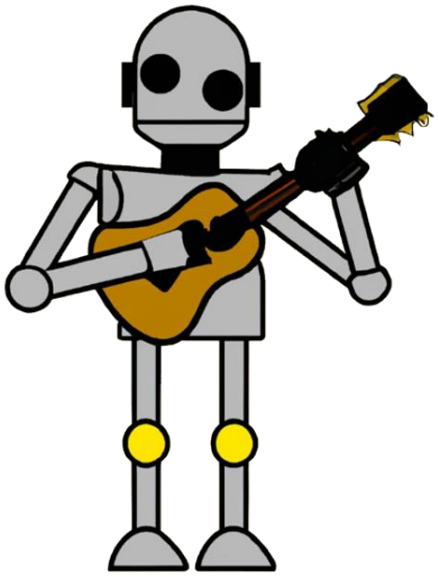
The UkeMaster 3000



Final Report

Team

The Dukes of Uke

Members

Nikodem Gazda

Ryan Simoneau

Contents

Introduction:

This project showcases an automatic ukulele that replays any recorded audio input on the ukulele. The UkeMaster 3000 utilizes a powerful DSP processor from TI capable of computing the Fast Fourier Transform to detect the notes played in a song. These notes then are sent to the DE-10 Lite FPGA to play the ukulele with a system of solenoids and servo motors. The UkeMaster also features a 144 LED strip that shows the spectrum of sound in real time to the user and an LED bar located on the front panel to indicate volume level.

The motivation for the UkeMaster 3000 is to serve as an educational tool for beginner ukulele players to learn how to play the instrument. Players can learn by using the device's recording feature to record songs they want to learn or themselves playing another instrument, and by studying the movements of the UkeMaster 3000 to improve their skills.

Competitive Products:

There are several products on the market that accomplish the same goals as the UkeMaster 3000 in slightly different ways, including self-playing pianos that can play pre-downloaded songs and other DIY projects that use the guitar and function very similarly to the UkeMaster 3000. Unlike the UkeMaster 3000, however, none of these other products feature any audio recording capabilities.

Figure 1: Self-Playing Guitar Example from YouTube

Project Architecture:

The DSP in the UkeMaster 3000 serves as the project’s main controller. It controls the LED bar and strip, the user interface, signal processing, and when a song records and plays. The FPGA controls the servos, which strum the strings of the ukulele, and the solenoids, which actuate down to fret the strings of the ukulele and change the notes. To play songs, the DSP and the FPGA have to communicate with one another to find the notes and then play them.

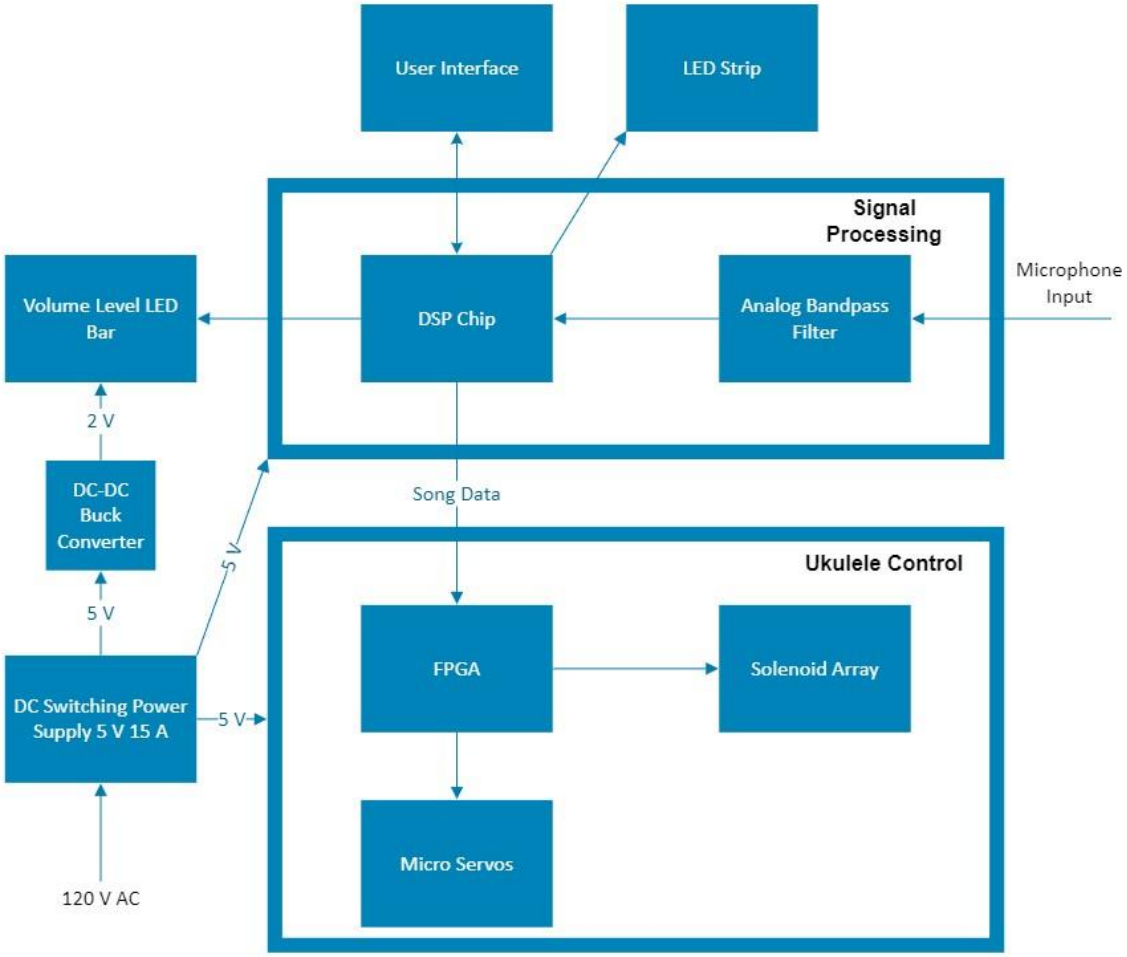


Figure 2: UkeMaster 3000 Block Diagram

Hardware and Software Design:

DSP Board:

The DSP Board contains the circuitry for the F28379D, the UI, the buck converter for the LED bar, and the pre-amp for the audio input for the F28379D.

