Musical Shoes

Final Report
December 7, 2006

Nathaniel Schlossberg and John Parker

Abstract

To encourage toddlers to practice walking and develop rhythm, the team developed a pair of musical shoes to play a song, note-by-note on each step. Embedded pressure sensors in the shoes provide data to a PIC microcontroller to determine when a shoe step has occurred. At each step, the PIC outputs the next note of the song for a set duration to the FPGA. The FPGA decodes the note and outputs an oscillating signal corresponding to the note's frequency. The oscillating signal is passed through an audio amplifier before reaching the speakers in the shoes; the shoe that steps plays the note. The system successfully identifies a shoe step and plays the corresponding note greater than 95% of the time. The notes can be heard clearly in a quiet room at approximately 4 meters away.

Introduction

Following the trend of toys designed to help children grow and develop, the team has designed a product to encourage toddlers to practice walking and develop rhythm: musical shoes. While wearing these shoes, users will be able to play music as they walk or tap their feet. With each step, the shoes will play the next note in the song, so that the user can hear cheerful music as s/he walks, while reinforcing rhythm. The user can choose from among 4 different songs, in case one particular song gets too repetitive.

The entire device was constructed at Harvey Mudd College in the Fall of 2006. It consists of a pressure sensor and speaker in each shoe (2 in total), an audio amplifier, 2 toggle switches, and the Harrisboard PIC microcontroller and FPGA. A block diagram of the entire system can be found in Figure 1.

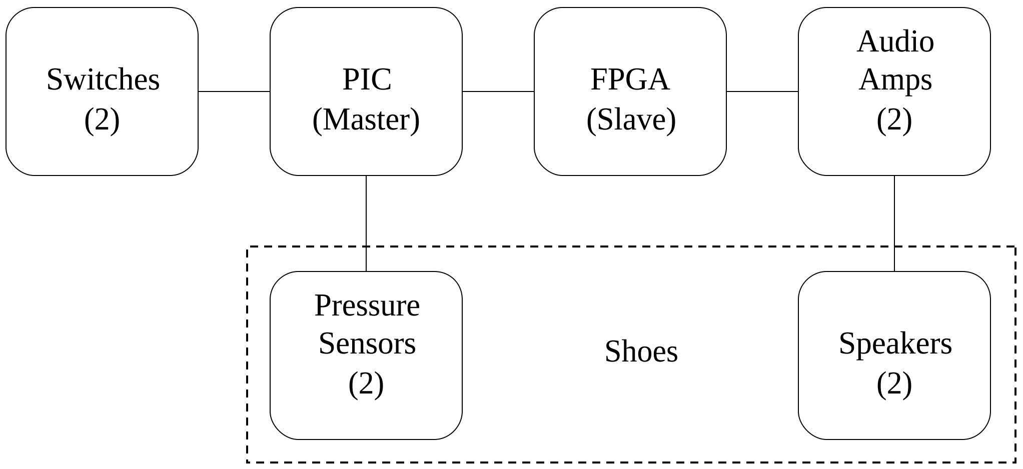

Figure 1: System Block Diagram

The two toggle switches allow the user to select from 4 different songs, which are encoded in the PIC. The pressure sensors act as resistors that change in value when the user steps down. The pressure sensors are used in a voltage divider circuit that feeds into an A/D port in the PIC. The PIC reads the voltage from the pressure sensor. When it recognizes a rising edge in pressure, signifying a step, it serially sends the next encoded note of the current song to the FPGA. The FPGA receives the encoded note and drives the shoe speakers to oscillate at the necessary frequency. Before reaching the speakers, the output runs through an audio amplifier to provide more power to the speakers than the FPGA could by itself.

New Devices

Two Motorola MC31119P audio amps were used to amplify the signals from the FPGA before they reached the speakers. As Figure 2 shows, the audio signal from the FPGA board is fed into the "Audio input," and the speakers are attached to V_{O1} and V_{O2}.

