Stroboscopic Instrument Tuner

Final Project Report
December 8th, 2017

E155

Zachary Shattler and Chen Jie Xin

Abstract:

Frequency determination using hardware-implemented N-Point FFT is useful when extracting multiple unknown frequencies from an input signal; however, due to a lack of resolution for the relatively low values of N easily implementable in hardware, precision is often too low for applications such as accurately tuning musical instruments. Thus, a stroboscopic instrument tuner was developed that relies on the principles of sampling and aliasing to detect even minute frequency differences of an input signal from a desired pitch. User input is retrieved, processed by a Raspberry Pi 3b microcontroller, and transferred to the MuddPi Mk. IV FPGA board, which contains logic for determining the difference in frequency between the desired and actual pitches. Deviations are displayed as a shifting pattern on an LED dot matrix display; if the pattern stands still, the note is in tune.

I. Introduction

The goal of this project is to build a stroboscopic instrument tuner for audio frequencies in the range 69.3 Hz (C#2) to 1.0465 kHZ (C6). The project comprises of the following subsystems, listed along with their functions and the hardware used to realize these functions:

1. Audio signal input and processing—take in a pure audio signal from a musical instrument (e.g., a single string played on a violin), filter and convert to a square wave usable by the FPGA board, and generate a pulse train of the same frequency to drive the power pins of the LED dot matrix display.
   Hardware: ADA1063 mic + amplifier breakout board, LM393N comparator, MuddPi Mk. IV FPGA board.
2. LED dot matrix driver—drive an LED dot matrix display to be a periodically shifting pattern at a frequency determined by note being tuned to.
   Hardware: MuddPi Mk. IV FPGA board, two 5x8 bi-colour LED dot matrix displays (Arntd LTP2558AA) and 16 PNP transistors (2N3906).
3. User interface—take user input to determine the note and octave being tuned to, parse into a binary representation, and input to the FPGA.
   Hardware: Raspberry Pi 3B, ADA1115 LCD 16x2 character display.

The block diagram for the project is as shown below:

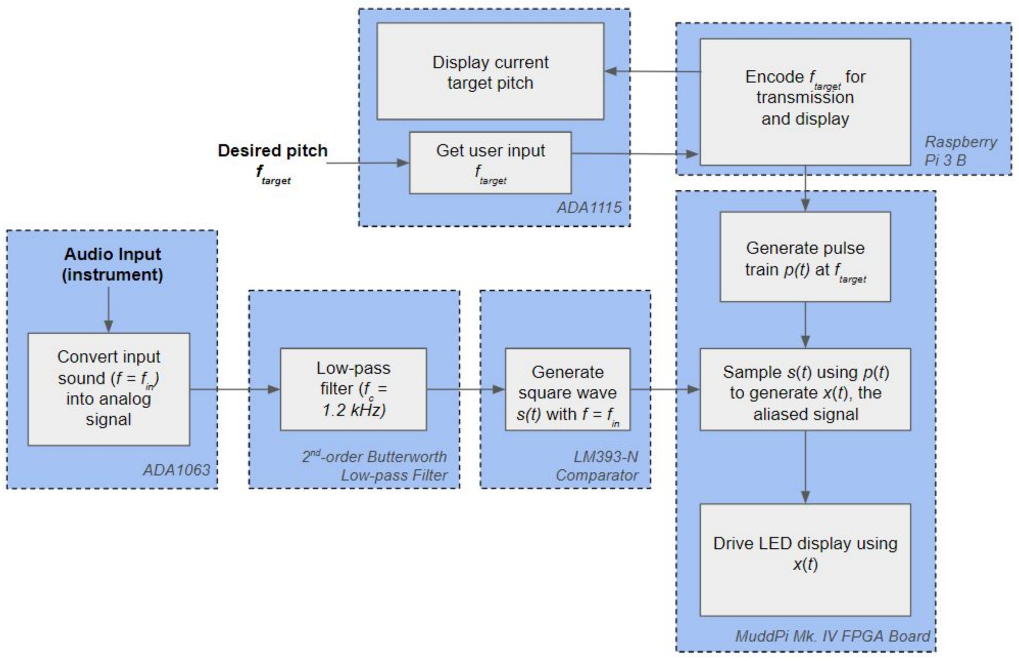

Display current target pitch
Encode f_{target} for transmission and display
Raspberry Pi 3 B
Desired pitch f_{target}
Get user input f_{target}
ADA1115
Audio Input (instrument)
Convert input sound ($f = f_{in}$) into analog signal
ADA1063
Low-pass filter (f_c = 1.2 kHz)
2nd-order Butterworth Low-pass Filter
Generate square wave $s(t)$ with $f = f_{in}$
LM393-N Comparator
Generate pulse train $p(t)$ at f_{target}
Sample $s(t)$ using $p(t)$ to generate $x(t)$, the aliased signal
Drive LED display using $x(t)$
MuddPi Mk. IV FPGA Board

Figure I.a. Block diagram of stroboscopic system tuner, with hardware components designated to perform each task shown in the blue dashed boxes.

