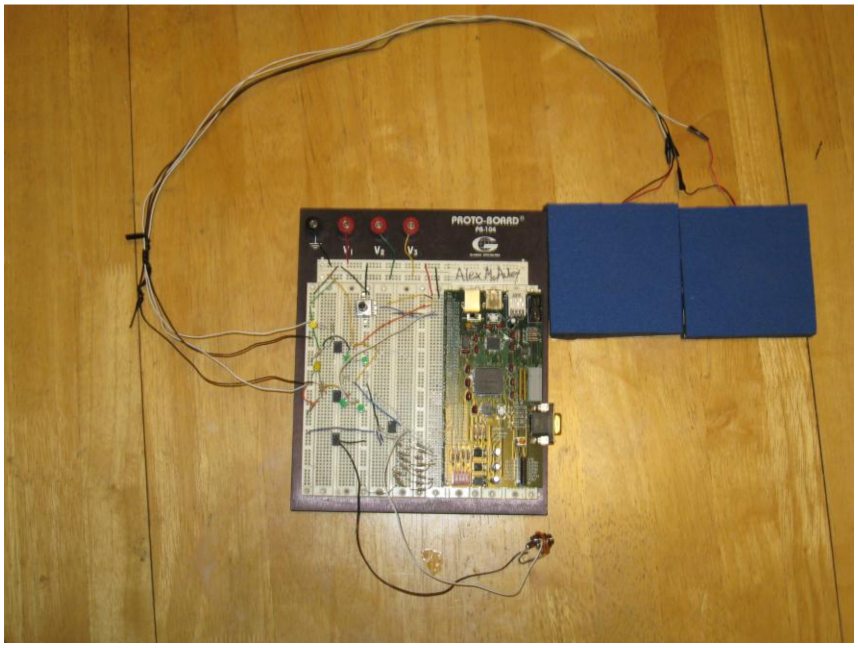
Electronic Drum Kit

Final Project Report
December 10, 2010
E155

Alex McAuley and Tim Nguyen



Abstract

The goal of the project was to create an electronic drum kit using piezoelectric sensors and actual sound samples of various percussion instruments. The inputs to the kit are small drum pads, which house a piezoelelctric transducer. The output voltage from this device is detected and measured through an analog-to-digital converter located on the PIC. The PIC passes volume and drum information through a Serial Peripheral Interface (SPI) connection to the FPGA. This information is processed through an FSM which controls audio playback. In the case that one sound is still playing when another drum hit is detected, the FPGA has the ability to mix the samples together. The project produced a kit that would take inputs from two different pads and produce either a snare drum or a cymbal sound when hit. Future work would involve implementing an EEPROM for additional sound storage, as well as adding additional drums to the drum kit.

Introduction

The team's project is an electronic drum kit. The user hits custom-built "drums" that contain a piezoelectric transducer, and an output waveform will be played on a speaker corresponding to the drum or drum(s) the user hit. For example, hitting the snare "drum" will play back a snare sound over the speaker.

High Level Design

The electronic drum kit was designed to be a platform that would allow playback of multiple drums using inexpensive piezoelectric transducers to detect drum hits. A block diagram of the signal flow is shown below in Figure 1.

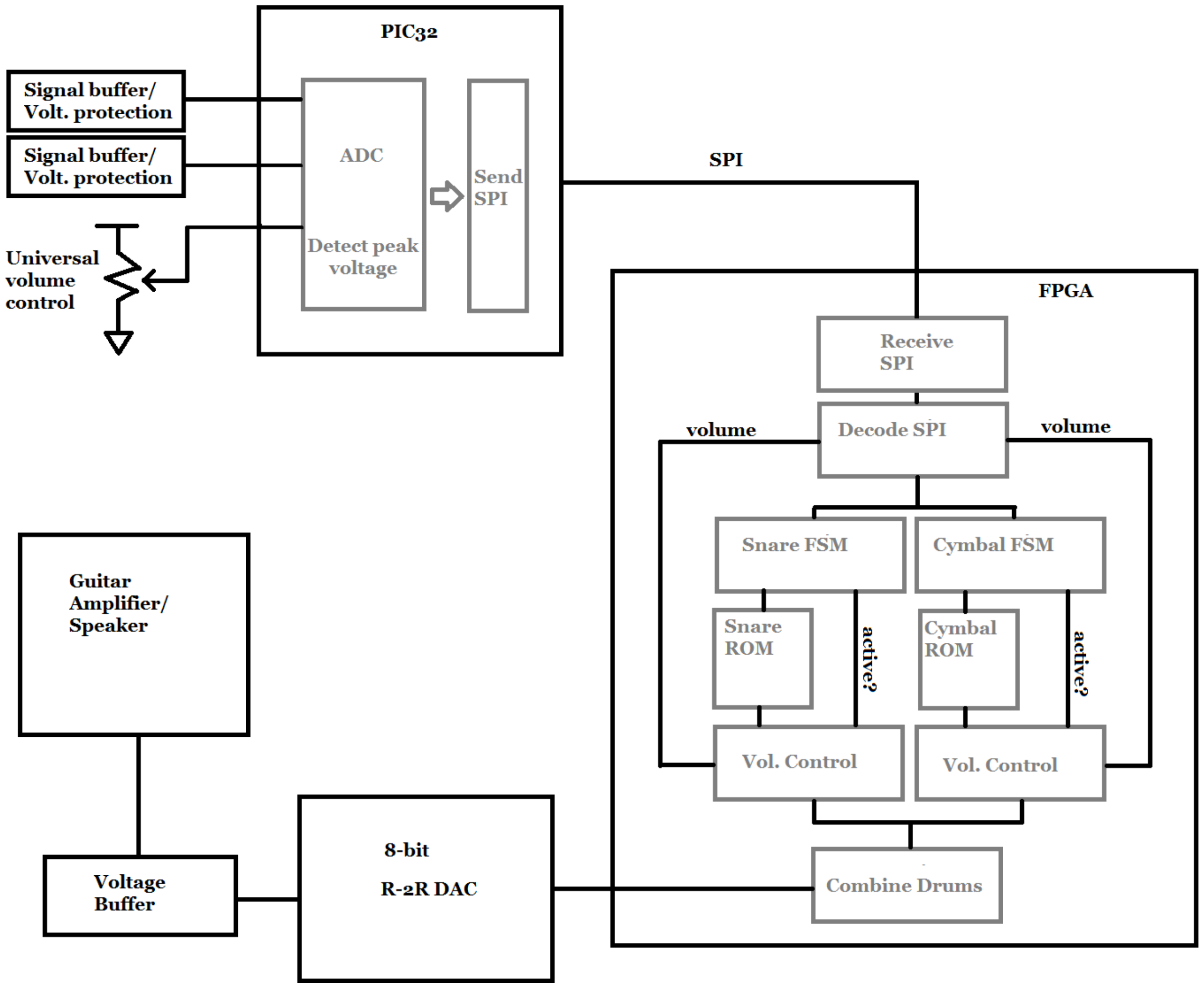


Figure 1. Data flow through the system.

