Electric Guitar Multi-FX and Recording Device

Final Project Report
December 14, 2018
E155

Matthew Calligaro and Giselle Serate

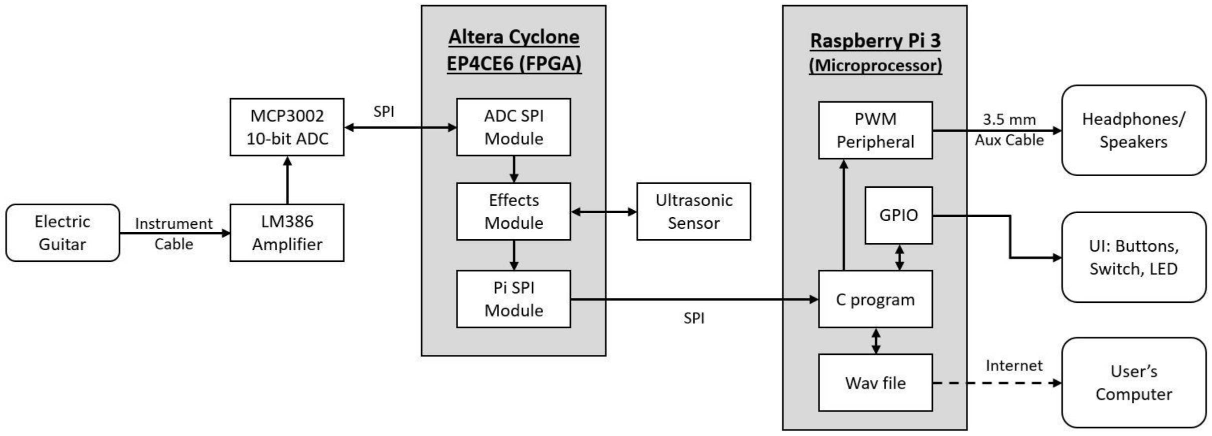
Abstract

An electric guitarist requires over a thousand dollars of equipment to apply and modulate effects while both recording and playing audio through speakers. Our product simplifies this process into a single affordable device. We convert an analog guitar signal to digital and allow the user to apply overdrive, delay, chorus, and distortion effects. These effects can be modulated by a distance sensor which attaches to their guitar. The processed audio is sent to a microcontroller which can record and play back with a pushbutton interface. Finally, the microcontroller can upload the audio to the internet as a WAV audio file, allowing the musician to easily download their recording to their computer.

Introduction

Electric guitarists traditionally need over a thousand dollars of electronic equipment to play and record. First, they need one or more effects devices to create the desired tonal quality. Second, they need an expression pedal or similar device to modulate these effects while playing. Finally, they need a mixer to convert this processed audio into a signal which can be sent to speakers or a computer for recording.

We simplified this process into a single affordable device. Our product samples an analog guitar input at 48 kHz (DVD quality) and applies up to four digital effects with an FPGA. It includes a distance sensor which can attach to the guitar's audio jack, allowing the user to modulate these effects while playing by changing the distance between the sensor and their hand or a wall. The processed audio signal is sent to a microcontroller, which uses PWM to produce a high-quality audio signal sent to a 3.5 mm audio jack. An interface of buttons and switches enables the user to record and play back their audio, allowing them to play along with their own tracks. Lastly, the user can export their recording as a 48 kHz WAV audio file which is accessible from the internet.



System-level block diagram

High quality audio requires consistent timing, so we use the FPGA to control the sampling and processing of the signal. The FPGA serves as the SPI master to both the ADC and the microcontroller and thus sets the clock speed for both devices, allowing us to sample and play at a consistent 48 kHz. We apply the audio effects with combinational logic and, in some cases, by adding in past signals stored in a ring buffer. Lastly, the FPGA interfaces with the ultrasonic sensor, which also requires precise timing.

The microcontroller is used to record and play back the audio because it has significantly more RAM. While the FPGA only has enough memory to store 0.34 seconds of audio, the Raspberry Pi can easily hold over 5 minutes of recording. Because the Pi runs a Linux operating system, it is ideal for saving this recording as a WAV file and hosting it on an Apache web server.

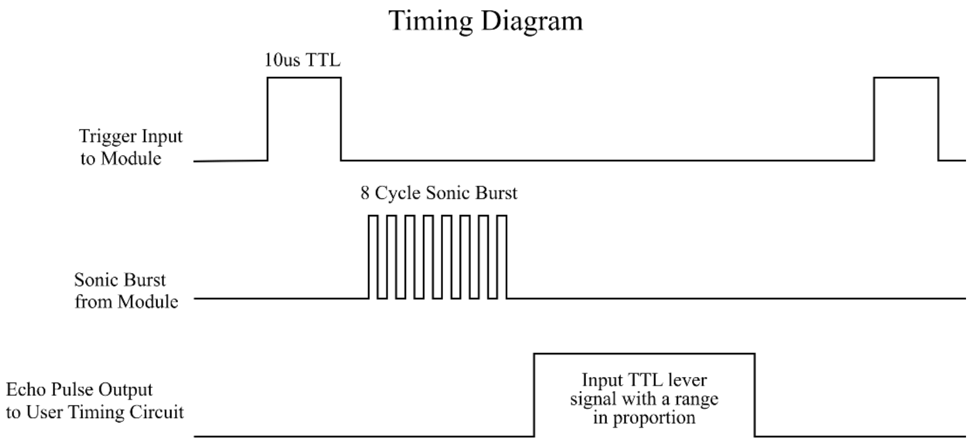
New Hardware

To modulate the effects applied to the guitar signal, we use an HC-SR04 ultrasonic rangefinder attached to the audio jack of the guitar. The FPGA interfaces with this sensor and uses the detected distance to modulate the intensity of its effects.

Inputs	**Outputs**
• Trigger	• Echo

(The device also has a 5-volt V_{DD} pin and a ground pin)

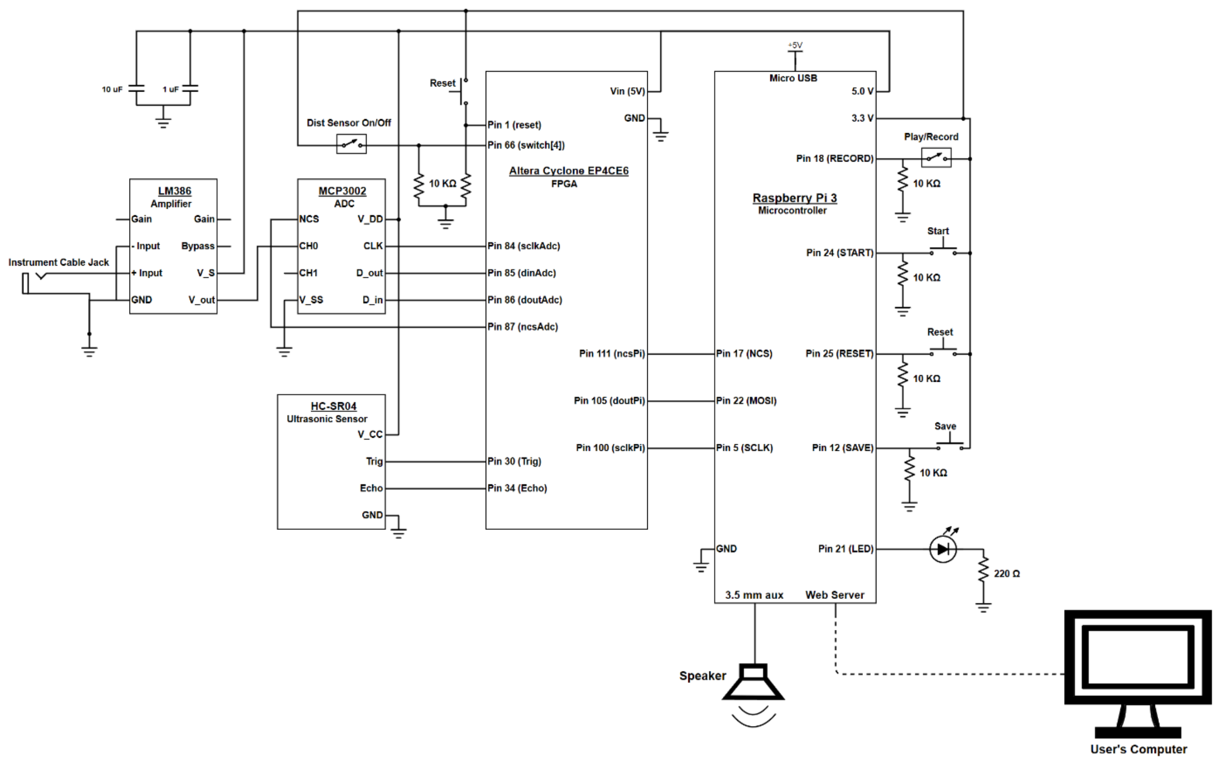
Following the datasheet at [5], to take a measurement, we begin by raising the trigger pin for 10 μs, which causes the detector to emit a short series of ultrasonic waves. It measures how long it takes these waves to return, which corresponds to the distance to the object in front of the device. It then raises the echo pin for a time proportional to this distance. The FPGA tracks how long the echo pin is pulled high to calculate an intensity value. The sensor must be given at least 60 ms between each trigger to complete the cycle.


Timing Diagram
10us TTL
Trigger Input to Module
8 Cycle Sonic Burst
Sonic Burst from Module
Echo Pulse Output to User Timing Circuit
Input TTL lever signal with a range in proportion

Timing diagram for the HC-SR04 ultrasonic sensor, reproduced from [5]

We found that the device is most reliable between the range of 1 inch to 1.5 feet and is most successful at detecting large, flat objects. To help account for inconsistent measurements, we took a moving average of past distances. To do so, we kept a running sum and stored past values in a ring buffer—each cycle, we added the most recent measurement and subtracted out the oldest. Averaging more points makes the control less responsive but smoother, and we found that eight points provided a good balance. We then converted this average to an intensity value from 0 to 8 so that it could be easily incorporated into the FPGA's effects module.

Schematics



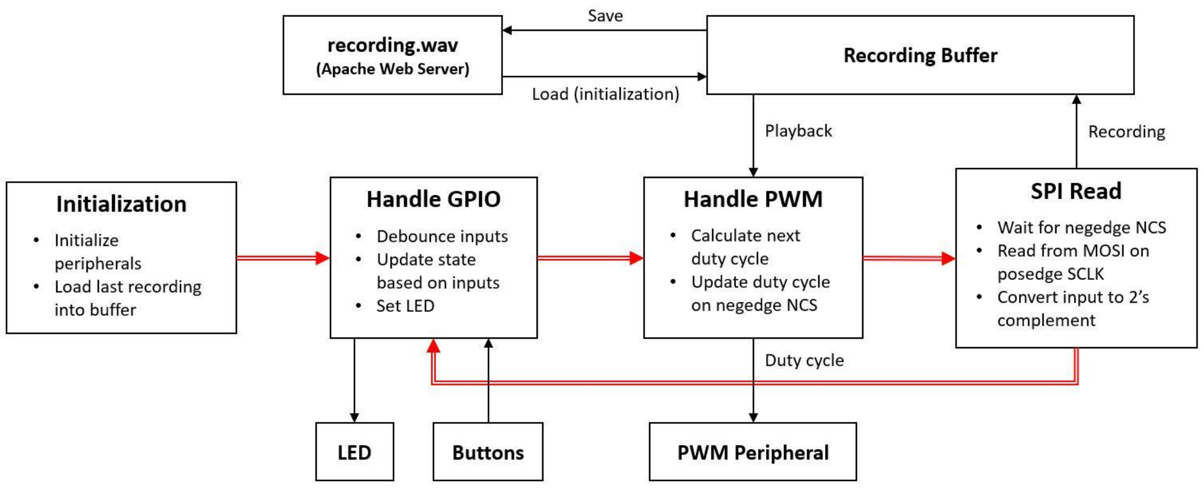
Complete circuit diagram

We can trace the signal path through the circuit as follows:

1. **Instrument Cable Jack**: The user connects their guitar here to provide an analog signal.
2. **LM386 Amplifier**: This amplifies the input and adds a voltage bias so the signal ranges from 0 to 5 volts. This allows the full signal to be captured by the ADC, which can only sample positive voltages.
3. **MCP3002 ADC**: This converts the analog signal to a digital signal and sends it to the FPGA over SPI.
4. **FPGA**: The FPGA removes the voltage bias and noise from the signal and optionally adds effects modulated by the **HC-SR04 Ultrasonic Sensor**. It sends the modified signal to the microcontroller over SPI with the FPGA as master.
5. **Microcontroller**: The microcontroller uses PWM to convert the digital signal to an analog signal on the 3.5 mm audio jack. It can also record and play back the waveform and save it to a WAV file available on a webpage.
6. **Powered Speaker or Headphones**: Any audio device which takes a 3.5 mm audio cable can play the output by connecting to the audio jack on the Raspberry Pi.

The entire system is powered by the micro USB power supply on the Raspberry Pi. To allow the user to play at a reasonable distance from the system, the wires connected to the HC-SR04 are 12 feet long. This wire can be attached to the guitar's audio cable and thus presents no additional inconvenience to the musician.

Microcontroller Design



Logic diagram of the microcontroller program

The microcontroller plays, records, and saves the processed audio signal received from the FPGA. This is done with a single C program. It also hosts an Apache web server to allow easy access to the recorded WAV file. See **Appendix B, Figure 24** for the source code.

