EE Senior Capstone - Surface Electromyogram-Controlled (sEMG) Analog Audio Synthesizer

Helen Bovington, Anya Trumbach, Daryl Choo, Kayla Montgomery

May 15, 2025

Contents

1 Introduction

The surface electromyogram (sEMG)-controlled audio synthesizer is a set of wearable sleeves that make real-time effects on sound output through selected arm movements and hand gestures, as outlined in the "List of Movements" section. The sleeve allows users to select different waveforms, control amplitude and frequency, and add a sustaining effect to the audio, creating a vibrato effect. Together, these features make the device a flexible, non-restrictive musical instrument.

To achieve this functionality, this project uses two sEMG sensors and accelerometer (xyz-axis) data from a 9DOF sensor placed on key forearm and upper arm muscles (see images A & B) to collect training data. This data was used to train a machine learning model that maps muscle activity to specific selector commands. Inside the sleeve is an ESP32 chip paired with a calibration system used to normalize amplitude output of muscle movements, making the system flexible to different users. The sensors interface with ESP32 chips, which transmit data via wireless Bluetooth Light. The server ESP32 chip directly feeds the control voltages to the analog circuit, allowing the user to modify the waveform in real time. More details on the analog circuit are provided in section 6.

2 List of Movements

Hand or Arm Movement	Left Arm - Hand & Forearm Orientation (Begin with your arm at a 90 degree angle, palm facing down.)	Right Arm - Hand & Forearm Orientation (Begin with your arm at a 90 degree angle, palm facing down.)
Arm	**Amplitude (Volume) → Max. Volume:** Bend your elbow and raise your forearm up until your forearm is perpendicular to the ground. **Min. Volume:** Straighten your elbow until your forearm is perpendicular to the ground.	Pitch → **Max. Pitch:** Bend your elbow and raise your forearm up until your forearm is perpendicular to the ground. **Min. Pitch:** Straighten your elbow until your forearm is perpendicular to the ground.
Hand State 1	**Make a Fist:** Close your hand into a fist.	**Sustain (Vibrato Effect) → Make a Fist:** Close your hand into a fist.
Hand State 2	**Extend motion:** Bend your wrist backwards until your palm is facing away from you.	
Hand State 3	**Flex Motion:** Bend your wrist forward until your palm is facing you.	

*Note: Any arm movement corresponds to the accelerometer from the 9DOF sensor, any hand movement corresponds to the EMG sensor.

3 System Block Diagram

The blow block diagram shows the whole system integration including the user, the 9DOF and sEMG sensors, ESP32 chips, communication protocol, analog circuit, and power amp leading to the speaker.

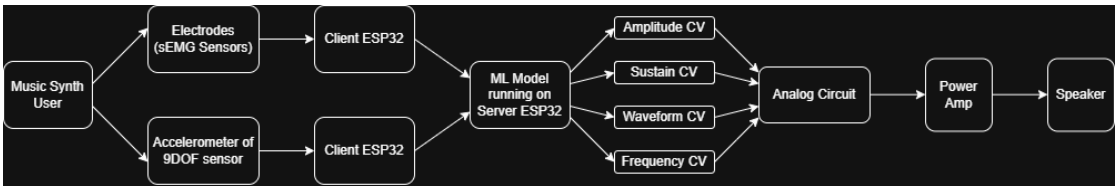


Figure 1: System Block Diagram

4 Machine Learning/State Detection

Our method for movement classification was to determine the most effective machine learning model. We tested linear discriminant analysis and the k-nearest neighbor algorithm, but in the end decided to use decision tree learning. The following sections include our code for developing the decision tree programmed in the left armband. EMG data is notoriously messy, so efforts were made to make the initial data collection as consistent as possible. We also removed any outliers before running the following code.

4.1 Decision Tree Development Code

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = pd.read_csv("rawMovementData.csv")

# Separate features (EMG data) and labels
X = data.iloc[:, 1:3]  # Features
y = data.iloc[:, 0]   # Labels

# X = X / 100 # Divide all values in X by 100

print("Number of rows:", X.shape[0])
print("Max value in X:", X.max().max())  # Gets the highest value across all columns

# Drop rows with NaN values in 'y_train' and corresponding rows in 'X_train'
X = X[y.notna()]
y = y[y.notna()]

X = X[(X <= 800000).all(axis=1)]
y = y.loc[X.index]  # Ensure labels match filtered features

# Split the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define a DecisionTreeClassifier and the parameter grid for GridSearch
clf = DecisionTreeClassifier(random_state=42)
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [6],
    'max_features': [None],
    'min_samples_leaf': [15],
    'min_samples_split': [2],
    'splitter': ['best']
}

# Drop rows with NaN values in 'y_train' and corresponding rows in 'X_train'
x_train = x_train[x_train.notna().any(axis=1)]  # Get rows where any column is not NaN
y_train = y_train[x_train.index]

smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(x_train, y_train)

scaler = StandardScaler()
X_train = scaler.fit_transform(x_train)
X_test = scaler.transform(x_test)
```

```
# Perform GridSearchCV for optimal hyperparameters
grid_search = GridSearchCV(clf, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best model and evaluate it
best_clf = grid_search.best_estimator_
y_pred = best_clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print("Best Parameters:", grid_search.best_params_)
print("Test Accuracy:", accuracy)
```

Sample Output Number of rows: 6638 Max value in X: 1143854 Number of rows after dropping 0s: 6638 Number of rows after filtering: 6562 Best Parameters: 'criterion': 'entropy', '$\max_{d}epth'$: 6,'$max_{f}eatures'$: $None$,'$min_{s}amples_{l}eaf'$: 15,'$min_{s}amples_{s}plit'$: 2,'$splitter'$:'$best'TestAccuracy$: 0.88

4.2 Training the Decision Tree

```
# Train the DecisionTreeClassifier with the best parameters
clf = DecisionTreeClassifier(criterion='entropy', max_depth=6, max_features=None, min_samples_leaf=15, min_sa
bestclf.fit(X, y)

# Extract tree structure as text
tree_rules = export_text(clf, feature_names=["EMG1", "EMG2"])

print("Decision Tree Rules:")
print(tree_rules)

Sample Output:
    |--- EMG1 <= 97.50
    |   |--- EMG2 <= 585.50
    |   |   |--- class: 0
    |   |--- EMG2 >  585.50
    |   |   |--- EMG2 <= 99774.50
    |   |   |   |--- EMG2 <= 56678.50
    |   |   |   |   |--- EMG2 <= 18637.00
    |   |   |   |   |   |--- EMG2 <= 13778.50
    |   |   |   |   |   |   |--- class: 0
    ...
```

Lastly, the full decision tree is converted to the if/else statements in Section 5.2. The possible states returned by the decision tree are 0, 1, 2, 3, and 4. These states correspond to no movement, wrist flexion, making a fist, wrist extension, and wrist rotation. Rotate was not used in the final movement vocabulary and instead resulted in no changes to the sound in the same way that no movement would.

5 Communications Description

5.1 ESP32 Master Communication Protocol

```
// Code edited from https://randomnerdtutorials.com/esp32-ble-server-client/

#include <Wire.h>
#include <Arduino.h>
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>
#include <string>

// BLE Server Name
```

```
#define bleServerName "ESP32_Main"

// ESP32 Pin Definitions
#define MUX0_BIT_0 18
#define MUX0_BIT_1 19
#define VCA 22
#define VCO 23
#define Sustain 25

BLEServer *pServer = NULL;
BLECharacteristic *currentCharacteristic = NULL;
bool deviceConnected = false;
bool newValue1 = false;
bool newValue2 = false;
int leftState = 0;
int rightState = 0;
float leftDOF = 0.0;
float rightDOF = 0.0;
float xAccel = 0.0;
int sound = 0;

#define SERVICE_UUID "3b7ce6b7-3190-4d39-b5dd-01df08fef25f"
#define EMG1_UUID "0a6686f6-1e4b-485e-a252-3c353e0c875a"
#define EMG2_UUID "a7a50cff-be29-48c6-81eb-406b39fe602b"

BLECharacteristic emg1Characteristics(EMG1_UUID, BLECharacteristic::PROPERTY_WRITE);
BLECharacteristic emg2Characteristics(EMG2_UUID, BLECharacteristic::PROPERTY_WRITE);

//Callback Functions
class MyServerCallbacks: public BLEServerCallbacks{
    void onConnect(BLEServer* pServer){
      deviceConnected = true;
    };
    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
    }
};

class MyCallbacks : public BLECharacteristicCallbacks {
  void onWrite(BLECharacteristic *pCharacteristic) {
    std::string currentValue = pCharacteristic->getValue();
    int split_index_0 = currentValue.find(",");
    int split_index_1 = currentValue.find(",", split_index_0 + 1);
    std::string current_UUID = (pCharacteristic->getUUID()).toString();
    int length = currentValue.length();
    // leftState will be returned as 1, 2, 3, or 4
    // rightState will be returned as 1 or 2
    if (length > 0) {
      if (current_UUID == EMG1_UUID){
        leftState = std::stoi(currentValue.substr(0, split_index_0));
        leftDOF = std::stoi(currentValue.substr(split_index_0 + 2, length));
        newValue1 = true;
      } else if (current_UUID == EMG2_UUID){
        rightState = std::stoi(currentValue.substr(0, split_index_0));
        rightDOF = std::stoi(currentValue.substr(split_index_0 + 1, split_index_1));
        float xAccel = std::stoi(currentValue.substr(split_index_1+1, length));
        Serial.print("YAccel: ");
        Serial.print(rightDOF);
        Serial.print(", xAccel: ");
        Serial.println(xAccel);
        if (xAccel <= 4.0){
```

```
            sound = 1;
        } else {
            sound = 0;
        }
        newValue2 = true;
      }
    } else {
        Serial.println("Device Values Not Received");
    }
  }
};

float findControlV(float dof, float range){
  float offset = 9.8;
  float dof_range = 9.8*2;
  float controlVal = (dof+offset)*range/dof_range;
  if (controlVal <= 0.0) {
    return 0.0;
  } else if (controlVal >= 250){
    return 250;
  } else {
    return controlVal;
  }
}

void changeWaveform(int waveform){
    if (waveform == 1){
      // sine
        digitalWrite(MUX0_BIT_0, LOW);
        digitalWrite(MUX0_BIT_1, LOW);
    }
    if (waveform == 2) {
      // triangle
        digitalWrite(MUX0_BIT_0, LOW);
        digitalWrite(MUX0_BIT_1, HIGH);
    }
    if (waveform == 3) {
      // sawtooth
        digitalWrite(MUX0_BIT_0, HIGH);
        digitalWrite(MUX0_BIT_1, LOW);
    }
    if (waveform == 4) {
      // AM Modulation/Gilbert Mixer
        digitalWrite(MUX0_BIT_0, HIGH);
        digitalWrite(MUX0_BIT_1, HIGH);
    }
}

void advertise(){
  // Create BLE Device and Server
  BLEDevice::init(bleServerName);
  pServer = BLEDevice::createServer();
  pServer -> setCallbacks(new MyServerCallbacks());

  //Create BLE Service
  BLEService *emgService = pServer->createService(SERVICE_UUID);

  // Create BLE Characteristics and Descriptors
  emgService->addCharacteristic(&emg1Characteristics);
  emgService->addCharacteristic(&emg2Characteristics);
  emg1Characteristics.setCallbacks(new MyCallbacks());
```

```
  emg2Characteristics.setCallbacks(new MyCallbacks());

  //Start the service
  emgService->start();

  //Start advertising
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pServer->getAdvertising()->start();
}

void setup() {
  Serial.begin(115200);
  pinMode(MUX0_BIT_0, OUTPUT);
  pinMode(MUX0_BIT_1, OUTPUT);
  pinMode(VCA, OUTPUT);
  pinMode(VCO, OUTPUT);
  pinMode(Sustain, OUTPUT);
  Serial.println("Initializing BLE...");
  advertise();
  Serial.println("Waiting for a client connection to notify...");
}

void loop() {
  if (deviceConnected) {
    // Check if new values have been received.
    if (newValue1 && newValue2){
      std::string data = "Left State & YAccel: " + std::to_string(leftState) + " " +
        std::to_string(leftDOF) + ", Right State & YAccel: " + std::to_string(rightState) + " "
         + std::to_string(rightDOF) + " " + std::to_string(xAccel);
      Serial.println(data.c_str());

      // The default waveform is a sine wave.
      int waveform = 1;
      if (leftState == 1 && rightState == 1){
        waveform = 1; // sine
      } else if (leftState == 2 && rightState == 1){
        waveform = 2; // triangle
      } else if (leftState == 3 && rightState == 1){
        waveform = 3; // sawtooth
      } else if (leftState == 1 && rightState == 2){
        waveform = 4; // AM Modulation
      } else {
        // if no states are recognized then no waveform change is made
        waveform = 0;
      }

      if (waveform != 0){
        changeWaveform(waveform);
        // Serial.print("Waveform:");
        // Serial.println(waveform);
      }

      if (sound == 1){
        float vca_control = findControlV(leftDOF, 255);
        analogWrite(VCA, vca_control);
        Serial.print("VCA_control: ");
        Serial.print(vca_control);
      } else {
        analogWrite(VCA, 0);
      }
```

```
      float vco_control = findControlV(rightDOF, 255);
      analogWrite(VCO, vco_control);
      Serial.print(" VCO_control: ");
      Serial.println(vco_control);

      // Checks if sustain is selected separately
      // Currently sustain and waveform can't be changed at the same time
      if (leftState == 2 && rightState == 2){
        digitalWrite(Sustain, HIGH);
        Serial.println("Sustain On");
      } else {
        digitalWrite(Sustain, LOW);
      }

      newValue1 = false;
      newValue2 = false;
    }
    delay(10);
  }
  if (!deviceConnected) {
    Serial.println("In loopServer() function, server and client are NOT connected");
    delay(1000);
    pServer->startAdvertising(); //restart advertising
  }
}
```

5.2 ESP32 Left Arm Communication Protocol

```
// Code edited from https://randomnerdtutorials.com/esp32-ble-server-client/

#include <Arduino.h>
#include <Wire.h>
#include <MPU9250.h>
#include <BLEDevice.h>
#include <string>

// BLE Server Name
#define bleServerName "Main_ESP32"

// ESP32 Pin Definitions
MPU9250 sensor(Wire, 0x68);
#define MYOWARE_0 34
#define MYOWARE_1 35
#define BUTTON 18
#define LED 19

