Electronic Guitar Tuner (Or Guitar String Triggered Random Number Generator [with light show])

Final Project Report
December 8, 2017
E155

Sean Mahre and Owen Morrison

Abstract:

A key step before playing a musical instrument is ensuring that the instrument is in tune, and electric guitars are no exception. There are several ways to do this, including tuning by ear and tuning using a guitar tuner. We set out to design and build an electronic guitar tuner using a microcontroller, Analog-to-Digital Converter, FPGA, LCD character display, and row of 5 LEDs. The guitar signal is routed to the ADC through an amplifier and lowpass filter circuit, where it is sampled by the FPGA. The FPGA then computes an FFT on the sampled data, and sends the fundamental frequency of the sampled data to the microcontroller via SPI. The microcontroller then processes this frequency and makes an in-tune/out-of-tune designation, displaying the results on the LCD display and LED row. We were unsuccessful in implementing the FFT, and thus were not able to build a functional guitar tuner.

Introduction

For the final project, we set out to design and build an electronic guitar tuner. The goal for this project was to take the output signal of an electric guitar as input, and output the note being played on an LCD display and an indication on LEDs if the note was sharp, flat, or in tune. It was designed to work with notes spanning from B_1 (61.74 Hz) to B_4 (493.88 Hz). This range contains standard guitar tunings as well as most popular alternate guitar tunings. Several subsystems are required for the tuner, which can be divided into three parts: pre-processing via an amplifier and low-pass filter on the breadboard, obtaining the frequency content of the signal via an FFT on an FPGA, and post-processing/displaying information via a Raspberry Pi and LCD Character Display. We were not successful in creating the FFT, and thus were not able to complete our goals for this project.

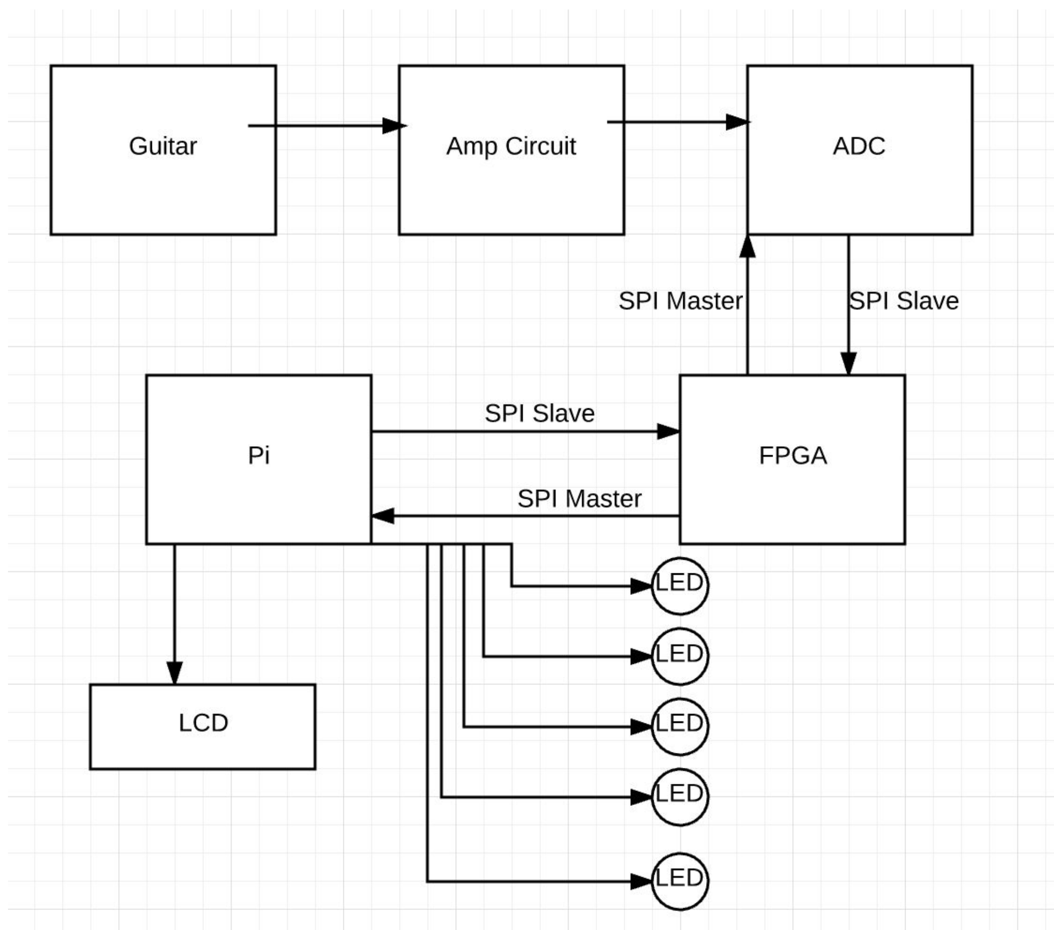

Figure 1: System Block Diagram

Breadboard Schematic

A schematic of the circuitry on our breadboard for this project is as follows:

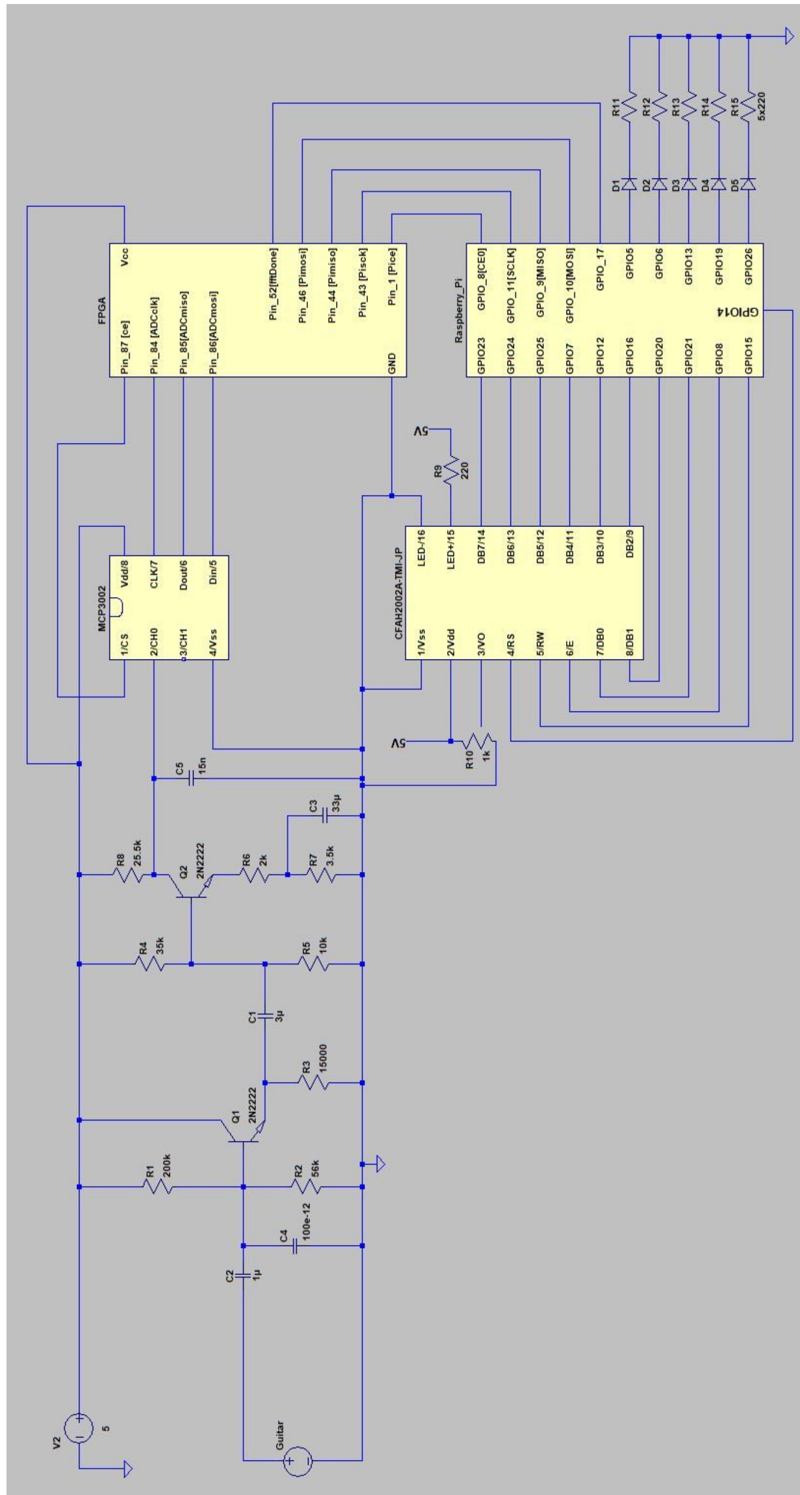

Figure 2: Breadboard Schematic

Amplifier and Lowpass Filter

Beginning with pre-processing, we first connected an electric guitar to an oscilloscope to investigate the waveform of the output signal. We noticed that when a note is played, a sinusoid at the frequency of the string is the dominant part of the signal, in that it has the highest amplitude. There is also additional content to the signal, which are harmonics of the frequency of the string, which are lesser in amplitude than the main frequency. The amplitudes of the harmonics seem to always be lower in magnitude than the main frequency of the string. The relative magnitudes are dependent on a variety of factors, including how hard the string was plucked, and what the "Tone" settings on the guitar is set to - if "Tone" is rolled to 10, there is more harmonic content of the signal than if it is rolled to 0. We also noted that the signal measured about 200 mV peak-to-peak - however, this value will vary depending on the type of pickups in the guitar.

With this analysis, we decided what pre-processing would need to be carried out. First, we wanted to try and filter out as much harmonic content as possible. This is done by implementing a low-pass filter. A challenge was that we could not filter out all harmonics - for example, the low E-string of a guitar is tuned to E2, or 82.41 Hz. A harmonic of this note would be E4, or 329.63 Hz, which happens to correspond to the high e-string of a guitar. So, to not filter out the highest frequency, we needed to set our corner at a frequency greater than 329.63 Hz, with room to accommodate the high e-string being too high when checking for tuning. We implemented a first order filter with a cutoff frequency at 400 Hz. This filters out higher harmonics and allowed us to set our sampling frequency at 1024 Hz. This decision was made with Nyquist analysis in mind - to avoid aliasing, we must sample at a frequency at least twice our bandwidth.

The second pre-processing step we investigated was amplifying the input signal. As we noted with the oscilloscope, the amplitude of the output signal is around 200 mV peak-to-peak. This was for a particularly hard pluck, and we wanted to be able to get an accurate reading for a full range of pluck strengths. For this reason, we decided to implement an amplifier with a gain of 10. This would amplify the signal to a higher voltage range, while not distorting the signal. We used a two-stage BJT amplifier with high input impedance so as to protect against varying output impedances of the guitar. The output is DC biased at 2.5 V, right in the middle of our supply provided to the amplifier and our ADC.

The output of this amplifier was routed to an ADC, which was directly connected to I/O pins on the FPGA.

FPGA Design

The FPGA design includes SPI communication modules with the Raspberry Pi and ADC, as well as a hardware implementation of an FFT.

The FPGA receives data from the ADC. This is done using an SPI master module on the FPGA. This data is sampled at a frequency of 1024 Hz.

The data received from the ADC is then passed through an FFT module implemented on the FPGA. The FFT design used in this design is that described in "The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation" by G. William Slade. The sampling rate used in this was 1024 hz and the number of samples was 1024.

The FFT contains four modules, the butterfly unit, the address generation unit, the two port ram modules, and the memory data blocks module. The butterfly unit takes in two 16 bit data samples. It then multiplies one of these data samples by a predetermined twiddle factor, then performs an addition and subtraction operation as indicated in Figure 3.The Address generation unit iterates through the 9 levels necessary for the FFT and the 1024 samples in order to generate the addresses needed for storing and retrieving data from the data memory blocks. The Data memory blocks creates 4 two port ram modules to store the current data used in the FFT. At each step in the FFT data in the memory blocks at the address specified by the AGU was passed into the butterfly unit. The processed data was then stored in the next generated address. The overall process is shown in Figure 4.

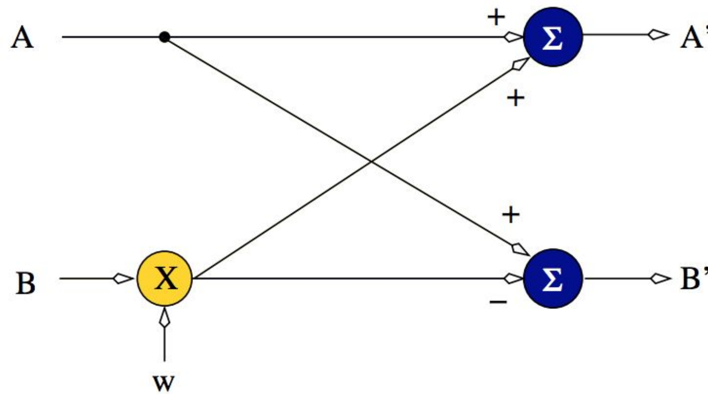

Figure 3: ButterFly Operation [1]

