Lab X: Programmable Guitar Pedal

Author: John Poirier

Partner: Grant Floyd

Lab Dates: November 12 – December 3, 2024

Report Date: December 12, 2024

Introduction

The purpose of this lab was to create a unique project to explore the use of microcontrollers. For this project, a programmable guitar pedal was chosen and designed. This project utilized an Arduino UNO, an ADC, a DAC, Operational Amplifier, and various other circuit components, along with a guitar and a guitar amp. Using these components, a guitar pedal was designed and built, and various Arduino IDE codes were written to create desired effects. This task posed significant challenges and was unable to be fully implemented within the allotted time.

Project Goals

This project set out to create a programmable guitar pedal using an Arduino UNO and other components, allowing one to create custom effects for a guitar. This project ran into significant issues preventing many goals from being met directly.

The first goal of this project was to process an input signal to be read by the Arduino, and then process a signal from the Arduino to be read by the guitar amp. This utilized a signal processing circuit that is described in detail in the next section.

The next goal of the project was to have direct audio passthrough from using the Arduino. The Arduino would read the processed input from the guitar and pass it through to the output to be processed for the guitar amp. This phase is the source of issues for this project. These complications are described in detail in the *Arduino, ADC, and DAC Implementation* section.

The final goal was to implement effects using Arduino IDE code to manipulate the signal and create new sounds to output to the guitar amp. These codes were created and successfully tested but were not able to be fully implemented due to the complications caused by the output of the Arduino.

Signal Processing Circuit

The first step of this project was to create a circuit that processed the input from the guitar to be a suitable range to be manipulated by the Arduino. To do this, three op-amps were used to amplify and shift the signal to be within the Arduino's expected input range. The circuit used is shown in **figure 1** below.

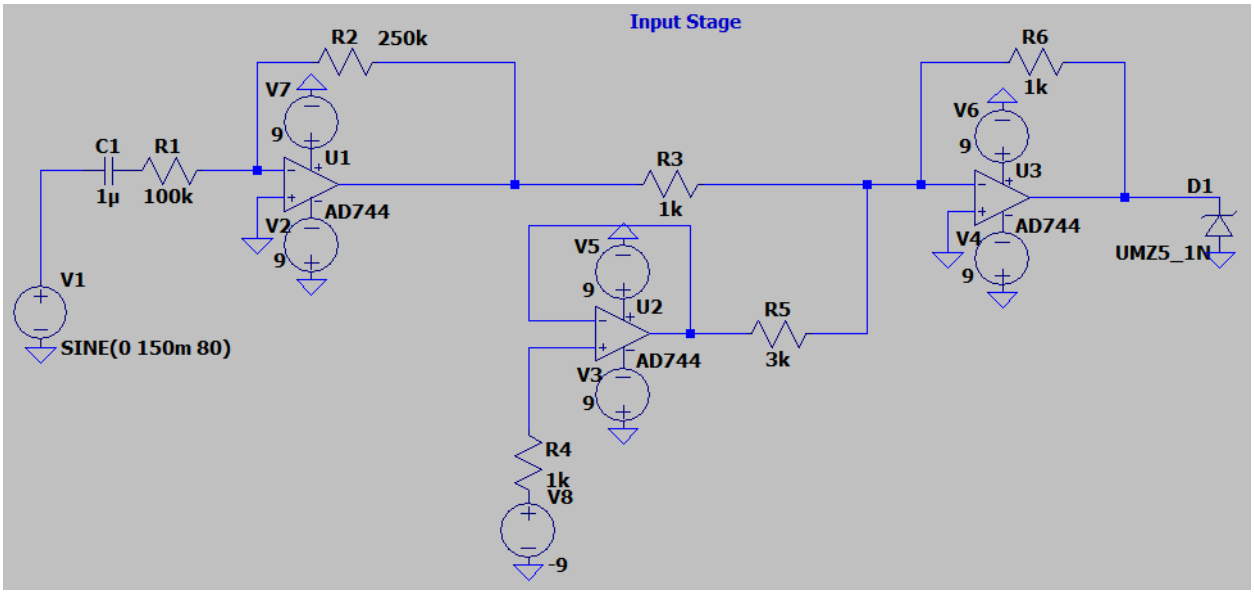

Figure 1: Input Processing Circuit Schematic

This circuit receives an input from the guitar, which typically ranges from 0-2V amplitude and anywhere from about 40 Hz to 5 kHz. This input goes through a capacitor to ensure a purely AC signal input, then into the first op-amp (from left to right). This op-amp amplifies and inverts the circuit. The ratio of R1 and R2 in the circuit schematic determines the gain of the op-amp, and the signal is connected to the inverting terminal of the op-amp.

After being amplified and inverted, the signal receives a shift down by about 2.5V. The second op-amp in the schematic adds a DC shift to the signal before that signal is processed by the final op-amp. This shifting component is powered by a 9V battery. All the op-amps in this project are powered by 9V batteries at saturation voltages, requiring two batteries, one for the positive saturation and one for the negative saturation.

Finally, the signal goes through the third op-amp, which inverts the signal again. Since the signal was shifted down by the DC component, this inversion causes that shift to be positive. This also un-does the inversion caused by the first op-amp. It is necessary to have the first op-amp invert the signal, as the AD741 is not able to produce high gains for a non-inverting configuration, and since a gain of about 25 is necessary for this processing, it must be done in an inverting configuration. After this processing, the signal that enters the Arduino has been shifted and amplified to be an appropriate range for the Arduino to process. **Figure 2** below shows a sample signal into the processing circuit compared to the signal after the processing circuit that goes to the Arduino.

