LED Music Visualizer

Final Project Report

December 9, 2016

E155

Hill Balliet and James Palmer

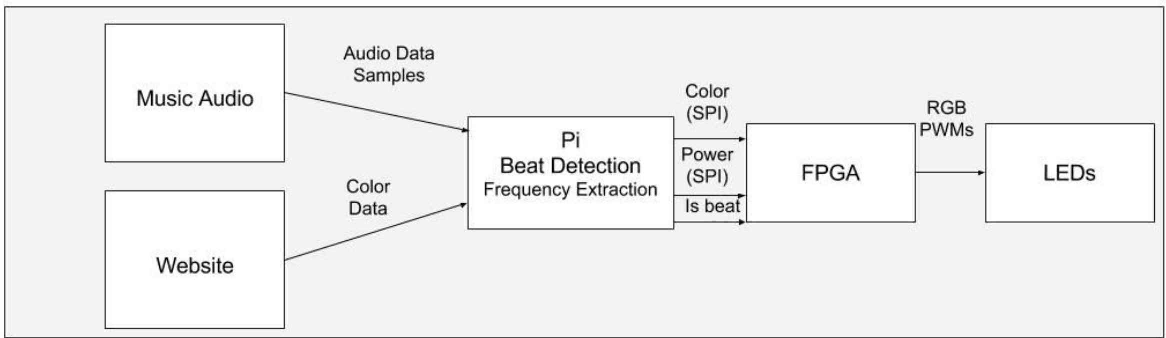
Abstract

One's environment significantly impacts his or her mood, and since people spend a large percentage of their time in indoor environments, such as the home or office, they often like to fill these spaces with art and music to make their time inside more pleasant. While people have some commercial options for visualizing music thereby uniting their visual and auditory environments, most current solutions are essentially visual equalizers. This presents a problem for the rhythmically inclined because it is not the loudness of different frequency spectrums that moves them and compels them to dance, but rather the beat. This project prototypes an auditory visualizer that flashes and changes color to the beat of a song in real time. The user can play a song on a Raspberry Pi and pick a color from a web page, and the Pi will sample and filter the song to find the beat using a thresholding algorithm. The Pi will then notify an FPGA of information about the music and what color should be displayed. The FPGA then drives an LED array to visualize the music.

Introduction

This project prototypes an auditory visualizer that flashes and changes color to the beat of a song in real time. Since the beat of music is often one of its most compelling aspects, flashing to the beat rather than just the loudness of different frequencies creates a better visualization. Accurate beat tracking also enables improvisation around the beat that could add to the music, which was implemented in this project as the LEDs changing color on every other beat.

The block diagram below shows the high level processes of the system.



To initiate the visualization, the user plays a song on a Raspberry Pi by running the program and picks a color from a web page. The color can be changed any time the user desires. The Pi will then sample and filter the song in real time to find the beat using a thresholding algorithm. The Pi will next notify an FPGA of information about the music and what color should be displayed. All the while, the FPGA is driving the LED board based on the information provided from the Pi.

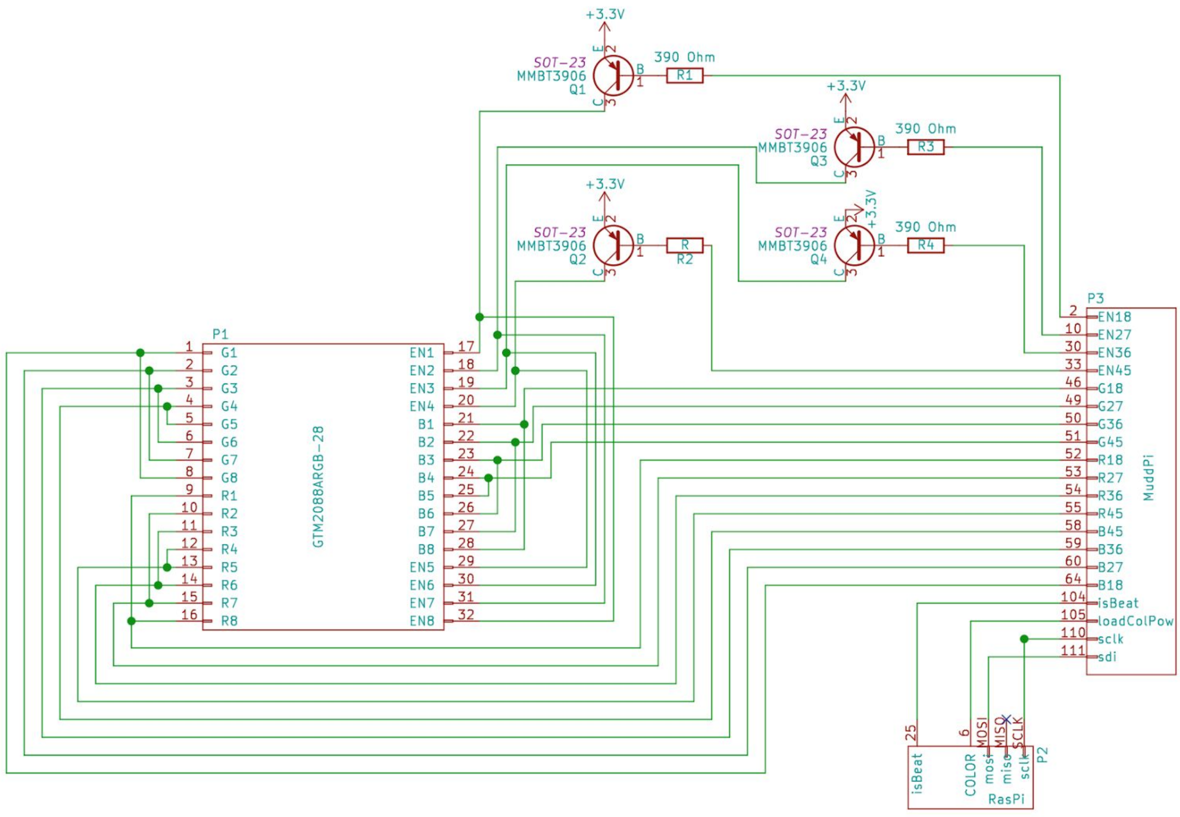
New Hardware

An eight-by-eight LED matrix was used to visualize the music. The part number is GTM2088ARGB-28 and its datasheet can be found at: https://forums.parallax.com/discussion/download/102104/GTM2088ARGB-28.pdf. One aspect to note about the datasheet is that the pinouts for the green LEDs are incorrect. While the datasheet lists pins 28 through 21 as corresponding to the green LEDs in columns one through eight respectively, the order is reversed in reality so pin 21 corresponds to column one and pin 22 corresponds to column two and so on. The correct pinouts are shown in the schematic in the “Schematics” section.

The matrix is a common anode setup with eight row enables and 24 column enables, one for red, green, and blue in each column. The intensity of the red, green, and blue LEDs were controlled by a pulse width modulation (PWM) intensity signal at the cathode and a time multiplexed enable at the anode. The LED driver was calibrated such that at full power, the LEDs would appear white with the constraint that the voltage could not be higher than the FPGA can output, which is 3.3V. The brightness as a function of the duty cycle of a PWM signal was then assumed to be linear with a y intercept of zero, such that when the voltage across the cathode and anode was always zero, the LEDs would be off. The table below lists the measured maximum values that produced white while remaining under 3.3V.

