5: Sample Data with Detected Peaks after FPGA Processing

If a potential peak is confirmed to be an actual peak, the output ‘pulse’ is set to one, and another module called ‘calc_period’ restarts the counter and counts the number of cycles of adc_done (high when ADC is done transferring data) that passes until the next pulse. This number is then stored along with the past seven periods and averaged out. The averaged periods is the value sent to the Pi upon request with SPI communication.

Microcontroller Design

Summary

The Pi controls the main logic for the system. It hosts a website with a play button, once the button is pushed the C code is run. The C code first waits five seconds so the periods received could be more accurate, asks for a period value and uses that to find a song. There are three arrays that store information, one is an array that has the bpm and the length of the song. The second array includes the filename of the song, and the third one is a string that has the official title and artist. All the songs are placed in the same order so the indexing is the same. It iterates through the array that has the bpm and length of the song and finds the one that is the closest and chooses that as the song. Then it goes into a loop that plays the song. It plays portions of the song at two seconds at a time and adjusts the tempo of each two second chunk so that the song tempo will reflect the period received from the player. Our website also displays the song name and artist and current speed in real time. The website also has a cancel button that cancels the process

Finding BPM

The Pi receives the period value from the FPGA. The period value received is actually the number of cycles of adc_done passed. To compute the frequency of running, the MuddPi board’s clock (40MHz) is divided by 2^{15} and again by 16 (takes 16 cycles to collect 2 bytes of data from ADC) to obtain adc_done’s frequency. The received period is then multiplied by the inverse of adc_done’s frequency to find the actual time period between each peak. The BPM of the runner is then computed by taking the inverse of this period multiplied by 60 seconds/minute.

Playing Music

Playing the music uses an external library called SoX (Sound eXchange). SoX allows users to trim part of the song as well as adjust the tempo. How a typical system call in the C program with SoX is formatted is "play songname.wav trim start_time length_of_trim tempo tempo_rate." The song is selected earlier, the start_time is iterated through, the length_of_trim is two seconds, and the tempo_rate is calculated as shown above. One problem we had earlier was because we needed to wait for the Pi to receive the period value from the FPGA there was a noticeable gap between the two sections of the song being played. To solve this problem, at the end of the system call we added an "&" which meant the command is forked into a sub shell and run asynchronously. Then we had to delayMillis, because when the song is run in a sub shell, it goes immediately back to the main program and we want that section to play before we play the next section. We delayMillis slightly less than how fast the song would play so the main program could ask for the period from the FPGA and it would be able to start a new call as the old call ended. As a result, the song played sounded continuous.

Website

The website uses Javascript to fetch the current song, artist and frequency. In the C code, it writes the current song, artist, and frequency to a txt file. In the Javascript it uses XML to fetch the text from the txt file and display it on the website. There are currently some issues with the Javascript because it starts updating the text as soon as the play button is pushed but the there should be no current song and frequency so it loads what is previously on the txt file. The fix we attempted was in our C code, before it chooses a song, it writes "Loading Data…" into the txt file so that it shows that it is still loading before it actually is playing the song. Our website also has a cancel button which calls a C program that makes a system call to kill the processes. Although it successfully ends the program, it also leads to a 500 internal server error because the play program is canceled. We were unable to find a way to prevent the 500 internal server error from showing. For a better product, we would fix the Javascript and 500 internal server problems.

Results

We successfully implemented our project, which included accurate detection of the runner's landing rate, correct selection of songs of similar BPM to play, as well as ability to speed up or slow down the song's speed depending on the runner's speed. The website implemented is also able to successfully start and cancel playing the song depending on the user's control. It is also able to display the song title and runner's current BPM.

There is room for improvement in the robustness of our system. Because we are swinging the accelerometer mounted on a breadboard, sometimes the wires would come loose and we would be perplexed as to why the system suddenly stopped working. Another aspect that could be improved is to tailor the sensitivity of the accelerometer to suit our purposes better. While the accelerometer is able to obtain reasonable data, it requires horizontal positioning of the board, or else it would be too sensitive to unintentional tilts and human motions.

