Real-Time Music Transcriber

Final Project Report

December 9, 2016

E155

Alex Goldstein and Jane Wu

Abstract

While coming up with a melody for a song can be as easy as humming a few notes, transcribing music requires sophisticated audio filtering and signal processing. Our project transcribes audio in real time by displaying notes on an LED matrix, and recording melodies which can be replayed as square waves. The microphone data is filtered through a low pass filter and processed by an FPGA. The Fast Fourier Transform (FFT) implemented on the FPGA determines the frequency of the note that is played. After, the data is sent to the Raspberry Pi, which displays the notes being played on a 16x32 LED matrix. Additional features include a playback button, audio speaker, and potentiometer to control the LED matrix scrolling rate.

Introduction

Our project is a real-time music transcriber that takes in audio input from a microphone, processes the signals using a Fast Fourier Transform (FFT) implemented on the FPGA, and streams the notes on a 16x32 LED matrix connected to the Pi. The Pi also controls an audio speaker, which is used to replay recorded notes.

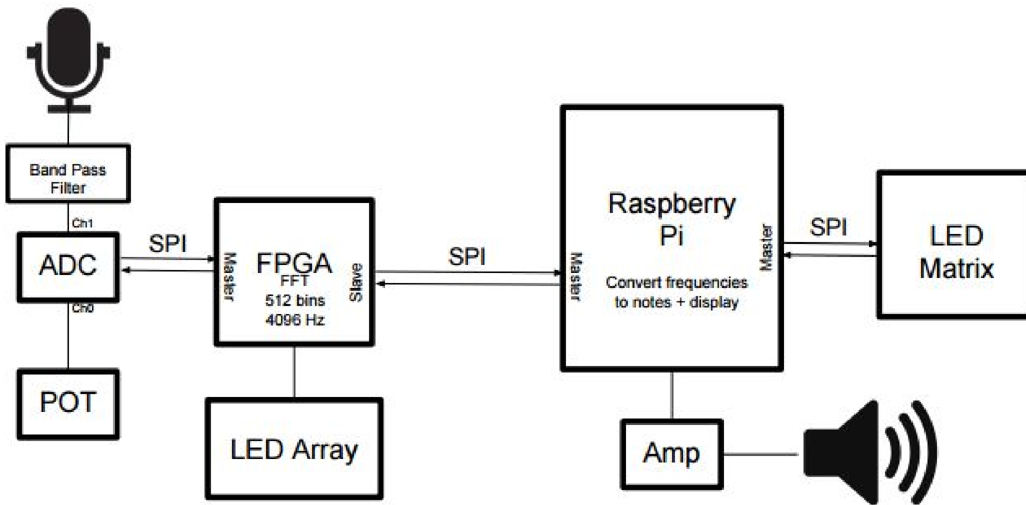

Figure 1: Block diagram of overall system.

New Hardware

The new hardware we worked with was an Adafruit 32x16 Red Green Dual LED Matrix. The matrix is controlled by four HT1632C LED drivers, which each control one quartile of the board. The chips each have a block of 64x4 bit memory, which control a 16x8 section of the board in 2 colors. The chips are controlled with 8 common grounds and 32 rows.

The 4 chips can be selected by sending two signals to the board, CS and CLK. Each board is selected by pulsing CLK 4 times and setting CS to go low for one or more of those pulses, where the pulses where CS is low determines the board selected.

Data is written to the board by sending a 3-bit data write command, a 7-bit address, and then as many consecutive 4-bit blocks of LED data as desired, which if more than one are sent go into consecutive addresses after the one specified. Before data is written, startup commands are sent to all boards to determine how they will be controlled.

Schematics

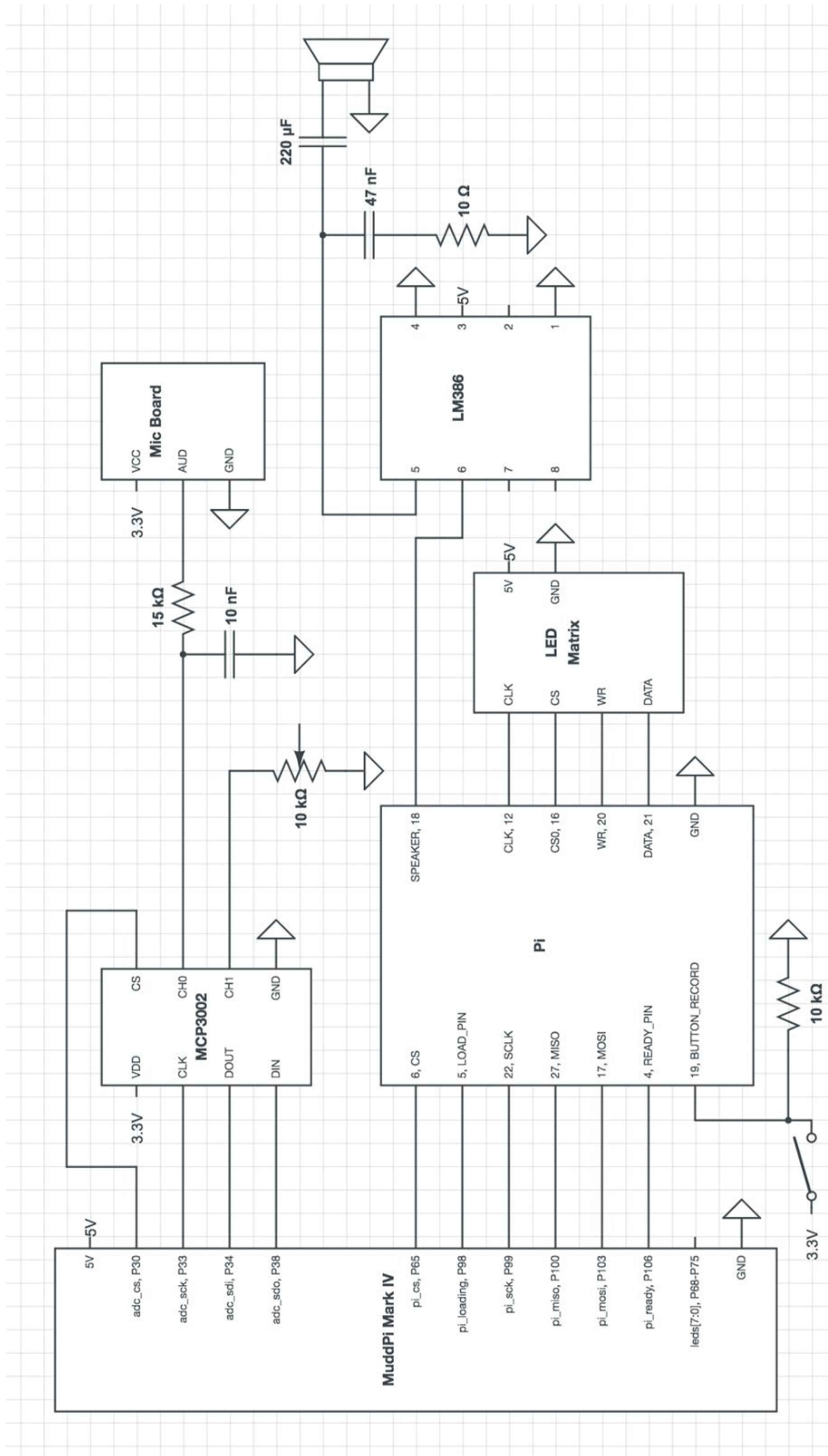

Figure 2: Schematics of overall system.