The signal path begins with the piezoelectric transducers and ends with the output speaker. The piezoelectric transducers are connected to a voltage buffer and voltage-limiting LEDs (shown in the schematics in Appendix C.1) in order to safely interface with the PIC's A/D converter. The PIC detects the highest voltage level in the drum hit and relays this information to the FPGA via Serial Peripheral Interface (SPI). Data about the universal volume level set by the potentiometer is also sent in the same SPI transmission. Higher voltage from the transducer corresponds to louder output sound. The FPGA will use this information to adjust the intensity of the output signal sent to the digital-to-analog converter and ultimately the speaker.

Hardware

To generate signals, piezoelectric transducers are connected to a driver circuit consisting of an op-amp, several resistors, and LEDs (Appendix C.1). The LEDs are used to protect the ADC pins for voltages outside their -0.3-5.5V rating. Assuming the LEDs are active for voltages over 2V, the yellow LED will be active whenever the piezoelectric output voltage is below -2V. When this occurs, the LED acts as a low impedance path to ground. Physical testing shows that the piezoelectric transducers are not powerful enough to drive the low impedance of an active LED, and so virtually all of the transducer's power is spent across the LED. In other words, the transducer cannot drive both the active LED and the voltage divider network, so the voltage divider only experiences a brief pulse of energy at voltages below -2V before the LED activates and shunts the signal power to ground. This means that although the ADC pins might experience voltage spikes outside of the -.3-5.5V range, these pulses are so short lived that no damage is done. Likewise, the LEDs may experience voltages of -30V, but for such a brief time that they are not damaged. The green LEDs are more for decoration; the LM741 is able to drive the ADC pins high even if these LEDs are active. The 91k and 9.1k voltage divider resistors, however, should limit the input ADC voltage to 3V based on the measured peak piezoelectric voltage of 30V. The green LEDs, then, only serve as warning lights should the ADC pin voltage somehow exceed approximately 4V. The colors of the LEDs are unimportant - they simply made visual identification of different parts of the breadboarded circuit easier.

An R-2R resistor ladder digital-to-analog converter was created in order to produce eight-bit sound (Appendix C.2). The output of the resistor ladder is fed into an op amp buffer, and then to an open circuit audio jack. A guitar amplifier is connected to the board in this way, providing an easy method of amplifying the outbound signal. The resistor ladder is made solely of one and two megaohm resistors. These resistors were chosen because they were simple to implement - the larger resistor was twice the rated resistance of the smaller resistor value.

The team attempted, unsuccessfully, to store additional sound samples in an external EEPROM. The EEPROM needed to be programmed by a separate Verilog program, and then checked by another program to make sure that the data was stored correctly. The EEPROM programmer/reader schematic is located in Appendix C.3.

Physical Components

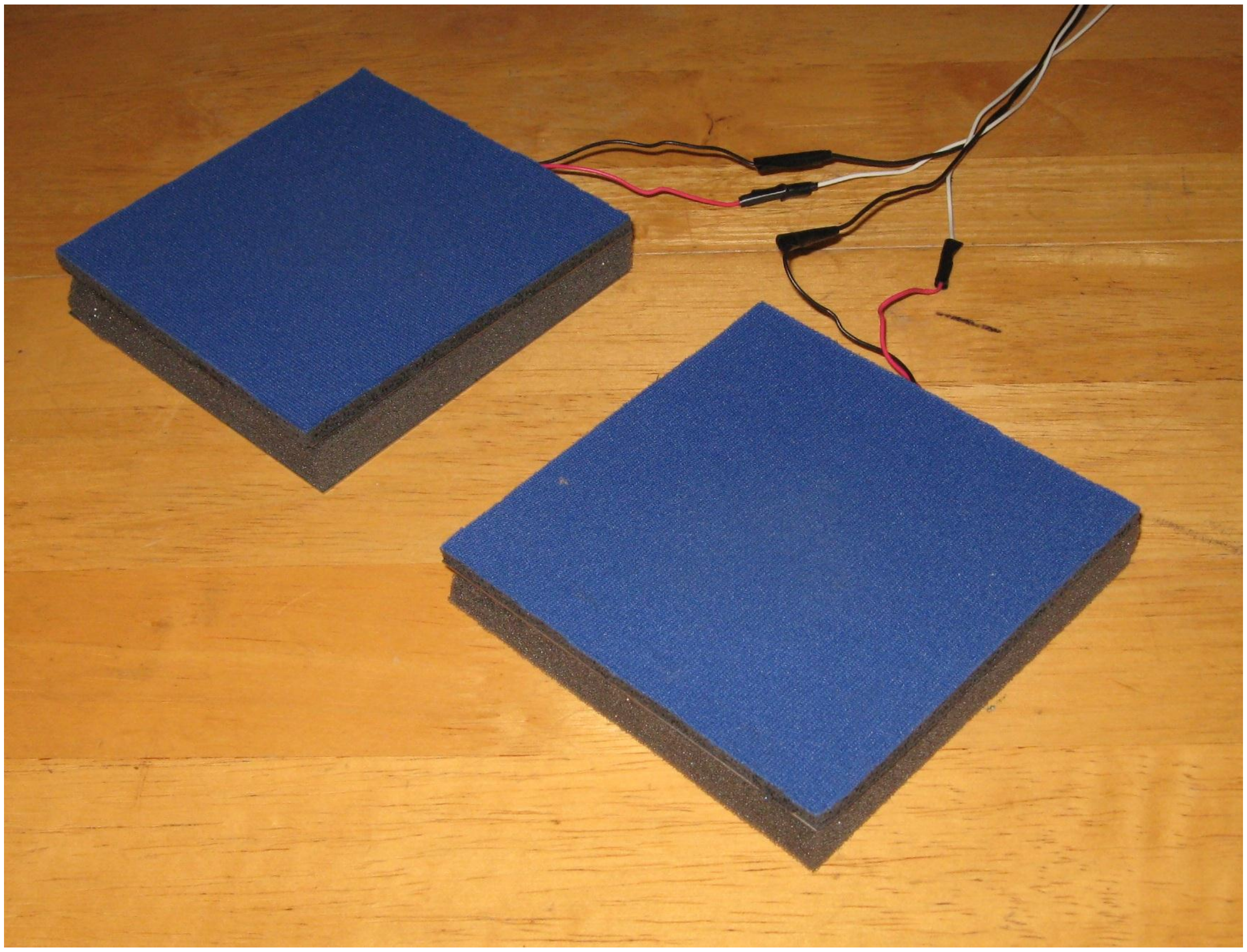


Figure 2. Constructed drum pads

The drum pads were designed to absorb most of the noise and impact energy caused by hitting the tops of the pads. Each layer that is present in the final construction can be seen in Figure 3.

