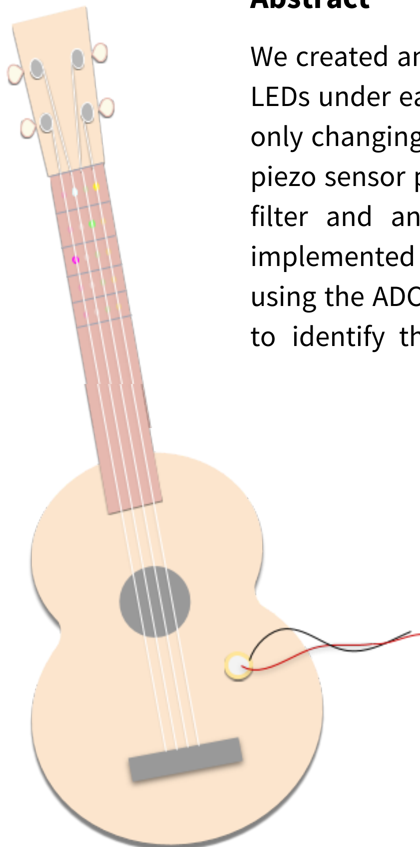
The Ukucorn

A Ukulele Teacher with
Chord Recognition & Interactive LEDs

Lauren Hu & Sarah Wang
E155 - Fall 2017



Abstract

We created an interactive fretboard to facilitate learning the ukulele. Five rows of LEDs under each string were embedded in the fretboard to display a target chord, only changing after the correct chord has been played. The analog signal from a piezo sensor placed on the ukulele body is passed to the FPGA through a low pass filter and analog-to-digital converter (ADC). An FFT (fast fourier transform) implemented in hardware on the FPGA identifies the magnitude of frequencies using the ADC output. Software on the Raspberry Pi uses this frequency analysis to identify the chord played and compare it to the target. Upon successful execution of the target chord, the LEDs on the fretboard change to display the next target chord in a hard-coded song.

Introduction

Learning a new instrument can be intimidating, expensive, and time-consuming with a human teacher. However, self-taught musicians also struggle to learn proper techniques and give up after too many unsuccessful attempts to read sheet music and play at the same time. We sought to replace frustration with a fun, digital training tool that sits right on the neck of a ukulele.

Because every chord on a ukulele can be played on the first five frets, we embedded LEDs in the first five rows of the fretboard. These LEDs display a certain chord until the user plays it correctly by strumming all four strings. Upon successful execution of the chord, the rows of LEDs quickly flash from fret one to fret five signaling this success to the user, immediately before displaying the next chord in the song. Once the song is completed, the LEDs show the user a “victory” sequence for a few seconds.

The notes played correspond to the frequency of each string, which can be found by sensing the vibrations on the body of the ukulele via contact microphone. The contact microphone is composed of a piezo element and a signal conditioning circuit. The analog signal from the piezo element is sent through a 4th order low-pass filter before being converted to a digital signal through the analog to digital converter (ADC). The FPGA uses a SPI protocol to get the ADC data and perform a fast fourier transform (FFT) in hardware. This 1024-point FFT is then sent via SPI to the Raspberry Pi for analysis and LED control. The Pi uses the FFT data to find the frequencies with the highest magnitude and compares these notes to the correct notes being displayed on the fretboard. If the chords match, the song proceeds to display the next chord. If they don’t, then the LEDs keep displaying the chord. This high-level datapath is summarized in the block diagram below in Figure 1. The following sections describe and illustrate each module in detail.

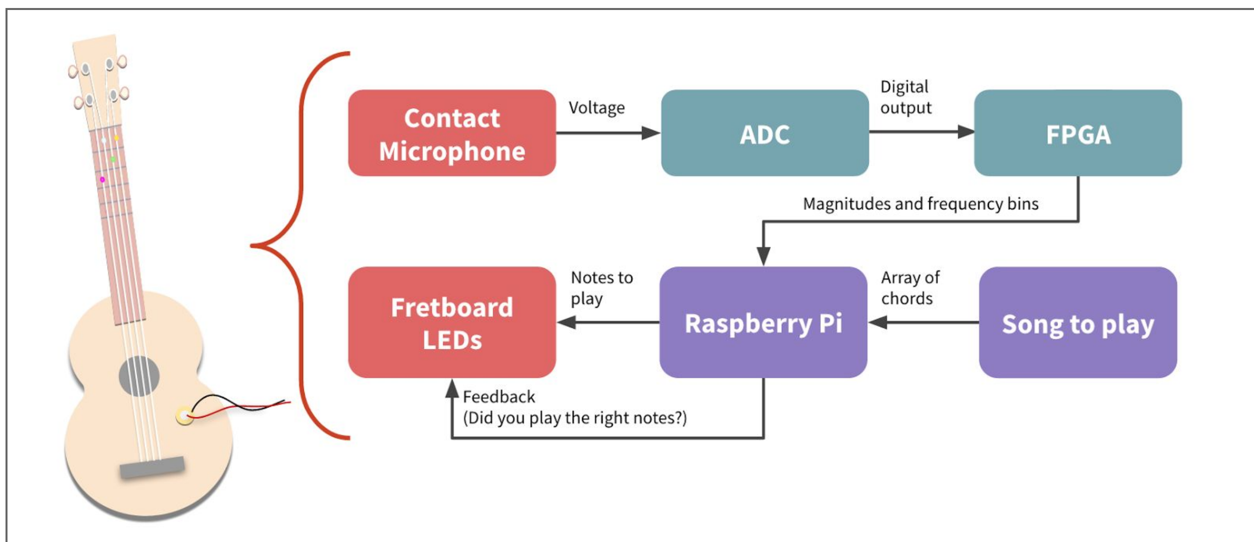


Figure 1: High-level system block diagram. The Ukucorn’s datapath from vibration data to chord recognition and feedback LEDs.

Hardware

Several new pieces of hardware were used in designing the Ukucorn including a ukulele, a piezo element, ICs for filtering, and many thin LEDs.

Ukulele	A DIY ukulele kit was needed to easily modify the neck and fretboard to embed the LEDs.
Piezo Sensor	A piezo sensor was securely taped to the body of the ukulele near the strings to sense vibrations. The usual output seen from this piezo sensor had amplitudes on the order of a few hundred millivolts as measured by strumming various chords.
LTC6240	Considering the sensitivity of audio signals, the LTC6240 op-amp was chosen for use in the low-pass filter given its low noise characteristics. [1]
LilyPad LEDs	These thin LEDs were chosen so they could be embedded with minimal sanding of the fretboard and neck. Additionally, each LED was mounted on its own thin PCB with a 151 Ω resistor in series. When connected from 3.3 V to ground, each LED only pulls 3 mA of current, well below the maximum current draw for Raspberry Pi GPIO pins.

Ukulele Fabrication

Ukulele fabrication presented several mechanical challenges regarding the fretboard LEDs. Playing a ukulele requires the user to firmly press down on various strings in the same places the LEDs should be visible. To minimize interference from the modifications on the user, only very small holes were drilled to allow light to come through. This ultimately allowed the ukulele to function as a normal, playable instrument, and eliminated the possibility of splintering the user. A countersink was used under each of the drilled holes so the LEDs could be as flush as possible with the back of the fretboard. The LEDs were wired up in rows and columns, and hot glued in place.

The neck was modified to fit the rest of the LED array protruding from the fretboard and all its wires. A cylindrical sander was used to hollow out the neck until the LED “sandwich” became flush as shown below in figure 2. A 5/16” hole was also drilled at the neck base to feed the nine wires out the back.

Figure 2: LED “sandwich.” The glue holding the LED array in place can be seen between the modified fretboard and neck.

The rest of the ukulele was assembled using the parts in the kit, wood glue, and superglue. The tuning pegs were screwed in, and the neck and bridge were glued to the body. After waiting an hour for the glue to dry, The neck and fretboard were glued and clamped together for another two hours. The four strings were attached and tuned, resulting in a fully functional ukulele.

Figure 3: Full Ukulele Assembly. The small holes shown on the fifth fret have little to no impact on the user, while the first four frets show these holes allow adequate light to shine through, even in daylight and brightly lit rooms.

Contact Microphone

We created a contact microphone from a raw piezoelectric element. A quick probe of the signal (produced by attaching the sensor to the ukulele and plucking a few strings) revealed the following issues:

1. The peak-to-peak voltage for a fairly hard strum reaches only up to 500 mV, i.e., the signal needs to be amplified especially with the limited precision of the 10-bit ADC.
2. Voltages oscillate between positive and negative relative to center, i.e., a DC offset is required if operating between 0 and +3.3 V.
3. The signal exhibits many high frequency components, which could impact our FFT analysis if they are not removed.

We then developed the following specifications for our signal conditioning circuit, which includes a fourth-order low-pass filter:

1. Single supply from 0 to +3.3 V
2. Cut-off frequency of around 1 kHz
3. Biased at +1.6 V
4. Amplification such that a hard strum produces a peak-to-peak amplitude of 1 V

The schematic for our signal conditioning circuit is illustrated in Appendix A. First, we apply +1.5 V offset to the signal. A fourth-order Butterworth filter with a cut-off frequency of approximately 1.06 kHz removes the high frequency components. Finally, we recenter the signal at +1.6 V and apply a gain of 2.5 via an instrumentation amplifier. The pre- and post-conditioned signal are shown in Figure 5.

Figure 5: Sample Waveform. Oscilloscope capture of the contact microphone output when the A string of the ukulele is plucked. The green trace shows that before any filtering, the output exhibits a fair amount of high frequency noise and small peak-to-peak amplitude. After filtering, most of the high frequency noise has been removed and a gain has been applied.