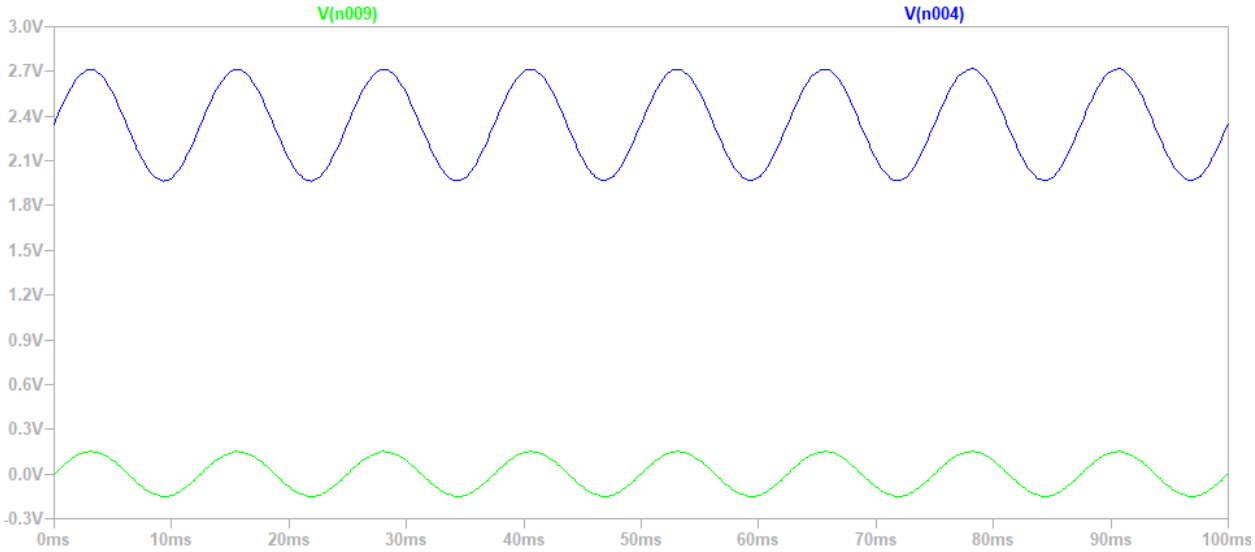

Figure 2: Sample Input (Green) and Pre-Processed Output (Blue) of Input Stage

Once the Arduino reads, manipulates, and outputs a signal, that Arduino output will have the same general range as the processed signal into the Arduino. However, the guitar amp expects a signal directly from the guitar, so the Arduino output signal must be processed to de-amplify and un-shift the Arduino output. This is accomplished using the circuit shown in the schematic in **figure 3** below.

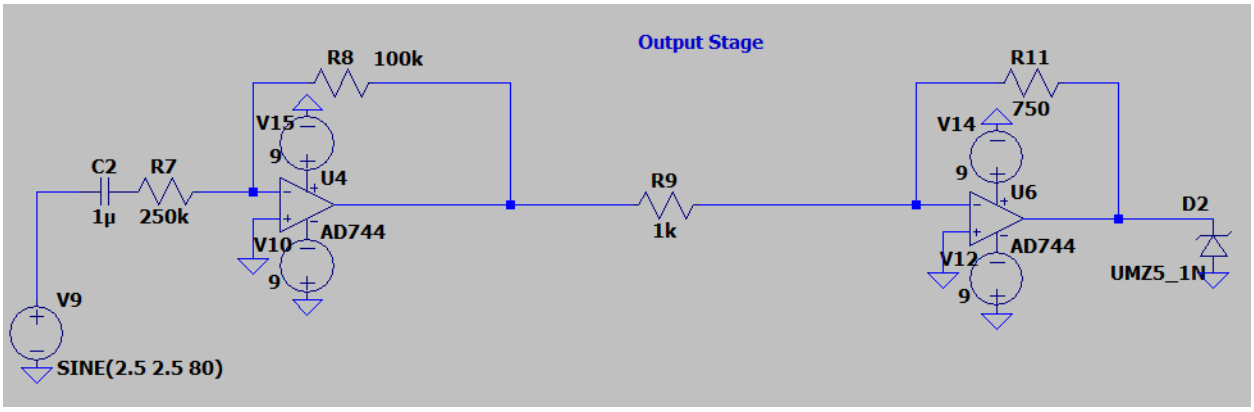

Figure 3: Output Processing Circuit Schematic

This circuit receives an input from the Arduino, which has a typical range of 0-5V. This input goes through a capacitor to ensure an AC signal, then into the first op-amp. This op-amp has a gain of 1/25, which reverses the gain from the input stage. This is also an inverting configuration, so the signal is inverted. The second op-amp un-inverts the signal and shifts the

signal down to be centered around 0 again. Applying the output of the input stage as the input to the output stage provides the following results shown in **figure 4**.