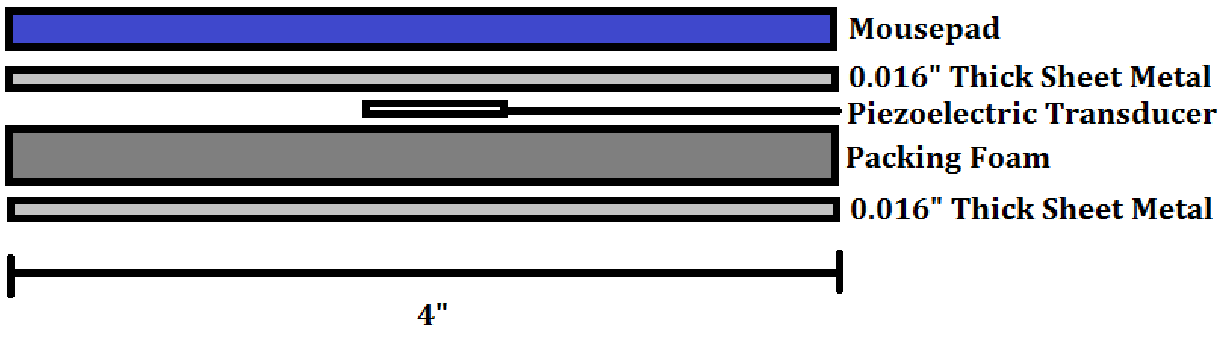


Figure 3. Diagram of drum pad layers

The piezoelectric transducer is epoxied onto a thin layer of sheet metal. This metal layer is then glued onto several different layers. The mousepad and foam layers exist to negate most of the noise and transferred force, respectively. The drum pads can easily be adapted to meet different shapes and sizes. Circular drum pads are one possible variant of the design.

Microcontroller Design

The PIC microcontroller handles configuring the ADC for automatic scanning of the input channels, drum decoding, and SPI communication with the FPGA. The code for the PIC is divided into three main functions: *main*, *spi_init*, and *write_spi*.

main (Appendix B.1) is the highest-level function. It first configures the PIC to use PORTB as an ADC. The ADC checks and scans three different inputs. Two of these inputs come from the drum pads, while the third monitors a potentiometer that is used to set a universal, maximum volume level. The two inputs connected to the drum pads monitor changes in voltage. For each drum hit, the maximum voltage reading is recorded and sent to the FPGA. A hit begins when the voltage level rises above a defined threshold and it ends when the voltage falls below a lower threshold. Hysteresis is thus applied to prevent "double hits" from occurring, which are caused by the piezoelectric voltage reverberating after a hit. After each hit the processor also waits a short amount of time so that reverberations have completely died down. The waiting time was determined through experimentation and is set by the STALL value. In addition to the tasks described, the main function also uses the helper functions *spi_init* and *write_spi* to configure and send data over SPI.

spi_init (Appendix B.2) configures the SPI ports on the PIC for usage. SPI1 (in the location of PORTD) is configured to send data. The PIC is designated as the master, while the FPGA is designated as the slave device. The SPI is configured to send 16-bit transfers - this allows the team to send out information on drum number and volume in one package. The baud rate is 32 times slower than the base clock speed of 40 MHz. This slows the transfer down in order to safeguard the system against any timing errors that might result from using an SPI baud rate that is too large. Other alerts are disabled in order to prevent interruptions.

write_spi (Appendix B.2) sends data from the PIC to the FPGA by writing the contents of the main program into the SPI buffer. Since the output transfer size is 16 bits, the PIC can send which drum is being hit and volume in one data transfer. This makes decoding the PIC signal simple. In the *main* function, the drum number and universal volume is represented in the eight most significant bits, and the volume is transferred in the eight least significant bits.

FPGA Design

The FPGA hardware consists of eight main modules: *top, spi_rcv, drum_decode, playbackFSM, lpm_rom_snare, lpm_rom_cymbal, instr_volume_cntrl,* and *add_sounds*. This section will describe the operation of each module.

top (Appendix A.1) is the top level module of the design. Its inputs are the 40MHz board clock and SPI clock and data connections. The output is the 8-bit value sent to the DAC. Parameters *{snare,cymbal}_duration_limit* and *{snare,cymbal}_addr_limit* define sample playback speed and number of samples stored for each drum.

spi_rcv (Appendix A.2) is the SPI receiver. Since the FPGA does not communicate back to the PIC and because it is the only device being communicated to, there is no need for MISO output or slave select signals. The module expects 16-bit data from the PIC and outputs any new data on the 16th cycle of the SPI clock. The data is output as two 8-bit values corresponding to hit strength and drum ID / universal volume information. The most significant six bits of the received message correspond to the universal drum volume, or the maximum output volume as set by the potentiometer. The next 2 bits correspond to the drum ID, which is the information that tells the FPGA which drum was hit. The last 8 bits correspond to how hard the drum was hit and are also used in determining output volume. The module is clocked at the faster 40MHz board clock to ensure it behaves nicely with the other synchronous modules.

drum_decode (Appendix A.3) takes in the the hit strength information (signal *SPI_volume)* from the received SPI data and the drum ID (signal *drum_num*). It uses the drum ID information to pass the hit strength on to the appropriate playback FSM through signal *to_snare* or *to_cymbal*. The output to each drum is assigned 0 unless that drum is indicated by the drum ID.