Microcontroller Design

Main Control

The main control code runs a while loop with two if statements. The first if statement executes when the ready to read signal from the FPGA is at a rising edge. If this edge goes high, the Pi reads in data from the FPGA using SPI and saves it.

The second if statement executes when the Pi is ready to decide on the most recent note and update the display. This is determined by checking if a certain timestep has elapsed since the last update started and is based on the potentiometer data read in.

To update the display, the Pi uses the past FFT bin data sent from the FPGA to determine what note was most likely played. The Pi receives the two highest peaks generated from the hardware FFT. If the second largest bin is nonzero, the second bin's frequency is used in place of the first bin's frequency. This is done to filter out natural harmonics common in higher pitches. If the average bin size sent by the FFT was not higher than our threshold value, the Pi assumes no note was played.

If the recording button is pressed, the Pi records the note in a list. The first time the recording button switches low, the Pi plays back the past recorded notes and clears the recorded list.

LED Matrix Display

The processed notes from the FPGA are displayed on the 32x16 Red Green Dual LED Matrix. They are controlled as documented in the new hardware section. Initialization uses consecutive address writing, Note updating just addresses specific addresses.

Speaker Recording and Playback

During playback, the Pi first adds 26 "rest" notes to the end of the recorded notes for visualization purposes. It then loads the first 26 notes into the list of notes to be visualized, and begins playing the first note for its duration. It then shifts all of the notes in the note list by that duration, and repeats playing the first note in the visualizer until all notes have been played.

Each note is played in the same manner as lab 5, by alternating a GPIO pin that drives the speaker at the note's frequency.

FPGA Design

Microphone Preprocessing

The microphone data is preprocessed using a first order low-pass filter and Hann envelope function. The purpose of the low-pass filter is to filter out possible high frequency harmonics, and the envelope function is applied to improve the accuracy of the hardware FFT (see Figure 1).

FFT Implementation

The FFT hardware implementation is based on the paper titled "The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation" by G. William Slade [1]. The sampling rate (*Fs)* is 4096 Hz, and the number of samples (*N)* is 512. Consequently, the bin resolution of our system is 8 Hz, with 256 bins total. The range of relevant signal frequencies is 261.63Hz (C4) to 1046.50 (C6).

The high level FFT Verilog module is modularized to take the number of samples as an input. The subcomponents in the FFT are the Address Generation Unit (AGU), Butterfly Unit (BFU), and Memory Block modules. Two-port RAM was used to store samples and intermediate data produced by the FFT. Samples are read from the microphone at the sampling rate and stored in RAM. To accurately detect note changes, we always input the most recent 512 samples to the FFT.

Simulation

The FFT implementation was tested with 32 point data prior to deploying on the FPGA. Below is a replication of the sample simulation included in the reference paper. In particular, the timing diagram below shows the final iteration of the FFT, where pairs of data are read and written to the memory blocks in consecutive clock cycles.

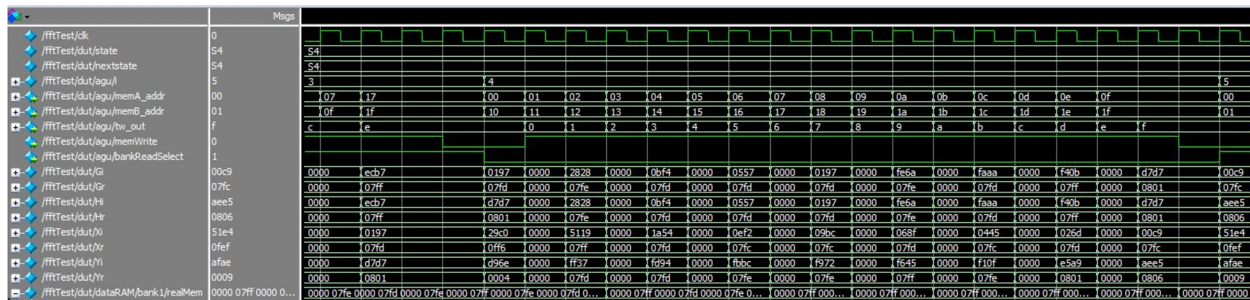

Figure 3: Simulation of Hardware FFT with 32 point test data.

Peak Finding

To determine which note is currently playing, we compare the relevant bins from the FFT to find the two largest peaks. The peak bin numbers and magnitudes are then sent to the Pi via SPI.

Results

We successfully implemented our project, which included completing the hardware FFT, displaying correct notes on the LED matrix, and adjustable scrolling speed using the potentiometer. We are able to detect notes in the frequency range 261.63Hz (C4) to 1046.50 (C6) as discussed above. We also tested our project on a number of audio sources, and found the most success with pure tones and voice pitches.

In the process, we encountered challenges in both the FPGA and Raspberry Pi components of the project. For the hardware FFT, we discovered errors and asynchronous design in the reference paper. To control the LED matrix, we reverse engineered a sample Arduino library to determine the timing for configuring each of the four chips onboard. We also wrote our own library to handle nanosecond timing delays using the clock_nanosleep library. In implementing SPI, we encountered unexpected behavior with the Pi's SPI interface, and we ultimately implemented our own SPI library using GPIO pins.