FPGA

System Overview

The primary functions of the FPGA are to:

1. Sample data from the ADC via SPI
2. Load data into memory when data exceeds a certain threshold
3. Perform FFT on 1024-point data
4. Communicate the results of FFT to the Raspberry Pi via SPI

These functions are performed in separate states as described below.

listen	Wait for the `piReady` signal from Raspberry Pi to before entering idle. (Default state.)
idle	Sample data from the ADC. Continuously check if the data exceeds the digital quivalent of 1.6 V.
load	Obtain 1024 samples from the ADC and store into memory in bit-reversed order
fft	Calculate the complex coefficients of the FFT transform of the data
compute	Calculate the complex magnitude at each frequency bin once FFT is complete
send	Send the magnitude and frequency bin information to the Raspberry Pi

The block diagram provided in Appendix B describes the modules, memory blocks, and state transitions of the FPGA. The SystemVerilog code of for the main module is provided in Appendix C3.

Fast Fourier Transform

The FPGA implements a pipelined FFT architecture as described in “The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation” by George Slade [2].

We chose our FFT parameters based on a frequency range of interest is from 261.63 Hz (C4) to 553.37 Hz (C5). By the Nyquist Theorem, our minimum sampling frequency would be 2 x 553.37 Hz = 1066.74 Hz, but for good measure we used 4884 Hz, approximately four times our largest frequency of interest and also easily implemented by dividing the FPGA's 40 MHz clock. To differentiate between notes, we required a frequency resolution of less than 8 Hz because the smallest frequency interval between notes (C4 to C#4) is 16 Hz. These constraints required that our number of samples N = 4884 Hz / 4 Hz = 1221, or N = 1024 (closest power of 2).

We first simulated the FFT in ModelSim with the 32-point data set provided in Slade's paper. Figure 6 features a snapshot of our simulation results and shows indications of a properly orchestrated process. Once simulation results were deemed satisfactory, we tested the FFT on a real system with 1024-point data. Figure 7 illustrates the FFT results from plucking the ukulele string individually as well as strumming all four strings together. The SystemVerilog code for this process is provided in Appendix C1.

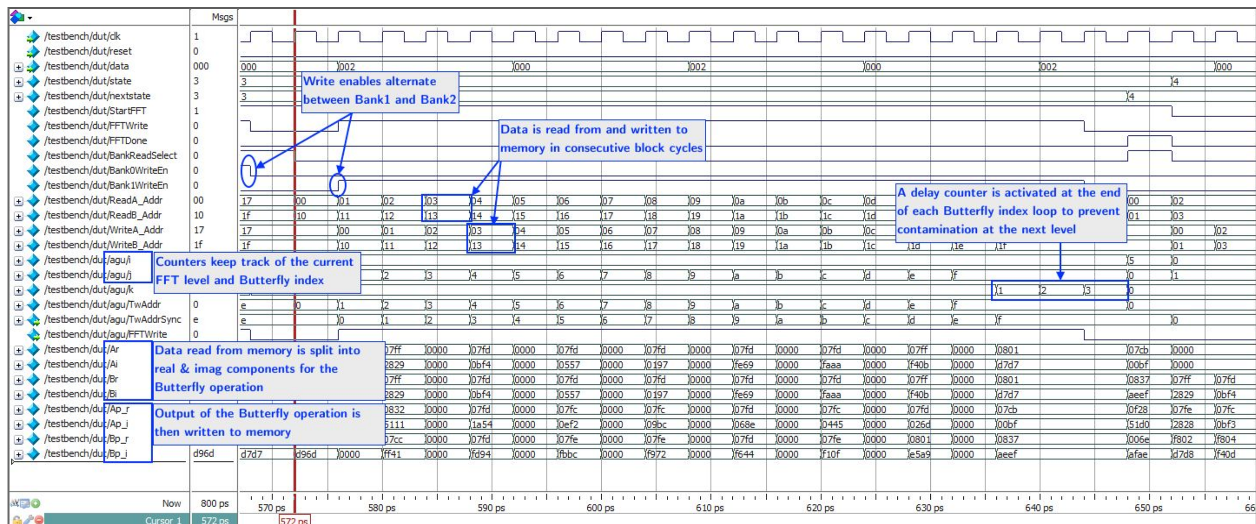


Figure 6: Timing diagram highlighting the key components to the pipelined FFT process.

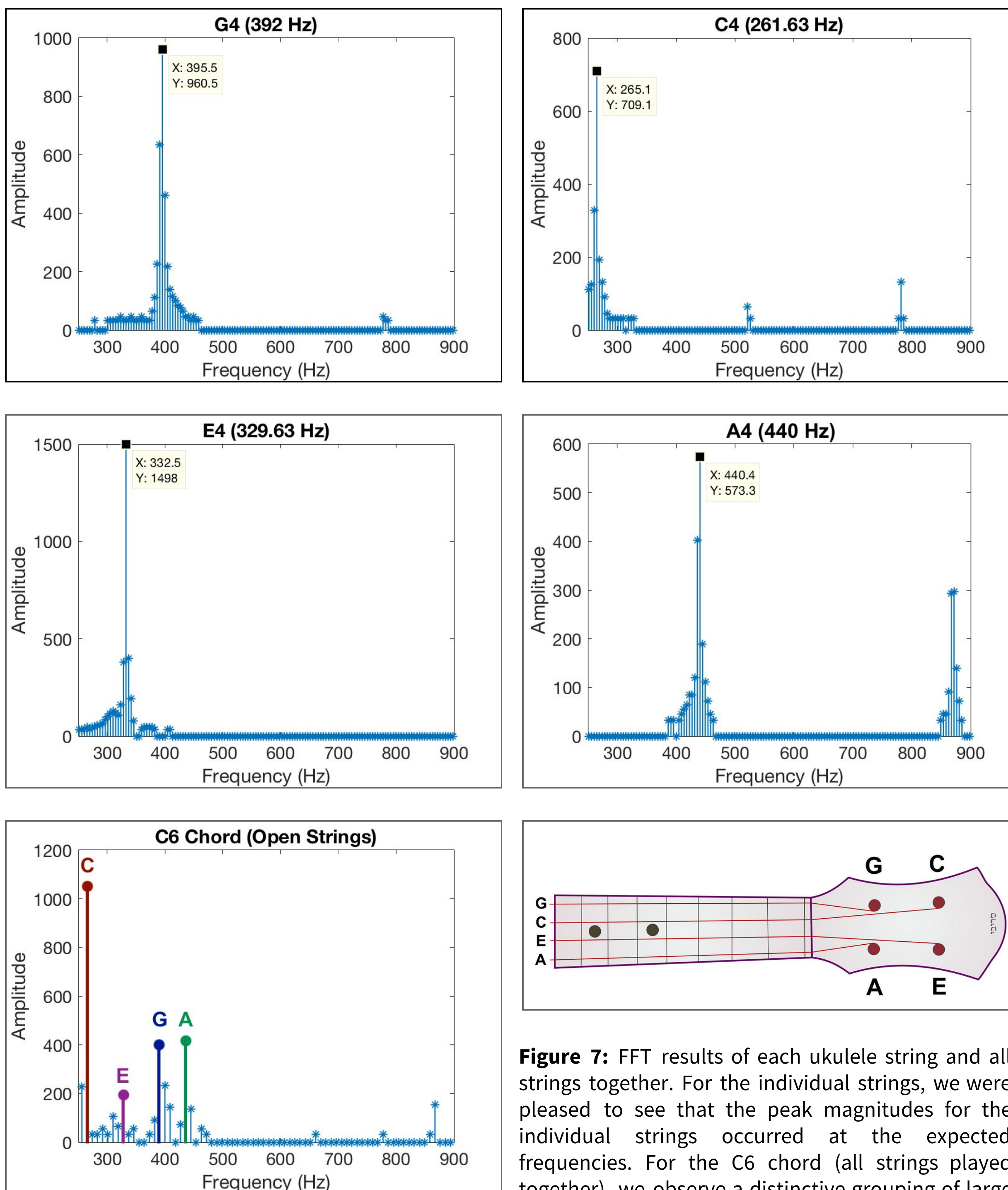


Figure 7: FFT results of each ukulele string and all strings together. For the individual strings, we were pleased to see that the peak magnitudes for the individual strings occurred at the expected frequencies. For the C6 chord (all strings played together), we observe a distinctive grouping of large magnitudes that match up with each string.

Interface with the ADC

Our desired sampling frequency is 4884 Hz, but data from the ADC is obtained over 16 clock cycles. Therefore, the FPGA samples the ADC at frequency of 16 x 4884 Hz = 78144 Hz. The ADC and FPGA communicate over SPI, with the ADC acting as the slave and the FPGA acting as the master.

The FPGA continuously samples the ADC and only stores samples when the data exceeds a threshold that corresponds to about 1.6 V, upon which it enters the `load` state. Prior to each sample, `adcEn` is pulsed high for one clock cycle to enable data transfer. After each complete sample (16 clock cycles), the data is latched onto

a flip-flop and loaded into memory in bit reversed order. This repeats until 1024 samples have been obtained, after which the FPGA exits the `load` state and enters the `fft` state. Figure 9 provides a block diagram overview of the primary signals and counters involved in this process. The SystemVerilog code for the SPI master module is provided in Appendix C2.