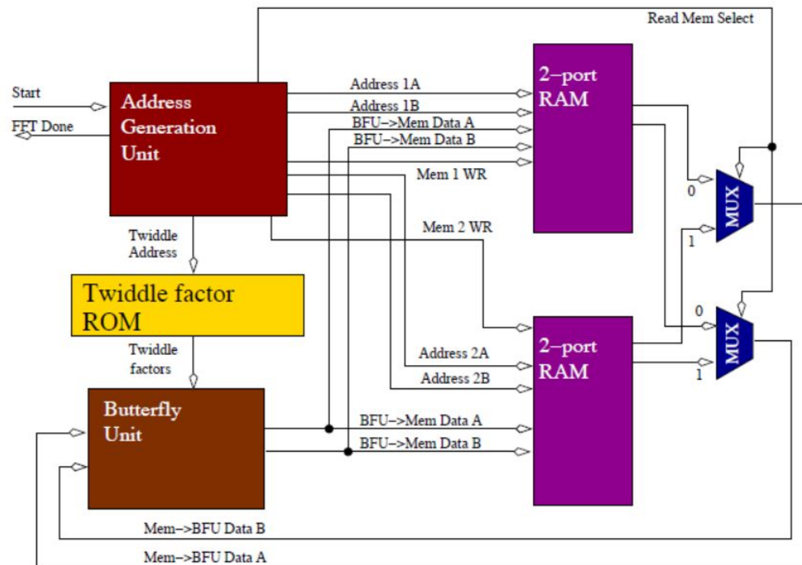

Figure 4: Full FFT block diagram [1]

During the 9th iteration of the FFT, after each sample was processed for the final time, the data was passed into the findMaxPeaks module. This module iterated through the data and determined the maximum frequency of the signal. This signal was the frequency signal passed into the raspberry Pi.

The overall guitarTuner module had a state machine that cycled through several states. Each state produced the necessary signals to operate each module. Upon completion of the FFT, the system outputs an fftDone signal. This signal indicates that it is time to run the SPI slave module to communicate with the raspberryPi.

The SPI slave module sends the max frequency signal to the raspberry Pi.

Raspberry Pi

The Pi is responsible for obtaining the frequency information from the FPGA, processing it into a note, and displaying the in-tune/out-of-tune designation on a row of LEDs. When the FPGA evoked the fftDone signal, the Pi initialized SPI communication to receive the maximum frequency from the FPGA. Because we used a 1024 point FFT and sampled at 1024 Hz, we expected 512 bins at a resolution of 1 Hz. Thus, no further processing of bin-to-frequency would need to be carried out by the Pi.

After receiving the frequency from the FPGA, the Pi checked the frequency against standard frequency values for defined notes to predict which note was being played by the guitar. Once this was determined, the note name and frequency were displayed on an LCD character display. In addition, the Pi checked the frequency against different ranges within the predicted note's frequency range. From this comparison, the Pi determined if the note was in tune, sharp, or flat, and displayed the results on a row of LEDs attached to GPIO pins. A green LED in the middle represented in tune, and two red LEDs on either side represented sharp and flat, with two different levels of out-of-tune.

LCD Character Display

The LCD Character Display was our new piece of hardware. We used a 20x2 Parallel Character Display (Crystalfontz part number CFAH2002A-TMI-JT) to display information from the Pi. The display was wired to GPIO pins on the Pi as explained in the display's datasheet [2], and a header file was created on the Pi to initialize the display as well as communicate information to it, again as explained in the datasheet.

Results

The design did not accomplish all of our goals. The current design was able to read in a signal from a guitar, pass the signal through the amp circuit and ADC, handle communications between the FPGA and the ADC, run the FFT, pass data into the raspberry PI, and display information on the LCD and LEDs. However, the frequency displayed on the LCDs was inaccurate and wildly variable, even with a constant sine wave input. This indicated that there was some error with the FFT. The FFT was not able to generate the correct amplitudes and frequencies.

The FFT was without a doubt the most difficult portion of this design. We severely underestimated how long the FFT would take to design, and did not allocate the appropriate amount of time to get it to a fully functional state. Were we to do this project again, we would start with this portion and ensure that it was correctly working earlier in the project. We would also have designed a more contained FFT module so that we could more easily pass correct inputs into a simulation of the FFT. This would have allowed us to more clearly see where we were encountering issues in the FFT process.

As it sits, our project is essentially a guitar string triggered random number generator. When no signals are being driven into the input of the system, the display reads a frequency of 0 Hz. As soon as a signal is driven into the input, the display begins cycling through numbers, in what appears to be a random fashion. These are still processed by the Pi, and the row of LEDs flashes according to the number generated, thus making a guitar string triggered random number generator with light show.

Parts List

Part	Source	Vendor Part #	Price
20x2 Parallel Character Display	Crystalfontz	CFAH2002A-TMI-JT	Retrieved from MicroPs lab
6.35 mm Phone Jack	Digikey	SC1089-ND	$2.26
16x2 Breakout Board	Proto Advantage	DR254D254P16F	$7.89

References

[1] “The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation,” G. William Slade, March 21, 2013, [Online]. Available: pages.hmc.edu/harris/class/e155/FFTtutorial121102.pdf. [Accessed December 8, 2017].

[2] “CFAH2002A-TMI-JP,” Crystallfontz America, Inc., [Datasheet, Online]. Available: https://www.crystalfontz.com/product/cfah2002atmijt-20x2-character-display-module. [Accessed December 8, 2017].