	Red (V)	Green (V)	Blue (V)
Max voltage from cathode to anode	2.50	3.08	3.19

Schematics

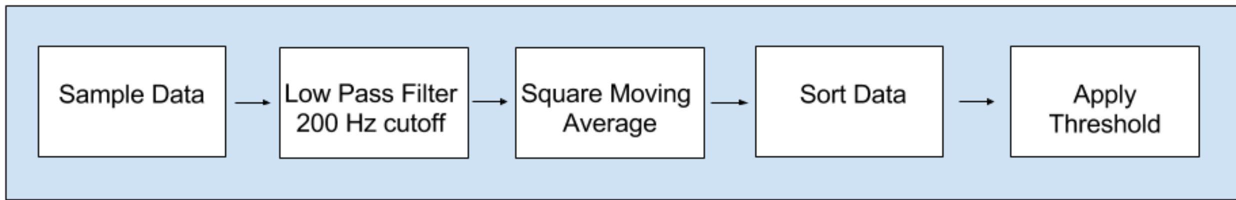


The schematics above show the circuitry required to connect the Raspberry Pi, the FPGA, and the LED matrix. Because the FPGA could not provide enough current to drive the LED matrix, the FPGA powered the LED matrix by turning transistors on and off. The current going through the base was limited by a 390 Ω resistor so that the transistor would not draw too much current from the FPGA.

Note that many of the pins on the LED matrix are shorted together. This was done to increase efficiency because the design on the board was symmetrical and is addressed in further detail in the “FPGA Design” section.

Microcontroller Design

The figure below shows the data flow for the microcontroller audio processing on the Raspberry Pi. The code for the audio processing on the Raspberry Pi can be found in Appendix A.



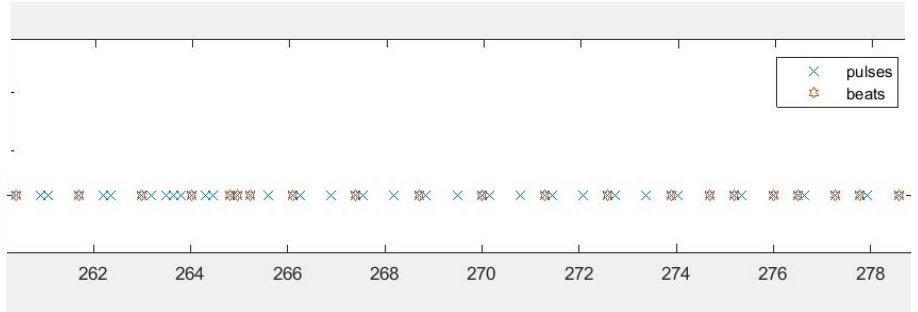
Data is sampled every 500 microseconds for a sampling rate of 2 kHz. Once the data has been sampled, it is shifted into a linked list. Only 1.2 seconds of audio is stored at a time since beats are statistically unlikely to be more than 1.2 seconds apart in modern music. After 1.2 seconds, old audio samples will be shifted out. A low pass elliptic filter with a 200 Hz cutoff frequency is then applied to the raw data to make the bass and drumlines more prominent compared to the melody. The data is then squared and put through a moving average filter to find the envelope of the signal. Before the threshold is applied to determine whether the song is currently at a beat, the envelope data is stored in a sorted array so that quantile information can be found easily. Finally, the threshold is applied and the algorithm determines whether a beat is occurring.

The Raspberry Pi also provides a web interface for users to change the primary color of the LEDs. The user can select a color, which is sent to the FPGA via SPI. The code for this is found in Appendix B.

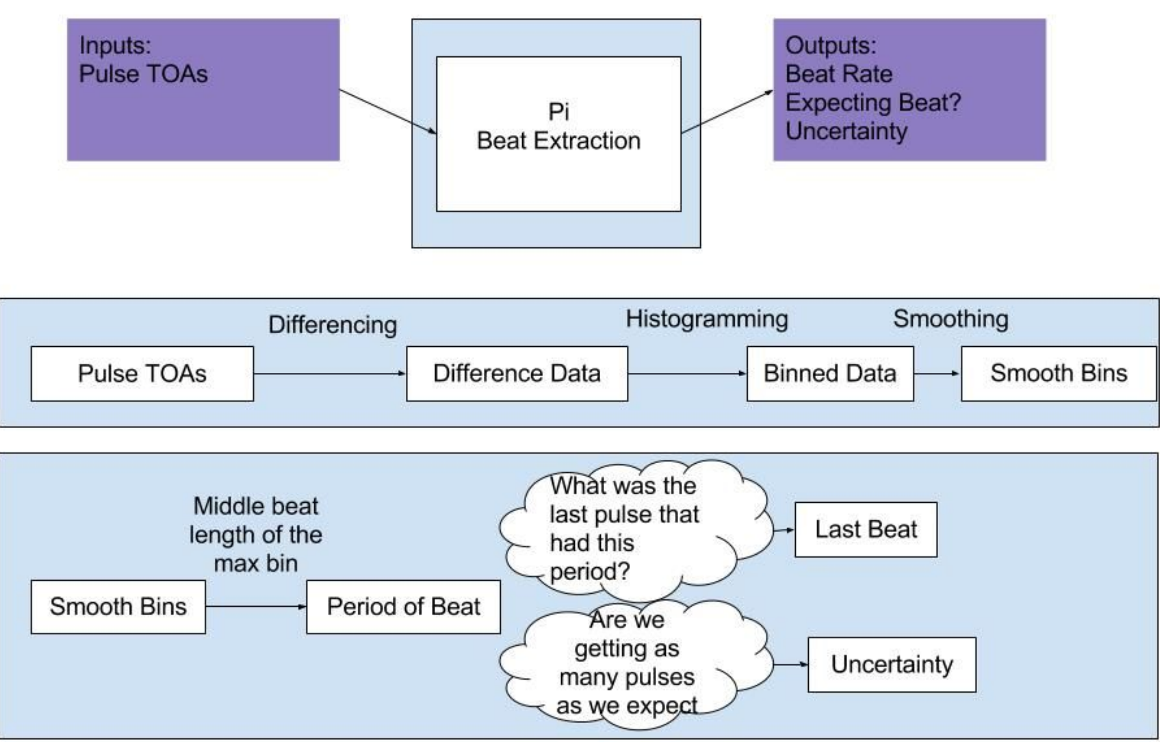
Suggestions for Future Improvement

The final implemented algorithm could be improved by incorporating feedback from the beat frequency estimator. This algorithm extracts the period of the beat and calculates the most likely last true beat detected. From these two pieces of information the time until the next expected beat can be calculated. This can be used in a feedback loop to set a variable threshold for the beat detector where the threshold is lower when a beat is expected and grows exponentially when the next expected beat is farther away. This feedback loop could help reduce the rate of both false positive and false negative detection.

Below is a Matlab plot showing the beats detected by the thresholding algorithm alone, labeled “pulses,” and the beats detected using the feedback loop, labeled “beats.”