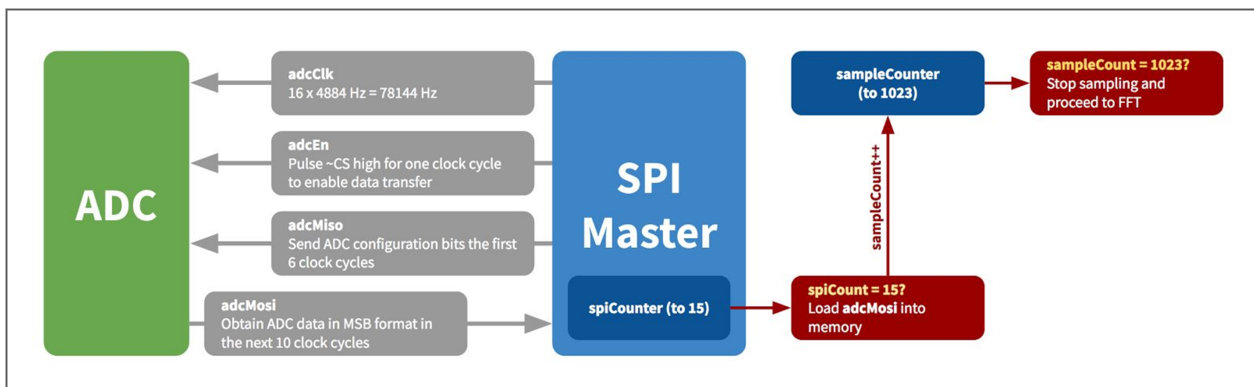


Figure 9: Block diagram of SPI communication with the ADC and counters that keep track of the number samples that have been stored into memory.

Interface with Raspberry Pi

Once the FPGA is finished processing the FFT, we reduce the information transfer to the Raspberry Pi by computing the 16-bit complex magnitude at each frequency bin, which is then written to the memory block `spiRam`. This is serves as the communication hub for the Raspberry Pi and the FPGA, which acts as the master and slave, respectively. The SPI slave (on the FPGA) reads and sends the data at `spiAddr` starting from 0. Because data is 16 bits long, each magnitude / frequency bin pair must be transferred over two SPI requests. To coordinate this, `spiCycle` alternates between 0 and 1 between every SPI request. The SPI slave sends the last 8 bits if `spiCycle` is 0 and the first 8 bits if `spiCycle` is 1. After each pair of the SPI requests, `spiAddr` increments so that the SPI slave can proceed to the next sample. This process is illustrated in Figure 10. The SystemVerilog code for the SPI slave module is provided in Appendix C2.

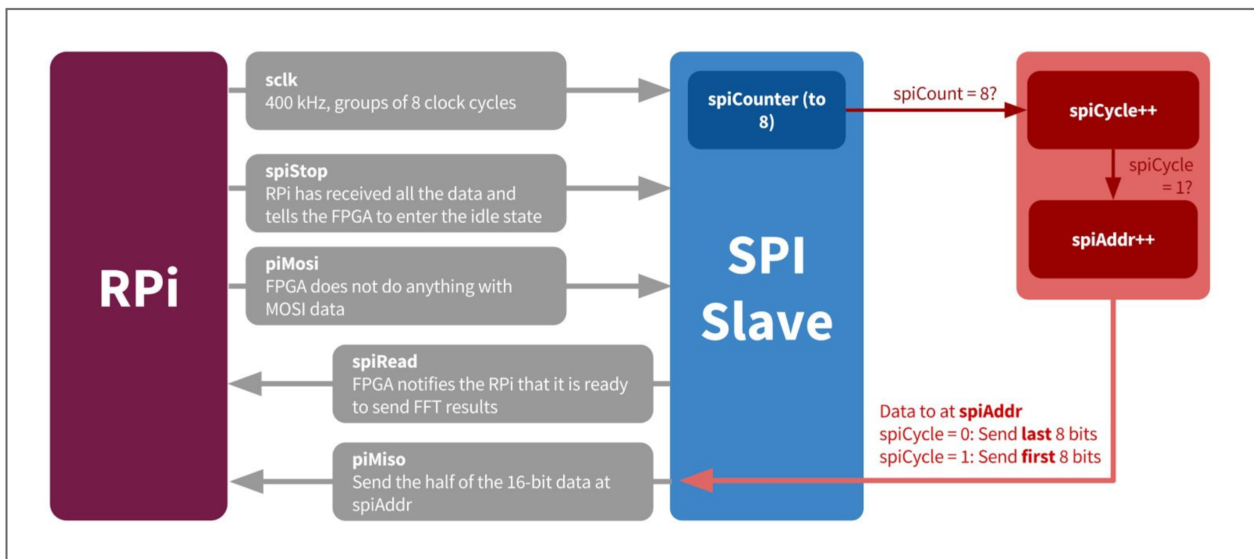


Figure 10: Block diagram of SPI communication with the Raspberry Pi and counters that keep track of the number samples that have been sent.

Raspberry Pi

We implemented a couple of key algorithms on the Raspberry Pi ("Pi"), including chord recognition and LED matrix control, both of which can be found in `ukuDemo.c` (Appendix D1). Additionally, a long list of chords and songs were encoded as arrays. Every major, minor, and 7th chords are encoded in the header file `uku.h`, and each of these chords is encoded as an array of four `const ints`. Songs were encoded as two dimensional arrays by listing a series of chords (Appendix D2).

The only inputs to the Pi were from the SPI protocol with the FPGA. The outputs included four GPIO pins for the strings, five GPIO pins for the frets, and outputs necessary for the SPI protocol.

User Interface

The user must access the Pi's terminal to start a song on the Ukucorn. After running the executable `./ukuDemo`, the terminal prints a list of seven songs and asks the user to enter a song number in the command line. The number entered sets the song's size `songSize` and pointer to the song array `songPtr`. The main `while` loop then runs until the song is complete, i.e., when the counter `chordsSoFar` reaches `songSize`.

LED Matrix Controller

The LED array is controlled using time multiplexing where frets and strings can be treated as rows and columns. For any specific LED to turn on, its *string* must be pulled low while its *fret* is pulled high. This was implemented in the function `displayNote()` which takes in a string GPIO and fret GPIO, and displays a single note before clearing it one millisecond later. `displayChord()` time multiplexes the strings by running `displayNote()` for each of the four strings while iterating through the chord array for the *fret* input.

Interface with the FPGA

In the main loop, the Pi continuously polls the FPGA for the `spiRead` signal, which indicates that the FPGA is ready to send FFT results. When `spiRead` is high, the Pi enters `while` loop of `spiSendReceive()` requests. The results of two successive `spiSendReceive()` requests form a 16-bit sample, which is stored to `fftArray`. When 1024 samples have been received, the Pi exists this loop proceeds to the next set of instructions in the main loop.

Chord Recognition

Our chord recognition algorithm assumes that each ukulele chord contains four notes. In theory, the frequencies associated with the four largest magnitudes are the four notes that have been played.

At the beginning of the main loop, the Pi extracts the current chord to be played, i.e. the `chordsSoFar`th array in of the song array, and encodes the notes in `encurrentChord` by assigning a number from 0 to 15, with 0 being the lowest note (C4) and 15 begin the highest note (C5).

Once the `fftArray` has been populated, the Pi finds the maximum magnitudes in the range of frequency bins associated with each note. To determine the four notes played, the indices of the four maximum magnitudes are stored in `maxChord`. Because we encode the notes using integers between 0 and 15, by knowing the index of the maximum values, we also know the note at which the maximum magnitudes occur.

A counter `score` is used to compare the measured chord (`maxChord`) to the target chord (`encurrentChord`). If each note in `maxChord` matches with any note in `encurrentChord`, `score` will increment. When it reaches 4, and the next chord will be displayed after running the indicator LED function `goodJob()`. At this point, `chordSoFar` will also increment to refresh the LED display and show the next set of notes. If `chordSoFar` equals `songSize`, the player has successfully played all the chords correctly. The Pi will exit the while loop, and before exiting the program, `victory()` will run to show an exciting sequence of flashing LEDs.

Results

The system performs as expected and meets all goals presented in our initial proposal. The Ukucorn will occasionally fail to register the correct chord being played, but this is largely due to the ukulele being out of tune or the Raspberry Pi freezing. When everything is running properly and the ukulele is in tune, failure to proceed in the song is due to the user playing the displayed chord incorrectly. The Ukucorn is a strict teacher and will not waver for a frustrated user. During Demo Day, we were able to entertain many students with this new learning device. Overall, we are very proud of our product.

We encountered challenges with both the Raspberry Pi and FPGA. In addition to the challenge of implementing FFT on the FPGA, we learned that it was difficult to coordinate the SPI and FFT modules and nearly impossible to debug the whole system if each of the modules were not tested individually before integration.

We were also surprised to experience major problems debugging C code on the Pi. While we wanted to be more concise with our code by using `switch` case statements, we encountered vague errors and eventually resorted to using verbose `if` statements. Additionally, we found that several of our arrays were too large to store as `const ints` and were forced to use pointers. Figuring out how to initialize and dereference pointers, especially double pointers, presented major difficulties. However, we were able to demonstrate a working product at the end and treat all of these challenges as valuable learning experiences.