// Default Values
bool calibrated = false;
float scalingValue = 0;
int currentState = 1;
bool collecting = false;
unsigned long collectionStart = 0;
int decisionCounts[4] = {0};
int previousEMG1 = 0;
int previousEMG2 = 0;
static boolean doConnect = false;
static boolean connected = false;
static boolean doScan = true;

//Client Version of Characteristics and Descriptors
```

```
static BLEUUID emgServiceUUID("3b7ce6b7-3190-4d39-b5dd-01df08fef25f");
static BLEUUID emg1CharacteristicsUUID("0a6686f6-1e4b-485e-a252-3c353e0c875a");

//Address of the peripheral device "the server," found while scanning.
static BLEAddress *pServerAddress;
static BLERemoteCharacteristic* emg1Characteristic;
static BLEAdvertisedDevice* myDevice;

std::vector<int> analogValues0;

float calculateScalarInt(float max_observed_value) {
    if(max_observed_value == 0.00){
        Serial.println("Sensor Calibration Error");
        return 0.00;
    }
  return 8000/max_observed_value;
}

int calibrate() {
    delay(1000);
    Serial.println("Flex Hand to calibrate device");

    while (!calibrated){
      int on = digitalRead(BUTTON);
      if (on == 1) {
        int analogValue0 = analogRead(MYOWARE_0);
        analogValues0.push_back(analogValue0);
      }
      else {
        calibrated = true;
        Serial.println("device done calibrating");
      }
    }

    int max0 = 0;
    for (int i = 0; i < analogValues0.size(); i++) {
      if (analogValues0[i] > max0) max0 = analogValues0[i];
    }

    float scalar = calculateScalarInt(max0) * 100;
    Serial.print("Scaling Value:");
    Serial.println(scalar);
    return scalar;
}

// State identifying function
int predict2(int EMG1, int EMG2) {
    if (EMG2 <= 1) {
        if (EMG1 <= 1) {
            return 0;
        } else {
            if (EMG1 <= 579) {
                if (EMG1 <= 219) {
                    if (EMG1 <= 169) {
                        return 4;
                    } else {
                        return 4;
                    }
                } else {
                    if (EMG1 <= 226) {
                        return 4;
```

```
                } else {
                    return 4;
                }
            }
        } else {
            if (EMG1 <= 1291) {
                if (EMG1 <= 1279) {
                    return 4;
                } else {
                    return 2;
                }
            } else {
                if (EMG1 <= 2600) {
                    return 4;
                } else {
                    return 4;
                }
            }
        }
    }
} else {
    if (EMG2 <= 1653) {
        if (EMG1 <= 1) {
            if (EMG2 <= 814) {
                if (EMG2 <= 290) {
                    return 0;
                } else {
                    return 0;
                }
            } else {
                if (EMG2 <= 1300) {
                    return 0;
                } else {
                    return 1;
                }
            }
        } else {
            if (EMG1 <= 1037) {
                if (EMG2 <= 513) {
                    return 2;
                } else {
                    return 2;
                }
            } else {
                if (EMG1 <= 2299) {
                    return 4;
                } else {
                    return 4;
                }
            }
        }
    } else {
        if (EMG1 <= 1) {
            if (EMG2 <= 2896) {
                if (EMG2 <= 2254) {
                    return 1;
                } else {
                    return 1;
                }
            } else {
                if (EMG2 <= 4653) {
```

```
                    return 1;
                } else {
                    return 1;
                }
            }
        } else {
            if (EMG2 <= 6619) {
                if (EMG2 <= 2144) {
                    return 3;
                } else {
                    return 3;
                }
            } else {
                if (EMG2 <= 9414) {
                    return 1;
                } else {
                    return 1;
                }
            }
        }
    }
}

// Connect to BLE Server with specificed name, Service, and Characteristics
bool connectToServer(BLEAddress pAddress) {
    BLEClient* pClient = BLEDevice::createClient();
    pClient->connect(pAddress);
    Serial.println("Connected to server");

    // Obtain reference to the service
    BLERemoteService* pRemoteService = pClient->getService(emgServiceUUID);
    if (pRemoteService == nullptr) {
        Serial.print("Failed to find service UUID: ");
        Serial.println(emgServiceUUID.toString().c_str());
        return false;
    }
    Serial.println("Found service");

    //Obtain reference to the characteristics
    emg1Characteristic = pRemoteService->getCharacteristic(emg1CharacteristicsUUID);
    if(emg1Characteristic == nullptr){
        Serial.print("Failed to find characteristic UUID");
        return false;
    }
    Serial.println("Found characteristics");

    connected = true;
    return true;
}

// scan for BLE servers
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice){
        Serial.print("BLE Advertised Device found: ");
        Serial.println(advertisedDevice.toString().c_str());
        // If we found the right device, stop the scan and remember the address
        if (advertisedDevice.getName() == bleServerName){
            advertisedDevice.getScan()->stop();
            pServerAddress = new BLEAddress(advertisedDevice.getAddress());
            doConnect = true;
```

```
            doScan = true;
            Serial.println("Device found. Connecting!");
        }
    }
};

void connectToBLE(){
    Serial.println("Starting BLE Client application...");
    BLEDevice::init("");

    //Retrieve a Scanner and set callback to inform us of new detected Device.
    //Specify active scanning and start scan for 30 seconds
    BLEScan* pBLEScan = BLEDevice::getScan();
    pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
    pBLEScan->setActiveScan(true);
    pBLEScan->start(30);
    Serial.println("Waiting for Bluetooth connection...");
}

void setup() {
    Serial.begin(115200);
    Wire.begin();
    pinMode(LED, OUTPUT);
    pinMode(BUTTON, INPUT);

    while(!calibrated){
        int on = digitalRead(BUTTON);
        if (on == 1) {
            Serial.println("Device not calibrated");
            delay(1000);
            continue;
        } else {
            Serial.println("Button pressed to begin calibration process");
            digitalWrite(LED, HIGH);
            scalingValue = calibrate();
            digitalWrite(LED, LOW);
            Serial.println("Calibration Complete");
            delay(1000);
        }
        delay(100);
    }

    if (!sensor.begin()) {
        Serial.println("Sensor not found! Check connections.");
        while (1);
    }

    connectToBLE();
}

void loop() {
    // If the flag "doConnect" is true then we have scanned for and found the desired
    // BLE Server with which we wish to connect. Set the connected flag to be true.
    if (doConnect == true) {
        if (connectToServer(*pServerAddress)){
            Serial.println("We are now connected to the BLE Server.");
            connected = true;
        } else {
            Serial.println("Failed to connect to the server; Restart your device to scan for
            nearby BLE server again.");
        }
```

```
    doConnect = false;
} else {
    Serial.println("Bluetooth not connected. Waiting for connection...");
}

// If connected to a peer BLE Server, update the characteristic
    each time we receive new sensor values.
if (connected){
    sensor.readSensor();
    String accelYData = String(sensor.getAccelX_mss(), 2);

    float analogValue0 = analogRead(MYOWARE_0)*scalingValue;  // Read analog signal
    float analogValue1 = analogRead(MYOWARE_1)*scalingValue;

    // Check for 0 to non-zero transition
    bool trigger = (previousEMG1 == 0 && analogValue0 > 0) || (previousEMG2 == 0 && analogValue1 > 0);
    previousEMG1 = analogValue0;
    previousEMG2 = analogValue1;

    // Start collection window on trigger
    if (trigger && !collecting) {
        collecting = true;
        collectionStart = millis();
        memset(decisionCounts, 0, sizeof(decisionCounts));
    }

    // Collect decisions during active window
    if (collecting) {
        int decision = predict2(analogValue0, analogValue1);
        decisionCounts[decision]++;

        Serial.print("Decision: ");
        Serial.println(decision);

        // Check if window has expired
        if (millis() - collectionStart >= 250) {
            collecting = false;

            // Find majority decision
            int maxCount = -1;
            int newState = currentState;
            for (int i = 1; i < 4; i++) { // Skip state 0 in voting
                if (decisionCounts[i] > maxCount) {
                    maxCount = decisionCounts[i];
                    newState = i;
                } else if (decisionCounts[i] == maxCount) {
                    newState = max(newState, i); // Tiebreaker: higher state number
                }
            }

            // Update state if majority is not 0
            if (decisionCounts[0] < maxCount) {
                currentState = newState;
            }
        }
    }

    std::string state = std::to_string(currentState);
    String st = state.c_str();

    String state_9dof = st + ", " + accelYData;
```

```
        Serial.println(state_9dof);

        // Send the voltage string over Bluetooth
        emg1Characteristic->writeValue(state_9dof.c_str(), false);
        Serial.println("Data Sent!");
    } else if(doScan){
        BLEDevice::getScan()->start(0); //start scan at disconnect
    }
    delay(10);
}
```

ESP32 Right Arm Communication Protocol

The same as the left arm except for the following lines.

```
// Code edited from https://randomnerdtutorials.com/esp32-ble-server-client/

//Sensor Definition
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
Adafruit_MPU6050 sensor;

//Client Version of Characteristics and Descriptors
static BLEUUID emg2CharacteristicsUUID("a7a50cff-be29-48c6-81eb-406b39fe602b");
static BLERemoteCharacteristic* emg2Characteristic;

// State identifying function
int predict2 (int EMG1){
    if (EMG1 != 0){
        return 2;
    }
    return 1;
}

// Connect to BLE Server with specificed name, Service, and Characteristics.
//Obtain reference to the characteristics
    emg2Characteristic = pRemoteService->getCharacteristic(emg2CharacteristicsUUID);
    if(emg2Characteristic == nullptr){
        ...
    }

void setup() {
    sensor.setAccelerometerRange(MPU6050_RANGE_8_G);
    ...
}

void loop() {
    // If we are connected to a peer BLE Server, update the characteristic
        each time we are receive new sensor values.
    if (connected){
        sensors_event_t a, g, temp;
        sensor.getEvent(&a, &g, &temp);
        float accelYData = a.acceleration.y;
        float accelXData = a.acceleration.x;

        // Collect decisions during active window
        if (collecting) {
            int decision = predict2(analogValue0);
            ...
        }
        ...
    }
```

```
        String state_6dof = st + ", " +  accelYData + " " + accelXData;
        Serial.println(state_6dof);