Appendix A: FPGA Code

```
// Owen Morrison and Sean Mahre
// December 8, 2017
// omorrison@g.hmc.edu and smahre@g.hmc.edu

// Main module for Guitar Tuner
module GuitarTuner2 #(parameter logN = 10)
                                        (input logic clk,
                                         input logic adcmiso,
              // MISO for ADC SPI
                                         input logic Pice, Pisck,Pimosi,            // CE,
CLK and MOSI for PI SPI
                                         output logic Pimiso,
              // MISO for PI SPI
                                         output logic adcmosi, adcClk, adcCe,    // MOSI,
CLK, and CE for ADC SPI
                                         output logic fftDone);
       // fft Done signal

    // Main FSM logic
    logic ADCreset,FSMreset,cntreset;
    logic sampleClk;
    logic ADCdone;
    logic [15:0]sendData,receivedData;
    logic [15:0] bottomTenFromADC;
    logic [logN-1:0] sampleCnt;
    logic [14:0] cnt;

    // Data mem block logic
    logic loadDataWrite,Bank0WrEN,Bank1WrEN,BankReadSelect;
    logic [15:0]Datar,Datai,Xr,Xi,Yr,Yi;
    logic [logN-1:0]loadDataAddr,readAddrG,writeAddrG,readAddrH,writeAddrH;
    logic [15:0]Gr,Gi,Hr,Hi;

    // FFT logic
    logic fftStart;

    // AGU logic
    logic [logN-2:0]tw;
    logic memWrite;

    // Peak find logic
    logic peakEN,peakReset,peakFound;
    logic [3:0]i; // assume logN < 15
    logic [logN-1:0]maxFreqAddr;

    // SPI slave logic
    logic [15:0]Pid,Piq;
```

```
assign sendData = 16'h6800; // Send to ADC for 10-bit data, MSB first
assign bottomTenFromADC = receivedData & 16'h03FF; // Get bottom 10 bits from ADC

typedef enum logic [2:0] {START,SPIMASTER,DELAY,SPIDONE,FFT,FINDPEAK,SPISLAVE} statetype;
statetype state,nextstate;


always_ff@(posedge clk or posedge FSMreset or posedge cntreset)
    if(cntreset | FSMreset) cnt <= 0;
    else cnt <= cnt + 15'b1;

always_ff@(posedge clk)
    state <= nextstate;

// Generate sampling clk, currently ~ 1024 Hz
always_ff@(posedge clk)
    if(cnt < 9750) // change depending on logN
            begin
                    cntreset <= 0;
                    sampleClk <= 1;
            end
    else if(cnt < 19500) // change depending on logN
            begin
                    cntreset <= 0;
                    sampleClk <= 0;
            end
    else
            begin
                    cntreset <= 1;
                    sampleClk <= 1;
            end

// Counter for number of samples
always_ff@(posedge sampleClk or posedge FSMreset)
    if(FSMreset) sampleCnt <= 0;
    else sampleCnt <= sampleCnt + 10'b1;

// Nextstate logic
always_comb
    case(state)
            START:          nextstate <= SPIMASTER;
            SPIMASTER:      if(ADCdone) nextstate <= DELAY;
                                    else nextstate <= SPIMASTER;
            DELAY:          if(cntreset) nextstate <= SPIMASTER;
                                    else if(sampleCnt == 1023) nextstate <= SPIDONE;
                                    else nextstate <= DELAY;
            SPIDONE:      nextstate <= FFT;
            FFT:                    if(fftDone) nextstate <= FINDPEAK;
                                    else nextstate <= FFT;
            FINDPEAK:       if(peakFound) nextstate <= SPISLAVE;
```

```
                                        else nextstate <= FINDPEAK;
            SPISLAVE:   nextstate <= START;
            default:     nextstate <= START;
        endcase

    // Output logic
    assign ADCreset = (state == START | state == DELAY);
    assign FSMreset = (state == START);
    assign fftStart = (state == FFT);
    assign peakEN = (state == FINDPEAK);
    assign peakReset = (state == SPIMASTER);

    assign loadDataWrite = (state == SPIMASTER);
    assign Datar = bottomTenFromADC;
    assign Datai = bottomTenFromADC & 16'h0000;

    logic reseti;
    assign reseti = (state == SPIDONE);

    // Load data into RAM in bit reversed order
    genvar k;
    generate
        for(k = 0; k < logN; k = k+1)
            begin: bitreverse
                assign loadDataAddr[k] = sampleCnt[logN-1-k]; // Bit reverse
            end
    endgenerate

    spi_master ADC(clk,sendData,adcmiso,ADCreset,adcCe,adcmosi,receivedData,adcClk,ADCdone);
    dataMemory #(logN)
mem(clk,loadDataWrite,Bank0WrEN,Bank1WrEN,Datar,Datai,BankReadSelect,loadDataAddr,
                                      readAddrG,writeAddrG,readAddrH,writeAddrH,
                                      Xr,Xi,Yr,Yi,Gr,Gi,Hr,Hi);
    AGU #(logN) agu(clk,fftStart,reseti,fftDone,readAddrG,readAddrH,tw,memWrite, i,
BankReadSelect);

    always_ff @(posedge clk)
        begin
            writeAddrG <= readAddrG;
            writeAddrH <= readAddrH;
        end


    assign Bank0WrEN = memWrite & BankReadSelect;
    assign Bank1WrEN = memWrite & ~BankReadSelect;

    butterfly #(logN) bfy(clk,Gr,Gi,Hr,Hi,tw,Xr,Yr,Xi,Yi);
    findMaxPeak #(logN) maxPeak(clk,peakEN, peakReset, Gr, Gi, readAddrG, i, maxFreqAddr,
peakFound);
```

```
    assign Pid = maxFreqAddr;

    spi_slave pi(Pisck,Pimosi,Pimiso,Pice,Pid,Piq);

endmodule

// SPI Master module
module spi_master(input logic clk,
                                     input logic [15:0]d, //data to send
                                    input logic miso,
                                     input logic reset,
                                    output logic ce,
                                    output logic mosi,
                                    output logic [15:0]q, // Data received
                                     output logic adcClk,
                                     output logic done);

    logic [4:0]cnt;
    logic [15:0]data,qtemp;
    logic [3:0] clkCnt;

    typedef enum logic [2:0] {IDLE,DATA,COMPLETE}
                                                                   statetype;

    statetype state,nextstate;

    always_ff@(posedge clk or posedge reset)
        if(reset) clkCnt <= 0;
        else clkCnt <= clkCnt + 4'b1;
    assign adcClk = clkCnt[3]; // 2.5 MHz clk

    always_ff@(posedge adcClk or posedge reset)
        if(reset) begin
                                     state <= IDLE;
                                     cnt <= 0;
                               end
        else     begin
                              state <= nextstate;
                              cnt <= cnt + 5'b1;
                      end

    always_comb
        case(state)
               IDLE: nextstate <= DATA;
               DATA: if(cnt == 16) nextstate <= COMPLETE;
                              else nextstate <= DATA;
               COMPLETE: nextstate <= COMPLETE;
               default: nextstate <= COMPLETE;
        endcase
```

```
    assign ce = (state == IDLE | state == COMPLETE);
    assign done = (state == COMPLETE);
    assign mosi = data[15];

    always_ff@(posedge adcClk)
        if(state == IDLE) data <= d;
        else if(state == DATA) data <= {data[14:0],miso};
        else data <= data;

    always_comb
        case(state)
                IDLE:
                        begin
                                q <= qtemp;
                                qtemp <= 0;
                        end
                DATA:
                        begin
                                q <= 0;
                                qtemp <= 0;
                        end
                COMPLETE:
                        begin
                                q <= data;
                                qtemp <= data;
                        end
                default:
                        begin
                                q <= 0;
                                qtemp <= 0;
                        end
        endcase

endmodule

// Datamemory block
module dataMemory #(parameter logN = 5)
                                        (input logic clk,
                                         input logic loadDataWrite,
                                         input logic Bank0WrEN,
                                         input logic Bank1WrEN,
                                         input logic [15:0]Datar,
                                         input logic [15:0]Datai,
                                         input logic BankReadSelect,
                                         input logic [logN-1:0]loadDataAddr,
                                         input logic [logN-1:0]readAddrG,
                                         input logic [logN-1:0]writeAddrG,
                                         input logic [logN-1:0]readAddrH,
                                         input logic [logN-1:0]writeAddrH,
                                         input logic [15:0]Xr,
```

```
                                    input logic [15:0]Xi,
                                    input logic [15:0]Yr,
                                    input logic [15:0]Yi,
                                    output logic [15:0]Gr, Gi, Hr, Hi);
        logic [logN-1:0] addrA0, addrA1, addrB0, addrB1;
        logic [15:0] DataA0_r, DataA0_i;
        logic [15:0] preG0r, preG0i, preH0r, preH0i, preG1r, preG1i, preH1r, preH1i;
        logic Bank0_A_WR, Bank0_B_WR;
        logic [15:0] DataB_r, DataB_i, DataA1_r, DataA1_i;
        logic Bank1WrEnDelay;

        assign Bank0_A_WR = Bank0WrEN | loadDataWrite;
        assign DataA0_r = loadDataWrite ? Datar : Xr;