GPIO Inputs	GPIO Outputs
• Record/Play mode (switch) • Start (pushbutton) • Reset (pushbutton) • Save (pushbutton) • NCS (SPI) • MOSI (SPI) • SCLK (SPI)	• LED (indicates recording/playing) **Other Outputs** • PWM to 3.5 mm audio jack

SPI Slave Module

The Raspberry Pi currently does not contain the necessary drivers to implement the SPI slave peripheral indicated by the Broadcom data sheet. Thus, we have implemented an SPI slave module over GPIO by designating NCS, MOSI, and SCLK input pins. MISO is not needed since the Pi does not send data back to the FPGA. At the start of a read, we wait for NCS to go low, and then at each rising edge of SCLK, we read MOSI and add this bit to our input. Once we have obtained 11 bits, we stop reading. Occasionally, the Pi will context switch away in the middle of an SPI transfer. We can detect this because NCS will appear to go high before we receive 11 bits. In this case, we use the previous successful transfer instead. We also run the program with the command **sudo nice -n -20 ./receiver** to give it maximum thread priority.

PWM Module

To convert the digital audio signal to analog, we use the PWM peripheral with output to the 3.5 mm audio jack. We initialize the PWM frequency to the maximum frequency of 100 MHz and use the peripheral PWM algorithm to produce the highest quality audio. To calculate the duty cycle, we bias and scale the SPI input so that it ranges from 0 to 1, as a negative duty cycle is not possible. We update the duty cycle immediately after NCS goes low since this is the most consistent event that occurs at 48 kHz.

User Interface

The user controls audio recording and playback with one DIP switch and three pushbuttons controlled through GPIO. Each input has a 10kΩ pulldown resistor to ensure a good zero. The DIP switch sets whether the device is in playback or record mode. The pushbuttons perform the following actions in these modes:

Button	Playback Mode	Record Mode
Start	Toggle between playing and paused	Start and stop recording
Reset	Return to the start of the recording	Clear the current recording
Save	Save recording as a WAV file and send it to the Apache web server	

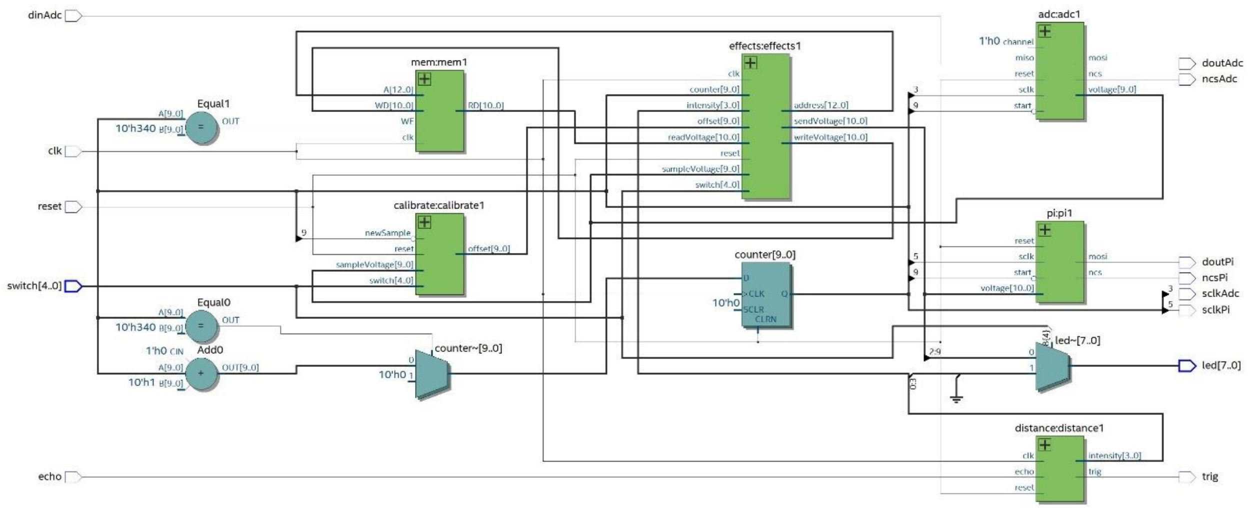
We allow these inputs 5 ms to debounce and only respond to the rising edge of a button press. This way, one button press always corresponds to exactly one input signal. Our GPIO module also controls an LED which indicates the following:

LED Status	Playback Mode	Record Mode
On	Playing audio	Recording audio
1 flash	Returned to the start of the recording	Cleared current recording
3 flashes	WAV file successfully saved to the Apache web server	

Recording and Playback

When the device is placed in record mode, each sample received from the Pi is added to a recording buffer. This recording buffer is encoded as a WAV file and saved to the Apache web server when the user presses the save button, allowing them to download the recording over the internet. On startup, the program detects the device's IP address and after saving, it uses this to display the location of the saved file. We use a recording buffer with 2^{24} entries, allowing for 5 minutes and 49 seconds of recording at 48 kHz. When the device is placed in play mode, the incoming audio signal is added to the samples stored in the recording buffer, allowing the user to listen to and play along with their recording.

FPGA Design



Top level block diagram of FPGA modules

The FPGA coordinates the timing of the many systems on our device and performs the digital audio processing. It uses the onboard 40 MHz clock to consistently sample and send at 48 kHz and interface with the ultrasonic sensor. It processes the incoming audio by removing the voltage bias and noise and applying up to four effects. Lastly, it displays the amplitude of the processed audio to an onboard LED array to help visualize the sound. See **Appendix A, Figures 1 through 22** for the corresponding System Verilog and block diagrams.

Inputs	Outputs
• Reset (pushbutton) • ADC MISO (SPI) • Echo (from distance sensor) • Switch[0]: overdrive • Switch[1]: delay • Switch[2]: chorus • Switch[3]: distortion • Switch[4]: use distance sensor	• ADC SCLK (SPI) • ADC MOSI (SPI) • ADC NCS (SPI) • Pi SCLK (SPI) • Pi MOSI (SPI) • Pi NCS (SPI) • Trig (to distance sensor) • LED Array (onboard)

SPI Modules (Appendix A, Figures 3 through 6)

The FPGA serves as the SPI master to both the ADC and the microcontroller. It sends the ADC the proper configuration data and reads the 10-bit sampled voltage into a shift register. We use a 1.25 MHz slave clock for the ADC. To communicate with the microcontroller, we send the 11-bit[1] processed audio signal with a shift register and receive no information in return. We use a 625 kHz slave clock because this is the slowest clock speed with a power of two clock divider that allows us to complete a transfer at 48 kHz.

[1] The processed audio signal is sent in sign-magnitude format so has 11 bits

Calibration Module **(Appendix A, Figures 7 and 8)**

Because the amplifier applies a voltage bias, we must remove this offset before applying effects. On reset, we average across 4096 samples to determine the average voltage bias. Our calibration also includes a configurable noise gate, which the user sets on reset with the five DIP switches. For each audio signal, we begin by removing the determined bias and eliminating noise at the provided gate level.

Effects Module **(Appendix A, Figures 9 and 10)**

With this preprocessed signal, the FPGA can apply any of the following four effects. We also save the preprocessed signal in a ring buffer, allowing the FPGA to access the past 0.341 seconds of audio which is used for the delay and chorus effects.

0. **Overdrive**: The signal is amplified with a left shift and saturated by maxing out voltages above a certain threshold. This intentionally "clips" the signal, producing a sound quality common in 70's era rock (**Appendix C, Figure 26**).
1. **Delay**: The oldest sample stored in RAM is added to the amplitude. This repeats the signal with a delay of 0.341 seconds, enabling auto-harmonizing melodies (**Appendix C, Figure 27 and Appendix D, Figure 32**).
2. **Chorus**: The samples from 21, 42, and 64 ms ago are added to the signal by reading from the addresses 512, 1024, and 1536 below the current write address in the ring buffer. This creates a "softer" and fuller sound as though there are multiple guitars playing the same note (**Appendix C, Figure 28**).
3. **Distortion**: First, all signals below a certain threshold are set to 0 to reduce noise. Then, the additive inverse of the signal is taken, which adds significant distortion and amplifies quiet sounds while dampening louder sounds. This produces a sound quality commonly used in guitar solos in 80's era metal and brings out quieter techniques such as hammer-ons and tapping (**Appendix C, Figure 29**).

Distance Module **(Appendix A, Figures 11 through 18)**

The distance module interfaces with the ultrasonic sensor to determine the detected distance and turn this into an intensity value (see New Hardware). When switch 4 is turned on, we use this intensity value to modify the magnitude of these effects as follows:

0. **Overdrive**: The intensity value determines the left shift amount and threshold to modulate both volume and clipping.
1. **Delay**: The intensity value determines the delay amount by changing the read address in ring buffer.
2. **Chorus**: The intensity value determines the read address in the ring buffer, with larger delays producing a more noticeable effect.
3. **Distortion**: When the distance sensor is on, we multiply the signal by a square wave to create a "repeater" effect, with frequency determined by the intensity value.

Appendix C, Figure 30 demonstrates the repeater effect modulated across varying distances.

Results

Our project satisfies all design specifications. We tested our system extensively across multiple days and found each element to be fully reliable. The system produces high quality audio, both when played through speakers and saved as a WAV file. While the effects are not as tonally developed as their professional equivalents, they emulate them reasonably well. In practice, we found it difficult to control the ultrasonic sensor with our hand while playing, and the sensor has difficulty reliably detecting a hand at a large range. However, if the user plays next to a wall, they can modulate the effects with high reliability by altering their distance from the wall.

This project presented several challenging aspects and debugging opportunities. Our initial project idea involved using the Adafruit Bluefruit SPI Friend to allow the musician to wirelessly connect their guitar to the microcontroller. In this design, the SPI Friend would interface with the FPGA and send data to the Pi's built in Bluetooth receiver. Unfortunately, there was not much documentation or working example code on the internet—we discovered that much of it was written for older versions of the Bluetooth stack, or for use with a phone instead of a Pi. In all, we succeeded in using a command line tool to pair the Friend with the Pi and receive acknowledgment that we had written some data to the Friend, but it was not clear how to execute an end to end data transfer. We were also only able to port some of this functionality to a C program, since modifying the code for our purposes required a fairly in-depth understanding of the gdbus library. Due to time constraints, we had to eliminate Bluetooth from the project altogether and substitute in SPI over wires to interface between the Pi and the FPGA.

Producing high quality audio also proved to be a difficult task, with challenges at nearly every step in the signal path. At first, the audio produced by the Pi was terribly distorted. To isolate the issue, we created a test program which played a clean audio file through PWM. This test program exhibited similar distortion, so we knew that our issue had to stem from the microcontroller. With more debugging, we identified some timing issues and determined that the PWM settings in EasyPIO.h are not ideal for producing audio. After carefully studying the PWM data sheet, we adjusted the values and were able to play our testing audio with great clarity.

We had originally intended for the Raspberry Pi to serve as the SPI master, but because the operating system can context switch away from our program, this resulted in inconsistent timing and undesired distortion in our audio. It also made the timing on the FPGA significantly more complex. Thus, we chose to make the microcontroller the SPI slave, but the Raspberry Pi does not contain any SPI slave drivers. We therefore had to implement this functionality over GPIO. Sometimes, the Pi would context switch out of our program during an SPI transfer and thus corrupt an audio sample. To accommodate for this, we restructured our FPGA code to allow for the slowest SPI clock possible and added filtering on the Pi's end to detect and throw out bad transfers.

After establishing successful end to end communication, we identified that we lost the negative halves of our waveforms because the ADC can only sample positive voltages. After experimenting with different solutions, we settled on using the LM386 audio amplifier to bias the signal in the 0 to 5 volt range. On the FPGA, we then added a calibration module to detect and remove this voltage bias to re-center the signal at 0 volts.

Unfortunately, the LM386 audio amplifier introduced a small amount of noise into our signal which became more noticeable after applying effects. We used both hardware and software approaches to minimize this noise. In hardware, we used a more reliable power source and added bypass capacitors. In software, we added a configurable noise gate which is set on restart. This allows the user to precisely tune the gate to filter out whatever noise the system is producing in the current environment.

We originally had issues with the ultrasonic sensor giving "jumpy" and inconsistent readings. To accommodate, we collected samples into a small ring buffer and took the moving average. This helped significantly and allowed for consistent readings, especially against flat, solid objects such as a binder or wall.

Overall, we are highly satisfied with this project. As engineers, we learned a lot from these several debugging opportunities and have gained a much deeper understanding of the elements in our system. As musicians, we find the device fun and easy to use, with high quality audio and valuable effects. In the future, we hope to take the project even further by adding more effects on the FPGA and more functionality on the Pi such as volume control and looping.