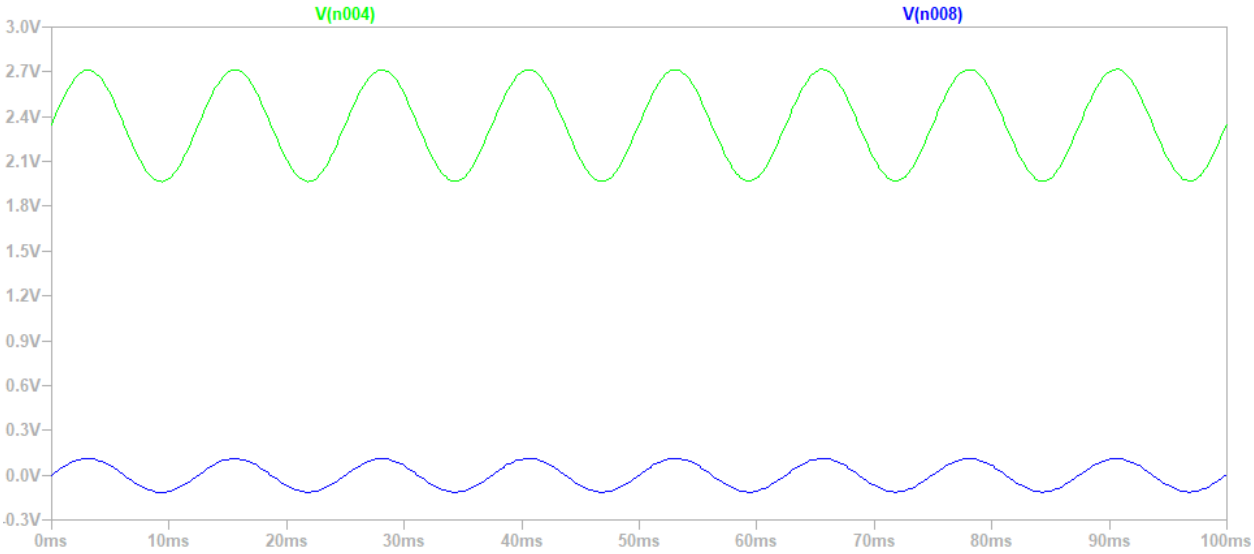

Figure 4: Sample Input (Green) and Post-Process Output (Blue) of Output Stage

Comparing the output of the output stage with the sample input to the input stage, the signal is accurately reconstructed after processing. Constructing the circuit physically and applying the same test signal produced the following waveforms as shown in **figure 5**.

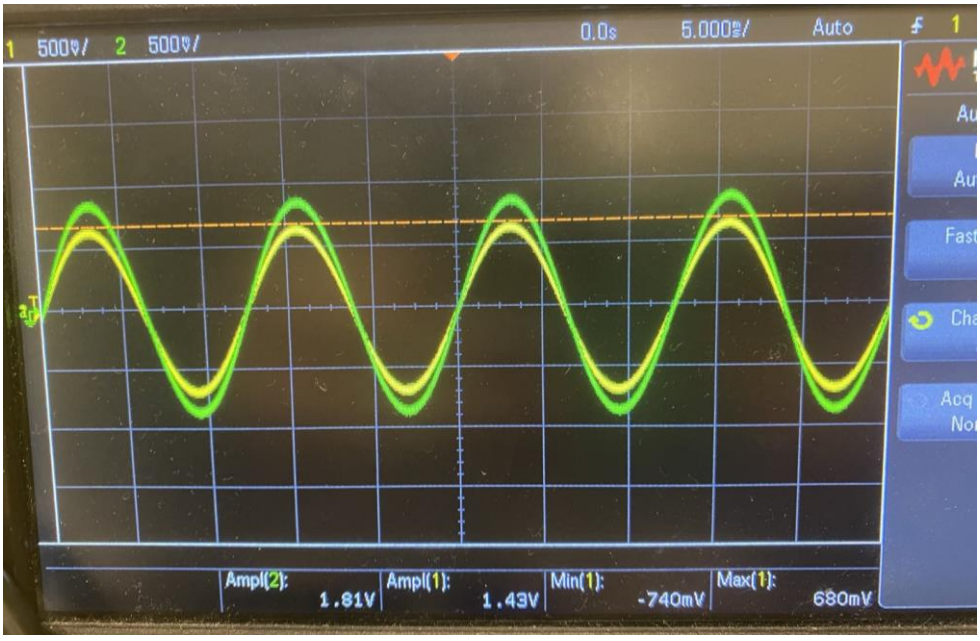

Figure 5: Sample Input (Green) and Processed Output (Yellow) Waveforms

The circuit successfully recreates the input signal to a reasonable degree of accuracy, enough to safely use the circuit in combination with the guitar, Arduino, and guitar amp.

Arduino, ADC and DAC Implementation

The next step was to implement the Arduino along with the ADC and DAC. This is a simple connection, as both the ADC and DAC use I2C communication with the Arduino. The Arduino only needs to connect to the ADC and DAC, the signal after the input stage processing connects to the ADC, and the output from the DAC connects to the input for the output processing stage. This final circuit is shown below in **figure 6**.