The above graph was generated using an earlier iteration of both algorithms, but serves to demonstrate how the feedback loop can reduce false positive detections.

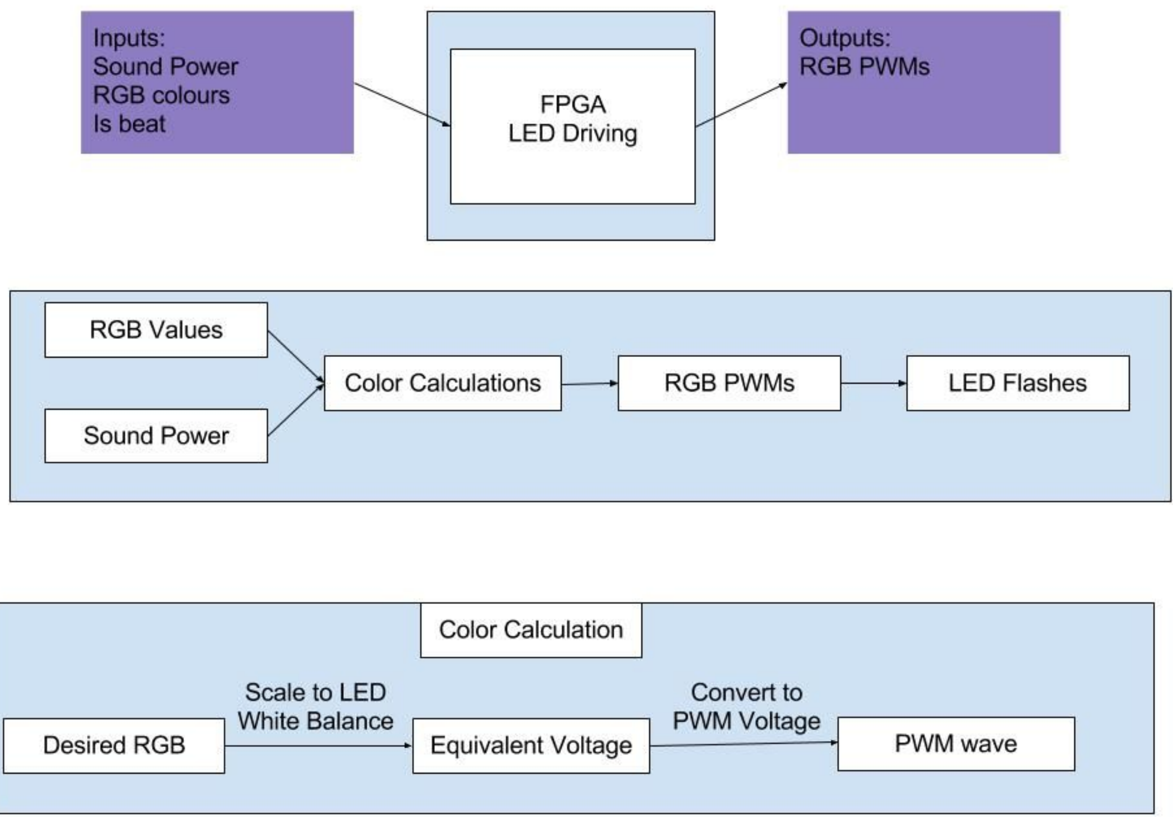


The above block diagram sketches out the process for adding the feedback loop with a sequential beat differentiation algorithm. The algorithm is based off of the idea of a cumulative difference algorithm[2]. The algorithm takes the difference in time between each of the most recent 25 pulses and five of its neighbour pulses. It then creates a histogram of these differences. The bins of the histogram are then summed with their neighbours to reduce the effect of noise in the data. The max of these smoothed bins is then taken to be the beat. The most recent saved pulse to have a neighbor the beat period away is taken as the last true beat. The expected next beat is estimated by adding the beat period to the time the last beat occurred.

MATLAB code for the beat differentiation algorithm is included in Appendix C.

FPGA Design

Overall, the FPGA is responsible for driving the LED matrix based on the information that is passed by the Raspberry Pi. Specifically, the FPGA takes the current power of the song, the primary color of the LED matrix, and a one bit input specifying whether a beat is occurring as input. It outputs pulse width modulated signals that create the visualization on the LED matrix. Verilog code for the FPGA modules can be found in Appendix D.



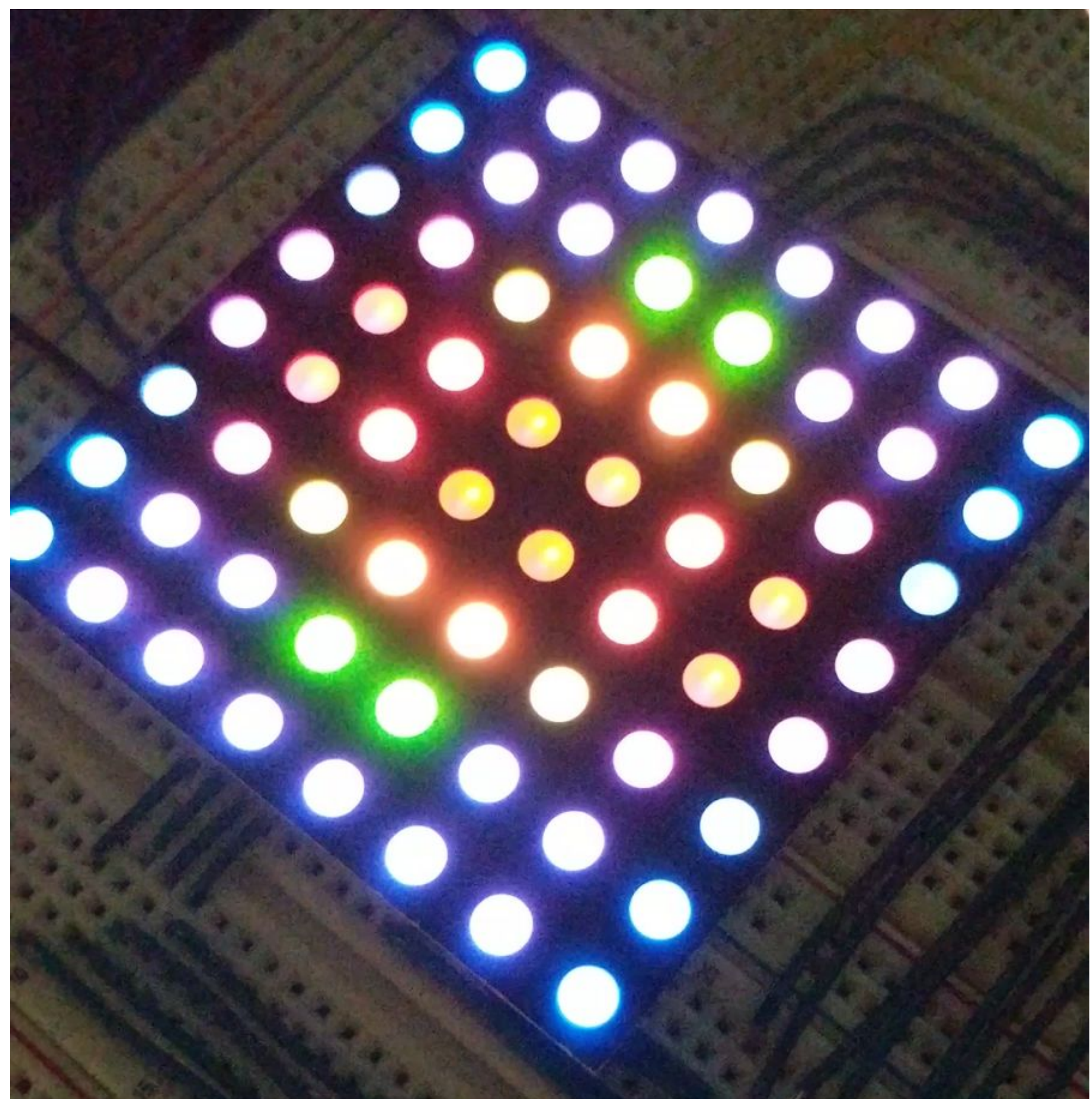
The FPGA receives the primary color and power data from the Pi via an spi slave connection. The specific color to display on each LED is then calculated. There are three types of LED in the prototype. The first type varies based on the power of the song so the color of these LEDs maintains the ratio of red, green, and blue given by the Pi, but the intensity is scaled by the ratio of the power to the maximum power. The second type of LED blinks every time a beat is detected. The color of these LEDs is calculated by flipping the bits of the color sent by the Pi. This is done to create visual interest. The third type of LED also blinks every time a beat is detected, but changes color on every other beat. The color for these LEDs is calculated by rotating the bits of