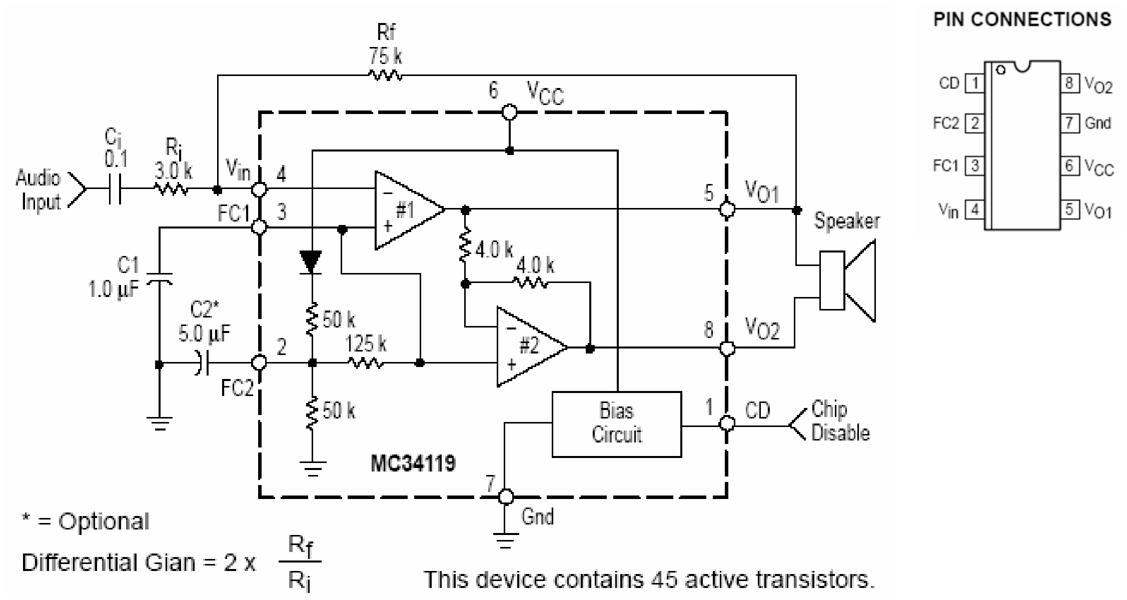

Figure 2: Differential Audio Amplifier Schematic MC34119P
The manufacturer's schematic for the standard layout of the low-power audio amplifiers.[1]

External Hardware

As shown in Figure 3, the two sensors are connected in a pull-up network such that when sufficient pressure occurs, the sensor resistance will decrease and the voltage at the input channel will increase. Two PIC ports are used as digital inputs for the song switch values (RA2, RA4). The figure illustrates that the FPGA has two output pins for the left (P5) and right speaker (P6), and a reset pin (P1). The FPGA is attached to an audio amplifier to boost the power to speaker.

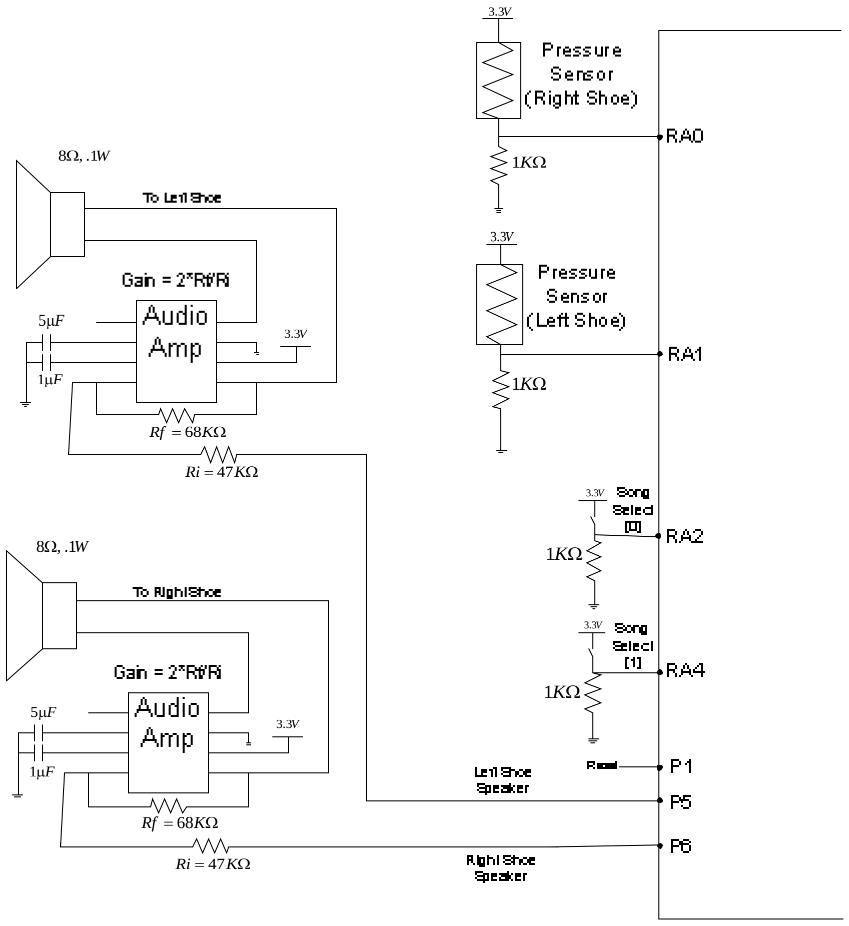

Figure 3: External Hardware

The external hardware shoes the schematic of the external hardware for the musical shoes. The audio amplifier was set with a differential gain of ~3. The speakers are located within the outer heel padding on each shoe. The pressure sensors are situated underneath the removable sole near the ball of the foot.

One sensor and one speaker is placed in each shoe, with their wires channeled through the shoe's padding and out a side air-hole. The speakers draw up to 100 mW and have a resistance of $8\,\Omega$. The amplifier outputs a voltage of $\frac{Vcc - 0.7}{2}$, this means V_{cc} has a maximum of 3.8 V, so we conveniently used the 3.3V output from our board to power the amplifier.

PIC Microprocessor

The PIC program, "MusicShoes.c," was coded in C and compiled with MPLAB C18.[2-4] The program outputs the next note of the selected song to the FPGA on each sensor press. The notes are read from one of four arrays at each sensor trigger. Each array has two eight-bit columns: the first stores the encoded note and the second stores its play duration. The arrays are stored in data memory, creating a limit of 127 notes per song (to keep the size below the 256 byte page size). When the PIC reads a null (0x00) value for the note, the song is restarted.

To determine when a shoe trigger occurs, the microprocessor converts two analog signals from the shoe sensors to digital signals, using empirically determined threshold values. Each of the sensors has a tailored threshold due to small differences between the shoe placements. The song is selected via two digitally input switches.