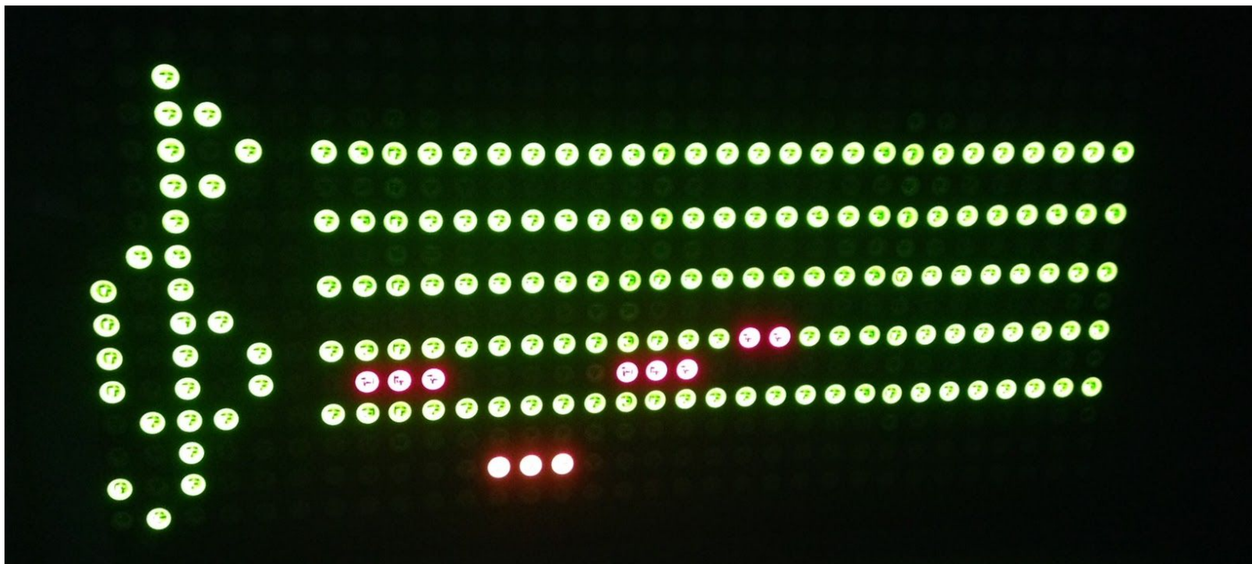

Figure 4: Completed project transcribing voice input.

References

Digital Design and Computer Architecture, Sarah L. Harris & David Money Harris

Datasheets:

- HT1632C: https://cdn-shop.adafruit.com/datasheets/ht1632cv120.pdf
- Microphone Breakout: http://cdn.sparkfun.com/datasheets/Sensors/Sound/CEM-C9745JAD462P2.54R.pdf

Design References:

- Hardware FFT: http://pages.hmc.edu/mspencer/fa15/e155/resources/FFTtutorial121102.pdf
- LED Matrix: https://github.com/gauravmm/HT1632-for-Arduino

Parts List

Part	Source	Vendor Part #	Price
32x16 Red Green Dual Color LED Dot Matrix	Adafruit	DE-DP14211	$39.95
Electret Microphone Breakout	Sparkfun	BOB-12758 ROHS	$5.95
Rotary Potentiometer - 10k Ohm, Linear	Sparkfun	COM-09939	0.95
Silver Potentiometer Dial	Sparkfun	COM-10001 ROHS	$1.50