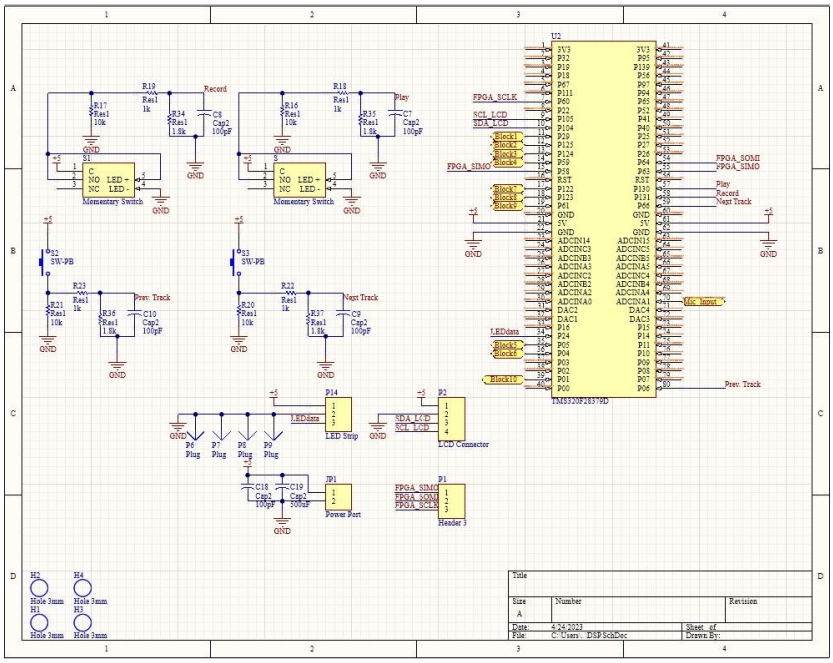


Figure 3: DSP Board Schematic Diagram

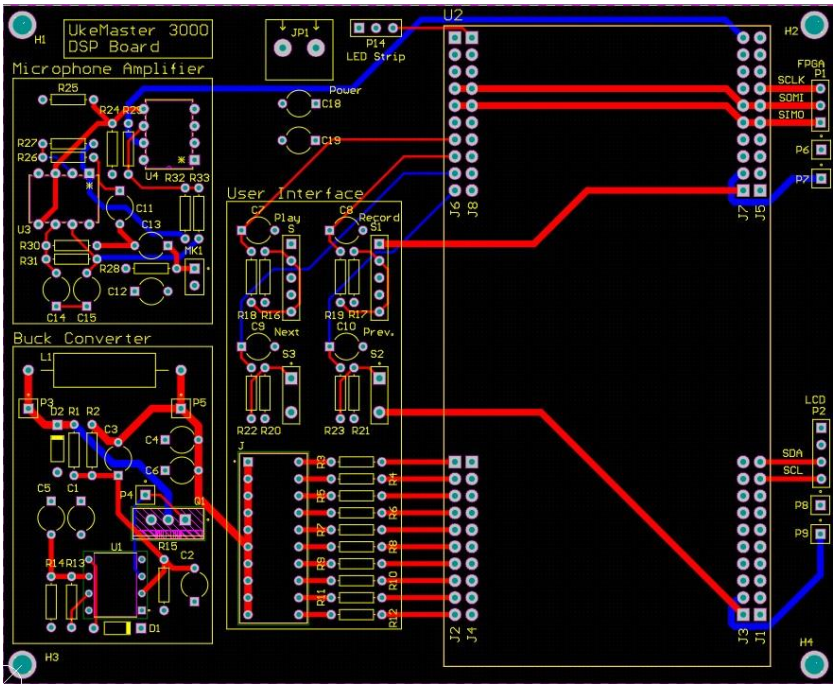


Figure 4: DSP Board PCB (Shelved Copper Pours)

User Interface:

Figure 5: UkeMaster User Interface

The UI is a simple state machine that the user can control via push buttons. There are 3 tracks that the user can record songs to each being 30 seconds in length at max, and there are 2 demo tracks that demonstrate playing a song, and the LED strip. The LCD screen at the center of the UI is used to provide user feedback and acknowledge the operations that the UkeMaster is doing.The UI is comprised of 4 buttons, an LED bar, an LCD display, and a Microphone. All 4 buttons are normally open, and the play and record buttons light up when they are pressed via LEDs that are internal to the buttons. To make the LEDs turn on when the buttons are pressed, the LED Positive terminal is connected to the NO terminal which is also connected to a pull-down resistor as shown in figure 3. With this topology, the LEDs turn on only when the buttons are pressed. Additionally, there are small capacitors shunt with the GPIO inputs to debounce all the buttons. The LCD is controlled via I2C from the DSP.

The state machine that controls the UI is based on a variable called “Task”. The value of Task will change due to user input through the push buttons, and this will in turn change what state the UI is in. This process is outlined in the flowchart below.

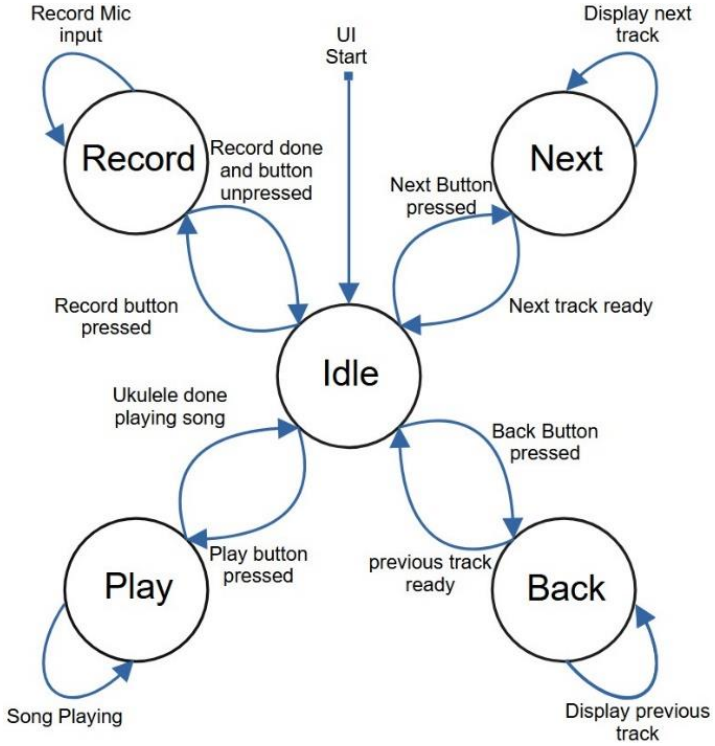


Figure 6: UI Flowchart

Buck Converter:

The Buck converter is necessary to power the 10 LED Bar used to indicate the microphone volume level. Each LED has a nominal forward voltage requirement of 2 V, and the supply voltage is 5 V. The LEDs are controlled via the DSP by connecting the cathode of the LEDs to GPIO through 80 Ohm resistors. Depending on the state of the GPIO pin, the LEDs will either be in forward or reverse bias. The Buck Converter contributes to the UkeMaster's Analog complexity design requirement.

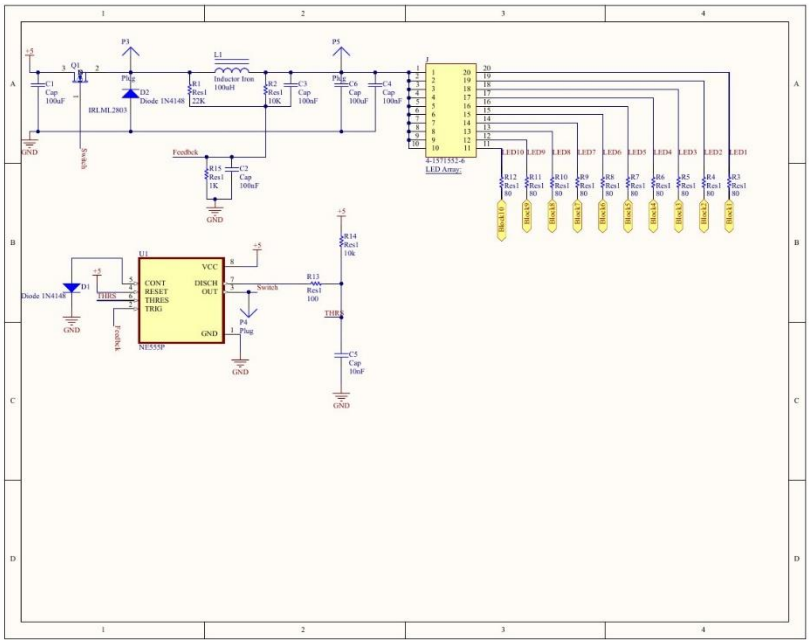


Figure 7: Buck Converter schematic

Microphone Amplifier/Filter:

The microphone circuit had a filter that limited the audio to a range of frequencies common to most instruments, and an amplifier that boosted the signal to the largest range allowed by the DSP chip.

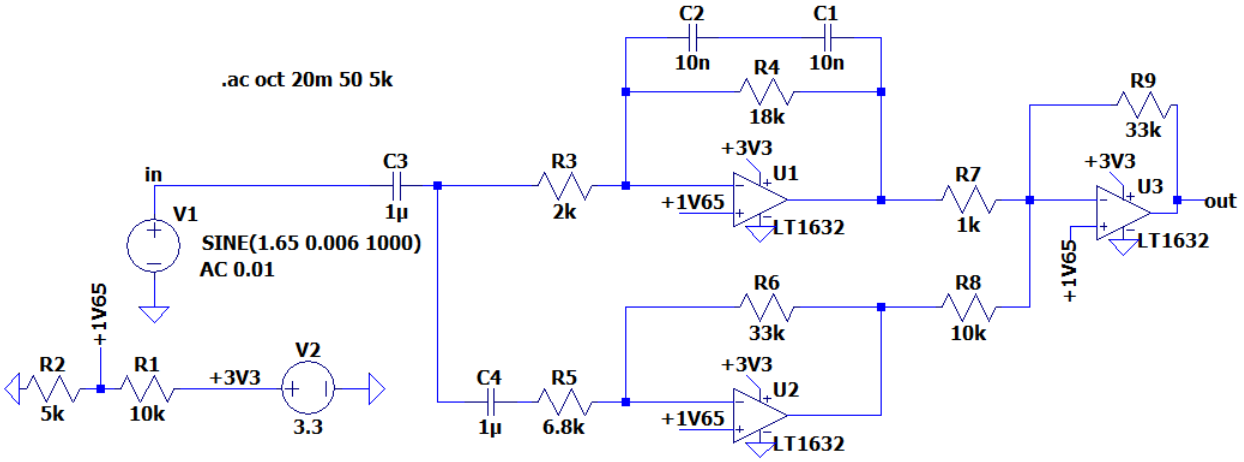


Figure 8: Spice schematic for the amplification/filtering circuit.