References

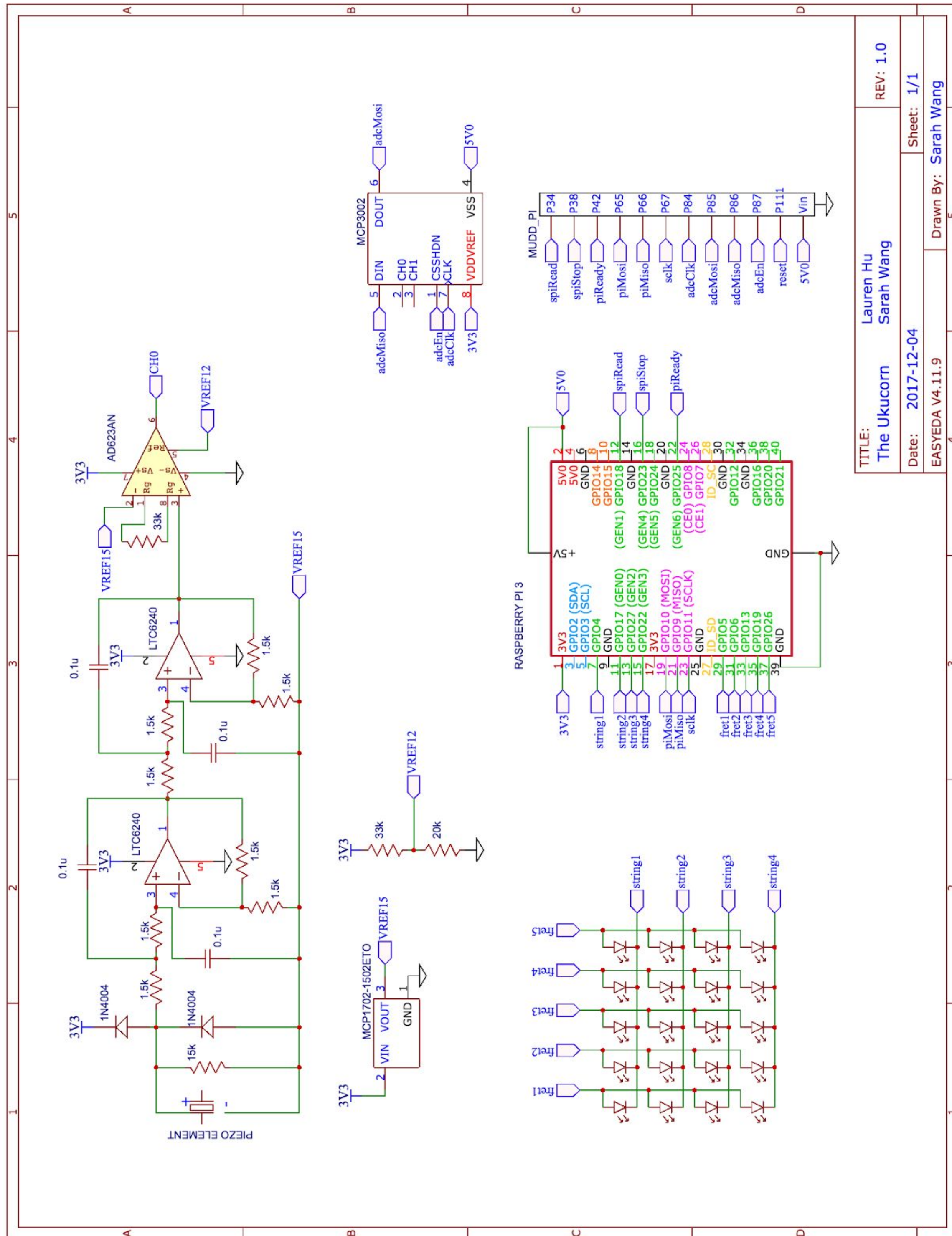
[1] Linear Technology, *LTC6240 Single 18MHz, Low Noise, Rail-to-Rail Output, CMOS Op Amp,* http://cds.linear.com/docs/en/datasheet/624012fe.pdf

[2] G. William Slade, *The Fast Fourier Transform in Hardware: A Tutorial Based on FPGA Implementation*, 2013.

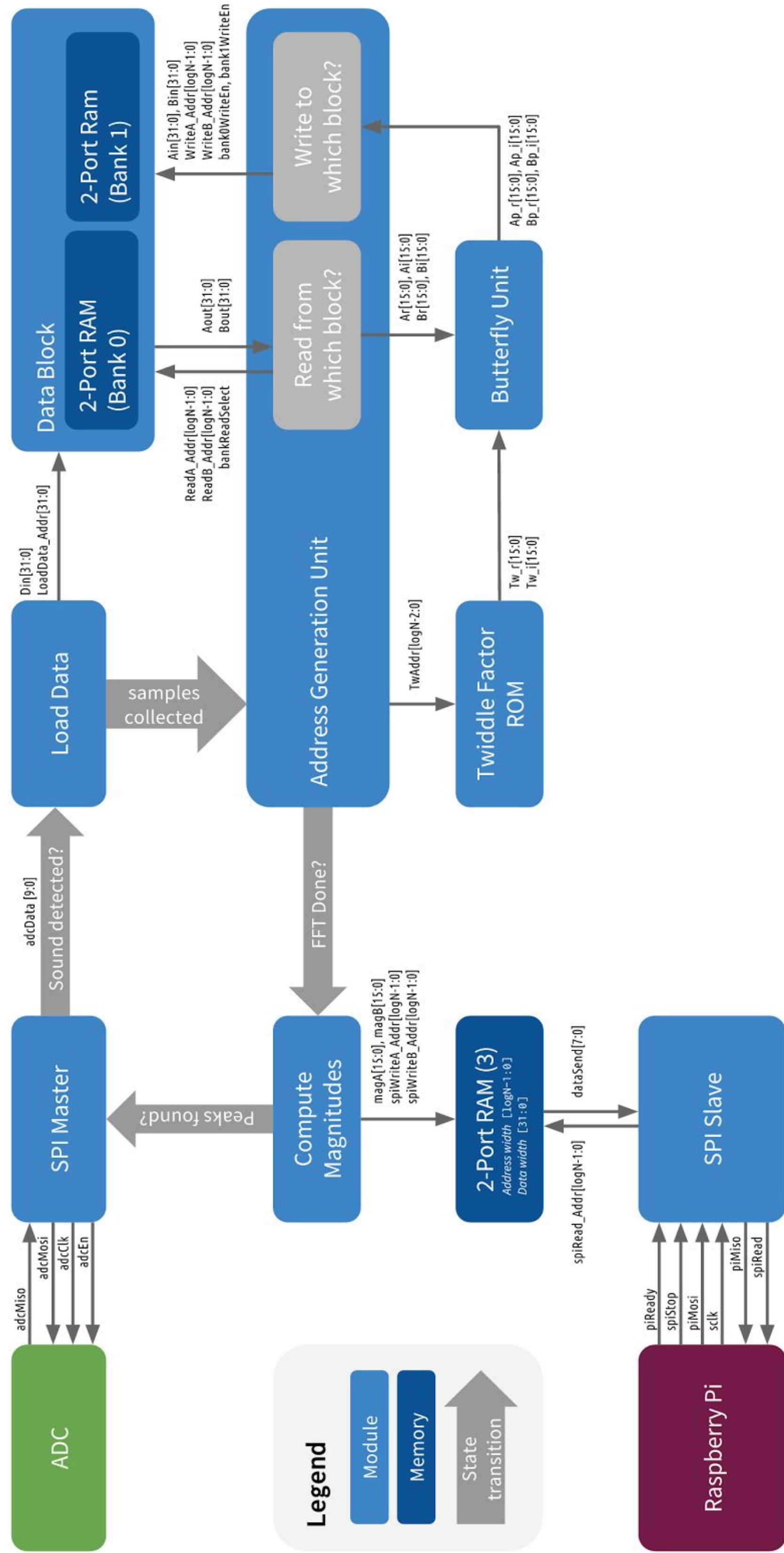
Parts List

Part	**Vendor + P/N**	**Qty**	**Unit Cost**	**Subtotal**	**Link**
Low-Noise op-amp	Linear Technology, LTC6240	2			LTC6240 datasheet
Instrumentation amplifier	Texas Instruments, AD623AN	1			AD623AN datasheet
Analog-to-digital converter	MCP3002	1			MCP3002 datasheet
DIY Ukulele Kit	Amazon	1	$39.99	$39.99	Amazon listing
Small Enclosed Piezo w/Wires	Adafruit	2	$0.95	$1.90	https://www.adafruit.com/product/1740
LEDs (packs of 5)	Sparkfun	6	$2.95	$17.70	https://www.sparkfun.com/products/14010
LED (rainbow pack)	Sparkfun	1	$3.50	$3.50	https://www.sparkfun.com/products/13903
Shipping - UPS Ground	Adafruit	1	$9.15	$9.15	
Shipping - UPS Ground	Sparkfun	1	$7.59	$7.59	
			Total	**$93.77**	