Appendix A: FPGA Verilog modules

```
// ag_jw.sv
// Alex Goldstein, Jane Wu
// agoldstein@hmc.edu, jhwu@hmc.edu, 6 December 2016
//
// Hardware FFT implementation

module ag_jw
	#(parameter logN = 9)
	(
	input logic clk, reset,
	input logic pi_loading, pi_sck, pi_mosi, pi_cs,
	output logic pi_miso, pi_ready,
	input logic adc_sdi,
	output logic adc_sck, adc_sdo, adc_cs,
	output logic[7:0] leds);


	logic [17:0] count;
	logic startFFT, ldwDone, adcDone, fftDone, peakFound;

	logic firWrite, memWrite, bankReadSelect;
	logic loadDataWrite, bank0WriteEN, bank1WriteEN;

	logic [logN-1:0] loadDataAddr, memGAddr, memHAddr, writeGAddr, writeHAddr;
	logic [logN-2:0] tw_addr;
	logic [15:0] dataR, dataI, Xr, Xi, Yr, Yi;
	logic [15:0] Gr, Gi, Hr, Hi;

	logic [9:0] micData, potData;
	logic [23:0] potMicData, fftData, secondfftData;

	logic peakReadEN;
	logic [logN-1:0] peakReadAddr;
	logic [7:0] outputBin, secondBin;
	logic[15:0] outputValue, secondValue;

	assign leds = outputBin;//dataR[9:2];


	// counter on how often to read in data and do the fft
	always_ff @(posedge clk, posedge reset)
	begin
		if (reset) count <= 0;
		else
			begin
			count <= count + 1;
			if (count == 9765)
				count <= 0;
```

```
                end
        end

        typedef enum logic [2:0] {S0, S1, S2, S3, S4, S5} statetype;
        // s0: waiting for clock to say start again
        // s1: read in from ADC
        // s2: Update FIR
        // s3: load data to FFT memory bank
        // s4: run FFT
        // s5: find bins

        statetype state, nextstate;
        always_ff @(posedge clk, posedge reset)
        begin
                if(reset) state <= S0;
                else state <= nextstate;
        end

        always_comb
                case (state)
                        S0: if (count == 1) nextstate = S1;
                                else nextstate = S0;
                        S1: if(adcDone) nextstate = S2;
                        else nextstate = S1;
                        S2: nextstate = S3; // only run for one clock cycle
                        S3: if(ldwDone) nextstate = S4;
                        else nextstate = S3;
                        S4: if(fftDone) nextstate = S5;
                        else nextstate = S4;
                        S5: if(peakFound) nextstate = S0;
                        else nextstate = S5;
                        default: nextstate = S0;
                endcase


        spi_slave slave(clk, pi_cs, pi_loading, pi_sck, pi_mosi, potMicData, fftData,
secondfftData, pi_miso);
        spi_master master(clk, reset, adc_sdi, count, adc_sck, adc_sdo, adc_cs, potData,
micData, adcDone);

        assign pi_ready = (state == S0) & count < 7065 & count > 1; // allow  2700 clock
cycles for pi spi to run
        assign potMicData = {2'b00, potData, 2'b00, micData};
        assign fftData = {outputBin, outputValue};
        assign secondfftData = {secondBin, secondValue};

        //////// DATA LOADING/FIR ////////
        assign firWrite = (state == S2);
        assign loadDataWrite = (state == S3);
        assign startFFT = (state == S4);
```

```
	always_ff @(posedge clk, posedge reset)
	begin
		if (reset) loadDataAddr = 0;
		else
			if (loadDataWrite)
				{ldwDone, loadDataAddr} <= loadDataAddr + 1;
	end

    logic [logN-1:0] correctedLDAddr;
	assign correctedLDAddr = loadDataAddr + 1;

	FIR #(logN) fir(clk, reset, firWrite, {2'b00,micData}, correctedLDAddr, dataR);
	assign dataI = 0;


	//////// FFT MODULES ////////
	AGU #(logN) agu(startFFT, clk, fftDone, memGAddr, memHAddr, tw_addr, memWrite,
bankReadSelect);
	always_ff @(posedge clk)
	begin
		writeGAddr <= memGAddr;
		writeHAddr <= memHAddr;
	end

	BFU #(logN) bfu(Gr, Gi, Hr, Hi, tw_addr, Xr, Xi, Yr, Yi);

	assign bank0WriteEN = bankReadSelect & memWrite; //Read from 1, write to 0
	assign bank1WriteEN = ~bankReadSelect & memWrite; //Read from 0, write to 1

	dataBlocks #(logN) dataRAM(clk, loadDataWrite, bank0WriteEN, bank1WriteEN,
bankReadSelect, peakReadEN,
	loadDataAddr, memGAddr, memHAddr, writeGAddr, writeHAddr, peakReadAddr,
	dataR, dataI, Xr, Xi, Yr, Yi, Gr, Gi, Hr, Hi);

	//// Peak Finding ////
	assign peakReadEN = (state == S5);
	peaks #(logN) peakFinder(clk, reset, peakReadEN, Gr, Gi, peakReadAddr, outputBin,
secondBin, outputValue, secondValue, peakFound);

endmodule

/////////////////////////////////////////////
// SPI Master
//   Sends mic, pot, and bin data to the FPGA on request
/////////////////////////////////////////////
module spi_slave(
		input logic clk, cs, loading, sck, mosi,
		input logic [23:0] potMicData, fftData, secondfftData,
		output logic miso);
```

```
logic [23:0] send;
logic [7:0] receive;
logic [2:0] mode;
logic [2:0] count; //Count 8 bits
logic senddelayed;
logic wasloading;

logic [23:0] sendData;

assign mode = receive[7:5];

assign senddelayed = sendData[23];

// 3-bit counter tracks when full 3 bytes are transmitted
always_ff @(negedge sck, posedge cs)
begin
	if(cs) begin
		count <= 0;
	end
	else count <= count + 1;
end

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

// Align miso to falling edge of sck
always_ff @(posedge sck)
begin
	if (wasloading) sendData <= (send << 1);
	else sendData <= (sendData << 1);
end

always_ff @(posedge sck)
	wasloading <= loading;

assign miso = (wasloading) ? send[23] : senddelayed;

//Mode select
always_comb
case(mode)
	3'b100: send = potMicData;
	3'b101: send = fftData;
	3'b111: send = secondfftData;
```

```
		default: send = fftData;
	endcase

endmodule

/////////////////////////////////////////////
// SPI Master
//   Reads in microphone and potentiometer data from the ADC
/////////////////////////////////////////////
module spi_master(input logic clk, reset, sdi,
					input logic [17:0] count,
					output logic sck, sdo, cs,
					output logic[9:0] potData, micData,
					output logic adcDone);
	logic[4:0]   sckCounter;
	logic [15:0] dataOut;
	assign sck = sckCounter[4];

	logic micRead, potRead, read, channel, readDone;

	// we read if we only try to read from one
	// ch0 is mic, ch1 is pot
	assign read = micRead ^ potRead;
	assign channel = potRead;

	assign cs = ~read;

	assign adcDone = (count > 1000);

	logic [3:0] spiCounter;
	logic [15:0] spiCommand;
	assign spiCommand = {13'bxxxx_xxxx_xxxx_1,channel, 2'b11};

	always_ff @(posedge clk, posedge reset)
		if (reset) sckCounter <= 0;
		else sckCounter <= sckCounter + 1;

	// logic to determine when to start reading in
	always_ff @(posedge clk, posedge reset)
		begin
		if (reset)
			begin
			micRead <= 0;
			potRead <= 0;
			end
		else
			begin
			if (count == 5)
				micRead <= 1;
			else if (count == 1000)
```

```
				potRead <= 1;
			else if (readDone)
				begin
				micRead <= 0;
				potRead <= 0;
				end
			end
		end

	// acutal spi code
	always_ff @(posedge sck, posedge reset)
		begin
		if (reset)
			begin
			readDone <= 0;
			spiCounter <= 0;
			end
		else if (read)
			begin
			sdo <= spiCommand[spiCounter];
			if ((spiCounter >= 5) && (spiCounter < 15))
				if (micRead)
					micData[14-spiCounter] <= sdi;
				else
					potData[14-spiCounter] <= sdi;
			else if (spiCounter == 15)
				readDone <= 1;

			spiCounter <= spiCounter + 1;
			end
		else
			begin
			readDone <= 0;
			spiCounter <= 0;
			end
		end
endmodule

/////////////////////////////////////////////
// peaks
//   Find the largest peak in 110 bins (C4 to C6, roughly)
/////////////////////////////////////////////
module peaks
#(parameter logN = 5)
(input logic clk, reset, enable,
	input logic signed [15:0] peakValueRe, peakValueIm,
	output logic [logN-1:0] peakAddr,
	output logic [6:0] firstBin, secondBin,
	output logic [15:0] firstScore, secondScore,
	output logic peakFound);
```

```
logic [logN-1:0] bin, lastBin, bestBin, secondBestBin;
logic signed [15:0] score, bestScore, secondBestScore;

logic signed [31:0] bigScore;
assign bigScore = peakValueIm*peakValueIm + peakValueRe*peakValueRe;

assign score = bigScore[31:16];
assign peakAddr = bin + 30;

always_ff @(posedge clk, posedge reset)
	begin
	if (reset)
		begin
		bin <= 0;
		end
	else if (enable)
