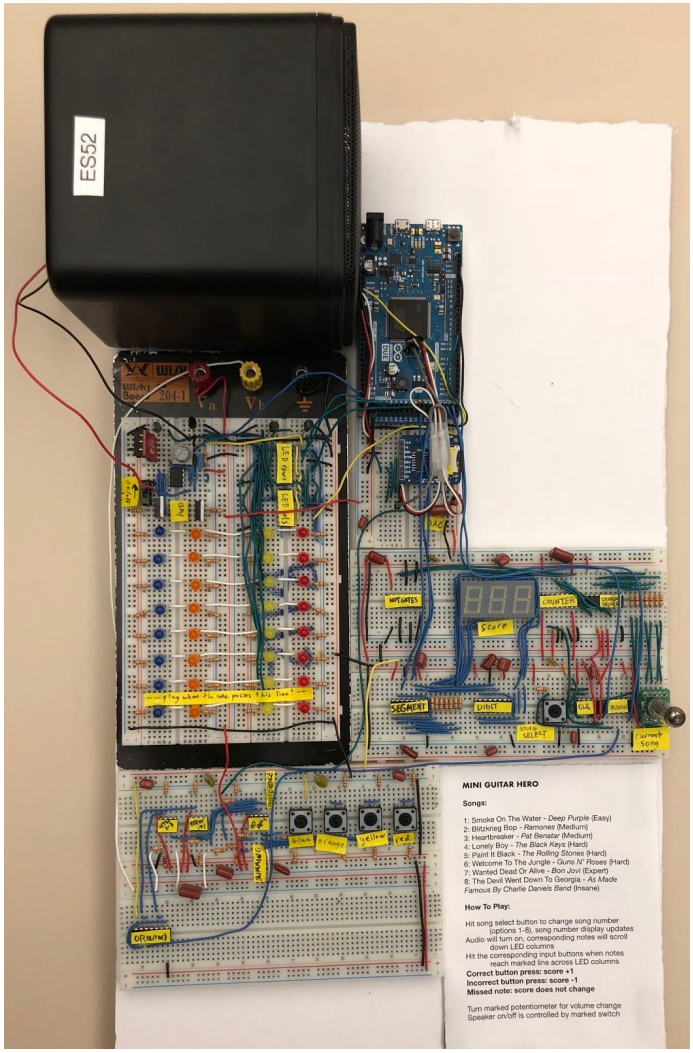
ES52 Final Project:
Miniature Guitar Hero

Andrew Shackelford and Vivian Qiang



Abstract:

Using digital and analog circuit design components, this circuit functions as a miniaturized Guitar Hero game. It has a playlist of 8 songs that the user can choose from a song selection input. While the song plays from the speaker, scrolling columns of LEDs are lit up, not unlike the display of the actual Guitar Hero game. Users can play by pressing corresponding note buttons when the lit LEDs hit the bottom of the column, and are shown their score using the score display included within the circuitry. Through continual testing and demonstration, this circuit has been shown to function properly and as expected, providing a fun gameplay experience for users.

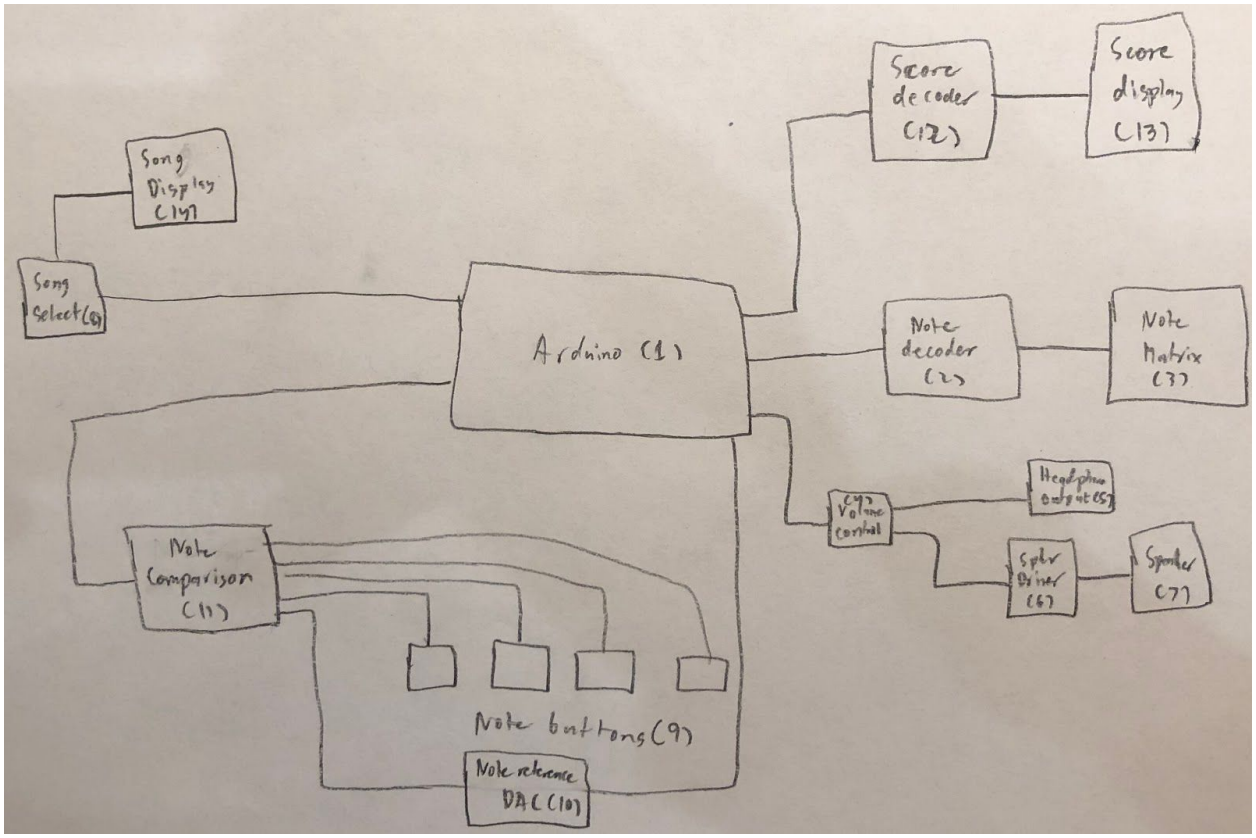
Introduction:

With the increasing popularity of gaming on screens, first with TV consoles and later with phones, people are losing the charm of physically interactive gaming. To address this problem, while at the same time appealing to those who like to keep their games portable (much like their smartphones) and taking inspiration from a sure cult classic, we aimed to design a miniature Guitar Hero game on a breadboard.

The game is played by pressing four buttons on the board. These four buttons ("note buttons") correspond to four columns of LEDs ("the note display"), which light up as notes travel down the columns. When the corresponding column's bottom LED lights up, the player presses the button to hit that note. The game also plays a song along with these notes, giving the player auditory cues. This results in a easy to learn, but hard to master game that engages the player visually, auditorily, and physically.

To create the miniature Guitar Hero appropriate for a breadboard, we brainstormed and finalized the units we would need to build, and set about designing each unit separately and testing after each build to make sure it worked. The final design, described below, incorporated an Arduino Due as the central unit, processing feedback and giving output to various systems such as note checking, a score display, and a note display.

Systems Breakdown:



All core elements of our project interacted with the Arduino (1), which acted as the central brain of our circuit.

The Arduino sent digital signals to both the score display decoder (12) and the note display decoder (2). These decoders controlled their respective displays, an 8-row, 4-column LED matrix (7) and a 3-digit 7-segment display (13). The Arduino also sent an analog audio signal using its built-in DAC to the volume control (4) for the headphone output (5) and speaker driver (6) and output (7).

The Arduino sent a signal to a PWM DAC (10) that controlled the note checking circuit (11), which determined if the user pressed the correct button(s) (9) at the right time. This signal was then sent back into the Arduino to update the score. In addition, the song select button (10) controlled the current song display (14), and also sent a signal to the Arduino instructing it to change to the next song (1).

Our project is powered with a bench power supply, at +9V and -9V. The Arduino is powered off the +9V and supplies +5V and +3.3V through its voltage regulators.