The program operates using two interrupts, one high priority for timer overflow, and one low priority for A/D conversion completion. The sixteen-bit timer is set to run for 16.67 ms (corresponding to 60 Hz polling) by presetting the timer register to 0xFEBA and using a scale factor of 1/256 at the 20 MHz external clock speed. The timer allows the program to check the triggers status at 60 Hz, which debounces the triggers. The high priority interrupt function updates the status of each shoe's present and previous pressure to determine whether a shoe step has occurred. In addition, the high priority interrupt function restarts the A/D process. The low priority interrupt function, called at the end of each A/D completion, records the current value of the A/D register and restarts the A/D process after switching the input channel.

The code for the PIC can be found in Appendix A. The PIC outputs the notes, via SPI, to the FPGA with an encoding to designate the shoe speaker on which to play.

Note Encoding

Each musical note has its own distinct sound frequency. When a speaker is driven to oscillate high and low at a note's particular frequency, it will play that note. In order to drive the speakers at the correct oscillation, the FPGA slows down its internal clock by the necessary amount, running a counter to a certain period and oscillating the output bit with each reset of the counter. The period is calculated from the 20 MHz clock speed, divided by twice the frequency of the note (since one cycle consists of a high and low). Frequencies and counter periods for each note can be found in Appendix C.

Most songs contain notes that span over multiple octaves. Our device plays notes spanning over three octaves. Increasing the octave of a note corresponds to doubling its frequency. Therefore, the FPGA encodes periods for the highest octave, and left-shifts if necessary to multiply the note's period by two or four (to decrease the octave by one or two).

There are twelve notes in an octave, which require 4 bits to express. Since typical octaves in music begin with C (middle C beginning octave 4), the notes are encoded in numbers 0000 (C) through 1011 (B). Any other 4-bit numbers (1100-1111) default to C. Since there are three octaves, two bits describe the octave of the note. The lowest octave (highest period) is encoded as 00, 01 corresponds to the middle octave, and 10 and 11 correspond to the high octave. Finally, one play bit encodes whether to play a note or to rest, and one bit determines which shoe speaker to play (left or right). This totals 8-bits of information, which made the serial port an easy medium to transfer note encodings.

FPGA

The FPGA accepts the 8-bit note encoding from the PIC and oscillates the speakers at the corresponding frequency. It consists of four main modules: a serial-to-parallel shift register, a note decoder, octave left-shifters and multiplexer, and a speaker driver. The shift register allows the FPGA to receive the 8-bit encoding sent from the PIC and send it to the other modules. The note decoder accepts bits [3:0], which contain the encoded note, and reads the proper 17-bit period from a lookup table, sending it to the octave modules. The two octave left-shifters multiply the period by 2 and 4 respectively, creating the two additional octaves. The 3-way multiplexer inputs the 2 octave bits, [5:4], and decides which period to input into the driver. The driver uses the 20 MHz clock to count up to the given period value and invert the speaker's output bit at each reset. Two 3-input AND gates accept this output bit, the "play" bit (bit 6), and either the "shoe bit" (bit 7) or the inverse of the shoe bit, so that only one shoe plays at a time. Verilog Code for the FPGA and a top-level schematic can be found in Appendix B.

Results

The most difficult aspect of the project arose when trying to capture foot presses from a wide range of movements. Three different settings similarly influence the capture of the shoe steps:

1. The polling frequency determines how often the shoe's pressure data is analyzed for presses and releases. A polling frequency set too high causes bounce on each foot step, and when set too low the system misses rapid foot steps.
2. The trigger threshold set on the Analog to Digital Converter determines the voltage values for the analog signal that would correspond to high and to low. The trigger needs to be set lower than the voltage at a shoe step, and higher than the non-step voltage when a user raises their foot or sits in a chair with their feet at rest.
3. The sensor placement determines the voltage range for the pressure values and the non-step pressure value when the user rests their foot. The ideal sensor configuration provides the widest range of pressure values between non-step to step and where ambient foot movement has little effect.

The voltage ranges were calculated for several types of foot movement, including tapping while sitting, stepping on tip-toes, walking, and hopping. At the optimal setting, all of these foot steps were accommodated for, except hopping where the pressure on the sensor remains at a high voltage for the duration of the hop; this is due to the foot pressing against the sole of the shoe mid-air, rather than solely when in contact with the floor. The hopping was accommodated in one test by raising the threshold value; however, proper capture of the other foot steps was lost. Therefore we decided not to incorporate hopping into the normal use of the device.

For the main shoe movements (tapping while sitting, stepping on tip-toes, and walking), the PIC trigger-sensing system on average receives one false foot press for each song, corresponding to successful note play of greater than 95% of the time. The speaker can be heard clearly up to 4 meters away. The hardware in the shoe does not obstruct the user’s movement or stepping comfort. Future work will seek to embed the PIC, audio amps, FPGA, switches, and a battery within the shoes, with wireless communication to make the shoes more versatile.