the color given by the Pi by one or two bytes so that the red, green, and blue values are swapped, once again creating visual interest.

To create the PWM signal, the color data is first used to determine an effective voltage to apply to each LED. The effective voltages are found by linearly interpolating between zero and the maximum values described in the “New Hardware” section. Then the resulting voltage is used to calculate the duty cycle for each color’s PWM wave that will be sent to the LEDs.

In order to be efficient with the hardware, the rows of the matrix are time-multiplexed, and the symmetry of the pattern displayed was used to reduce the number of signals generated. Columns of the same color, equidistant from the center were shorted as well as rows of enables that were equidistant from the center because they will always have the same value. The time-multiplexing and use of symmetry reduced the necessary hardware data paths from 64 to 4.

The photo of the LED matrix in action below shows the inherent symmetry of the design.



Results

The final prototype from the project is able to detect beats consistently for the song Africa by Toto, and others with a pronounced and simple drumline. The prototype is less effective for songs with complex rhythm in the bass and drumline, but is still somewhat effective. The beat finding algorithm is ineffective for legato songs that lack both a bassline and drumline as can be found in some traditional acapella and violin music, but visualizing the beat is not as relevant for this type of music since it is not prominent and compelling to the listener in the way that it is in more modern, secular music.

The system was originally proposed to sample the music through a microphone, but that aspect of the project was abandoned because of signal quality issues. The microphone that was purchased had a variable gain, which would keep the signal from clipping, but the variable gain had the side effect of reducing the dynamic range. This meant that the peaks in the signal amplitude that are characteristic of beats were compressed and essentially indistinguishable from the rest of the song.

The original proposal also put the signal processing algorithm on the FPGA, but the algorithm took longer to design than anticipated, so in order to finish the project by the deadline, the algorithm was instead implemented in C on the Raspberry Pi.

References

[1] Linköpings Universitet,
http://www.isy.liu.se/edu/kurs/TSEA81/lecture_linux_realtime.html
[2] Improved algorithm for the deinterleaving of radar pulses
http://ieeexplore.ieee.org/document/120767/

Parts List

Part	Source	Vendor Part #	Price
GEEEtech LED Matrix 8x8	Amazon	GTM2088ARGB-28	$7.95
Electret Microphone Amplifier - MAX9814 with Auto Gain Control	Adafruit	1713	$7.95