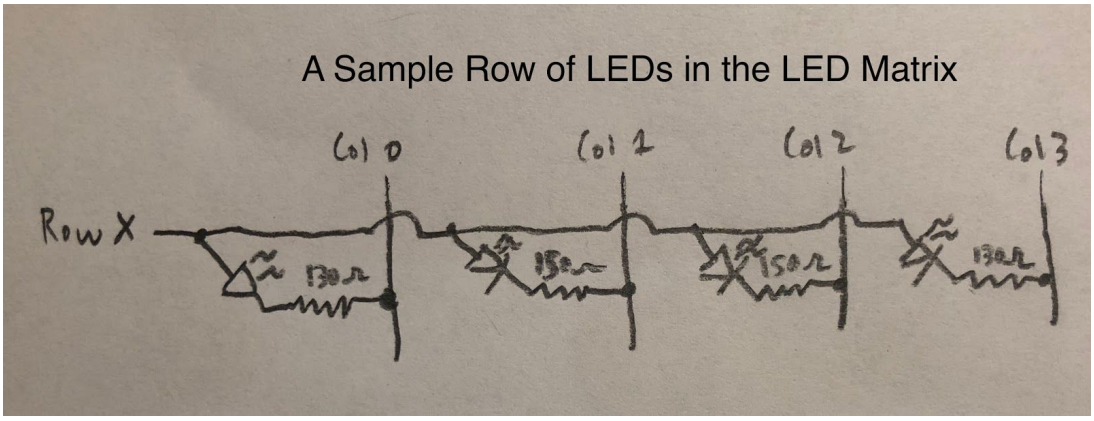
Subsystems:

SD Card:

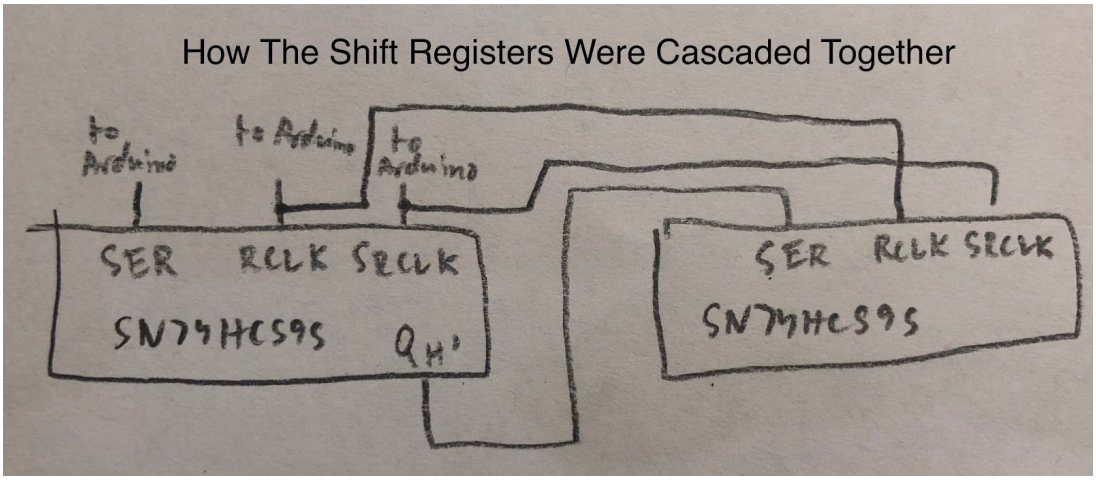
To read in the note and song files, we connected the Arduino to a Adafruit 254 SD Card Reader, which stores all the necessary files for the songs and notes. We connected this reader to the Arduino using the built-in SPI header on the Arduino, and powered the reader with the built-in +5V voltage regulator on the Arduino.

Note Decoder/Display:

To build the note display, first we had to perform research on how to build a matrix display. A matrix display operates by multiplexing -- driving each column of the display one at a time, but at a rate fast enough that the human eye cannot notice. While each row is connected to the same source, only one column is connected to ground at a time so only one column will light up. For example, to light up the LEDs on Row C and Row H on Column 2, we would set Row C and Row H to be high and Column 2 to be low (leaving Columns 0, 1, and 3 set high). If we cycle through each column very quickly, we can light up each LED individually without any noticeable flickering. Since we have 8 rows and 4 columns, as well as easy access to 8-bit shift registers in the lab, we decided to use one 8-bit shift register to control the rows and one to control the columns.

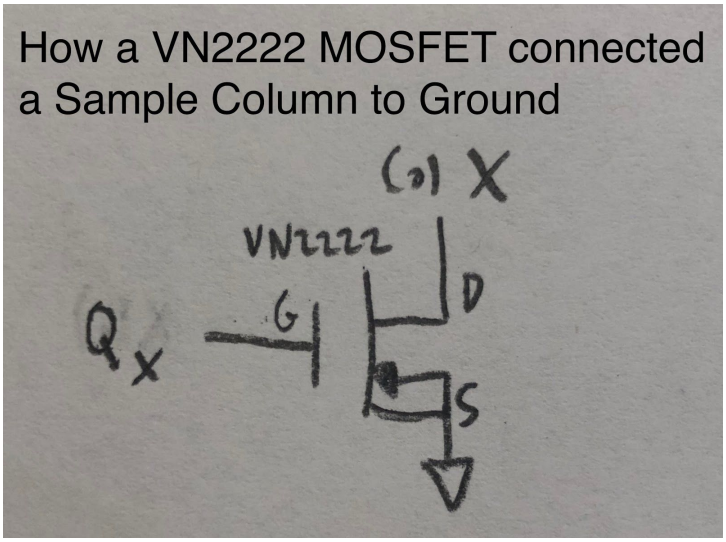


Once we determined that we needed to use two shift registers, it was a simple matter of wiring the U9 outputs Q_A - Q_H to Rows A - H and U10 outputs Q_A - Q_D to Columns 0 - 3. We connected the shift registers together by connecting U9's $Q_{H'}$ to U10's SER input and having the clock (SRCLK) and load (RCLK) signals shared by both shift registers.



By carefully testing the LED types individually (as they came with no documentation other than approximate forward voltage), we determined that the LEDs stopped getting brighter at a current of 5 mA. Given their respective forward voltages, we calculated that the blue and red LEDs needed current limiting resistors of 130 ohms (9-3) / 0.05 = 120 ~ 130 Ω, and the yellow and orange LEDs needed 150 ohms (9-2) / 0.05 = 140 ~ 150 Ω. Upon testing, we found that 5 mA was in fact uncomfortably bright, but found that changing the input voltage from 9 V to 3.3 V made the LEDs operate at a comfortable brightness.

Looking at the datasheet for the SN74HC595 shift register, we found that it could support a maximum of 6 mA per pin, so while each row could be driven by the shift register itself, we would need to use MOSFETs for each column. We wired each column to the drain of a MOSFET whose gate was controlled by a shift register and source was tied to ground, so that when the gate went HIGH, the column was connected to ground and the LEDs could sink current.



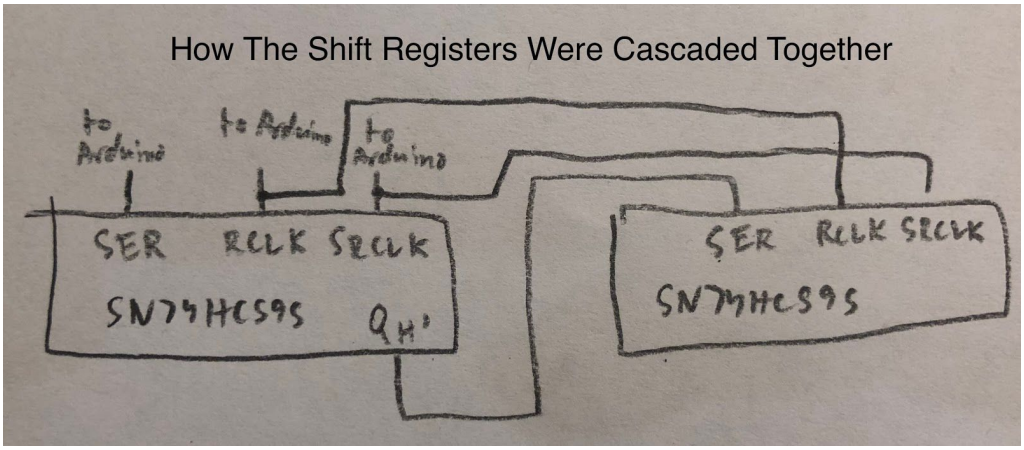
Once the circuit was built and confirmed to be operational by testing each LED individually with a command from the Arduino, we tested the multiplexing at different rates and found 1000 Hz to be a good compromise between how often the Arduino needed to update the display and showing any noticeable flickering effect.

We did run into an issue trying to get the Arduino to multiplex the display at a reliable 1000 Hz. We found that this rate dropped drastically when we tried to play audio from the Arduino at the same time. We solved this by using timer interrupts that forced the Arduino to update the display every millisecond regardless of what code was currently running, which ensured that the display looked good and there was no visible flickering.

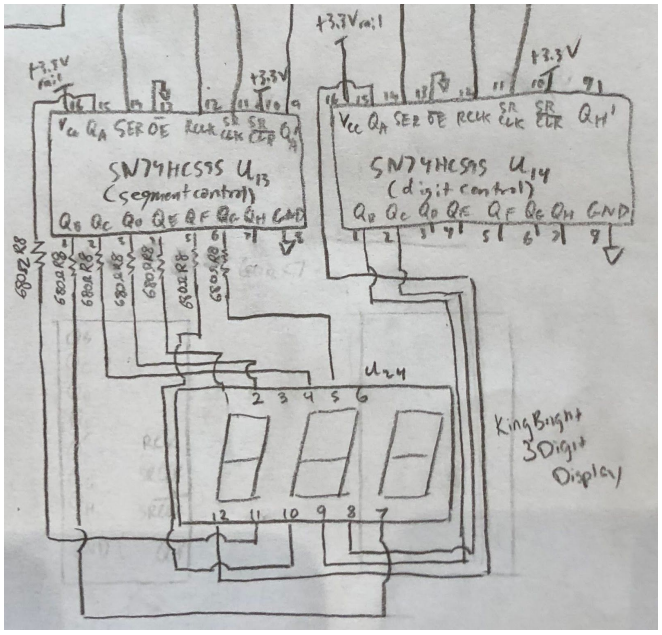
Serial data output data (yellow), load (blue), and clock (pink)

Score Decoder/Display:

Like the note display, this 3-digit 7-segment display also had to be multiplexed. Thanks to our previous work with the note display, we already knew how to connect a multiplexed display and drive it in software. We again used two 8-bit shift registers, one for each segment of the number, and one for each digit.