Our project is publicly available at the following GitHub repository:
https://github.com/MatthewCalligaro/E155Final

References

[1] Harris, D. M., & Harris, S. L., *Digital Design and Computer Architecture*, Amsterdam: Elsevier, 2013.
[2] Intel, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf
[3] Jameco, https://www.jameco.com/Jameco/Products/ProdDS/839826.pdf
[4] Microchip, http://ww1.microchip.com/downloads/en/devicedoc/21294e.pdf
[5] Mouser Electronics, https://www.mouser.com/ds/2/813/HCSR04-1022824.pdf
[6] Raspberry Pi, https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

Parts List

Part	Source	Vendor Part #	Price
Instrument Cable Jack	Pure Tone Technologies	N/A	Already owned
HC-SR04 Ultrasonic Sensor	SainSmart	101-60-142	$5.38

Appendix A: FPGA modules

```
// Name: Matthew Calligaro
// Email: mcalligaro@g.hmc.edu
// Date: 11/7/2018
// Summary: Top-level module for FPGA multi-effects

module FPGA(input logic clk,                            // 40 MHz clock
            input logic reset,                          // hardware reset
            input logic dinAdc,                         // MISO from ADC
            input logic echo,                           // echo pin of ultrasonic sensor
            input logic [4:0] switch,                   // hardware DIP switches
            output logic sclkAdc, doutAdc, ncsAdc,      // output for ADC SPI interface (FPGA is master)
            output logic sclkPi, doutPi, ncsPi,         // output for PI SPI interface (FPGA is master)
            output logic trig,                          // trigger pin of ultrasonic sensor
            output logic [7:0] led);                    // LED array on the MuddPi board

    // Registers
    logic [9:0] counter;        // counter to sample at 48 kHz and generate sclks

    // Wires between modules
    logic [12:0] address;       // address at which we read or write to RAM
    logic WE;                   // write enable for RAM
    logic [9:0] sampleVoltage;  // voltage read from ADC (unsigned)
    logic [10:0] readVoltage;   // voltage read from RAM (sign-magnitude)
    logic [10:0] writeVoltage;  // voltage to write into RAM (sign-magnitude)
    logic [10:0] sendVoltage;   // voltage to send to pi (sign-magnitude)
    logic [9:0] offset;         // voltage bias produced by amplifier (unsigned)
    logic [3:0] intensity;      // inverse of distance detected by ultrasonic sensor

    // Modules
    adc adc1(sclkAdc, reset, !counter[9], 1'b0, dinAdc, doutAdc, ncsAdc, sampleVoltage);
    pi pi1(sclkPi, reset, !counter[9], sendVoltage, doutPi, ncsPi);
    mem mem1(clk, WE, address, writeVoltage, readVoltage);
    calibrate calibrate1(reset, !counter[9], switch, sampleVoltage, offset);
    distance distance1(clk, reset, echo, trig, intensity);
    effects effects1(clk, reset, switch, counter, sampleVoltage, offset, readVoltage, intensity,
        sendVoltage, writeVoltage, address);

    // Increment counter
    always_ff @(posedge clk, posedge reset)
        if (reset)  counter <= 1'b0;
        else        counter <= (counter == 10'd832) ? 1'b0 : counter + 1'b1;

    // Assign wires
    assign WE = (counter == 10'd832);           // store writeVoltage on last clk of each cycle
    assign sclkAdc = counter[3];                // ADC's sclk = 2.5 MHz (16 clks)
    assign sclkPi = counter[5];                 // Pi's sclk  = 625 kHz (64 clks)
    assign led = switch[4] ?                    // LEDs display distance sensor if it is on
        {4'b0, intensity} : sendVoltage[9:2];   // otherwise, they display sendVoltage
endmodule
```

Figure 1: System Verilog for FPGA top level module

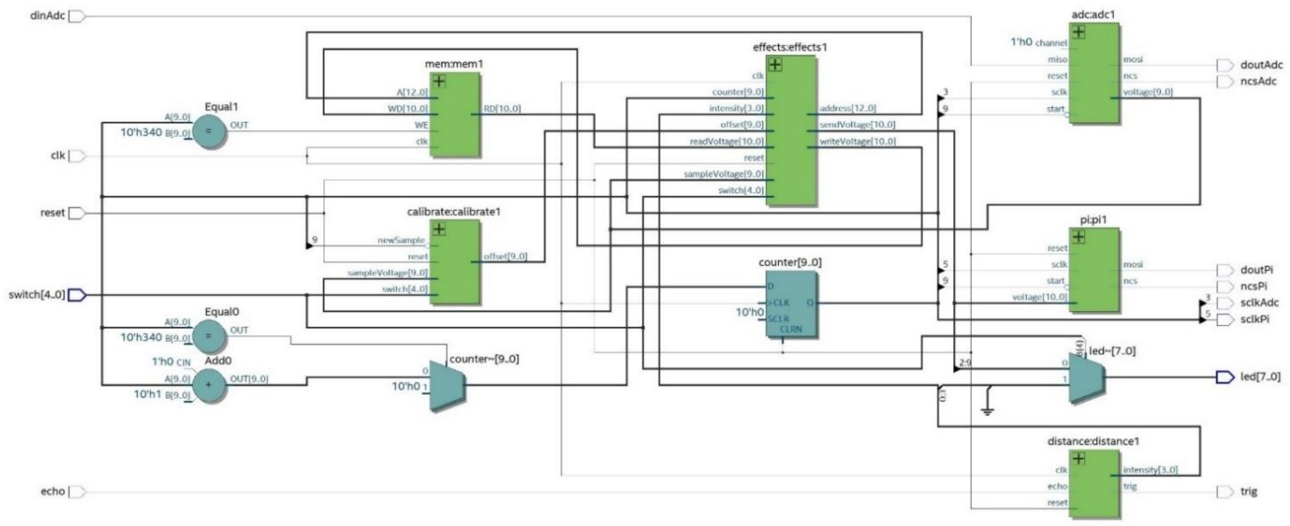


Figure 2: Block diagram for FPGA top level module

```
// Name: Matthew Calligaro
// Email: mcalligaro@g.hmc.edu
// Date: 11/7/2018
// Summary: SPI master to extract sampled voltages from ADC

module adc(input logic sclk,            // ADC slave clock (2.5 MHz)
           input logic reset,           // hardware reset
           input logic start,           // positive edge indicate the start of a new round
           input logic channel,         // channel to read from ADC
           input logic miso,            // MISO (data read from ADC)
           output logic mosi,           // MOSI (data sent to ADC)
           output logic ncs,            // chip select for ADC
           output logic [9:0] voltage); // shift register storing voltage read from ADC

    // Registers
    logic [3:0] counter;    // counter to keep track of bits in the SPI exchange
    logic lastStart;        // value of start last cycle

    // Send data on the negative edge of the clock
    always_ff @(negedge sclk, posedge reset)
        // Clear registers on reset
        if (reset)  counter <= 0;
        else begin
            // Reset counter on raising edge of start, otherwise count up to 15 and stop
            counter = (start && !lastStart) ? 4'b0 : (counter == 4'hF) ? 4'hF : counter + 4'b1;

            // Hold ncs low for first 15 cycles
            ncs = counter == 4'hF;

            // ODD/SIGN = channel, all calibration bits are 1
            mosi = channel ? 1'b1 : counter != 4'h2;
            lastStart = start;
        end

    // Read data on the positive edge of the clock
    always_ff @(posedge sclk)
        if (counter >= 4'h5 && counter < 4'hF) voltage <= {voltage[8:0], miso};
endmodule
```

Figure 3: System Verilog for ADC SPI module

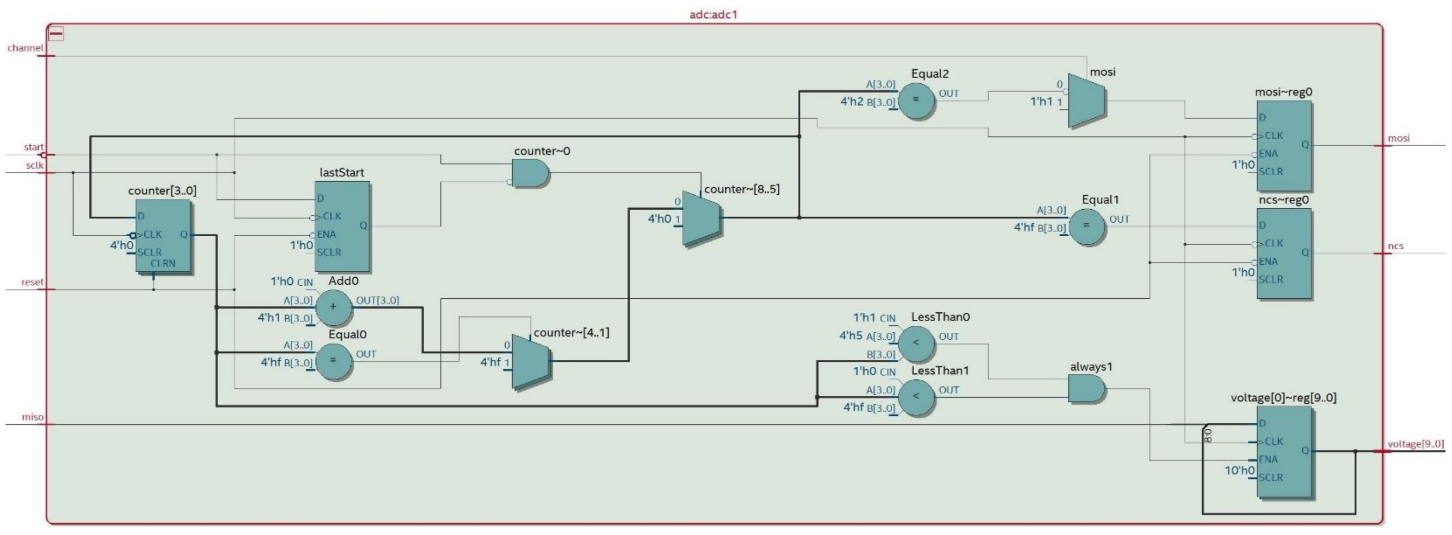


Figure 4: Block diagram for ADC SPI module

```
// Name: Matthew Calligaro
// Email: mcalligaro@g.hmc.edu
// Date: 11/13/2018
// Summary: SPI master to send voltages to the Raspberry Pi

module pi(input logic sclk,              // Pi slave clock (625 kHz)
          input logic reset,             // hardware reset
          input logic start,             // positive edge indicates the start of a new round
          input logic [10:0] voltage,    // voltage to send to the Pi (sign-magnitude)
          output logic mosi,             // MOSI (data sent to Pi)
          output logic ncs);             // chip select for Pi

    // Registers
    logic [3:0] counter;        // counter to keep track of bits in the SPI exchange
    logic [10:0] sendVoltage;   // shift register storing next bit to send
    logic lastStart;            // value of start last cycle

    // Send data on the negative edge of the clock
    always_ff @(negedge sclk, posedge reset) begin
        // Clear registers on reset
        if (reset)  counter <= 0;
        else begin
            // Reset counter on start, otherwise count up to 15 and stop
            counter = (start && !lastStart) ? 1'b0 : (counter == 4'hF) ? 4'hF : counter + 1'b1;

            // Load sendVoltage on new cycle and shift out one bit at a time
            if (counter == 0)   sendVoltage = voltage;
            else                sendVoltage = {sendVoltage[9:0], 1'b0};

            // Hold ncs low for first 11 clock cycles
            ncs = counter >= 11;
            lastStart = start;
        end
    end

    // Send top bit from shift register
    assign mosi = sendVoltage[10];
endmodule
```

Figure 5: System Verilog for Raspberry Pi SPI module

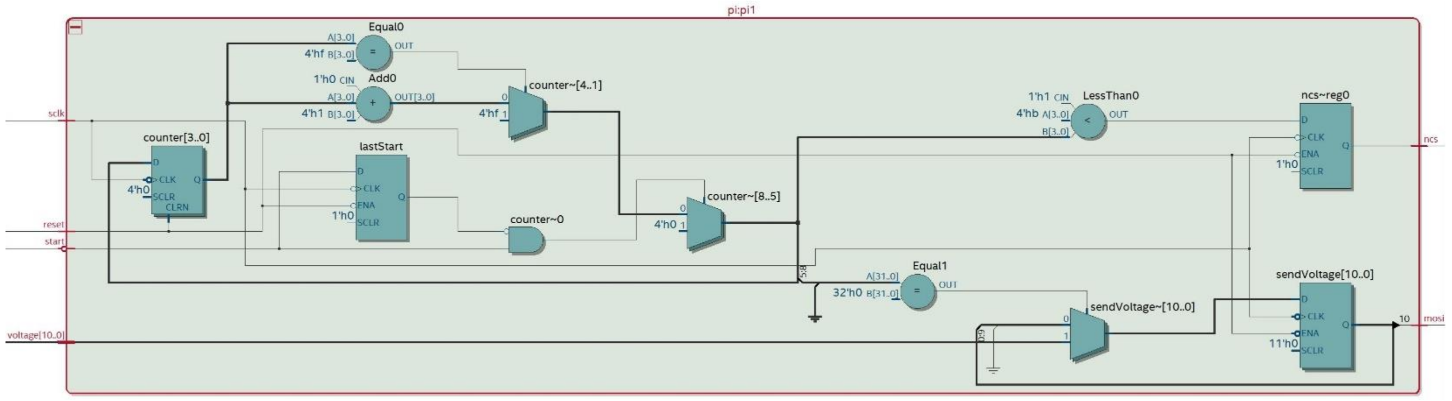


Figure 6: Block diagram for Raspberry Pi SPI module

```
// Name: Matthew Calligaro
// Email: mcalligaro@g.hmc.edu
// Date: 11/30/2018
// Summary: Calculates the voltage bias of the input signal

module calibrate(input logic reset,                 // hardware reset
                 input logic newSample,             // positive edge indicates new sample is ready
                 input logic [4:0] switch,          // hardware DIP switches
                 input logic [9:0] sampleVoltage,   // voltage sampled from the ADC
                 output logic [9:0] offset);        // voltage bias of the amplifier

    // Registers
    logic [31:0] offsetSum;             // sum of all samples collected during calibration period
    logic [12:0] calibrationSamples;    // the number of calibration samples taken so far
    logic [4:0] secondOffset;           // secondary offset configured with switches on reset

    always_ff @(posedge newSample, posedge reset) begin
        // Clear registers on reset (so we begin taking samples again) and load secondOffset
        if (reset) begin
            offsetSum <= 1'b0;
            calibrationSamples <= 1'b0;
            secondOffset <= switch;
        end

        // Collect 4096 calibration samples and sum in offsetSum
        else if (calibrationSamples < 13'h1000) begin
            offsetSum <= offsetSum + sampleVoltage;
            calibrationSamples <= calibrationSamples + 1'b1;
        end
    end

    // Offset is the average of the 4096 calibration samples minus secondOffset
    assign offset = offsetSum[21:12] - (secondOffset << 1'b0);
endmodule
```