References

[1] Motorola. ***Low Power Audio Amplifier: MC34119,***

http://www.datasheetcatalog.com/datasheets_pdf/M/C/3/4/MC34119.shtml

[2] Microchip. ***MPLAB C18 C Compiler: User's Guide.*** **15 November 2006.**

http://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB_C18_Users_Guide_51288j.pdf

[3] Microchip. ***MPLAB C18 C Compiler: Libraries.*** **15 November 2006.**

http://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB_C18_Libraries_51297f.pdf

[4] Microchip. ***MPLAB C18 C Compiler: Getting Started.*** **15 November 2006.**

http://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB_C18_Getting_Started_51295f.pdf

Parts List

Part	Source	Vendor Part #	Price	Quantity
Speaker	Radio Shack	273-093	$2.59	2
Wire (22 gauge stranded)	Radio Shack	2781218	$4.99	1
Used Shoes	Goodwill		$10	1 pair

Appendix A: PIC Shoe Triggering Code in C

```
#include<p18f452.h>
#include<timers.h>

/*Function Prototypes*/
void main(void);
void AtoD(void);
void CheckShoes(void);

/*Global Variables*/
char duration;          //time for note to be played
char shoeRprev;                //prior value from A/D
char shoeLprev;                //prior value from A/D
char shoeLpres;                //current value from A/D
char shoeRpres;                //current value from A/D
char note;                     //note encoding
char shoeLtrigger;      //0xff is trigger-on (send note), 0x00 is trigger off (do not send
note)
char shoeRtrigger;      //shoe right trigger

#pragma udata song1 = 0x100 //defines 256 byte uninitialized data memory block for song1
char song1[0x7F][0x2];     //arrays are declared to be less than max banksize 256 bytes
                                 //2*127 bytes = 254 bytes
                                 //errors occur when, array size equals max banksize

#pragma udata song2 = 0x200 //defines 256 byte uninitialized data memory block for song2
char song2[0x7F][0x2];

#pragma udata song3 = 0x300 //song3
char song3[0x7F][0x2];

#pragma udata song4 = 0x400 //song4
char song4[0x7F][0x2];

char r;                     //song [r][c]
char shold;                        //Holds Select value to determine when select is
changed

//MPLAB C compiler provides the jump command at 0x00 to main
#pragma code low_vector = 0x18 //insert jump to low interrupt at program memory 0x18
void low_interrupt(void)      {
        _asm
                goto AtoD
        _endasm
}
#pragma code high_vector = 0x08 //insert jump to high interrupt at program memory 0x08
void interrupt(void)   {
        _asm
                goto CheckShoes
        _endasm
}
#pragma code start = 0x100
void main(void) {
        int temp = 0x00;               //initialize variables
        unsigned int i;                //variable for loop
        unsigned char d;               //variable for loop
        note = 0x0f;           //default note value (no note playing)

        //initial values
        shoeLprev       = 0x01;        //no previous shoe values at start
        shoeRprev       = 0x01;        //no previous shoe values at start
        shoeLtrigger = 0x00;   //no trigger have been recorded
        shoeRtrigger = 0x00;   //no trigger have been recorded
        shoeLpres       = 0x00;        //default value 0x00 until a note is played
        shoeRpres       = 0x00;        //default value 0x00 until a note is played


        //write song to array => Why do you build me up, buttercup baby
```

```
	//page length is 256 bytes so array (song) max length is 127 notes
	song1[0x0][0]=0x60;//note
	song1[0x0][1]=0x10;//duration: 0x10 is 1/8th note, 0x20 is quarter note, 0x40 half
note etc
	song1[0x1][0]=0x59;
	song1[0x1][1]=0x10;
	song1[0x2][0]=0x57;
	song1[0x2][1]=0x10;
	song1[0x3][0]=0x55;
	song1[0x3][1]=0x10;
	song1[0x4][0]=0x54;
	song1[0x4][1]=0x10;
	song1[0x5][0]=0x54;
	song1[0x5][1]=0x10;
	song1[0x6][0]=0x55;
	song1[0x6][1]=0x10;
	song1[0x7][0]=0x54;
	song1[0x7][1]=0x10;
	song1[0x8][0]=0x54;
	song1[0x8][1]=0x10;
	song1[0x9][0]=0x52;
	song1[0x9][1]=0x10;
	song1[0xa][0]=0x54;
	song1[0xa][1]=0x10;
	song1[0xb][0]=0x54;
	song1[0xb][1]=0x10;
	song1[0xc][0]=0x52;
	song1[0xc][1]=0x10;
	song1[0xd][0]=0x50;
	song1[0xd][1]=0x10;