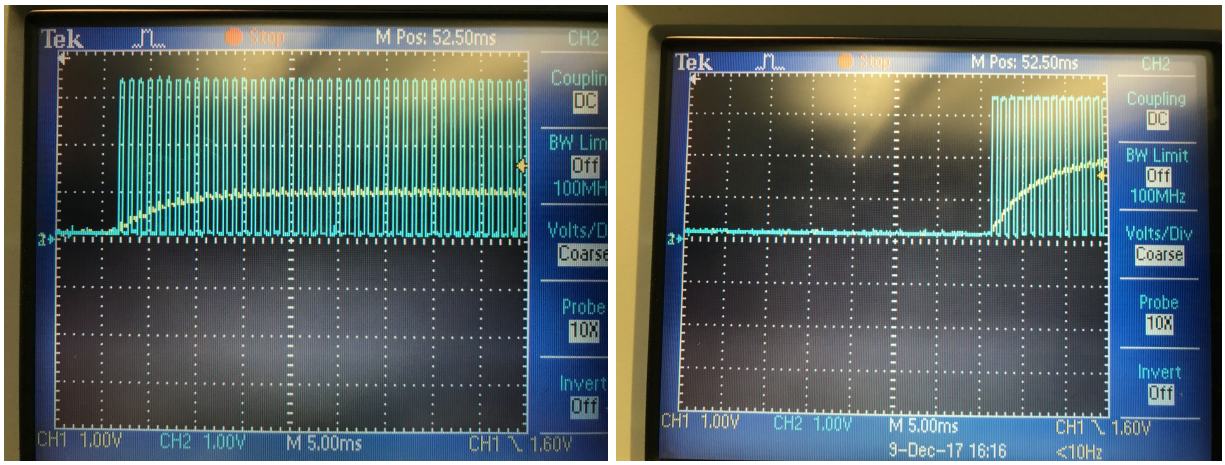
However, while the LED matrix sank current from row to column, the score display sank current from digit to segment. This required us changing up the wiring somewhat, with the digit active high and the segment active low. We also found that while the max forward current from the KingBright display datasheet was 30mA, we only needed 5mA current at our desired brightness, so we did not need to use MOSFETs to sink the current out of each segment. Our current limiting resistor (R8), placed on each segment, was 680 Ohms, and this gave us about 4.5mA of current.



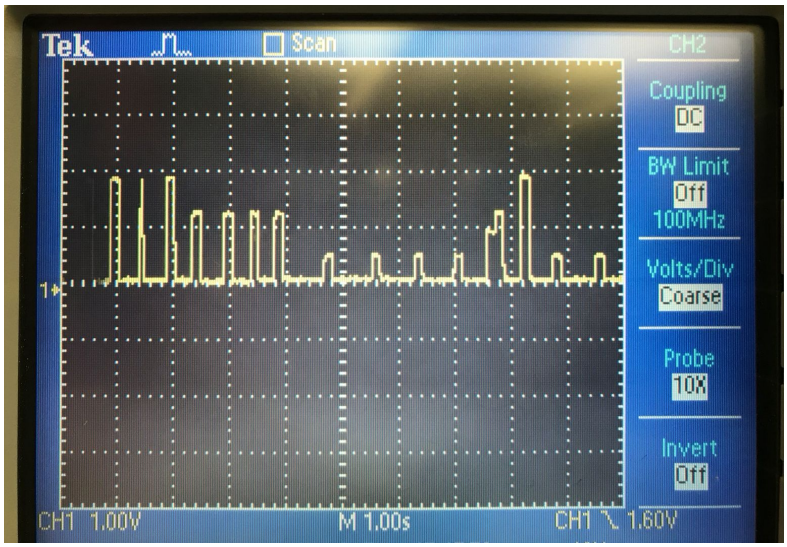
Score display wiring

Note Reference DAC:

To provide a note reference signal, we needed to convert the Arduino's PWM signal to an analog reference voltage. Since the Arduino's PWM carrier frequency was 1000 Hz, we used a passive low-pass filter (R6, C2) with a f_c of 34 Hz that provided a settling time less than 25 ms (the time between a note reference update and the next note check) but minimized ripple. We did this with a resistor value (R6) of 4.7KΩ and a capacitor value (C2) of 1 µF. This signal was then fed into the note checking circuit that determined if the note presses were correct.



Arduino PWM DAC input (blue) and output (yellow) (testing)



Arduino note reference output (playing a song)

Volume control:

We used a potentiometer (U8) connected to the Arduino's DAC output (which ranges from 0 to 3.3V) and GND to create an adjustable voltage divider that we used as a volume control.

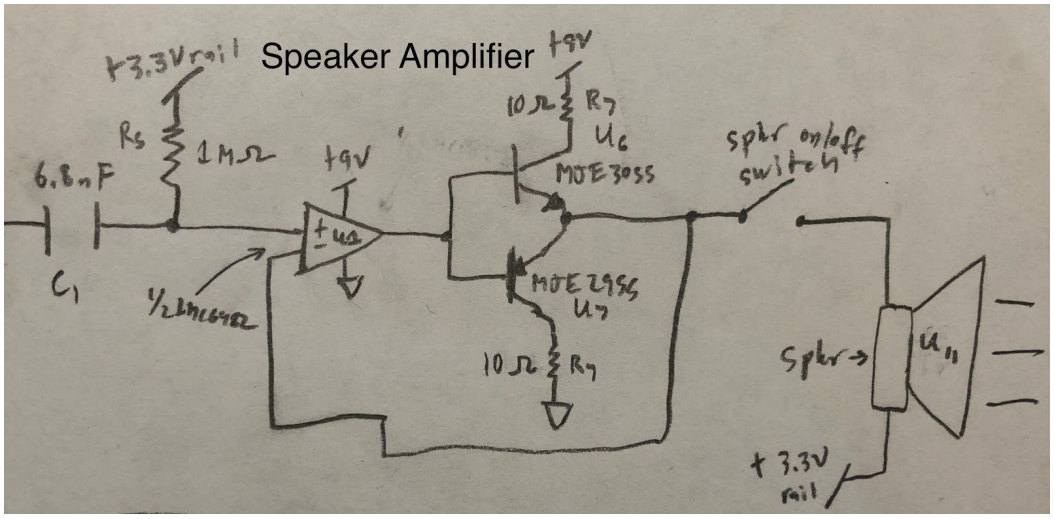
Headphone jack:

After sending the volume-controlled signal through a buffer (U1), we connected the output to the Tip (L audio) and Ring 1 (R audio) of a TRRS headphone jack (U12). We then connected Ring 2 (GND) to ground, and left the Sleeve (Mic) disconnected.

Speaker:

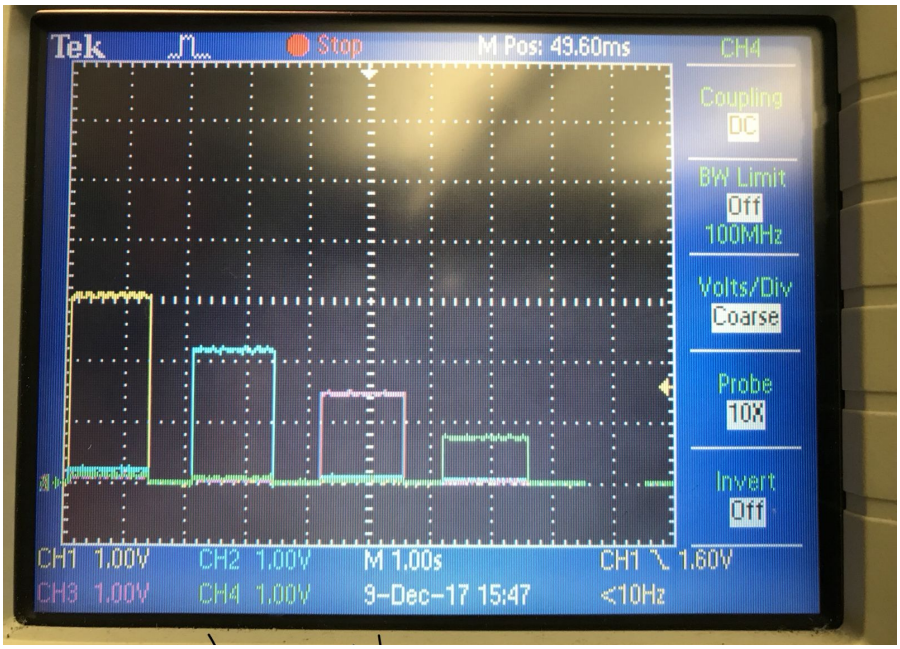
We connected the output of the volume control (U8) to a low-pass filter (C1, R5) that shifted the signal's DC offset to 3.3V, allowing us to use a pair of BJTs (U6, U7) as a push-pull amplifier for a speaker. If we did not offset the signal, the collector-to-emitter voltage drop would not allow us to properly amplify any input voltages below 0.6V. We used a filter with a f_c of 23.4 Hz (f_p = 46.8 Hz), ensuring that no audible frequencies (> 60Hz) are attenuated. To do this, we used a resistor (R5) value of 1 MΩ and a capacitor value (C1) of 6.8 nF.

We amplified the DC-offset signal using a push-pull amplifier similar to the one we used in Lab 3. We connected the signal to the + input of an op amp (U1) whose output was connected a pair of BJTs (U6, U7) set up as a push-pull amplifier, with the output of the amplifier connected to the - input of the op amp (U1) in order to eliminate crossover distortion. We used fuse resistors (R7) of 10Ω to ensure we didn't send too much current through the amplifier. This output was then connected to the speaker (U11) using a switch in order to allow us to turn the speaker off.