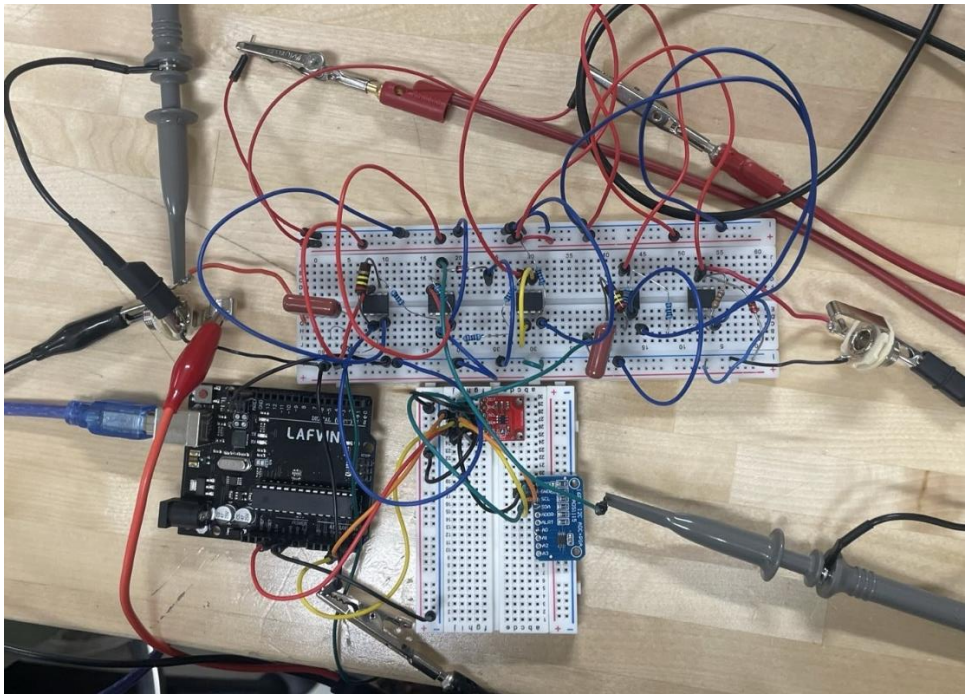

Figure 6: Completed Processing Circuit

Although the physical implementation of the Arduino, ADC, and DAC was simple, the functionality of these components left much to be desired. The circuit was fed a sample signal from the function generator to mimic a possible signal from the guitar, with the Arduino set to do audio passthrough, and the output of the DAC was measured, before going through the post-processing portion of the circuit. These measurements are shown below in **figure 7**.

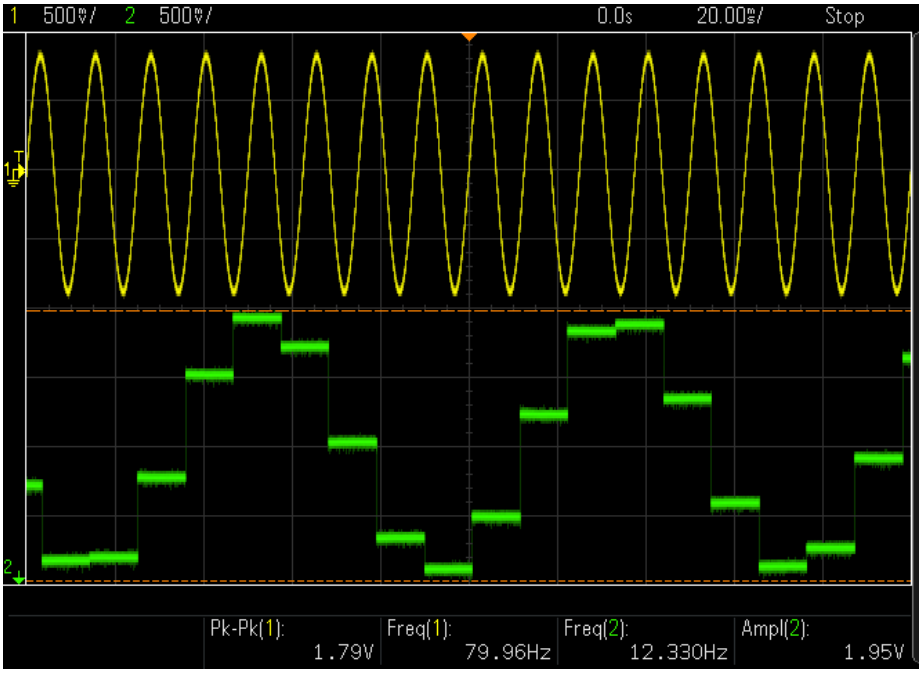

Figure 7: Waveforms of Sample Input (Yellow) and Output at DAC (Green)

Clearly, the output is not as desired for an audio passthrough. From these waveforms, it can be concluded that either the Arduino or the DAC cannot sample the signal fast enough or change the signal fast enough to accurately recreate the input signal. The form of a sine wave is present, but it is in discrete steps, far from a continuous signal. This issue was unable to be resolved with the hardware available, and severely limited the progress of this project.

Pedal Effects

Despite the limitations of the hardware complications of this project, codes for pedal effects were still created and tested. Simple codes that pass the signal directly without manipulating it were created for testing purposes. The first of these, labeled “Audio Passthrough V1” in the Appendix, does not utilize the ADC or DAC. The other, labeled “Audio Passthrough V2” in the Appendix, is compatible with the ADC and DAC circuit. Additionally, a simple test code was created to ensure the functionality of the ADC and DAC. This code is shown in the Appendix labeled “ADC_DAC Test”.

An Arduino IDE code was created to cause a delay in the output of the circuit, creating a delay pedal. This simplified version of a delayed pedal takes the input, and as the input is read, applies that to the output after a short buffer period. The code begins by setting up an array to read the values but starts with all zeros in the array. Then, as the code reads the inputs, it replaces those zeros with the inputs, and outputs the values from the array, creating a delay in the output.

This code is shown in the Appendix labeled “Delay Pedal”. This effect was tested with a sample input, and the output at the DAC was measured. This is shown below in **figure 8**.