Appendix A: Full Schematic



Appendix B: FPGA Block Diagram



Appendix C1: FPGA SystemVerilog - fft.sv

```
/////////////////////////////////////////
//                fft.sv               //
/////////////////////////////////////////

// Enabled counter with sync reset
module counter_en #(parameter N = 8)
                                 (input logic clk, reset, en,
                                  output logic [N-1:0] q);
       always_ff@(posedge clk)
              if (reset)    q <= 0;
              else if (en) q <= q + 9'b1;
endmodule

// Enabled counter with async reset (non-enabled)
module counter_en_async #(parameter N = 8)
                                 (input logic clk, reset, en,
                                  output logic [N-1:0] q);
       always_ff@(posedge clk, posedge reset)
              if (reset)    q <= 0;
              else if (en) q <= q + 1;
endmodule

// Counter with async reset (non-enabled)
module counter #(parameter N = 8)
                                 (input logic clk, reset,
                                  output logic [N-1:0] q);
       always_ff@(posedge clk, posedge reset)
              if (reset)    q <= 0;
              else q <= q + 1;
endmodule

/*
 * Address Generation Unit
 * Functions:
 *   1) Generate addresses for reading and writing of data RAM
 *   2) Retrieve twiddle factors
 *   3) Generate write signals for the data RAM
 * Keep track of:
 *   1) Which butterfly we are executing
 *   2) Which FFT level we are working on
 */

module agu
  #(parameter logN = 10)
       (input logic clk,
        input logic startFFT,
        output logic fftDone,
        output logic [logN-1:0] MemA_Addr, MemB_Addr,
        output logic [logN-2:0] TwAddrSync,
```

```
        output logic fftWrite,
        output logic bankReadSelect);

       logic [3:0] i; // Level of FFT counter
       logic [logN-2:0] j; // Butterfly index counter
       logic [logN-1:0] ja, jb; // For calculating addresses
       logic [1:0] k; // Wait until A' and B' are written to memory

       // SyncBronize startFFT and assert clearFFT
       logic startFFT1;
       logic clearFFT;
       logic fftEnable;

       always_ff @(posedge clk) begin
         startFFT1 <= startFFT;
              if ((i == logN - 1) & &k) fftDone <= 1;
              else fftDone <= 0; end

       assign clearFFT = startFFT & ~startFFT1;
       assign fftEnable = startFFT & startFFT1;

       logic i_reset, j_reset, k_reset;
       logic i_en, j_en, k_en;

       counter_en #(4) fftCounter(clk, i_reset, i_en, i); // FFT Level counter
       counter_en #(logN-1) butterflyCounter(clk, j_reset, j_en, j); // Butterfly
index counter
       counter_en #(2) delayCounter(clk, k_reset, k_en, k); // Delay counter to flush
pipeline

       assign i_reset = clearFFT | (i == logN);
       assign j_reset = clearFFT | &k; // Reset when delay counter maxes out
       assign k_reset = clearFFT | &k; // Automatic reset

       assign i_en = fftEnable & ((&k) & (i < logN)); // enable when j < N/2 - 1
       assign j_en = fftEnable & (~(&j)); // enable when k = 0
       assign k_en = fftEnable & (&j); // enable when j is maxed out

       // Generate addresses for data and twiddles
       assign ja = j << 1;
       assign jb = ja + 1;
       assign MemA_Addr = (ja << i) | (ja >> (logN - i));
       assign MemB_Addr = (jb << i) | (jb >> (logN - i));

       // Twiddle mask generator - a right sBift reAister
       // that fills up with 1s as the level counter is incremented
       logic [logN-2:0] ones = ~0;
       logic [logN-2:0] zeros = 0;

       logic [logN-2:0] TwAddr;

       assign TwAddr = ({ones, zeros} >> i) & j;
```

```
	always_ff @(posedge clk)
		TwAddrSync <= TwAddr;


	logic fftWriteLow;

	assign bankReadSelect = i[0];
	assign fftWriteLow = &k | (j == 0);
	assign fftWrite = fftEnable & ~fftWriteLow;

endmodule // agu

/*
 * Butterfly Unit
 *   1) Complex multiply and add
 *   2) Also contains look-up table of twiddle factors
 */
module bfu
	#(parameter logN = 10)
	(input logic signed [15:0] Ar, Ai, // A
	 input logic signed [15:0] Br, Bi, // B
	 input logic [logN-2:0] TwAddr,
	 output logic signed [15:0] Ap_r, Ap_i,  // A'
	 output logic signed [15:0] Bp_r, Bp_i); // B'

	logic signed [15:0] Tw_r, Tw_i;

	// Twiddle ROM
	// parameter RomDepth = 1 << (logN - 1); // N/2
	logic [15:0] TwRom_r[0:511]; // 16-bit entries, N total
	logic [15:0] TwRom_i[0:511];
	initial $readmemh("Tw1024_r.txt",TwRom_r);
	initial $readmemh("Tw1024_i.txt",TwRom_i);

	assign Tw_r = TwRom_r[TwAddr];
	assign Tw_i = TwRom_i[TwAddr];

	logic signed [31:0] temp_r, temp_i;
	logic signed [15:0] temp_r_p, temp_i_p; // pruned version of T1

	// Complex multiplier
	assign temp_r = (Br * Tw_r) - (Bi * Tw_i);
	assign temp_i = (Br * Tw_i) + (Bi * Tw_r);

	// Prune T1
	assign temp_r_p = temp_r[30:15];
	assign temp_i_p = temp_i[30:15];

	// Complex adder for A'
	assign Ap_r = Ar + temp_r_p;
	assign Ap_i = Ai + temp_i_p;
```

```
	// Complex adder for B'
	assign Bp_r = Ar - temp_r_p;
	assign Bp_i = Ai - temp_i_p;
endmodule // BFU

// Quartus II Verilog Template
// Single Port ROM

/*
 * http://www.cs.columbia.edu/~sedwards/classes/2015/4840/memory.pdf
 */
module TwoPortRAM
	#(parameter width = 32,
	  parameter logN = 10)
	(input logic clk,
	 input logic [logN-1:0] AddA, AddB, // address
	 input logic [width-1:0] Ain, Bin, // data in
	 input logic WriteA, WriteB, // write enables
	 output logic [width-1:0] Aout, Bout);

	// parameter N = 1 << logN; // N = 2^logN
	logic [width-1:0] Mem [1023:0];

	always_ff @(posedge clk) begin
		if (WriteA) begin
			Mem[AddA] <= Ain;
			Aout <= Ain; end
		else Aout <= Mem[AddA]; end

	always_ff @(posedge clk) begin
		if (WriteB) begin
			Mem[AddB] <= Bin;
			Bout <= Bin; end
		else Bout <= Mem[AddB]; end
endmodule // TwoPortRAM

/* DataBlock
 *   1) Contains two memory blocks (bank0 and bank1)
 *      2) A0_Addr is used to load data into bank0
 */

module DataBlock
	#(parameter logN = 10)
	(input logic clk, loadData, bank0WriteEn, bank1WriteEn, bankReadSelect,
	 input logic [logN-1:0] loadData_Addr, readA_Addr, readB_Addr, writeA_Addr,
writeB_Addr,
	 input logic [31:0] Din, Ain, Bin,
	 output logic [31:0] Aout, Bout);

	logic bank0_AWrite;
	assign bank0_AWrite = loadData | bank0WriteEn;
```

```
	logic [logN-1:0] A0_Addr, A1_Addr;
	logic [logN-1:0] B0_Addr, B1_Addr;

	assign A0_Addr = loadData ? loadData_Addr : (bank0WriteEn ? writeA_Addr :
readA_Addr);
	assign A1_Addr = bank1WriteEn ? writeA_Addr : readA_Addr;
	assign B0_Addr = bank0WriteEn ? writeB_Addr : readB_Addr;
	assign B1_Addr = bank1WriteEn ? writeB_Addr : readB_Addr;

	logic [31:0] A0out, A1out, B0out, B1out;
	logic [31:0] A0in;

	assign A0in = loadData ? Din : Ain;

	TwoPortRAM #(32, logN) Ram0(clk, A0_Addr, B0_Addr, A0in, Bin, bank0_AWrite,
bank0WriteEn, A0out, B0out);
	TwoPortRAM #(32, logN) Ram1(clk, A1_Addr, B1_Addr, Ain, Bin, bank1WriteEn,
bank1WriteEn, A1out, B1out);

	assign Aout = bankReadSelect ? A1out : A0out;
	assign Bout = bankReadSelect ? B1out : B0out;
endmodule // DataBlock
```

Appendix C2: FPGA SystemVerilog - spi.sv

```
//////////////////////////////////////////
//                spi.sv               //
//////////////////////////////////////////

module spiMaster
  #(parameter div = 2)
       (input logic clk, reset,
        input logic adcEn,
        output logic adcSelect, // Slave select
        input logic miso, // From slave
        output logic mosi, // To slave
        // output logic sample_done,
        output logic [9:0] q,  // Data sampled
        output logic sClk, // sClk to assign
        output logic [3:0] spiCounter);

  // generate the slave clock
  // parameter div = 2; // sample at ....
  logic [div-1:0] sClkCounts;
  counter #(div) sClkCounter(clk, reset, sClkCounts);
  assign sClk = sClkCounts[div-1];

  // A/D_CS_BAR
  assign adcSelect = ~adcEn;
```

```
  // counts from 0 to 15 during sample period

  // adc config bits
  logic [15:0] adc_config;

  // configure bits for MCP3002
  logic channel;
  assign channel = 1'b0;
  assign adc_config = {13'bxxxx_xxxx_xxxx_1, channel, 2'b11};

  // first four clock cycles are ADC config
  always_ff@(posedge sClk, posedge reset)
    if (reset) spiCounter <= 0;
    else if (adcEn) begin
        // 0 - 4th bit are adc configuration bits
        mosi <= adc_config[spiCounter];
        // shift register to load slave data
        if ((spiCounter >= 5) && (spiCounter < 15))
          q[14-spiCounter] <= miso;
        // increment the counter
        spiCounter <= spiCounter + 1; end
endmodule



module spiSlave
       (input logic sck, reset,  // From master
        input logic mosi, // From master
        output logic miso, // To master
        input logic [7:0] d, // Data to send
        output logic [7:0] q,
        output logic [2:0] cnt); // Data received

        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
```