References

D. M. Harris and S. L. Harris, "I/O Systems," in Digital Design and Computer Architecture: ARM Edition, 1st ed. Burlington, MA: Morgan Kaufmann, 2015, ch. 9, pp. 531.e9.

Ford, Matthew P., et al. "Arm constraint and walking in healthy adults." *Gait & Posture*, vol. 26, no. 1, 2007, pp. 135–141., doi:10.1016/j.gaitpost.2006.08.008.

"Small, Low Power, 3 Axis +/-g Accelerometer" ADXL335 datasheet, Analog Devices, Inc. 2009. https://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf

Tom O'Haver. "Peak Finding and Measurement." *Peak Finding and Measurement*, terpconnect.umd.edu/~toh/spectrum/PeakFindingandMeasurement.htm#findpeaksb.

"2.7V Dual Channel 10-Bit A/D Converter with SPI Serial Interface," MCP3002 datasheet, Microchip Technology Inc. 2011.

Parts List

Part	Source	Part Number	Price

Accelerometer	https://www.sparkfun.com/products/9269	ADXL335	$14.95
Headphones	Personal	Bose SoundLink on-ear wireless headphones	N/A
ADC	E155 Cabinet	MCP3002	N/A
Capacitor	E155 Cabinet	030KQ	N/A