The spice simulation in Figs. 9 and 10 showed the amplifier would have a max gain of 50dB, which would produce a maximum output amplitude of 1.6V for an input signal of 5mA. This was desirable because the ADC in the DSP chip had a range of 0-3.3V, allowing for a maximum output amplitude of 1.65V. The simulation also showed the lower and higher cutoff frequencies to be around 100Hz and 2kHz, respectively, which would output a signal with the highest resolution in the ADC and within a relevant range of frequencies spanning approximately 4.5 octaves.

Figure 9: Maximum amplitude and lower cutoff frequency for the amplification/filtering circuit. The input waveform has a magnitude of -40dB, and the output waveform has a magnitude of ~10dB, resulting in an amplification of ~50dB. The lower cutoff frequency is at ~100Hz, or around the note G2.

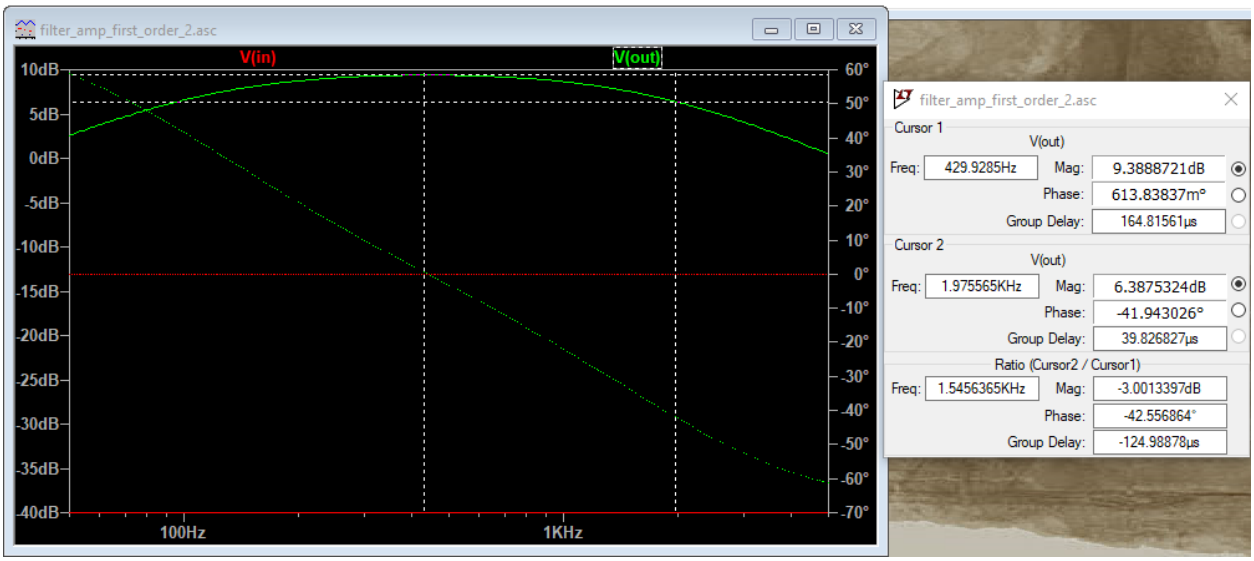


Figure 10: Maximum amplitude and upper cutoff frequency for the amplification/filtering circuit. The upper cutoff frequency is at ~2kHz, or around the note B6.

However, during implementation, the measured amplification was only 37dB, which was 13dB less than the simulation. To compensate for this loss without affecting the cutoff frequencies, the feedback resistor for the rightmost op amp in Figure 8 was replaced with a 50kOhm resistor. As a result, the frequency range had cutoff frequencies of around 80Hz and 2kHz, which correspond to the notes E2 and C7 respectively, instead of the notes G2 and B6 as in the simulation, leaving a slightly wider range of frequencies.

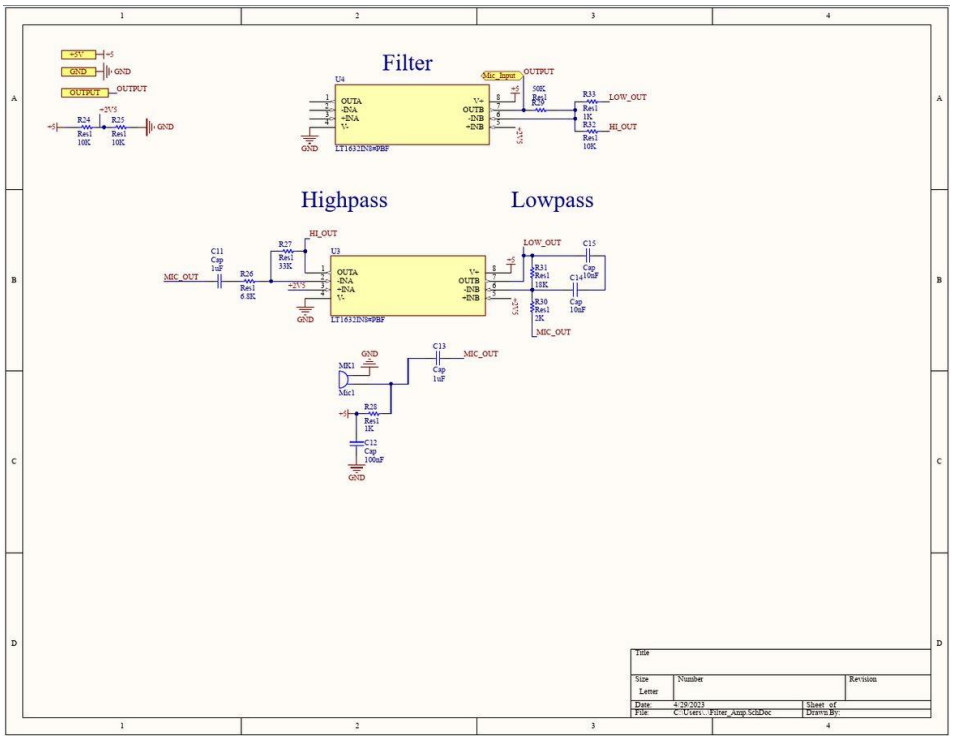


Figure 11: Schematic for implemented amplification/filter circuit. This schematic differs from the previous spice schematic in the feedback resistor. This value needed to be adjusted to increase the gain of the circuit.

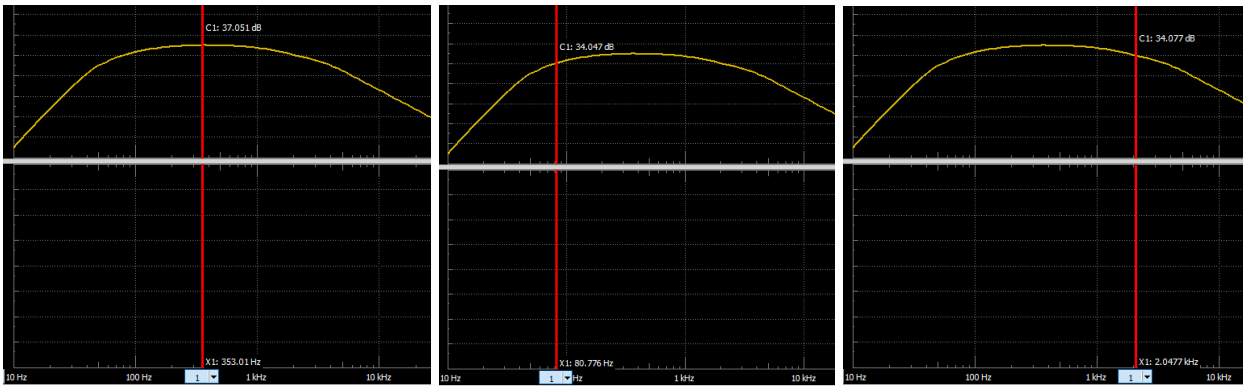


Figure 12: Maximum magnitude and cutoff frequencies for implemented amplification/filter circuit. These plots show that the implemented circuit had a maximum gain of ~37dB, ~13dB less than the simulated circuit. The feedback resistor was raised in the summing amplifier labeled as "Filter" in Figure 11. The cutoff frequencies for the implemented circuit were around 80Hz and 2kHz, or around the notes E2 and C7.

Solenoid and Servo Motor PCB:

The schematic and PCB responsible for the solenoid and servo controls are displayed below. The main purpose of this board is to interpret information sent from the DSP chip and translate it to the corresponding arrangement of activated solenoids and servos.