	song1[0xe][0]=0x50;
	song1[0xe][1]=0x10;
	song1[0xf][0]=0x47;
	song1[0xf][1]=0x10;
	song1[0x10][0]=0x55;
	song1[0x10][1]=0x10;
	song1[0x11][0]=0x55;
	song1[0x11][1]=0x10;
	song1[0x12][0]=0x55;
	song1[0x12][1]=0x10;
	song1[0x13][0]=0x54;
	song1[0x13][1]=0x10;
	song1[0x14][0]=0x52;
	song1[0x14][1]=0x10;
	song1[0x15][0]=0x50;
	song1[0x15][1]=0x10;
	song1[0x16][0]=0x55;
	song1[0x16][1]=0x10;
	song1[0x17][0]=0x54;
	song1[0x17][1]=0x10;
	song1[0x18][0]=0x54;
	song1[0x18][1]=0x10;
	song1[0x19][0]=0x47;
	song1[0x19][1]=0x10;
	song1[0x1A][0]=0x55;
	song1[0x1A][1]=0x10;
	song1[0x1B][0]=0x54;
	song1[0x1B][1]=0x10;
	song1[0x1C][0]=0x54;
	song1[0x1C][1]=0x10;
	song1[0x1D][0]=0x52;
	song1[0x1D][1]=0x10;
	song1[0x1E][0]=0x52;
	song1[0x1E][1]=0x10;
	song1[0x1F][0]=0x54;
	song1[0x1F][1]=0x10;
	song1[0x20][0]=0x52;
	song1[0x20][1]=0x10;
	song1[0x21][0]=0x50;
	song1[0x21][1]=0x10;
	song1[0x22][0]=0x50;
```

```
song1[0x22][1]=0x10;
song1[0x23][0]=0x47;
song1[0x23][1]=0x10;
song1[0x24][0]=0x55;
song1[0x24][1]=0x10;
song1[0x25][0]=0x55;
song1[0x25][1]=0x10;
song1[0x26][0]=0x55;
song1[0x26][1]=0x10;
song1[0x27][0]=0x54;
song1[0x27][1]=0x10;
song1[0x28][0]=0x54;
song1[0x28][1]=0x10;
song1[0x29][0]=0x55;
song1[0x29][1]=0x10;
song1[0x2A][0]=0x57;
song1[0x2A][1]=0x10;
song1[0x2B][0]=0x54;
song1[0x2B][1]=0x10;
song1[0x2C][0]=0x55;
song1[0x2C][1]=0x10;
song1[0x2D][0]=0x57;
song1[0x2D][1]=0x10;
song1[0x2E][0]=0x54;
song1[0x2E][1]=0x10;
song1[0x2F][0]=0x55;
song1[0x2F][1]=0x10;
song1[0x30][0]=0x57;
song1[0x30][1]=0x10;
song1[0x31][0]=0x59;
song1[0x31][1]=0x10;
song1[0x32][0]=0x57;
song1[0x32][1]=0x10;
song1[0x33][0]=0x55;
song1[0x33][1]=0x10;
song1[0x34][0]=0x54;
song1[0x34][1]=0x10;
song1[0x35][0]=0x55;
song1[0x35][1]=0x10;
song1[0x36][0]=0x55;
song1[0x36][1]=0x10;
song1[0x37][0]=0x55;
song1[0x37][1]=0x10;
song1[0x38][0]=0x54;
song1[0x38][1]=0x10;
song1[0x39][0]=0x52;
song1[0x39][1]=0x10;
song1[0x3A][0]=0x50;
song1[0x3A][1]=0x20;
song1[0x3B][0]=0x47;
song1[0x3B][1]=0x10;
song1[0x3C][0]=0x55;
song1[0x3C][1]=0x10;
song1[0x3D][0]=0x54;
song1[0x3D][1]=0x10;
song1[0x3E][0]=0x54;
song1[0x3E][1]=0x10;
song1[0x3F][0]=0x54;
song1[0x3F][1]=0x10;
song1[0x40][0]=0x52;
song1[0x40][1]=0x10;
song1[0x41][0]=0x52;
song1[0x41][1]=0x10;
song1[0x42][0]=0x52;
song1[0x42][1]=0x10;
song1[0x43][0]=0x50;
song1[0x43][1]=0x10;
song1[0x44][0]=0x4B;
song1[0x44][1]=0x10;
song1[0x45][0]=0x50;
song1[0x45][1]=0x20;
```

```
song1[0x46][0]=0x00;//end of song, null value for note
song1[0x46][1]=0x00;//end of song


//write song   => Tetris Music
song2[0x0][0]=0x64;//note
song2[0x0][1]=0x10;//duration
song2[0x1][0]=0x5B;
song2[0x1][1]=0x10;
song2[0x2][0]=0x60;
song2[0x2][1]=0x10;
song2[0x3][0]=0x62;
song2[0x3][1]=0x10;
song2[0x4][0]=0x60;
song2[0x4][1]=0x10;
song2[0x5][0]=0x5B;
song2[0x5][1]=0x10;
song2[0x6][0]=0x59;
song2[0x6][1]=0x10;
song2[0x7][0]=0x59;
song2[0x7][1]=0x10;
song2[0x8][0]=0x60;
song2[0x8][1]=0x10;
song2[0x9][0]=0x64;
song2[0x9][1]=0x10;
song2[0xa][0]=0x62;
song2[0xa][1]=0x10;
song2[0xb][0]=0x60;
song2[0xb][1]=0x10;
song2[0xc][0]=0x5B;
song2[0xc][1]=0x10;
song2[0xd][0]=0x5B;
song2[0xd][1]=0x10;
song2[0xe][0]=0x60;
song2[0xe][1]=0x10;
song2[0xf][0]=0x62;
song2[0xf][1]=0x10;
song2[0x10][0]=0x64;
song2[0x10][1]=0x10;
song2[0x11][0]=0x60;
song2[0x11][1]=0x10;
song2[0x12][0]=0x59;
song2[0x12][1]=0x10;
song2[0x13][0]=0x59;
song2[0x13][1]=0x10;
song2[0x14][0]=0x62;
song2[0x14][1]=0x10;
song2[0x15][0]=0x62;
song2[0x15][1]=0x10;
song2[0x16][0]=0x65;
song2[0x16][1]=0x10;
song2[0x17][0]=0x69;
song2[0x17][1]=0x10;
song2[0x18][0]=0x67;
song2[0x18][1]=0x10;
song2[0x19][0]=0x65;
song2[0x19][1]=0x10;
song2[0x1A][0]=0x64;
song2[0x1A][1]=0x10;
song2[0x1B][0]=0x64;
song2[0x1B][1]=0x10;
song2[0x1C][0]=0x60;
song2[0x1C][1]=0x10;
song2[0x1D][0]=0x64;
song2[0x1D][1]=0x10;
song2[0x1E][0]=0x62;
song2[0x1E][1]=0x10;
song2[0x1F][0]=0x60;
song2[0x1F][1]=0x10;
song2[0x20][0]=0x5B;
song2[0x20][1]=0x10;
```

```
song2[0x21][0]=0x5B;
song2[0x21][1]=0x10;
song2[0x22][0]=0x60;
song2[0x22][1]=0x10;
song2[0x23][0]=0x62;
song2[0x23][1]=0x10;
song2[0x24][0]=0x64;
song2[0x24][1]=0x10;
song2[0x25][0]=0x60;
song2[0x25][1]=0x10;
song2[0x26][0]=0x59;