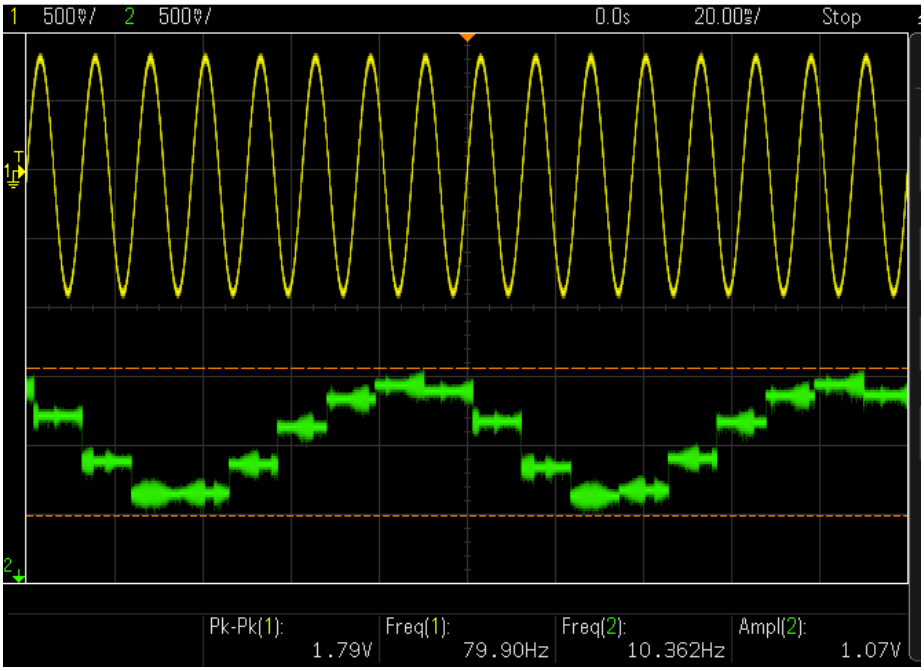

Figure 8: Sample Input (Yellow) and Delay Pedal Output at DAC (Green)

Another effect that was created for this project was a distortion pedal. For a guitar signal, distortion refers to clipping the signal at both extremes, causing a desired “unclean” sound. To accomplish this using the Arduino circuit, a simple code is constructed similar to the audio passthrough. However, in the distortion pedal, the output is constrained to a certain value depending on the desired intensity of the effect. This code is shown in the Appendix labeled “Distortion Pedal”. This effect was tested with a sample input, and the output at the DAC was measured. This is shown below in **figure 9**.

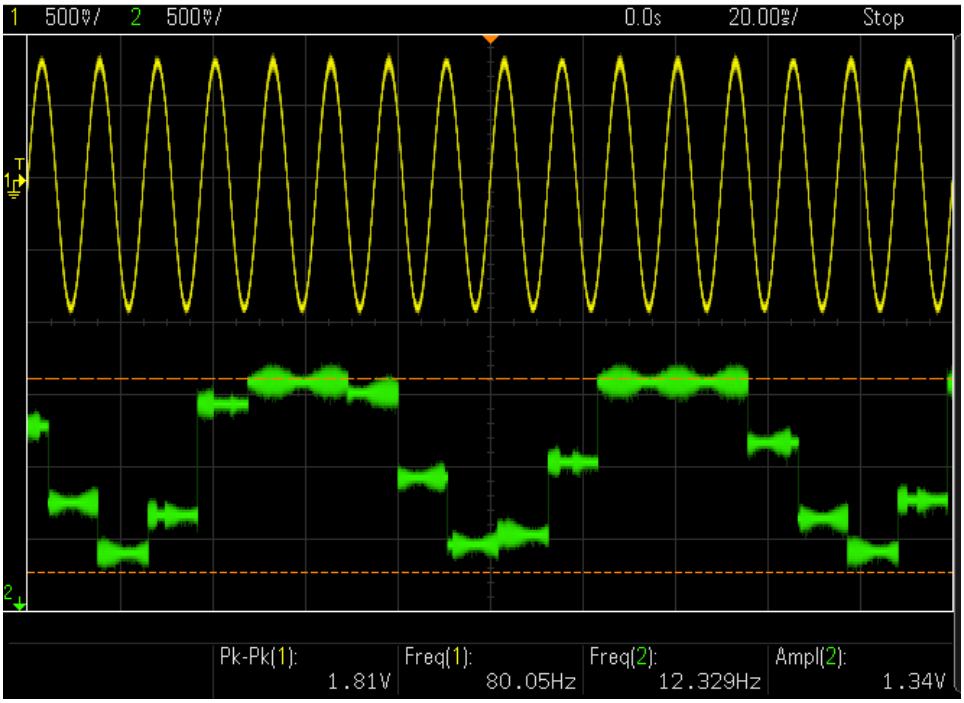

Figure 9: Sample Input (Yellow) and Distortion Pedal Output at DAC (Green)

Finally, a loop pedal effect was also created. Due to limited resources and the issues with the output of the Arduino and DAC, this function was not able to be tested but remains as a proof of concept for the project. This effect allows one to record inputs from the guitar, and playback that recording in a loop. This requires an SD card for additional memory, and utilizes two buttons, one to start and stop recording and playback, and one to reset the recording. When the record button is pressed, the code checks if there is already a recording, and if not, begins recording. Then, when the record button is pressed again, the recording is stopped. From then on, when the record button is pressed, the existing recording is played back in a loop. If the reset button is pressed, the existing recording is removed and the pedal is ready to start a new recording. This code is shown in the Appendix labeled “Loop Pedal”.