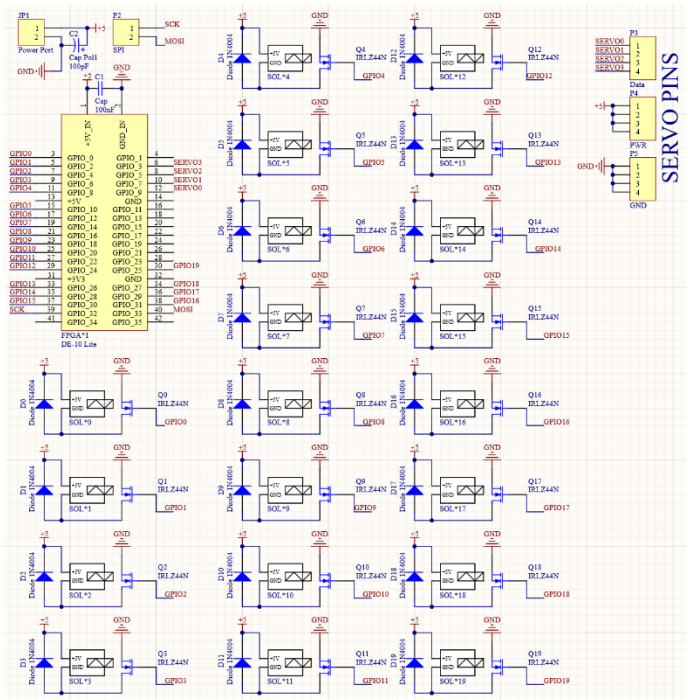


Figure 13: Schematic for the solenoid and servo motor controller PCB.

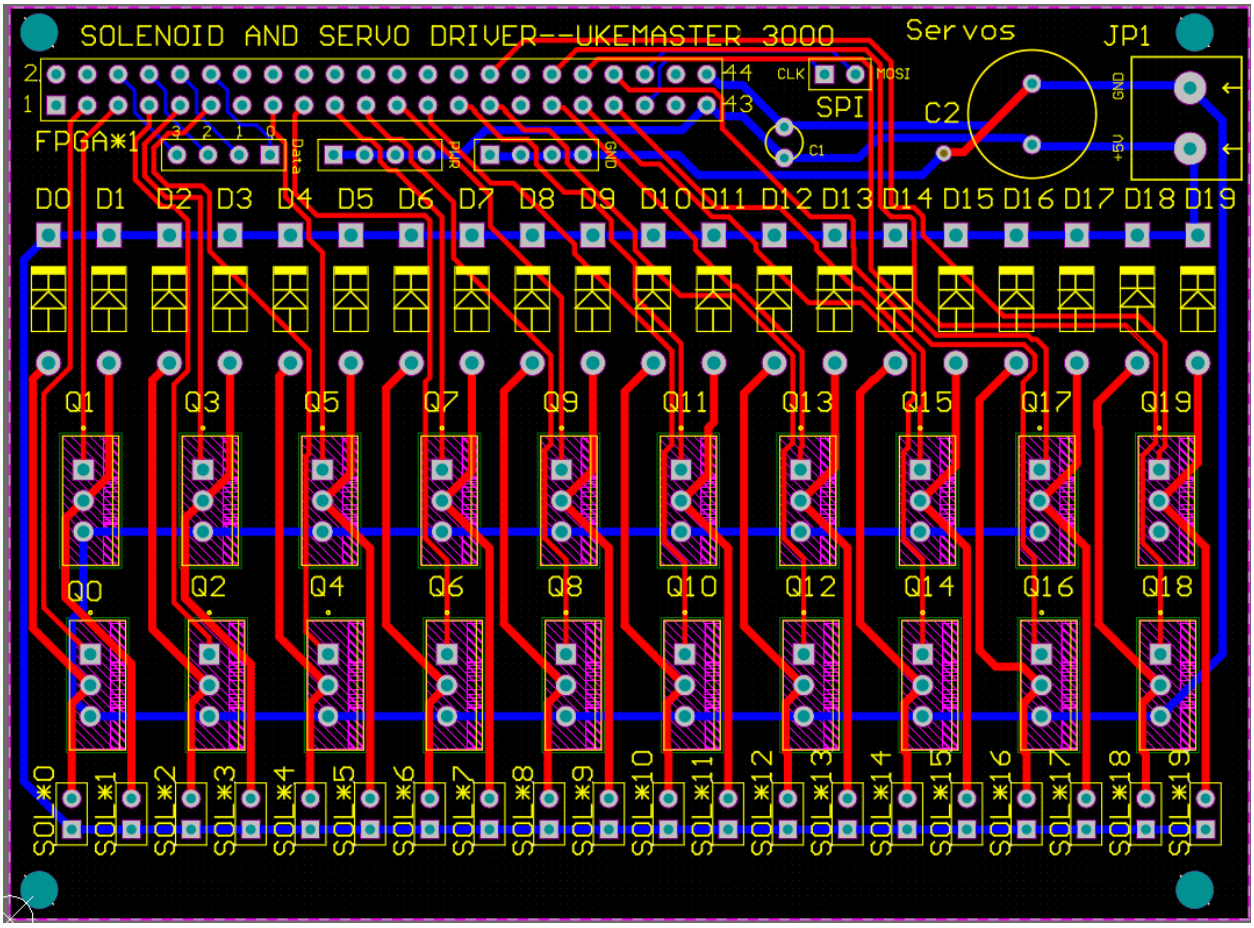


Figure 14: Schematic for the solenoid and servo motor controller PCB.

UkeMaster 3000 Housing:

The housing for the UkeMaster 3000 is made out of wood, and it was designed using CAD. We used OnShape to test our design and we made models of the Ukulele, power supply, solenoids, and other parts with accurate dimensions to understand what considerations we would have to make when we made the final construction.