		if (bin < 110)// this is where we lose modularity
			begin
			bin <= bin + 1;
			peakFound <= 1'b0;
			end
		else
			begin
			firstBin <= bestBin;
			firstScore <= bestScore;
			secondBin <= secondBestBin;
			secondScore <= secondBestScore;
			bin <= 0;
			peakFound <= 1'b1;
			end
		end

always_ff @(posedge clk)
	lastBin <= bin;

always_ff @(posedge clk)
	begin
	if (~enable)
		begin
		bestScore <= 0;
		secondBestScore <= 0;
		end
	else if (score > bestScore)
		begin
		secondBestScore <=bestScore;
		secondBestBin <= bestBin;
		bestScore <= score;
		bestBin <= lastBin;
		end
```

```
            end

endmodule

/////////////////////////////////////////////
//FIR Filter
// rotating memory, allows reading off multiplied blocks
//     with envelope function, with bit reversal of the address
/////////////////////////////////////////////
module FIR
#(parameter logN = 5)
(input logic clk, reset, dataWrite,
 input logic signed [11:0] dataIn,
 input logic unsigned [logN-1:0] readAddr,
 output logic signed[15:0] dataOut);

      logic unsigned [logN-1:0] startAddr, flippedAddr;
      logic [3:0] envelope[0:(1 << (logN)) - 1];
      initial   $readmemh("envelope.txt", envelope);

      logic signed [15:0] datum, oldestData;
      logic signed [4:0] env;

      logic[logN + 11: 0] sum;
      logic[11:0] avg;
      assign avg = (sum >> (logN));

      logic [logN-1:0] sinceResetCounter;
      logic steadySum;
      assign steadySum = sinceResetCounter[logN-1];

      logic[logN-1:0] addrA, addrB;

      assign addrA = dataWrite ? startAddr-1 : flippedAddr;
      assign addrB = startAddr-1;

      twoport3 firRAM(clk, addrA, addrB, dataIn, dataIn, dataWrite, dataWrite, datum,
oldestData);


      genvar k;
      generate
      for(k = 0; k <logN; k = k + 1)
            begin: swiz
            assign flippedAddr[k] = readAddr[logN-1-k];
            end
      endgenerate

      always_ff @(posedge clk, posedge reset)
            if (reset)
```

```
                sinceResetCounter <= 0;
            else if (~steadySum)
                sinceResetCounter <= sinceResetCounter + 1;


    always_ff @(posedge clk, posedge reset)
            begin
            if (reset)
               begin
                startAddr <= 0;
                sum <= 0;
                end
            else if (dataWrite)
                begin
                // update the sum to follow the current data
                if (steadySum)
                    sum <= sum + dataIn - oldestData;
                else
                    sum <= sum + dataIn;
                startAddr <= startAddr - 1;
                end
            else
                begin
                env <= {1'b0,envelope[flippedAddr]};
                end
            end

    assign dataOut = ((datum - 500)*env) >> 3;

endmodule

/////////////////////////////////////////////
// dataBlocks
//   Reading/writing data
/////////////////////////////////////////////
module dataBlocks
#(parameter logN = 5)
(input logic clk, loadDataWrite, bank0WriteEN, bank1WriteEN, bankReadSelect, peakReadEN,
input logic [logN-1:0] loadDataAddr, readGAddr, readHAddr, writeGAddr, writeHAddr,
peakReadAddr,
input logic [15:0] dataR, dataI, Xr, Xi, Yr, Yi,
output logic [15:0] Gr, Gi, Hr, Hi);

    logic bank0_AWrite;
    logic [logN-1:0] addrA0, addrA1, addrB0, addrB1;
    logic [15:0] dataA_r, dataA_i;

    assign bank0_AWrite = loadDataWrite | bank0WriteEN;
```

```
	assign addrA0 = loadDataWrite ? loadDataAddr : (bank0WriteEN ? writeGAddr :
readGAddr);
	assign addrA1 = peakReadEN ? peakReadAddr :  (bank1WriteEN ? writeGAddr : readGAddr);
	assign addrB0 = bank0WriteEN ? writeHAddr : readHAddr;
	assign addrB1 = bank1WriteEN ? writeHAddr : readHAddr;

	assign dataA_r = loadDataWrite ? dataR : Xr;
	assign dataA_i = loadDataWrite ? dataI : Xi;

	logic [15:0] G0r, G0i, H0r, H0i, G1r, G1i, H1r, H1i;
	twoport3 ram0re(clk, addrA0, addrB0,dataA_r, Yr, bank0_AWrite, bank0WriteEN,G0r, H0r);
	twoport3 ram0im(clk, addrA0, addrB0,dataA_i, Yi, bank0_AWrite, bank0WriteEN,G0i, H0i);
	twoport3 ram1re(clk, addrA1, addrB1, Xr, Yr, bank1WriteEN, bank1WriteEN,G1r, H1r);
	twoport3 ram1im(clk, addrA1, addrB1, Xi, Yi, bank1WriteEN, bank1WriteEN,G1i, H1i);

	assign Gr = (bankReadSelect | peakReadEN) ? G1r : G0r; //If bankReadSelect is 1 or
we're reading peaks, read from 1
	assign Gi = (bankReadSelect | peakReadEN) ? G1i : G0i;
	assign Hr = bankReadSelect ? H1r : H0r;
	assign Hi = bankReadSelect ? H1i : H0i;

endmodule

/////////////////////////////////////////////
// AGU
//   Address Generator Unit
/////////////////////////////////////////////
module AGU
	#(parameter logN = 5)
	(input logic startFFT,
	input logic clk,
	output logic doneFFT,
	output logic [logN-1:0] memA_addr,
	output logic [logN-1:0] memB_addr,
	output logic [logN-2:0] tw_out,
	output logic memWrite,
	output logic bankReadSelect);
	logic startFFT2, clrHold;
	logic [1:0] delayCounter;
	logic [3:0] i; // 0-logN Level counter, logN assumed < 15
	logic [logN-2:0] j, tw_addr; // 0-N/2 Index counter
	logic [logN-1:0] j_shift; // j left shifted by 1


	// sketch things to make twiddle factor bit shifting work for
	// arbitrary logN without using an async clock
	logic [logN-2:0] ones = ~0;
	logic [logN-2:0] zeros = 0;

	always_ff @(posedge clk)
```

```
begin
	startFFT2 <= startFFT;
end

assign clrHold = ~startFFT2 & startFFT;

always_ff @(posedge clk, posedge clrHold)
begin
	if (clrHold)
		begin
			j <= 0;
			delayCounter <= 0;
		end
	else if (j != (1 << (logN-1)) - 1) // j < n/2 - 1
		j <= j + 1;
	else
		begin
		delayCounter <= delayCounter + 1;
		if (delayCounter == 2)
			begin
			 j <= 0;
			 delayCounter <= 0;
			 end
		end
end

always_ff @(posedge clk, posedge clrHold)
begin
	if (clrHold)
		begin
		i <= 0;
		doneFFT <= 0;
		end
	else if (delayCounter == 2)
		if (i < logN)
			i <= i + 1;
		else
			begin
			i <= 0;
			doneFFT <= 1;
			end
end

assign j_shift = j << 1;
assign memA_addr = (j_shift << i) | (j_shift >> (logN - i));
assign memB_addr = ((j_shift + 1) << i) | ((j_shift + 1) >> (logN - i));

assign tw_addr = ({ones, zeros} >> i) & j;
always_ff @(posedge clk)
	tw_out <= tw_addr;
```

```
	assign bankReadSelect = i[0];
	assign memWrite = (~doneFFT & ~((delayCounter == 2) | (j == 0))); //memWrite low only
when j = 0 and clock cycle before

endmodule



//////////////////////////////////////////
//Two Port RAM Template
// Taken from the internet, which was taken from verilog
// example 2-port ram code
//////////////////////////////////////////
module twoport3(
input logic clk,
input logic [8:0] aa, ab,
input logic [15:0] da, db,
input logic wa, wb,
output logic [15:0] qa, qb);
logic [15:0] mem [511:0];
always_ff @(posedge clk) begin
if (wa) begin
mem[aa] <= da;
qa <= da;
end else qa <= mem[aa];
end
always_ff @(posedge clk) begin
if (wb) begin
mem[ab] <= db;
qb <= db;
end else qb <= mem[ab];
end
endmodule

/////////////////////////////////////////////
// BFU
//   Butterfly Unit
//   Performs 2-point FFT
/////////////////////////////////////////////
module BFU
#(parameter logN = 5)
                    (input logic signed [15:0]  G_real, G_imag, H_real, H_imag,
                    input logic [logN-2:0]  Tw_addr,
                    output logic signed [15:0]   Xr, Xi, Yr, Yi);
	logic signed [35:0] temp_r1, temp_i1, temp_r2, temp_i2;
	logic signed [35:0] temp_r, temp_i;

	// Twiddle factors implemented as a ROM
	logic [15:0] twf_real[0:(1 << (logN-1)) - 1]; //TODO: create text files
```

```
	logic [15:0] twf_imag[0:(1 << (logN-1)) - 1];
	initial   $readmemh("twf_real.txt", twf_real);
	initial   $readmemh("twf_imag.txt", twf_imag);

	logic signed [15:0] t_real, t_imag;

	assign t_real = twf_real[Tw_addr];
	assign t_imag = twf_imag[Tw_addr];

	assign temp_r = (H_real * t_real) - (H_imag * t_imag);
	assign temp_i = (H_real * t_imag) + (H_imag * t_real);

	logic signed [15:0] temp_r_s, temp_i_s;

	assign temp_r_s = temp_r[30:15];
	assign temp_i_s = temp_i[30:15];

	assign Yr = G_real - temp_r_s;
	assign Yi = G_imag - temp_i_s;
	assign Xr = G_real + temp_r_s;
	assign Xi = G_imag + temp_i_s;
endmodule
```