        assign addrA0 = loadDataWrite ? loadDataAddr : (Bank0WrEN ? writeAddrG : readAddrG);
        assign addrB0 = Bank0WrEN ? writeAddrH : readAddrH;
        assign DataA0_i = loadDataWrite ? Datai : Xi;
        assign addrA1 = loadDataWrite ? loadDataAddr : (Bank1WrEN ? writeAddrG : readAddrG);
        assign addrB1 = Bank1WrEN ? writeAddrH : readAddrH;

        always_ff@(posedge clk) // For delays
               begin
                      Bank0_B_WR <= Bank0WrEN;
                      DataB_r <= Yr;
                      DataB_i <= Yi;
                      DataA1_r <= Xr;
                      DataA1_i <= Xi;
                      Bank1WrEnDelay <= Bank1WrEN;
               end

        // Instantiate RAM
        twoportram #(logN)
real0(clk,DataA0_r,Yr,addrA0,addrB0,Bank0_A_WR,Bank0WrEN,preG0r,preH0r);
        twoportram #(logN)
im0(clk,DataA0_i,Yi,addrA0,addrB0,Bank0_A_WR,Bank0WrEN,preG0i,preH0i);
        twoportram #(logN) real1(clk,Xr,Yr,addrA1,addrB1,Bank1WrEN,Bank1WrEN,preG1r,preH1r);
        twoportram #(logN) im1(clk,Xi,Yi,addrA1,addrB1,Bank1WrEN,Bank1WrEN,preG1i,preH1i);

        assign Gr = BankReadSelect ? preG1r : preG0r;
        assign Gi = BankReadSelect ? preG1i : preG0i;
        assign Hr = BankReadSelect ? preH1r : preH0r;
        assign Hi = BankReadSelect ? preH1i : preH0i;

endmodule

// Two port RAM module
module twoportram #(parameter logN = 5)
                               (input logic clk,
                                    input logic [15:0] dataA, dataB,
                                    input logic [logN-1:0] addrA, addrB,
                                    input logic writeEnA, writeEnB,
```

```
                                output logic [15:0]qA, qB);
    logic [15:0] mem[1023:0];

    always_ff @(posedge clk)
        begin
            if (writeEnA)
                begin
                    mem[addrA] <= dataA;
                    qA <= dataA;
                end
            else
                qA <= mem[addrA];
        end

    always_ff @(posedge clk)
        begin
            if (writeEnB)
                begin
                    mem[addrB] <= dataB;
                    qB <= dataB;
                end
            else
                qB <= mem[addrB];
        end
endmodule

// Address Generation Unit
module AGU #(parameter logN = 5)
                (input logic clk,
                 input logic start,
                 input logic reseti,
                 output logic done,
                 output logic [logN-1:0] addrA,
                 output logic [logN-1:0] addrB,
                 output logic [logN-2:0] tw,      //twiddle factor
                 output logic memWrite,
                 output logic [3:0]i,
                 output logic BankReadSelect);

    logic prevStart;
    logic reset;
    logic [logN-2:0]j;
    logic [logN-2:0]twAddr;
    logic [logN-1:0]ja, jb;

    always_ff @(posedge clk)
        prevStart <= start;

    assign reset = ~prevStart & start;      // fft was not running, but now has to run again
    always_ff @(posedge clk, posedge reset)
```

```
        begin
               if (reset)
                      begin
                             j <= 0;
                      end
               else if (j < 511)   // assume butterfly index of 16
                      j <= j + 1;
               else
                      j <= 0;
        end

    always_ff @(posedge clk, posedge reset, posedge reseti)
        begin
               if (reseti | reset)
                      begin
                             i <= 0;
                             done <= 0;
                      end
               else if (j==511) //2^logN
                      begin
                             if (i < 9)
                                    begin
                                           i <= i + 4'b1;
                                    end
                             else
                                    begin
                                           i <= 0;
                                           done <= 1;
                                    end
                      end
        end
    assign ja = j << 1;
    assign jb = ja + 1;

    assign addrA = ((ja << i) | (ja >> (logN - i)));
    assign addrB = ((jb << i) | (jb >> (logN - i)));

    assign twAddr = (((32'hFFFFFFF0 >> i) & 4'hF) & j); //twiddle addresses

    always_ff @(posedge clk)
        tw <= twAddr;

    assign memWrite = ~done & ~(j==0);
    assign BankReadSelect = i[0];
endmodule

// Butterfly unit
module butterfly #(parameter logN = 5)
                                     ( input logic  clk,
                                      input logic [15:0]Gr, Gi, Hr, Hi,
```

```
                                        input logic [logN-2:0] twAddr,
                                        output logic [15:0]Xr,
                                        output logic [15:0]Yr,
                                        output logic [15:0]Xi,
                                        output logic [15:0]Yi);

    logic [15:0]delayedGr;
    logic [15:0]holdGr;
    logic signed [31:0]twiddledHr;
    logic [15:0]delayedGi;
    logic [15:0]holdGi;
    logic [31:0]twiddledHi;
    logic [15:0]tw_r[511:0];
    logic [15:0]tw_i[511:0];
    logic [15:0] tempTwr, tempTwi;
    logic signed [15:0]signedGr, signedGi,  signedHr, signedHi;

    // Twiddle factors
    initial
        begin
              // Twiddle factors would go here. Removed for clarity of code
        end


    assign signedGr = Gr;
    assign signedGi = Gi;
    assign signedHr = Hr;
    assign signedHi = Hi;

    assign twiddledHr = signedHr*tw_r[twAddr];
    assign twiddledHi = signedHi*tw_i[twAddr];

    assign Xr = signedGr + twiddledHr[30:15];
    assign Yr = signedGr - twiddledHr[30:15];
    assign Xi = signedGi + twiddledHi[30:15];
    assign Yi = signedGi - twiddledHi[30:15];

endmodule

// Find max bin in RAM
module findMaxPeak #(parameter logN = 5)
                                       (input logic clk,
                                        input logic en,
                                        input logic reset,
                                        input logic [15:0]Gr, Gi,
                                        input logic [logN-1:0] addrA,
                                        input logic [3:0]i,
                                        output logic [logN-1:0] maxFreqAddr,
                                        output logic peakFound);
    logic  [15:0] maxAmp;
```

```
    logic [logN-1:0] maxBin, tempBin;
    logic  [15:0] magG, tempMax;
    logic signed[15:0]signedGr,signedGi;
    logic signed[31:0]signedMagG;

    assign signedGr = Gr;
    assign signedGi = Gi;
    assign signedMagG = signedGr*signedGr + signedGi*signedGi;
//Gr*Gr + Gi*Gi;
    assign magG = signedMagG[30:15];

    always_ff@(posedge clk)
        if(i==logN-1)
                begin
                                if (magG > tempMax)
                                        begin
                                                tempMax <= magG;
                                                tempBin <= addrA;
                                        end
                                else
                                        begin
                                                tempBin <= tempBin;
                                                tempMax <= tempMax;
                                        end
                end
        else
                begin
                        tempMax <= 0;
                        tempBin <= 0;
                end

    always_ff@(posedge clk)
        if (reset)
                begin
                        maxAmp <= 0;
                        maxBin <= 0;
                end
        else
                begin
                        if (tempMax > maxAmp)
                                begin
                                        maxAmp <= tempMax;
                                        maxBin <= tempBin;
                                end
                        else
                                begin
                                        maxAmp <= maxAmp;
                                        maxBin <= maxBin;
                                end
                end
```

```
    always_ff@(posedge clk)
        if(en)
                begin
                        maxFreqAddr <= maxBin;
                        peakFound <= 1;
                end
        else
                begin
                        maxFreqAddr <= maxFreqAddr;
                        peakFound <= 0;
                end

endmodule

// SPI slave module
module spi_slave(input logic clk,
                                input logic mosi,                       //to slave
                                output logic miso,              //to master
                                input logic reset,
                                input logic [15:0]d,            //data to send
                                output logic [15:0]q);          //data to recieve

        logic [3:0] cnt;
        logic qdelayed;
        always_ff@(negedge clk)
                if(reset)cnt = 0;
                else cnt = cnt + 4'b1;

        always_ff@(posedge clk)
                q <= (cnt == 0) ? {d[14:0], mosi} : {q[14:0], mosi};
        always_ff@(negedge clk)
                qdelayed = q[15];
        assign miso = (cnt == 0)? d[15]: qdelayed;
endmodule
```