        // Send the voltage string over Bluetooth
        emg2Characteristic->writeValue(state_6dof.c_str(), false);
    }
    ...
}
```

6 Analog Circuit

This section gives a detailed overview of the analog components and schematics. Below is the overall analog block diagram, and each subsection will detail the individual blocks. To tinker with any of the schematics, please see the Cadence_files folder in github.

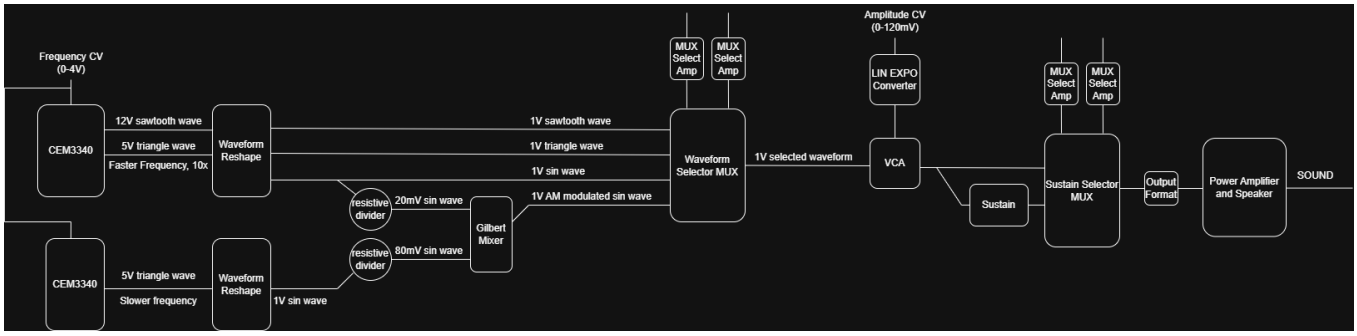


Figure 2: Analog Block Diagram

Photos of the physical board are also below.

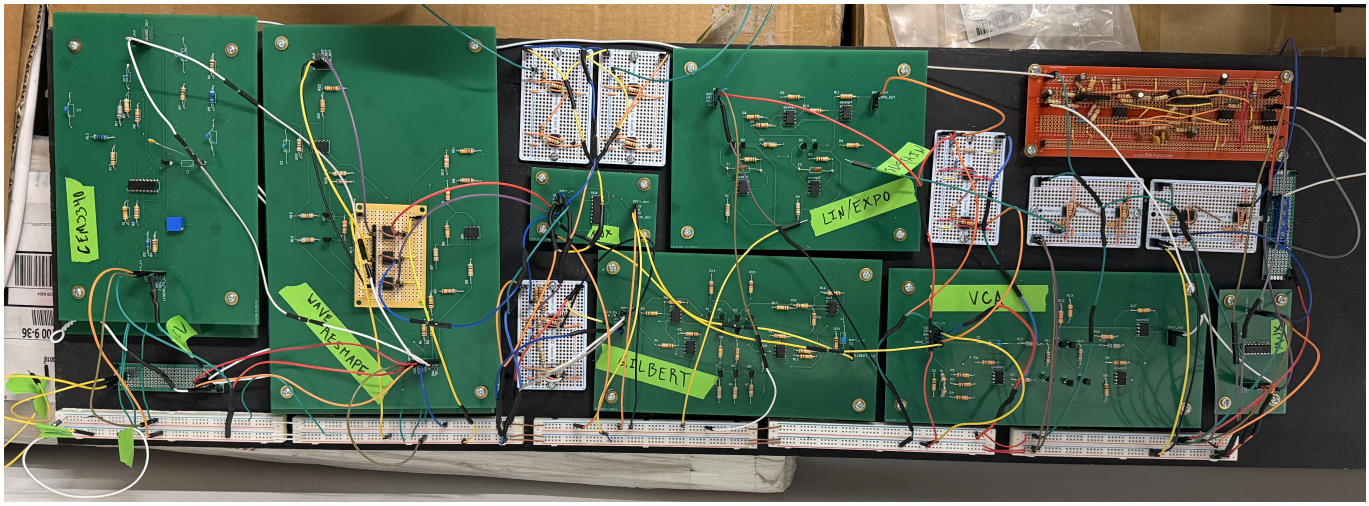


Figure 3: Photo of the physical circuit, note that the CEM3340 and WAVE_RESHAPE boards each have two copies, they are stacked on top of each other to make efficient use of the wooden base board.

6.1 CEM3340 (VCO)

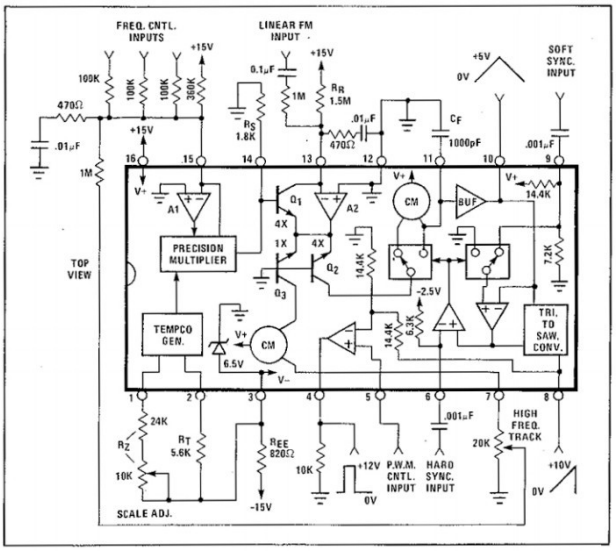


Figure 4: CEM3340 Schematic

A note on the CEM3340:

The CEM3340 is a unique chip that allows multiple waveforms, each running at the same frequency, to be changed at the same rate with the same control voltage. Early in this capstone project, it became clear that it would be near impossible to design a working VCO with multiple waveform outputs by scratch in the given timeframe, thus the CEM3340 was sourced.

www.birthofasynth.com commenting on the CEM3340, "Designed by the genius that was Doug Curtis, the CEM3340 was a rock solid, thermally stable Voltage Controlled Oscillator. If a designer could get his/her hands on a CEM3340, gone was the need to laboriously match transistors or search vainly through the ever dwindling selection of matched transistor pairs available to mere mortals. Gone was the need to find elusive tempco resistors or deal with the heartbreak and mess of thermal compound. Gone was the worry that the damn thing would drift with temperature. All of this was possible with a minumum of parts; so few parts, in fact, that the CEM3340 helped make possible the polyphonic analog synthesizer. Mr. Curtis' design made all of these things a reality."