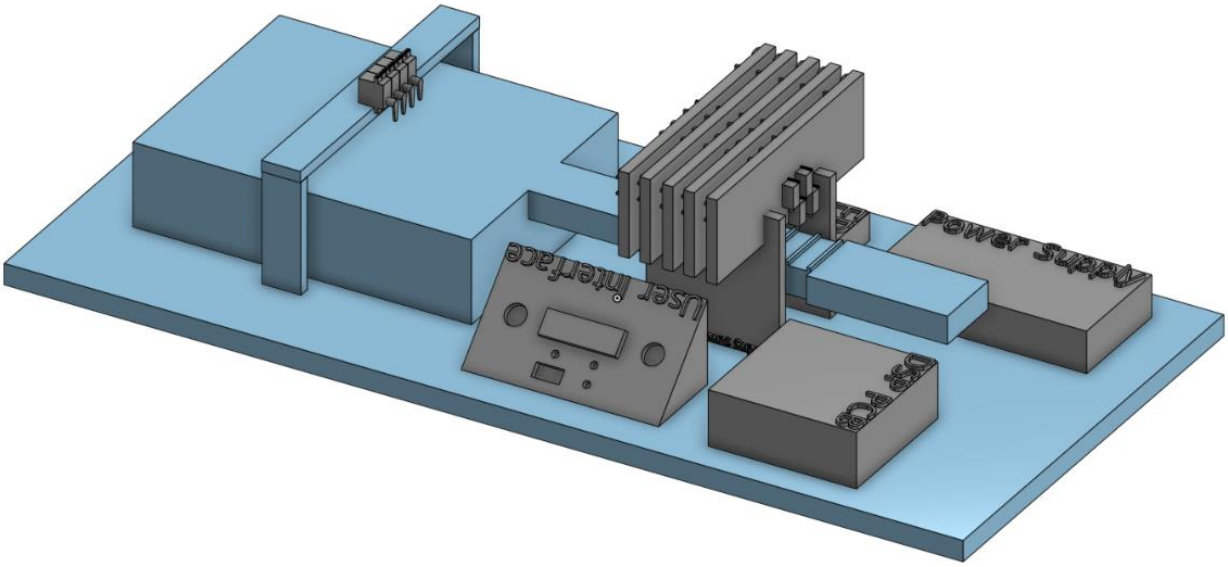


Figure 15: OnShape Proof of Concept

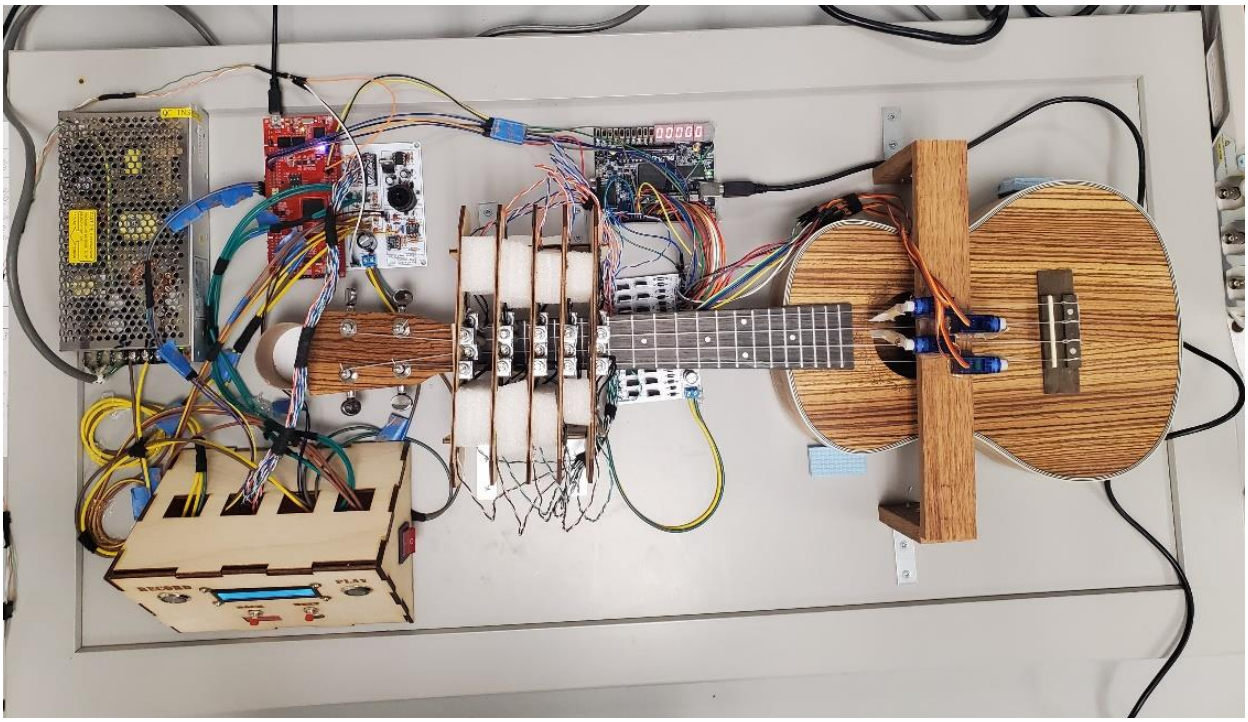


Figure 16: Final Design

Cad Designs:

After making a proof of concept in OnShape, we took the ideas that we had and moved them into Solidworks so that our designs could be cut out of wood using the shop's laser engraver.

Based on what we learned from the OnShape model, there were several modifications that we would need to make to the design for the UkeMaster to work properly. First, the Solenoids are too wide to be lined up directly next to each other and reach all 4 strings, so the solenoids had to be staggered vertically to make them all fit. Additionally, the width of each fret on the ukulele reduces fret after fret, and the 5th fret is only x inches wide. To accommodate this we added cutouts where the solenoids are placed so that there was enough room for all the solenoids to be placed above the frets.

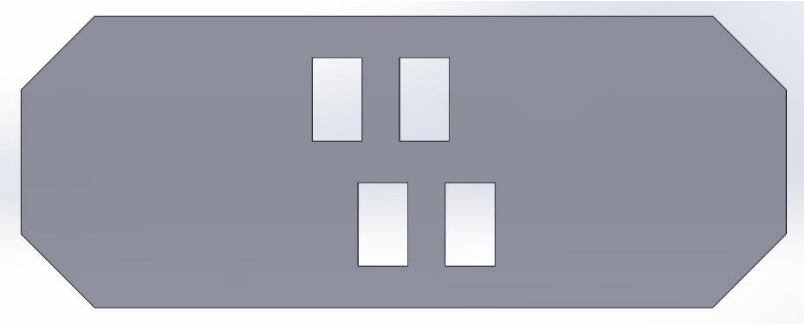


Figure 87: Solenoid Mount

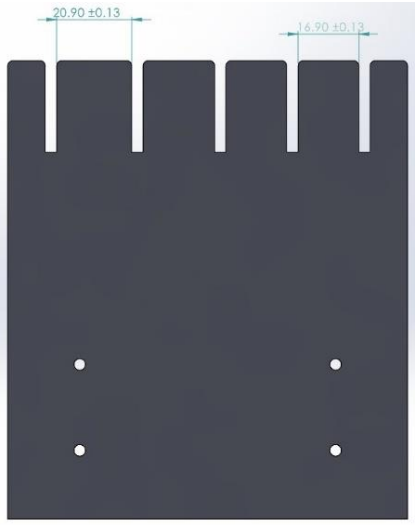


Figure 18: solenoid mount holder

We also found that the Servo motors used to strum the ukulele were too large to put them directly next to each other as well, so we had to stagger them on the platform they rested on to have them fit. Additionally, the original idea for the UI box to be a right triangle had to be changed because the online tool we used to make the box only had an option for equilateral triangles.

DSP Code:

The DSP code is structured into a large switch statement that both controls the state of the UI and what operations the DSP will complete depending on what the user input is. The switch state that controls the UI is elaborated in the "User Interface" sub-section under the "DSP Board" section in figure 6. Below is a flow chart that outlines the high level structure of the program.

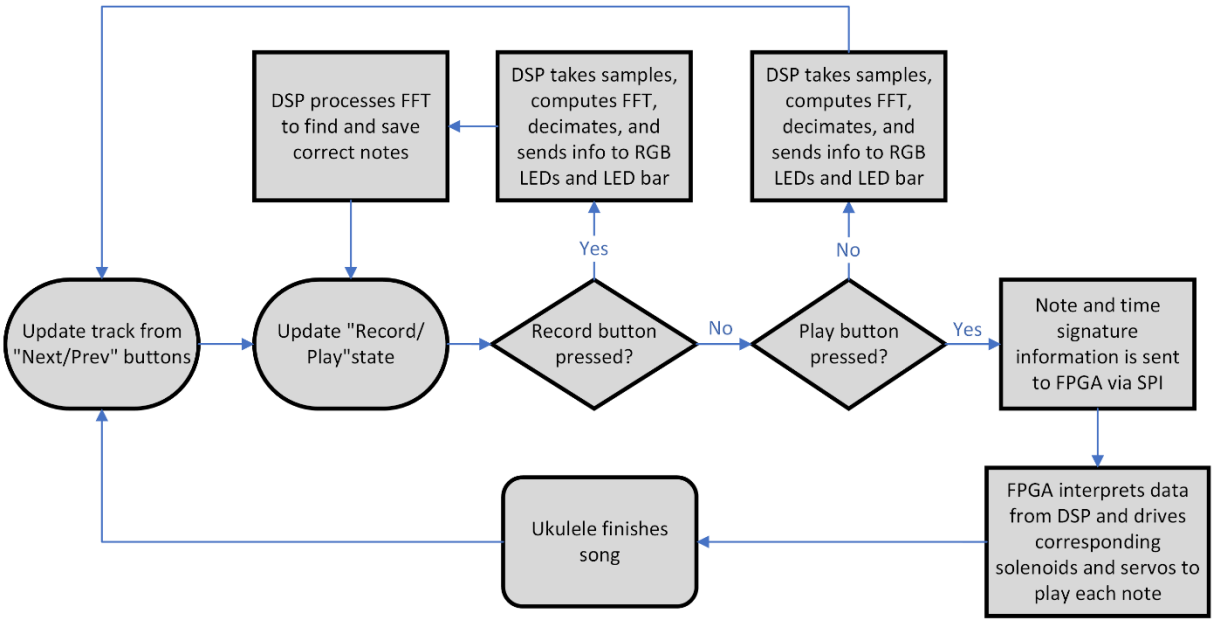


Figure 19: Software flowchart.

FFT Implementation

The FFT we implemented utilizes the Contrhigh-levelelerator (CLA) available on the F28379D which is a separate processing unit built into the processor itself. Since the CLA is a separate processor, it is possible for it to compute the FFT in parallel while the main processor does signal processing on previous FFTs.

There are several key parameters that are important for properly calculating the FFT and they are outlined in the table below.

Parameter	Value	Notes
Number of Bins	1024	This is the largest number of bins supported by the CLA built in FFT algorithm
Sample Frequency (Fs)	10240 Hz	This is set intentionally at 10x the # bins so that the frequency resolution is set to 10 Hz/bin
Frequency Resolution	10 Hz/Bin	10 Hz/Bin was chosen to balance between distinguishing notes within our desired frequency range and to avoid aliasing the higher frequencies
ADC Resolution	0.806 mV/bit	Achieved with a 0 to 3.3 V ADC and 12 – bit resolution.
FFTs per second	10	10 FFTs per second gives us 100 ms between each which is enough time to detect when notes change

Signal Processing:

All signal processing is done in the “Record” state after the FFT is computed for the current iteration. After taking the FFT, each captured sample undergoes 4 processes before being recorded in the current recording track: finding the peaks, encoding the data into a format readable by the FPGA, checking for repeat codes, and reducing erratic behavior.