Appendix B: C code on Pi

```c
// Owen Morrison and Sean Mahre
// December 8, 2017
// omorrison@g.hmc.edu and smahre@g.hmc.edu
// This .c file reads the frequency from an FPGA using SPI and
// determines the note being played, compares it to standard
// frequencies, and displays the information on an LCD character
// display and row of 5 LEDs

#include "lcdControl.h"
#include <string.h>
#include <math.h>

// no_of_digits and i_to_a retreived from
//https://www.careercup.com/question?id=2794
int no_of_digits(int num)
{
	int digit_count = 0;
	while(num > 0)
	{
	digit_count++;
	num /= 10;
	}
	return digit_count;
}

char * i_to_a(int num)
{
	char *str;
	int digit_count = 0;
	if(num < 0)
	{
	num = -1*num;
	digit_count++;
	}
	digit_count += no_of_digits(num);
	str = (char *)malloc(sizeof(char)*(digit_count+1));
	str[digit_count] = '\0';
	while(num > 0)
	{
	str[digit_count-1] = num%10 + '0';
	num = num/10;
	digit_count--;
	}
	if(digit_count == 1)
	str[0] = '-';
	return str;
}
```

```
// createFreqMessage creates the frequency string to be displayed by an LCD character display
char* createFreqMessage(int freq){
    char *freqString = i_to_a(freq); // Convert int frequency to string
    char *text = "Frequency: ";
    char *freqDisplay = malloc(strlen(freqString)+strlen(text)+1); // Allocate enough memory
to store new string

    // Only update if frequency is within range
    if(freq>61.74 && freq<508.56){
        strcpy(freqDisplay,text);
        strcat(freqDisplay,freqString);
    }
    else{
        freqDisplay = text;
    }
    return(freqDisplay);
}

// Display tune, sharp, or flat distinction on LEDs connected to GPIO pins
void displayLED(int freq, float inTuneFreq, float hiLimit, float lowLimit){
    // Clear LEDs
    digitalWrite(26,0);
    digitalWrite(19,0);
    digitalWrite(13,0);
    digitalWrite(6,0);
    digitalWrite(5,0);
    float intuneMD = 1.00289; // Based of */ 5 cents
    float totalRange = hiLimit - lowLimit;
    if(inTuneFreq == 0){}
    else if(freq>inTuneFreq/intuneMD && freq<inTuneFreq*intuneMD){
        digitalWrite(13,1);
       }
       else if(freq<inTuneFreq && freq>inTuneFreq-totalRange/4.0){
              digitalWrite(19,1);
       }
    else if(freq<inTuneFreq){
        digitalWrite(26,1);
    }
    else if(freq>inTuneFreq && freq<inTuneFreq+totalRange/4.0){
        digitalWrite(6,1);
    }
    else{
        digitalWrite(5,1);
    }
}

/* determineNote does several key things for the guitar tuner. Based off of an
   input frequency, it predicts what note was being played to the nearest half
   step. It then compares the frequency to a range of acceptable frequencies
   for each note, and writes the corresponding pin high to indicate the tune
```

```
   status.
*/
char* determineNote(int freq){
    // Frequencies of notes spanning B1 -> B4
    float B1 = 61.74;
    float C2 = 65.41;
    float Db2 = 69.30;
    float D2 = 73.42;
    float Eb2 = 77.78;
    float E2 = 82.41;
    float F2 = 87.31;
    float Gb2 = 92.50;
    float G2 = 98.00;
    float Ab2 = 103.83;
    float A2 = 110.00;
    float Bb2 = 116.54;
    float B2 = 123.47;
    float C3 = 130.81;
    float Db3 = 138.59;
    float D3 = 146.83;
    float Eb3 = 155.56;
    float E3 = 164.81;
    float F3 = 174.61;
    float Gb3 = 185.00;
    float G3 = 196.00;
    float Ab3 = 207.65;
    float A3 = 220.00;
    float Bb3 = 233.08;
    float B3 = 246.94;
    float C4 = 261.63;
    float Db4 = 277.18;
    float D4 = 293.66;
    float Eb4 = 311.13;
    float E4 = 329.63;
    float F4 = 349.23;
    float Gb4 = 369.99;
    float G4 = 392.00;
    float Ab4 = 415.30;
    float A4 = 440.00;
    float Bb4 = 466.16;
    float B4 = 493.88;

    // Cutoff frequencies, where each is an upper limit
    float C2cutoff = (Db2+C2)/2.0;
    float Db2cutoff = (D2+Db2)/2.0;
    float D2cutoff = (Eb2+D2)/2.0;
    float Eb2cutoff = (E2+Eb2)/2.0;
    float E2cutoff = (F2+E2)/2.0;
    float F2cutoff = (Gb2+F2)/2.0;
    float Gb2cutoff = (G2+Gb2)/2.0;
```

```
float G2cutoff = (Ab2+G2)/2.0;
float Ab2cutoff = (A2+Ab2)/2.0;
float A2cutoff = (Bb2+A2)/2.0;
float Bb2cutoff = (B2+Bb2)/2.0;
float B2cutoff = (C3+B2)/2.0;
float C3cutoff = 2.0*C2cutoff;
float C4cutoff = 2.0*C3cutoff;
float Db3cutoff = 2.0*Db2cutoff;
float Db4cutoff = 2.0*Db3cutoff;
float D3cutoff = 2.0*D2cutoff;
float D4cutoff = 2.0*D3cutoff;
float Eb3cutoff = 2.0*Eb2cutoff;
float Eb4cutoff = 2.0*Eb3cutoff;
float E3cutoff = 2.0*E2cutoff;
float E4cutoff = 2.0*E3cutoff;
float F3cutoff = 2.0*F2cutoff;
float F4cutoff = 2.0*F3cutoff;
float Gb3cutoff = 2.0*Gb2cutoff;
float Gb4cutoff = 2.0*Gb3cutoff;
float G3cutoff = 2.0*G2cutoff;
float G4cutoff = 2.0*G3cutoff;
float Ab3cutoff = 2.0*Ab2cutoff;
float Ab4cutoff = 2.0*Ab3cutoff;
float A3cutoff = 2.0*A2cutoff;
float A4cutoff = 2.0*A3cutoff;
float Bb3cutoff = 2.0*Bb2cutoff;
float Bb4cutoff = 2.0*Bb3cutoff;
float B3cutoff = 2.0*B2cutoff;
float B4cutoff = 2.0*B3cutoff;

char* note;

// Predict note based off of frequency
if(freq>B1 && freq<=C2cutoff){
    note = "C2";
    displayLED(freq,C2,C2cutoff,B1);
}
else if(freq>C2cutoff && freq<=Db2cutoff){
          note = "C#2/Db2";
          displayLED(freq,Db2,Db2cutoff,C2cutoff);
}
else if(freq>Db2cutoff && freq<=D2cutoff){
          note = "D2";
          displayLED(freq,D2,D2cutoff,Db2cutoff);
}
else if(freq>D2cutoff && freq<=Eb2cutoff){
    note="D#2/Eb2";
          displayLED(freq,Eb2,Eb2cutoff,D2cutoff);
   }
else if(freq>Eb2cutoff && freq<=E2cutoff){
```

```
    note="E2";
          displayLED(freq,E2,E2cutoff,Eb2cutoff);