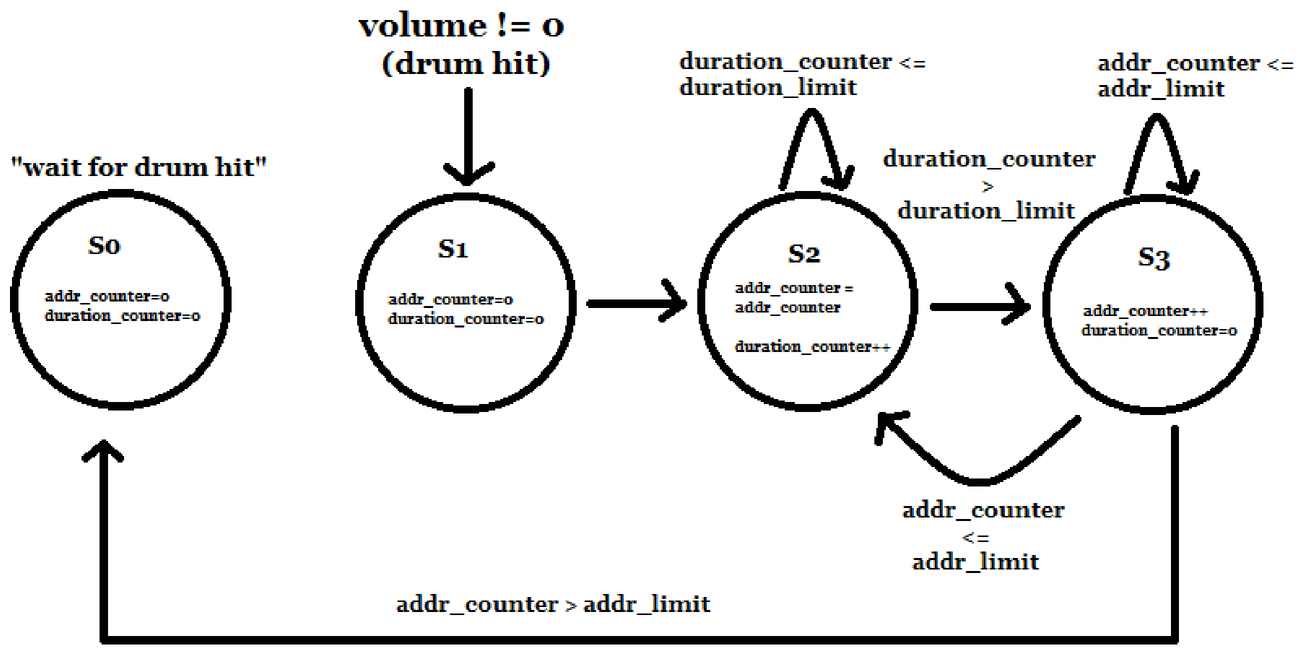


Figure 4. Sound-sample playback FSM

The sound-sample playback FSM, called *playbackFSM* (Appendix A.4), is shown in Figure 4. Each drum has an associated *playbackFSM* module, which is responsible for selecting the correct sound sample to be sent to the volume control modules and eventually the speaker. The FSM waits in state 0 until a drum hit is detected, as indicated by the hit strength signal received over SPI (called *volume* in the module). The FSM then enters state 1, which resets *duration_counter* and *addr_counter* to 0 before transitioning to state 2. In state 2, the FSM increments *duration_counter* until it is greater than input *duration_limit*, which determines how long each sample value should be played for and depends on the sampling rate the sound was recorded at. In state 3, *duration_counter* is reset and *addr_counter* is incremented, thus selecting the next sound sample from memory. If *addr_counter* exceeds input *addr_limit*, all samples have been played, and the FSM transitions back to state 0 where it waits for the next drum hit. The outputs of the module are the address of the sound sample that should be played, *addr*, and the FSM_RUNNING flag which indicates that the FSM is not in waiting state S0. Because the two playback FSMs run independently of one another, it is possible to play both drums at the same time.

lpm_rom_snare and *lpm_rom_cymbal* are modules created by the Quartus II MegaWizard. The Verilog for these modules is not shown because it was generated by Quartus and not written by the team. They are 8-bit synchronous ROMs that hold the sound samples after MATLAB processing. Each one is holds 16k samples, which was the smallest size that could hold the samples. The drum samples were found online (see References section) and converted in MATLAB to the desired 8-bit range (0-255). All unused memory addresses are automatically initialized to zero, although the playback modules should never try to access these addresses in the first place.

instr_volume_cntrl (Appendex A.4) takes in a sound sample and modifies its value based on the universal volume *universal_volume* and hit strength *instr_volume* to produce output *instr_out.* This is accomplished by right-shifting the sample depending on how quiet the volume levels are; the lower the volume, the more the sample is right-shifted. Right shifting is used because it is a fast method of scaling the signal. There are too stages of right-shifting. In the first stage, the sample is shifted from 0-3 bits. The thresholds for each shift amount were determined based on typical hit strength values received over SPI. The next stage of shifting is dependent on the universal volume. Here the thresholds are determined by evenly dividing the range of *universal_volume* values.

playback_active (Appendex A.5) is high when the drum *instr_volume_cntrl* is connected to is playing back samples. This is used so that the universal volume or hit strength readings don't get set to zero in the middle of playback because the SPI receiver only outputs these signals as non-zero for one cycle of the the SPI clock. Without registering the values, they would get set to zero in the middle of playing back a sample.

One of the drawbacks of using 8-bit samples is that the lower volume the samples are shifted to, the less resolution each sample has. For example, if an 8-bit sample is right-shifted by 5 bits, it effectively only has 3 bits of resolution. It is hard to hear this loss of resolution on the working device, however, possibly because the low resolution only occurs at low volumes.

Finally, *add_sounds* (Appendex A.6) takes in the volume-modified samples from the snare and cymbal and combines them into a single output by adding them together. This addition holds the possibility for overflowing the 8 output bits, which would heavily distort the sound, but this has not been a problem in practice. It would be possible to ensure no saturation by right-shifting each input by one bit (dividing it by two), but this would further limit the resolution of the already low resolution sound.

Results and Conclusions

The final deliverable met all of the goals stated in the project proposal, and was successful overall. The two pads could be used to play cymbal and snare drum noises at varying volumes, and the built in volume control could be manipulated to give greater control of audio levels. The different drums could be played at the same time, and the sounds overlapped without overflowing the 8-bit signal to the DAC.

Our optional goal was not met due to lack of time and board issues. The EEPROM was successfully programmed and tested, but implementation of the EEPROM with the existing system proved to be problematic, causing a few board lockups. Code for these processes is located in Appendix A.7 and A.8. Additional drum sources could also be implemented fairly easily with this system; one would simply need to modify the code slightly to accommodate additional devices to scan. If more room was needed to store sound data, data could also be stored on the PIC and transferred over another SPI connection, if necessary.

References

Snare drum samples found on <http://members.tripod.com/artificial_2/samples/drums.htm>.
Cymbal samples found on <http://www.cymbalsonly.com/index.htm>.
Kick drum samples (for EEPROM) found on <http://bigsamples.free.fr/bdkick.html>.

Bill of Materials

Part	Supplier	Part No.	Price/Part	Quantity	Total Price
256k EEPROM	Digi-Key	AT28C256-15PU	$7.37	1	$7.37
Piezo Transducer	RadioShack	273-073	$1.99	2	$3.98
Mousepad	Amazon	F8E089-BLK	$4.75	2	$9.50
1/4” Mono Panel-Mount Audio Jack	Radioshack	274-252	$2.99	1	$2.99
Sheet Metal (0.016” STL)	HMC	N/A	$1.00	1	$1.00
uMudd32 board	HMC	N/A	$75.00	1	$75.00
				Total	$99.84
				Tax/ Shipping	$3.86
				Total	$103.70

Appendices

Appendix A. Verilog Code

A.1. Top Level module

```
/*
	This module instantiates the lower-level modules and sets different parameters
	for playback. It receives information from the PIC via SPI, and outputs sound
	from an eight-bit file to the external DAC.
*/

module top(	input clk,
		input SPI_clk,
		input SPI_in,
		output [7:0] sound_out,
		);

	//44MHz/997 = 441.kHz (audio sample rate)
	parameter snare_duration_limit = 27'd997;
	parameter snare_addr_limit = 14'd11600; //11600 snare samples

	parameter cymbal_duration_limit = 27'd2750; //16kHz audio sample rate
	parameter cymbal_addr_limit = 14'd12986; //# of cymbal samples

	wire [13:0] snare_addr;	//current snare sample
	wire [7:0] snare_volume;

	wire [13:0] cymbal_addr;
	wire [7:0] cymbal_volume;

	wire [7:0] SPI_info;	// universal volume level and drum ID
	wire [7:0] SPI_volume;	//drum hit strength

	wire [7:0] snare_rom_out; //8 bit snare sample value from ROM
	wire [7:0] cymbal_rom_out;//cymbal sample value

	wire [7:0] snare_out;	//snare sample adjusted for volume info
	wire [7:0] cymbal_out;	//adjusted cymbal sample

	wire snare_running; //flag that goes high when snare is playing back samples
	wire cymbal_running;//flag that goes high when cymbal is playing back samples

	//get info from PIC
	spi_rcv SPIrcv (clk, SPI_clk, SPI_in, SPI_info, SPI_volume);

	//decode the SPI info
	drum_decode drum_decoder (clk, SPI_info [1:0], SPI_volume, snare_volume,
					cymbal_volume);
	//in charge of playing back snare samples
	playbackFSM snare_FSM (	clk, snare_volume, snare_duration_limit,
					snare_addr_limit, snare_addr, snare_running);
	//in charge of cymbal samples
	playbackFSM cymbal_FSM (	clk, cymbal_volume, cymbal_duration_limit,
				cymbal_addr_limit, cymbal_addr, cymbal_running);
	//snare sample memory
	lpm_rom_snare snare (snare_addr, clk, snare_rom_out);
	//cymbal sample memory
	lpm_rom_cymbal cymbal (cymbal_addr, clk, cymbal_rom_out);
	 //adjust snare volume
	instr_volume_cntrl snare_vol (	clk, snare_rom_out, snare_volume,
					SPI_info[7:2], snare_running, snare_out);
	//adjust cymbal volume
	instr_volume_cntrl cymbal_vol (	clk, cymbal_rom_out, cymbal_volume,
					SPI_info[7:2], cymbal_running, cymbal_out);
	//combine samples into single output
	add_sounds sound_adder (snare_out, cymbal_out, sound_out);

	always@(posedge clk)

endmodule
```