Appendices

Appendix A: SystemVerilog Code

```
////////////////////////////////////////////
// final_ll_sc.sv
// Lilly Liu, Sabrina Chang
// lliu@hmc.edu, schang@hmc.edu
// December 2, 2017
// Finds the periods between peaks
////////////////////////////////////////////

module final_ll_sc(input logic pi_clk, clk, reset,
                                input logic adc_miso, pi_mosi,
                                output logic adc_mosi, pi_miso,
                                output logic adc_clk,
                                output logic adc_cs,
                                output logic led, adc_done);

      logic [9:0] data; //accelerometer data
      logic [7:0] pi_data, send_period; //pi_data not used, placeholder
      pi_spi pi(pi_clk, pi_mosi, pi_miso, reset, send_period, pi_data);
      adc_spi adc(clk, reset, adc_miso, adc_mosi, adc_clk, adc_cs, data, adc_done);
      calc_period run_per(reset, data, adc_done, send_period);

      assign led = 1;
endmodule

////////////////////////////////////////////
// Pi Master, FPGA Slave
// Sends data to Raspberry pi on request
// Based off of code found in
```

```
// Digital Design and Computer Architecture by Harris and Harris
//////////////////////////////////////////

module pi_spi(input logic sck, //From master
                              input logic mosi, //From master
                              output logic miso, //To master
                              input logic reset, // System reset,
                              input logic[7:0] d, // Data to send
                              output logic [7:0] q); // Data recieved

       logic [2:0] cnt;
       logic qdelayed;

       //3-bit counter tracks when full byte is transmitted
       always_ff@(negedge sck, posedge reset)
              if (reset) cnt = 0;
              else cnt = cnt + 3'b1;

       // Loadable shift register
       // Loads d at the start, shifts mosi into bottom on each step
       always_ff@(posedge sck)
              q <= (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

              // Align miso to falling edge of sck
              // Load d at the start
       always_ff@(negedge sck)
              qdelayed = q[7];

       assign miso = (cnt == 0) ? d[7] : qdelayed;
endmodule

//////////////////////////////////////////
// FPGA Master, ADC Slave
// Sends accelerometer data on request
//////////////////////////////////////////
module adc_spi(input logic clk, reset,
                                  input logic adc_miso,
                                  output logic adc_mosi,
                                  output logic adc_clk,
                                  output logic adc_cs,
                                  output logic[9:0] data,
                                  output logic adc_done);

              typedef enum logic {S0, S1} statetype;
              statetype state, next_state;

              logic [14:0] sclk;

              always_ff@(posedge clk, posedge reset)
                     if (reset) sclk <= 15'b0;
```

```
	else sclk <= sclk + 15'b1;

assign adc_clk = sclk[14];
logic [15:0] send;
assign send = 16'b1111000000000000;

logic next_adc_mosi, next_adc_cs;
logic [5:0] i, next_i;

always_comb
	case(state)
		S0: begin
				adc_done = 0;
				next_adc_mosi = send[15 - i];
				next_adc_cs = 0;
				next_i = i + 1;
				if (i < 15)
					begin
					next_state = S0;
					end
				else
					begin
					next_state = S1;
					end
			end
		S1: begin
				adc_done = 1;
				next_adc_mosi = 0;
				next_i = 0;
				next_adc_cs = 1;
				next_state = S0;
			end
		endcase

always_ff@(posedge adc_clk, posedge reset)
	if (reset) state <= S0;
	else state <= next_state;

always_ff@(posedge adc_clk, posedge reset)
	if (reset) i <= 0;
	else i <= next_i;

always_ff@(posedge adc_clk)
	if (i == 0) data = 0;
	else if ((i >= 5) && (i < 16)) data[15 - i] <= adc_miso;

always_ff@(negedge adc_clk)
	adc_cs <= next_adc_cs;

always_ff@(negedge adc_clk)
```

```
                    adc_mosi <= next_adc_mosi;

endmodule

//////////////////////////////////////////
// Finds peaks in data
// Given accelerometer data and sends out a pulse when it detects a peak
// Based off Matlab peak finding algorithm
//////////////////////////////////////////
module find_peaks(input logic reset,
                                        input logic [9:0] data,
                                        input logic adc_done,
                                        output logic pulse);

              logic [10:0] curr_data, prev_data, curr_slope, prev_slope, mult_slope;
              logic [10:0] temp_max, temp_min, new_select, avg_select, peak_mag,
new_peak, new_valley, valley_mag;
              logic [43:0] prev_selects, new_prev_selects; //prev_selects stores past 4
selects; new_prev_selects has new_select shifted in
              logic next_pulse;

              assign curr_data[10] = 0;  //sign extention    since data from adc is
unsigned
              assign curr_data[9:0] = data[9:0];

              always_ff@(posedge adc_done, posedge reset)
                    if (reset) begin
                          pulse <= 0;
                          prev_data <= 0;
                          prev_slope <= 0;
                    end else begin
                          pulse <= next_pulse; //nextpulse is output from state
                                               machine when peak is detected
                          prev_data <= curr_data; //data and slopes shifted on next
cycle
                          prev_slope <= curr_slope;
                    end

              assign curr_slope = curr_data - prev_data; //current slope defined
              assign mult_slope = curr_slope * prev_slope; //multiplication of current
slope and previous slope shows whether there is sign change or zero (getting all
potential peaks)

              //peak_mag is set to 0 in state machine unless an actual peak is
confirmed, and new peak's magnitude value is output
              always_ff@(posedge adc_done, posedge reset)
                    if (reset) new_peak <= curr_data;
                    else if (peak_mag != 0) new_peak <= peak_mag;
                    else new_peak <= new_peak;
```

```
            //valley_mag is set to 0 in state machine unless an actual peak is
confirmed, valley magnitude used in calculating select
            always_ff@(posedge adc_done, posedge reset)
                  if (reset) new_valley <= curr_data;
                  else if (valley_mag != 0) new_valley <= valley_mag;
                  else new_valley <= new_valley;

            //temp_min is the temporary minimum before a potential peak; resets once
peak is detected;
            always_ff@(posedge adc_done, posedge pulse, posedge reset)
                  if (pulse || reset) temp_min <= curr_data;
                  else if (curr_data < temp_min) temp_min <= curr_data;
                  else temp_min <= temp_min;

            //temp_max is temporary maximum; changes to current data if it fulfills
conditions
            always_ff@(posedge adc_done, posedge pulse, posedge reset)
                  if (pulse || reset) temp_max <= 0;
                  else if ((curr_data > temp_max) && (curr_data > (temp_min +
avg_select))) temp_max <= curr_data;
                  else temp_max <= temp_max;

            assign new_select = (new_peak-new_valley) >>> 2; //new_select calculated