Appendix A: Raspberry Pi Audio Processing

micData.c

```
// E155 Final Project
// micData.c
// Hill Balliet - wballiet@g.hmc.edu - Dec 2, 2016
//
// Stores and processes the song data using a low pass filter to isolate bass
// frequencies and a moving average filter to find the envelope of the audio.
//
// Also sorts the incoming data points by amplitude so that quantile information
// can easily be found later.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SAMPERSEC 2000
#define DATALENGTH 2400
#define NUMTOAS 25
#define OFFSET 380
#define Y3LENGTH 1035020

typedef struct node {
    double val;
    struct node *next;
    struct node *prev;
} node_t;

typedef struct linkedList {
    node_t *first;
    node_t *last;
    int numSamples;
} linkedList_t;

// Keep track of time of arrivals
linkedList_t Toas;

FILE *fp;

// Keep track of mic data, filtered mic data, and ordered filtered mic data
linkedList_t DataShiftReg;
linkedList_t BassData;
linkedList_t SquarAvData;
double *OrderedMicData[DATALENGTH];

// Keeps track of the current time in sample number since the beginning
long CurTime = 0;
```

```
/// Initialize a new node with the given value
node_t* initNode(double value) {
    node_t *newData = (node_t *) malloc(sizeof(node_t));
    newData->val = value;
    newData->next = NULL;
    newData->prev = NULL;

    return newData;
}

/// Deletes the first node from the global data list and returns that node
// The popped node needs to be deallocated because destroy list will no longer
// deallocate it now that it's not in the list
node_t* popFirst(linkedList_t *list) {
    node_t *oldNode = list->first;

    list->first = oldNode->next;
    if (list->first != NULL)
        list->first->prev = NULL;

    list->numSamples--;

    return oldNode;
}

/// Adds the given element to the end of the global data list
void pushBack(linkedList_t *list, node_t *newNode) {
    if (list->first == NULL)
        list->numSamples = 0;

    if (list->numSamples > 0) {
        list->last->next = newNode;
        newNode->prev = list->last;
    }
    else {
        list->first = newNode;
    }

    list->last = newNode;


    list->numSamples++;
}

/// Adds the given element to the ordered data array in sorted order
void addOrdered(double *newData, double *oldData) {
    int newDataIndex = DATALENGTH - SquarAvData.numSamples;
    double *temp[DATALENGTH];
    char added = 0;
```

```
    int initial = DATALENGTH - SquarAvData.numSamples;

    for (int oldDataIndex = initial; oldDataIndex < DATALENGTH;oldDataIndex++) {
        if (oldDataIndex == DATALENGTH - 1 && !added) {
            temp[newDataIndex] = newData;
        }
        // Skip the old data
        else if (oldData == OrderedMicData[oldDataIndex]) {
            newDataIndex--;
        }
        // Insert the new data in the middle and advance only in new array
        else if (*newData < *OrderedMicData[oldDataIndex] && !added) {
            temp[newDataIndex] = newData;
            oldDataIndex--;
            added = 1;
        }
        // Copy over
        else {
            temp[newDataIndex] = OrderedMicData[oldDataIndex];
        }
        newDataIndex++;
    }

    // Copy the ordered data back into the real array
    for (int i = DATALENGTH - SquarAvData.numSamples; i < DATALENGTH; i++) {
        OrderedMicData[i] = temp[i];
    }
}

/**
* Applies a 10th order elliptic filter with 1 dB of passband ripple, 60dB
* to stopband, and wc = 200 Hz to the raw mic data.
*
* From
* https://www.mathworks.com/help/signal/ug/filter-implementation-and-analysis.html
* "This is the standard time-domain representation of a digital filter, computed
* starting with y(1) and assuming a causal system with zero initial conditions.
* This representation's progression is
*
* y(1)=b(1)x(1)
* y(2)=b(1)x(2)+b(2)x(1)-a(2)y(1)
* y(3)=b(1)x(3)+b(2)x(2)+b(3)x(1)-a(2)y(2)-a(3)y(1)
* ⋮"
*
* **Note that Matlab indexes from 1 as opposed to C, which indexes from 0**
*
*
* As calculated by Matlab, using: [b, a] = ellip(10, 1, 60, 0.1*2);
* b = [0.0024, -0.0131, 0.0372, -0.0700, 0.0984, -0.1093, 0.0984, -0.0700,
*       0.0372, -0.0131, 0.0024]
```

```
*
* a = [1.0000, -8.1884, 31.3052, -73.3202, 116.2455, -130.1808, 104.2046,
*       -58.8533, 22.4489, -5.2256, 0.5644]
*
*/
double bassFilter() {
    // Initialize constants that define the filter.
    const int numFilterPoints = 11;
    double y = 0;

    double b[11] =
{0.0023828536056377,-0.0130958334492431,0.0372141533382333,-0.0700301623207206,0.0983815662084
311,-0.1093276527751890,0.0983815662084310,-0.0700301623207205,0.0372141533382332,-0.013095833
4492431,0.0023828536056377};

    double a[11] =
{1.0000000000000000,-8.1883658956990786,31.3051598439003200,-73.3201552438257380,116.245533675
9196882,-130.1807939032020158,104.2045701322850846,-58.8532949438235136,22.4489181172146317,-5
.2255736205161751,0.5644254019455265};


    // Apply the filter to the current data as described above
    node_t *curX = DataShiftReg.last;
    node_t *curY = BassData.last;

    for (int i = 0; i < fmin(DataShiftReg.numSamples, numFilterPoints); i++) {
        if (i == 0) {
            y += (double)curX->val * b[i];
            curX = curX->prev;
        }
        else {
            y += ((double)curX->val * b[i]) - ((double)curY->val * a[i]);
            curX = curX->prev;
            curY = curY->prev;
        }
    }
    return y;
}

/**
* Squares the signal and takes a numPoints point moving average.
*/
double squarAvFilter() {
    double sum = 0;
    int numPoints = SAMPERSEC/20;

    node_t *curNode = BassData.last;

    for (int i = 0; i < fmin(BassData.numSamples, numPoints); i++) {
        sum += curNode->val * curNode->val;
```

```
        curNode = curNode->prev;
    }

    return sum/fmin(BassData.numSamples, numPoints);
}

void storeSample(int micData) {
    // subtract out the offset from the ADC
    micData -= OFFSET;

    // Init the elements in ordered array to 0 so that sorting works
    if (SquarAvData.numSamples == 0) {
        for (int i = 0; i < DATALENGTH; i++) {
            OrderedMicData[i] = 0;
        }
    }

    // Initialize the new data point and shift it in
    node_t *newData = initNode(micData);
    pushBack(&DataShiftReg, newData);

    // Initialize the new bass data point and shift it in
    node_t *newBassData = initNode(bassFilter());
    pushBack(&BassData, newBassData);

    // Initialize the new squarAv data point and shift it in
    node_t *newSquarAvData = initNode(squarAvFilter());
    pushBack(&SquarAvData, newSquarAvData);

    // Remove old data and put the new data into the ordered array
    if (DataShiftReg.numSamples > DATALENGTH) {

        node_t *oldData = popFirst(&DataShiftReg);
        node_t *oldBassData = popFirst(&BassData);
        node_t *oldSquarAvData = popFirst(&SquarAvData);

        addOrdered(&(newSquarAvData->val), &(oldSquarAvData->val));

        // Deallocate the memory for the old data.
        free(oldData);
        free(oldBassData);
        free(oldSquarAvData);
    }
    else {

        addOrdered(&(newSquarAvData->val), NULL);
    }

    CurTime++;
}
```

```
/// Deallocates all of the memory associated with the list
void destroyLists() {
    while (DataShiftReg.numSamples != 0) {
        node_t *oldData = popFirst(&DataShiftReg);
        free(oldData);
    }
    while (BassData.numSamples != 0) {
        node_t *oldBassData = popFirst(&BassData);
        free(oldBassData);
    }
    while (SquarAvData.numSamples != 0) {
        node_t *oldSquarAvData = popFirst(&SquarAvData);
        free(oldSquarAvData);
    }
}
```

beatFinding.c

```
// E155 Final Project
// beatFinding.c
// Hill Balliet - wballiet@g.hmc.edu - Dec 3, 2016
//
// Finds the beat of the song using the processed song data

// Since songs will not go above 250 BPM we can be sure that there will
// not be more than one beat per 0.15 seconds
#define WAITTIME (int)(SAMPERSEC*0.15)

// Determines whether there is currently a beat or not
int isBeat() {

    // Set the threshold based on Matlab simulations
    float quantile = 0.95;

    double offset = (int)roundf((1.0-quantile)*(SquarAvData.numSamples)) - 1;
    double threshold = *OrderedMicData[DATALENGTH - offset];

    // We only want to check the current state of beat or not beat
    node_t *curData = SquarAvData.last;

    // If the signal is crossing the quantile
    if (curData->val > threshold && curData->prev->val < threshold)
        return(1);
    else
        return(0);
}
```