Altogether, the system functions as an extremely accurate adjustable stroboscopic tuner. The user interface is such that the current note being tuned to is always displayed on the LCD monitor, and updates continuously as new inputs are provided (i.e., new values of f_{target} are selected). Furthermore, the visualization method used for the LED matrix is such that, for a given audio input frequency f_{in} deviating in pitch from the pitch being tuned to f_{target} by frequency $\Delta f = |f_{\text{in}} - f_{\text{target}}|$, a pattern of four lit LED columns (Fig. I.b.) will shift across the screen every $1/\Delta f$; if Δf is 0, then four-column pattern will remain stationary on the display (although due to the extreme accuracy of the tuner, this will rarely be perfectly attained). Further details on how the system operates can be found in the sections below.

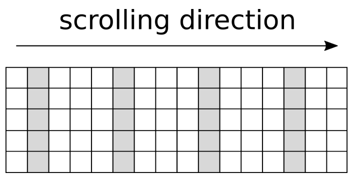


Figure I.b. Cartoon of line pattern scrolling across 5x16 LED dot matrix

II. New Hardware

The main new hardware used in this project are the two 5x8 bi-colour LED dot matrix display, the ADA1063 microphone and amplifier breakout board, and the ADA1115 LCD 16x2 character display and keypad kit. For analog signal pre-processing, we also used an LM393N comparator.

The LED dot matrix displays operate in a manner very similar to the 7-segment displays used in various MicroPs labs throughout the semester, and as such will not be discussed in depth here. The wiring schematic for the LED dot matrix can be found in Sec. III.

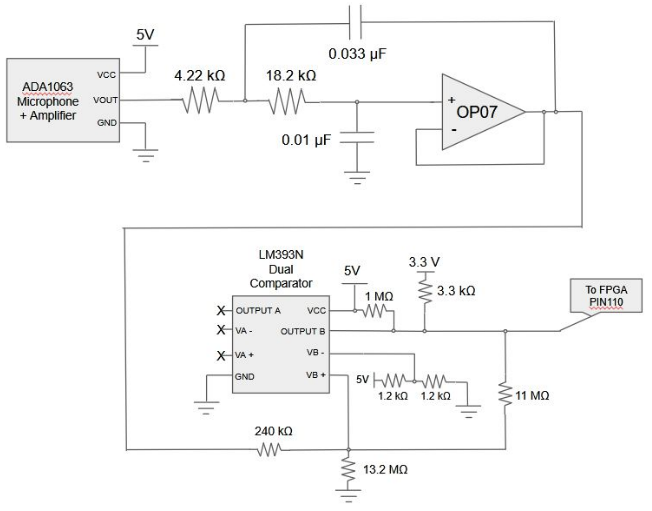
The ADA1063 mic and amplifier board has low power draw and includes an on-board low-pass filter to eliminate high-frequency noise (above 20 kHz), as well as an MAX4466 adjustable-gain amplification IC to provide a low-noise, high-amplitude output signal centered around 2.5 V [1].

The ADA1115 LCD display and keypad kit comes with a PCB designed to allow communication with the Pi over I^2C, as opposed to the relatively complicated procedure for interfacing with the LCD display on its own. In addition, Adafruit offers a Python library with functions for writing characters to the LCD, reading user input from the on-board buttons, and other useful functions for using the display. The PCB can also be used as a Raspberry Pi shield, eliminating the need for the cobbler (this does, however, obstruct access to the majority of the GPIO pins, even though the LCD actually only needs to use 4 pins).

III. Schematics

III.I Audio Pick-up and Analog Pre-Processing

A circuit diagram for all breadboarded components for the microphone and analog pre-processing done on the mircophone signal is shown in Fig. III.a. The signal flow goes from top left to bottom right, and represents the bottom-left three blocks in the block diagram in Section I



.

Figure III.a. Circuit diagram for audio pick-up with microphone, analog filtering and pre-processing

Audio signal processing begins when input audio (e.g., a single violin note being continuously played) with some frequency f_{in} is converted into an analog waveform with DC offset $V_{DC} = 2.5V$ and ranging between 0 and 5V peak to peak and of the same frequency by the ADA1063 chip. 2nd-order Butterworth filter in the Sallen-Key topography, which cutoff frequency f_c = 1 kHz. This cutoff frequency was chosen due to the maximum tunable frequency being set at 1046 Hz; this filter will significantly attenuate any frequencies greater than this, thus eliminating high-frequency noise from the input signal while passing through other frequencies with unity gain.

The filtered signal is then provided as the non-inverting input to an LM393N analog comparator with reference voltage $V_- = 2.5V = V_{DC}$. Nominally, this circuit would output logical high when the input signal has voltage $V_+ > 2.5V$, and logical low when $V_+ < 2.5V$. However, due to small oscillations superimposed on the input signal by noise, this threshold can be crossed many times when the nominal value of the input is 2.5V, and generate a square wave signal that switches between logical high and low very rapidly for a short period of time; this would clearly have a negative impact on later circuit performance.

To combat this, hysteresis can be added to the circuit, which causes the comparator to go logic high when an upper threshold V_{high} is exceeded, and logic low when V_{in} is below a low threshold V_{low}. Given a wide enough range between V_{high} and V_{low}, hysteresis can almost entirely eliminate the quick switching effect mentioned above. In our application, a hysteresis window of 0.1V was chosen, due to the relative instability of the input signal, especially in a noisy environment [2].

Logical high is set by a pullup resistor connecting the comparator output to a 3.3V supply rail, which ensures that the comparator output falls within a voltage range suitable for input to the FPGA. Altogether, this comparator circuit serves to convert the periodic input waveform of frequency f_{in} into a square wave ranging from 0 to 3.3V also of frequency f_{in} with duty cycle 50%. An example model using a sinusoidal input waveform is shown in Fig. II.b.