}
else if(freq>E2cutoff && freq<=F2cutoff){
    note = "F2";
    displayLED(freq,F2,F2cutoff,E2cutoff);
}
else if(freq>F2cutoff && freq<=Gb2cutoff){
    note="F#2/Gb2";
    displayLED(freq,Gb2,Gb2cutoff,F2cutoff);
}
else if(freq>Gb2cutoff && freq<=G2cutoff){
    note="G2";
    displayLED(freq,G2,G2cutoff,Gb2cutoff);
}
else if(freq>G2cutoff && freq<=Ab2cutoff){
    note="G#2/Ab2";
    displayLED(freq,Ab2,Ab2cutoff,G2cutoff);
}
else if(freq>Ab2cutoff && freq<=A2cutoff){
    note="A2";
    displayLED(freq,A2,A2cutoff,Ab2cutoff);
}
else if(freq>A2cutoff && freq<=Bb2cutoff){
    note="A#2/Bb2";
    displayLED(freq,Bb2,Bb2cutoff,A2cutoff);
}
else if(freq>Bb2cutoff && freq<=B2cutoff){
    note="B2";
    displayLED(freq,B2,B2cutoff,Bb2cutoff);
}
else if(freq>B2cutoff && freq<=C3cutoff){
    note="C3";
    displayLED(freq,C3,C3cutoff,B2cutoff);
}
else if(freq>C3cutoff && freq<=Db3cutoff){
    note="C#3/Db3";
    displayLED(freq,Db3,Db3cutoff,C3cutoff);
}
   else if(freq>Db3cutoff && freq<=D3cutoff){
    note="D3";
    displayLED(freq,D3,D3cutoff,Db3cutoff);
}
   else if(freq>D3cutoff && freq<=Eb3cutoff){
     note="D#3/Eb3";
    displayLED(freq,Eb3,Eb2cutoff,D3cutoff);
}
   else if(freq>Eb3cutoff && freq<=E3cutoff){
    note="E3";
    displayLED(freq,E3,E3cutoff,Eb3cutoff);
```

```
}
   else if(freq>E3cutoff && freq<=F3cutoff){
    note="F3";
    displayLED(freq,F3,F3cutoff,E3cutoff);
}
   else if(freq>F3cutoff && freq<=Gb3cutoff){
    note="F#3/Gb3";
    displayLED(freq,Gb3,Gb3cutoff,F3cutoff);
}
   else if(freq>Gb3cutoff && freq<=G3cutoff){
    note="G3";
    displayLED(freq,G3,G3cutoff,Gb3cutoff);
}
   else if(freq>G3cutoff && freq<=Ab3cutoff){
    note="G#3/Ab3";
    displayLED(freq,Ab3,Ab3cutoff,G3cutoff);
}
   else if(freq>Ab3cutoff && freq<=A3cutoff){
    note="A3";
    displayLED(freq,A3,A3cutoff,Ab3cutoff);
}
   else if(freq>A3cutoff && freq<=Bb3cutoff){
    note="A#3/Bb3";
    displayLED(freq,Bb3,Bb3cutoff,A3cutoff);
}
   else if(freq>Bb3cutoff && freq<=B3cutoff){
    note="B3";
    displayLED(freq,B3,B3cutoff,Bb3cutoff);
}
   else if(freq>B3cutoff && freq<=C4cutoff){
    note="C4";
    displayLED(freq,C4,C4cutoff,B3cutoff);
}
   else if(freq>C4cutoff && freq<=Db4cutoff){
    note="C#4/Db4";
    displayLED(freq,Db4,Db4cutoff,C4cutoff);
}
   else if(freq>Db4cutoff && freq<=D4cutoff){
    note="D4";
    displayLED(freq,D4,D4cutoff,Db4cutoff);
}
   else if(freq>D4cutoff && freq<=Eb4cutoff){
    note="D#4/Eb4";
    displayLED(freq,Eb4,Eb4cutoff,D4cutoff);
}
   else if(freq>Eb4cutoff && freq<=E4cutoff){
    note="E4";
    displayLED(freq,E4,E4cutoff,Eb4cutoff);
}
   else if(freq>E4cutoff && freq<=F4cutoff){
```

```
        note="F4";
        displayLED(freq,F4,F4cutoff,E4cutoff);
    }
       else if(freq>F4cutoff && freq<=Gb4cutoff){
        note="F#4/Gb4";
        displayLED(freq,Gb4,Gb4cutoff,F4cutoff);
    }
       else if(freq>Gb4cutoff && freq<=G4cutoff){
        note="G4";
        displayLED(freq,G4,G4cutoff,Gb4cutoff);
    }
       else if(freq>G4cutoff && freq<=Ab4cutoff){
        note="G#4/Ab4";
        displayLED(freq,Ab4,Ab4cutoff,G4cutoff);
    }
       else if(freq>Ab4cutoff && freq<=A4cutoff){
        note="A4";
        displayLED(freq,A4,A4cutoff,Ab4cutoff);
    }
       else if(freq>A4cutoff && freq<=Bb4cutoff){
        note="A#4/Bb4";
        displayLED(freq,Bb4,Bb4cutoff,A4cutoff);
    }
       else if(freq>Bb4cutoff && freq<=B4cutoff){
        note="B4";
        displayLED(freq,B4,B4cutoff,Bb4cutoff);
    }
    else{ // Frequency is too high
        note="Out of range";
        displayLED(freq,0,0,0);
    }
    return(note);
}

// Creates the note message to be deisplayed by an LCD character display
char* createNoteMessage(int freq){
    char *note = determineNote(freq);
    char *text = "Note: ";
    char *noteDisplay = malloc(strlen(note)+strlen(text)+1);
    strcpy(noteDisplay,text);
       strcat(noteDisplay,note);
       return(noteDisplay);
}

// Display messages on an LCD character display
void displayMessages(char *freqDisplay, char *noteDisplay){
    lcdWrite(0x01,INSTR); // Clear display
    delayMicros(1530);
    lcdPrintString(freqDisplay);
    lcdWrite(0xC0,INSTR);
```

```
    lcdBusyWait();
    lcdPrintString(noteDisplay);
}

void main(void){
    int received; // Int to store data received via SPI
    int receivedOld=0; // Store previous received
    pioInit(); // Initialize GPIO
    spiInit(2000000,0); // Initialize SPI, 2 MHz clk, default settings
    pinMode(22,INPUT); // Used to read when FPGA is ready to send
    pinMode(27,OUTPUT); // Used to tell FPGA that SPI read is complete
    int LEDPins[] = {26,19,13,6,5};
    pinModes(LEDPins,5,OUTPUT);
    while(true){
        if(digitalRead(22)){ // Only update on SPI read
               receivedOld = received;
               received = spiSendReceive16(0); // Receive data, send doesn't matter
               digitalWrite(27,1); // Data transfer complete
               delayMicros(10); // Ensure FPGA receives signal
               digitalWrite(27,0);
        }
        else{
               receivedOld = received;
        }
        if(received != receivedOld){ // Update display only if a new frequency has been
determined
               char *freqDisplay = createFreqMessage(received);
               char *noteDisplay = createNoteMessage(received);
               displayMessages(freqDisplay,noteDisplay);
               delayMicros(1000000); // Wait at least one sec for next note
        }
    }
}
```