Final_Project.c

```
// E155 Final Project
// Final_Project.c
// Hill Balliet - wballiet@g.hmc.edu - Dec 4, 2016
//
// Wraps the functionality of storing data, processing data, finding beats, and
// communicating with the FPGA.
//
// Compile with:
// gcc -std=gnu11 -lm -lrt -o witnessMe -g Final_Project.c -Wall -Wextra

#include "micData.c"
#include "io.c"
#include "beatFinding.c"

#include <unistd.h>
#include <time.h>
#include <sched.h>

#define ISBEATPIN 19

struct timespec ts;

static void sleep_until(struct timespec *ts, int delay);
void sample(void);
void playSong(void);

// Initializes all of the necessary IO and executes all of the core
// functionality in the while loop in the proper order
int main(void) {

    // Init the I/O functionality
    pioInit();
    spioInit();
    sysTimerInit();
    pinMode(COLOR,OUTPUT);

    digitalWrite(COLOR,1);

    pinMode(8,ALT0);
    pinMode(9,ALT0);
    pinMode(10,ALT0);
    pinMode(11,ALT0);
    unsigned int freq = 500000;
    SPI0CLK = 250000000/freq;    // set SPI clock to 250MHz / freq
    pinMode(ISBEATPIN, OUTPUT);
    fp = fopen("piMusic.wav","r");

    // There won't be beats closer together than WAITTIME in typical music
```

```
int wait = 0;
   playSong();

   // System timing functionality taken from:
   // http://www.isy.liu.se/edu/kurs/TSEA81/lecture_linux_realtime.html
   // Set our thread to real time priority
   struct sched_param sp;
   sp.sched_priority = 1; // Must be > 0. Threads with a higher
   // value in the priority field will be schedueld before
   // threads with a lower number.  Linux supports up to 99, but
    // the POSIX standard only guarantees up to 32 different
   // priority levels.

   // Note: If you are not using pthreads, use the
   // sched_setscheduler call instead.
    if (sched_setscheduler(0, SCHED_FIFO, & sp) != 0) {
          perror("sched_setscheduler");
          exit(EXIT_FAILURE);
    }

clock_gettime(CLOCK_MONOTONIC, &ts);

while(1) {
    int beat = 0;

    // Get a new sample from the ADC and store it in the data structuresjjjj
    sample();

    // get current signal power
    char power = (char)round(SquarAvData.last->val);

    // get color data from the web interface to the FPGA
    char colors[3];
    FILE *colorFile = fopen("/home/pi/.webcolors.txt", "r");
       fscanf(colorFile, "%6x", (unsigned int*)colors);
    fclose(colorFile);

    // Send current signal power and color data
    sendColorsPower(colors[2], colors[1], colors[0], power);

    // check if we are currently at a beat
    if (wait == 0) {
        beat = isBeat();

        if (beat) {
            wait = WAITTIME;
           }
       }
    else {
        wait--;
```

```
            }
        // let the FPGA know if we found a beat
        digitalWrite(ISBEATPIN, beat);

        // Delay until the next sample should be taken
            sleep_until(&ts, 500000);
    }
}

// Collects data from the ADC and stores it in the data structures
void sample(void) {
    // Collect data
    int newAudio = 380 + getAudioSample();

    // store data
    storeSample(newAudio);
}


// System timing functionality taken from:
// http://www.isy.liu.se/edu/kurs/TSEA81/lecture_linux_realtime.html
// Adds "delay" nanoseconds to timespecs and sleeps until that time
static void sleep_until(struct timespec *ts, int delay)
{

       ts->tv_nsec += delay;
       if(ts->tv_nsec >= 1000*1000*1000){
              ts->tv_nsec -= 1000*1000*1000;
              ts->tv_sec++;
       }
       clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, ts,  NULL);
}

// Author: James Palmer
void playSong(void) {
    system("aplay actualMusic.wav &");
}
```

Appendix B: Web Interface

colorTransfer.c

```
// colorTransfer.c
// wballiet@g.hmc.edu 19 Nov 2016
//
// Saves the user input color to a file to be sent to the FPGA


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

#include <stdio.h>
#include <stdlib.h>

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

int main(void) {
  char* len = getenv("CONTENT_LENGTH");
  char colors[*len];

  // Read in the colors from stdin and skips the junk (assumes values are posted)
  fgets(colors, *len, stdin);

  // Write the new colors to a file
  FILE *fp = fopen("/home/pi/.webcolors.txt", "w+");
  fputc(colors[9], fp);
  fputc(colors[10], fp);
  fputc(colors[11], fp);
  fputc(colors[12], fp);
  fputc(colors[13], fp);
  fputc(colors[14], fp);

  fclose(fp);

  // HTML header
  printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);

  // Redirect to the color picker with no delay
  printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/index.html\">");
  return 0;
}
```

Index.html (Depends on Farbastic packages and JQuery)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
 <title>The Wall</title>
 <script type="text/javascript" src="jquery.js"></script>
 <script type="text/javascript" src="farbtastic.js"></script>
 <link rel="stylesheet" href="farbtastic.css" type="text/css" />
 <script type="text/javascript" charset="utf-8">
  $(document).ready(function() {
    $('#demo').hide();
    $('#picker').farbtastic('#color');
  });
 </script>
</head>
<body>
<h1>Pick a Primary LED Color</h1>

<div id="demo" style="color: red; font-size: 1.4em">jQuery.js is not present. You must install
jQuery in this folder for the demo to work.</div>

<form action="cgi-bin/colorTransfer" method="POST" style="width: 400px;">
  <div class="form-item"><label for="color">Color:</label><input type="text" id="color"
name="color" value="#123456" /></div><div id="picker"></div>
  <input type="submit" value="Update Color" />
</form>

</body>
</html>
```

Appendix C: Matlab Beat Frequency Calculation Code

```
function [bpm, sigma, expectedNextBeat] = bpmestimation(toas, curTime)
%Inputs: toas:An array of the time for pulses
%         curTime: The current time
%Outputs: bpm: The calculated beats per minute
%         sigma: an uncertainty in our estimate of the next beat
%         expectedNextBeat: the time when the next beat is expected to occur
%beats are missed
nbins=50;
len = length(toas);
numNeighbours = 5;
difs = zeros(len-1,numNeighbours);

%calculate the differences of each pulse with its preceding neighbours
for i = 1:length(toas)
    for j = 1:numNeighbours
        if(i-j) >= 1
            dif = toas(i)-toas(i-j);
            difs(i,j) = dif;
        end
    end
end

%find the standard deviation of the time between pulses
%this is a decent indicator of how well the beat detection algorithm can do
firstDifVar = std(difs(:,1));

%the range in seconds we expect the period of a beat to be
lowPeriod = 0.25;
highPeriod = 1.25;

%edges of the histogram bins
edges = lowPeriod:(highPeriod-lowPeriod)/(nbins):highPeriod;

%create the histogram that hasn't been smoothed
[oldbins,edges] = histcounts(difs(:),edges);

%smooth the histogram by averaging it with its neighbours
bins = oldbins;
for i = 2:length(oldbins)-1
    bins(i) = sum(oldbins(i-1:i+1));
end

%handle the case where no data in the proper range has been passed in
if max(bins) == 0
    bpm = 0;
    sigma = 0;
    expectedNextBeat = 0;
```

```
    return
end

%find the maximum bin which corresponds to the period
pribin = find(max(bins)==bins,1);

%find the period values that correspond to the bins
lower = edges(max(pribin-1,1));
upper = edges(min(pribin+2, length(edges)));

%calculate the expected period and corresponding bpm
topPRI = (lower+upper)/2;
bpm = 1/topPRI;

%find all the differences that are in the acceptable range to the
%calculated beat period
%Then find the last such beat, use that as the last beat
%This data can be used more heavily for predicting future beats better
targets = difs<=upper & difs>=lower;
toaind = find(targets,1,'last');
beatNum = mod(toaind,len)+1;
lastToa = toas(beatNum); %get time of the last beat


%uncertainty is related to how long it has been since the middle toa as if
%there is a song switch for example, then it will stop finding new beats,
%but we don't want to just reset to no uncertainty once we have found a
%single beat.
%playing with how far back we look to calculate uncertainty can help with
%transitions between songs with looking less far back allowing for
%smoother, rapider transitions.
sigma = abs((((curTime-toas(round(len/2)))/topPRI) /(len/2)))*firstDifVar;

%estimate the next beat to me a multiple of the beat period greater than
%the last beat
expectedNextBeat = lastToa + topPRI - curTime;
while(expectedNextBeat + sigma  < 0)
    expectedNextBeat = expectedNextBeat + topPRI;
End
```

Appendix D: FPGA Code

finalProject.sv