Appendix B: Raspberry Pi C Files

```c
// main.c
// Alex Goldstein, Jane Wu
// agoldstein@hmc.edu, jhwu@hmc.edu, 6 December 2016
//
// Main project file

////////////////////////////////////////////////
// #includes
////////////////////////////////////////////////

#include <stdio.h>
#include <unistd.h>
#include  <time.h>
#include "realpio.h"
#include "binsToNotes.h"
#include "led_matrix.h"

////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////
#define LOAD_PIN 5
#define READY_PIN 4
#define FFT 0b10100000
#define MICPOT 0b10000000
#define NUM_BINS 108
#define POT_MAX 1024

#define SPI_RATE 1000000
////////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////////

void getData(char mode);
long getTimeMilis();
void updateData();
void updateDisplay(int note);
char fpgaReady();
void printBytes(char*, int);

////////////////////////////////////////////////
// Variables
////////////////////////////////////////////////

struct timespec spec;
char lastReady; // boolean

unsigned char fftData[3];
unsigned char adcData[3];
```

```
unsigned char fftResults[4096];
size_t fftCounter;

unsigned short potData, micData;
short fftValue;
unsigned char fftBin;

int notes[26];

////////////////////////////////////////////////
// main
////////////////////////////////////////////////

int main(void) {
  pioInit();
  matrixInit(); //Display initial staff
  pinMode(LOAD_PIN, OUTPUT);
  pinMode(READY_PIN, INPUT);
  spiInit(SPI_RATE, 0); //CS0 is "reset"

  long lastTime = getTimeMilis();
  fftCounter = 0;
  potData = 0;
  lastReady = 0;

  int j;
  for(j = 0; j < 26; j++) {
    notes[j] = -1;
  }

  while(1) {
    if(fpgaReady()) {
      getData(FFT);
      getData(MICPOT);
      updateData();
      //printf("Mic: %d, pot: %d, bin: %d, val: %d\n",micData, potData, fftBin, fftValue);
      fftResults[fftCounter] = fftBin;
      fftCounter++;
    }
    //printf("Time: %d\n",getTimeMilis());

    // range is 1/4 to 1/2 of a second (60-120 bpm)
    if (getTimeMilis() > lastTime + 250*(2.0*POT_MAX-potData)/POT_MAX) {
      lastTime = getTimeMilis();

      int noteCounter[15];

      size_t i;
      for (i = 0; i < 15; ++i) {
```

```
        noteCounter[i] = 0;
      }
      for (i = 0; i < fftCounter; ++i) {
        noteCounter[BINS_TO_NOTES[fftResults[i]]]++;
      }
      int bestNote = 0;
      short mostHits = 0;
      for (i = 0; i < 15; ++i) {
        if (noteCounter[i] > mostHits) {
          mostHits = noteCounter[i];
          bestNote = i;
        }
      }
      updateDisplay(bestNote);
      fftCounter = 0;
    }
  }
  return 0;
}

void updateDisplay(int note) {
  int j;

  for(j = 0; j < 26; j++) {
    if(notes[j] >= 0) {
      updateNote(j+6, 14-notes[j], 0);
    }
  }
  for(j = 0; j < 25; j++) {
    notes[j] = notes[j+1];
  }
  notes[25] = note;

  for(j = 0; j < 26; j++) {
    if(notes[j] >= 0) {
      updateNote(j+6, 14-notes[j], 1);
    }
    fflush(stdout);
  }
}
////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////

void getData(char mode) {
  unsigned char data[3];
  SPI0CSbits.TA = 1;          // turn SPI on with the "transfer active" bit

  digitalWrite(LOAD_PIN, 1); // Sending mode to FPGA in 3 bytes
  delayMicroseconds(10);
```

```
  spiSendReceive(mode);
  delayMicroseconds(10);

  digitalWrite(LOAD_PIN, 0);

  int i;
  for(i = 0; i < 3; ++i) {
    data[i] = spiSendReceive(0);
    //printBytes(data[i], 1);
    if(mode == FFT) fftData[i] = data[i];
    else adcData[i] = data[i];
  }
  SPI0CSbits.TA = 0;          // turn off SPI
}

void updateData() {
  potData = (unsigned short)(adcData[0] << 4) | adcData[1] >> 4;
  micData = (unsigned short)(adcData[1] << 8 & 0x0FFF) | adcData[2];
  fftBin = fftData[0];
  fftValue = (short)(fftData[1] << 8) | fftData[2];
}

long getTimeMilis() {
       clock_gettime(CLOCK_REALTIME, &spec);
       return 1000*spec.tv_sec + spec.tv_nsec / 1e6;
}

char fpgaReady() {
  char ready = digitalRead(READY_PIN);
  if (ready && !lastReady) {
    lastReady = ready;
    return 1;
  }
  else {
    lastReady = ready;
    return 0;
  }
}

void printBytes(char *text, int numBytes) {
  int i;

  for(i = 0; i < numBytes; ++i) {
    printf("%02x ",text[i]);
  }
  printf("\n");
}
```

```
// led_matrix.h
// Alex Goldstein, Jane Wu
// agoldstein@hmc.edu, jhwu@hmc.edu, 21 November 2016
//
// Control 16x32 dual color LED matrix


////////////////////////////////////////////////
// #includes
////////////////////////////////////////////////

#include <stdio.h>
#include <unistd.h>
#include <time.h>
//#include "easypio.h"
//#include "realpio.h"
#include "led_matrix_background.h"

////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////

#define CLK 12
#define CS0 16
#define WR 20
#define DATA 21

#define COM_SIZE 16
#define OUT_SIZE 32
#define NUM_CHANNEL 2
#define NUM_ACTIVE_CHIPS 4
#define USE_NMOS 1
#define PIXELS_PER_BYTE 8

#define ADDR_SPACE_SIZE (COM_SIZE * OUT_SIZE / PIXELS_PER_BYTE)

// Commands
#define ID_CMD 0b100
#define ID_WR 0b101
#define ID_LEN 3 // 3-bit IDs

#define CMD_SYSDIS 0x00 // Turn off system oscil