Appendix C3: FPGA SystemVerilog - ukucorn.sv

```
//////////////////////////////////////////
//              ukucorn.sv              //
//////////////////////////////////////////

module test
       (input logic gClk, reset, // pin 111
        input logic adcStart,
        output logic adcEnLed,
        output logic adcDoneLed,
        output logic loadEnLed,
        output logic loadResetLed,
        output logic spiCycle,

        // Pi signals
        input logic piReady,
        input logic spiStop,
        input logic sclk, // 67
        input logic piMosi, // 65
        output logic piMiso, // 66
        output logic spiRead,

        // ADC signals
        input logic adcMiso, // 85, read Dout
        output logic adcClk, // 84
        output logic adcSelect, // 87
        output logic adcMosi, // 86, assign Din

        // Debug signals
        output logic [2:0] stateLed);

       parameter logN = 10; // N = number of samples
       parameter divClk = 3; // Slow down the clock to 5 MHz
       logic clk;
       logic [divClk-1:0] clkCount;
       counter #(divClk) clkCounter(gClk, reset, clkCount);
       assign clk = clkCount[divClk-1];

       /////////////////////////////////////
       //////          STATES         //////
       /////////////////////////////////////

       // listen: wait for piReady signal
       // idle: nothing to do...
       // load: sample from ADC and load data into ram0
       // send: send data to Pi over SPI
       logic [2:0] state, nextstate;
       typedef enum logic [2:0] {listen, idle, load, fft, compute, send} statetype;
       assign stateLed = state;
```

```
	// Next state register
	always_ff @(posedge clk, posedge reset)
		begin
			if (reset) state <= listen;
			else state <= nextstate;
		end

	//////////////////////////////////////
	//////    SAMPLE & LOAD DATA    //////
	//////////////////////////////////////

	// parameter threshold = 0;

	logic adcEn; // adcSelect,
	logic adcDone;
	logic adcEn1; //, adcDone1;
	logic [9:0] adcData;
	logic [3:0] adcCount;
	logic adcEnLow;

	always_ff @(posedge adcClk)
		adcEn1 <= adcDone;

	assign adcEn = ((state == idle) | (state == load)) & ~adcEn1;
	assign adcDone = &adcCount;


	parameter divAdcClk = 6; // log2( 5 MHz / (4096 * 16) ) ~ 9
	spiMaster #(divAdcClk) master2adc(clk, reset, adcEn, adcSelect, adcMiso,
adcMosi, adcData, adcClk, adcCount);

	assign adcEnLed = adcEn;
	assign adcDoneLed = adcDone;

	logic [9:0] memData;
	logic [logN-1:0] loadCount;
	logic loadReset, loadEn;
	assign loadResetLed = loadReset;
	assign loadReset = (state != load);
	assign loadEn = (state == load) & adcEn1;
	assign loadEnLed = loadEn;
	counter_en #(logN) loadCounter(adcClk, loadReset, loadEn, loadCount); // Load
data address counter

	always_ff @(negedge adcDone) begin
		memData <= adcData; end

	logic loadDone; // signal to exit load state
	logic loadLast; // delayed loadEn for the last sample
	assign loadDone = loadLast & ~loadEn; // level to pulse converter for loadDone
```

```
always_ff @(posedge clk) begin
       if (&loadCount & loadEn) loadLast <= 1;
       else loadLast <= 0; end

//////////////////////////////////////
//////   BIT-REVERSED ADDRESS   //////
//////////////////////////////////////

logic [logN-1:0] loadData_Addr;

genvar i;
generate
for (i = 0; i < logN; i = i + 1)
      begin: bitReverse
      assign loadData_Addr[i] = loadCount[logN-1-i];
      end
endgenerate

logic signed [15:0] misoLong;
always @(posedge clk)
    if (loadCount < 256) misoLong <= 16'h03ff; // 1
    else misoLong <= 16'hfc01; // -1

//////////////////////////////////////
//////            FFT           //////
//////////////////////////////////////
logic startFFT;
assign startFFT = (state == fft);

// AGU
logic fftWrite, fftDone;
logic fftReadSelect;
logic bankReadSelect;
logic [logN-2:0] TwAddr;
logic [logN-1:0] readA_Addr, readB_Addr;
logic [logN-1:0] fftReadA_Addr, fftReadB_Addr;
logic [logN-1:0] fftWriteA_Addr, fftWriteB_Addr;

agu #(logN) agu0(clk, startFFT, fftDone,
 fftReadA_Addr, fftReadB_Addr, TwAddr,
 fftWrite, fftReadSelect);

always_ff @(posedge clk) begin
             fftWriteA_Addr <= fftReadA_Addr;
             fftWriteB_Addr <= fftReadB_Addr;
       end

// Data block
logic bank0WriteEn;
logic bank1WriteEn;
assign bank0WriteEn = fftReadSelect & fftWrite; // Read from 1, write to 0
```

```
	assign bank1WriteEn = ~fftReadSelect & fftWrite; // Read from 0, write to 1

	logic [15:0] zero16 = 16'b0;
	logic [5:0] zero6 = 6'b0;
	logic [31:0] Din, Ain, Aout;
	logic [31:0] Bin, Bout;

	logic [logN-1:0] magA_Addr; // Read
	logic [logN-1:0] magB_Addr;

	assign bankReadSelect = startFFT ? fftReadSelect : 1'b1;
	assign readA_Addr = startFFT ? fftReadA_Addr : magA_Addr;
	assign readB_Addr = startFFT ? fftReadB_Addr : magB_Addr;

	DataBlock #(logN) datablock(clk, loadEn,
		bank0WriteEn, bank1WriteEn, bankReadSelect,
		loadData_Addr, readA_Addr, readB_Addr,
		fftWriteA_Addr, fftWriteB_Addr,
		Din, Ain, Bin, Aout, Bout);

	// Butterfly Unit
	logic signed [15:0] Ar, Ai;
	logic signed [15:0] Br, Bi;
	logic signed [15:0] Ap_r, Ap_i;
	logic signed [15:0] Bp_r, Bp_i;

	assign Ar = Aout[31:16];
	assign Ai = Aout[15:0];
	assign Br = Bout[31:16];
	assign Bi = Bout[15:0];

	assign Din = {zero6, memData, zero16};
	assign Ain = {Ap_r, Ap_i}; // A'r
	assign Bin = {Bp_r, Bp_i};

	bfu #(logN) bfu0(Ar, Ai, Br, Bi, TwAddr, Ap_r, Ap_i, Bp_r, Bp_i);

	// Find frequencies
		logic computeMag, computeMag1; //, computeMag2; // sketchy way to delay
spiRamWrite
	logic spiRamWrite;
	logic computeDone;
	logic [logN-1:0] spiWriteA_Addr, spiWriteB_Addr;
	logic [logN-1:0] spiReadA_Addr;
	logic [logN-1:0] spiA_Addr;

	assign computeMag = (state == compute);
	assign spiRamWrite = ~computeDone & computeMag & computeMag1;

	always_ff @(posedge clk) begin
		computeMag1 <= computeMag;
		// computeMag2 <= computeMag1;
```

```
       if (&spiWriteB_Addr) computeDone <= 1;
       else computeDone <= 0; end

logic clearSpiRam;
assign clearSpiRam = (nextstate == compute) & ~(state == compute);

always_ff @(posedge clk, posedge clearSpiRam)
 if (clearSpiRam) begin
   magA_Addr <= 0;
   magB_Addr <= 1; end
 else if (computeMag) begin
   magA_Addr <= magA_Addr + 2;
   magB_Addr <= magB_Addr + 2; end

// Compute magnitude
logic signed [31:0] magALong, magBLong;
logic signed [15:0] magA, magB;
logic [15:0] magAout, magBout;

always_ff@(posedge clk) begin
   spiWriteA_Addr <= magA_Addr;
   spiWriteB_Addr <= magB_Addr; end

assign magALong = Ar*Ar + Ai*Ai;
assign magBLong = Br*Br + Bi*Bi;
assign magA = magALong[30:15];
assign magB = magBLong[30:15];

assign spiA_Addr = computeMag ? spiWriteA_Addr : spiReadA_Addr;

////////////////////////
//////  SPI RAM   //////
////////////////////////
logic spiRamClk;
logic clearSpi;
logic spiSend, spiSend1;

assign spiRead = spiSend;
assign spiSend = (state == send);
assign spiRamClk = spiRead ? sclk : clk; // need to switch to sclk when reading
assign clearSpi = spiSend & ~spiSend1;

TwoPortRAM #(16, logN) spiRam(spiRamClk, spiA_Addr, spiWriteB_Addr,
       magA, magB, spiRamWrite, spiRamWrite, magAout, magBout);

//////////////////////////////////////////
//////  FPGA (slave) <-> Pi (Master) //////
//////////////////////////////////////////

logic [2:0] cnt;
logic [7:0] dataSend, dataReceive;
```

```
	always_ff @(posedge sclk)
		spiSend1 <= spiSend;

	assign dataSend = spiCycle ? magAout[15:8] : magAout[7:0];


	always_ff @(negedge sclk, posedge clearSpi)
		if (clearSpi) begin
		  spiReadA_Addr <= 0;
		  spiCycle <= 1; //
		end
		else if (&cnt) begin
			spiCycle <= spiCycle + 1; // alternates between 0 and 1
			if (spiCycle) spiReadA_Addr <= spiReadA_Addr + 1; end

	spiSlave slave2pi(sclk, clearSpi, piMosi, piMiso, dataSend, dataReceive, cnt);

	//////////////////////////////////////
	//////     NEXT STATE LOGIC     //////
	//////////////////////////////////////
	logic [9:0] trigger = 10'b1000001000;
	always_comb
	 case(state)
	    idle:    if (memData > trigger) nextstate = load; // something detected!
	            else nextstate = idle;
	    load:   if (loadDone) nextstate = fft; // N samples reached
	                         // if (loadDone) nextstate = echo;
	            else nextstate = load;
	    fft:    if (fftDone) nextstate = compute;
	            else nextstate = fft;
	    compute: if (computeDone) nextstate = send;
	                         else nextstate = compute;
	        // clearSpi: nextstate = send; // run for one clock cycle
	    send:    if (spiStop) nextstate = listen;
	                         else nextstate = send;
	        listen : if (piReady) nextstate = idle;
	                         else nextstate = listen;
	    default: nextstate = listen;
	 endcase
endmodule
```