Figure 7: System Verilog for calibration module

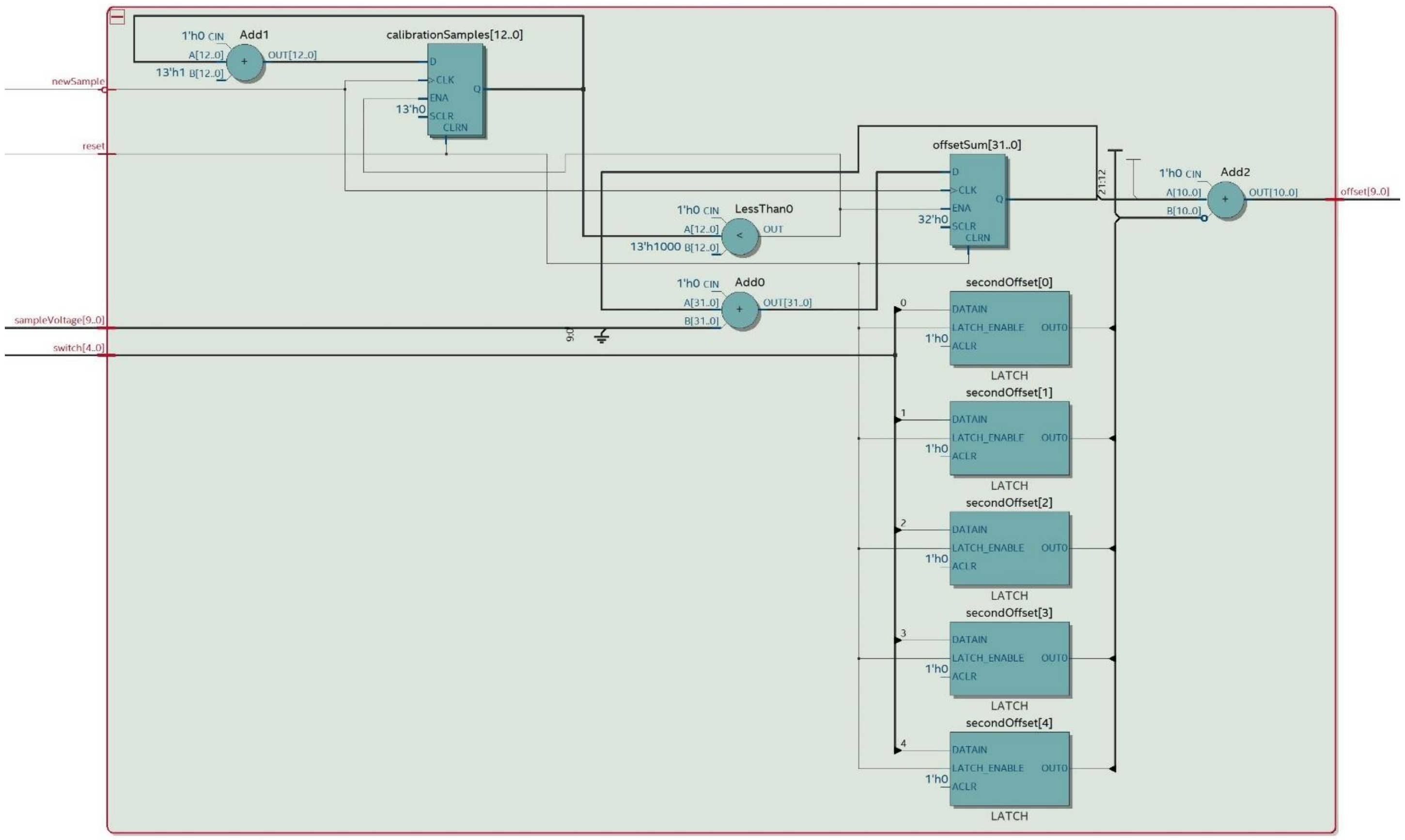


Figure 8: Block diagram for calibration module

```
// Name: Matthew Calligaro
// Email: mcalligaro@g.hmc.edu
// Date: 11/30/2018
// Summary: Applies effects to audio signal

module effects(input logic clk,                    // 40 MHz clock
               input logic reset,                  // hardware reset
               input logic [4:0] switch,           // hardware switches choosing effects
               input logic [9:0] counter,          // counter used to synchronize rounds
               input logic [9:0] sampleVoltage,    // voltage sampled from the ADC
               input logic [9:0] offset,           // voltage bias detected in calibration
               input logic [10:0] readVoltage,     // voltage read from ring buffer
               input logic [3:0] intensity,        // intensity detected by distance sensor
               output logic [10:0] sendVoltage,    // voltage to send to Pi
               output logic [10:0] writeVoltage,   // voltage to write to ring buffer
               output logic [12:0] address);       // register storing address of ring buffer

    // Registers
    logic [15:0] repCounter;    // counter used for repeater effect
    logic [12:0] writeAdr;      // current address we are writing to
    logic increaseAdr;          // indicates whether we should increase writeAdr this cycle
    logic [15:0] sumVoltage;    // voltage calculated as effects are applied (2's comp)

    // Wires
    logic [15:0] sampleExt;     // sampleVoltage extended to 15 bits
    logic [15:0] offsetExt;     // offset extended to 15 bits
    logic [15:0] offsetVoltage; // voltage after removing offset (2's comp)
    logic [15:0] sumVoltageAbs; // absolute value of sumVoltage (unsigned)
    logic [15:0] overdriven;    // signal with overdrive affect applied if on (unsigned)
    logic [15:0] threshold;     // value at which to saturate (unsigned)
    logic [9:0] saturated;      // saturated signal (unsigned)
    logic [9:0] sendVoltageMag; // magnitude of sendVoltage (unsigned)

    always_ff @(posedge clk, posedge reset) begin
        // Clear registers on reset
        if (reset) begin
            writeAdr <= 0;
            repCounter <= 0;
            increaseAdr <= 0;
        end else begin
            // Apply effects that require loading data from ring buffer and add into sumVoltage
            case (counter)
                10'h0: begin        // load in preprocVoltage
                    writeAdr <= writeAdr + increaseAdr;
                    increaseAdr <= !increaseAdr;
                    address <= writeAdr + 1'b1 + (switch[4] ? intensity << 9 : 1'b0);
                    repCounter <= repCounter + intensity;
                    sumVoltage <= offsetVoltage;
                end
                10'h1: begin        // add digital delay signal (if on)
                    if (switch[1])  sumVoltage <= (readVoltage[10] ?
                        (sumVoltage - readVoltage[9:0]) : (sumVoltage + readVoltage[9:0]));
                    address <= writeAdr - (switch[4] ? (intensity + 1'b1) << 8 : 13'h200);
                end
                10'h2: begin        // add chorus 1 signal (if on)
                    if (switch[2])  sumVoltage <= (readVoltage[10] ?
                        (sumVoltage - readVoltage[9:1]) : (sumVoltage + readVoltage[9:1]));
                    address <= writeAdr - (switch[4] ?
                        ((intensity + 1'b1) << 8) + ((intensity + 1'b1) << 7) : 13'h300);
                end
                10'h3: begin        // add chorus 2 signal (if on)
                    if (switch[2])  sumVoltage <= (readVoltage[10] ?
                        (sumVoltage - readVoltage[9:1]) : (sumVoltage + readVoltage[9:1]));
                    address <= writeAdr - (switch[4] ? (intensity + 1'b1) << 9 : 13'h400);
                end
                10'h4: begin        // add chorus 3 signal (if on)
                    if (switch[2])  sumVoltage <= (readVoltage[10] ?
                        (sumVoltage - readVoltage[9:1]) : (sumVoltage + readVoltage[9:1]));
                    address <= writeAdr;
                end
            endcase
        end
    end
```

```
always_comb begin
        // Extend input voltages to 15 bits
        sampleExt = sampleVoltage;
        offsetExt = offset;

        // Remove offset from sample voltage
        offsetVoltage = sampleExt - offsetExt;

        // Absolute value of sumVoltage
        if (sumVoltage[15])     sumVoltageAbs = -sumVoltage;
        else                    sumVoltageAbs = sumVoltage;

        // Apply overdrive (if on) by amplifying voltage above a certain threshold
        if (switch[0] && sumVoltageAbs > 8'h1F)
            if (switch[4])  overdriven = sumVoltageAbs << intensity;
            else            overdriven = sumVoltageAbs << 2;
        else                overdriven = sumVoltageAbs;

        // Calculate threshold of saturation (lower if overdriven)
        if (switch[0])
            if (switch[4])  threshold = (intensity << 6) - 1'b1 + 16'h7F;
            else            threshold = 16'hFF;
        else                threshold = 16'h3FF;

        // Saturate signal at calculated threshold
        if (overdriven > threshold)     saturated = threshold[9:0];
        else                            saturated = overdriven[9:0];

        // Apply solo effect (if on) by filtering voltages below a threshold and inverting
        // Also apply repeater effect (if on)
        if (switch[3])
            if (saturated < 4'hF || (switch[4] && repCounter[15] && intensity > 4'h1))
                    sendVoltageMag = 1'b0;
            else    sendVoltageMag = (-saturated) >> 1;
        else        sendVoltageMag = saturated;

        // Construct sendVoltage as sign-magnitude
        sendVoltage = {sumVoltage[15], sendVoltageMag};

        // Convert offsetVoltage to sign-magnitude to store in ring buffer
        if(offsetVoltage[15])   writeVoltage = {1'b1, -offsetVoltage[9:0]};
        else                    writeVoltage = {1'b0, offsetVoltage[9:0]};
    end
endmodule
```

Figure 9: System Verilog for effects module

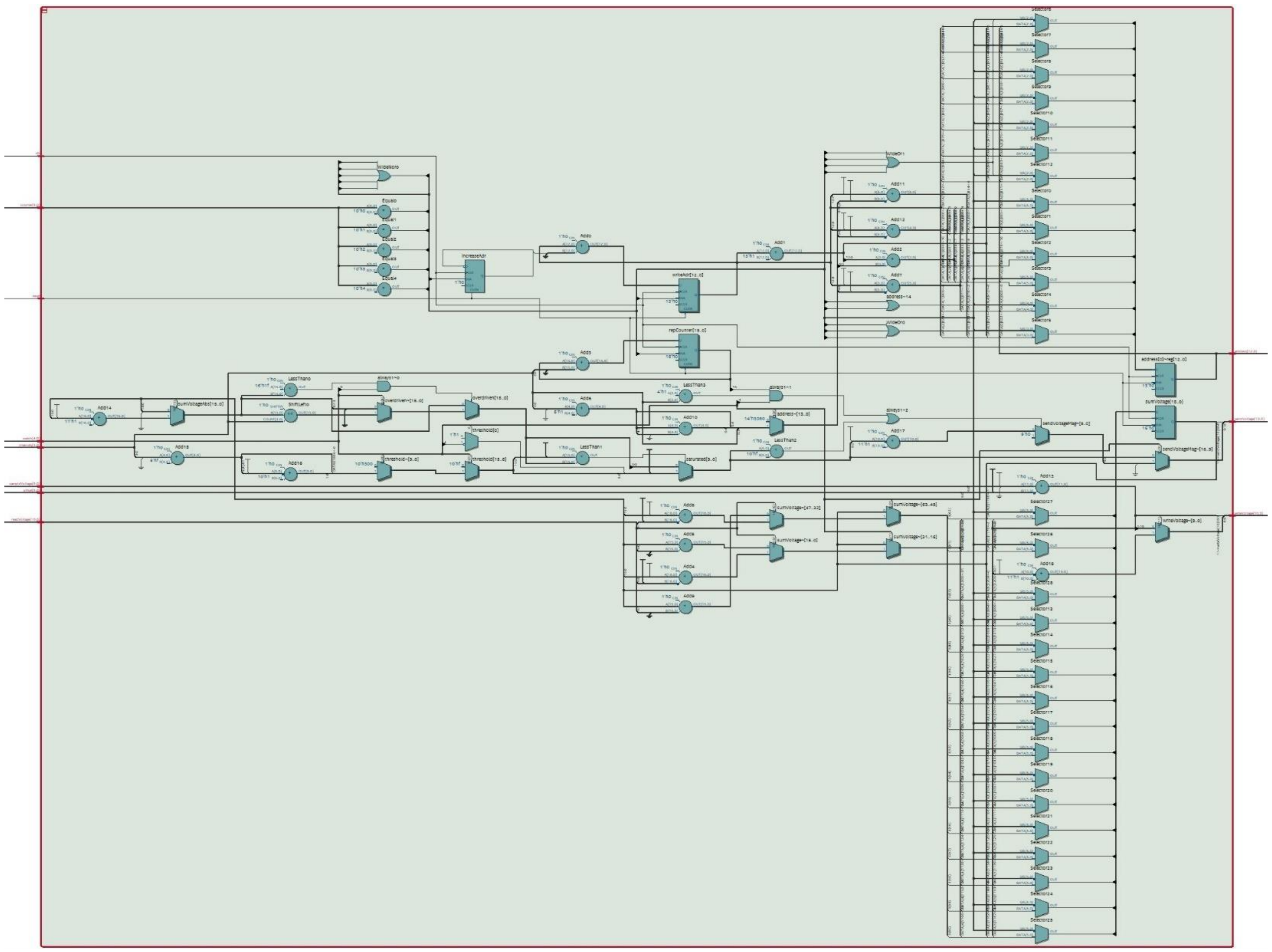


Figure 10: Block diagram for effects module

```
// Name: Giselle Serate
// Email: gserate@g.hmc.edu
// Date: 11/30/2018
// Summary: Top-level module for interfacing with distance sensor

module distance(input logic clk,                // 40 MHz clock
                input logic reset,              // hardware reset
                input logic echo,               // echo pin of distance sensor
                output logic trig,              // trigger pin of distance sensor
                output logic [3:0] intensity);  // intensity level with higher meaning closer

    // Wires between modules
    logic [11:0] newest, oldest, average;
    logic [2:0] address;
    logic WE;

    // Modules
    averager averager1(clk, reset, trig, newest, oldest, average);
    memSmall memSmall2(clk, !trig, address, newest, oldest);
    distsensor distsensor1(clk, reset, echo, trig, newest, address, WE);
    intensity getintensity(average, intensity);
endmodule
```