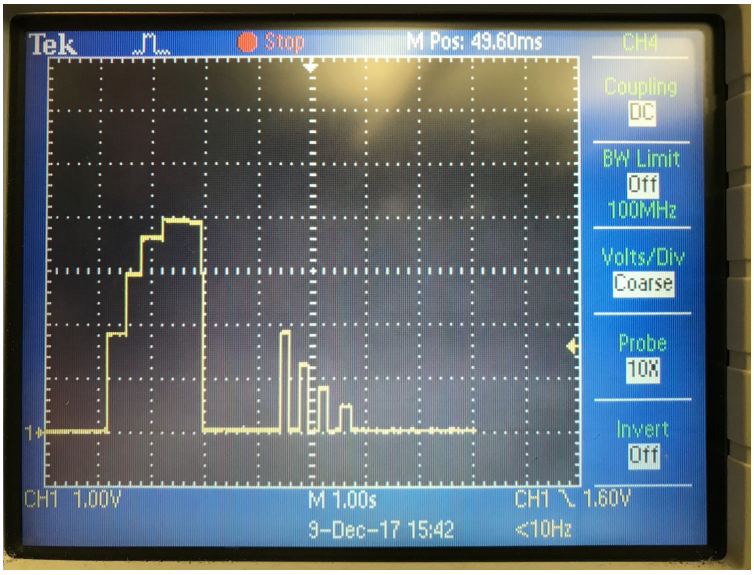
Note Checking:

To check if the user pressed the correct notes, we connected each button to a unique voltage reference specific to that button's note.



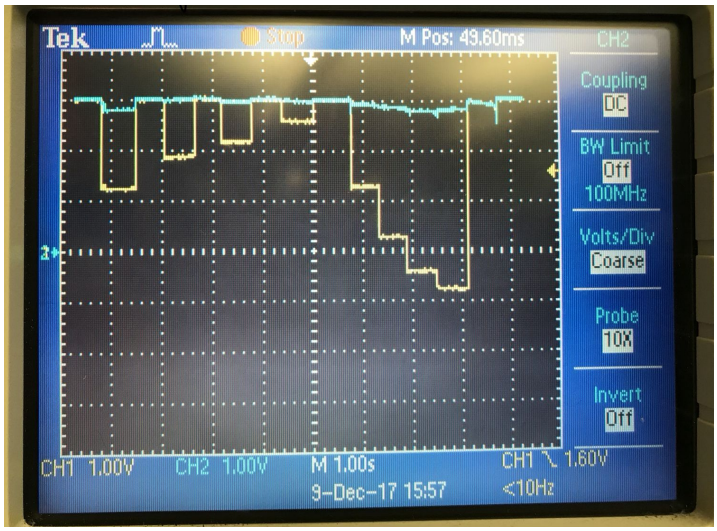
Direct output of each note button

We then summed the outputs of each button together using a summing op amp (U15), allowing us to determine which button(s) were pressed by analyzing one specific voltage.



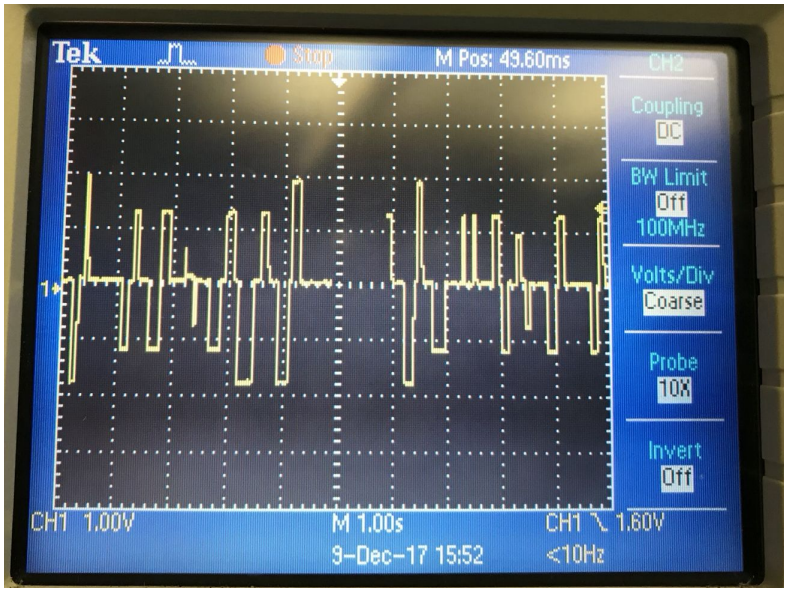
Summing op amp output

We took this voltage and subtracted the correct reference voltage provided by the Arduino using a differential amplifier (U15). This oscilloscope graph shows the output of the differential amplifier when the summed input voltages are subtracted from the reference voltage used for testing. As expected, every time a button is pressed, voltage dips below the blue reference line.

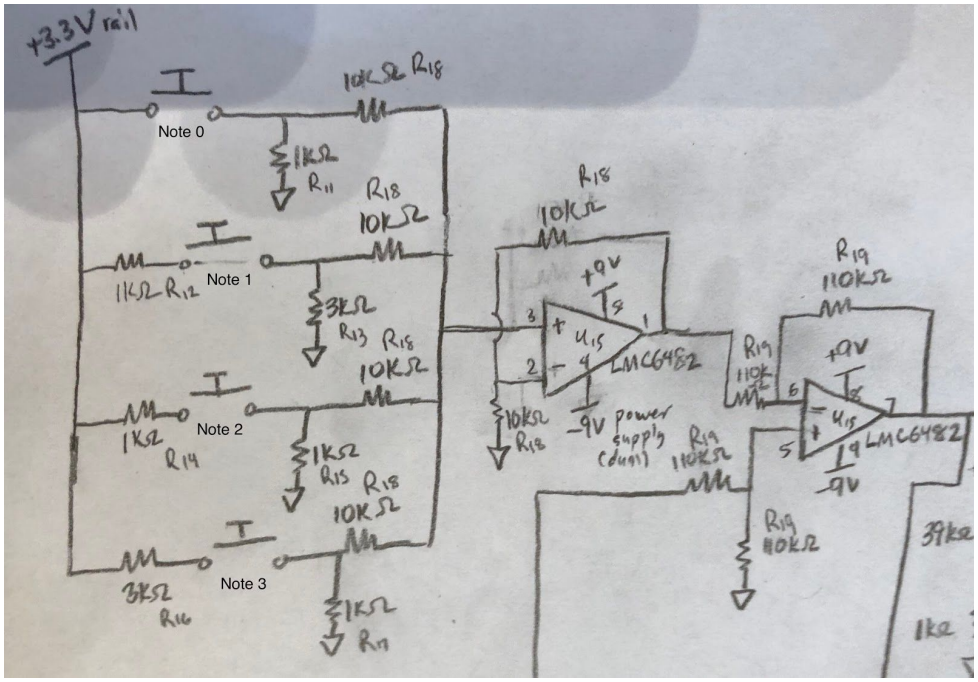


Differential op amp output (testing, simplified)

This oscilloscope graph shows the difference of the Arduino's note reference output and the output of the summing op-amp (total voltage of notes pressed). If these two inputs to the differential op-amp are the same, then they subtract to 0 V. If the input buttons are pressed, for instance, when no note should be pressed, output of the differential op-amp goes negative, as shown in the picture.



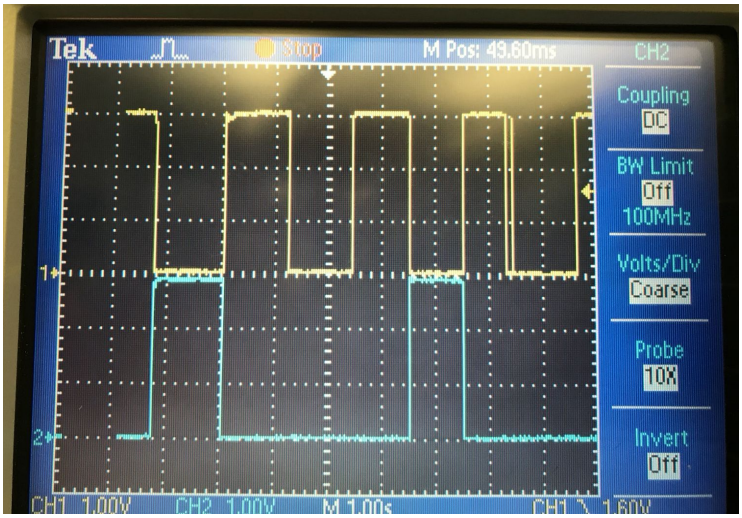
Differential op amp output (playing song)



Note buttons, summing op amp, and differential amplifier

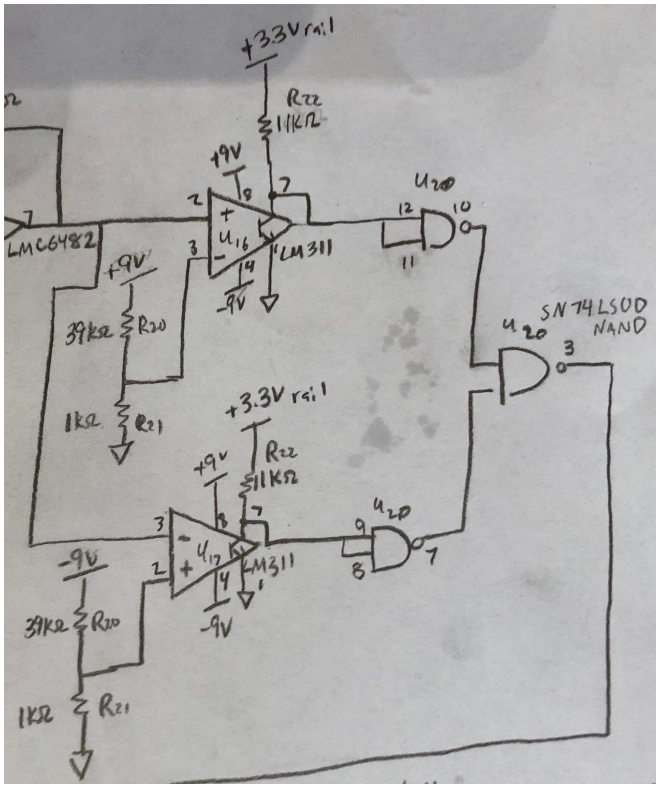
If the button(s) pressed are correct, this voltage should be approximately 0 V. To check this, we connected the voltage to two comparators, one that compared it to a positive error threshold of 0.225 V (U16), and one that compared it to a negative error threshold of -0.225 V (U17). We wired up both of these comparators to be active low (i.e. if the voltage is less than the positive error threshold, the output is low, and if the voltage is greater than the negative error threshold, the output is low).