song2[0x26][1]=0x10;
song2[0x27][0]=0x59;
song2[0x27][1]=0x10;
song2[0x28][0]=0x00;//end of song, null value for note
song2[0x28][1]=0x00;//end of song

//write song   => Darth Vader Music
song3[0x0][0]=0x47;//note
song3[0x0][1]=0x10;//duration
song3[0x1][0]=0x47;
song3[0x1][1]=0x10;
song3[0x2][0]=0x47;
song3[0x2][1]=0x10;
song3[0x3][0]=0x43;
song3[0x3][1]=0x10;
song3[0x4][0]=0x4A;
song3[0x4][1]=0x10;
song3[0x5][0]=0x47;
song3[0x5][1]=0x10;
song3[0x6][0]=0x43;
song3[0x6][1]=0x10;
song3[0x7][0]=0x4A;
song3[0x7][1]=0x10;
song3[0x8][0]=0x47;
song3[0x8][1]=0x10;
song3[0x9][0]=0x52;
song3[0x9][1]=0x10;
song3[0xa][0]=0x52;
song3[0xa][1]=0x10;
song3[0xb][0]=0x52;
song3[0xb][1]=0x10;
song3[0xc][0]=0x53;
song3[0xc][1]=0x10;
song3[0xd][0]=0x4A;
song3[0xd][1]=0x10;
song3[0xe][0]=0x46;
song3[0xe][1]=0x10;
song3[0xf][0]=0x43;
song3[0xf][1]=0x10;
song3[0x10][0]=0x4A;
song3[0x10][1]=0x10;
song3[0x11][0]=0x47;
song3[0x11][1]=0x10;
song3[0x12][0]=0x57;
song3[0x12][1]=0x10;
song3[0x13][0]=0x47;
song3[0x13][1]=0x10;
song3[0x14][0]=0x57;
song3[0x14][1]=0x10;
song3[0x15][0]=0x56;
song3[0x15][1]=0x10;
song3[0x16][0]=0x55;
song3[0x16][1]=0x10;
song3[0x17][0]=0x54;
song3[0x17][1]=0x10;
song3[0x18][0]=0x53;
song3[0x18][1]=0x10;
song3[0x19][0]=0x54;
song3[0x19][1]=0x10;
song3[0x1A][0]=0x48;
```

```
song3[0x1A][1]=0x10;
song3[0x1B][0]=0x51;
song3[0x1B][1]=0x10;
song3[0x1C][0]=0x50;
song3[0x1C][1]=0x10;
song3[0x1D][0]=0x4B;
song3[0x1D][1]=0x10;
song3[0x1E][0]=0x4A;
song3[0x1E][1]=0x10;
song3[0x1F][0]=0x49;
song3[0x1F][1]=0x10;
song3[0x20][0]=0x4A;
song3[0x20][1]=0x10;
song3[0x21][0]=0x43;
song3[0x21][1]=0x10;
song3[0x22][0]=0x46;
song3[0x22][1]=0x10;
song3[0x23][0]=0x43;
song3[0x23][1]=0x10;
song3[0x24][0]=0x4A;
song3[0x24][1]=0x10;
song3[0x25][0]=0x47;
song3[0x25][1]=0x10;
song3[0x26][0]=0x43;
song3[0x26][1]=0x10;
song3[0x27][0]=0x4A;
song3[0x27][1]=0x10;
song3[0x28][0]=0x47;
song3[0x28][1]=0x10;
song3[0x29][0]=0x00;//end of song, null value for note
song3[0x29][1]=0x00;//end of song

//write song   => Hall of the Mountain King
song4[0x0][0]=0x4B;//note
song4[0x0][1]=0x10;//duration
song4[0x1][0]=0x51;
song4[0x1][1]=0x10;
song4[0x2][0]=0x52;
song4[0x2][1]=0x10;
song4[0x3][0]=0x54;
song4[0x3][1]=0x10;
song4[0x4][0]=0x56;
song4[0x4][1]=0x10;
song4[0x5][0]=0x52;
song4[0x5][1]=0x10;
song4[0x6][0]=0x56;
song4[0x6][1]=0x10;
song4[0x7][0]=0x55;
song4[0x7][1]=0x10;
song4[0x8][0]=0x51;
song4[0x8][1]=0x10;
song4[0x9][0]=0x55;
song4[0x9][1]=0x10;
song4[0xa][0]=0x54;
song4[0xa][1]=0x10;
song4[0xb][0]=0x50;
song4[0xb][1]=0x10;
song4[0xc][0]=0x54;
song4[0xc][1]=0x10;
song4[0xd][0]=0x4B;
song4[0xd][1]=0x10;
song4[0xe][0]=0x51;
song4[0xe][1]=0x10;
song4[0xf][0]=0x52;
song4[0xf][1]=0x10;
song4[0x10][0]=0x54;
song4[0x10][1]=0x10;
song4[0x11][0]=0x56;
song4[0x11][1]=0x10;
song4[0x12][0]=0x52;
song4[0x12][1]=0x10;
```

```
        song4[0x13][0]=0x56;
        song4[0x13][1]=0x10;
        song4[0x14][0]=0x5B;
        song4[0x14][1]=0x10;
        song4[0x15][0]=0x59;
        song4[0x15][1]=0x10;
        song4[0x16][0]=0x56;
        song4[0x16][1]=0x10;
        song4[0x17][0]=0x52;
        song4[0x17][1]=0x10;
        song4[0x18][0]=0x56;
        song4[0x18][1]=0x10;
        song4[0x19][0]=0x59;
        song4[0x19][1]=0x10;
        song4[0x1A][0]=0x00;//end of song, null value for note
        song4[0x1A][1]=0x00;//end of song

        TRISC = 0x00;                   //Port C is an output port to the FPGA
        TRISA = 0x17;                   //Port A has four input ports, 2 from A/D, 2 digital
for song select[1:0]
                                                //RA0, RA1 are set analog, and RA2, RA4 are
set digital



        ADCON1 = 0x04;                  //A/D with 3 analog, 5 digital and internal voltage
references
        ADCON0 = 0x81;                  //turns on A/D
        /*wait 33 cycles (12 microsecs) for A/D required acquisition time*/
        for(i=0x0; i < 0x21; i++)       {//33 in decimal = 0x21
                temp += i;                      //delay variable
        }

        PIR1bits.ADIF           = 0x0;  //clears A/D interrupt flag
        PIE1bits.ADIE           = 0x1;  //set up A/D interrupt