based on current peak and valley different divided by 4

            find_select fndsel(prev_selects, new_select, avg_select,
                new_prev_selects); //calls find_select module to compute the average
                      of past 4 select values

            //stored previous select values get updated if there is a new select
              occurs when there is a pulse
            always_ff@(posedge adc_done, posedge reset)
                  if (reset) prev_selects <= 44'd0;
                  else if (pulse) prev_selects <= new_prev_selects;
                  else prev_selects <= prev_selects;

            //State machine to pick peaks
            typedef enum logic {s0, s1} statetype;
            statetype state, next_state;

            always_ff@(posedge adc_done, posedge reset)
                  if (reset) state <= s0;
                  else state <= next_state;

            always_comb
                  case(state)
                        s0: begin
                                    next_pulse = 0;
```

```
                                        //passes first check of peak validation if
it's a potential peak due to change in slope; current data should be less than
previous data and temp_max because of how the slope of current data is calculated
                                        if ( ((mult_slope[10] == 1) || (mult_slope ==
                                       11'b0)) && (curr_data < prev_data) &&
                                       ((curr_data < temp_max) || (curr_data ==
                                       temp_max)) && (curr_data > (temp_min +
                                       avg_select))) begin
                                              next_state = s1;
                                              peak_mag = 0;
                                              valley_mag = 0;
                                              end
                                        else begin
                                              next_state = s0;
                                              peak_mag = 0;
                                              valley_mag = 0;
                                              end
                                  end
                            //if the temporary maximum is greater than temporary
current                                    minimum, or valley, a peak is detected
                            s1: begin
                                        if ((temp_max > (temp_min +
avg_select))||(temp_max == (temp_min + avg_select))) begin
                                              next_pulse = 1;
                                              next_state = s0;
                                              peak_mag = temp_max;
                                              valley_mag = temp_min;
                                       end else begin
                                              next_pulse = 0;
                                              next_state = s0;
                                              peak_mag = 0;
                                              valley_mag = 0;
                                       end
                                  end
                     endcase

endmodule

//////////////////////////////////////////
// Calculates select variable
// Helper module for peak finder
// Select value is the average of the past 4 select values
//////////////////////////////////////////
module find_select(input logic [43:0] prev_selects,
                                        input logic [10:0] new_select,
                                        output logic [10:0] avg_select,
                                        output logic [43:0] new_prev_selects);
              logic [11:0] sum, avg;
              assign new_prev_selects[43:11] = prev_selects[32:0];
              assign new_prev_selects[10:0] = new_select;
```

```
            assign sum = prev_selects[43:33] + prev_selects[32:22] +
prev_selects[21:11] + prev_selects[10:0];
            assign avg = (sum >>> 2);
            assign avg_select = avg[10:0];
endmodule

//////////////////////////////////////////
// Calculates average period
// Helper module for calc_period
// Average of the past 8 period values
//////////////////////////////////////////
module find_period(input logic [63:0] prev_periods,
                                      input logic [7:0] new_period,
                                      output logic [7:0] avg_period,
                                      output logic [63:0] new_prev_periods);

            logic [10:0] sum,avg;
            assign new_prev_periods[63:8] = prev_periods[55:0];
            assign new_prev_periods[7:0] = new_period;
            assign sum = new_prev_periods[63:56] + new_prev_periods[55:48] +
new_prev_periods[47:40] + new_prev_periods[39:32] + new_prev_periods[31:24] +
new_prev_periods[23:16] + new_prev_periods[15:8] + new_prev_periods[7:0];
            assign avg = (sum >>> 3);
            assign avg_period = avg[7:0];
endmodule

//////////////////////////////////////////
// Calculates average of 4 periods from pulses output by find_peaks
//////////////////////////////////////////
module calc_period(input logic reset,
                                          input logic [9:0] data,
                                          input logic adc_done,
                                          output logic [7:0] send_period);
            logic pulse, reset_counter;
            logic [7:0] period_counter, temp_period, new_period, avg_period;
            logic [63:0] prev_period, new_prev_period;


            find_peaks peaks(reset, data, adc_done, pulse);

            assign reset_counter = (pulse || reset);
            counter #(8) peakcount(adc_done, reset_counter, period_counter);

            flopr #(8) floppin(adc_done, reset, period_counter, temp_period);
            flopenr #(8) moarflop(adc_done, reset, pulse, temp_period, new_period);

            always_ff@(posedge reset_counter)
                  prev_period <= new_prev_period;
```

```
            find_period findperiod(prev_period, new_period, avg_period,
new_prev_period);

            always_ff@(posedge reset_counter)
                  send_period <= avg_period;

endmodule

module counter #(parameter WIDTH = 10)
                                       (input logic clk, reset,
                                              output logic [WIDTH-1:0] q);
                                       always_ff@(posedge clk, posedge reset)
                                              if(reset) q<=0;
                                              else q<=q+1;
endmodule

module flopenr #(parameter WIDTH = 10)
                                       (input logic clk, reset, en,
                                        input logic [WIDTH-1:0] d,
                                        output logic [WIDTH-1:0] q);
            always_ff@(posedge clk, posedge reset)
            if (reset) q<=0;
            else if (en) q<=d;
endmodule

module flopr #(parameter WIDTH = 10)
                                (input logic clk, reset,
                                 input logic [WIDTH-1:0] d,
                                 output logic [WIDTH-1:0] q);
            always_ff@(posedge clk, posedge reset)
            if (reset) q<=0;
            else q <= d;
endmodule
```