In this oscilloscope graph, yellow corresponds with the output of the comparator that measures positive error margin, and blue corresponds with the output of the comparator that measures negative error margin. A “high” output on either indicates error (larger than margin). For testing, the comparators are taking the output of the differential op-amp that with a 3V voltage reference. The first time yellow falls, buttons 1, 2, and 3 were pressed (4.5V total). The second time yellow falls, buttons 1 and 3 were pressed (3V total). Accordingly, the blue is high when there is a negative difference of 1.5V while yellow is low, yellow if high and blue is low when no button is pressed and the “difference” is 3V, and both yellow and blue are low when the buttons pressed are “correct” and the difference is 0V.

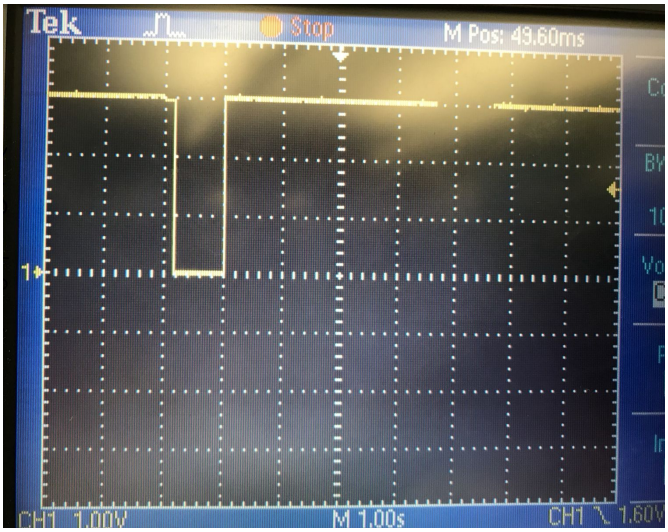


Comparator outputs

We used an OR gate (U20) to combine the output of the two comparators, outputting a high signal when either error threshold is not satisfied, and outputting a low signal only when both error thresholds are satisfied. Lastly, we connected the output of this OR gate to a flip-flop (U18) with a 1000 Hz clock in order to debounce the output.



Comparators and OR gate



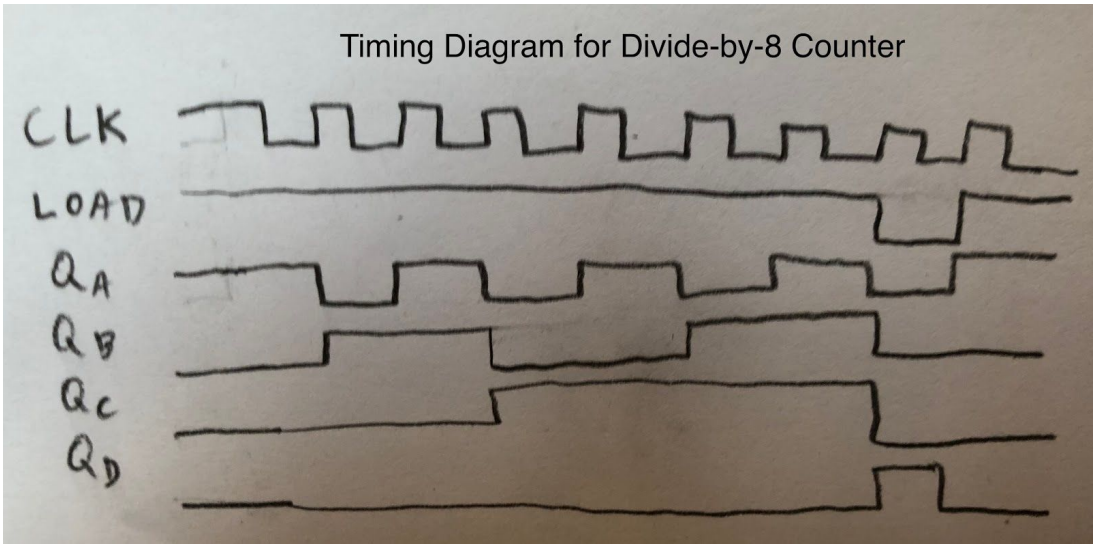
Debounced output of OR gate (testing)

The output of this flip-flop (U18) was then sent to the Arduino, which registered the interrupt and checked to see whether the note hit was correct. For more information on how the Arduino did this, see the **Arduino Code** section below.

Song Select / Display:

To allow the user to change songs, we provided a “Song Select” button that cycled through every available song. The output of this button was connected to a flip-flop (U18) with a 1000 Hz clock in order to debounce the output. The debounced button press was then connected to the clock input of a divide-by-8 counter (U21). The debounced button press was also sent to the Arduino, which registered the interrupt and immediately changed the song. The outputs of this counter were connected to a display driver (U22) that drove a Numitron display (U23) that showed the current song.

This divide-by-8 counter allowed us to cycle through Songs 1-8. We did this by using a 74HC161 counter. We connected load input A high, and load inputs B-D low, which meant that the counter would load in the number 1 when its LOAD input was pulled low. To control LOAD, we connected the Q_D output of the counter through a NOT gate to LOAD. Although Q_D goes high at 8, or 0b1000, LOAD is asynchronous, so the counter does not load in 1 until it receives another clock signal (i.e. the button is pressed again). This gives us the desired operation of a cycle from 1 through 8.



Arduino Code:

The Arduino serves six main functions:

1) Read in new note data for the LED matrix
2) Send out a reference voltage for the note checking circuit
3) Check whether a note was a hit
4) Read and send out audio to the speaker/headphone jack
5) Send out data to the LED matrix to display upcoming notes
6) Send out data to the score display so the user knows how well they are playing

The first four of these functions are accomplished through the main `loop()` method, and the last two are accomplished using timed interrupts that multiplex their respective displays.

Crucial to understanding how the Arduino code works is understanding how the notes are stored. Each “note file” is simply a text file with sequences of four bytes. Each set of four bytes corresponds to the four columns of the LED matrix, with each byte’s MSB representing the bottom LED and LSB representing the top LED. If the corresponding bit is `1`, the LED should be on, if `0`, the LED should be off.

These files were generated manually by Python scripts that recorded keypresses in sync with music. For more detail (as these don’t pertain to the circuit itself), see the attached Python scripts in the Appendix. Each set of 4 bytes has been generated at 25 ms time intervals, giving us a 40 Hz refresh rate of the LED matrix, well above the cinema standard of 24 Hz.

The `setup()` method sets the appropriate pins to input and output, attaches ISRs for the song change and note check methods (to be explained later), initializes the SD card, and sets the multiplexing functions to be called by a timer interrupt at 1000 Hz. These multiplexing functions will be explained later. It then loads in the first song and starts playing.

The `loop()` method in the Arduino code starts by checking the two ISR flags.

The first flag, `ShouldCheckSong`, calls the `changeSong()` method. This interrupt is triggered when the user presses the "Song Select" button. It closes the old song and note files and loads in the new song and note files. It also skips the first 7 frames of note data, which is necessary to get the notes and song in sync.

The second flag, `ShouldCheckNoteHit`, calls the `checkNoteHit()` method. This interrupt is triggered when the output of the note checking circuit changes. It looks at the output of the note checking circuit, and does one of the following:

a) if the button(s) pressed are correct:
    i) if there is a note that should be played, award the user a point
    ii) if there is no note that should be played, do nothing
b) if the button(s) pressed are incorrect:
    i) if there is or is about to be a note that should be played, do nothing
    ii) if there is no note that should be played, take a point away from the user

The `loop()` method then checks to see whether a new frame of note data should be loaded. If so, it calls the `updateNoteData()` method to read in a new set of 4 bytes, and calls the `updateNoteReference()` method to update the PWM note reference output.

Then, if more audio data is available, it reads in the next kilobyte of audio data to a buffer and uses the Audio library to output that buffer over the Arduino's built-in DAC.

The last two tasks of the Arduino, multiplexing the note and score display, are accomplished using the `sendData()` function that is called every millisecond by a timed interrupt. Using a global variable, it sends out each column of the note display and each digit of the score display sequentially, leaving the score display completely off for one millisecond for simplicity's sake so the displays can be multiplexed together with the same timer.