As shown in the schematic above, the output of the CEM3340 is a 5V triangle wave, 10V sawtooth wave, and a 12V square wave. Due to the rarity of the chip and that it is a true feat of engineering, simulation specs for this chip have never, and likely will never, be published. The schematic above is the correct peripheral circuity to make the chip function, and in simulation, it perfectly suffices to use voltage sources simulating the chip's waveform outputs.

To showcase the functionality of the Gilbert Mixer, this circuit mixed two different frequencies generated from two different CEM3340 chips. The different frequencies were created by using different capacitor values for the capacitor straddling pins 12 and 13. The difference in frequencies is by a factor of 10. If, for example, one CEM3340 is set to 10Hz, the second is set to 100Hz. The CV also adjusts the two frequencies at the same rate, maintaining the factor of 10 difference regardless of the frequency range.

6.2 WAVE_RESHAPE

Converts the output 5V tri and 12V saw waves of the CEM3340 to 1V tri, saw, and sin waves.

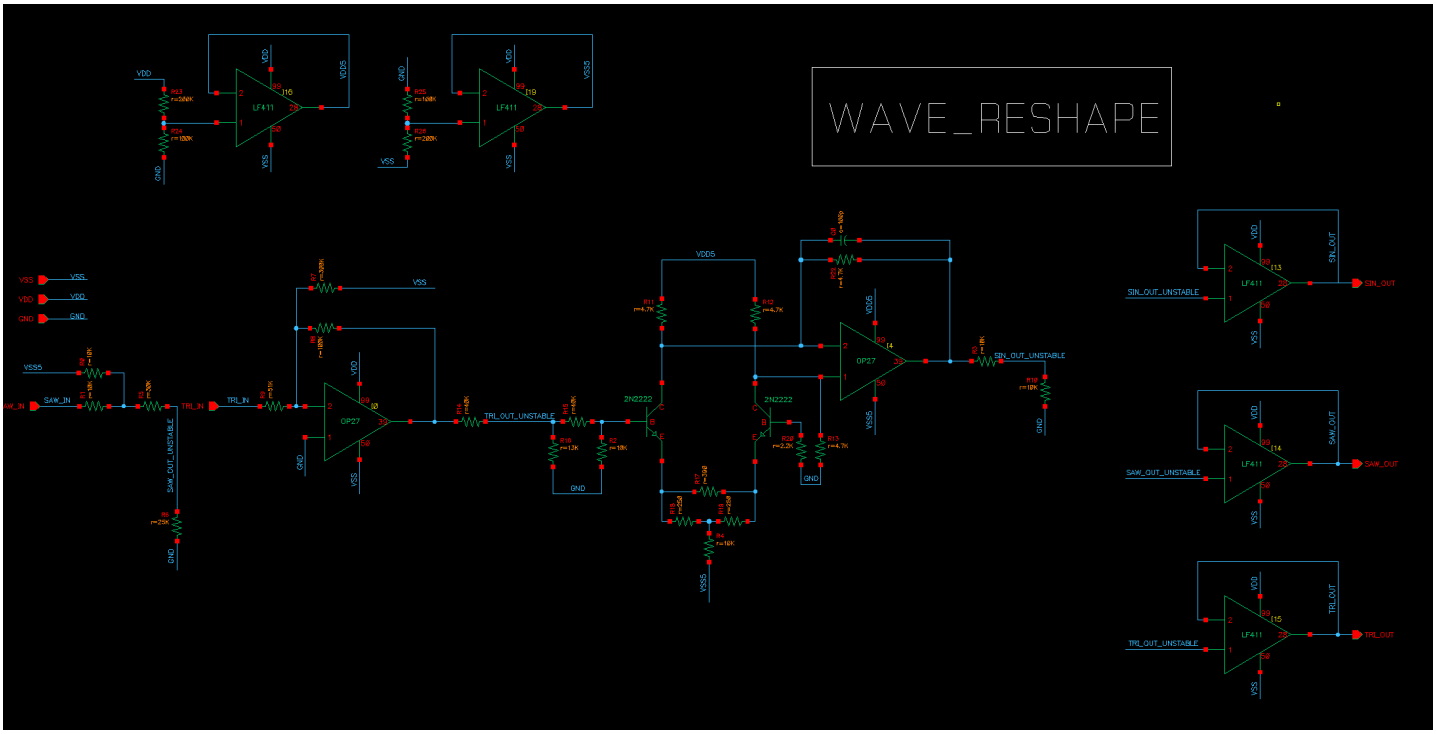


Figure 5: WAVE_RESHAPE Schematic

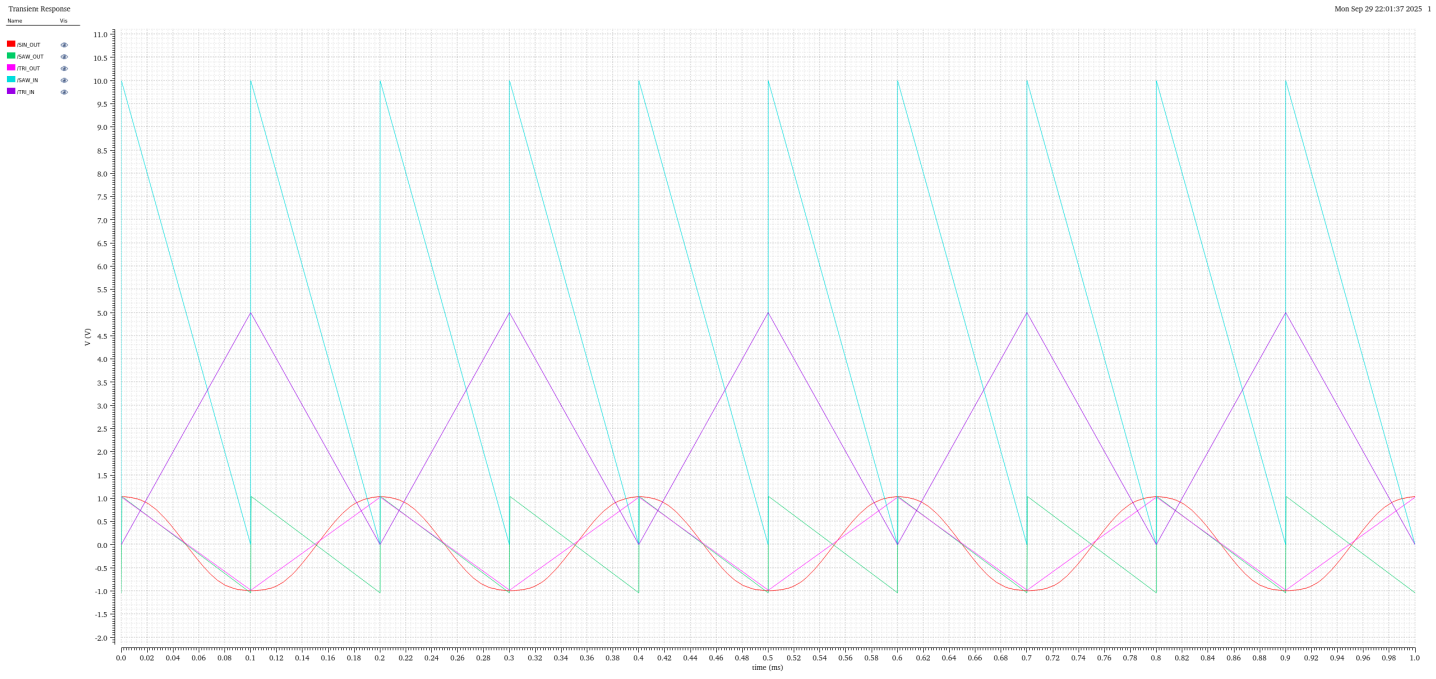


Figure 6: WAVE_RESHAPE Waveform

6.3 20mV/80mV_DIVIDER

Steps the output 1V sin waves of WAVEFORM_RESHAPE to 20mV and 80mV sin waves to prepare them for the GILBERT_MIXER

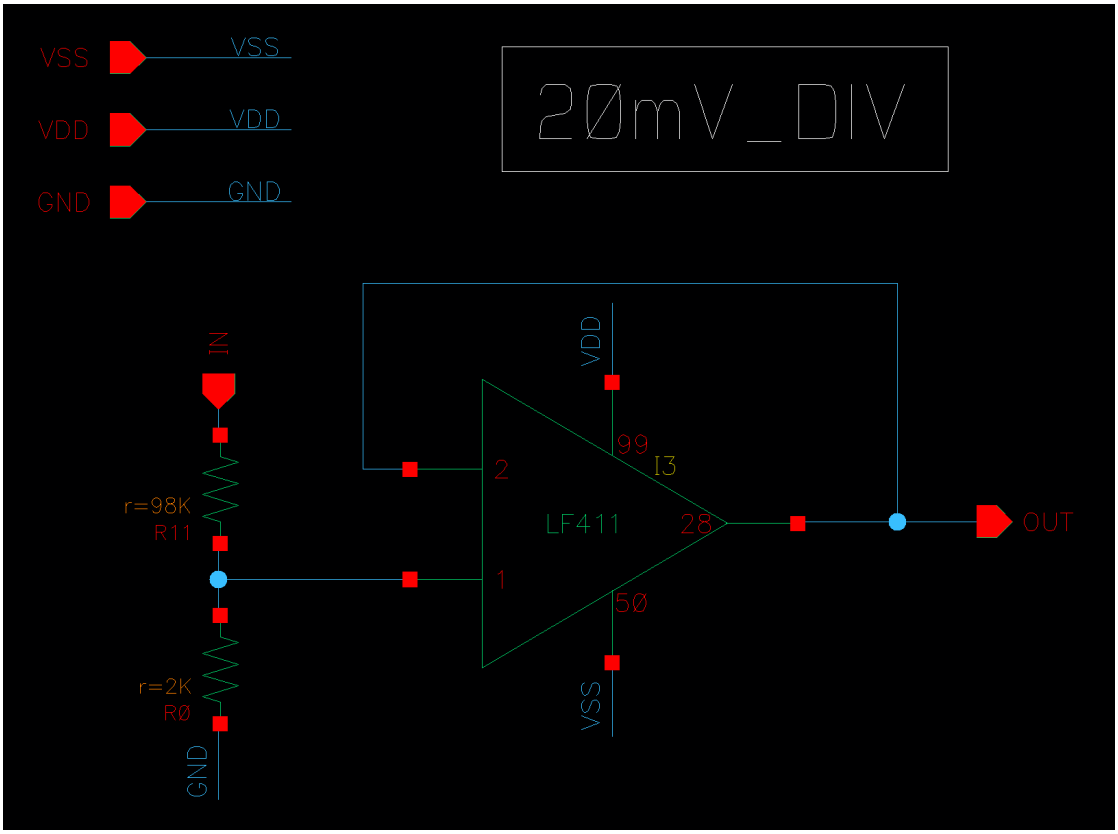


Figure 7: 20mV_DIVIDER Schematic

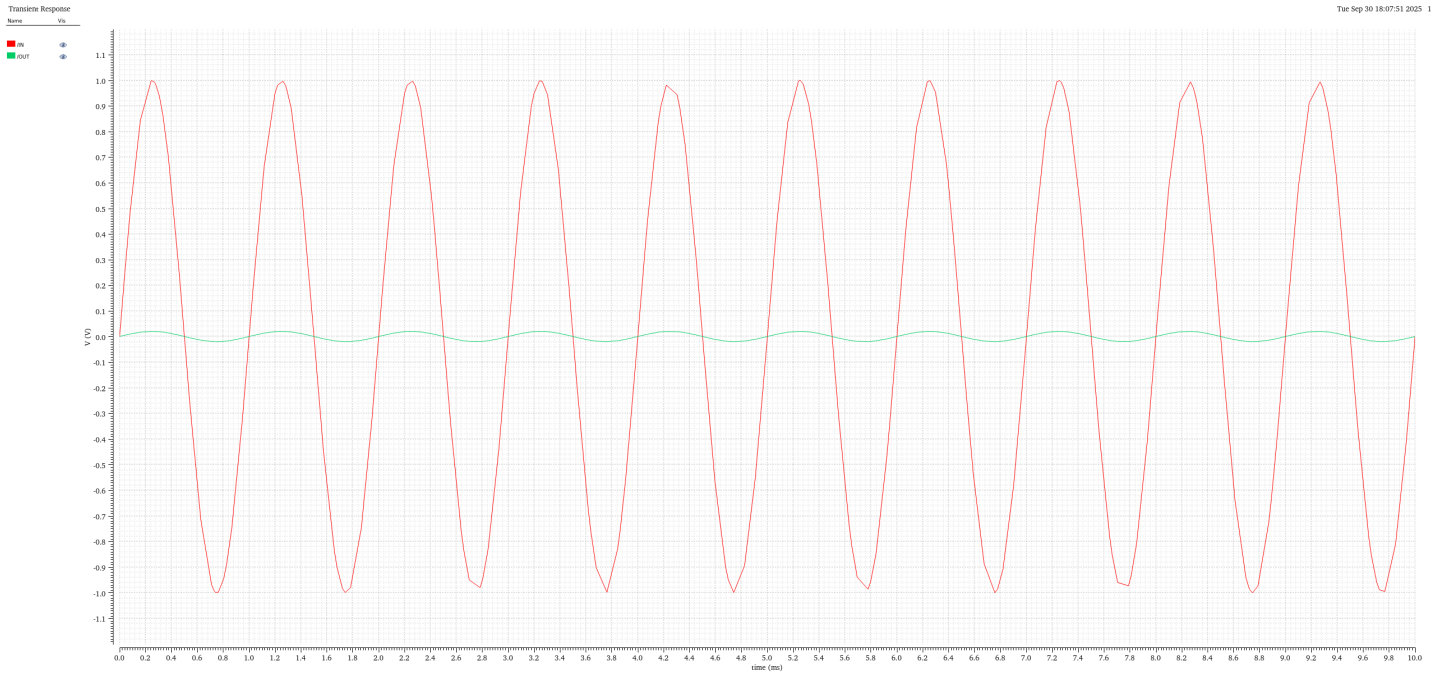


Figure 8: 20mV_DIVIDER Waveform

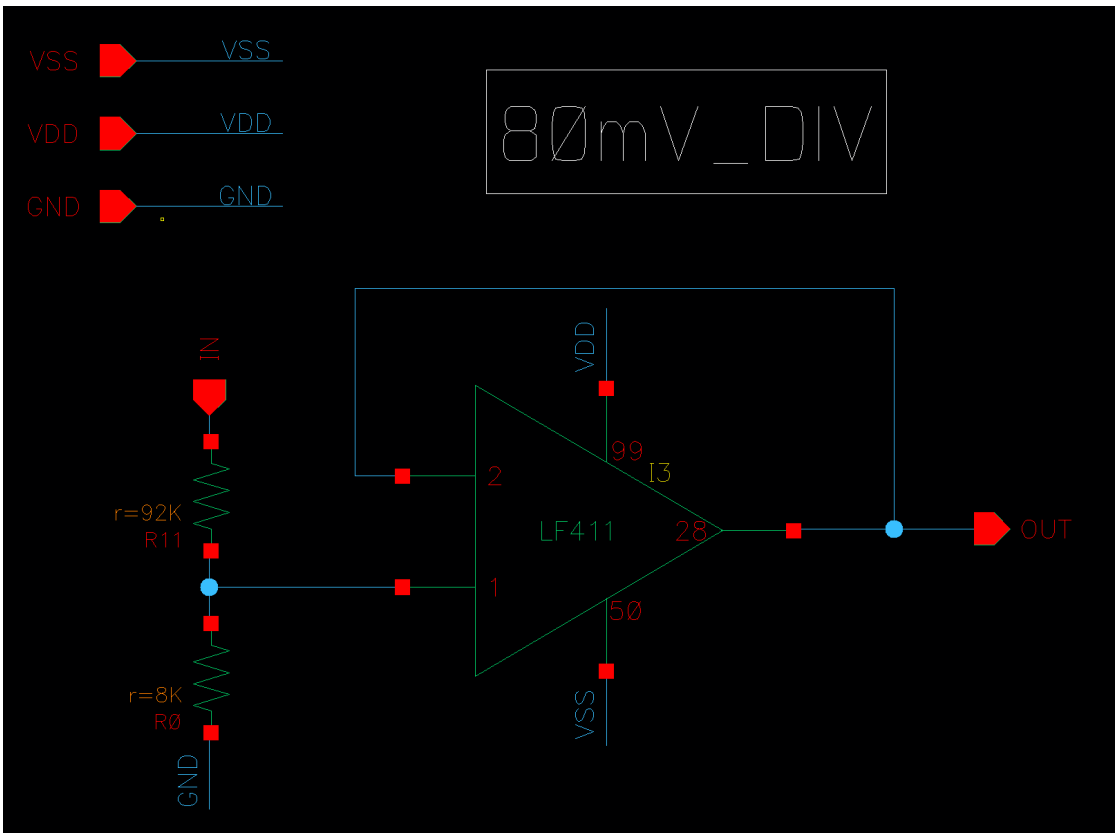


Figure 9: 80mV_DIVIDER Schematic

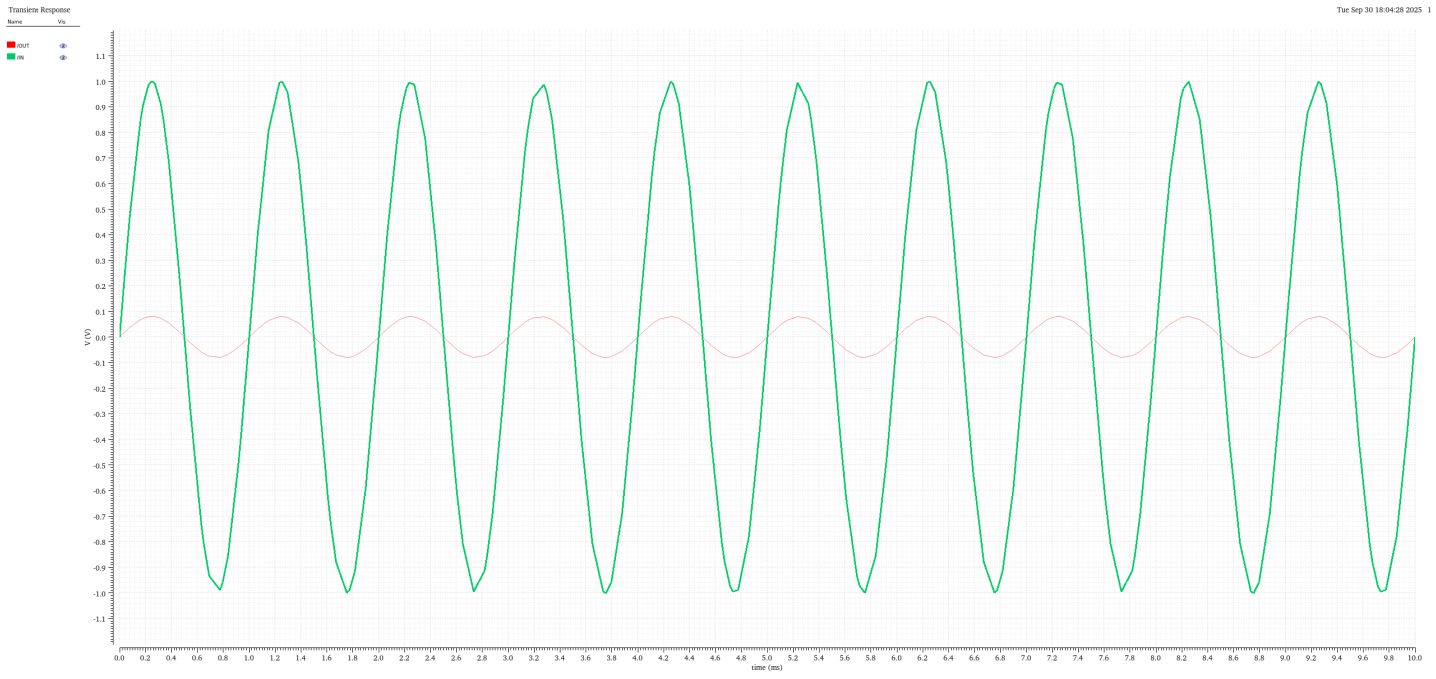


Figure 10: 80mV_DIVIDER Waveform

6.4 GILBERT_MIXER

An analog multiplier circuit that combines two input signals, typically an RF signal and a local oscillator for the AM radio application, to produce sum and difference frequencies for frequency conversion.

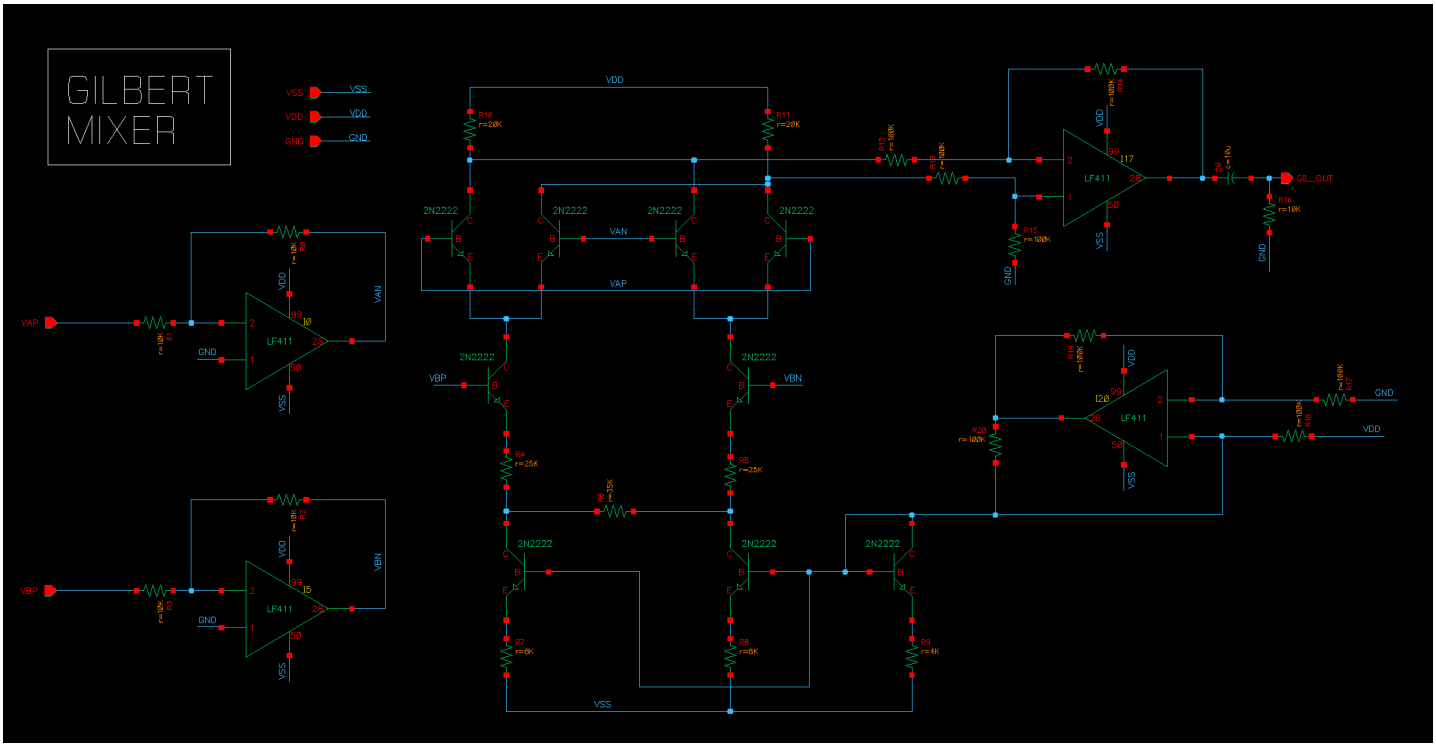


Figure 11: GILBERT MIXER Schematic

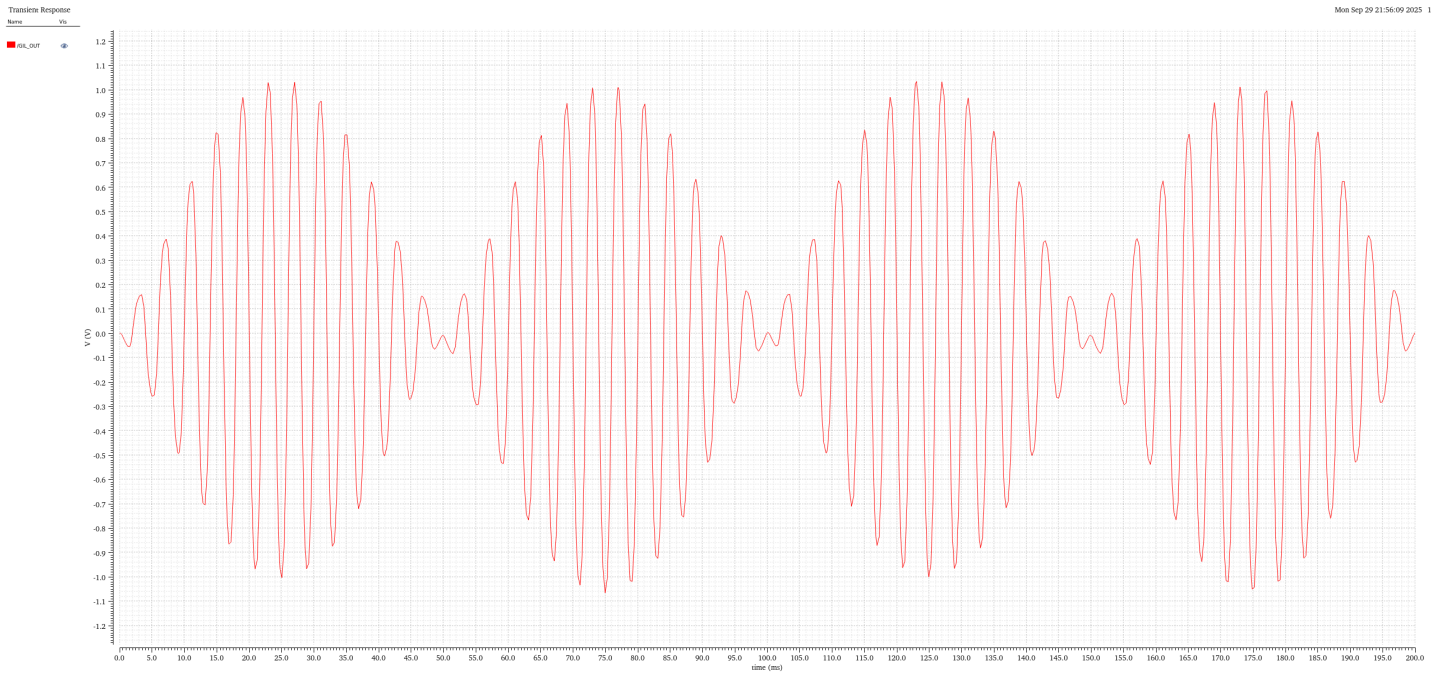


Figure 12: GILBERT MIXER Waveform

6.5 ADG5082 (analog MUX)

Selects between analog waveform inputs depending on 3 bit input selection

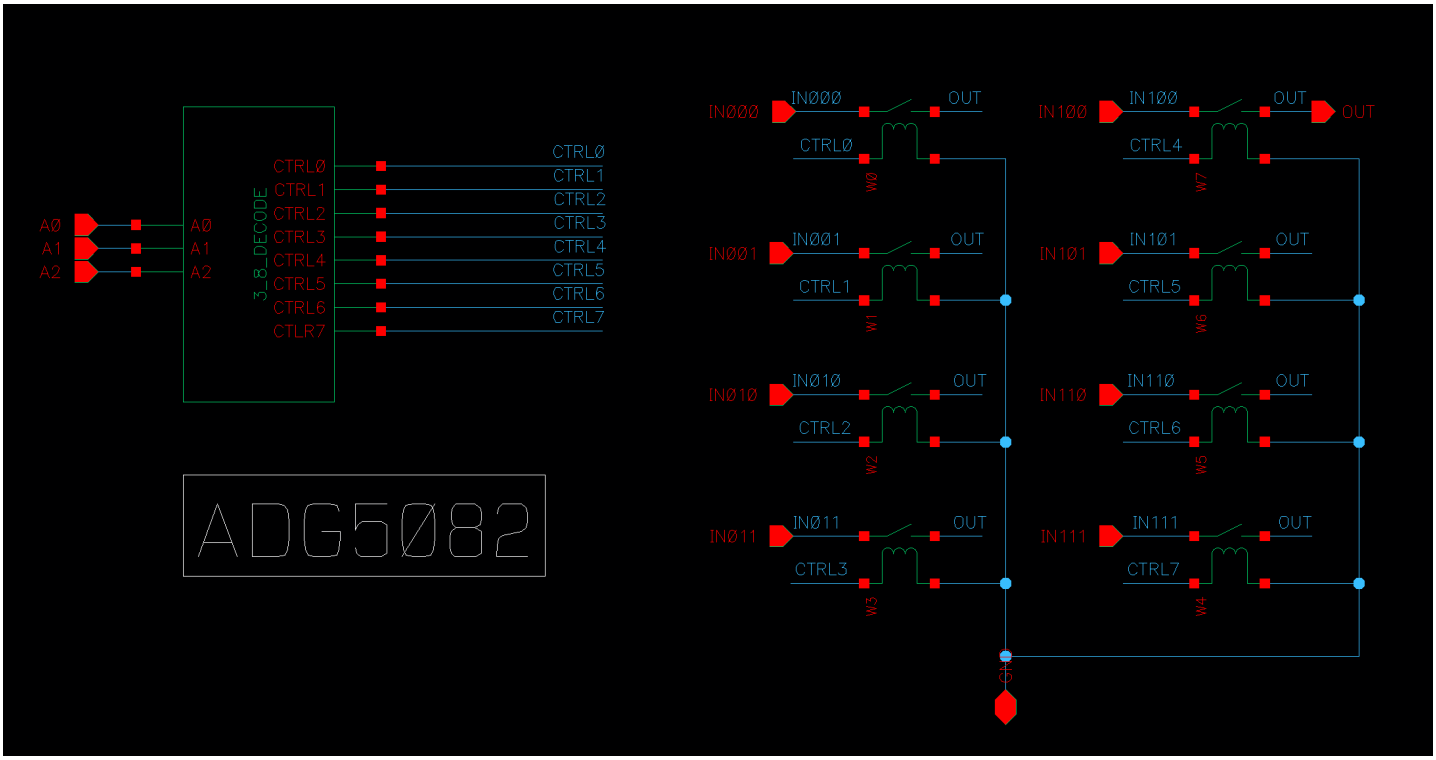


Figure 13: ADG5082 Schematic

Figure 14: ADG5082 Waveform

Detailed schematic of the 3-to-8 one hot decoder. The reader can see that when all three selectors are low, output bit CTRL0 correctly outputs logic HIGH.