Challenges

As previously stated, the most significant challenge of this project lies in the limitations of the hardware used. Because of the Arduino and DAC, the output of the pedal is unable to match the input closely enough to produce the desired sound. Furthermore, none of the hardware used is designed to be used for audio processing. There exists ADCs, DACs, and op-amps that are all specifically designed for audio processing, which would likely provide much better results for this project. Additionally, an Arduino with faster processing, memory, and clock speed would likely provide significant benefits for the performance of this project.

Another point of conflict is that the signal entering the output processing circuit was not close enough to an AC signal, and so the op-amps were not able to properly process the signal. Noise also had a significant effect, since the input from the guitar is a relatively small signal, any noise is also amplified along with the signal, causing significant distortion in the output. Measuring the output of the final post-processing phase of the circuit provides the waveforms in **figure 10** below.

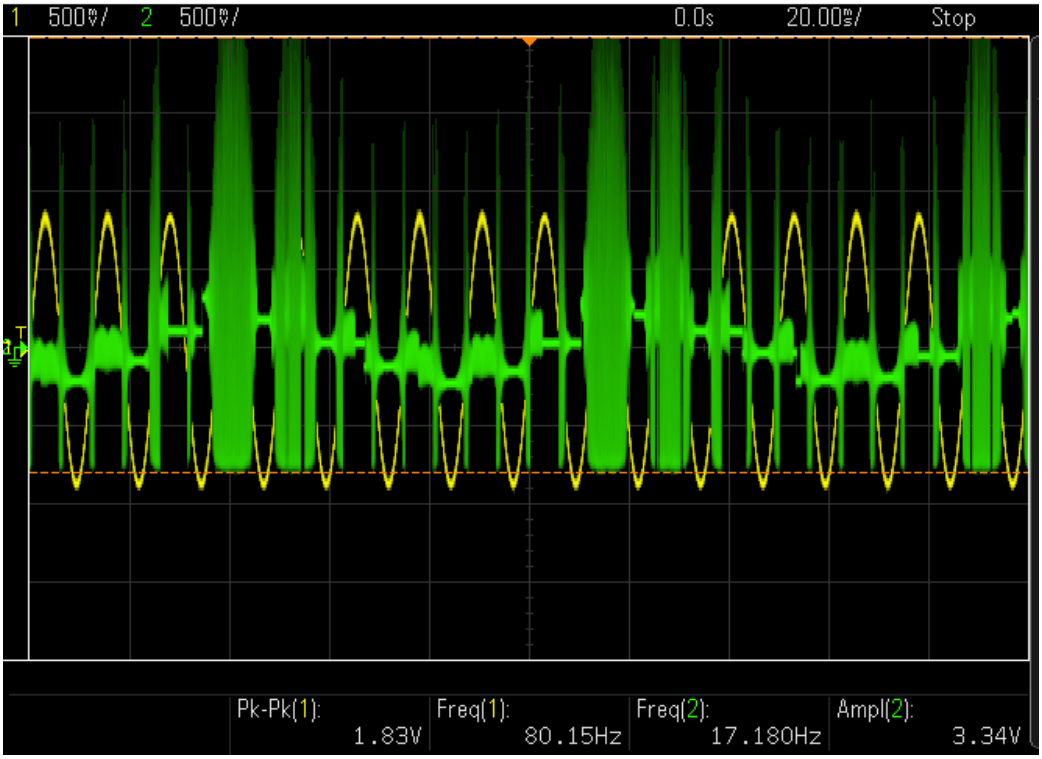

Figure 10: Output of Final Circuit (Green) Compared to Input (Yellow)

Continued Work

There are many areas for improvement for this project. As mentioned before, using components that are specifically designed for audio processing would allow for appropriate speeds of sampling for the desired output. After some research, one can quickly find components that would be more suitable for this project. For an ADC, the PCM186x series ADCs from Texas Instruments provides significant benefits over the ADS1115 ADC that was used. Similarly, the PCM5100A DAC, also from Texas Instruments, would be a significant upgrade over the MCP4725 DAC. A new model of op-amp would also be beneficial, switching the AD741 for an audio processing model, such as the TL072 from Texas Instruments should improve the quality of the signal processing.

Using a stronger Arduino would also provide benefits, with a higher memory and clock speed being able to change outputs more frequently, getting closer to an AC signal. An Arduino with a higher memory capacity could also allow for several effects to be loaded onto the Arduino at a time, which would allow for a more customizable pedal, being able to switch effects at any time.

Additionally, with more time, a custom PCB shield and case can be created for this circuit. Building this circuit on a breadboard with jumper wires becomes messy and introduces opportunities for connection issues. Creating a PCB would solve this problem, while also improving the form factor of the project, making it more practical.

Finally, given more time and access to sufficient resources and hardware, more effects could be implemented. More effects could be added through code alone, but some effects that require additional hardware could be implemented as well. These effects, which a microcontroller alone cannot produce, could be implemented by having multiple circuits that the Arduino could output to, still allowing one to switch between effects, and giving a much wider range of possibilities for effects.

Conclusion

Although this project was not able to meet every goal set for it, experience was gained in utilizing microcontrollers and Arduino specifically, as well as working with hardware to design a circuit to fit specific needs. This lab also explored setting goals for a project and working effectively for time management. A greater understanding of Arduino and its limitations and capabilities was gained through experimentation. Knowledge of physical and digital signal processing was gained as well, allowing for this project to continue to be worked on and improved.