Appendix B: Main C Code

```
#include "EasyPIO.h"
#include <string.h>

#define DONE_PIN 24

const char* song_names[12] = {"Octahate by Ryn Weaver", "Slow Hands by Niall Horan",
"Cheap Thrills by Sia", "Youth by Glass Animals", "Another One Bites The Dust by
Queen", "Should I Stay Or Should I go by The Clash", "Shower by Becky G",
 "Sweater Weather by The Neighborhood", "Shut Up And Dance by Walk The Moon", "The
Walker by Fitz And The Tantrums", "Ex's & Oh's by Elle King", "Breezeblocks by Alt
J"};

const char* song_locations[12] = {"octahate.wav", "slowHands.wav", "cheapThrills.wav",
"youth.wav", "anotherOneBitesTheDust.wav", "shouldIStayOrShouldIGo.wav", "shower.wav",
```

```
"sweaterWeather.wav", "shutUpAndDance.wav", "theWalker.wav", "exsAndOhs.wav",
"breezeblocks.wav"};


int song_data[][12] = {{80, 204}, {86, 188}, {90, 212}, {96, 231}, {110, 215}, {113,
189}, {120, 206}, {124, 240}, {128, 199}, {131, 233}, {140, 202}, {150, 227}}
char current_song[1000] = "";
FILE *fp;

// Finds the current running frequency
float find_freq(void) {
        unsigned short received;
        unsigned short received_period;
        float run_freq;
        spiInit(244000, 0);
        received = spiSendReceive(0b00000000);
        if (received == 0) { //Should only happen at beginning
                return 95;
        }
        run_freq = (4030)/received; // calculating freq off period
        fp = fopen("/var/www/html/liveData.txt", "w+");
        char curr_song[500];
        strcpy(curr_song, current_song);
        strcat(curr_song, "Frequency of Runner: %f \n");
        printf("%s \n", curr_song);
        fprintf(fp, curr_song, run_freq);
        fflush(fp);
        return run_freq;
}
// Returns a system call to sox that is duration seconds long
// from start at the speed of tempo
const char* song_string(int song_index, int start, int duration, float tempo_ratio) {
        char play[500] = "play ";
        char song_name[400] = "/home/pi/";
        char trim[50] = " trim ";
        char tempo[50] = " tempo ";
        char empty[50] = " ";
        char str_start[50];
        char str_tempo[50];
        char  str_duration[50];
        strcat(song_name, song_locations[song_index]);
        strcat(play, song_name);
        sprintf(str_start, "%d", start);
        strcat(trim, str_start);
        sprintf(str_duration, "%d", duration);
        strcat(empty, str_duration);
        sprintf(str_tempo, "%f", tempo_ratio);
        strcat(tempo, str_tempo);
        strcat(play, trim);
        sprintf(str_start, "%d", start);
```

```
        strcat(empty, tempo);
        strcat(empty, " &");
        strcat(play, empty);
        char *return_val = malloc(sizeof(play));
        strcpy(return_val, play);
        printf("%s \n", return_val);
        return return_val;
}


// Finds song with similar bpm
int find_song(void) {
        float run_freq = find_freq();
        int i;
        printf("Found song with this freq: %f \n", run_freq);
        for (i = 0; i < 12; i++) {
                if (song_data[i][0] > run_freq) {
                        if (i != 0) {
                                if (run_freq - song_data[i][0] < run_freq -
song_data[i-1][0]) {
                                        return i;
                                } else {
                                        return i - 1;
                                }
                        } else {
                                return 0;
                        }
                }
        }
        return 10;
}

// Plays the song that is chosen
void play_song(int song_index, int song_bpm, int song_length) {
        float run_freq;
        float tempo_ratio;
        int last_length;
        int i = 0;
        while (i < song_length) {
                run_freq = find_freq();
                printf("Run Freq: %f \n", run_freq);
                tempo_ratio = (float)run_freq/song_bpm;
                printf("Tempo ratio %f \n", tempo_ratio);
                const char* play = song_string(song_index, i, 2, tempo_ratio);
                system(play);
                delayMillis(1980.0/tempo_ratio);
                i = i + 2;
        }
        if (i % 2) {
                run_freq = find_freq();
```

```
                tempo_ratio = song_bpm/run_freq;
                const char* play = song_string(song_index, -1, 1, tempo_ratio);
                system(play);
        }
}

void main(void) {
        pioInit();
        pinMode(DONE_PIN, INPUT);
        fp = fopen("/var/www/html/liveData.txt", "w+");
        fprintf(fp, "Loading Data...");
        fflush(fp);
        delayMillis(5000);
        int song_index = find_song();
        char song[500] = "Current Song: ";
        strcat(song, song_names[song_index]);
        strcat(song, "\n");
        strcpy(current_song, song);
        printf("%s \n", current_song);
        play_song(song_index, song_data[song_index][0], song_data[song_index][1]);
}
```