```
module finalProject (input logic clk, sdi, sclk, notLoad, isBeat,
                                                            output logic [11:0] rgbSig,
                                                            output logic [3:0] enables);
    // Hill Balliet - wballiet@g.hmc.edu - 11/20/16
    // This is the top level module and it takes in microphone data and color
    // data from a SPI connection and then powers an LED matrix as a
    // visualization of the microphone data.
        //
        // the bits of rgbSig correspond to {r18, g18, b18, r27, g27, b27, r36, g36, b36, r45,
g45, b45} from MSB to LSB
        // the bits of enables correspond to {EN18, EN27, EN36, EN45} from MSB to LSB
        logic [7:0] red, green, blue, power;
        logic [11:0] notRGBSig;
        logic load;

        assign load = ~notLoad;

    spiColPowIn spiColPow(sclk, sdi, load, clk, red, green, blue, power);
        // For debugging with the test LEDs
    //arrayDriver leds(red, green, blue, power, clk, isBeat, notRGBSig, testLEDs, enables);
        arrayDriver leds(red, green, blue, power, clk, isBeat, notRGBSig, enables);

        // arrayDriver works for common anode LED arrays, but we are using a common cathode
LED array
        assign rgbSig = ~notRGBSig;
endmodule

module spiColPowIn(input logic sck, sdi, load, clk,
                    output logic [7:0] red, green, blue, power);
    // Hill Balliet - wballiet@g.hmc.edu - 11/20/16
    // This module creates an SPI slave to take in color and power data from the
    // Raspberry Pi.

    logic [7:0] presyncR, presyncG, presyncB, presyncP, syncR, syncG, syncB, syncP;


    always_ff @(posedge sck)
        if (load) begin
                    presyncR[7:1] <= presyncR[6:0];
                    presyncR[0] <= presyncG[7];
                    presyncG[7:1] <= presyncG[6:0];
                    presyncG [0] <= presyncB[7];
                    presyncB[7:1] <= presyncB[6:0];
                    presyncB[0] <= presyncP[7];
                    presyncP[7:1] <= presyncP[6:0];
                    presyncP[0] <= sdi;
```

```
		end

	// Synchronize the input from SPI
	always_ff @(posedge clk) begin
		red <= syncR;
		green <= syncG;
		blue <= syncB;
		power <= syncP;
		syncR <= presyncR;
		syncG <= presyncG;
		syncB <= presyncB;
		syncP <= presyncP;
	end

endmodule
```

LEDs.sv

```
// This is the current setup for which LED on the array is of which type
// The indexes on the columns indicate which rgb group it corresponds to
// The indexes on the rows indicate which enable group it corresponds to
//
// P = varies with signal power
// X = flashes on the beat
// 2 = flashes at 1/2x the beat
//
//   1 2 3 4 4 3 2 1
// 1 P P P P P P P P
// 2 P P P X X P P P
// 3 P P X 2 2 X P P
// 4 P X 2 X X 2 X P
// 4 P X 2 X X 2 X P
// 3 P P X 2 2 X P P
// 2 P P P X X P P P
// 1 P P P P P P P P


module arrayDriver (input logic [7:0] red, green, blue, sigPower,
                                            input logic clk, isBeat,
                                            output logic [11:0]rgbSig,
                                            //output logic [2:0] testLEDs,
                                            output logic [3:0] enables);
       // Hill Balliet - wballiet@g.hmc.edu - 11/19/16
       // This module brings all of the logic together for driving an 8x8 LED array
       // in the pattern shown above with the given color
       //
       // rgbSig is composed of:
       // {r18, g18, b18, r27, g27, b27, r36, g36, b36, r45, g45, b45}
       //
       // enables is composed of:
       // {EN18, EN27, EN36, EN45}

       logic [23:0] color18, color27, color36, color45, colorPTest, colorBTest, colorTwoTest;
       logic [23:0] colors;
       logic [16:0] counter;

       // Init to 0 for simulation
       initial
              counter = 0;

       assign colors = {red, green, blue};

       setColors color1(colors, sigPower, counter[16:15], 2'b00, isBeat, clk, color18);
       setColors color2(colors, sigPower, counter[16:15], 2'b01, isBeat, clk, color27);
       setColors color3(colors, sigPower, counter[16:15], 2'b10, isBeat, clk, color36);
       setColors color4(colors, sigPower, counter[16:15], 2'b11, isBeat, clk, color45);
```

```
	// For debugging
	//setColors pTest({8'hFF, 16'b0}, sigPower, 2'b00, 2'b00, isBeat, clk, colorPTest);
	//setColors bTest({8'hFF, 16'b0}, sigPower, 2'b11, 2'b11, isBeat, clk, colorBTest);
	//setColors twoTest({8'hFF, 16'b0}, sigPower, 2'b10, 2'b11, isBeat, clk, colorTwoTest);

	pwm rgb18(color18, clk, rgbSig[11:9]);
	pwm rgb27(color27, clk, rgbSig[8:6]);
	pwm rgb36(color36, clk, rgbSig[5:3]);
	pwm rgb45(color45, clk, rgbSig[2:0]);

	//assign testLEDs = {colorPTest[23], colorBTest[23], colorTwoTest[23]};


	// Always have exactly one group of enables on and all other groups off
	// If two rows are equal, their enables are grouped
	// Since we are using PNP transistors, grounding the base corresponds to saturating the
	// transistor, so the enable bits are flipped
	always_comb
		case (counter[16:15])
			2'b00: enables = ~4'b1000; // Rows 1 and 8
			2'b01: enables = ~4'b0100; // Rows 2 and 7
			2'b10: enables = ~4'b0010; // Rows 3 and 6
			2'b11: enables = ~4'b0001; // Rows 4 and 5
		endcase


	// Count clock cycles to reduce the 40 MHz clk to ~152Hz
	always_ff @(posedge clk)
		counter++;

endmodule

module pwm(input logic [23:0] digColors,
			input logic clk,
			output logic [2:0] pwmColors);
	// Hill Balliet - wballiet@g.hmc.edu - 11/21/16
	// This module converts the digital representations of colors to a pulse
	// width modulation "analog" signal.  It linearly scales the digital
	// representation between the minimum on voltage for the LED and maximum
	// output voltage of 3.3V.

	logic [7:0] rmin, gmin, bmin, rchange, gchange, bchange;
	logic [7:0] rvolt, gvolt, bvolt;
	logic [15:0] rvoltlong, gvoltlong, bvoltlong;
	logic [11:0] counter, toDutyCycle;
	logic [19:0] ronTime, gonTime, bonTime;


	// Init to 0 for simulation
	initial
```

```
        counter = 0;

    assign rmin = 8'h0;
    assign gmin = 8'h0;
    assign bmin = 8'h0;

    // These are hard coded rather than found by subtraction so that leading
    // zeros do not mess up the floating point multiplication
    // slope should be (max-min)/255
// division by 255 is implied since in binary it is a bit shift
    // Maximums are calibrated to show white when all colors are maxed
    // rmax = 2.50V = 10_100000
    // gmax = 3.08V = 11_000101
    // bmax = 3.19V = 11_001100
    assign rchange = 8'b10_100000;
    assign gchange = 8'b11_000101;
    assign bchange = 8'b11_001100;