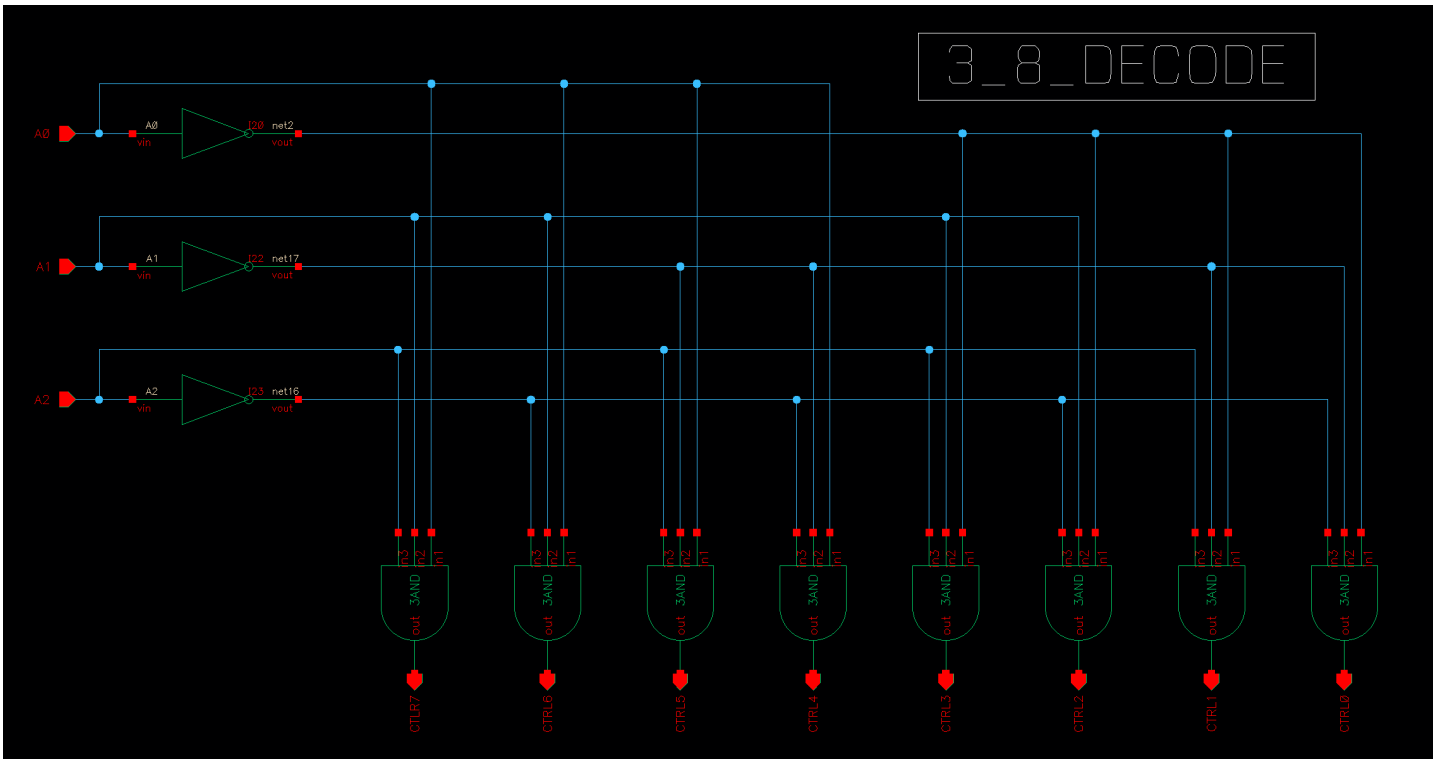


Figure 15: 3-to-8 DECODER Schematic

Figure 16: 3-to-8 DECODER Waveform

6.6 MUX_SELECTION_AMPLIFIER

Due to the peripheral circuitry and additional software for this project, the voltages generated to drive the MUX selection are much less than the required 15V. Therefore, an amplifier is necessary at the input of each actively used MUX selection bit. It is also important that the gain not be so large that if the low selector value is riding slightly above 0V, it will falsely bar to high.

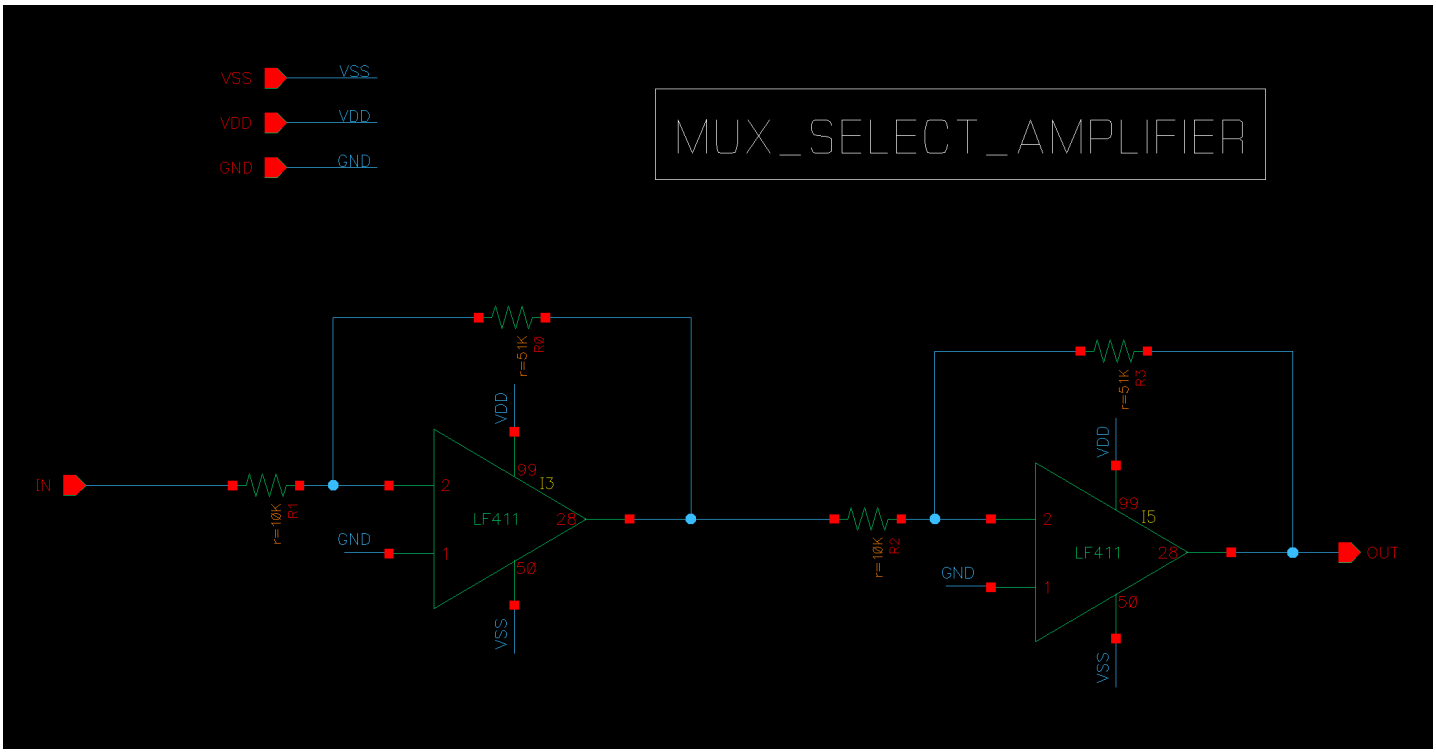


Figure 17: MUX_SELECTION_AMPLIFIER Schematic

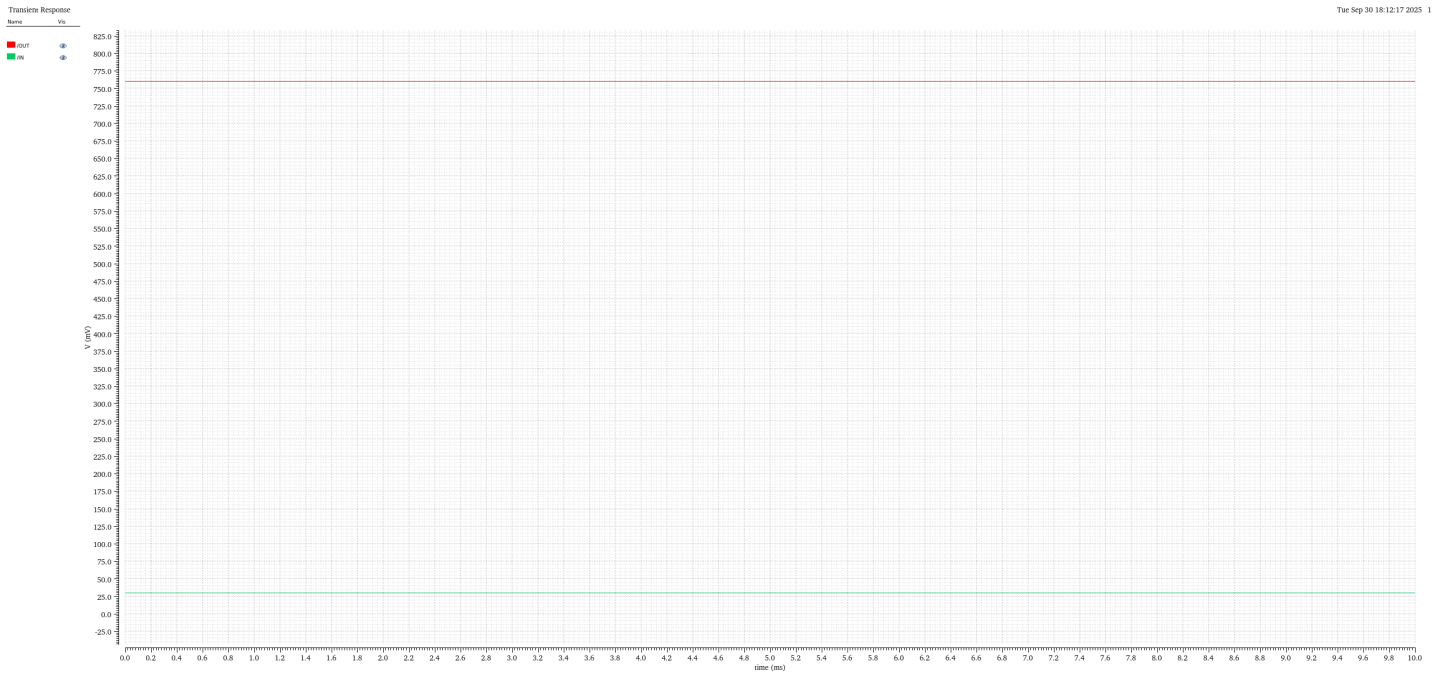


Figure 18: MUX_SELECTION_AMPLIFIER Waveform

6.7 LIN_EXPO_CONVERTER

A linear-to-exponential (lin-to-expo) converter is needed in an analog music synthesizer because humans perceive pitch and loudness logarithmically, so converting a linearly controlled voltage to an exponential response allows intuitive control over frequencies and amplitudes in a musically natural way. In this project, only the VCA CV is processed through the LIN_EXPO_CONVERTER because the CEM3340 already does so for pitch. Due to the unique aspects of this project, the CV for the VCA must not be greater than 4.5V. The LIN_EXPO_CONVERTER first converts the linear 110mV wave into a roughly 15V exponential curve (node "OUT_PRE_DIVIDER") before using a resistive divider to step the exponential curve down to 4.5V.

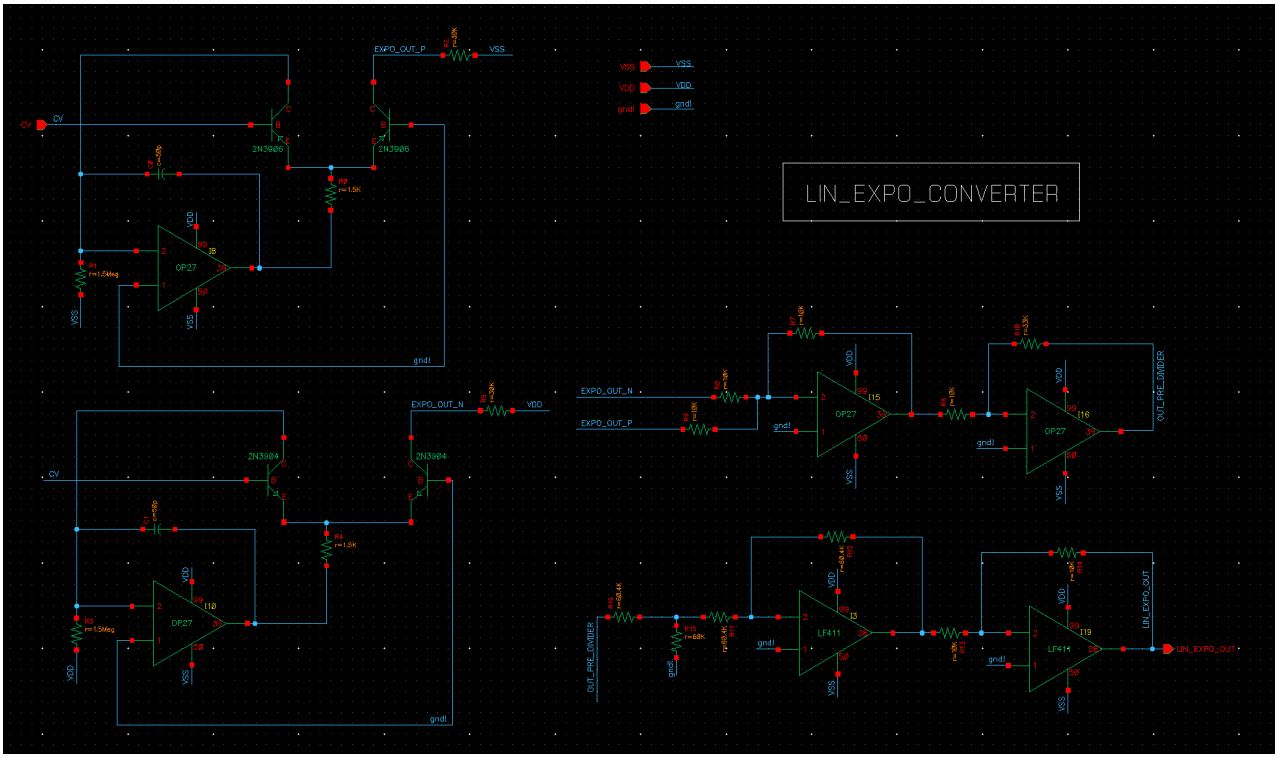


Figure 19: LIN_EXPO_CONVERTER Schematic

Figure 20: LIN_EXPO_CONVERTER Waveform

6.8 VCA

A Voltage-Controlled Amplifier (VCA) is an analog circuit that adjusts the amplitude of an input signal based on a control voltage.

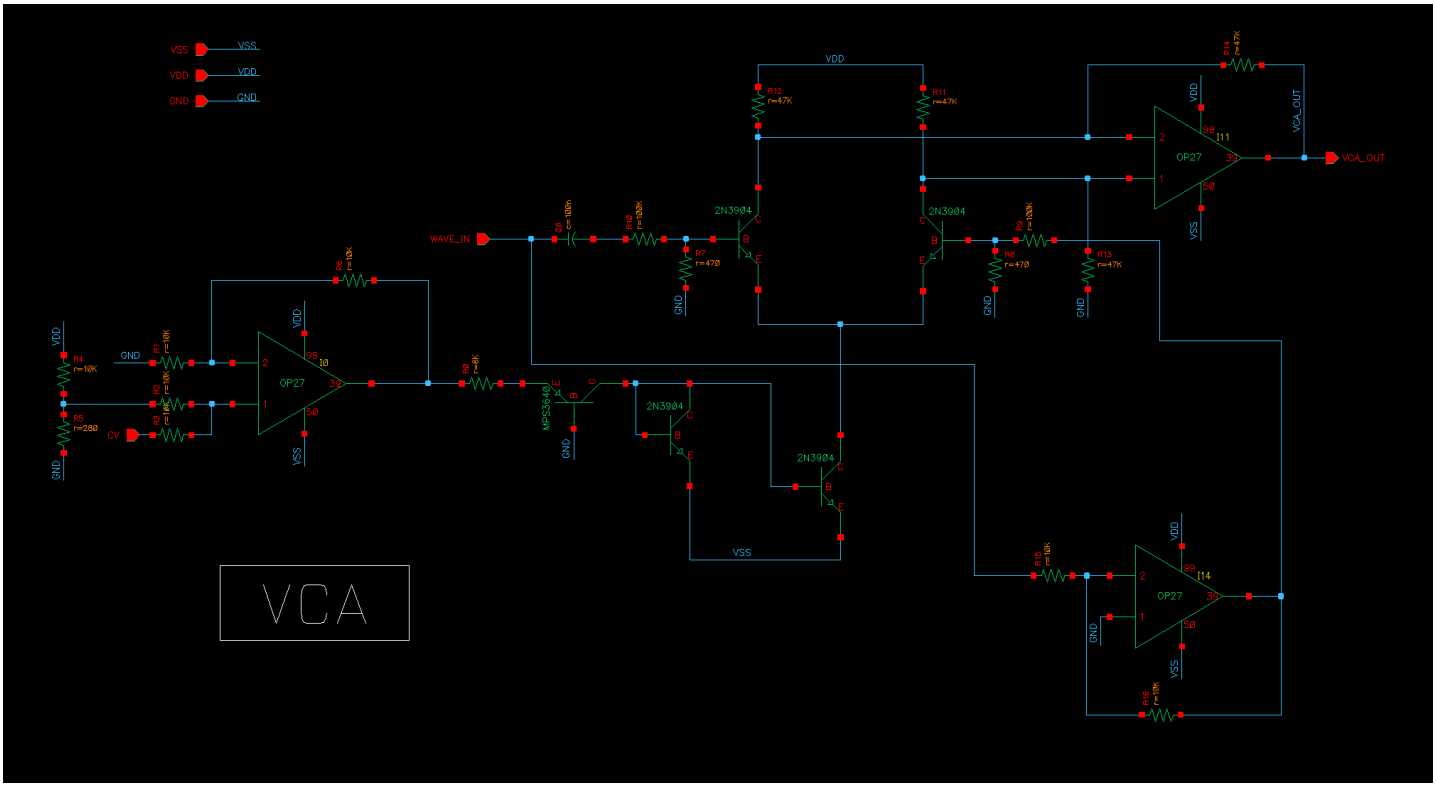


Figure 21: VCA Schematic

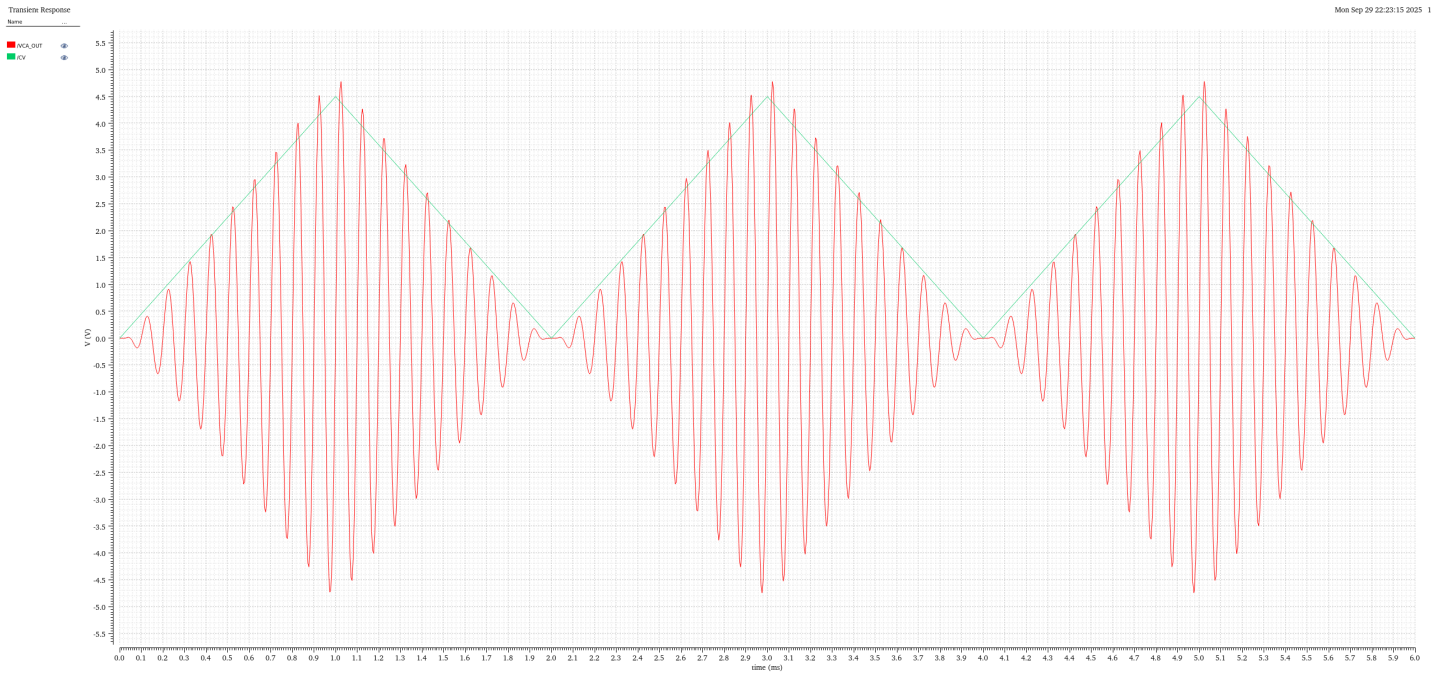


Figure 22: VCA Waveform

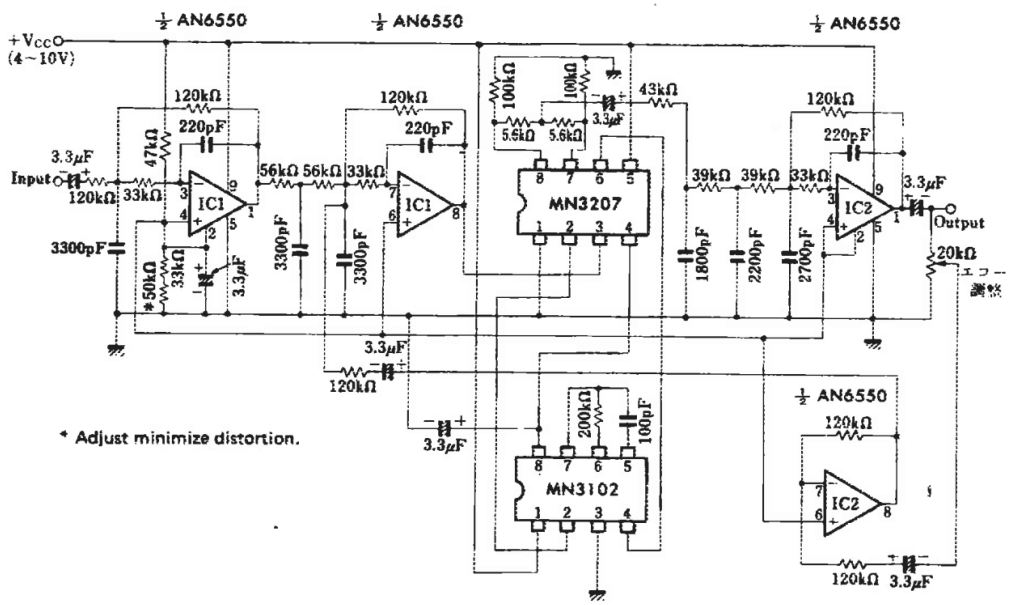
6.9 SUSTAIN (BBD)

The successful implementation of the sustain circuit is, without a doubt, the most unique aspect of this project. It requires use of a bucket brigade device (BBD).