For the full code files, see the Appendix.

Results:

While it is difficult to quantitatively measure the success of a game, the oscilloscope graphs shown above prove that our note checking circuitry and serial data output function correctly. Our display is multiplexed at the correct 1000 Hz rate thanks to the timed interrupts, and the speaker output never stuttered or stopped functioning. In addition, anecdotally, our project had the most visitors at the demo, a definitive quantitative measure of success.

Qualitatively, the game was fun to play, the songs sounded good, and every player seemed to enjoy the experience. The note display never flickered or stopped working, and provided a smooth gameplay experience. The score display also did not flicker, and correctly incremented the score each time the player hit a note.

Design Narrative:

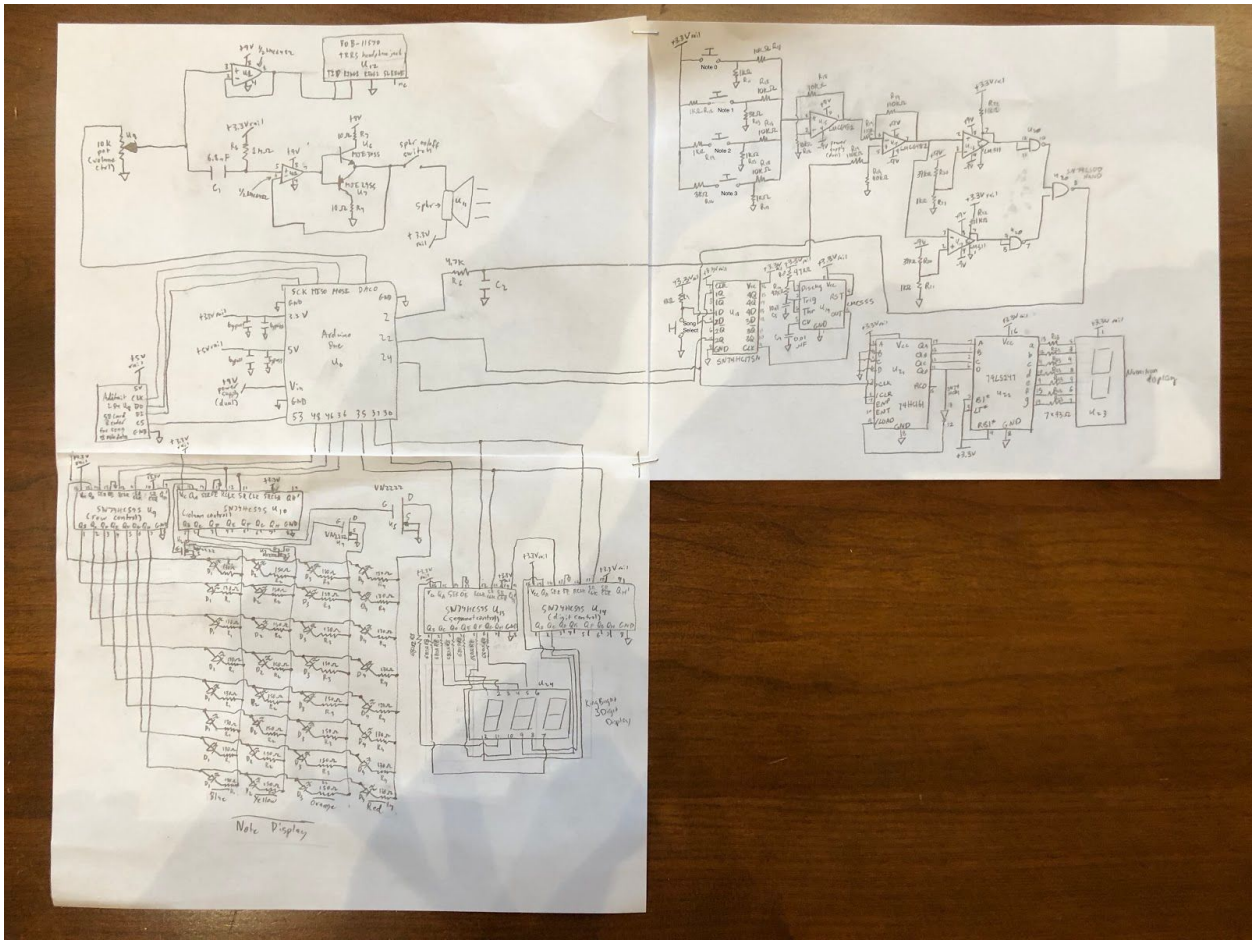
Through the course of designing and building this project, we truly realized the importance of building one subsystem at a time and testing right after building. This ultimately made it much easier to debug our project, and also made our work more efficient. In the beginning, while we had not fully implemented this build and check system, every time there was a mistake in the design or wiring, debugging was a much longer process.

Referencing previous work, homework, and labs was of course relatively easy, but coming up with new ways to implement what we had already learned was relatively more difficult. With previous labs, we could follow directions and design schematics given to us, but this project forced us to think about the possible pitfalls, pros, and cons of every possible design idea we had. This is more akin to what happens in a non-classroom setting, where inconsistencies during testing do not have such a clear-cut solution.

When inconsistencies during testing occurred during our process we were also confronted with a dilemma between using more time to debug the current design and trying to implement a different design with the same goal and function in mind. This tradeoff informed a few of our decisions within our design (e.g. the use of comparators to note-check, the debouncing of the OR output).

Ultimately, we were able to hone our circuit design skills while working on a project we were passionate about building.

Schematic:



For a full resolution copy, visit
https://www.dropbox.com/s/vvfec0k9ynobsqo/IMG_9653.JPG?dl=0
and click “Download >> Direct download”.

Conclusion:

In retrospect, we were able to complete our basic goal of creating a functional miniature Guitar Hero game. Using digital and analog circuit design components, our circuit can successfully and consistently play 8 songs, light up corresponding columns of LEDs, play the song through the speaker, and allow for an interactive experience with users by allowing for users to hit corresponding input note buttons and select song choice using another input button.

In the future, if given the opportunity to improve upon the project, we would include a motor on the circuit board that vibrates when the player misses a note and flash the bottom row of LEDs red when the player misses a note and green when the player hits a note. We would also like to incorporate more songs if we had the time.

Bibliography:

https://appelsiini.net/2011/how-does-led-matrix-work/
https://www.allaboutcircuits.com/technical-articles/turn-your-pwm-into-a-dac/
https://www.allaboutcircuits.com/technical-articles/low-pass-filter-a-pwm-signal-into-an-analog-voltage/
http://www.cablechick.com.au/blog/understanding-trrs-and-audio-jacks/