A.2. SPI Reception Module

```
/*
	This module contains code to receive data over an SPI connection.
	Since we only have one slave SPI device, there is no need to worry about
	configuring several devices for SPI usage, which simplifies things somewhat.
	Data from the PIC comes in 2 bytes:
			MSB [8 bits drum info] [8 bits hit strength] LSB,
	where the drum info contains both the universal volume level and drum ID:
			MSB [6 bits universal volume, 2 bits drum ID] LSB
*/

module spi_rcv(clk, SPI_CLK, SPI_IN, signal_1, signal_2);
input clk;
input			SPI_CLK;
input			SPI_IN;
output	reg [7:0]	signal_1;
output	reg [7:0]	signal_2;

// extra registers for the steps below
reg [15:0]	temp;
reg [3:0]	count;
```

```
// We don't really have to worry about SSEL being a wrong value unless
// we have multiple devices to configure for SPI

// I update a temporary value when I see posedge SPI_CLK
always @ (posedge SPI_CLK)
	begin
		count <= count + 1;
		temp <= temp << 1;
		temp[0] <= SPI_IN;
	end

// I count to 16 and update the output when appropriate
always @ (posedge clk)
	if(count == 0)
	begin
		signal_1 <= temp[15:8];
		signal_2 <= temp[7:0];
	end
	else
	begin
		signal_1 <= 0;
		signal_2 <= 0;
	end

endmodule
```

A.3. Drum Decoder Module

```
/*
	This module determines which drum has been hit from the SPI input. It then
	sends volume and signal to the corresponding FSM.
*/

module drum_decode (input clk,
				input [1:0] drum_num,
				input [7:0] SPI_volume,
				output reg [7:0] to_snare,
				output reg [7:0] to_cymbal);



	always@(posedge clk)
	case(drum_num)
		2'b01: begin //cymbal hit
				to_snare = SPI_volume;
				to_cymbal = 8'h0;
			end
		2'b10 : begin//snare hit
```

```
				to_snare = 8'h0;
				to_cymbal = SPI_volume;
			end
		default: begin
				to_snare = 8'h0;
				to_cymbal = 8'h0;
			end
	endcase

endmodule
```

A.4. Playback FSM Module

```
/*
The playbackFSM module uses an FSM to control the playback of the sound samples.

INPUTS:

	volume: how hard was the drum hit? The FSM starts playing back samples when
		a nonzero volume is read for the drum. The actual magnitue of volume
		is not used by this module.

	duration_limit: how many clock cycles should each sample be held for? This
		value corresponds to the sampling period the drum was recorded at.

	addr_limit: how many samples are in the recorded drum waveform? Stop trying to
		play samples once we've played them all!

OUTPUTS:

	addr: the address of the current sample. This will be used as input to the
		memory in which the samples are stored.

	FSM_RUNNING: the instr_volume_cntrl module needs to know if the drum sample
		is currently being played back.


FSM STATES:

	S0:	the waiting state - stay here until the drum is hit (volume != 0). Don't
		advance through the samples.

	S1:	reset addr_counter and duration_counter - we're just starting to play
		back the sample!

	S2:	advance the duration_counter. It's time to get the next sample when
			duration_counter > duration_limit.
	S3:	get the next sample by incrementing addr_counter. if we've exceeded
		addr_limit, the drum waveform has been played and it's time to return to
		the waiting state, else reset duration_counter and return to state 2.
*/

module playbackFSM (input clk,
				input [7:0] volume,
				input [26:0] duration_limit,
				input [13:0] addr_limit,
				output [13:0] addr,
				output FSM_RUNNING);

	reg [1:0] state, nextstate;
	reg [26:0] duration_counter;
	reg [13:0] addr_counter;

	//define states
	parameter S0 = 2'b00;
	parameter S1 = 2'b01;
	parameter S2 = 2'b10;
	parameter S3 = 2'b11;

	/*state register - start advancing through samples when we get a volume
	signal.
	start playing it again from the beginning even if we're already in the middle
	of playing the waveform - this allows for effects such as a drum-roll! */

	always@(posedge clk)
	begin
		if(volume != 8'b0)
			state <= S1;
		else
			state <= nextstate;
	end


	//duration_counter reg
	always@(posedge clk)
	case(state)
		S0: duration_counter = 0;
		S1: duration_counter = 0;
		S2: duration_counter = duration_counter + 1;
		S3: duration_counter = 0;
	endcase

	//addr_counter reg
	always@(posedge clk)
	case(state)
```

```
		S0: addr_counter = 0;
		S1: addr_counter = 0;
		S2: addr_counter = addr_counter;
		S3: addr_counter = addr_counter + 1;
	endcase

	//nextstate logic
	always@(*)
	case(state)
		S0: nextstate = S0;
		S1: nextstate = S2;
		S2: nextstate = duration_counter > duration_limit ? S3 : S2;
		S3: nextstate = addr_counter > addr_limit ?  S0 : S2;
	endcase

	//output logic
	assign addr = addr_counter;
	assign FSM_RUNNING = state != S0; //not waiting state

endmodule
```

A.5. Volume Control Module

```
/*
	This module takes in the instrument sample and shifts the output value
	according to what the volume values indicate. There are four different volume
	levels, which can be influenced by how hard one hits the drum and also by the
	potentiometer on the board (universal volume). Bit shifting is used to adjust
	volume as it is much faster than multiplication/division.
*/

module instr_volume_cntrl(input clk,
                                              input [7:0] instr_sample,
                                              input [7:0] instr_volume,
                                              input [5:0] universal_volume,
                                              input playback_active,
                                              output reg [7:0] instr_out);

	reg [7:0] volume_reg;
	reg [5:0] univ_vol_reg;
	reg [7:0] instr_vol_out;

	always@(posedge clk)
	case(playback_active)
		0:	begin
				volume_reg <= instr_volume;
				univ_vol_reg <= universal_volume;
```

```
                //not playing back samples - hold on to last volume
        grabbed             //over SPI
        end
    1:  begin
                if(instr_volume != 0 && instr_volume != volume_reg)
                        volume_reg <= instr_volume;
                        //update volume during playback
                else
                        volume_reg <= volume_reg;
                if(    universal_volume != 0 && universal_volume !=
                univ_vol_reg)
                        univ_vol_reg <= universal_volume;
                        //update univeral volume during playback
                else
                        univ_vol_reg <= univ_vol_reg;
        end
endcase

always@(*)
begin
        if (volume_reg < 128)
                instr_vol_out = instr_sample >> 3;
        else if(volume_reg >= 128 && volume_reg < 160)
                instr_vol_out = instr_sample >> 2;
        else if(volume_reg >= 160 && volume_reg < 192)
                instr_vol_out = instr_sample >> 1;
        else
                instr_vol_out = instr_sample;
end

/*left shift sample (decrease volume) based on universal volume val. */
always@(*)
begin
        if (univ_vol_reg < 8)
                instr_out = 8'b0; //mute
        else if(univ_vol_reg >= 8 && univ_vol_reg < 16)
                instr_out = instr_vol_out >> 7;
        else if(univ_vol_reg >= 16 && univ_vol_reg < 24)
                instr_out = instr_vol_out >> 6;
        else if(univ_vol_reg >= 24 && univ_vol_reg < 32)
                instr_out = instr_vol_out >> 5;
        else if(univ_vol_reg >= 32 && univ_vol_reg < 40)
                instr_out = instr_vol_out >> 4;
        else if(univ_vol_reg >= 40 && univ_vol_reg < 48)
                instr_out = instr_vol_out >> 3;
        else if(univ_vol_reg >= 48 && univ_vol_reg < 56)
                instr_out = instr_vol_out >> 2;
        else if(univ_vol_reg >= 56 && univ_vol_reg < 63)
				instr_out = instr_vol_out >> 1;
			else
				instr_out = instr_vol_out; //loudest
		end


endmodule
```