Figure 11: System Verilog for top level module for detecting distance

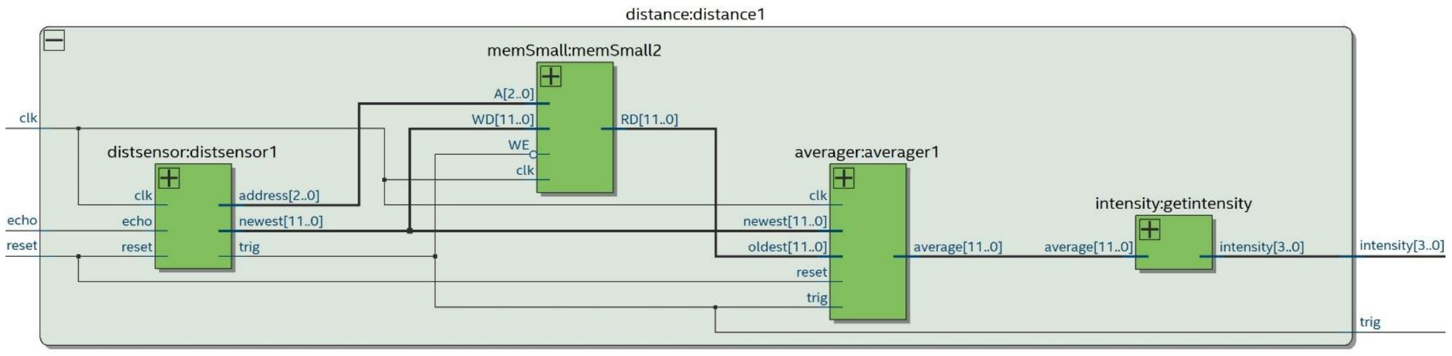


Figure 12: Block diagram for top level module for detecting distance

```
// Name: Matthew Calligaro, Giselle Serate
// Email: mcalligaro@g.hmc.edu, gserate@g.hmc.edu
// Date: 12/9/2018
// Summary: Average the last 8 readings from the distance sensor

module averager(input logic clk,                // 40 MHz clock
                input logic reset,              // hardware reset
                input logic trig,               // trigger pin of distance sensor
                input logic [11:0] newest,      // most recent distance detected
                input logic [11:0] oldest,      // oldest distance stored in ring buffer
                output logic [11:0] average);   // average of past 8 distances

    // Registers
    logic [15:0] sum;       // sum of previous 8 distances
    logic [31:0] counter;   // counter to identify first 8 cycles
    logic lastTrig;         // value of trig last cycle

    always_ff@(posedge clk, posedge reset)
        // Clear registers on reset
        if(reset) begin
            sum <= 1'b0;
            counter <= 1'b0;
            lastTrig <= 1'b0;
        end else begin
            lastTrig <= trig;

            // Update sum and counter on falling edge of trigger
            if (!trig & lastTrig) begin
                // Do not subtract oldest on first 8 cycles
                sum <= sum + newest - (counter > 32'h8 ? oldest : 1'b0);
                counter <= counter + 1;
            end
        end

    // Average is sum divided by 8
    assign average = sum[14:3];
endmodule
```

Figure 13: System Verilog for calculating moving average of past distance measurements

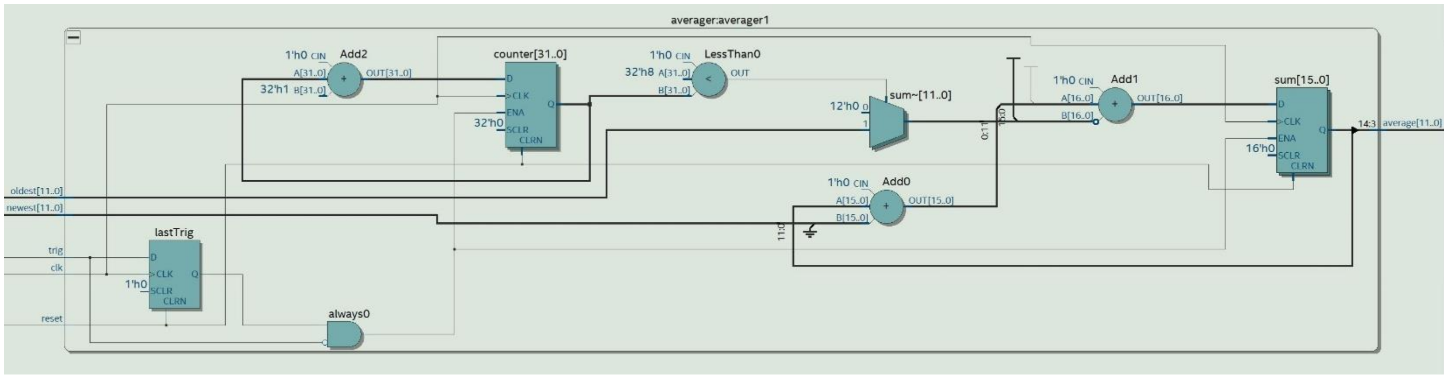


Figure 14: Block diagram for calculating moving average of past distance measurements

```
// Name: Matthew Calligaro, Giselle Serate
// Email: mcalligaro@g.hmc.edu, gserate@g.hmc.edu
// Date: 12/9/2018
// Summary: Reads distance from the distance sensor

module distsensor(input logic clk,              // 40 MHz clock
                  input logic reset,            // hardware reset
                  input logic echo,             // echo pin of distance sensor
                  output logic trig,            // trigger pin of distance sensor
                  output logic [11:0] newest,   // register storing most recent distance detected
                  output logic [2:0] address,   // register storing address for ring buffer
                  output logic WE);             // write enable for ring buffer

    // Registers
    logic [23:0] counter;       // counter to control timing
    logic [16:0] accumulator;   // how long echo has been raised

    always_ff@(posedge clk, posedge reset)
        // Clear registers on reset
        if(reset) begin
            counter <= 1'b0;
            address <= 1'b0;
        end else begin
            // Count to 0x249F00 so that each round lasts 60 ms
            counter <= (counter == 24'h249F00) ? 1'b0 : counter + 1'b1;

            // Raise trigger high and save past reading at start of each round
            if (counter == 24'h0) begin
                trig <= 1'b1;
                accumulator <= 1'b0;
                newest <= accumulator[16:5];
            end

            // Increase address after writing
            else if (counter == 24'h1)      address <= address + 1'b1;

            // Lower trigger after 20 us
            else if (counter == 24'h320)    trig <= 1'b0;

            // Count how long echo is held high, which corresponds to distance
            else if (counter > 24'h320)     accumulator <= accumulator + echo;
        end

    // Write on the 1-index clk of each round
    assign WE = (counter == 1'b1);
endmodule
```

Figure 15: System Verilog for distance sensor module

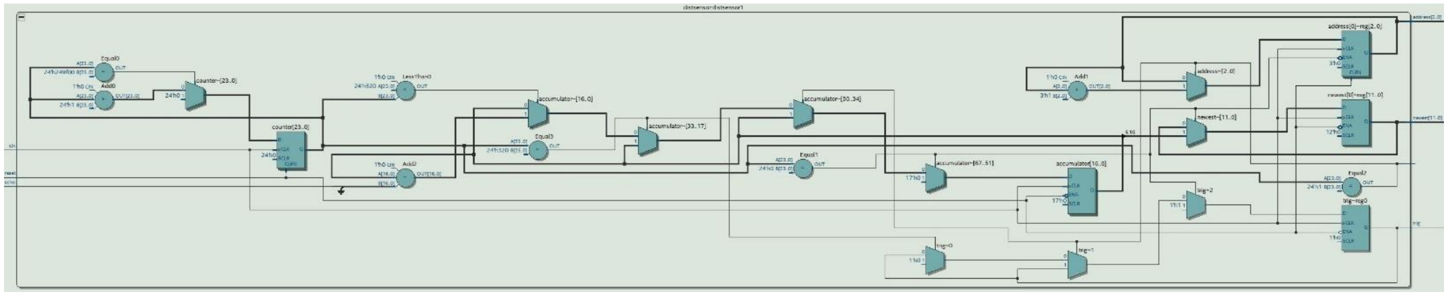


Figure 16: Block diagram for distance sensor module

```
// Name: Giselle Serate
// Email: gserate@g.hmc.edu
// Date: 11/30/2018
// Summary: Converts a measured distance into an intensity

module intensity(input logic [11:0] average,    // average distance detected by sensor
                 output logic [3:0] intensity); // 9-level intensity corresponding to distance

    always_comb begin
        if      (average > 12'h900)     intensity = 4'h0;   // farthest = least intense.
        else if (average > 12'h800)     intensity = 4'h1;
        else if (average > 12'h700)     intensity = 4'h2;
        else if (average > 12'h600)     intensity = 4'h3;
        else if (average > 12'h500)     intensity = 4'h4;
        else if (average > 12'h400)     intensity = 4'h5;
        else if (average > 12'h300)     intensity = 4'h6;
        else if (average > 12'h200)     intensity = 4'h7;
        else                            intensity = 4'h8;   // closest = most intense.
    end
endmodule
```

Figure 17: System Verilog for module converting distance into an intensity value

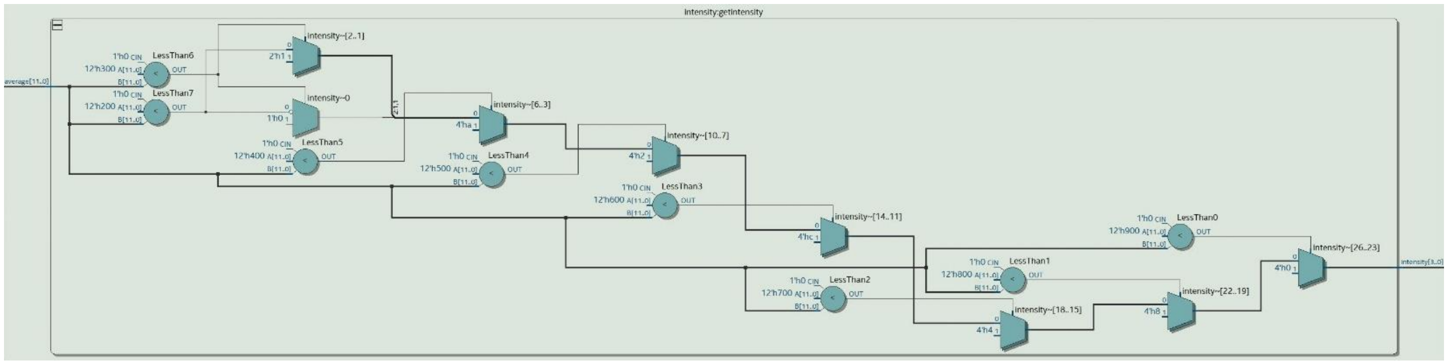


Figure 18: Block diagram for module converting distance into an intensity value

```
// Name: Matthew Calligaro
// Email: mcalligaro@g.hmc.edu
// Date: 11/7/2018
// Summary: RAM module for effects ring buffer
// Code adapted from Digital Design and Computer Architecture, 455

module mem(input logic clk,              // 40 MHz clock
           input logic WE,               // write enable
           input logic [12:0] A,         // address
           input logic [10:0] WD,        // data to write to A when WE is asserted
           output logic [10:0] RD);      // data read from A

    logic [10:0] RAM[8191:0];
    assign RD = RAM[A];

    // Write data if WE is asserted
    always_ff@(posedge clk)
        if (WE) RAM[A] <= WD;
endmodule
```