Appendix:

```
/*****************************************
 * final_project.c                       *
 * 11/21/2017                            *
 * ES52 Final Project: Miniature Guitar Hero *
 * Andrew Shackelford and Vivian Qiang       *
 * ashackelford@college.harvard.edu          *
 * vqiang@college.harvard.edu                *
 * This program controls an Arduino Due      *
 * to run a miniature Guitar Hero style game *
 * using 32 LEDs and 4 push-buttons.         *
 *****************************************/

// libraries
#include <SD.h>
#include <SPI.h>
#include <Audio.h>
#include <DueTimer.h>

// SD PIN
#define SD_SELECT 53

// NOTE PINS
#define NOTE_SDO 48
#define NOTE_CLK 36
#define NOTE_LOAD 46

// SCORE PINS
#define SCORE_CLK 35
#define SCORE_SDO 30
#define SCORE_LOAD 31

// INT PINS
#define SONG_INT_PIN 22
#define NOTE_INT_PIN 24

// VOLTAGE REFERENCES
#define NOTE_REF_OUT 2
const int REFERENCES[4] = {144, 97, 69, 34}; // out of 255

// CONSTANTS
const int NUM_SONGS = 8;
const int BUFFER_SIZE = 1024; // number of audio samples to read
const int VOLUME = 1024;

// global variables for keeping track of songs
```

```
46 int Score = 0;
47 int SongNum = 0;
48
49 // global variables for keeping track of notes
50 bool Notes[4][8];
51 long CurFrameTime = 0;
52 int Column = 0;
53
54 // global variables for keeping track of files / audio
55 short AudioBuffer[BUFFER_SIZE];
56 File NoteFile;
57 File SongFile;
58
59 // ISR variables
60 volatile bool ShouldChangeSong = false;
61 volatile bool ShouldCheckNoteHit = false;
62
63 /* helper functions */
64
65 /***********************************
66 * converts a byte to a bool array *
67 ***********************************/
68 void byteToArr(unsigned char c, bool b[8])
69 {
70  for (int i = 0; i < 8; i++) {
71    b[i] = (c & (1<<i)) != 0;
72  }
73 }
74
75 /******************
76 * sets a pin low *
77 ******************/
78 void setPinLow(int pin) {
79  digitalWrite(pin, LOW);
80 }
81
82 /*******************
83 * sets a pin high *
84 *******************/
85 void setPinHigh(int pin) {
86  digitalWrite(pin, HIGH);
87 }
88
89 /***********************
90 * sends a clock pulse *
91 ***********************/
92 void sendClock(int pin) {
93  setPinLow(pin);
```

```
 94   setPinHigh(pin);
 95 }
 96
 97 /* ISRs */
 98
 99 /*********************************
100 * ISR to change to the next song *
101 *********************************/
102 void changeSongISR() {
103   ShouldChangeSong = true;
104 }
105
106 /*******************************
107 * ISR to check for a note hit *
108 *******************************/
109 void checkNoteISR() {
110   ShouldCheckNoteHit = true;
111 }
112
113 /***************************
114 * change to the next song *
115 ***************************/
116 void changeSong() {
117   // update ISR flag
118   ShouldChangeSong = false;
119
120   // close files
121   SongFile.close();
122   NoteFile.close();
123
124   // update song num, reset score data
125   SongNum += 1;
126   if (SongNum > NUM_SONGS) {
127     SongNum = 1;
128   }
129   Score = 0;
130
131   // load in notes and new song
132   loadNoteData();
133   loadSongData();
134
135   // update start time
136   CurFrameTime = millis();
137
138   // skip the first few frames to get the notes
139   // and music in sync
140   for (int i = 0; i < 7; i++) {
141     updateNoteData();
```

```
142  }
143
144 }
145
146 /*************************************
147 * compare combination of user input and *
148 * reference output to see if user       *
149 * hit or missed note                    *
150 *************************************/
151 void checkNoteHit() {
152  ShouldCheckNoteHit = false;
153  if (!digitalRead(NOTE_INT_PIN)) {
154    // if the button(s) pressed are correct
155    if (Notes[0][7] || Notes[1][7] || Notes[2][7] || Notes[3][7]) {
156      // the player has pressed the button correctly,
157      // increment the score
158      if (Score < 999) {
159        Score++;
160      } else {
161        Score = 999;
162      }
163    } else {
164      // the button press is correct but there are no notes,
165      // so don't change the score
166    }
167  } else {
168    // the button(s) pressed are not correct
169    if (Notes[0][6] || Notes[1][6] || Notes[2][6] || Notes[3][6] ||
170        Notes[0][7] || Notes[1][7] || Notes[2][7] || Notes[3][7]) {
171      // the player pressed or released the button a bit early,
172      // but within the margin of error so it's all good
173      // (this also occurs when the player simply misses a note,
174      // but we will simply not give them a point instead of
175      // penalizing them a point)
176    } else {
177      // the player pressed the button when there was not a note,
178      // so penalize them a point
179      if (Score > 0) {
180        Score--;
181      } else {
182        Score = 0;
183      }
184    }
185  }
186 }
187
188 /* song data loading */
189
```

```
/***********************
* load in a song file *
***********************/
void loadSongData() {
 char str[8];
 sprintf(str, "%d", SongNum);
 strcat(str, ".wav");
 SongFile = SD.open(str);
}

/* note data loading/updating */

/***********************
* load in a note file *
***********************/
void loadNoteData() {
 char str[8];
 sprintf(str, "%d", SongNum);
 strcat(str, ".txt");
 NoteFile = SD.open(str);
 updateNoteData();
}

/************************************************
* update the note array with a new set of frames *
************************************************/
void updateNoteData() {
 for (int i = 0; i < 4; i++) {
   if (NoteFile.available()) {
     byteToArr(NoteFile.read(), Notes[i]);
   } else {
     byteToArr(0x00, Notes[i]);
   }
 }
}

/* serial data sending */

/*******************************************
* send the note and score data (multiplexed) *
*******************************************/
void sendData() {
 // reset Column
 if (Column > 3) {
   Column = 0;
 }

 // send note data
```

```
238  sendNoteData();
239
240  // send score data
241  switch (Column) {
242    case 0:
243      sendDigit(Score % 10, Column);
244      break;
245    case 1:
246      sendDigit((Score/10) % 10, Column);
247      break;
248    case 2:
249      sendDigit(Score/100, Column);
250      break;
251  }
252
253  // increment column
254  Column++;
255 }
256
257
258 /************************************************
259 * sends out note data to 2 8-bit shift registers *
260 * for a 4x8 multiplexed LED array                *
261 ************************************************/
262 void sendNoteData() {
263  // send LED row data
264  for (int r = 0; r < 4; r++) {
265    if (r == Column) {
266      setPinHigh(NOTE_SDO);
267    } else {
268      setPinLow(NOTE_SDO);
269    }
270    sendClock(NOTE_CLK);
271  }
272
273  // send LED column data
274  for (int y = 0; y < 8; y++) {
275    if (Notes[3-Column][7-y]) {
276      setPinHigh(NOTE_SDO);
277    } else {
278      setPinLow(NOTE_SDO);
279    }
280    sendClock(NOTE_CLK);
281  }
282
283  sendClock(NOTE_LOAD);
284 }
285
```

```
286 /*******************************************************
287 * sends out score data to 2 8-bit shift registers for  *
288 * a 3-digit 7-segment multiplexed display              *
289 *******************************************************/
290 void sendDigit(int n, int d) {
291  // send digit data
292  setPinLow(SCORE_SDO);
293  for (int i = 0; i < d; i++) {
294    sendClock(SCORE_CLK);
295  }
296
297  setPinHigh(SCORE_SDO);
298  sendClock(SCORE_CLK);
299
300  setPinLow(SCORE_SDO);
301  for (int i = 2; i > d; i--) {
302    sendClock(SCORE_CLK);
303  }
304
305  // send number data
306  bool b[8];
307  intToSevenSegment(n, b);
308  for (int i = 7; i >= 0; i--) {
309    if (b[i]) {
310      setPinHigh(SCORE_SDO);
311    } else {
312      setPinLow(SCORE_SDO);
313    }
314    sendClock(SCORE_CLK);
315  }
316
317  sendClock(SCORE_LOAD);
318 }
319
320 /*************************************************************
321 * convert a one-digit integer to a 7-segment boolean array *
322 *************************************************************/
323 void intToSevenSegment(int n, bool b[8]) {
324  // active low
325  switch (n) {
326    case 0:
327      b[0] = 0; // a
328      b[1] = 0; // b
329      b[2] = 0; // c
330      b[3] = 0; // d
331      b[4] = 0; // e
332      b[5] = 0; // f
333      b[6] = 1; // g
```

```
334       b[7] = 1; // dp
335       break;
336     case 1:
337       b[0] = 1; // a
338       b[1] = 0; // b
339       b[2] = 0; // c
340       b[3] = 1; // d
341       b[4] = 1; // e
342       b[5] = 1; // f
343       b[6] = 1; // g
344       b[7] = 1; // dp
345       break;
346     case 2:
347       b[0] = 0; // a
348       b[1] = 0; // b
349       b[2] = 1; // c
350       b[3] = 0; // d
351       b[4] = 0; // e
352       b[5] = 1; // f
353       b[6] = 0; // g
354       b[7] = 1; // dp
355       break;
356     case 3:
357       b[0] = 0; // a
358       b[1] = 0; // b
359       b[2] = 0; // c
360       b[3] = 0; // d
361       b[4] = 1; // e
362       b[5] = 1; // f
363       b[6] = 0; // g
364       b[7] = 1; // dp
365       break;
366     case 4:
367       b[0] = 1; // a
368       b[1] = 0; // b
369       b[2] = 0; // c
370       b[3] = 1; // d
371       b[4] = 1; // e
372       b[5] = 0; // f
373       b[6] = 0; // g
374       b[7] = 1; // dp
375       break;
376     case 5:
377       b[0] = 0; // a
378       b[1] = 1; // b
379       b[2] = 0; // c
380       b[3] = 0; // d
381       b[4] = 1; // e
```

```
382       b[5] = 0; // f
383       b[6] = 0; // g
384       b[7] = 1; // dp
385       break;
386     case 6:
387       b[0] = 0; // a
388       b[1] = 1; // b
389       b[2] = 0; // c
390       b[3] = 0; // d
391       b[4] = 0; // e
392       b[5] = 0; // f
393       b[6] = 0; // g
394       b[7] = 1; // dp
395       break;
396     case 7:
397       b[0] = 0; // a
398       b[1] = 0; // b
399       b[2] = 0; // c
400       b[3] = 1; // d
401       b[4] = 1; // e
402       b[5] = 1; // f
403       b[6] = 1; // g
404       b[7] = 1; // dp
405       break;
406     case 8:
407       b[0] = 0; // a
408       b[1] = 0; // b
409       b[2] = 0; // c
410       b[3] = 0; // d
411       b[4] = 0; // e
412       b[5] = 0; // f
413       b[6] = 0; // g
414       b[7] = 1; // dp
415       break;
416     case 9:
417       b[0] = 0; // a
418       b[1] = 0; // b
419       b[2] = 0; // c
420       b[3] = 1; // d
421       b[4] = 1; // e
422       b[5] = 0; // f
423       b[6] = 0; // g