A.6. Sound Adder Module

```
/*
	This module adds the sounds together from separate FSMs. Initially, the output
data had been right shifted in order to prevent output saturation, but testing proved
that this was unnecessary in a two-drum kit.
*/

module add_sounds(
				input [7:0] snare,
				input [7:0] cymbal,
				output[7:0] combined_out);

	wire [8:0] temp;

	assign temp =  snare + cymbal;
	assign combined_out = temp[7:0];//DAC only handles 8 bits

endmodule
```

A.7. Read EEPROM Module

```
/*
This module is used to check what has been programmed onto the EEPROM by checking
addresses in numerical order.
*/

module read_eeprom(
	input clk,
	input [7:0] data_in,
	output [14:0] addr,
	output [7:0] data_out,
	output not_CE,
	output not_OE,
	not_WE
	);

	reg [40:0] counter;

	always@(posedge clk)
```

```
         counter = counter + 1;

        assign addr = counter[40:26];
        assign data_out = data_in; //to LEDs

        assign not_CE = 0;
        assign not_OE = 0;
        assign not_WE = 1;

endmodule
```

A.8. Write EEPROM Module

```
/*
This module writes the contents of kick_rom, a file generated from the sound of a
kick drum, to the EEPROM module. This code can be modified to add any sort of
information to the EEPROM in question.
*/

module write_eeprom(input clk,
                                output not_CE,
                                output reg not_WE,
                                output not_OE,
                                output reg [14:0] addr,
                                output [7:0] data,
                                output reg WRITE_LED,
                                output reg DONE_LED
                                );

        reg [25:0] count;

        parameter addr_limit = 7754; //number of samples to transfer

        always@(posedge clk)
        if(addr <= addr_limit)
        begin
                count <= count + 1;

                if(count == 1000000) // next byte
                begin
                        addr <= addr + 1;
                        count <= 0;
                end
                else
                        addr <= addr;

                if(count > 800000 && count < 1000000) //write byte
                begin
                not_WE <= 0; //latch address
                WRITE_LED <= 1;
            end
            else
            begin
                not_WE <= 1; //latch data
                WRITE_LED <= 0;
            end

            DONE_LED <= 0;
        end

        else
            begin
            not_WE <= 1;
            DONE_LED <= 1;
            end

        assign not_CE = 0;
        assign not_OE = 1;

        kick_rom transfer_rom (addr[12:0], clk, data);

endmodule
```

Appendix B. PIC Code

B.1. ADC Decoder Module/Signal Generator

```
/*
      This module deciphers the analog input from the piezoelectric capacitors and
      determines which drum has been hit.

/*
Alex McAuley,
amcauley@hmc.edu
November 12, 2010
*/

#include <stdio.h>
#include <p32xxxx.h>


#define THRESH 1    //after crossing to below thresh, send max_val to FPGA
#define HYST 10            //HYSTERESIS factor
#define STALL 5000         //diminish piezo rebound if HYST is not enough - value
found                           //through testing

int main(){
/*ADC setup modified from Ex.17-4 of PIC32 Family Reference
      Sect.17 - 10-Bit A/D Converter*/

int ADCVal1;
int ADCVal2;
int ADCVal3;

int count;

int max_val1 = 0;   //maximum value of ADC channel 1 for current drum hit
int max_val2 = 0;   //.............................2.............
int max_update_en1 = 1;   //enable flag for updating max_val1
int max_update_en2 = 1;   //enable flag for updating max_val2

spi_init();

TRISB = 0xFFFF; //input (analog)

AD1PCFG = 0x0000;           //all PORTB = analog
AD1CON1 = 0x00E0;           //auto-scan
AD1CSSL = 0x0034;          //scan RB2, RB4, RB5
```

```
AD1CON3 = 0x0F00;           //sample time = 15*Tad
AD1CON2 = 0x040C;         //interrupt after 3 samples, enable scanning

AD1CON1SET = 0x8000;        //turn on ADC

while(1){
      IFS1CLR = 0x0002;         //clear ADC flag
      AD1CON1SET = 0x0004;            //auto start sampling
      while(!(IFS1 & 0x0002));  //conversion done?
      AD1CON1CLR = 0x0004;            //yes, stop adc
      ADCVal1 = ADC1BUF0;
      ADCVal2 = ADC1BUF1;
      ADCVal3 = ADC1BUF2;

      /*now we have ADC samples, let's monitor them to look for the highest
      input voltage, which we'll store in max_val.*/

      /*    update enable logic + send output. Send data upon transitioning to
            below threshold level. Use hysteresis for threshold because ADC lower
            bits are quite unstable. Data is sent as 16 bits:
            MSB [6 bits universal volume (from ADC3)],
                [2 bits drum ID number (based on which channel input is from)],
                [8 bits hit strength (value read from ADC channel)] LSB
            10 bit ADC values are bit shifted to fit these fields
      */
      if (ADCVal1 < (max_update_en1 ? THRESH : HYST*THRESH)){
            if(max_update_en1){
                  write_spi(1 | (ADCVal3>>4)<<2,max_val1<<2);
                  write_spi(0,0);     //clear spi buffer so FPGA starts playback
                  max_val1 = 0;       //reset to prepare for next hit
            for(count=0; count<STALL; count++);    //avoid piezo rebounding
            }
            max_update_en1 = 0; //don’t allow new hits to be recorded unless signal
                                //rises above HYST*THRESH (which will trigger else
                                // statement (see below)
      }
      else{
            max_update_en1 = 1; //track input from piezo
      }
      /*----Same as above, but now for ADC channel 2----*/
      if (ADCVal2 < (max_update_en2 ? THRESH : HYST*THRESH)){
            if(max_update_en2){
                  write_spi(2 | (ADCVal3>>4)<<2,max_val2 <<2);
                  write_spi(0,0); //clear spi buffer
                  max_val2 = 0;
                  for(count=0; count<STALL; count++);
            }
            max_update_en2 = 0;
```