#define CMD_SYSEN 0x01 // Enable system oscil
#define CMD_LEDOFF 0x02
#define CMD_LEDON 0x03
#define CMD_BLOFF 0x08 // Blink ON
#define CMD_BLON 0x09 // Blink OFF
#define CMD_RCCLK 0x18 // Master Mode, external clock
#define CMD_COMS00 0x20 // Common Option 1
#define CMD_PWM_T 0xA0 // PWM duty cycle - template
```

```
#define CMD_PWM(lvl) (CMD_PWM_T | (lvl - 1))

#define CMD_LEN 8 // Commands are 8 bits long, excluding trailing bit
#define ADDR_LEN 7
#define WORD_LEN 4

#define ALL_CHIPS 4
#define NO_CHIPS 5

#define ERASE 0
#define STAFF 1

#define CLK_DELAY 10

////////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////////

void matrixInit(void);
void render(char);
void renderTarget(unsigned char);
void clear(void);
void delayMatrix(void);
void writeCommand(char);
void writeData(char, int);
void writeSingleBit(void);
void pulseClk(void);
void selectChips(int);
void updateNote(char, char, char);

////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////

void matrixInit(void) {
  pinMode(CS0, OUTPUT);
  pinMode(CLK, OUTPUT);
  pinMode(WR, OUTPUT);
  pinMode(DATA, OUTPUT);

  digitalWrite(CS0, 0);
  digitalWrite(CLK, 0);
  digitalWrite(WR, 0);
  digitalWrite(DATA, 0);

  selectChips(NO_CHIPS);
  selectChips(ALL_CHIPS);
  digitalWrite(CS0, 0);
  writeData(ID_CMD, ID_LEN); // Command mode
```

```
  writeCommand(CMD_SYSDIS); // Turn off system oscillator

  writeCommand(CMD_COMS00);

  writeCommand(CMD_RCCLK); // Master Mode, external clock

  writeCommand(CMD_BLOFF);

  writeCommand(CMD_SYSEN); // Turn on system
  writeCommand(CMD_LEDON); // Turn on LED duty cycle generator
  writeCommand(CMD_PWM(16)); // PWM 16/16 duty

  selectChips(NO_CHIPS);
  delayMicroseconds(100);
  render(STAFF);
}

void render(char image) {
  char chip;
  for(chip = 0; chip < 4; chip++) {
    selectChips(chip);

    // Output data
    writeData(ID_WR, ID_LEN);
    writeData(0, ADDR_LEN);

    int j;
    if (image == STAFF) {
      for(j = 0; j < 32; j++) {
        if (1) {
          writeData(MATRIX_BACKGROUND[chip][j] >> WORD_LEN, WORD_LEN);
          writeData(MATRIX_BACKGROUND[chip][j], WORD_LEN);
        }
      }
    }
    else if (image == ERASE) {
      for(j = 0; j < 64; j++) {
       writeData(0x00,4);
      }
    }
  }
}

void selectChips(int chip) {

  if (chip == ALL_CHIPS)
    digitalWrite(CS0, 0);
  if (chip == NO_CHIPS)
    digitalWrite(CS0, 1);
```

```
  int i;
  for(i = 0; i < NUM_ACTIVE_CHIPS; i++) {
    if(i == chip) {
      digitalWrite(CS0, 0);
      delayMatrix();
      pulseClk();
      digitalWrite(CS0, 1);
      delayMatrix();
    }
    else {
      pulseClk();
    }
  }
  delayMatrix();
}

void updateNote(char x, char y, char on) {
  char chip = 2*(1 - y/8) + (1-x/16);

  char greenAddress = 2*(x%16);
  char redAddress = 32 + 2*(x%16);

  char redData = on << (7-y%8);
  char greenData = MATRIX_BACKGROUND[chip][x%16] & ~(on << (7-y%8));

  selectChips(chip);
  writeData(ID_WR,ID_LEN);
  writeData(greenAddress, 7);
  writeData(greenData, 8);

  selectChips(chip);
  writeData(ID_WR,ID_LEN);
  writeData(redAddress, 7);
  writeData(redData, 8);
}

void delayMatrix(void) {
  delayMicroseconds(CLK_DELAY);
}

void writeCommand(char data) {
  writeData(data, CMD_LEN);
  writeSingleBit();
  delayMatrix();
}

// PRECONDITION: WR is LOW
void writeData(char data, int len) {
  int j;
  for(j = 0; j < len; j++) {
```

```
    digitalWrite(DATA, ((data >> (len-1-j)) & 0b1));
    delayMatrix();
    digitalWrite(WR, 1);
    delayMatrix();
    digitalWrite(WR, 0);
    delayMatrix();
  }
}

// PRECONDITION: WR is LOW
void writeSingleBit(void) {
  digitalWrite(DATA, 0);
  delayMatrix();
  digitalWrite(WR, 1);
  delayMatrix();
  digitalWrite(WR, 0);
  delayMatrix();
}

void pulseClk(void) {
  digitalWrite(CLK, 1);
  delayMatrix();
  digitalWrite(CLK, 0);
  delayMatrix();
}
```

```
// realpio.h
// Alex Goldstein, Jane Wu
// agoldstein@hmc.edu, jhwu@hmc.edu, 6 December 2016
//
// Adapted from Lab 5

#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

////////////////////////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////////////////////////

// GPIO FSEL Types
#define INPUT  0
#define OUTPUT 1
#define ALT0   4
#define ALT1   5
#define ALT2   6
#define ALT3   7
#define ALT4   3
#define ALT5   2

#define GPFSEL   ((volatile unsigned int *) (gpio + 0))
#define GPSET    ((volatile unsigned int *) (gpio + 7))
#define GPCLR    ((volatile unsigned int *) (gpio + 10))
#define GPLEV    ((volatile unsigned int *) (gpio + 13))

// Physical addresses
#define BCM2836_PERI_BASE        0x3F000000
#define GPIO_BASE               (BCM2836_PERI_BASE + 0x200000)
#define SPIO_BASE          (BCM2836_PERI_BASE + 0x204000)
#define BLOCK_SIZE (4*1024)
#define TIMER_BASE 0x3F003000

//Speaker pin
#define SPEAKER 23

// Pointers that will be memory mapped when pioInit() is called
volatile unsigned int *gpio; //pointer to base of gpio
volatile unsigned int *sys_timer;
volatile unsigned int *spi;

////////////////////////////////////////////////////////////////////
// SPI Registers
```