Figure 19: System Verilog for ring buffer storing past signal voltages

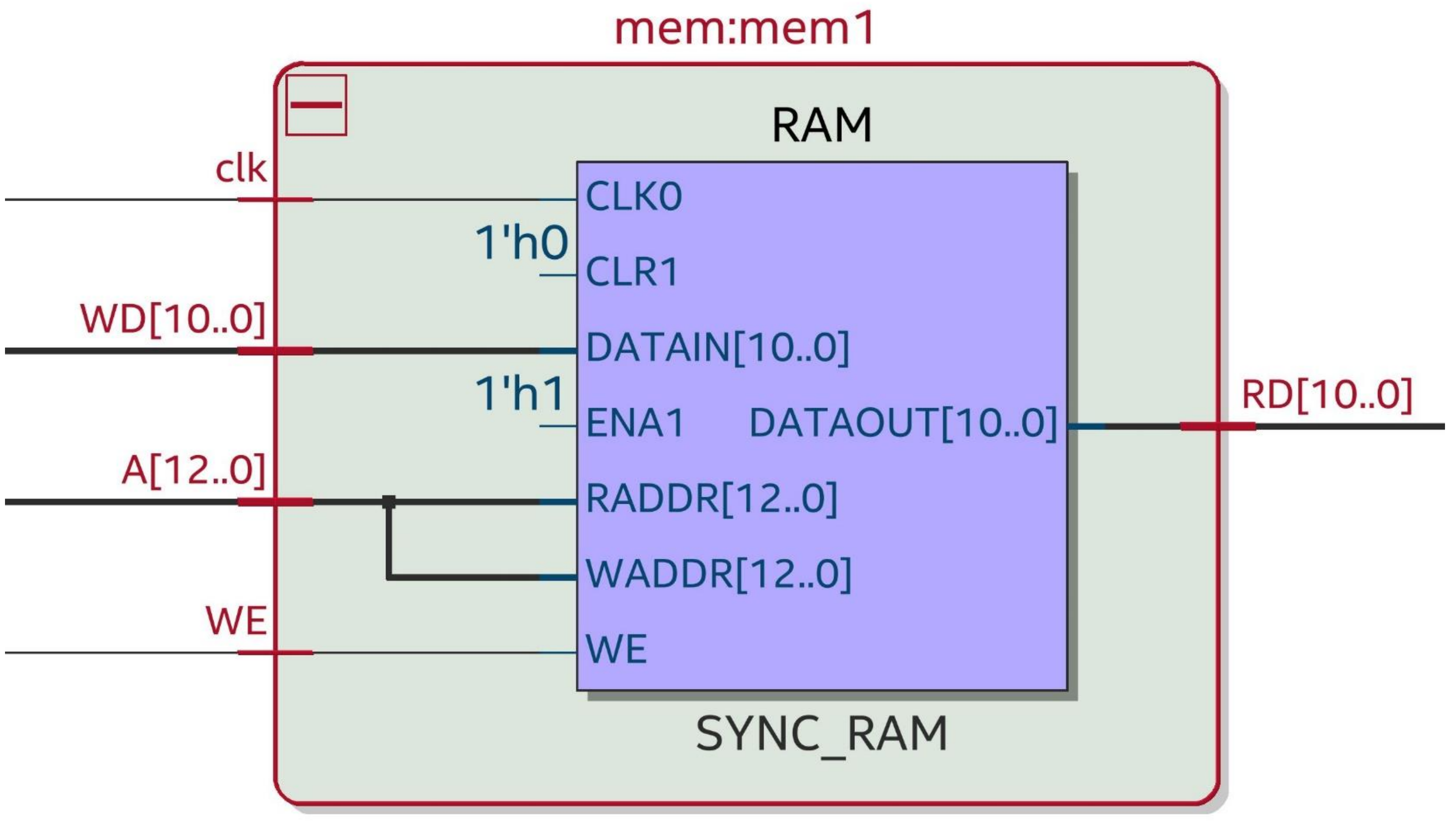


Figure 20: Block diagram for ring buffer storing past signal voltages

```
// Name: Matthew Calligaro
// Email: mcalligaro@g.hmc.edu
// Date: 11/7/2018
// Summary: RAM module for distance ring buffer
// Code adapted from Digital Design and Computer Architecture, 455

module memSmall(input logic clk,            // 40 MHz clock
                input logic WE,             // write enable
                input logic [2:0] A,        // address
                input logic [11:0] WD,      // data to write to A when WE is asserted
                output logic [11:0] RD);    // data read from A

    logic [11:0] RAM[7:0];
    assign RD = RAM[A];

    // Write data if WE is asserted
    always_ff@(posedge clk)
        if (WE) RAM[A] <= WD;
endmodule
```

Figure 21: System Verilog for ring buffer storing past distance measurements

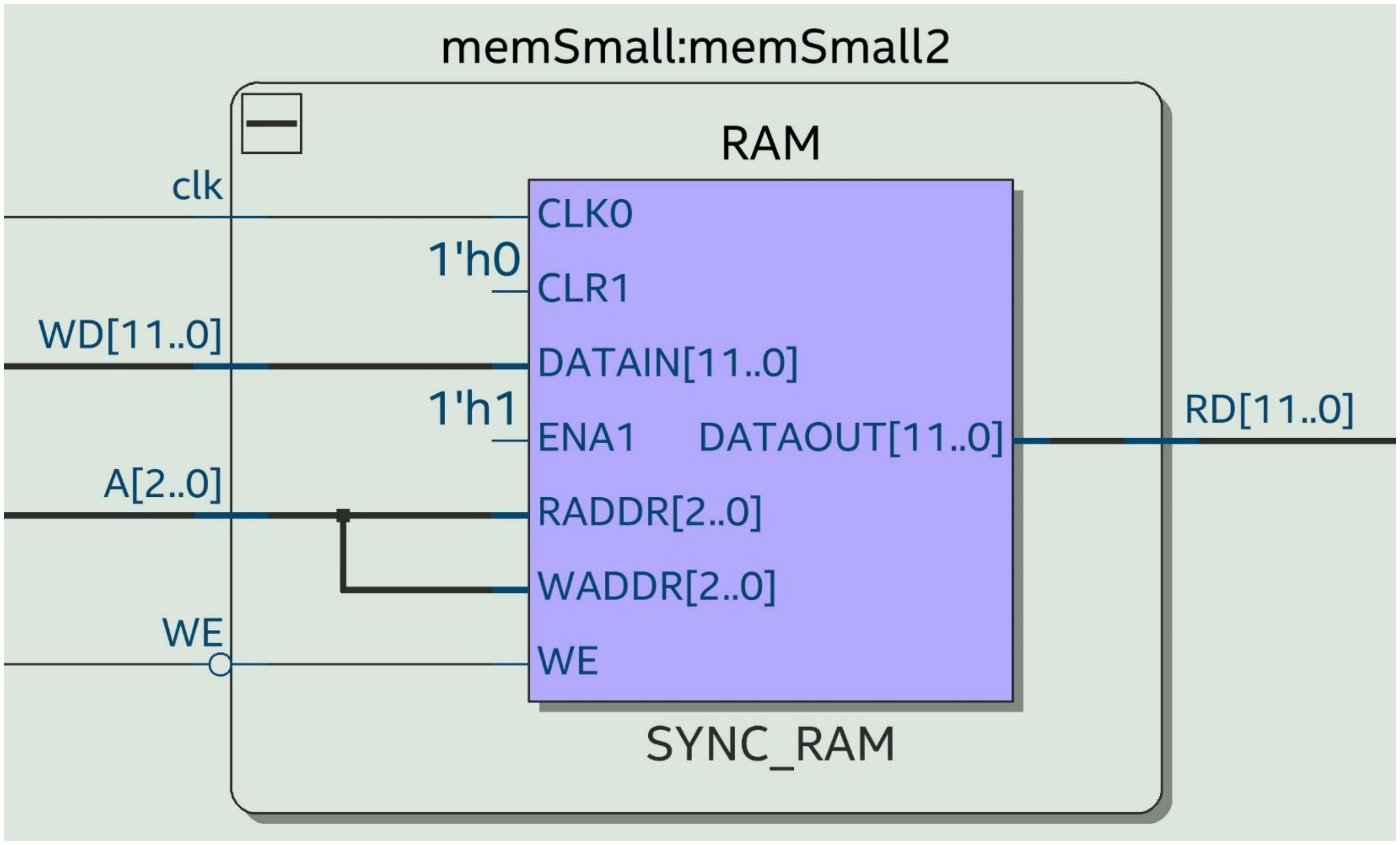
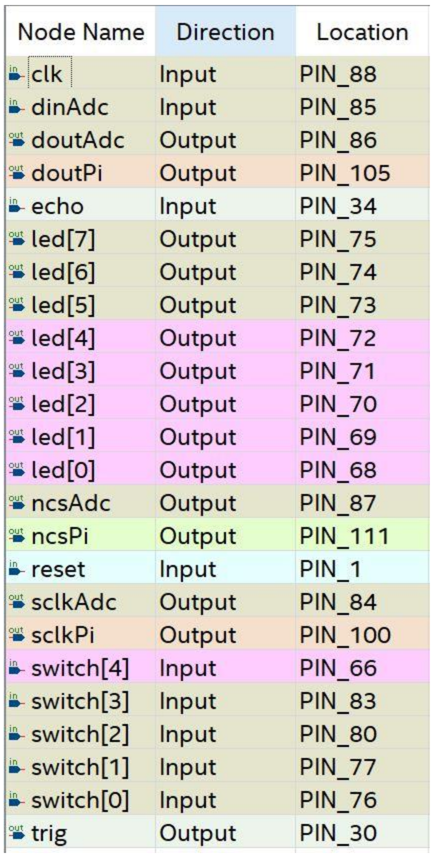


Figure 22: Block diagram for ring buffer storing past distance measurements



Node Name	Direction	Location
clk	Input	PIN_88
dinAdc	Input	PIN_85
doutAdc	Output	PIN_86
doutPi	Output	PIN_105
echo	Input	PIN_34
led[7]	Output	PIN_75
led[6]	Output	PIN_74
led[5]	Output	PIN_73
led[4]	Output	PIN_72
led[3]	Output	PIN_71
led[2]	Output	PIN_70
led[1]	Output	PIN_69
led[0]	Output	PIN_68
ncsAdc	Output	PIN_87
ncsPi	Output	PIN_111
reset	Input	PIN_1
sclkAdc	Output	PIN_84
sclkPi	Output	PIN_100
switch[4]	Input	PIN_66
switch[3]	Input	PIN_83
switch[2]	Input	PIN_80
switch[1]	Input	PIN_77
switch[0]	Input	PIN_76
trig	Output	PIN_30