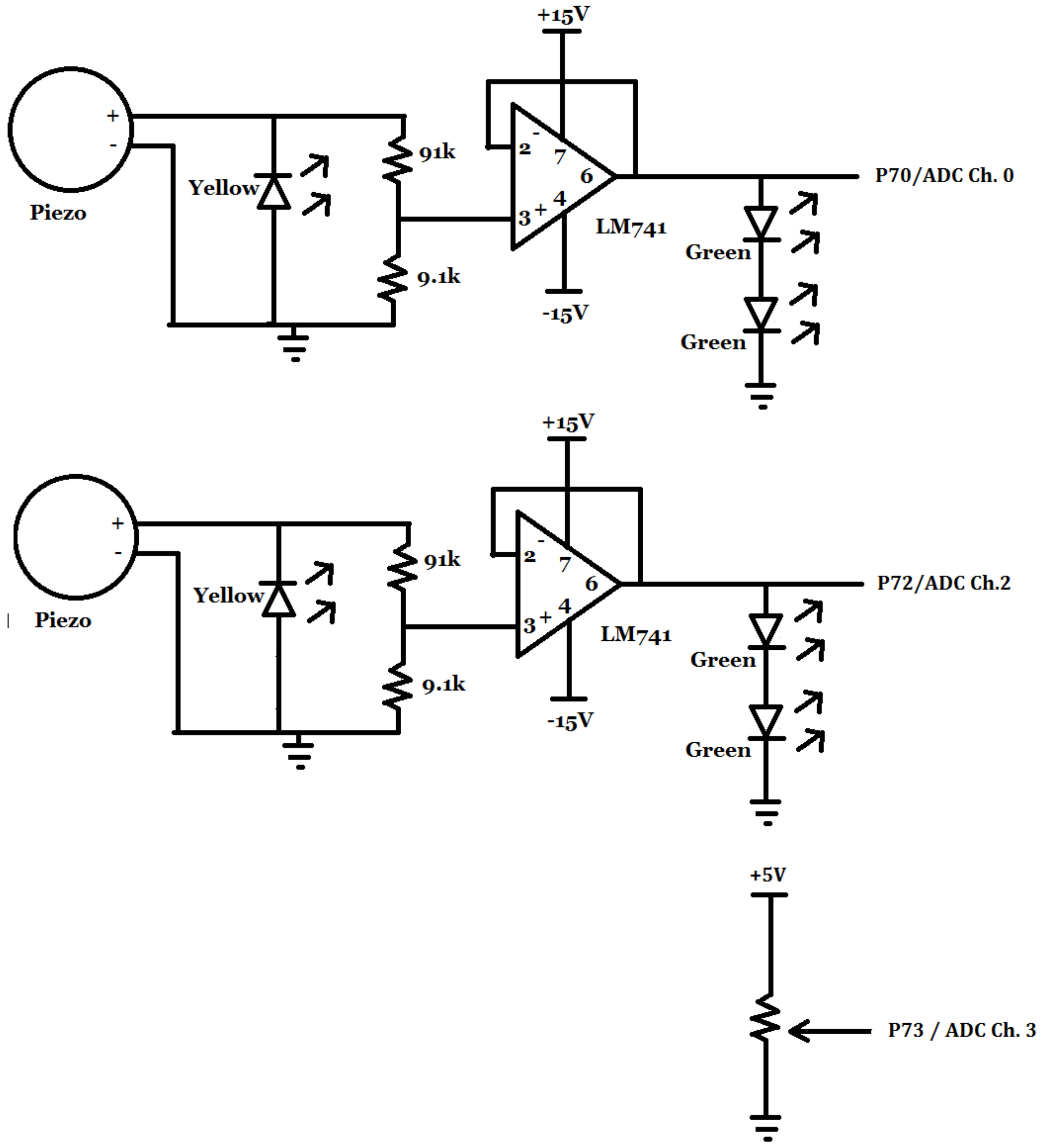
```
	}
	else{
		max_update_en2 = 1;
		//write_spi(0,0);
	}

	/*update max_vals if they're greater than previous max_vals*/
	if((ADCVal1 > max_val1) && max_update_en1){
		max_val1 = ADCVal1;
	}
	if((ADCVal2 > max_val2) && max_update_en2){
		max_val2 = ADCVal2;
	}

}                           //repeat

return 0;
}
```

B.2. SPI Configuration and Transfer Modules

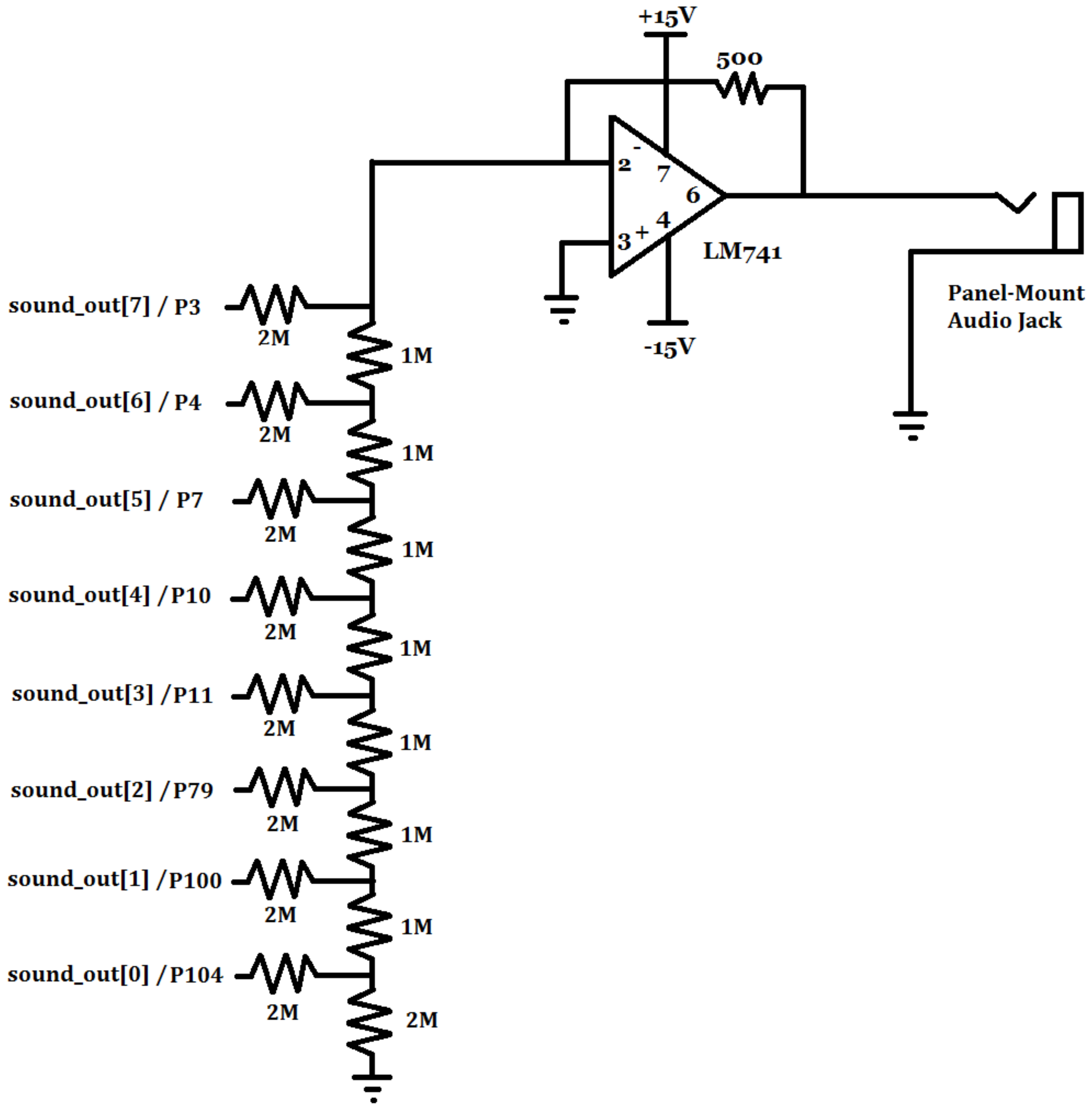
```
/*
	This code handles SPI operations from the PIC to the FPGA. The PIC is
configured          for 16-bit transfers.
*/

#include <p32xxxx.h>
#include <sys/attribs.h>

// set for Pb_clk = Sys_Clk/1

void spi_init(){
	// initialize SPI1
	IEC0CLR=0x03800000; // disable all interrupts
	IFS0CLR=0x03800000; // clear any existing event
	IPC5CLR=0x1f000000; // clear the priority
	IPC5SET=0x0d000000; // Set IPL=3, Subpriority 1
	IEC0SET=0x03800000; // Enable RX, TX and Error interrupts
	SPI1ASTATCLR=0x40;  // clear the Overflow
	SPI1ACON = 0;  // Stops and resets the SPI1
	SPI1ABRG = 15;        //
	SPI1ACON = 0x8720;  // 16-bit transfers, ON, Master EN
}

void write_spi(int input_1, int input_2){
	// write values to SPI1ABUF
	SPI1ABUF = (( (input_1) << 8)|(input_2) );
}
```