        RCONbits.IPEN           = 0x1;  //enables priority levels on interrupts
        INTCONbits.GIE                  = 0x1;  //set up interrupts
        INTCONbits.PEIE                 = 0x1;
        INTCONbits.TMR0IE       = 0x1;  //enable timer interrupt
        INTCON2bits.TMR0IP      = 0x1;  //set timer0 as high priority interrupt
        IPR1bits.ADIP           = 0x0;  //set A/D as low priority interrupt
        ADCON0bits.GO           = 0x1;  //Run A/D

        //start timer
        TMR0H = 0xFE;//set timer high value
        TMR0L = 0xBA;//set timer low
        //timer set to 65210 so that timer interrupts at 60hz
        T0CON = 0x87;//16bit counter with 1/256 prescale and start counter
        while(1)        {//runs continuously except when interrupts occur
                r = 0x00;//restarts song
                while (note != 0x00)    {//runs until table ends and then loops
                        //set up SPI
                        SSPCON1 = 0x22;                 //config sspcon
                        SSPSTATbits.SMP = 0x1; //config sspstat
                        SSPSTATbits.CKE = 0x1;          //config sspstat
                        if(shoeRtrigger || shoeLtrigger)        {
                                //set for loop time equal to duration
                                for(d=0x00; d < duration; d++)          {
                                        for(i=0x0000; i < 0x08ff; i++)          {
                                        //scales-up each duration increment
                                                if (shoeRtrigger)       {
                                                        shoeRtrigger = 0x00;
        //reset trigger
                                                        shoeLtrigger = 0x00;
        //double triggers will result in music only played on the right shoe
                                                        note = note | 0x80;
        //accesses the MSB makes it 1 (=rightshoe)
                                                        SSPBUF = note;
        //output note data to FPGA
                                                        SSPCON1 = 0x22;
        //config sspcon
```

```
                                                    SSPSTATbits.SMP = 0x1;
        //config sspstat
                                                    SSPSTATbits.CKE = 0x1;
        //config sspstat
                                                    d=0x00;
        //resets duration count if triggered again
                                            }
                                            else if(shoeLtrigger)  {//shoe left trigger
                                                    shoeLtrigger = 0x00;
        //reset trigger
                                                    SSPBUF = note;
        //output note data to FPGA
                                                    SSPCON1 = 0x22;
        //config sspcon
                                                    SSPSTATbits.SMP = 0x1;
        //config sspstat
                                                    SSPSTATbits.CKE = 0x1;
        //config sspstat
                                                    d = 0x00;
        //reset duration counter if triggered again
                                            }
                                    }
                            }
                            SSPBUF = 0x0F;//stop playing music after duration is met
                    }
            }
        }
}
#pragma interruptlow AtoD             //interrupt when A/D is finished
void AtoD(void)         {
        char i;
        char temp;
        temp = 0x0;
        if (ADCON0bits.CHS0 == 0x0)   {       //if A/D right shoes was just run
                shoeRpres = ADRESH;           //shoeR is assigned to A/D value
                ADCON0bits.ADON       = 0x0;  //turn A/D off
                ADCON0bits.CHS0       = 0x1;  //change to channel 1
                ADCON0bits.ADON       = 0x1;  //resets turn on A/D
                PIR1bits.ADIF  = 0x0;  //clear A/D interrupt flag
                PIE1bits.ADIE  = 0x1;  //set up A/D interrupt
                INTCONbits.GIE        = 0x1;  //set up interrupts