Appendix C: LCD Control Header File

```
// Owen Morrison and Sean Mahre
// December 8, 2017
// omorrison@g.hmc.edu and smahre@g.hmc.edu
// This header file is used to control an LCD character display using
// Raspberry Pi GPIO pins

#include "EasyPIO.h"
int LCDPins[] = {21,7,14,25,15,24,18,23}; // Pins to send info to LCD character display
int numPins = 8;
typedef enum {INSTR,DATA} mode;
#define RS 16 // Mode pin
#define RW 20 // Read/Write pin
#define E  12 // Enable pin
int init = 0;

// Write to the LCD
void lcdWrite(char val, mode md){
    pinModes(LCDPins,numPins,OUTPUT);
    digitalWrite(RS,(md==DATA));
    digitalWrite(RW,0);
    digitalWrites(LCDPins,numPins,val);
    digitalWrite(E,1);
    delayMicros(100);
    digitalWrite(E,0);
    delayMicros(100);
}

// Read from the LCD
char lcdRead(mode md){
    pinModes(LCDPins,8,INPUT);
    digitalWrite(RS,(md==DATA));
    digitalWrite(RW,1);
    digitalWrite(E,1);
    delayMicros(10);
    char c = digitalReads(LCDPins,numPins);
    digitalWrite(E,0);
    delayMicros(10);
    return c;
}

// Wait until busy flag has been cleared
void lcdBusyWait(void){
    while(lcdRead(INSTR) >> 7);
}

// Initialize LCD
void lcdInit(void){
    pioInit();
```

```
    pinMode(RS,OUTPUT);
    pinMode(RW,OUTPUT);
    pinMode(E,OUTPUT);
    delayMicros(15000);
    lcdWrite(0x30,INSTR);
    delayMicros(4100);
    lcdWrite(0x30,INSTR);
    delayMicros(100);
    lcdWrite(0x30,INSTR);
    lcdBusyWait();
    lcdWrite(0x3C,INSTR); // 2-line, 5x8 font
    lcdBusyWait();
    lcdWrite(0x08,INSTR); // Display off
    lcdBusyWait();
    lcdWrite(0x01,INSTR); // Clear Display
    delayMicros(1530);
    lcdWrite(0x06,INSTR); // Advance cursor
    lcdBusyWait();
    lcdWrite(0x0C,INSTR); // Display on
    lcdBusyWait();
    init = 1;
}

// Print a string on the LCD
void lcdPrintString(char *str){
    if(!init){ // Initialize LCD if it hasn't already been initialized
        lcdInit();
    }
    while(*str != 0){
        lcdWrite(*str,DATA);
        str++;
    }
}
```