Appendix C. Schematics

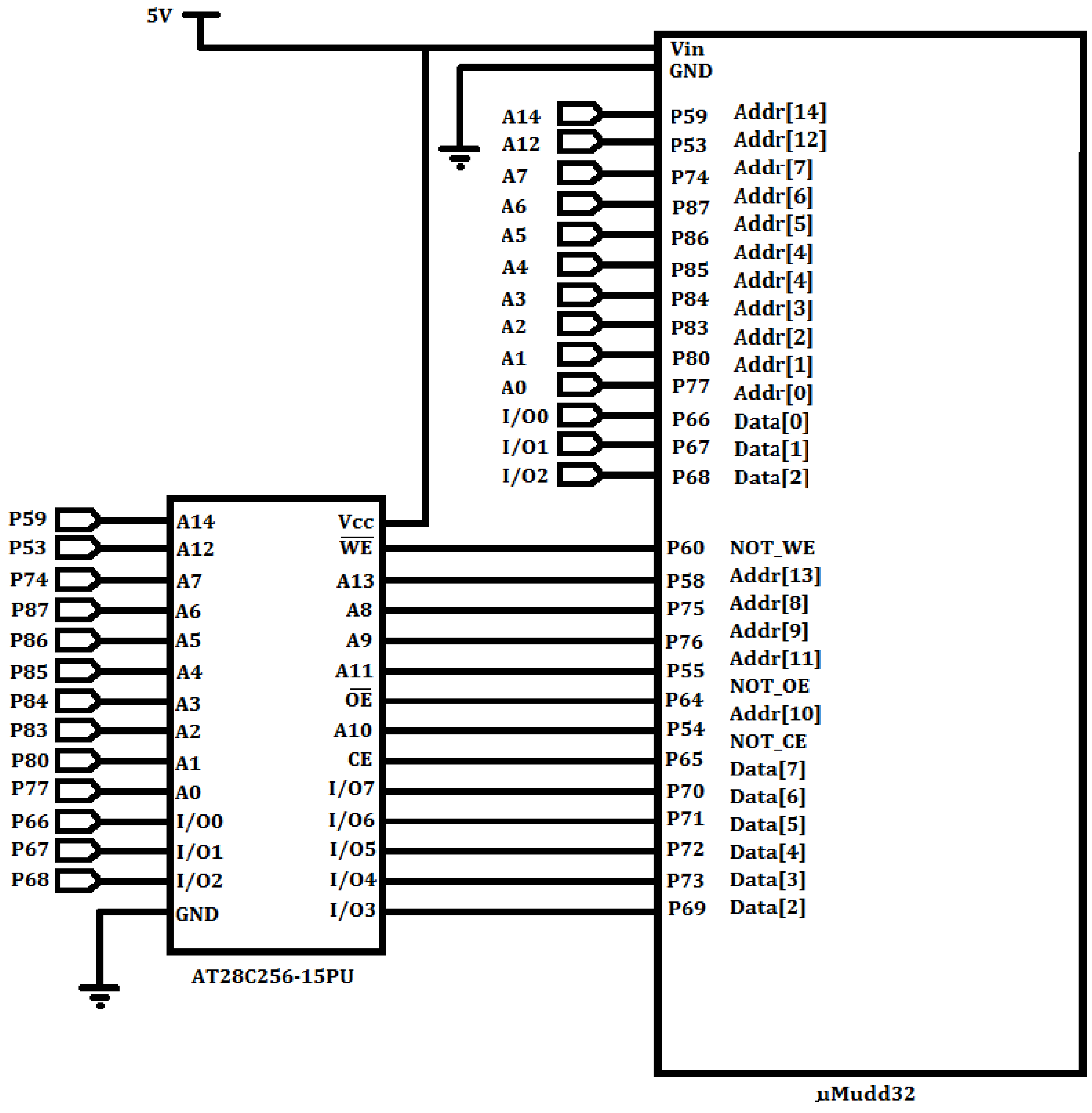
C.1. Piezoelectric Input Schematic



C.2. Output Schematic



C.3. EEPROM Board Schematic


5V
Vin
GND
A14
A12
A7
A6
A5
A4
A3
A2
A1
A0
I/O0
I/O1
I/O2
P59 Addr[14]
P53 Addr[12]
P74 Addr[7]
P87 Addr[6]
P86 Addr[5]
P85 Addr[4]
P84 Addr[4]
P83 Addr[3]
P80 Addr[2]
P77 Addr[1]
P66 Addr[0]
P67 Data[0]
P68 Data[1]
Data[2]
P59
P53
P74
P87
P86
P85
P84
P83
P80
P77
P66
P67
P68
A14
A12
A7
A6
A5
A4
A3
A2
A1
A0
I/O0
I/O1
I/O2
GND
Vcc
WE
A13
A8
A9
A11
OE
A10
CE
I/O7
I/O6
I/O5
I/O4
I/O3
P60 NOT_WE
P58 Addr[13]
P75 Addr[8]
P76 Addr[9]
P55 Addr[11]
P64 NOT_OE
P54 Addr[10]
P65 NOT_CE
P70 Data[7]
P71 Data[6]
P72 Data[5]
P73 Data[4]
P69 Data[3]
Data[2]
AT28C256-15PU
µMudd32