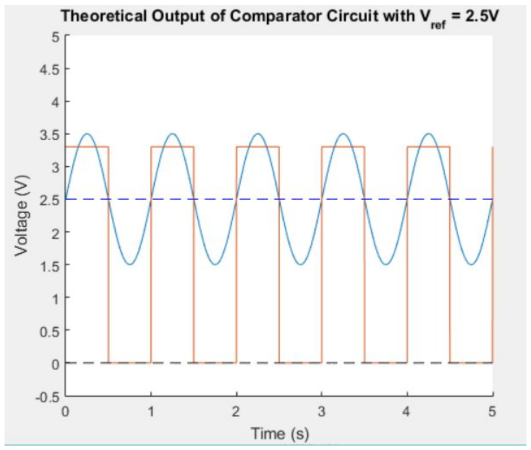


Figure II.b. Theoretical output of comparator circuit as described in Section II.

III.II Raspberry Pi and LCD Display Circuit

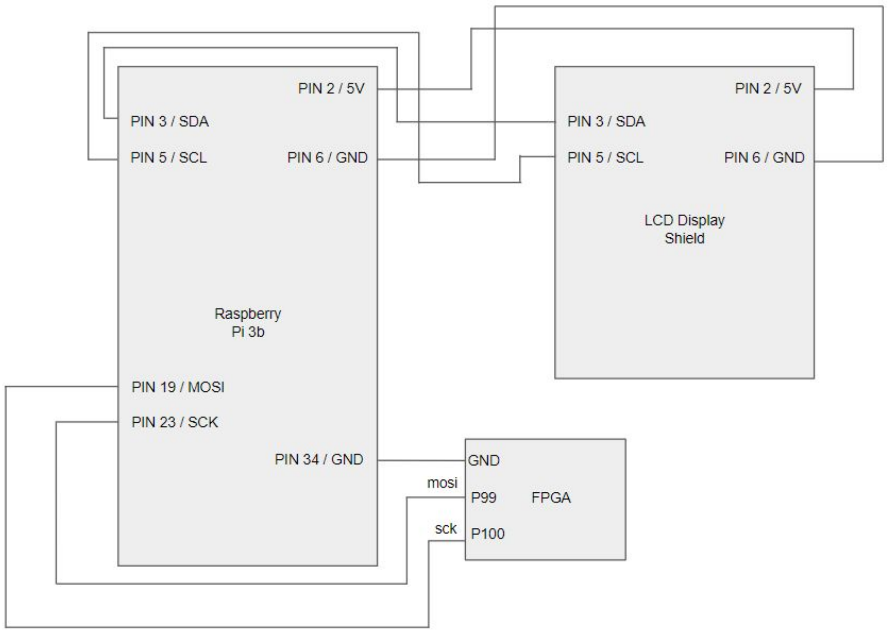


Figure III.c. Circuit diagram for the wiring of the Raspberry Pi to the LCD display and FPGA

Connections between the Raspberry Pi and the LCD display and FPGA consist entirely of straightforward serial buses and power lines (Fig. III.c):

The Pi communicates with the LCD using the I^2C protocol, and as such dedicated I^2C bus pins 3 (SDA) and 5 (SCK) on the Pi were connected to the relevant pins on the LCD display board; pin numbers on the LCD board schematic are not well-labeled in available documentation, but can be correlated directly with the Pi pinout due to the board being a Pi shield. Furthermore, power was provided to the LCD board from Pi pins 2 (5V) and 6 (GND).

The Pi and the FPGA communicated using one-way master-to-slave SPI communication with the Pi acting as the master, and as such dedicated SPI bus pins 19 (MOSI) and 23 (SCLK) were connected to FPGA input pins 100 and 99, respectively. In addition, to ensure that voltage levels were consistent across the SPI bus, the Pi and FPGA grounds were connected.