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

typedef struct
{
	unsigned CS		:2;
	unsigned CPHA	:1;
	unsigned CPOL	:1;
	unsigned CLEAR		:2;
	unsigned CSPOL		:1;
	unsigned TA		:1;
	unsigned DMAEN		:1;
	unsigned INTD		:1;
	unsigned INTR		:1;
	unsigned ADCS	:1;
	unsigned REN		:1;
	unsigned LEN		:1;
	unsigned LMONO		:1;
	unsigned TE_EN		:1;
	unsigned DONE	:1;
	unsigned RXD		:1;
	unsigned TXD		:1;
	unsigned RXR		:1;
	unsigned RXF		:1;
	unsigned CSPOL0	:1;
	unsigned CSPOL1	:1;
	unsigned CSPOL2	:1;
	unsigned DMA_LEN	:1;
	unsigned LEN_LONG	:1;
	unsigned			:6;
}spi0csbits;
#define SPI0CSbits (* (volatile spi0csbits*) (spi + 0))
#define SPI0CS (* (volatile unsigned int *) (spi + 0))

#define SPI0FIFO (* (volatile unsigned int *) (spi + 1))
#define SPI0CLK (* (volatile unsigned int *) (spi + 2))
#define SPI0DLEN (* (volatile unsigned int *) (spi + 3))

//////////////////////////////////////////////////////////////////////
// General Functions
//////////////////////////////////////////////////////////////////////

void pinMode(int pin, int function) {
	int reg = pin / 10;
	int offset = (pin % 10) * 3;
	GPFSEL[reg] &= ~((0b111 & ~function) << offset);
	GPFSEL[reg] |= ((0b111 & function) << offset);
}

int digitalRead(int pin) {
    int reg = pin / 32;
```

```
    int offset = pin % 32;

    return (GPLEV[reg] >> offset) & 0x00000001;
}

void digitalWrite(int pin, int val)
{
       int reg = pin / 32;
       int offset = pin % 32;
       volatile unsigned int* loc = gpio + reg + 7;
       if (val){
              *(loc) = 1 << offset;
       }
       else {
              loc += 3;
       }      *(loc) = 1 << offset;
}

void delayMicroseconds(unsigned int micro) {
       struct timespec duration;
       duration.tv_sec = 0;
       duration.tv_nsec = micro * 1000;

       clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &duration, NULL);
}

void delayNanoseconds(unsigned int nano) {
       struct timespec duration;
       duration.tv_sec = 0;
       duration.tv_nsec = nano;

       clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &duration, NULL);
}

void pioInit() {
       int  mem_fd;
       void *reg_map;
       void *timer_map;
       void *spi_map;

       // /dev/mem is a psuedo-driver for accessing memory in the Linux filesystem
       if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
             printf("can't open /dev/mem \n");
             exit(-1);
       }

       reg_map = mmap(
              NULL,             //Address at which to start local mapping (null means
don't-care)
              BLOCK_SIZE,       //Size of mapped memory block
```

```
        PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
        MAP_SHARED,       // This program does not have exclusive access to this memory
        mem_fd,           // Map to /dev/mem
        GPIO_BASE);       // Offset to GPIO peripheral

    timer_map = mmap(
        NULL,
        BLOCK_SIZE,
        PROT_READ|PROT_WRITE,
        MAP_SHARED,
        mem_fd,
        TIMER_BASE);

    spi_map = mmap(
        NULL,
        BLOCK_SIZE,
        PROT_READ|PROT_WRITE,
        MAP_SHARED,
        mem_fd,
        SPIO_BASE);

    if (reg_map == MAP_FAILED) {
        printf("gpio mmap error %d\n", (int)reg_map);
        close(mem_fd);
        exit(-1);
    }
    if(timer_map == MAP_FAILED) {
        printf("timer mmap error %d\n", (int)timer_map);
        close(mem_fd);
        exit(-1);
    }
    if(spi_map == MAP_FAILED) {
        printf("spi mmap error %d\n", (int)spi_map);
        close(mem_fd);
        exit(-1);
    }

    gpio = (volatile unsigned *)reg_map;
    sys_timer = (volatile unsigned *)timer_map;
     spi = (volatile unsigned *)spi_map;
}

//////////////////////////////////////////////////////////////////////
// SPI Functions
//////////////////////////////////////////////////////////////////////

#define CS_DELAY 1

//Pins
#define CS 6
```

```
#define MISO 27
#define MOSI 17
#define SCLK 22


void pulseClkSpi() {
    delayNanoseconds(300);
    digitalWrite(SCLK, 1);
    delayNanoseconds(300);
    digitalWrite(SCLK, 0);
}

void spiInit(int freq, int settings) {
    //set GPIO 6 (CE), 27 (MISO), 17 (MOSI), 22 (SCLK)
    pinMode(CS, OUTPUT);
    pinMode(MISO, OUTPUT);
    pinMode(MOSI, OUTPUT);
    pinMode(SCLK, OUTPUT);

    digitalWrite(CS, 1);
    delayMicroseconds(5);
}

char spiSendReceive(char send){
    char receive = 0;
    digitalWrite(CS, 0); // Get ready to send
    int i;
    for (i = 7; i >= 0; i--) {
      if (send & (1 << i)) {
        digitalWrite(MOSI, 1);
      }
      else {
       digitalWrite(MOSI, 0);
      }
      receive |= (digitalRead(MISO) << i);
      pulseClkSpi();
    }
   digitalWrite(CS, 1);
   return receive;


}
```

```
// led_matrix_background.h
// Alex Goldstein, Jane Wu
// agoldstein@hmc.edu, jhwu@hmc.edu, 6 December 2016

char MATRIX_BACKGROUND[4][32] = {
{
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000,
	0b00000000
},
{
	0b11110010,
	0b00001001,
	0b11111110,
	0b01001000,
	0b00110000,
	0b00000000,
	0b10101000,
	0b10101000,
	0b10101000,
	0b10101000,
```

```
        0b10101000,
        0b10101000,
        0b10101000,
        0b10101000,
        0b10101000,
        0b10101000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000
},
{
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
```

```
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000
},
{
        0b00000000,
        0b00000001,
        0b00111111,
        0b00010100,
        0b00001000,
        0b00000000,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00001010,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000,
        0b00000000
}
};
```

```
// bins_to_notes.h
// Alex Goldstein, Jane Wu
// agoldstein@hmc.edu, jhwu@hmc.edu, 6 December 2016

// Looking at bins from 240hz to 1104hz

int BINS_TO_NOTES[108] =
{
 0,0,0,0,0,                // c4: 240-280
 1,1,1,1,                  // d4: 280-312
 2,2,2,2,                  // e4: 312-344
 3,3,3,3,                  // f4: 344-376
 4,4,4,4,4,                // g4: 376-416
 5,5,5,5,5,5,              // a4: 416-464
 6,6,6,6,6,6,              // b4: 464-512
 7,7,7,7,7,                // c5: 512-552
 8,8,8,8,8,8,8,8,8,        // d5: 552-624
 9,9,9,9,9,9,9,                    // e5: 624-680
 10,10,10,10,10,10,10,10,  // f5: 680-744
 11,11,11,11,11,11,11,11,11,11,11,      // g5: 744-832
 12,12,12,12,12,12,12,12,12,12,12,12,   // a5: 832-928
 13,13,13,13,13,13,13,13,13,13,13,      // b5: 928-1016
 14,14,14,14,14,14,14,14,14,14,14 // c6: 1016-1104
};
```