Finding Peaks:

The algorithm for finding the peaks in the FFT iterates over each bin in the FFT and checks if the magnitude at that bin is greater than that of the bins before it as well as if it’s greater than a specified threshold minimum peak value. Once the bin with the greatest magnitude is located, the bin number and magnitude are recorded, and the next pass begins to find the bin with the second greatest magnitude. If the bin number for the second greatest magnitude is within a specified width of the bin with the greatest magnitude, the bin is not recorded. This method allows the algorithm to find as many peaks as needed in descending order of magnitude, which then is sorted in ascending order of frequency.

The inputs for the finding peaks algorithm take the pointer to the FFT, the bin/frequency to start iterating from, the bin/frequency where iteration ends, the width of each peak, a threshold for the minimum peak value, the number of peaks to find, and the pointers to the magnitudes and bin numbers of the selected peaks. In our implementation, we found the algorithm performed best when starting at 255 Hz and ending at 610 Hz, with a peak width of 20 Hz, minimum threshold peak value of 75, and a maximum number of 4 peaks. The bin/frequency range was chosen to restrict the system to notes playable by the UkeMaster 3000. A peak width of 20 Hz prevents the detection of multiple neighboring notes when only the center note with the highest magnitude truly is present. This choice was appropriate given the separation between each note in the lower frequencies is less than 20 Hz. The threshold minimum peak value and maximum number of peaks was tuned during the use of the UkeMaster 3000.

While testing, we also found the recordings tend to prefer higher frequency notes and sometimes omit lower frequency notes, so the algorithm was adjusted so that the threshold for frequencies below 360 Hz was 60% of the threshold for frequencies greater than 360 Hz. In other words, lower frequency peaks have a lower threshold for detection.

Encoding:

Once the algorithm finds the 4 highest peaks in the FFT, this information must be translated into a code readable by the FPGA. This is a two step process: choosing where to play each note on the ukulele and converting those locations to a 12-bit code.

Placing Notes on the Fretboard:

The difficulty in choosing where to place each detected note arises from the presence of the same notes on multiple strings. For example, within the section of the fretboard accessible to the solenoids and servos in Fig. 20, the note A4 exists on strings 1, 3, and 4. Which of these strings should this note be placed on when this note is detected? Furthermore, will the method we choose ensure each note is played when possible?

Pitch	Open	1	2	3	4	5	
4	A4		B4	C5		D5	String 4
2	E4	F4		G4		A4	String 3
1	C4		D4		E4	F4	String 2
3	G4		A4		B4	C5	String 1

Figure 20: Ukulele note chart.

For example, if the FFT were to capture a C chord comprised of notes C5, E4, C4, and G4, ideally the ukulele would play note C5 on string 4, note E4 on string 3, note C4 on string 2, and note G4 on string 1:

Pitch	Open	1	2	3	4	5	
4	A4		B4	C5		D5	String 4
2	E4	F4		G4		A4	String 3
1	C4		D4		E4	F4	String 2
3	G4		A4		B4	C5	String 1

Figure 21: Ukulele note chart for correct C chord example.

If our encoding algorithm instead chose string 1 for C5, string 3 for G4, and C4 for string 2, note E4 no longer would be able to be played on string 4. Moreover, which note should the algorithm choose when two notes only available on the same string are detected, for example, notes C4 and D4 (Fig. 20)?

Pitch	Open	1	2	3	4	5	
4	A4		B4	C5		D5	String 4
2	E4	F4		G4		A4	String 3
1	C4		D4		E4	F4	String 2
3	G4		A4		B4	C5	String 1

Figure 22: Ukulele note chart for incorrect C chord example.

We found iterating through the detected notes in ascending frequency and then iterating each note through each string in ascending pitch solved most overlapping conflicts. In the case of the C chord example, the list of notes in ascending order of frequency is C4, E4, G4, and C5, and the list of strings in ascending pitch is string 2, 3, 4, and 1. Applying this method would assign C4 to string 2, E4 to string 3, G4 to string 1, and C5 to string 4, placing all notes on the fretboard, as seen in Fig. 21. In the case where two notes only exist on one string, for example with notes C4 and D4, the note with the lower frequency is chosen. This method consistently placed the maximum number of notes possible on the fretboard with no overlapping errors.

Converting Notes to SPI Code:

Once the detected notes are placed on the fretboard, their placements need to be converted into a code readable by the FPGA. In this encoding process, 3 of the 12 available bits in each code are allocated to each string, demonstrated in Fig. 23. Each

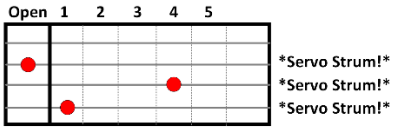
string can only play one note at a time and has 5 solenoids, so only 6 different states are needed to control the solenoids—one state for each solenoid to activate and one state where no solenoids are activated. The servos only require two states—strum or remain in the same position. Combining these states led to the encoding process demonstrated in the table of Fig. 23.

	String 1	String 2	String 3	String 4
Code:	111	111	111	111

Solenoid #	No note	Open	1	2	3	4	5
Code	000, 111	001	010	011	100	101	110

Examples:

Code: 010101001111



Code: 110100011010

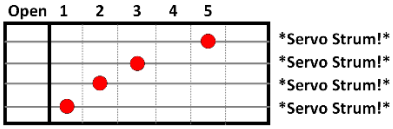


Figure 23: Demonstration of how the FPGA breaks down the SPI code sent from the DSP chip.

Repeat Codes:

While the UkeMaster 3000 is able to function after implementing the finding peaks algorithm and encoding process, the ukulele's output does not accurately resemble the audio input because the servos strum on every non-zero code. If the audio input were a sinusoidal wave for the note A4, for example, the entire recording track would be filled with the code 0b111 110 111 111. While this behavior is correct, the FPGA interprets this input as a series of individual notes and strums the string for each consecutive code. Ideally, all repeat codes would be replaced with the code indicating no note is played:

Recording Track
0b111 110 111 111
0b111 110 111 111
0b111 110 111 111
0b111 110 111 111
0b111 111 001 111
0b111 111 001 111
0b111 111 001 111
0b111 111 001 111

→

Recording Track
0b111 110 111 111
0b111 111 111 111
0b111 111 111 111
0b111 111 111 111
0b111 111 001 111
0b111 111 111 111
0b111 111 111 111
0b111 111 111 111

Figure 24: Demonstration of repeat code replacement in the recording track.

This implementation ensures each new note is played only once on the ukulele.

Erratic Behavior:

While testing the UkeMaster 3000, we noticed that after capturing a note the recording track sometimes also captured similar notes close to the original note. These additional notes would cause the playback of tracks on the ukulele to sound discordant and unpleasant. One way to solve this issue is to adjust the peak width value in the peak finding algorithm, however, adjusting this value any more caused the recording to miss some notes. This method also becomes more difficult when considering the number of bins between notes is much greater for higher frequency notes than lower frequency notes, so the peak finding algorithm would need to be reworked to adjust the peak width for each bin value.

Additionally, the UkeMaster 3000 was experiencing an issue where the solenoids would press down and return almost immediately, since new codes are sent to the FPGA at a rate of 10Hz, and the servo motor could not rotate quick enough to strum the string while the solenoid is pressed against the string. Therefore, we implemented a procedure which prevents any new codes from saving in the recording track within 4 consecutive codes of a unique code:

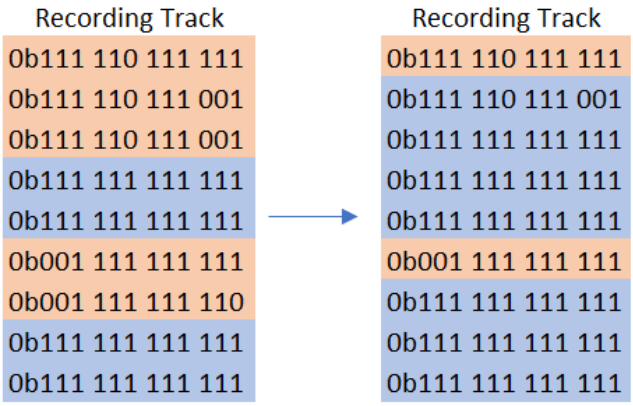


Figure 25: Demonstration of the removal of erratic code in the recording track.

This procedure accomplishes two tasks: the removal of erratic codes saved in the recording track as well as a buffer for the FPGA to hold solenoids down for longer. “Solenoid Modules” in the “FPGA VHDL” section of the report explores this topic further within the FPGA.