Figure 23: FPGA pin assignments

Appendix B: Microcontroller program

```c
// Name: Matthew Calligaro
// Email: mcalligaro@g.hmc.edu
// Date: 11/10/2018
// Summary: Plays and records the digital signal sent by the FPGA

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <ifaddrs.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "EasyPIO.h"

////////////////////////////////
//  Constants and Globals
////////////////////////////////

// Program constants
#define VOLUME 16           // volume multiplier
#define BUF_SIZE (1 << 24)  // size of recording buffer (allows for 5 mins 49 secs of recording)
#define INPUT_BITS 11       // bit depth of FPGA signal
#define FLASH_TIME 200      // LED flash time in miliseconds
#define DEBOUNCE_TIME 5     // time in miliseconds to wait for inputs to debounce

// Pins
#define PIN_RECORD 18       // switch to determine play or record mode
#define PIN_START 24        // pushbutton to start/stop play or record
#define PIN_RESET 25        // pushbutton to reset play or record
#define PIN_SAVE 12         // pushbutton to save recording
#define PIN_LED 21          // LED to indicate when playing or recording
#define NCS 17              // SPI chip select
#define MOSI 22             // SPI master out slave in
#define SCLK 5              // SPI clock

// WAV constants
#define CHANNELS 1          // number of channels (mono or stereo)
#define SAMPLE_RATE 48000   // samples per second
#define BIT_DEPTH 16        // bits per sample
#define BIT_RATE (SAMPLE_RATE * BIT_DEPTH * CHANNELS / 8)   // bits per second
#define BYTES_PER_SAMPLE (BIT_DEPTH * CHANNELS / 8)         // bytes per sample

// Global Variables
short buffer[BUF_SIZE];     // stores samples of the recording
                            // (must be a global variable to prevent segfault due to size)

////////////////////////////////
//  Structs
////////////////////////////////

/**
 * \brief Struct storing the 44-bit file header of a .wav file
 */
typedef struct
{
    char fileFormat[4];
    int fileLength;
    char fileType[4];
    char formatHeader[4];
    int formatLength;
    short formatType;
    short channels;
    int sampleRate;
    int bitRate;
    short bytesPerSample;
    short bitDepth;
    char dataHeader[4];
    int dataLength;
} WavHeader;
```

```c
////////////////////////////////
//  Functions
////////////////////////////////

/**
 * \brief Initialize peripherals
 */
void init()
{
    // Initialize peripherals
    pioInit();
    pwmInit();

    // Initialize user interface pins
    pinMode(PIN_RECORD, INPUT);
    pinMode(PIN_START, INPUT);
    pinMode(PIN_RESET, INPUT);
    pinMode(PIN_SAVE, INPUT);
    pinMode(PIN_LED, OUTPUT);

    // Initialize SPI pins
    pinMode(NCS, INPUT);
    pinMode(MOSI, INPUT);
    pinMode(SCLK, INPUT);
}

/**
 * \brief Save recorded audio to a .wav file on the website
 *
 * \param buffer        recorded audio samples
 * \param bufferSize    number of audio samples in buffer
 */
void saveRecording(short* buffer, size_t bufferSize)
{
    // Fill header with correct data
    WavHeader header;
    header.fileFormat[0] = 'R';
    header.fileFormat[1] = 'I';
    header.fileFormat[2] = 'F';
    header.fileFormat[3] = 'F';
    header.fileLength = bufferSize * sizeof(short) + sizeof(WavHeader);
    header.fileType[0] = 'W';
    header.fileType[1] = 'A';
    header.fileType[2] = 'V';
    header.fileType[3] = 'E';
    header.formatHeader[0] = 'f';
    header.formatHeader[1] = 'm';
    header.formatHeader[2] = 't';
    header.formatHeader[3] = ' ';
    header.formatLength = 16;
    header.formatType = 1;
    header.channels = CHANNELS;
    header.sampleRate = SAMPLE_RATE;
    header.bitRate = BIT_RATE;
    header.bytesPerSample = BYTES_PER_SAMPLE;
    header.bitDepth = BIT_DEPTH;
    header.dataHeader[0] = 'd';
    header.dataHeader[1] = 'a';
    header.dataHeader[2] = 't';
    header.dataHeader[3] = 'a';
    header.dataLength = bufferSize * sizeof(short);

    // Write header and buffer to recording.wav on the website
    FILE* file = fopen("/var/www/html/recording.wav", "w");
    fwrite(&header, sizeof(WavHeader), 1, file);
    fwrite(buffer, sizeof(short), bufferSize, file);
    fclose(file);
}

/**
 * \brief Load .wav from website into buffer
 *
 * \param buffer        array into which audio samples are loaded
 *
 * \returns number of samples loaded into buffer
 */
```

```
size_t loadRecording(short* buffer)
{
    size_t samples = 0;
    FILE* file = fopen("/var/www/html/recording.wav", "r");

    // If the file on the website exists, read it into buffer
    if (file != NULL)
    {
        // Read in the header (not useful for us)
        fread(buffer, 1, 44, file);

        // Read in the data so that it overwrites the header
        samples = fread(buffer, sizeof(short), BUF_SIZE, file);
        fclose(file);
    }

    return samples;
}

/**
 * \brief Flash the LED a given number of times
 *
 * \param numFlashes        number of times to flash LED
 */
void flashLED(int numFlashes)
{
    for (int i = 0; i < numFlashes; ++i)
    {
        digitalWrite(PIN_LED, 1);
        usleep(FLASH_TIME * 1000);
        digitalWrite(PIN_LED, 0);
        usleep(FLASH_TIME * 1000);
    }
}

/**
 * \brief Gets the IP address of the microcontroller
 *
 * \param buffer to hold IP address in human-readable format
 */
void getIPAddress(char* retIP)
{
    struct ifaddrs* addrs;
    struct ifaddrs* tmp;

    // Get all network interfaces as a linked list
    getifaddrs(&addrs);
    tmp = addrs;

    // Iterate through network interfaces searching for the IP address
    while (tmp)
    {
        if (tmp->ifa_addr && tmp->ifa_addr->sa_family == AF_INET)
        {
            struct sockaddr_in *pAddr = (struct sockaddr_in *)tmp->ifa_addr;

            // If the address is not localhost, this is the IP; return it
            if(strcmp(inet_ntoa(pAddr->sin_addr), "127.0.0.1"))
            {
                strcpy(retIP, inet_ntoa(pAddr->sin_addr));
                freeifaddrs(addrs);
                return;
            }
        }
        tmp = tmp->ifa_next;
    }

    // If IP address was not found, return a placeholder so user knows to look it up manually
    freeifaddrs(addrs);
    strcpy(retIP, "<yourIPAddress>");
}
```

```c
/**
 * \brief Entry point for program
 */
int main()
{
    // Initialize peripherals
    init();

    // GPIO variables
    int recording = digitalRead(PIN_RECORD);
    int lastRecording = recording;
    int running = 0;
    int start = 0;
    int lastStart = 0;
    int reset = 0;
    int lastReset = 0;
    int save = 0;
    int lastSave = 0;
    size_t inputSamples = 0;

    // PWM variables
    float dut;

    // Recording variables
    size_t recordIndex = loadRecording(buffer);
    size_t playIndex = 0;
    char IPAddress[16];
    getIPAddress(IPAddress);

    // SPI variables
    int curNCS = digitalRead(NCS);
    int lastNCS = curNCS;
    int curSCLK;
    int lastSCLK;
    int reading;
    short input;
    short lastInput;
    int bitsIn;

    printf("starting...\n");

    // One iteration of this loop corresponds to one sample from the FPGA
    while (1)
    {
        ////////////////////////////////
        //  Handle GPIO
        ////////////////////////////////

        // Allow inputs to debounce before reading
        inputSamples++;
        if (inputSamples / (SAMPLE_RATE / 1000) > DEBOUNCE_TIME)
        {
            recording = digitalRead(PIN_RECORD);
            start = digitalRead(PIN_START);
            reset = digitalRead(PIN_RESET);
            save = digitalRead(PIN_SAVE);
            inputSamples = 0;
        }

        // if user switches between play and recording mode, stop playing/recording
        if (recording != lastRecording)
        {
            running = 0;
        }

        // Handle "start" button
        if (start && !lastStart)
        {
            running = !running;
        }

        // Handle "reset" button
        if (reset && !lastReset)
        {
            if (recording)
            {
```

```
            recordIndex = 0;
        }
        else
        {
            playIndex = 0;
        }
        running = 0;
        flashLED(1);
    }

    // Handle "save" button
    if (save && !lastSave)
    {
        printf("saving...\n");
        saveRecording(buffer, recordIndex);
        printf("your recording is available at http://%s/recording.wav\n", IPAddress);
        flashLED(3);
        running = 0;
    }

    // Update lastX variables so we only trigger on the raising edge of inputs
    lastRecording = recording;
    lastStart = start;
    lastReset = reset;
    lastSave = save;

    // Turn on LED if playing or recording
    digitalWrite(PIN_LED, running);




    ////////////////////////////////
    //  Calculate next dut
    ////////////////////////////////

    // If in play mode, combine the input with the recording
    if (!recording && running)
    {
        dut = ((float)input + buffer[playIndex]) / (1 << 15);
        playIndex++;

        // If we reach the end of the recording, stop playing
        if (playIndex >= recordIndex)
        {
            running = 0;
            playIndex = 0;
        }
    }
    else
    {
        dut = ((float)input) / (1 << 15);
    }

    // Bias dut so that it is alaways positive
    dut = (dut / 2) + 0.5;
```

```
        ////////////////////////////////
        //  Handle SPI
        ////////////////////////////////

        // Reset SPI variables
        bitsIn = 0;
        lastInput = input;
        input = 0;
        reading = 0;

        // Read from SPI
        while (1)
        {
            curNCS = digitalRead(NCS);

            if (reading)    // NCS is low and we are reading
            {
                curSCLK = digitalRead(SCLK);

                // Read one bit on the positive edge of SCLK
                if (!lastSCLK && curSCLK)
                {
                    input = (input << 1) + digitalRead(MOSI);
                    bitsIn++;

                    // Stop reading once NCS is raised or we read all bits
                    if (curNCS || bitsIn >= INPUT_BITS)
                    {
                        // Convert 11-bit sign-magnitude to 16-bit 2's complement
                        if ((input >> 10) & 0x1)
                        {
                            input = -(input & 0x3FF);
                        }
                        input *= VOLUME;

                        // Don't use the sample if the SPI transfer failed
                        if (curNCS)
                        {
                            input = lastInput;
                        }

                        // If recording, add the sample to the recording buffer
                        if (recording && running)
                        {
                            buffer[recordIndex] = input;
                            recordIndex++;

                            // If we fill up the recording buffer, stop recording
                            if (recordIndex >= BUF_SIZE)
                            {
                                running = 0;
                                recordIndex = 0;
                            }
                        }
                        break;
                    }
                }
                lastSCLK = curSCLK;
            }
            else if (lastNCS && !curNCS)    // We are waiting for NCS to go low
            {
                // Set output volume with PWM
                setPWM(SAMPLE_RATE / 2, dut);
                reading = 1;
                curSCLK = digitalRead(SCLK);
                lastSCLK = curSCLK;
            }
            lastNCS = curNCS;
        }
    }
}
```

Figure 24: C code for microcontroller program

Appendix C: Example Waveforms

The following waveforms were recorded and downloaded from our device using our recording and exporting features. We have used Audacity to visualize the signals.

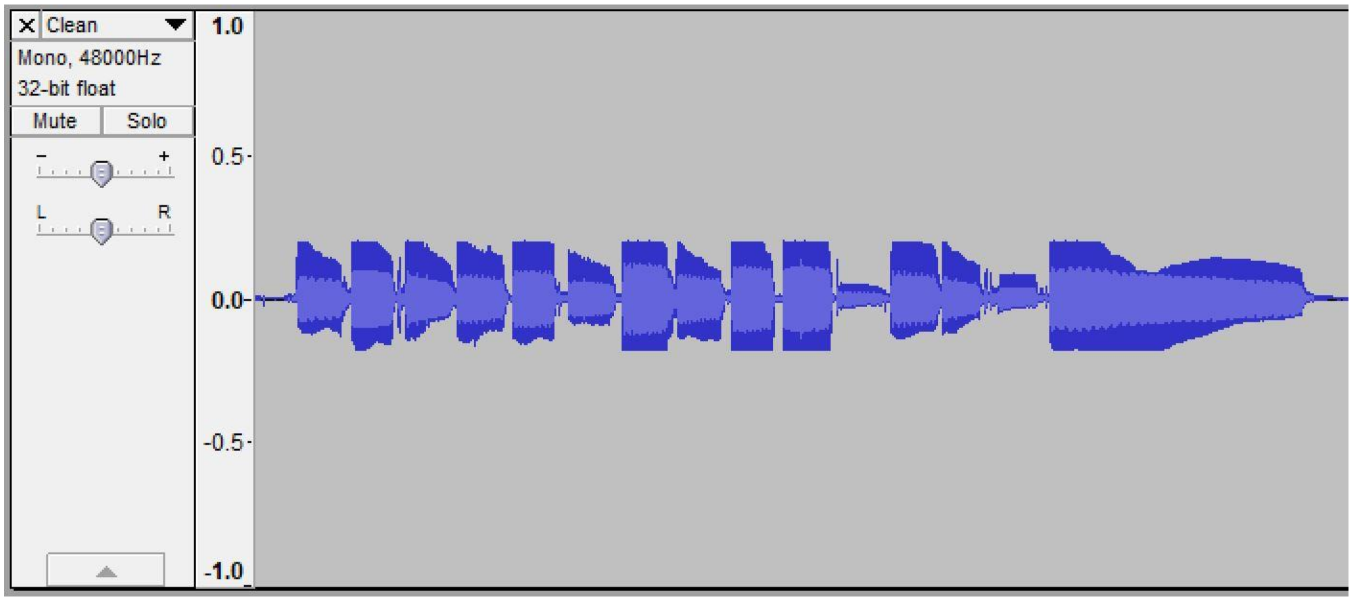


Figure 25: E major scale without effects

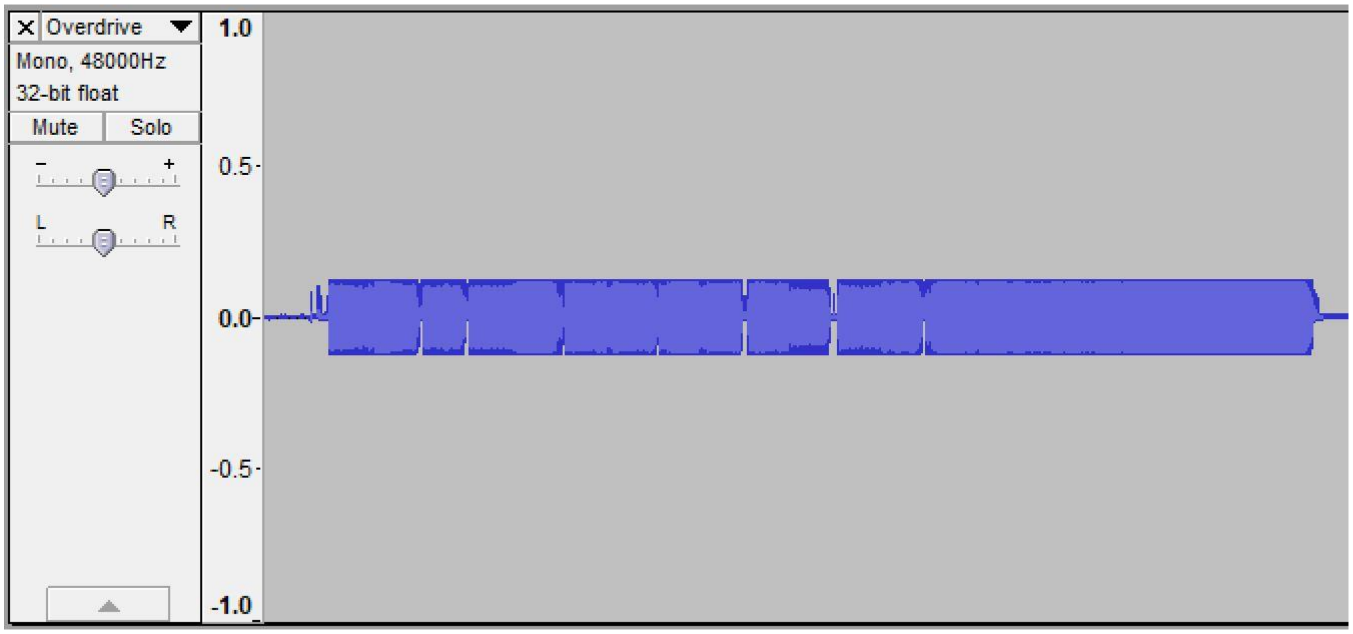


Figure 26: E major scale with overdrive effect applied (notice intentional clipping)

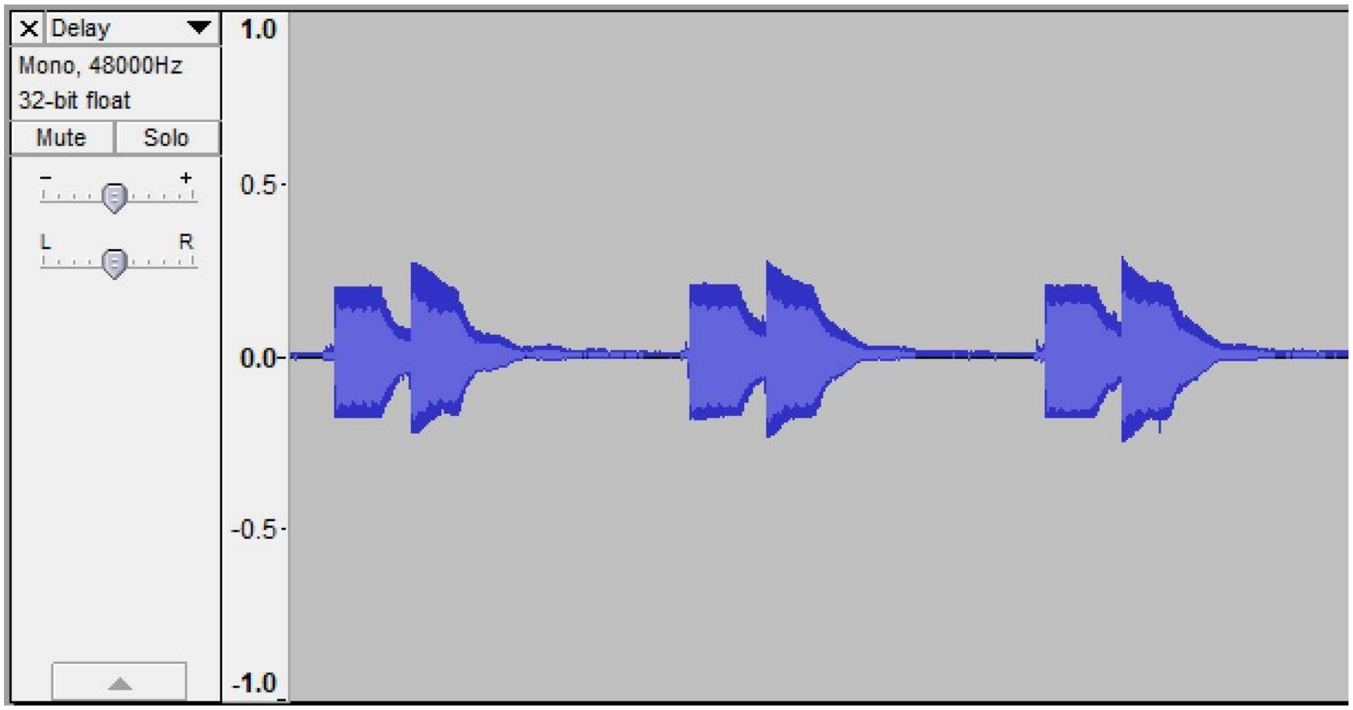


Figure 27: Three E notes with delay effect applied (notice repeated signal)