III.III LED Dot Matrix Circuit

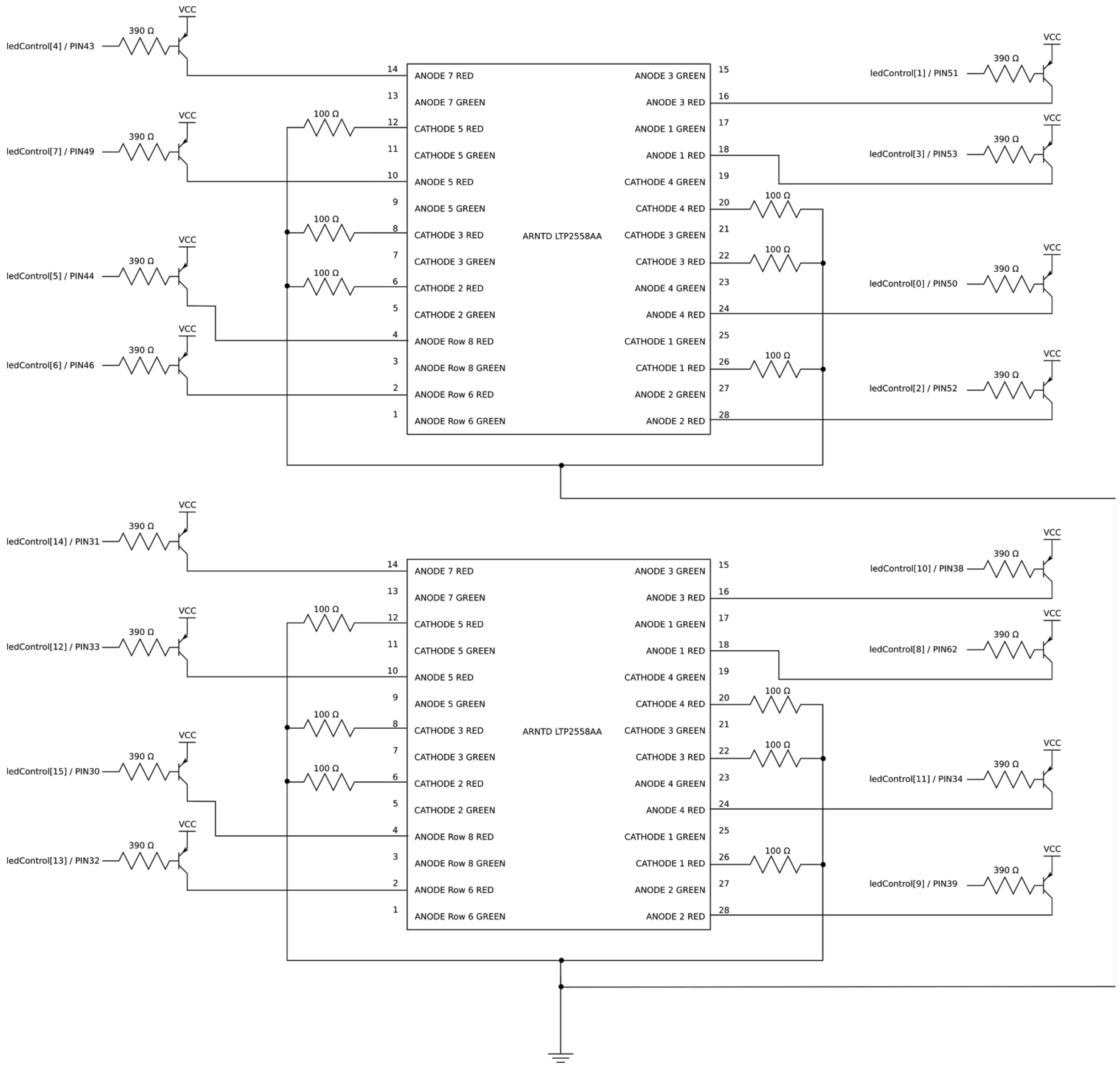


Figure III.d. Circuit diagram of the wiring of the LED dot matrix to the FPGA (numbered pins are FPGA pins)

The 5x8 LED dot matrix displays uses X-Y select to address the 40 dots on each display, with the two different LED colours (green and red-orange) being controlled by two disjoint sets of pins . For each LED colour—say, the red-orange LEDs—there are five cathode pins and eight anode pins. Each cathode pin is the cathode of the eight red-orange LEDs in a particular row[1] of the matrix and each anode pin corresponds to the anode of the the five red-orange LEDs in

[1] What we refer to as "rows" here are referred to as "columns" in the Arntd LTP2558AA datasheet (and vice versa), as the datasheet uses a rotated coordinate axis. In this report we use our coordinate convention.

particular column of the matrix. As such, in order to turn on an LED in a particular row and column, we will need to pull the cathode for that row low and the anode for that column high. For our purposes, we only use one of the LED colours (red-orange), so we only needed to wire half of the pins on each 5x8 LED dot matrix. Moreover, since all five LEDs in a particular column are either all on or all off, all ten cathodes (five for each 5x8 LED dot matrix) can be connected to the same node, which is grounded.

To select which column of LEDs turn on, we need to wire the 16 red-green LED anode pins on the two LED dot matrices to 16 output pins on the FPGA. When all five LEDs in a row are on, the average forward curent would be 65 mA and the peak forward current would be 500 mA, so we need a PNP transistor (2N3906) between each FPGA output and anode in order to supply the current. Pulling the FPGA pin which is connected to the base of the transistor low (high) turns on (off) the corresponding row of clumns. This configuration allows us to select the columns on the LED dot matrix to turn on such that we can scroll a line pattern across the LED dot matrix. Fig. III.d. shows the complete circuit diagram for the wiring of the LED dot matrix.

IV. Microcontroller Design

The Raspberry Pi 3b was used as the microcontroller in this project, and was primarily employed as a method for accepting, displaying, and transmitting user input. As discussed in Section II, the Pi uses I2C to communicate with LCD display/button plate; furthermore, an extensive Python library made available by Adafruit simplifies this communication by handling memory-mapping for I2C communication and providing functions such as `write(message)`, `clear_LCD()`, and `button.isPressed()` [4]. In addition to the LCD board, the Pi communicates with the FPGA via a MOSI-only SPI connection, with the Pi acting as SPI master, as discussed above. This communication was also simplified through the use of the `spi_dev` Python module, which handles memory-mapping for SPI communication and provides functions that allow the one-way transfer of an 8-bit value when called. With these methods established, the general algorithm for retrieving user input and sending it to the FPGA is as follows.