    // which is the scaling factor to get from V to on time
    assign toDutyCycle = 12'b010011011000; // 4095/3.3 = 1240

    // Calculate the correct voltage required to create the color
    // Assuming the brightness of the LED varies linearly within the range where
    // it is on, the slope will be (max-min)/255.
    //
    // 1/255 is approx 0.000000001 in binary so has been simplified out of the
    // calculations to save space
    //
    // the slope ({8'h00, (max-min)/255}) will be of the form 16'b00.000000xxxxxxxx
    // the color is of the form 16'h00XX
    // Therefore slope * color is of the form 16'hXX.xxxxxxxxxxxxxx
    //
    // the offset (min) is of the form 8'bXX.xxxxxx
    always_comb begin
        rvoltlong = {8'h00, rchange} * {8'h00, digColors[23:16]} + {rmin, 8'h00};
        gvoltlong = ({8'h00, gchange} * {8'h00, digColors[15:8]}) + {gmin, 8'h00};
        bvoltlong = ({8'h00, bchange} * {8'h00, digColors[7:0]}) + {bmin, 8'h00};
    end

    assign rvolt = rvoltlong[15:8];
    assign gvolt = gvoltlong[15:8];
    assign bvolt = bvoltlong[15:8];

    // Convert the desired voltage to a duty cycle by determining the ratio
    // of the desired voltage to the maximum voltage.
    // The duty cycle is calculated as V/3.3*4095
    //
    // voltage is in the form 20'b000000000000XX.xxxxxx
    // toDutyCycle is 20'h00XXX
    // their product is in the form 18'bXXXX XXXX XXXX.xxxxxx
```

```
	// Because the max value of voltage is bounded by 3.3, the product will not overflow
	always_comb begin
		ronTime = {12'b0, rvolt} * {8'b0, toDutyCycle};
		gonTime = {12'b0, gvolt} * {8'b0, toDutyCycle};
		bonTime = {12'b0, bvolt} * {8'b0, toDutyCycle};
	end

	// Use a counter to create three square waves with the given duty cycles
	// for each of the LED colors.
	always_ff @(posedge clk) begin
		if (counter <= ronTime[19:8]) pwmColors[2] <= 1;
		else pwmColors[2] <= 0;

		if (counter <= gonTime[19:8]) pwmColors[1] <= 1;
		else pwmColors[1] <= 0;

		if (counter <= bonTime[19:8]) pwmColors[0] <= 1;
		else pwmColors[0] <= 0;

		counter++;
	end

endmodule

module setColors(input logic [23:0] baseColor,
                                    input logic [7:0] sigPower,
                                    input logic [1:0] row, col,
                                    input logic isBeat, clk,
                                    output logic [23:0] color);

	// Hill Balliet - wballiet@g.hmc.edu - 11/20/16
    // This module outputs a digital representation of the color that should
    // be displayed at a given row and column based on the following rules.
    //
    // P type LEDs use the baseColor
    // P type LEDs are scaled by sigPower/maxSigPower to determine
	// brightness values
	//
    // X type LEDs use the gb values of baseColor but r = r + 20
    // X type LEDs flash when isBeat goes from 0 to 1
    //
    // 2 type LEDs use the gb values of baseColor but r = r + 50
    // 2 type LEDs flash when is2Beat goes from 0 to 1

    logic is2Beat, wasBeat;
    logic [1:0] beatCount, multCount;
    logic [21:0] hold;
    logic [15:0] rfloat, gfloat, bfloat, product, multiplicand;

    enum {pType, xType, twoType} ledType;
```

```
// Count beats so that every other beat is2Beat turns on
    // Init to 0 for simulation
    initial begin
          beatCount = 0;
          hold = 0;
          multCount = 0;
    end

    always_ff @(posedge clk)
          if (multCount == 2'b10) multCount = 2'b00;
          else multCount++;

    always_ff @(posedge clk)
          wasBeat <= isBeat;

always_ff @(posedge clk)
          if (!wasBeat && isBeat) beatCount <= beatCount + 2'b10;


assign is2Beat = beatCount[1];

// Keep X and 2 type LEDs on for 0.1 seconds
// Note that beats are guaranteed to be more than 0.1 seconds apart due to
// the nature of the beat finding algorithm
always_ff @(posedge clk)
   if (isBeat) hold <= 22'h3D0900;
   else if (hold > 0) hold--;

// Determine the current LED type by looking up the current row and col in
// the table at the top of LED type by position in the matrix.
   always_comb
          case ({row, col})
                 4'b0111: ledType = xType;
                 4'b1010: ledType = xType;
                 4'b1011: ledType = twoType;
                 4'b1101: ledType = xType;
                 4'b1110: ledType = twoType;
                 4'b1111: ledType = xType;
                 default: ledType = pType;
          endcase

          assign product = multiplicand * {8'b0, sigPower};

          // Multiplex the multiplication hardware because there isn't enough space
          always_ff @(posedge clk)
                 if (ledType == pType) begin
                        if (multCount == 2'b00) begin
                               rfloat <= product;
                               multiplicand <= {8'b0, baseColor[15:8]};
```

```
				end
				else if (multCount == 2'b01) begin
					gfloat <= product;
					multiplicand <= {8'b0, baseColor[7:0]};
				end
				else begin
					bfloat <= product;
					multiplicand <= {8'b0, baseColor[23:16]};
				end
			end
			else begin
				rfloat <= 0;
				gfloat <= 0;
				bfloat <= 0;
			end

	// Apply the color rules
	always_comb
		case (ledType)
			pType: begin
				// Do floating point multiplication to scale the colors
				// maxSigPower is approx 0.00000001 in bin so has been
				// simplified out of the calculations
				// sigPower is 16'b00.xx and baseColor is 16'b00XX so the result
				// will be 16'bXX.xx
				//rfloat = {8'b0, baseColor[23:16]}  * {8'b0, sigPower};
				//gfloat = {8'b0, baseColor[15:8]}  * {8'b0, sigPower};
				//bfloat = {8'b0, baseColor[7:0]}  * {8'b0, sigPower};

				color[23:16] = rfloat[15:8];
				color[15:8] = gfloat[15:8];
				color[7:0] = bfloat[15:8];
			end
			xType: begin
				if (hold > 0) begin
						color = ~baseColor;
					end
					else begin
						color = 24'h0000;
					end
			end
			twoType: begin
					// Switch between color 2 and color 3 at every beat.
					// Color 2 is found by rotating baseColor by 1 byte
					// Color 3 is found by rotating baseColor by 2 bytes
				if (hold > 0 && is2Beat) begin
						color[23:16] = baseColor[7:0];
						color[15:8] = baseColor[23:16];
						color[7:0] = baseColor[15:8];
					end
```

```
                                        else if (hold > 0) begin
                                                color[23:16] = baseColor[15:8];
                                                color[15:8] = baseColor[7:0];
                                                color[7:0] = baseColor[23:16];
                                        end
                                        else begin
                                                color = 24'h0000;
                                        end
                                end
                        endcase

endmodule
```