Appendix D: Main C Code

```
#include <stdlib.h>
#include <stdio.h>

void main(void) {
        system("sudo killall -9 play");
        system("sudo killall -9 final");
}
```

Appendix E: HTML Code

```
<!DOCTYPE html>
<html>
<head>
        <title> Lilly and Sabrina's final project </title>
        <meta http-equiv="content-type" content="text-html; charset=utf-8">
        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.0/jquery.min.js"></script>
        <script type="text/javascript" src="reloader.js"></script>
        <link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/cs$
        <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
```

```
</head>
<body>
        <h1 align="center"> Lilly and Sabrina's 155 Final Project </h1>
        </br>
        <form action="cgi-bin/final" method="POST" target="display" id="submit"
align="center">
                <button type= "submit" class="btn btn-success"> Play Song </button>
        </form>
        </br>
        <form action="cgi-bin/cancel" method="POST" align="center" id = "cancel">
                <button type= "submit" class="btn btn-danger"> Cancel Song</button>
        </form>
        </br>
        <div id="currentData" align="center">
                <p> Loading Data...</p>
        </div>
</body>
```

Appendix F: Javascript Code

```
$(document).ready(function () {
        $(document).on('click', '#submit', function() {
                setTimeout(loadSong(), 50000);
                reloadData();
        })

var req;

function reloadData()
{
   var now = new Date();
   url = 'liveData.txt';
   try {
      req = new XMLHttpRequest();
   } catch (e) {
      try {
         req = new ActiveXObject("Msxml2.XMLHTTP");
      } catch (e) {
         try {
            req = new ActiveXObject("Microsoft.XMLHTTP");
         } catch (oc) {
            alert("No AJAX Support");
            return;
         }
      }
   }

   req.onreadystatechange = processReqChange;
```

```
   req.open("GET", url, true);
   req.send(null);
}

function processReqChange()
{
   // If req shows "complete"
   if (req.readyState == 4)
   {
      dataDiv = document.getElementById('currentData');
      // If "OK"
      if (req.status == 200)
      {
         // Set current data text
         dataDiv.innerHTML = req.responseText;
        console.log(req.responseText);
         // Start new timer (1 min)
         setTimeout(reloadData(), 50000);
      }
      else
      {
         // Flag error
         dataDiv.innerHTML = '<p>There was a problem retrieving data: ' +
req.statusText + '</p>';
      }
   }
}

});
```