- Upon initialization of the program, open the SPI (with clock speed 122 kHz, CPOL = CPHA = 0) and I2C ports, generate an array of possible note and octave values, and default to tuning to A4; transmit this to the Pi over SPI. Display this value on the LCD, as well as instructions for changing the tuning pitch.
- Next, using A4 as a base case, begin querying the button panel for user input. This involves iterating over an array consisting of the five buttons on the board and checking whether their `.isPressed()` attribute is true.
- If a button is detected as being pressed, then wait 100 ms until the button is no longer being pressed, as each press should be registered only once. Then, either increment or decrement the octave or note being tuned to, or return the current note and octave,

depending on the button pressed. (If not returning, print the updated pitch to the LCD and wait for the next button press.

- Once the current note is returned, encode the note/octave pair as an 8-bit binary number, and send it to the FPGA via SPI.
- Repeat until the program is manually interrupted on the Pi (by using Ctrl-C) or the system is powered down.

This allows for continuous flow of input from the user interface to the FPGA logic using the Pi as a middleman for receival, encoding and transmission.

V. FPGA Design

There are two main parts to the hardware implemented on the FPGA: the first part receives and parses the user-selected target pitch sent to the FPGA over SPI from the Raspberry Pi, and the second part uses the user input and input signal to generate a control signal for to drive the LCD dot matrix display:

V.I. Receiving and Parsing User Input

The FPGA in this case is the SPI slave while the Raspberry Pi is the SPI master, with data transfer only occurring from master to slave, so the block for receiving SPI data is relatively straightforward. Following in the design in Ref. [5], the `spiSlaveReceiveOnly` module takes in two inputs, `sck` and `mosi`, and outputs the byte received from the Raspberry Pi as an 8-bit bus, which is updated by a shift register that shifts `mosi` into the least significant bit on the positive edge of `sck.`

The byte received from the Raspberry Pi through SPI, `spiOut`, is formatted in the form `00_xxxx_yy`, where the four bits after the leading zeros encode the user selected note, `nextNote`, and the least significant two bits encode the user selected octave, `nextOctave`; the note and octave encodings for each note in our tuning range is provided in Appendix C. Since `nextNote` and `nextOctave` are clocked to `sck`, we pass them through a register clocked to the positive edge of the 40 MHz FPGA clock in order to synchronize them with the other signals used in the LCD dot matrix display driver.

V.II. LCD Dot Matrix Display Driver

The output of the synchroniser of the user-selected note and octave, `currentNote` and `currentOctave` are used to as control-signals to a multiplexer whose output, `maxCount`, is the number of 40 MHz FPGA clock cycles in one period of the target pitch, i.e. $T_{\text{target}} = 1/f_{\text{target}}$.

`maxCount` is then used as the input to the `resettingCounter` module, which contains a counter that resets and emits a pulse every time the counter reaches `maxCount - 1`, i.e. every time the counter has gone through `maxCount` cycles of the 40 MHz FPGA clock . As such, the output of the `resettingCounter` module, `registerEn`, is an discrete impulse train at the frequency of the target pitch, f_{target}, with the width of each pulse, w, being one period of the FPGA clock.

Like its name suggests, `registerEn`, is used as the enable pin of a register, whose input, `nextInput`, is the analog signal output from the audio pick-up circuit (Sec. III) and whose output, `curretInput`, is the sampled signal.

Since `registerEn`, is a pulse train with frequency f_{target} and pulse width $w = 1/(40\text{ MHz})$, we are effectively sampling the analog signal with a sample frequency $f_s = f_{\text{target}}$. As such when we are near target pitch, i.e. the frequency of the analog signal, f_{in} is between $f_s/2$ and $3f_s/2$, aliasing occurs, and the frequency of the sampled signal is the absolute value of the difference between the analog signal and the sampling frequency, i.e.

$$f_{\text{aliased}} = |f_{\text{in}} - f_{\text{s}}| = |f_{\text{in}} - f_{\text{target}}| = \Delta f\ .$$

f_{aliased} is used as the clock of the module that generates the LED dot matrix control signals. `ledControl`. The output of `ledControl`, is just a 16-bit bus, each of which is used to toggle the turn on/off a PNP transistor (2N3904) that drives the anodes of each column in the LED dot matrix. 16-bits are the output of a circular shift register, which is clocked to f_{aliased}, so the pattern shifts every $1/f_{\text{aliased}}$.

For input frequencies in the vicinity of f_{target}, i.e. $f_{\text{s}}/2 < f_{\text{in}} < 3f_{\text{s}}/2$, $f_{\text{aliased}} = \Delta f$, so the pattern will shift every $1/\Delta f$. For instance if f_{in} is 2 Hz out of tune compared to the target pitch, i.e $\Delta f = 2$ Hz, the pattern shifts every 1/(2 Hz) = 0.5 s. Similarly, if $f_{\text{in}} = f_{\text{target}}$ such that $\Delta f = 0$, $1/\Delta f$ becomes infinity, which means the pattern becomes stationary. So, the pattern scrolling across the LED dot matrix is stationary when note played is in tune, and scrolls faster the more out of tune it is in the range $f_{\text{s}}/2 < f_{\text{in}} < 3f_{\text{s}}/2$.