"A bucket brigade or bucket-brigade device (BBD) is a discrete-time analogue delay line,[1] developed in 1969 by F. Sangster and K. Teer of the Philips Research Labs in the Netherlands. It consists of a series of capacitance sections C0 to Cn. The stored analogue signal is moved along the line of capacitors, one step at each clock cycle. The name comes from analogy with the term bucket brigade, used for a line of people passing buckets of water. In most signal processing applications, bucket brigades have been replaced by devices that use digital signal processing, manipulating samples in digital form. Bucket brigades still see use in specialty applications, such as guitar effects." [1]

BBDs were widely used in the 1970s and 1980s for chorus, flanger, and echo effects before digital delay chips became standard, making them a classic and somewhat rare component today. Their fully analog nature and distinctive, slightly degraded sound give them a unique character that is difficult to replicate digitally.

The exact circuit pictured in the schematic below is used in this project, although it could not be simulated prior to building the physical circuit as simulation files for BBDs are understandably not available. It is important to note, purchasing the AN6550 PLL chips is vital to the successful output of this circuit. This capstone group first only ordered the MN3*0* BBD chips, used available PLLs from the Columbia electronics stock, and determined the output unsuccessful. Upon specifically ordering the AN6550 PLL chips, however, the circuit worked.



Echo Effect Generation Circuit

Figure 23: SUSTAIN Schematic

The waveform pictured below is an actual photo of the oscilloscope probing the output of the SUSTAIN circuit

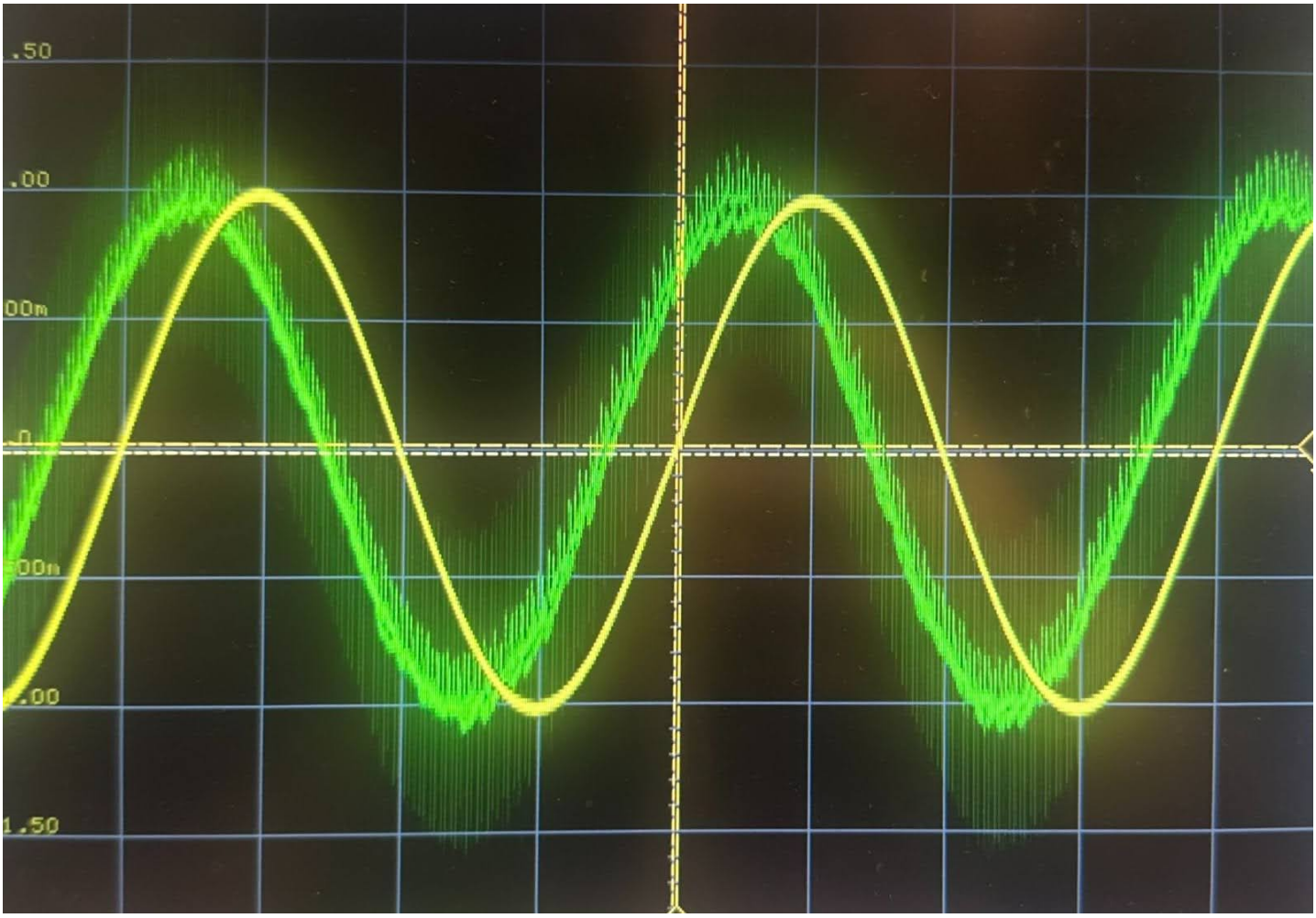


Figure 24: SUSTAIN Waveform

Please note, at the time of filming the "waveform_demo" video, the sustain circuit stop working as when shown in the picture. Even though a "vibrato" was added to the waveform, it is much more noisy and inaccurate than it was upon first achieving a working circuit.

6.10 OUTPUT_PROCESSING

The output processing circuit removes any offset with an RC circuit that has a corner frequency of 1Hz, and triples the amplitude of the output wave. The amplitude step-up is necessary because of design constrains further upstream limiting the max output frequency to 5V prior to the OUTPUT_PROCESSING circuit.