                INTCONbits.PEIE       = 0x1;
                temp = 0x0;
                for (i=0x0; i < 0x21; i++)    {//waits for A/D to acquire channel
                        temp = temp + 0x1;
                }
                ADCON0bits.GO = 0x1;//Run A/D left shoe
        }
        else    {//if A/D left shoes was just run
                shoeLpres = ADRESH;                   //shoeL is assigned to A/D value
                //does not clear A/D interrupt flag until A/D is run again
                ADCON0bits.ADON       = 0x0;          //turn A/D off
                ADCON0bits.CHS0       = 0x0;          //change to channel 0
                ADCON0bits.ADON       = 0x1;                  //resets turn on A/D
                PIR1bits.ADIF         = 0x0;          //clear A/D interrupt flag
                PIE1bits.ADIE         = 0x1;  //set up A/D interrupt
                INTCONbits.GIE                = 0x1;  //set up interrupts
                INTCONbits.PEIE               = 0x1;
        }
}

#pragma interrupt CheckShoes//timer interrupt
void CheckShoes(void)  {
        char s; //select song bit
        s = PORTA & 0x14; //0x00 for "song 1", 0x04 for "song 2", 0x10 for "song 3", 0x14
for "song 4"
        if (s != shold)       {
                r = 0x00;
        }
        shold = s;
        INTCONbits.TMR0IE = 0x1;//re-enable timer
```

```
	TMR0H = 0xFE;//set timer high value
	TMR0L = 0xBA;//set timer low
	INTCONbits.TMR0IF = 0x0;//reset timer interrupt flag
	T0CON = 0x87;//16 bit counter go
	if (shoeRpres > 0xC0)  {//threshold is 0xB0
		if (shoeRprev == 0x00)	{//shoe was not already pressed
			shoeRtrigger = 0xff;
			if (s == 0x00) {
				note = song1[r][0x0];//note read
				duration = song1[r][0x1];//duration read
			}
			else if (s == 0x04)	{
				note = song2[r][0x0];  //note read
				duration = song2[r][0x1];	//duration read
			}
			else if (s == 0x10)	{
				note = song3[r][0x0];  //note read
				duration = song3[r][0x1];	//duration read
			}
			else	{
				note = song4[r][0x0];  //note read
				duration = song4[r][0x1];	//duration read
			}
			r++;	//moves to the next note
		}
		else	{	//shoe was already pressed
			shoeRtrigger = 0x00;
		}
		shoeRprev = 0x01;//sets the current state to on
	}
	else {
		shoeRtrigger = 0x00;
		shoeRprev = 0x00;
	}
	if (shoeLpres > 0x90)  {	//threshold was empirically determined and set
		if (shoeLprev == 0x00)	{	//shoe was not already trigger
			shoeLtrigger = 0xff;
			if (shoeRtrigger == 0x00)	{
			//we check that right shoe was not trigger to avoid pulling two 
notes at the same time
				if (s == 0x00) {
					note = song1[r][0x0];//note read
					duration = song1[r][0x1];//duration read
				}
				else if (s == 0x04)	{
					note = song2[r][0x0];//note read
					duration = song2[r][0x1];//duration read
				}
				else if (s == 0x10)	{
					note = song3[r][0x0];//note read
					duration = song3[r][0x1];//duration read
				}
				else	{
					note = song4[r][0x0];//note read
					duration = song4[r][0x1];//duration read
				}
				r++;//moves to the next note
			}
		}
		else	{	//shoe was already triggered
			shoeLtrigger = 0x0;
		}
		shoeLprev = 0x01;	//sets the current state to on
	}
	else	{	//shoeLpres <= 0x90
		shoeLtrigger = 0x00;
		shoeLprev = 0x00;
	}

	/*	Can only accept one trigger at a time.
		The right shoe has priority, and if the user jumps with both feet
		only a note on the right shoe will play
	*/

	if (note == 0x00)	{//if the end of the song has been reached
		r=0x00;		//repeat song
	}
	PIR1bits.ADIF = 0x0;//clear A/D interrupt flag
	ADCON0bits.GO = 0x1;//Run A/D for shoeR
}
```

Appendix B: FPGA Code and Schematics for Note Decoder/Speaker Driver

(SEE NEXT PAGE)

Appendix C: Musical Note Encoding

Note	Frequency (Hz)	Divider index = 20 MHz / Freq / 2
C_5	523.25	19111
$C^{\#}_5/D^b_5$	554.37	18038
D_5	587.33	17026
$D^{\#}_5/E^b_5$	622.25	16070
E_5	659.26	15168
F_5	698.46	14317
$F^{\#}_5/G^b_5$	739.99	13513
G_5	783.99	12755
$G^{\#}_5/A^b_5$	830.61	12039
A_5	880	11363
$A^{\#}_5/B^b_5$	932.33	10725
B_5	987.77	10123

*For lower octaves (3 and 4), divider indices are shifted by 1 and 2 in hardware