VI. Results

We successfully implemented a stroboscopic instrument tuner with the specifications laid out in Sec. I and the initial project proposal. Due to the high sensitivity of the tuner to small changes in frequency, as well as the difficulties of playing a long sustained note with uniform frequency on an instrument, it is quite difficult to get the pattern on the tuner to stay absolutely still for a long period of time. However, when the note played is in tune, the pattern does stay stationary for a time long enough for the user to distinguish it from the out of tune state. When tested with the same input signal against a phone tuner application, our stroboscopic instrument tuner has comparable performance in its ability to distinguishing in/out of tune notes.

Running the user interface involves running the `getUI()` function in `ui.py` with the following commands:

```
$ sudo python
>>> from ui import *
>>> getUI()
```

Upon start-up, the program defaults to tuning to A440 (or A_4). The Left/Right arrow keys are used to change the note to tune to, while the Up/Down arrow keys are used to change the octave that the note is in. The Select key is used to select the target note to tune to.

Prior to live testing with the actual analog input from the microphone, we tested the signal processing on the FPGA with square wave frequencies generated with a function generator. With the function generator signal, the pattern remained stationary for a very long period of time when the function generator is set to the correct frequency. The pattern begins to move even for very small deviations, such as 0.5 Hz from the target frequency.

During live-testing, we used several different types of signals: notes from a phone tone generator application, notes played from a violin, notes played from a classical guitar, and whistled notes. In all cases, the microphone needed to be very close to the source in order to maximize signal, and on hindsight, perhaps a piezo pickup instead of a regular microphone may have suited our purposes better. In any case, for all the signals tried, the display remained stationary for a reasonably long period of time when the note played was in tune, and scrolls faster as the note played is more out of tune.

One extra feature that would have been useful if implemented would be an indicator that showed whether the out of tune note was sharp or flat. As is, since the display scrolling rate is determined by the absolute value of the frequency difference, a note that is 2 Hz sharper and 2 Hz flatter than the target pitch result in the same pattern. Currently, the way to distinguish which way the note is out of tune is to arbitrarily pick a direction to tune at first, and determine whether the pattern slower or faster to determine whether it is approaching or moving further away from the target pitch, respectively.

Finally, we note that when no audio signal is being picked up, e.g. in a quiet room, the pattern also remains stationary. However, as the tuner is clearly not in use when sitting in a quiet room, this was not considered to be a serious issue with the implementation. Altogether, the our implementation of a stroboscopic tuner has been quite successful.

VII. References

[1] Adafruit, “Electret Microphone Amplifier - MAX4466 with Adjustable Gain,” https://www.adafruit.com/product/1063. Accessed 6 December, 2017.

[2] Maxim Integrated, “Adding Extra Hysteresis to Comparators,” https://www.maximintegrated.com/en/app-notes/index.mvp/id/3616. Accessed 6 December, 2017.

[3] Lite-On Electronics, “Arntd LTP2558AA Datasheet,” https://www.jameco.com/Jameco/Products/ProdDS/2186048.pdf. Accessed 6 Decmber, 2017.

[4] Adafruit, “Overview | Adafruit 16x2 LCD Plus Keypad for Raspberry Pi,” https://learn.adafruit.com/adafruit-16x2-character-lcd-plus-keypad-for-raspberry-pi. Accessed 6 December, 2017.

[5] D. Harris and S. Harris, “I/O Systems,” in *Digital Design and Computer Architecture: ARM Edition* (2015).

VIII. Parts List

Part No.	**Part Description**	**Quantity**	**Price per unit (USD)**
Adafruit 1115	Blue & White 16x2 LCD + Keypad Kit for Raspberry Pi	1x	19.95
Adafruit 1063	Electret Microphone Amplifier - MAX4466 with Adjustable Gain	1x	6.95
Arntd LTP2558AA	2.3” 5x8 Bi-Color LED Dot Matrix Display - Red Orange/Green	2x	1.49
Total			**29.98 + shipping**

Appendix A: Raspberry Pi Code

```
################################################################################
# ui.py
#
#     Written by: Chen Jie Xin, Zachary Shattler
#     Contact:    cxin@g.hmc.edu, zshattler@g.hmc.edu
#     Date created: 25 November 2017
#     Date updated: 3 December 2017
#
# Functions for strobe tuner UI.
#     Created for FA2017, ENGR 155 final project.
#
# Usage:
#     $ sudo python
#     >>> from ui import *
#     >>> getUI()
################################################################################
import time
import sys
import Adafruit_CharLCD as LCD
import spidev

notes = ["C", "B", "A#/Bb", "A", "G#/Ab", "G", "F#/Gb", "F", "E", "D#/Eb", "D", "C#/Db"]
octaves = [5, 4, 3, 2]
buttons = {(LCD.SELECT, 1),
           (LCD.UP, 2),
           (LCD.DOWN, 3),
           (LCD.LEFT, 4),
           (LCD.RIGHT, 5)}

lcd = LCD.Adafruit_CharLCDPlate()

# Open SPI port
spi = spidev.SpiDev()
spi.open(0,0)

def getUI():
         # Set SPI mode to 0 and SCK to 122 kHZ and msbfirst mode
         spi.mode = 0
         spi.max_speed_hz = 122000
         spi.lsbfirst = False

         # Default to A440 (A3) on power up
         oldNote = 3
         oldOctave = 1

         # Encode note as a char
         toWrite = (oldNote << 2) | oldOctave

         # Send to LCD Display
         spi.xfer2([toWrite])

         # Main loop
         while True:
                  print "oldNote = " + str(oldNote) + ", oldOctave = " + str(oldOctave)

                  oldNote, oldOctave = getLCDInput(oldNote, oldOctave)
                  toWrite = (oldNote << 2) | oldOctave
                  print "Value sent: " + bin(toWrite)

                  spi.xfer2([toWrite])
                  print "SPI written!"

# Function for checking LCD for user input
```

```
def getLCDInput(prevNote, prevOctave):
        printNewNote(prevNote, prevOctave)

        noteIndex = prevNote
        octIndex = prevOctave
        while True:
                for button in buttons:
                        if lcd.is_pressed(button[0]):
                                while (lcd.is_pressed(button[0])):
                                        time.sleep(0.1)
                                if(button[1] == 1):
                                        return noteIndex, octIndex

                                elif(button[1] == 2):
                                        octIndex = (octIndex - 1)%4
                                elif(button[1] == 3):
                                        octIndex = (octIndex + 1)%4
                                elif(button[1] == 4):
                                        noteIndex = (noteIndex + 1)%12
                                elif(button[1] == 5):
                                        noteIndex = (noteIndex - 1)%12
                                printNewNote(noteIndex, octIndex)

# Function for updating note on LCD display
def printNewNote(note, octave):
        lcd.clear()
        if(note == 0):
                oct = octaves[octave] + 1
        else:
                oct = octaves[octave]

        lcd.message("Note: " + notes[note] + str(oct) + "\n^v +-8va  <> b/#")
```