424       b[7] = 1; // dp
425       break;
426     default:
427       b[0] = 1; // a
428       b[1] = 1; // b
429       b[2] = 1; // c
```

```
      b[3] = 1; // d
      b[4] = 1; // e
      b[5] = 1; // f
      b[6] = 1; // g
      b[7] = 1; // dp
      break;
  }
}

/* note reference */

/*************************************
* update the note reference output for *
* comparison to user input             *
*************************************/
void updateNoteReference() {
 int out = 0;
 for (int i = 0; i < 4; i++) {
   if (Notes[i][7]) {
     out += REFERENCES[i];
   }
 }
 analogWrite(NOTE_REF_OUT, out);
}


void setup() {
 // set our output pins
 pinMode(SD_SELECT, OUTPUT);
 pinMode(NOTE_REF_OUT, OUTPUT);

 pinMode(NOTE_SDO, OUTPUT);
 pinMode(NOTE_CLK, OUTPUT);
 pinMode(NOTE_LOAD, OUTPUT);

 pinMode(SCORE_SDO, OUTPUT);
 pinMode(SCORE_CLK, OUTPUT);
 pinMode(SCORE_LOAD, OUTPUT);

 // set our input pins
 pinMode(NOTE_INT_PIN, INPUT);
 pinMode(SONG_INT_PIN, INPUT);

 // attach interrupts
 attachInterrupt(digitalPinToInterrupt(NOTE_INT_PIN), checkNoteISR,
CHANGE);
 attachInterrupt(digitalPinToInterrupt(SONG_INT_PIN), changeSongISR,
FALLING);
```

```
476
477   // initialize SD card
478   Serial.begin(9600);
479   Serial.print("Initializing SD card...");
480   pinMode(SD_SELECT, OUTPUT);
481   if (!SD.begin(SD_SELECT)) {
482     Serial.println("initialization failed!");
483     return;
484   }
485   Serial.println("initialization done.");
486
487   Audio.begin(44100, 100); // Audio at 44100 kHz and 100 ms of
pre-buffering
488   Timer3.attachInterrupt(sendData).start(1000); // multiplexing timer at
1000 Hz
489 }
490
491 void loop() {
492   if (ShouldChangeSong) {
493     changeSong();
494   }
495
496   if (ShouldCheckNoteHit) {
497     checkNoteHit();
498   }
499
500   if (millis() > CurFrameTime + 25) {
501     // every 25 ms, send out a new frame of note data
502     CurFrameTime += 25;
503     updateNoteData();
504     updateNoteReference();
505   }
506
507   if (SongFile.available()) {
508     // send audio
509     SongFile.read(AudioBuffer, sizeof(AudioBuffer));
510     Audio.prepare(AudioBuffer, BUFFER_SIZE, VOLUME);
511     Audio.write(AudioBuffer, BUFFER_SIZE);
512   }
513 }
```

```
"""
note_record.py
11/21/2017
ES52 Final Project: Miniature Guitar Hero
Andrew Shackelford and Vivian Qiang
ashackelford@college.harvard.edu
vqiang@college.harvard.edu
This script records keystrokes while playing
a specified WAV file to allow the user
to record notes for use in our
Minature Guitar Hero project.
"""

import pyaudio
import wave
import threading
import readchar
import time
import sys

class Player(threading.Thread):

    def __init__(self, song_file):
        super(Player, self).__init__()
        self.song_file = song_file

    def play_song(self, song_file):
        #define stream chunk
        chunk = 1024

        #open a wav format music
        f = wave.open(song_file,"rb")
        #instantiate PyAudio
        p = pyaudio.PyAudio()
        #open stream
        stream = p.open(format = p.get_format_from_width(f.getsampwidth()),
                        channels = f.getnchannels(),
                        rate = f.getframerate(),
                        output = True)
        #read data
        data = f.readframes(chunk)

        #play stream
        while data:
            stream.write(data)
            data = f.readframes(chunk)

        #stop stream
```

```
        stream.stop_stream()
        stream.close()

        #close PyAudio
        p.terminate()

    def run(self):
        self.play_song(self.song_file)

def record():
    # get song_file and name
    song_file = sys.argv[1]
    song_name = song_file[:-4]

    # start playing the song
    player = Player(song_file)
    player.daemon = True
    player.start()
    start_time = time.time()

    # record each key (note) press
    notes = [[], [], [], []]
    note = ''
    while True:
        note = repr(readchar.readchar())
        note_time = time.time()
        print(note)
        if note == '\'d\'':
            notes[0].append(note_time - start_time)
        elif note == '\'f\'':
            notes[1].append(note_time - start_time)
        elif note == '\'j\'':
            notes[2].append(note_time - start_time)
        elif note == '\'k\'':
            notes[3].append(note_time - start_time)
        else:
            break

    # write the note timings out to four separate text files
    for i in range(4):
        with open(song_name + "_" + str(i) + '.txt', 'w') as f:
            for note in notes[i]:
                f.write(str(note) + '\n')

if __name__ == "__main__":
    record()
```

```
"""
note_convert.py
11/21/2017
ES52 Final Project: Miniature Guitar Hero
Andrew Shackelford and Vivian Qiang
ashackelford@college.harvard.edu
vqiang@college.harvard.edu
This script converts 4 lists of note timings
into LED frames for the Arduino to read.
"""

import sys
NOTE_SEP_TIME = 0.15 # number of seconds between note rows

def convert():
    note_list = [[], [], [], []]

    # open each list of note timings
    for i in range(4):
        with open(sys.argv[1] + "_" + str(i) + ".txt", 'r') as f:
            for line in f:
                time = float(line)
                note_list[i].append(time)
            if len(note_list[i]) == 0:
                note_list[i].append(-100.)

    # write out the LED frames for each set of notes as 4 sequential bytes,
    # with 1 byte per column and 25 ms per frame (40 Hz)
    time = 0.
    end = max(note_list[0][-1], note_list[1][-1], note_list[2][-1],
note_list[3][-1]) + 2.
    with open(sys.argv[1] + "_led.txt", 'w') as f:
        while time < end:
            for i in range(4):
                out = 0
                for note in note_list[i]:
                    if time - 1*NOTE_SEP_TIME < note <= time:
                        out += 128
                    if time < note <= time + 1*NOTE_SEP_TIME:
                        out += 64
                    if time + 1*NOTE_SEP_TIME < note <= time +
2*NOTE_SEP_TIME:
                        out += 32
                    if time + 2*NOTE_SEP_TIME < note <= time +
3*NOTE_SEP_TIME:
                        out += 16
                    if time + 3*NOTE_SEP_TIME < note <= time +
4*NOTE_SEP_TIME:
```

```
45                      out += 8
46                  if time + 4*NOTE_SEP_TIME < note <= time +
5*NOTE_SEP_TIME:
47                      out += 4
48                  if time + 5*NOTE_SEP_TIME < note <= time +
6*NOTE_SEP_TIME:
49                      out += 2
50                  if time + 6*NOTE_SEP_TIME < note <= time +
7*NOTE_SEP_TIME:
51                      out += 1
52              try:
53                  f.write(chr(out))
54              except:
55                  print(out)
56          time += 0.025
57
58 def main():
59     convert()
60
61 if __name__ == "__main__":
62     main()
```

```
"""
note_display.py
11/21/2017
ES52 Final Project: Miniature Guitar Hero
Andrew Shackelford and Vivian Qiang
ashackelford@college.harvard.edu
vqiang@college.harvard.edu
This script plays back a WAV file and its
corresponding file of note frames
to simulate what playback should look
like in the final product with
the Arduino and corresponding LEDs,
so that any note timing errors can be fixed
before loading onto the Arduino.
"""

import time
import sys
import wave
import threading
import pyaudio

class Player(threading.Thread):

    def __init__(self, song_file):
        super(Player, self).__init__()
        self.song_file = song_file

    def play_song(self, song_file):
        #define stream chunk
        chunk = 1024

        #open a wav format music
        f = wave.open(song_file,"rb")
        #instantiate PyAudio
        p = pyaudio.PyAudio()
        #open stream
        stream = p.open(format = p.get_format_from_width(f.getsampwidth()),
                        channels = f.getnchannels(),
                        rate = f.getframerate(),
                        output = True)
        #read data
        data = f.readframes(chunk)

        #play stream
        while data:
            stream.write(data)
            data = f.readframes(chunk)
```

```
49
50          #stop stream
51          stream.stop_stream()
52          stream.close()
53
54          #close PyAudio
55          p.terminate()
56
57      def run(self):
58          self.play_song(self.song_file)
59
60  """ print out a line of notes """
61  def print_line(note_zero, note_one, note_two, note_three):
62      if note_zero:
63          str_zero = "0 "
64      else:
65          str_zero = "_ "
66      if note_one:
67          str_one = "0 "
68      else:
69          str_one = "_ "
70      if note_two:
71          str_two = "0 "
72      else:
73          str_two = "_ "
74      if note_three:
75          str_three = "0 "
76      else:
77          str_three = "_ "
78      print(str_zero + str_one + str_two + str_three)
79
80  """ clear out the past lines so a new set can be printed """
81  def clear_lines():
82      for i in range(9):
83          sys.stdout.write("\033[F")
84
85  def display():
86      # get song and note files
87      song_name = sys.argv[1]
88      song_file = sys.argv[1] + ".wav"
89      note_file = sys.argv[1] + "_led.txt"
90      led_list = [[], [], [], []]
91
92      # load in all the note frames
93      count = 0
94      with open(note_file, 'r') as f:
95          while True:
96              byte = f.read(1)
```

```
            if not byte:
                break
            led_line = []
            for bit in '{0:08b}'.format(ord(byte)):
                led_line.append(int(bit))
            led_list[count%4].append(led_line)
            count += 1

    # start playing the song
    player = Player(song_file)
    player.daemon = True
    index = 1
    player.start()
    cur_time = time.time()
    start_time = cur_time

    # print the first frame
    for j in range(8):
        print_line(led_list[0][0][7-j], led_list[1][0][7-j],
                   led_list[2][0][7-j], led_list[3][0][7-j])
    print(time.time() - start_time)

    # print all subsequent frames
    while True:
        if time.time() > cur_time + 0.025:
            cur_time += 0.025
            clear_lines()
            for j in range(8):
                print_line(led_list[0][index][7-j], led_list[1][index][7-j],
                           led_list[2][index][7-j], led_list[3][index][7-j])
            print(time.time() - start_time)
            index += 1

if __name__ == "__main__":
    display()
```