Appendix D1: Raspberry Pi C - ukudemo.c

```
/////////////////////////////////////////////
//              ukuDemo.c                  //
/////////////////////////////////////////////

#include "EasyPIO.h"
#include "uku.h"

// clear all LEDs
void clearChord(void) {
	digitalWrite(string1,1);
	digitalWrite(string2,1);
	digitalWrite(string3,1);
	digitalWrite(string4,1);
	digitalWrite(fret1,0);
	digitalWrite(fret2,0);
	digitalWrite(fret3,0);
	digitalWrite(fret4,0);
	digitalWrite(fret5,0);
}

int displayNote(int stringn, int fretn) {
	clearChord();
	digitalWrite(stringn,0);
	if (fretn == fret0) {
		delayMillis(1); // time multiplexed display
	}
	else {
		digitalWrite(fretn,1);
		delayMillis(1);
	}
	return 0; // automatically turns off
}

void displayChord(const int chord[4]) { // intended usage in while loop
	displayNote(string1, chord[0]);
	displayNote(string2, chord[1]);
	displayNote(string3, chord[2]);
	displayNote(string4, chord[3]);
}

void displayTimedChord(const int chord[4], int millis) { // display chord for some
time in milliseconds
	int i;
	for (i=0; i<millis; i++) {
		displayNote(string1, chord[0]);
		displayNote(string2, chord[1]);
		displayNote(string3, chord[2]);
		displayNote(string4, chord[3]);
	}
	clearChord();
```

```
}

void goodJob() {
	displayTimedChord(good1, 6);
	displayTimedChord(good2, 6);
	displayTimedChord(good3, 6);
	displayTimedChord(good4, 6);
	displayTimedChord(good5, 6);
}
void victory() {
	int strings[4] = {string1, string2, string3, string4};
	int frets[5] = {fret1, fret2, fret3, fret4, fret5};
	int i = 0;
	int j, k;
	while (i < 30) {
		for (j=0;j<4;j++) {
			for (k=0;k<5;k++) {
				displayNote(strings[j],frets[k]);
				delayMillis(10);
			}
		}
		i++;
	}
}
//////////////////////////////////////////////
//              Initialization              //
//////////////////////////////////////////////

void setupSpi(){
	int freq = 200000; // set sclk
	int settings = 0; // phase & polarity = 0
	short adc_config = 0b0110100000000000;

	spiInit(freq, settings);
	printf("Setting up SPI...\n");
}

void setupGpio(){
	pioInit();

	pinMode(SPI_READ, INPUT);
	pinMode(SPI_STOP, OUTPUT);
	pinMode(PI_READY, OUTPUT);

	pinMode(string1, OUTPUT);
	pinMode(string2, OUTPUT);
	pinMode(string3, OUTPUT);
	pinMode(string4, OUTPUT);
	pinMode(fret1, OUTPUT);
	pinMode(fret2, OUTPUT);
	pinMode(fret3, OUTPUT);
	pinMode(fret4, OUTPUT);
```

```
	pinMode(fret5, OUTPUT);

	digitalWrite(PI_READY, 1);
	digitalWrite(SPI_STOP, 0);
	delayMillis(1); // for good measure

	printf("Setting up GPIOs... \n");
}

void spiReady(){
	digitalWrite(PI_READY, 1);
	digitalWrite(SPI_STOP, 0);

	// delayMillis(100);

	printf("SPI ready to receive data... \n");
}

//////////////////////////////////////////////
//                    Main                  //
//////////////////////////////////////////////

int main(void)
{
	const char *(*notePtr)[16] = &notes; // to print notes to terminal
	setupGpio();
	setupSpi();
	clearChord();

	int songNumber;
	int songSize; // number of chords in the song
	int** songPtr; // pointer to chord array in header files

	printf("1. friend in me\n");
	printf("2. somewhere over ohe rainbow\n");
	printf("3. mele kalikimaka\n");
	printf("4. i'm yours\n");
	printf("5. the show\n");
	printf("6. banana pancakes (short)\n");
	printf("7. mine (tswift)\n");
	printf("Enter a song number from the list above: ");
	scanf("%d",&songNumber);
	//printf("You entered: %d\n",songNumber);

	if(songNumber == 1) {
		songSize = 31;
		songPtr = &friendInMe;
		printf("now playing: friend in me\n");
	} else if(songNumber == 2) {
		songSize = 62;
		songPtr = &somewhere;
		printf("now playing: somewhere over the rainbow\n");
```

```
	} else if(songNumber == 3) {
		songSize = 31;
		songPtr = &meleKalikimaka;
		printf("now playing: mele kalikimaka\n");
	} else if(songNumber == 4) {
		songSize = 29;
		songPtr = &ImYours;
		printf("now playing: i'm yours\n");
	} else if(songNumber == 5) {
		songSize = 27;
		songPtr = &theShow;
		printf("now playing: the show\n");
	} else if(songNumber == 6) {
		songSize = 8;
		songPtr = &bananaPancakes;
		printf("now playing: banana pancakes (short)\n");
	} else if(songNumber == 7) {
		songSize = 30;
		songPtr = &mine;
		printf("now playing: mine (tswift)\n");
	}
	else {
		printf("invalid input: defaulting to song 1\n");
		songSize = 31;
		songPtr = &friendInMe;
		printf("now playing: friend in me\n");
	}

	printf("\nLet's start!\n\n");
	delayMillis(1000);

	int chordsSoFar=0;
	int fftReceived;
	short *spiArray = calloc(totalSamples, sizeof(short));
	int *fftArray = calloc(totalNotes, sizeof(int)); // max mag in bin range

	///////////////////////////////////////////////////
	//             main loop through song            //
	///////////////////////////////////////////////////
	while (chordsSoFar < songSize) {
		spiReady();
		// displayChord
		//////////////////////////////
		//    encode currentChord   //
		//////////////////////////////
		int encurrentChord[4] = {0,0,0,0};
		int currentChord[4] = {0,0,0,0};
		int j;
		printf("chord: %d\n", chordsSoFar);
		const int* chordAddr = songPtr[chordsSoFar]; // get the address to the
chord array
```

```
		for (j=0; j<4; j++) {
			int chordNote = chordAddr[j]; // dereference to get the fret
			currentChord[j] = chordNote;
			if(j==0){
				if (chordNote == 0) {encurrentChord[0] = 7;} // G
				if (chordNote == 5) {encurrentChord[0] = 8;} // Ab
				if (chordNote == 6) {encurrentChord[0] = 9;} // A
				if (chordNote == 13) {encurrentChord[0] = 10;} // Bb
				if (chordNote == 19) {encurrentChord[0] = 11;} // B
				if (chordNote == 26) {encurrentChord[0] = 12;} // C
			}
			if(j==1){
				if (chordNote == 0) {encurrentChord[1] = 0;} // C
				if (chordNote == 5) {encurrentChord[1] = 1;} // Db
				if (chordNote == 6) {encurrentChord[1] = 2;} // D
				if (chordNote == 13) {encurrentChord[1] = 3;} // Eb
				if (chordNote == 19) {encurrentChord[1] = 4;} // E
				if (chordNote == 26) {encurrentChord[1] = 5;} // F
			}
			if(j==2){
				//switch(chordNote) {
				if (chordNote == 0) {encurrentChord[2] = 4;} // E
				if (chordNote == 5) {encurrentChord[2] = 5;} // F
				if (chordNote == 6) {encurrentChord[2] = 6;} // Gb
				if (chordNote == 13) {encurrentChord[2] = 7;} // G
				if (chordNote == 19) {encurrentChord[2] = 8;} // Ab
				if (chordNote == 26) {encurrentChord[2] = 9;} // A
			}
			if(j==3){
				//switch(chordNote) {
				if (chordNote == 0) {encurrentChord[3] = 9;} // A
				if (chordNote == 5) {encurrentChord[3] = 10;} // Bb
				if (chordNote == 6) {encurrentChord[3] = 11;} // B
				if (chordNote == 13) {encurrentChord[3] = 12;} // C
				if (chordNote == 19) {encurrentChord[3] = 1;} // Db
				if (chordNote == 26) {encurrentChord[3] = 2;} // E
			}
		}

		printf("encurrentChord:  %d %d %d %d\n",
encurrentChord[0],encurrentChord[1],encurrentChord[2],encurrentChord[3]);
		printf("currentChord:  %d %d %d %d\n",
currentChord[0],currentChord[1],currentChord[2],currentChord[3]);

		displayTimedChord(currentChord, 100); // display current target chord
		//////////////////////////////
		//       get spi data       //
		//////////////////////////////

		fftReceived = 0;
		while (!fftReceived) {
			displayChord(currentChord);
```

```
		int spiRead;
		int spiDone;
		unsigned char data1;
		unsigned char data0;
		unsigned short data16;
		int sampleCount = 0;
		spiRead = digitalRead(SPI_READ); // check if fpga is ready to send

		if (spiRead) {
			printf("Receiving FFT results...");
			while (sampleCount < totalSamples) {
				data1 = spiSendReceive(1);
				data0 = spiSendReceive(1);
				data16 = (data1 << 8 | data0);
				spiArray[sampleCount] = data16; // store results

				// printf("%i   %04x \n",sampleCount, data16);
				sampleCount++;
			}
			printf("Finished reading FFT results \n");
			digitalWrite(SPI_STOP, 1); // signal fpga to stop
			digitalWrite(PI_READY, 0); // pull low while chord
comparison
			fftReceived = 1;
		}
	} // end of spi while loop

	//////////////////////////////
	//   process fft results    //
	//////////////////////////////

	int i;

	// downsample FFT to get maxes in ranges we specify
	for (i=0; i<totalNotes; i++){
		int minBin = binRange[0][i];
		int maxBin = binRange[1][i];
		int avgMag = maxVal(spiArray, minBin, maxBin);
		fftArray[i] = avgMag;
		printf("%s   %d   %d   %d \n", (*notePtr)[i], minBin, maxBin,
avgMag);
	}

	// find 4 max of noteBins array
	int maxChord[6] = {0,0,0,0,0,0};
	int noteIndex = 0;
	for(i=0; i<totalMax; i++){
		maxChord[i] = maxIndex(fftArray,16);
		noteIndex = maxChord[i];
		printf("note%d = %d (%s) \n",i, noteIndex,(*notePtr)[noteIndex]);
		fftArray[noteIndex] = 0; // force to 0 so we can find the second
maximum
```

```
        }

        //////////////////////////////
        //      compare chords      //
        //////////////////////////////
        int q;
        int score = 0;  // score = how many notes match
        for (q=0; q<4; q++) {
                if (encurrentChord[q] == maxChord[0]) {score++;}
                else if (encurrentChord[q] == maxChord[1]) {score++;}
                else if (encurrentChord[q] == maxChord[2]) {score++;}
                else if (encurrentChord[q] == maxChord[3]) {score++;}
                else if (encurrentChord[q] == maxChord[4]) {score++;}
                // else if (encurrentChord[q] == maxChord[5]) {score++;}
        }
        printf("score: %d\n\n",score);

        if (score > 3) { // move on if measured chord = target
                goodJob();
                chordsSoFar++; // if correct chord, show next chord
        }

    } // end of main while loop
    printf("You did it!\n");
    victory();
    clearChord();
    free(songPtr);
    free(spiArray);
    free(fftArray);
    return 0; // exit code
}

///////////////////////////////////////////////
//            Finding Max Functions          //
///////////////////////////////////////////////

// Returns the index of the maximum value in an array
    int maxIndex(int *array, int size){
        int i;
        int max=0;
        int index=0;
        for(i=0;i<size;i++){
                if (array[i]>max) {
                        max = array[i];
                        index = i;
                }
        }
        return(index);
    }

// Given a (pointer to) an array of shorts and range [a,b],
// determines the maximum value in that range
```

```
int maxVal(short *array, int a, int b) {
	int i;
	short  max= 0;
	for (i=a; i<=b; i++){
		if (array[i]>max){
			max = array[i];
		}
	}
	return(max);
}

int avgVal(short *array, int a, int b){
	int i;
	unsigned int avg = 0;
	unsigned int val;
	for (i=a; i<=b; i++){
		val = array[i]; // cast
		avg = avg + (int) val;
	}
	avg = avg/abs(b-a);
	return((int) avg);
}
```