Appendix B: SystemVerilog Code

```
/******************************************************************************
 * strobeTunerMain.sv
 *
 *    Written by:   Chen Jie Xin, Zachary Shattler
 *    Contact:      cxin@g.hmc.edu , zshattler@g.hmc.edu
 *    Date created: 25 November 2017
 *    Date updated: 3 December 2017
 *
 * Main for module for running a strobe tuner on an FPGA.
 *   Created for FA2017, ENGR 155 final project.
 *****************************************************************************/
module strobeTunerMain(
  input logic clk,                     // 40 MHz reference clock
  input logic reset,                   // System reset
  input logic sck,                     // SPI serial clock
  input logic mosi,                    // SPI MOSI
  input logic nextInput,               // Next sample of input frequency
  output logic refPitch,               // Reference pitch for debugging
  output logic ledClk ,                // Clock for LED dot matrix for debugging
  output logic [15:0] ledControlPins,  // LED control pins
  output logic [5:0] onBoardLED);      // Display current octave and pitch on onboard LEDs

  logic registerEn, spiDone, currentInput;

  logic [3:0] currentNote;
  logic [3:0] nextNote;

  logic [1:0] currentOctave;
  logic [1:0] nextOctave;

  logic [7:0] spiOut;
  logic [19:0] maxCount;

  // SPI receive next note and octave from Pi
  spiSlaveReceiveOnly ssro(sck, mosi, spiOut);

  // Update note and octave
  always_ff @(posedge clk)
    {currentNote, currentOctave} <= {nextNote, nextOctave};

  // Update input tone sample every time counter resets
  always_ff @(posedge clk, posedge reset)
    if (reset)           currentInput <= 0;
    else if (registerEn) currentInput <= nextInput;

  // Module for selecting maxCount based on currentNote and currentOctave
  maxCountSelect mcs(currentNote, currentOctave, maxCount);

  // Counter that emits a pulse for one 40 MHz period every maxCount clock cycles
  resettingCounter rc(clk, reset, maxCount, registerEn);

  // LED control module for scrolling pattern
  ledControl ledc(ledClk, reset, ledControlPins);

  // Pattern scrolling is clocked to positive edge of input samples
  assign ledClk = currentInput;

  // Debugging signals
  assign refPitch = registerEn;
  assign nextNote = spiOut[5:2];
  assign nextOctave = spiOut[1:0];
  assign onBoardLED = {currentNote, currentOctave};
endmodule
```

```
/******************************************************************************
 * spiSlaveReceiveOnly.sv
 *
 *    Adapted from: Digital Design and Computer Architecture, ARM Edition
 *    Written by:   Chen Jie Xin, Zachary Shattler
 *    Contact:      cxin@g.hmc.edu , zshattler@g.hmc.edu
 *    Date created: 25 Novemember 2017
 *    Date updated: 3 December 2017
 *
 * Module for using the FPGA as an SPI slave in receive only mode.
 *****************************************************************************/

module spiSlaveReceiveOnly
  (input logic sck,        // From master
   input logic mosi,       // From master
   output logic [7:0] q);    // Data received in format 00_<note>_<octave>


  // Shift in next bit on positive edge of sck and increment count by 1
  always_ff @(posedge sck)
      q <= {q[6:0], mosi};

endmodule

/******************************************************************************
 * maxCountSelect.sv
 *
 *    Written by:   Chen Jie Xin, Zachary Shattler
 *    Contact:      cxin@g.hmc.edu , zshattler@g.hmc.edu
 *    Date created: 27 November 2017
 *    Date updated: 27 November 2017
 *
 * Module for selecting maximum count based on input octave and note.
 *   Created for FA2017, ENGR 155 final project.
 *****************************************************************************/

module maxCountSelect(
  input logic [3:0] note,
  input logic [1:0] octave,
  output logic [19:0] maxCount);

logic [19:0] maxCountBase;

  always_comb
    begin
      case(note)
        4'b0000: maxCountBase = 20'h0_954f;  // C
        4'b0001: maxCountBase = 20'h0_9e2f;  // B
        4'b0010: maxCountBase = 20'h0_a797;  // A#
        4'b0011: maxCountBase = 20'h0_b18f;  // A
        4'b0100: maxCountBase = 20'h0_bc1d;  // G#
        4'b0101: maxCountBase = 20'h0_c74d;  // G
        4'b0110: maxCountBase = 20'h0_d327;  // F#
        4'b0111: maxCountBase = 20'h0_dfb5;  // F
        4'b1000: maxCountBase = 20'h0_ed03;  // E
        4'b1001: maxCountBase = 20'h0_fb1b;  // D#
        4'b1010: maxCountBase = 20'h1_0a09;  // D
        4'b1011: maxCountBase = 20'h1_19da;  // C#
        default: maxCountBase = 20'h0_0010;  // Default for simulation
    endcase
  end

  // Left shift to get period of the same note, but n octaves lower
  assign maxCount = maxCountBase << octave;

endmodule
```

```
/*******************************************************************************
 * resettingCounter.sv
 *
 *    Written by:   Chen Jie Xin, Zachary Shattler
 *    Contact:      cxin@g.hmc.edu , zshattler@g.hmc.edu
 *    Date created: 27 November 2017
 *    Date updated: 27 November 2017
 *
 * Counter that emits a pulse for one clock cycle and resets every maxCount
 *   clock cycles.
 *   Created for FA2017, ENGR 155 final project.
 ******************************************************************************/

module resettingCounter
  (input logic clk,
   input logic masterReset,
   input logic [19:0] maxCount,
   output logic pulse);

   logic counterReset;
   logic [19:0] currentCount;

   // Counter register with asynchronous master reset and synchronous counterReset
   always_ff @(posedge clk, posedge masterReset)
     if (masterReset)       currentCount <= 20'b0;
     else if (counterReset) currentCount <= 20'b0;
     else                   currentCount <= currentCount + 20'b1;

   // Counter starts from zero, so counterResets when count is maxCount - 1
   assign counterReset = (currentCount == maxCount - 20'b1);
   assign pulse = counterReset;

endmodule

/*******************************************************************************
 * ledControl.sv
 *
 *    Written by:   Chen Jie Xin, Zachary Shattler
 *    Contact:      cxin@g.hmc.edu , zshattler@g.hmc.edu
 *    Date created: 25 November 2017
 *    Date updated: 25 November 2017
 *
 * Module for scrolling a pattern across a 5x16 LED dot matrix.
 *   Created for FA2017, ENGR 155 final project.
 ******************************************************************************/

module ledControl(
  input logic clk,
  input logic reset,
  output logic [15:0] ledControl);

  always_ff @(posedge clk, posedge reset)
    if (reset) ledControl <= ~(16'b0010_0010_0010_0010);
    else       ledControl <= {ledControl[0], ledControl[15:1]};

endmodule
```