Appendix

Code

Audio Passthrough V1

```
/*
John Poirier and Grant Floyd
Programmable Guitar Pedal
Microcontrollers Lab X
Task: Audio Passthrough without ADC or DAC (Arduino and Op-Amp circuit only)
*/

const int inputPin = A0;  // Analog input from guitar
const int outputPin = 9; // PWM output to amp

void setup() {
    pinMode(outputPin, OUTPUT);
    pinMode(inputPin, INPUT_PULLUP);
}

void loop() {
    int sensorValue = analogRead(inputPin); //Read signal from guitar
    int outputValue = map(sensorValue, 0, 1023, 0, 255); // Map to 8-bit PWM
    analogWrite(outputPin, outputValue); //Output to amp
}
```

Audio Passthrough V2

```
/*
John Poirier and Grant Floyd
Programmable Guitar Pedal
Microcontrollers Lab X
Task: Audio Passthrough using ADC and DAC
*/

#include <Wire.h> // Library for I2C Communication
#include <Adafruit_ADS1X15.h>  // Library for ADS1115 ADC
#include <Adafruit_MCP4725.h>  // Library for MCP4725 DAC

// Initialize ADC and DAC
Adafruit_ADS1115 adc;
Adafruit_MCP4725 dac;
```

```
void setup() {
    Wire.begin();

    // Initialize the ADC
    adc.begin();

    // Initialize the DAC
    dac.begin(0x60);  // I2C Address for DAC

    // Set the ADC gain to match the expected input signal range
    // GAIN_ONE = +/- 4.096V input range (This is the expected input range after the first stage of Op-Amp Circuit)
    adc.setGain(GAIN_ONE);
    Wire.setClock(400000);
}

void loop() {
    int16_t adcValue = adc.readADC_SingleEnded(0); // Read guitar input from ADC
    uint16_t dacValue = map(adcValue, 0, 32767, 0, 4095); // Map from ADC ranges to DAC ranges
    dacValue = constrain(dacValue, 0, 4095); // Ensure output is of safe value
    dac.setVoltage(dacValue, false); // Set output
}
```

ADC_DAC Test

```
/* John Poirier and Grant Floyd
Programmable Guitar Pedal
Microcontrollers Lab X
Task: ADC and DAC test
*/

#include <Wire.h> // Library for I2C Communication
#include <Adafruit_ADS1X15.h> // Library for ADS1115 ADC
#include <Adafruit_MCP4725.h> // Library for MCP4725 DAC

Adafruit_ADS1115 adc;
Adafruit_MCP4725 dac;

void setup() {
  Wire.begin();

  // Initialize ADC
  adc.begin();
```

```
  adc.setGain(GAIN_ONE); // Set gain value

  // Initialize DAC
  dac.begin(0x60); // I2C address for MCP4725

  // Test Output
  dac.setVoltage(2048, false);
  delay(500);
}

void loop() {
  int16_t adcValue = adc.readADC_SingleEnded(0);  // Read from ADC

  uint16_t dacValue = map(adcValue, -32768, 32767, 0, 4095);  // Map ADC range to
DAC range

  dac.setVoltage(dacValue, false); // Read and output to DAC

  delay(100);
}
```

Delay Pedal

```
/*
Grant Floyd and John Poirier
Programmable Guitar Pedal
Microcontrollers Lab X
Task: Delay Pedal
*/

#include <Wire.h> // Library for I2C Communication
#include <Adafruit_ADS1X15.h>  // Library for ADS1115 ADC
#include <Adafruit_MCP4725.h>  // Library for MCP4725 DAC

// Initialize ADC and DAC
Adafruit_ADS1115 adc;
Adafruit_MCP4725 dac;

const int bufferSize = 512; // Buffer size
int16_t delayBuffer[bufferSize];
int bufferIndex = 0;
int delayTime = 125; // Adjusted delay time in milliseconds

const int sampleRate = 4000; // Define sampling rate (Hz)
```

```
void setup() {
    Wire.begin();

    // Initialize the ADC (ADS1115)
    adc.begin();
    adc.setGain(GAIN_ONE); // Set the gain to match expected input signal range

    // Initialize the DAC (MCP4725)
    dac.begin(0x60);  // I2C address for MCP4725

    Wire.setClock(400000); // Set I2C clock speed

    // Initialize delay buffer with zeros
    for (int i = 0; i < bufferSize; i++) {
        delayBuffer[i] = 0;
    }
}

void loop() {
    int16_t adcValue = adc.readADC_SingleEnded(0); // Read input from ADC
    delayBuffer[bufferIndex] = adcValue; // Store in Buffer

    // Calculate delayed output signal
    int delayedIndex = (bufferIndex + bufferSize - (delayTime * sampleRate /
1000)) % bufferSize;
    int16_t delayedValue = delayBuffer[delayedIndex];

    uint16_t dacValue = map(delayedValue, -32768, 32767, 0, 4095); // Map from
ADC to DAC values
    dacValue = constrain(dacValue, 0, 4095); // Ensure safe output
    dac.setVoltage(dacValue, false); // Output to DAC

    bufferIndex = (bufferIndex + 1) % bufferSize; // Increment buffer index
    delayMicroseconds(1000000 / sampleRate); // Delay to match sampling rate
}
```