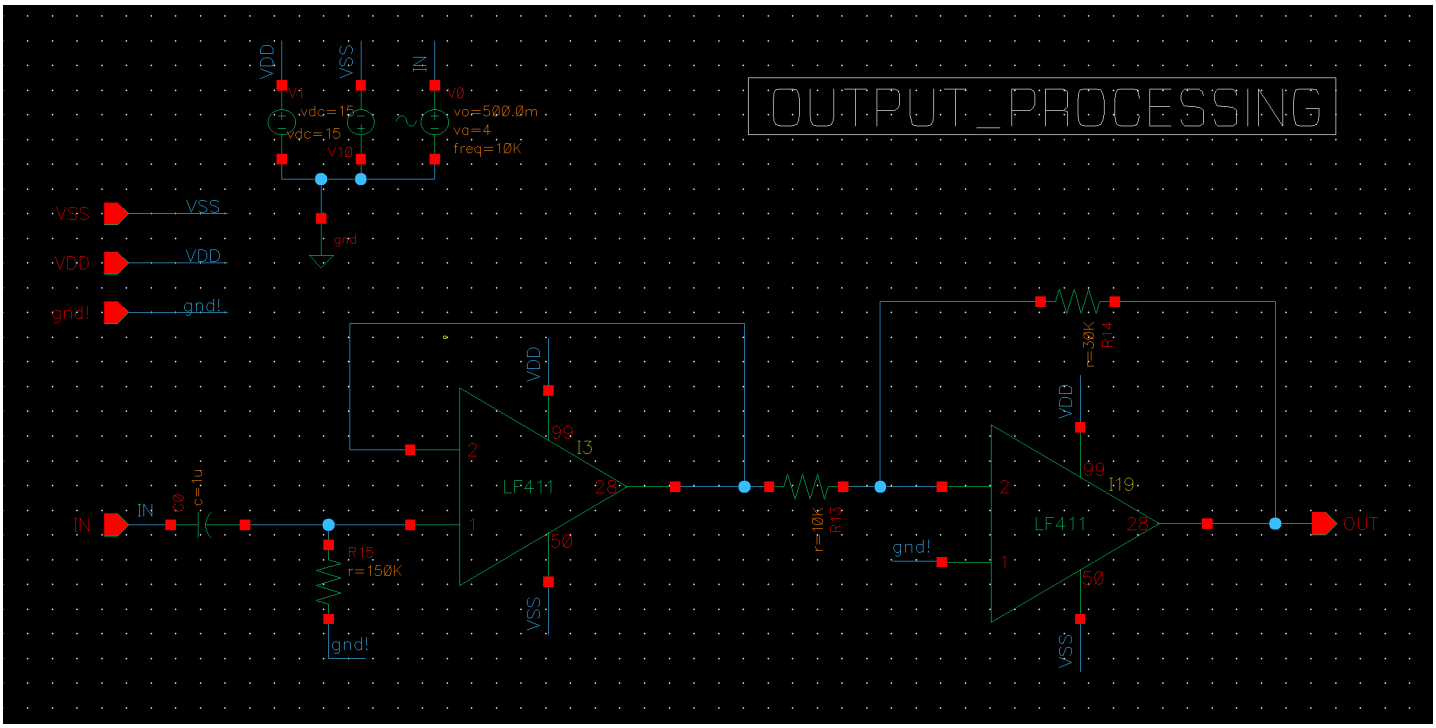


Figure 25: OUTPUT PROCESSING Schematic

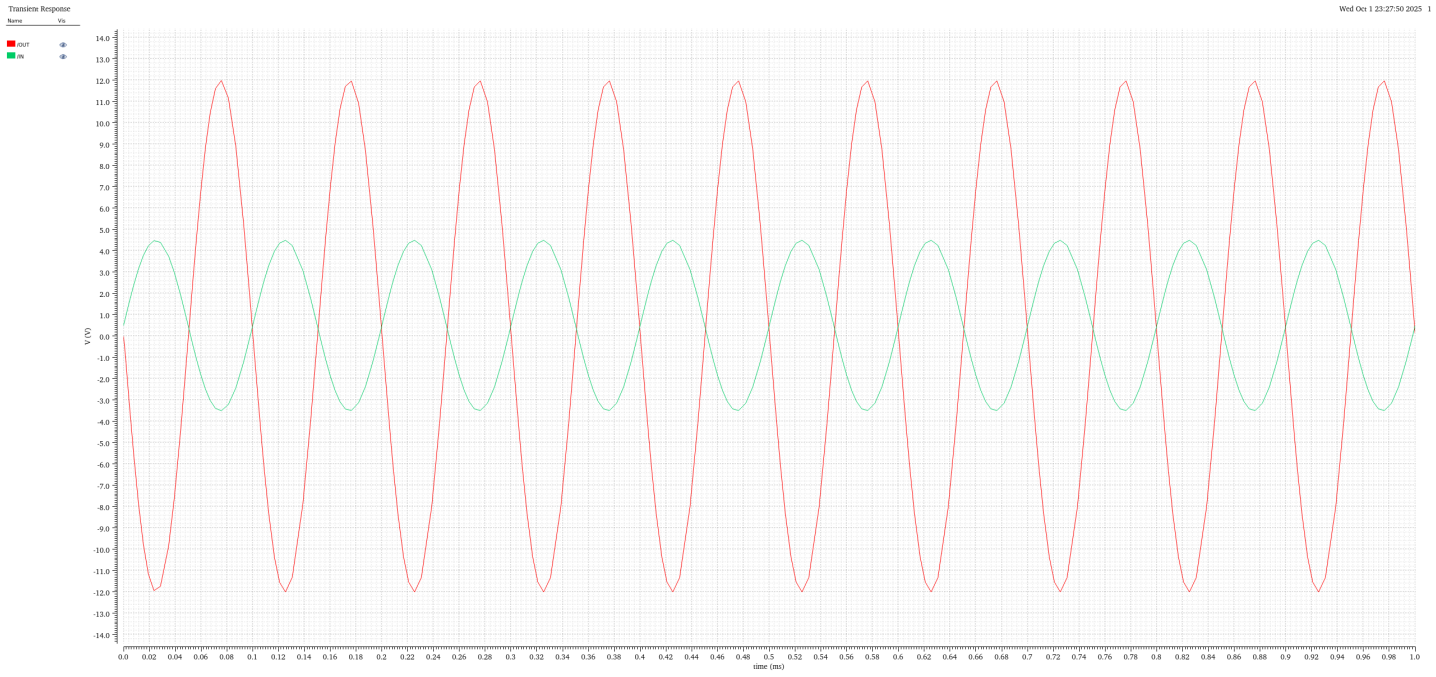


Figure 26: OUTPUT PROCESSING Waveform

6.11 COMPLETE INTEGRATION

Below, there is a screenshot of the analog circuit completely integrated in Cadence, not including the CEM3340 and SUSTAIN circuits, as they cannot be simulated.

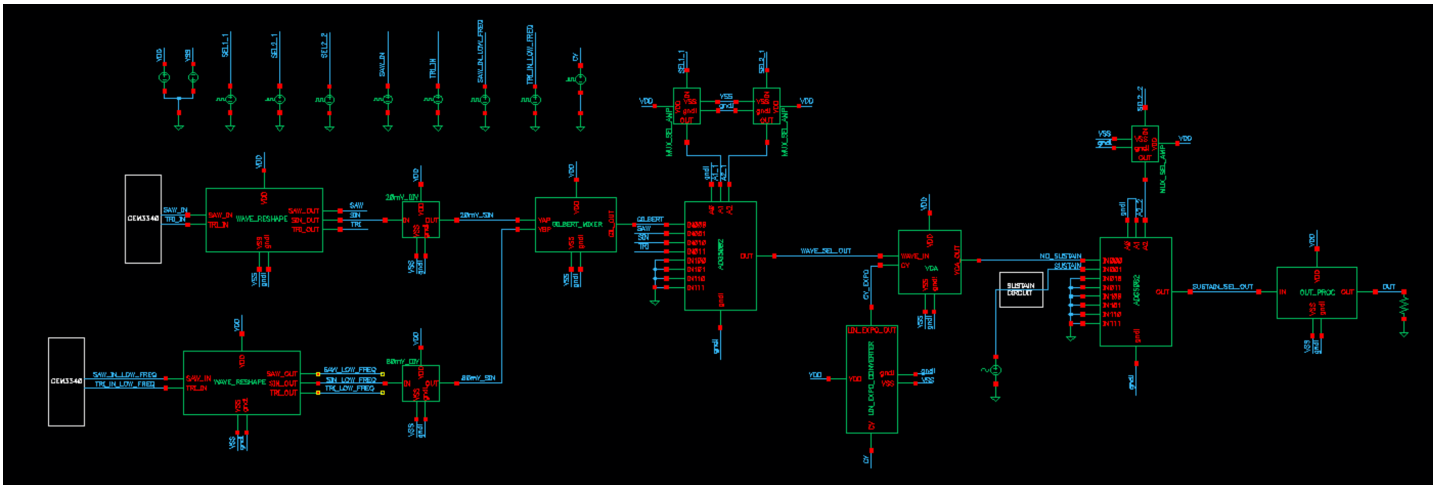


Figure 27: Fully Integrated Schematic

To demonstrate the fully integrated operation of the circuit, the first waveform snapshot shows two key signals: WAVE_SEL_OUT and the exponentiated VCA control voltage, CV_EXPO. As illustrated, the wave selector correctly switches between the available input waveforms, each normalized to a 1 V amplitude. This confirms that the GILBERT_MIXER is functioning as intended. Additionally, the VCA control voltage is successfully exponentiated, ensuring that amplitude changes occur on a logarithmic (dB) scale, producing a response that sounds natural to the human ear.

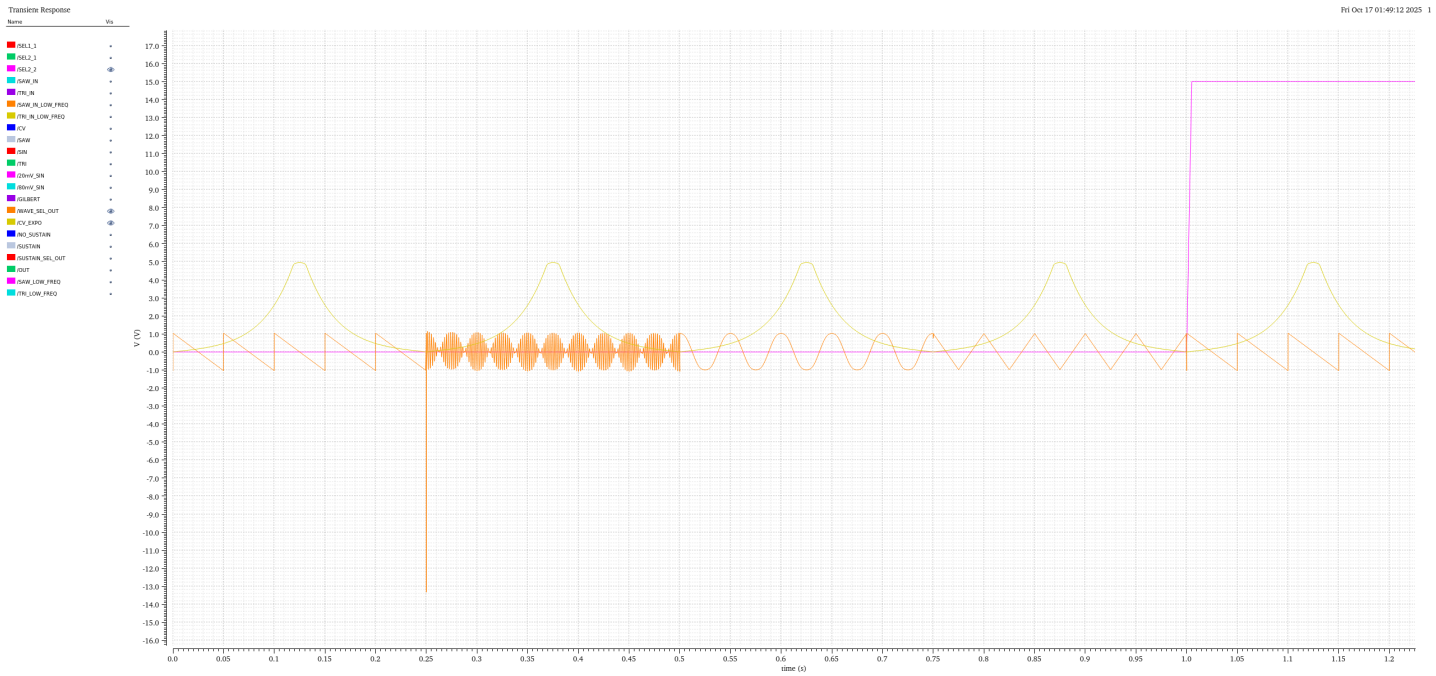


Figure 28: Output of WAVE_SELECT MUX and exponentiation of VCA CV

The next snapshot displays the output waveform of the VCA. As shown, the VCA accurately modulates the waveform's amplitude according to CV_EXPO. Additionally, the purple trace represents a mock-up of the SUSTAIN_MUX activation signal when it goes high. It indicates that the sustain effect has been selected. Since the sustain effect cannot be directly simulated in the integrated circuit, a raw sine wave is used to represent the expected sustain output waveform.

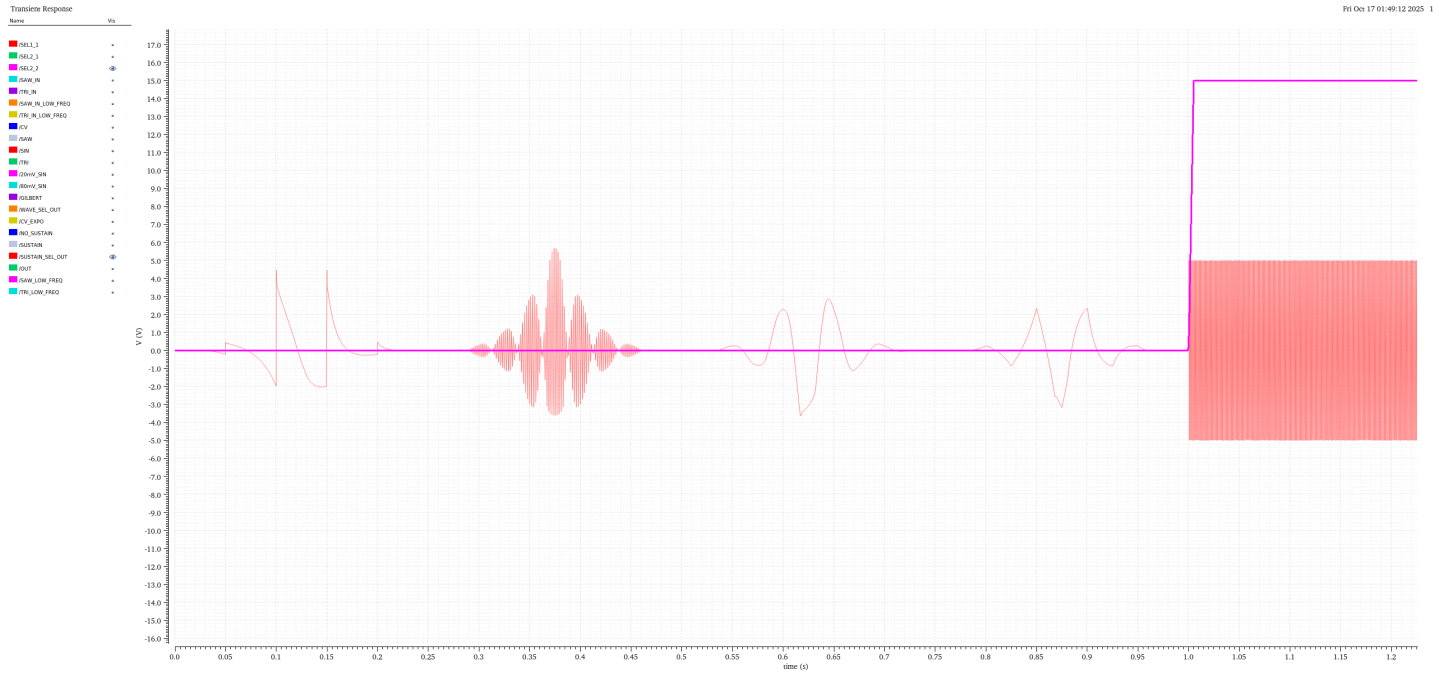


Figure 29: Output of the SUSTAIN_SEL MUX

The final snapshot shows the final output waveform. It is the same as the output of the VCA, except increased threefold. Due to the eccentricities of this design, the output of the VCA can only be up to 5V. In order to get the max volume given the constraints of the power supply, the waveform is tripled such that the max amplitude can be 15V.

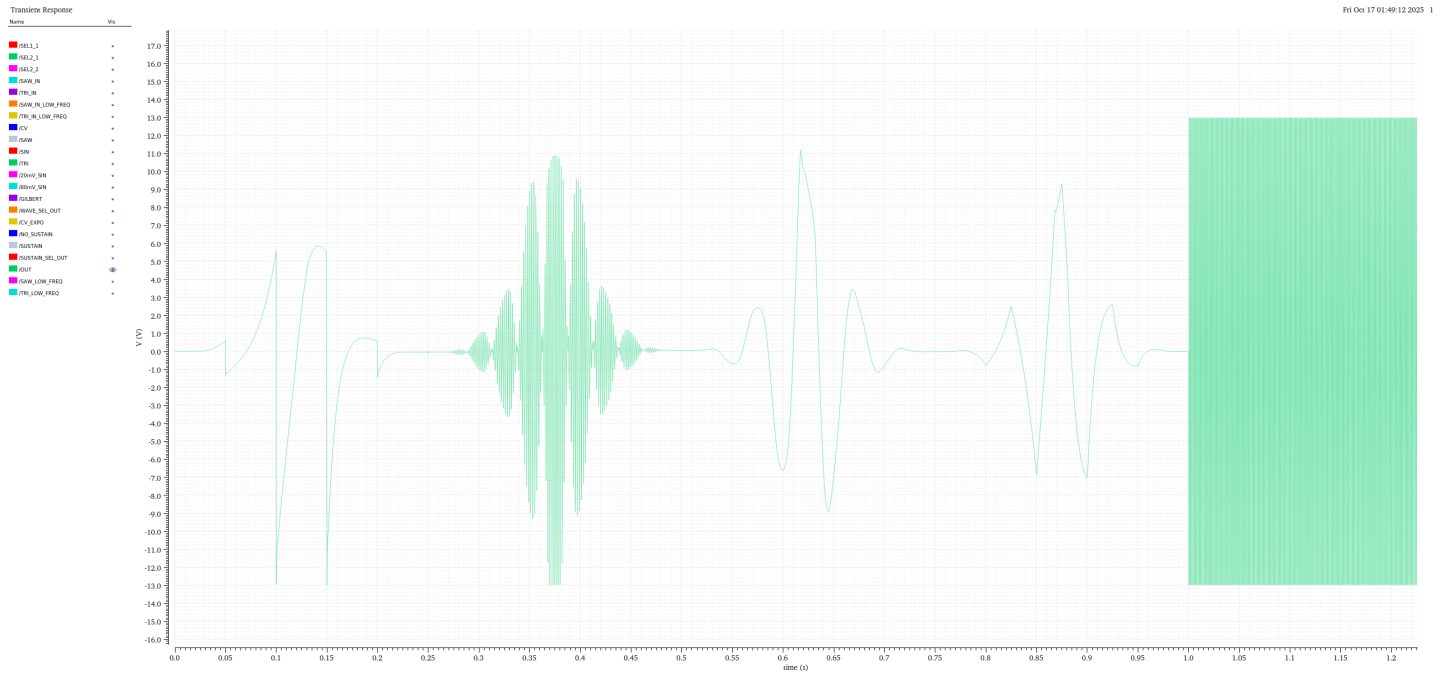


Figure 30: Final output waveform