Appendix C: List of Notes and Frequencies

Note	Freq (Hz)	note [3:0]	octave [2:0]	maxCount [20:0]
C#2/Db2	69.30	1011	11	8CEB1
D2	73.42	1010	11	8502B
D#2/Eb2	77.78	1001	11	7D8DF
E2	82.41	1000	11	76802
F2	87.31	0111	11	6FD9A
F#2/Gb2	92.50	0110	11	69930
G2	98.00	0101	11	63A63
G#2/Ab2	103.83	0100	11	5E0DD
A2	110.00	0011	11	58C74
A#2/Bb2	116.54	0010	11	53CBE
B2	123.47	0001	11	4F17D
C3	130.81	0000	11	4AA7B
C#3/Db3	138.59	1011	10	4676D
D3	146.83	1010	10	42828
D#3/Eb3	155.56	1001	10	3EC70
E3	164.81	1000	10	3B410
F3	174.61	0111	10	37EDA
F#3/Gb3	185.00	0110	10	34C98
G3	196.00	0101	10	31D32
G#3/Ab3	207.65	0100	10	2F078
A3	220.00	0011	10	2C63A
A#3/Bb3	233.08	0010	10	29E5F
B3	246.94	0001	10	278BF
C4	261.63	0000	10	25538

C#4/Db4	277.18	1011	01	233B7
D4	293.66	1010	01	21414
D#4/Eb4	311.13	1001	01	1F634
E4	329.63	1000	01	1DA04
F4	349.23	0111	01	1BF6A
F#4/Gb4	369.99	0110	01	1A64F
G4	392.00	0101	01	18E99
G#4/Ab4	415.30	0100	01	1783C
A4	440.00	0011	01	1631D
A#4/Bb4	466.16	0010	01	14F2F
B4	493.88	0001	01	13C5F
C5	523.25	0000	01	12A9D
C#5/Db5	554.37	1011	00	119DA
D5	587.33	1010	00	10A09
D#5/Eb5	622.25	1001	00	0FB1B
E5	659.25	1000	00	0ED03
F5	698.46	0111	00	0DFB5
F#5/Gb5	739.99	0110	00	0D327
G5	783.99	0101	00	0C74D
G#5/Ab5	830.61	0100	00	0BC1D
A5	880.00	0011	00	0B18F
A#5/Bb5	932.33	0010	00	0A797
B5	987.77	0001	00	09E2F
C6	1046.50	0000	00	0954F