Appendix D2: Raspberry Pi C - uku.h

```
//////////////////////////////////////////////
//                  uku.h                  //
//////////////////////////////////////////////

#define string1 4
#define string2 17
#define string3 27
#define string4 22

#define fret1 5
#define fret2 6
#define fret3 13
#define fret4 19
#define fret5 26
#define fret0 0

#define SPI_READ 18
#define SPI_STOP 23
#define PI_READY 25

const int totalSamples = 1024;
const int totalNotes = 16;
const int Fs = 4884; // sampling frequency
const int totalMax = 5;
```

```
const char *notes[] =
{"C4","Db4","D4","Eb4","E4","F4","Gb4","G4","Ab4","A4","Bb4","B4","C5","Db5","D5","Eb5
"};

const int binRange[2][16] = {{52,61,64,68,72,76,81,86,91,96,102,108,114,121,128,141},
// min bins

{60,63,67,71,75,80,85,90,95,101,107,113,120,127,140,150}}; // max bins

const int A[4] = {fret2, fret1, fret0, fret0};
const int Am[4] = {fret2, fret0, fret0, fret0};
const int A7[4] = {fret0, fret1, fret0, fret0};
const int Am7[4] = {fret0, fret0, fret0, fret0};

const int Bb[4] = {fret3, fret2, fret1, fret1};
const int Bbm[4] = {fret3, fret1, fret1, fret1};
const int Bb7[4] = {fret1, fret2, fret1, fret1};
const int Bbm7[4] = {fret1, fret1, fret1, fret1};

const int B[4] = {fret4, fret3, fret2, fret2};
const int Bm[4] = {fret4, fret2, fret2, fret2};
const int B7[4] = {fret2, fret3, fret2, fret2};
const int Bm7[4] = {fret2, fret2, fret2, fret2};

const int C[4] = {fret0, fret0, fret0, fret3};
const int Cm[4] = {fret0, fret3, fret3, fret3};
const int C7[4] = {fret0, fret0, fret0, fret1};
const int Cm7[4] = {fret3, fret3, fret3, fret3};
const int Cmaj7[4] = {fret0, fret0, fret0, fret2};
const int Cadd9[4] = {fret0, fret4, fret3, fret3};

const int Db[4] = {fret1, fret1, fret1, fret4};
const int Dbm[4] = {fret1, fret4, fret4, fret4};
const int Db7[4] = {fret1, fret1, fret1, fret2};
const int Dbm7[4] = {fret1, fret1, fret0, fret2};
const int Dsus2[4] = {fret2,fret2,fret0,fret0};

const int D[4] = {fret2, fret2, fret2, fret0};
const int Dm[4] = {fret2, fret2, fret1, fret0};
const int D7[4] = {fret2, fret2, fret2, fret3};
const int Dm7[4] = {fret2, fret2, fret1, fret3};

const int Eb[4] = {fret0, fret3, fret3, fret1};
const int Ebm[4] = {fret3, fret3, fret2, fret1};
const int Eb7[4] = {fret3, fret3, fret3, fret4};
const int Ebm7[4] = {fret3, fret3, fret2, fret4};

const int E[4] = {fret1, fret4, fret0, fret2};
const int Em[4] = {fret0, fret4, fret3, fret2};
const int E7[4] = {fret1, fret2, fret0, fret2};
const int Em7[4] = {fret0, fret2, fret0, fret2};
```

```
const int F[4] = {fret2, fret0, fret1, fret0};
const int Fm[4] = {fret1, fret0, fret1, fret3};
const int F7[4] = {fret2, fret3, fret1, fret3};
const int Fm7[4] = {fret1, fret3, fret1, fret3};

const int Gb[4] = {fret3, fret1, fret2, fret1};
const int Gbm[4] = {fret2, fret1, fret2, fret0};
const int Gb7[4] = {fret3, fret4, fret2, fret4};
const int Gbm7[4] = {fret2, fret4, fret2, fret4};

const int G[4] = {fret0, fret2, fret3, fret2};
const int Gm[4] = {fret0, fret2, fret3, fret1};
const int G7[4] = {fret0, fret2, fret1, fret2};
const int Gm7[4] = {fret0, fret2, fret1, fret1};

const int Ab[4] = {fret5, fret3, fret4, fret3};
const int Abm[4] = {fret4, fret3, fret4, fret2};
const int Ab7[4] = {fret1, fret3, fret2, fret3};
const int Abm7[4] = {fret1, fret3, fret2, fret2};

const int good1[4] = {fret1, fret1, fret1, fret1};
const int good2[4] = {fret2, fret2, fret2, fret2};
const int good3[4] = {fret3, fret3, fret3, fret3};
const int good4[4] = {fret4, fret4, fret4, fret4};
const int good5[4] = {fret5, fret5, fret5, fret5};


//////////////////////////////////////////////
//          You've Got A Friend In Me         //
//////////////////////////////////////////////

const int* friendInMe[31] = {F,C,F,F7,
                             Bb,Bb,F,F7,
                             Bb,Bb,F,F7,
                             Bb,F,A,Dm,
                             Bb,F,A,Dm,
                             Bb,F,A,Dm,
                             G7,C,F,D,G7,C,F}; // an array of 31 pointers to int arrays


//////////////////////////////////////////////
//          Somewhere Over The Rainbow        //
//////////////////////////////////////////////

const int* somewhere[62] = {C,G,Am,F,C,G,Am,Am,F,F,//intro
             C,C,Em,Em,F,F,C,C,//oooo-ooooo-ooooooo
             F,F,E7,E7,Am,Am,F,F,
             C,C,Em,Em,F,F,C,C,//chorus
             F,F,C,C,G,G,Am,Am,F,F,
             C,C,Em,Em,F,F,C,C,
             F,F,C,C,G,G,Am,Am,F,F};
```

```
//////////////////////////////////////////////
//              Mele Kalikimaka              //
//////////////////////////////////////////////

const int* meleKalikimaka[31] = {C,C,C,C,C,C,//mekalikimaka is the...
             G7,G7,G7,G7,G7,G7,G7,G7, //day... that's the...
             C,C,C7,C7,F,F,A7,A7, //sway.. here we know that...
             D7,D7,C,C,A7,A7,
             Dm,G7,C};

//////////////////////////////////////////////
//          I'm Yours - 4 chord song         //
//////////////////////////////////////////////

const int* ImYours[29] = {C,G,Am,F,
             C,G,Am,F,
             C,G,Am,F,
             C,G,Am,F,
             C,G,Am,F,D7,
             C,G,Am,F,
             C,G,Am,F};


//////////////////////////////////////////////
//          The Show - 4 chord song          //
//////////////////////////////////////////////

const int* theShow[27] = {C,G,Am,F,
             C,G,Am,F,
             C,G,Am,F,
             C,G,Am,F,G,
             C,G,Am,F,
             C,G,Am,F,G,C};


//////////////////////////////////////////////
//           Banana Pancakes Chords          //
//////////////////////////////////////////////

const int* bananaPancakes[8] = {G,D,Am,C7,G,D,Am,C7};


//////////////////////////////////////////////
//               Mine (t-swizzle)             //
//////////////////////////////////////////////

const int* mine[30] = {C,G,D,Em,C,G,D,
             C,G,D,C,G,D,C,G,D,C,G,D,
             C,G,D,Em,Dsus2,C,
             G,D,Em,Dsus2,C};
```