Note that these simulations do not include VCO modulation, as accurately modeling the CEM3340 in Cadence is not feasible. To visualize the complete behavior, imagine the displayed waveforms exhibiting time-varying frequency in addition to their shown amplitude and shape characteristics.

7 PCB Fabrication

This section briefly documents the work completed to fabricate the PCBs for the physical circuit. The reader will first notice that a single, fully integrated board, was not manufactured. While full integration would have been ideal, the decision was guided by risk management. If one subsystem is miswired, debugging is significantly easier on a smaller, isolated board, rather than on a monolithic design. Furthermore, in the event that a board could not be salvaged, the affected block could be rebuilt on a perfboard without compromising the overall functionality of the final product.

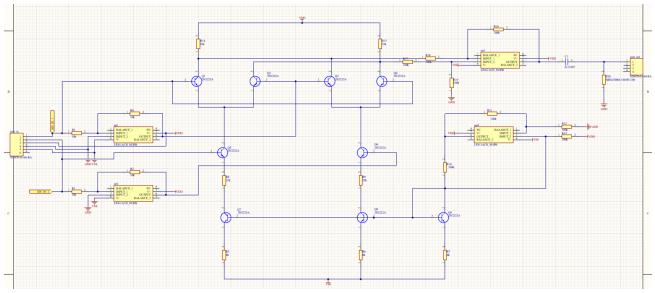
The fabricated boards also utilize through-hole components rather than the more conventional surface-mount packages. This decision was made for two primary reasons. First, it eliminates the risk of the project being stalled due to unavailable or mismatched components. Since all circuits were prototyped on breadboards prior to PCB fabrication, verified through-hole parts were already on hand. Second, while admittedly less practical, through-hole components contributed to the retro, bespoke aesthetic of this project.

All PCBs were designed in Altium, and both the schematic and layout files are available in the GitHub repository. Fabrication was performed by JLCPCB (https:jlcpcb.com). Given the relatively simple design and low layer count, the stack-up manager was configured as shown in the figure below.

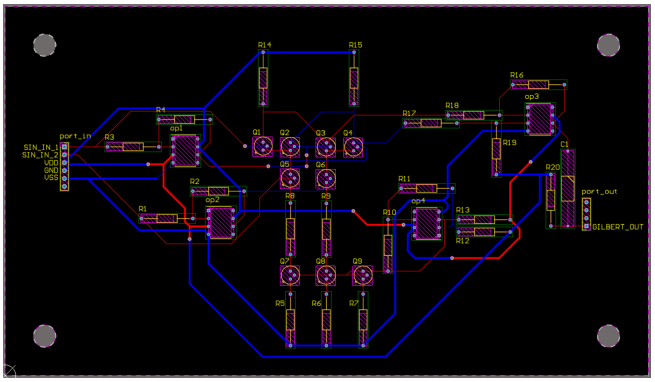
#	Name	Material	Type	Weight	Thickness	Dk	Df
	Top Overlay		Overlay				
	Top Solder	Solder Resist	Solder Mask		0.5mil	3.5	
1	Top Layer		Signal	1oz	1.378mil		
	Dielectric 1	FR-4	Dielectric		59.055mil	4.5	
2	Bottom Layer		Signal	1oz	1.378mil		
	Bottom Solder	Solder Resist	Solder Mask		0.5mil	3.5	
	Bottom Overlay		Overlay				

Figure 31: stack manager settings

Screenshots of the GILBERT_MIXER Altium schematic and layout are shown as an example of the layout design process



(a) SS of example Altium schematic (GILBERT_MIXER)



(b) SS of example Altium layout (GILBERT_MIXER)