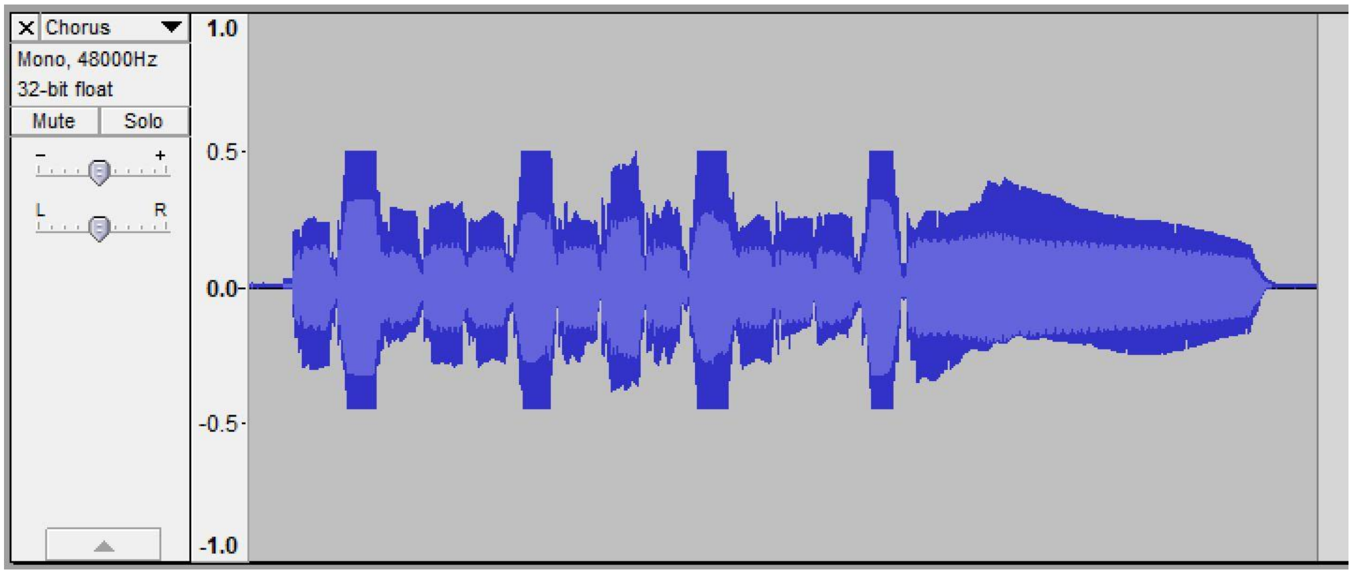


Figure 28: E major scale with chorus effect applied (notice softer edges)

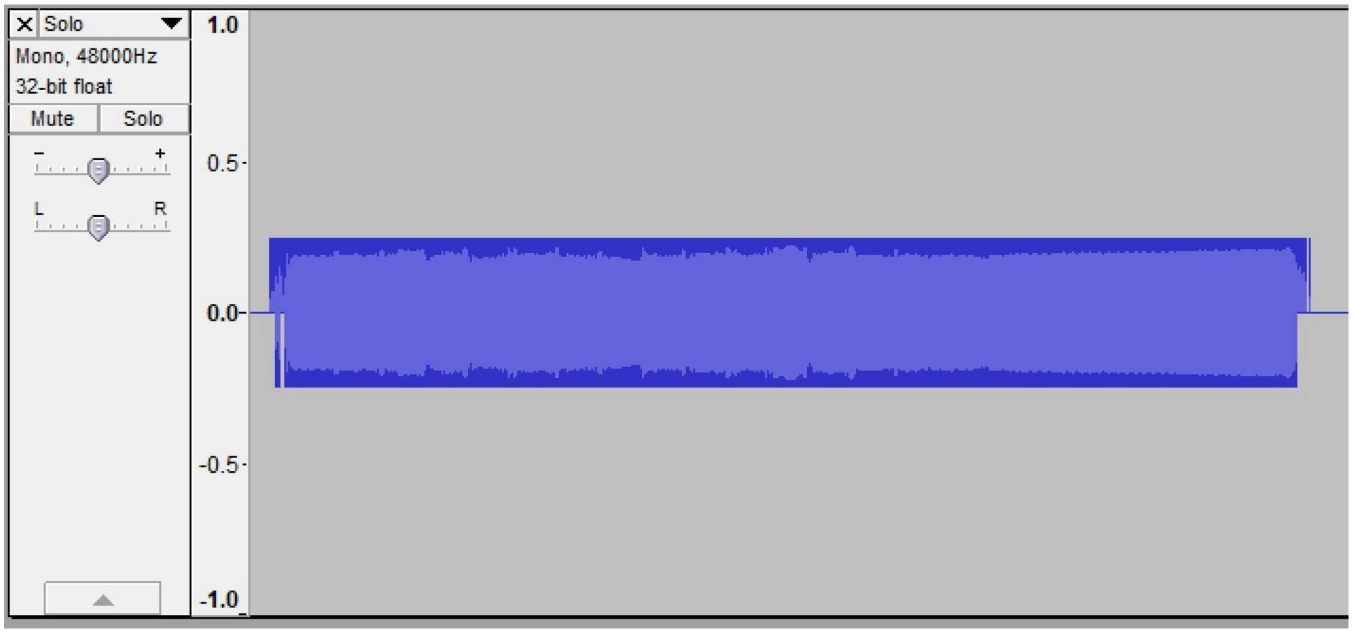


Figure 29: E major scale played with distortion effect applied (notice consistent amplitude)

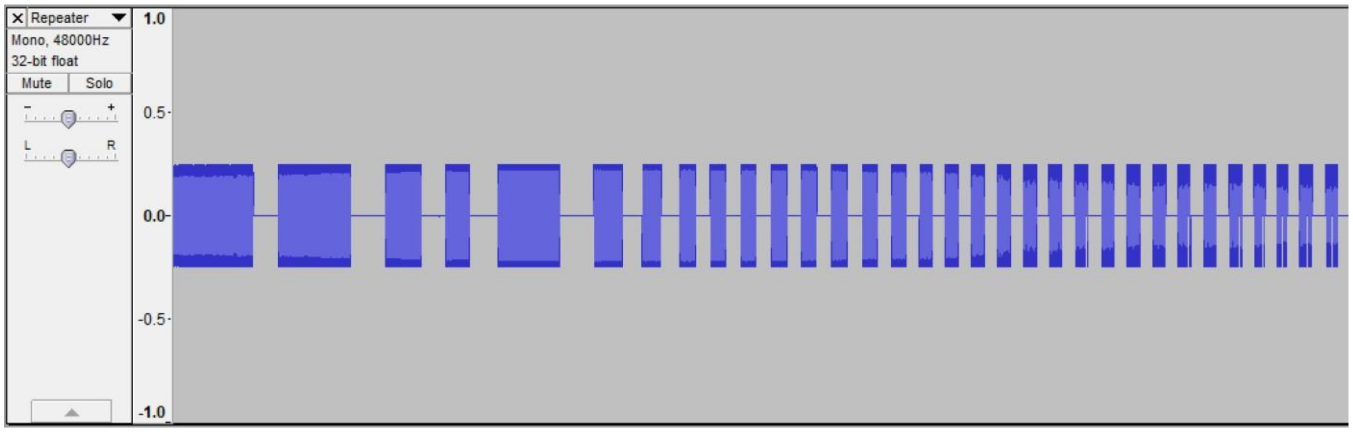


Figure 30: Repeater effect applied with increasing intensity from distance sensor

Appendix D: Miscellaneous

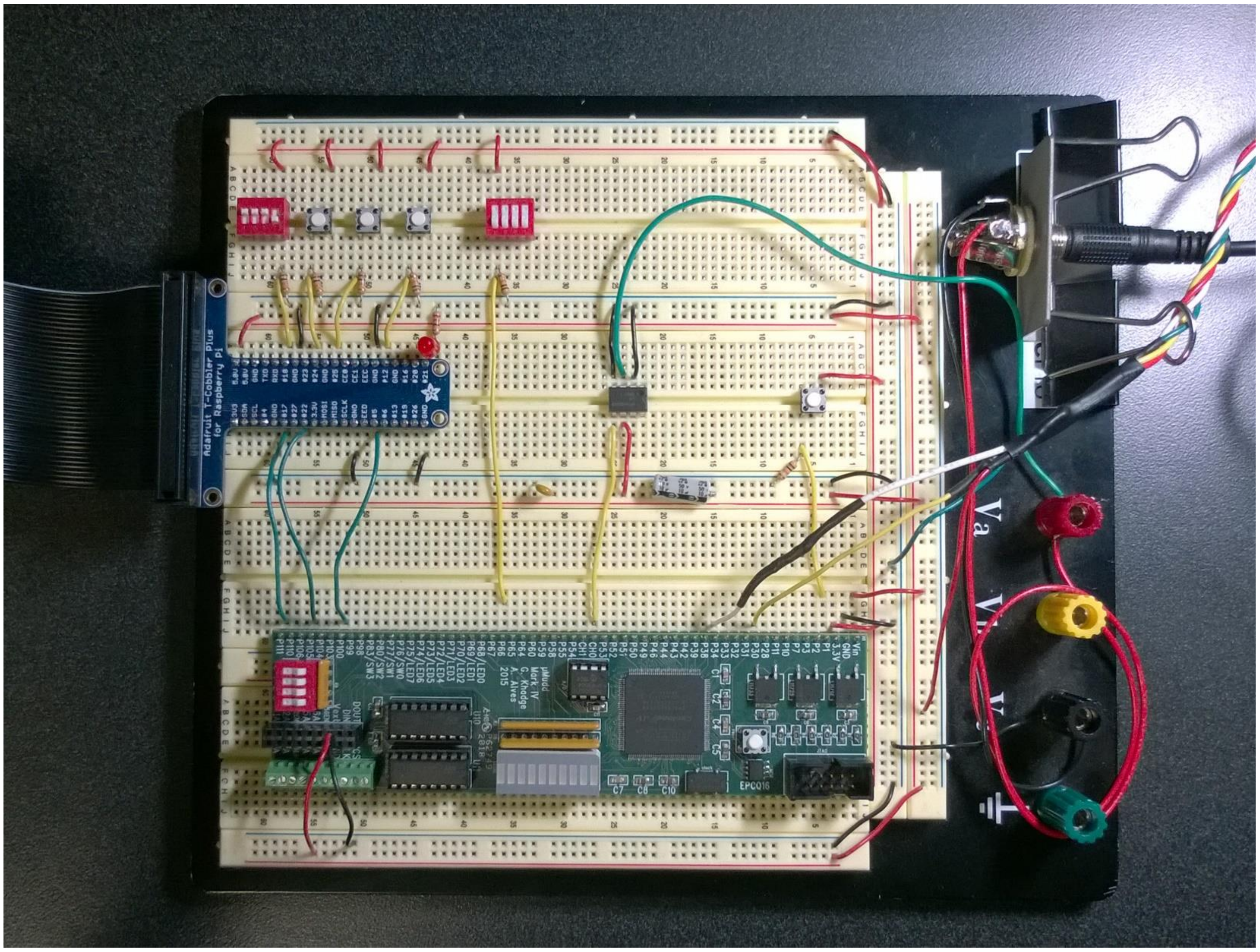


Figure 31: Photograph of final product

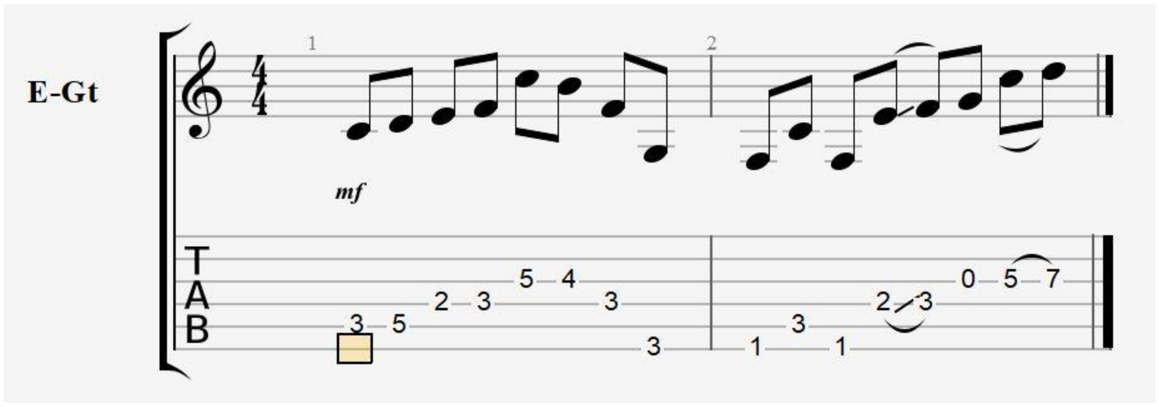


Figure 32: An example of a melody which auto-harmonizes with a quarter note delay