Distortion Pedal

```
/*
John Poirier and Grant Floyd
Programmable Guitar Pedal
Microcontrollers Lab X
Task: Distortion Pedal
*/

#include <Wire.h> // Library for I2C Communication
#include <Adafruit_ADS1X15.h>  // Library for ADS1115 ADC
#include <Adafruit_MCP4725.h>  // Library for MCP4725 DAC

// Initialize ADC and DAC
Adafruit_ADS1115 adc;
Adafruit_MCP4725 dac;

const int constraint_value = 2500;
void setup() {
    Wire.begin();

    // Initialize the ADC
    adc.begin();

    // Initialize the DAC
    dac.begin(0x60);  // I2C Address for DAC

    // Set the ADC gain to match the expected input signal range
    // GAIN_ONE = +/- 4.096V input range (This is the expected input range after the first stage of Op-Amp Circuit)
    adc.setGain(GAIN_ONE);
    Wire.setClock(400000);
}

void loop() {
    int16_t adcValue = adc.readADC_SingleEnded(0); // Read guitar input from ADC
    uint16_t dacValue = map(adcValue, 0, 32767, 0, 4095); // Map from ADC ranges to DAC ranges
    dacValue = constrain(dacValue, 0, constraint_value); // Limit outuput significantly to get clipping
    dac.setVoltage(dacValue, false); // Set output
}
```

Loop Pedal

```
/*
John Poirier and Grant Floyd
Programmable Guitar Pedal
Microcontrollers Lab X
Task: Loop Pedal Concept Code (Untested, requires SD Card)
*/

#include <SD.h> // Library for SD Card
#include <SPI.h> // Library for SPI Communication

// Pin Definitions
const int recordButton = 2;   // Start/stop recording and playback
const int resetButton = 3;    // Reset recording
const int audioInputPin = A0; // Guitar audio input
const int audioOutputPin = 9; // Guitar audio output
const int chipSelect = 10;    // SD card select pin

// State Variables and Flags
bool isRecording = false;
bool isPlaying = false;
unsigned long startTime = 0;
unsigned long endTime = 0;
const unsigned long sampleRate = 44100; // Sampling rate for audio (Hz)
unsigned long lastSampleTime = 0;

File audioFile; //Audio File

void setup() {
    // Set pin modes
    pinMode(recordButton, INPUT_PULLUP);
    pinMode(resetButton, INPUT_PULLUP);
    pinMode(audioInputPin, INPUT);
    pinMode(audioOutputPin, OUTPUT);

    // Initialize SD card
    if (!SD.begin(chipSelect)) {
        Serial.begin(9600);
        Serial.println("SD card initialization failed!");
        while (true); // Stop execution if SD card fails
    }
    Serial.println("SD card initialized.");
}
```

```
void loop() {
    // Record/Play Button Handling
    if (digitalRead(recordButton) == LOW) {
        delay(200); // Button debouncing
        if (!isRecording && !isPlaying) {
            startRecording(); //Begin recording
        } else if (isRecording) {
            endRecordingStartPlayback(); // Stop recording, play recorded audio
        } else if (isPlaying) {
            stopPlayback(); // Stop playback
        }
    }

    // Reset Button Handling
    if (digitalRead(resetButton) == LOW) {
        delay(200); // Debounce
        resetLoop(); // Reset functions
    }

    // Recoridng/Playback Handling
    if (isRecording) {
        recordAudio(); // Record
    }
    if (isPlaying) {
        playbackAudio(); // Playback
    }
}

// Start Recording
void startRecording() {
    Serial.println("Recording begin");
    // Set flags
    isRecording = true;
    isPlaying = false;
    audioFile = SD.open("audio.raw", FILE_WRITE); // Open file
    if (!audioFile) { // Error handling
        Serial.println("Error opening file for recording!");
        isRecording = false;
    }
    startTime = micros(); //Begin timing
}

// Stop Recording and Start Playback
void endRecordingStartPlayback() {
    Serial.println("Recording stopped. Starting playback.");
```

```
    //Set Flags
    isRecording = false;
    isPlaying = true;
    endTime = micros();// Stop timing
    if (audioFile) {
        audioFile.close(); // Close audio file
    }
    audioFile = SD.open("audio.raw", FILE_READ); //Open file to read
    if (!audioFile) { // Error handling
        Serial.println("Error opening file for playback!");
        isPlaying = false;
    }
}

// Stop Playback
void stopPlayback() {
    Serial.println("Playback stopped.");
    // Set Flags
    isPlaying = false;
    if (audioFile) { //Close file
        audioFile.close();
    }
}

// Reset
void resetLoop() {
    Serial.println("Loop reset.");
    // Set Flags
    isRecording = false;
    isPlaying = false;
    if (audioFile) { // Close file
        audioFile.close();
    }
    SD.remove("audio.raw"); // Delete existing loop file
}

// Record Audio
void recordAudio() {
    unsigned long currentTime = micros();// Check start time
    if (currentTime - lastSampleTime >= (1000000 / sampleRate)) {
        lastSampleTime = currentTime; // set sample time
        int sample = analogRead(audioInputPin) >> 2; // 10-bit input to 8-bit
output
        audioFile.write(sample); // Write to SD card file
    }
```

```
}

// Playback Audio
void playbackAudio() {
    if (audioFile.available()) { // If file available
        int sample = audioFile.read(); // Read audio file
        analogWrite(audioOutputPin, sample); // Write to output from audio file
    } else {
        Serial.println("End of loop reached. Restarting playback");
        audioFile.seek(0); // Restart playback
    }
}
```