LED Strip:

The LED strip is utilized in the project to provide real time user feedback. When anything is played into the microphone while the UkeMaster is idle or recording, the LED strip displays the spectrum of the audio from 200 Hz up to 1640 Hz to accommodate a multitude of sounds that can be played into the UkeMaster.

There are only 144 LEDs in the LED strip, and the FFT has 1024 points, so the FFT has to be decimated before it can be output to the LEDs. The decimation is completed by averaging neighboring bins so that some information from all bins is retained. Below is an FFT from the song “Slavic Fantasy in B Minor” and the decimation begins from bin 40.

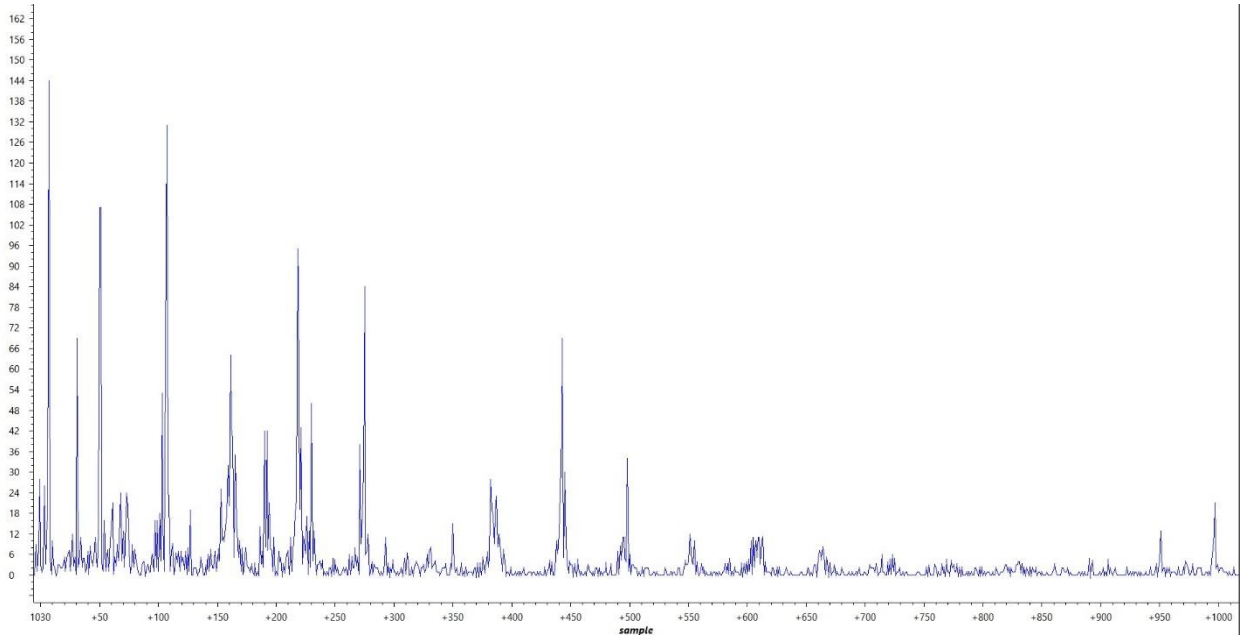


Figure 269: Undecimated FFT

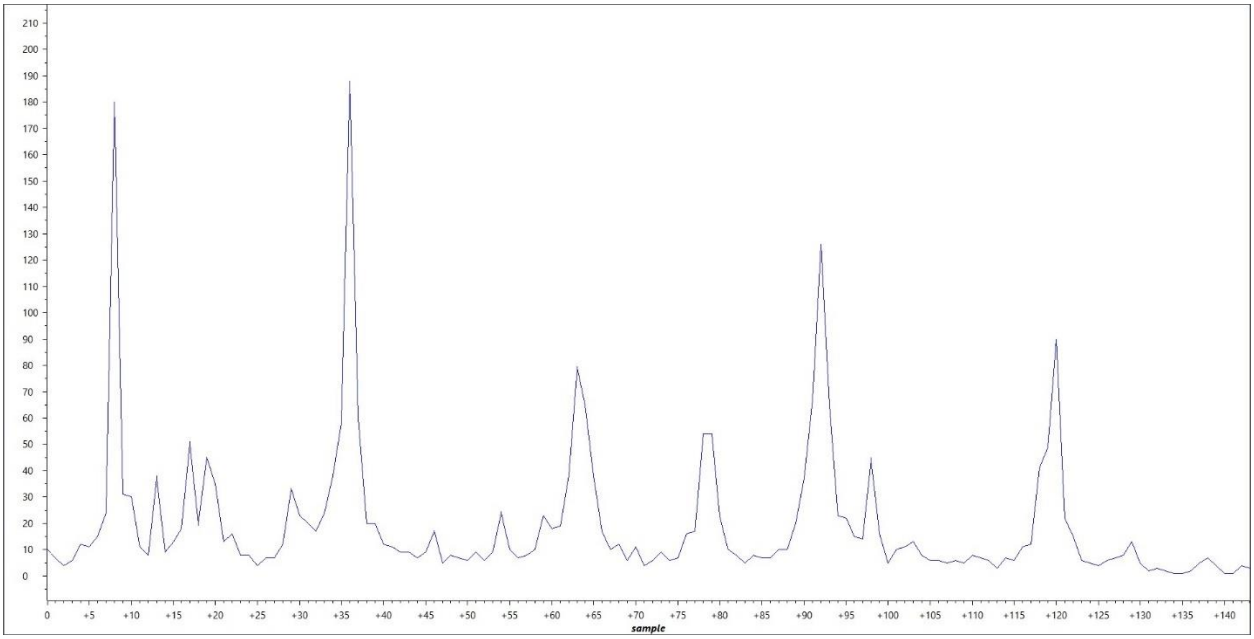


Figure 27: Decimated FFT

Once the FFT has been decimated, the raw data needs to be converted into a format that the LEDs can accept. The LEDs use the WS2812B control IC to allow them to be individually addressable, and specific timing and data formatting is required to have the LEDs work properly. Each LED expects to receive a 24 – bit RGB code, and after the first LED receives 24 bits the next 24 bits will be sent to the next LED. 24 bits is split into 3 8 – bit colors in G, R, B sequence. The data is sent serially and the timings that correspond to a 1 and a 0 are outlined below.

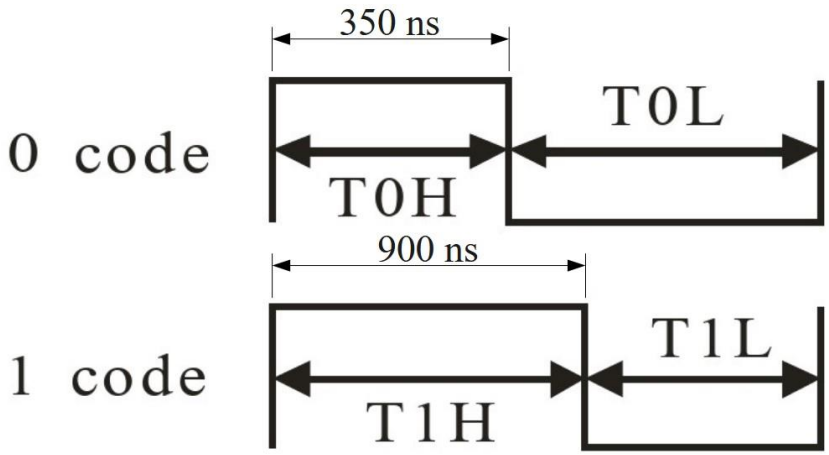


Figure 28: WS2812B Timing Requirements

The F28379D does not have any libraries to create these precise timing requirements like PICs and Raspberry Pi's do, so the drivers had to be made ourselves. Making the PWM signal using bit banging would be a very time-consuming process for the processor and could delay the Signal Processing stage of the project, so we needed a different approach if we wanted to implement the LEDs without adding another processor to the project. We settled on an approach we found from Microchip that utilizes a programmable logic block peripheral within PIC processors. The method uses SPI to send 8-bit data to the logic block, and the logic block takes the SPI data and converts it into WS2812B serial data using the SPI MOSI data, SPI CLK, and a separate PWM Clock. An explanation of the implementation is outlined in the figures below.

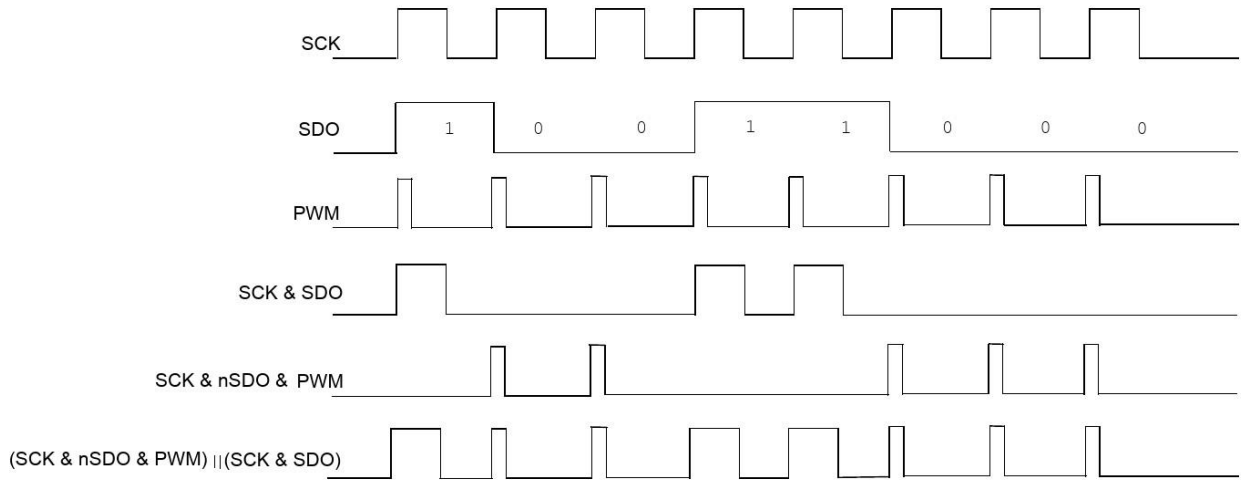


Figure 2910: WS2812 Timing Diagram

The Logic block generates '1' pulses based on the SCK and SDO, and '0' pulses based on SCK, ~SDO, and the PWM signal.

$$WS2812B = (SCK)(SDO) + (SCK)(\overline{SDO})(PWM)$$

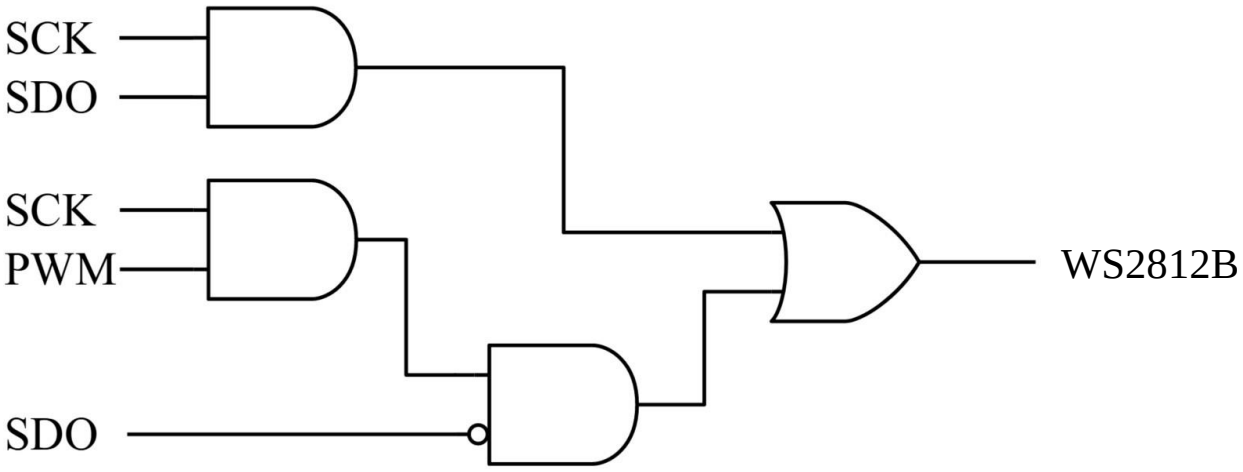


Figure 3011: WS2812B Logic Implementation

Interfacing Between Boards:

Once the user records a track, the DSP chip needs to send all the recorded notes to the FPGA using the SPI communication protocol. Each code in the recording track represents one audio sample. Given that the ADC collects 10 samples per second, the DSP chip also sends 10 unique codes to the FPGA per second. The DSP chip and FPGA interact through 5 signals: the SPI clock, the SPI data line, the SPI chip select, a reset signal, and a failed send signal.

Before the DSP chip begins transmitting a track to the FPGA, the reset is pulled low, the chip select is pulled high, and the failed send variable is initialized to 0. This means that unless a track currently is playing, the FPGA is always in reset. When data transmission begins, it begins at the third index of the recording track to prevent unwanted noises in the first 20ms of the recording from playing.

Each unique code transmission actually consists of two transmissions of the same code for data corruption checking. These two transmissions are compared in the FPGA and the failed send signal is set high to indicate an error occurred. After sending both transmissions, the DSP chip probes the FPGA for the failed send signal, and if set, the current iteration repeats until the unique code is sent without data corruption. Additionally, the chip select signal is set high after each transmission (and low before each transmission) to reset the counters within the FPGA and prevent data corruption from propagating past each individual transmission. After the successful transmission of a unique code, the DSP chip delays for 100ms before sending the next unique code to replicate the rate at which the ADC samples audio.

FPGA VHDL:

The main purpose of the FPGA is to receive streams of SPI code from the DSP chip and activate the solenoids and servos corresponding to the current bit stream. A breakdown of the main functionality is displayed in Fig. 23. The FPGA architecture can be described with three central components: The SPI module, solenoid modules, and servo modules:

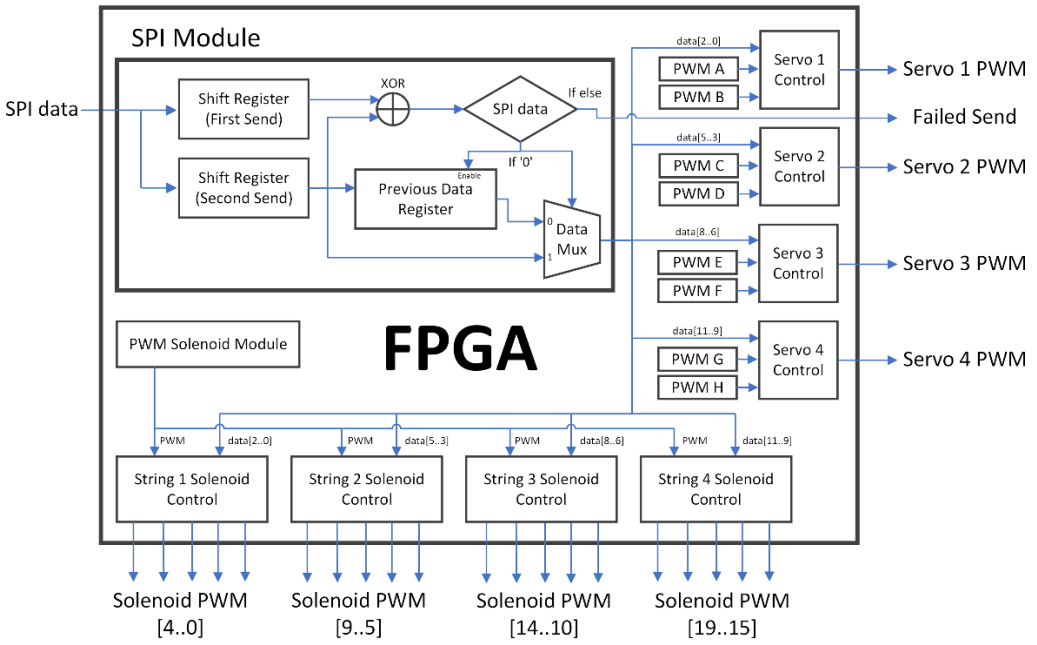


Figure 31: Simplified block design of the FPGA architecture.

Once the FPGA decodes the serial data received from the DSP chip, a PWM signal is sent through the corresponding solenoid and servo output for each string. While a PWM signal is necessary to operate the servo motors, in the case of the solenoids, a PWM signal is used to lower power consumption. If the FPGA were to decode the signal seen in the first example of Fig. 23, solenoid lines 0 and 8 would output a 200 Hz PWM signal with a duty cycle of 80% while the rest of the solenoid outputs stayed low. Additionally, if the received code were the first code in the track and all the servo motors were currently set to a duty cycle of 8%, then servo motors for strings 1, 2, and 3 would switch to output a 50 Hz PWM signal with a duty cycle of 10% while the servo for string 4 would remain at its original duty cycle of 8%.

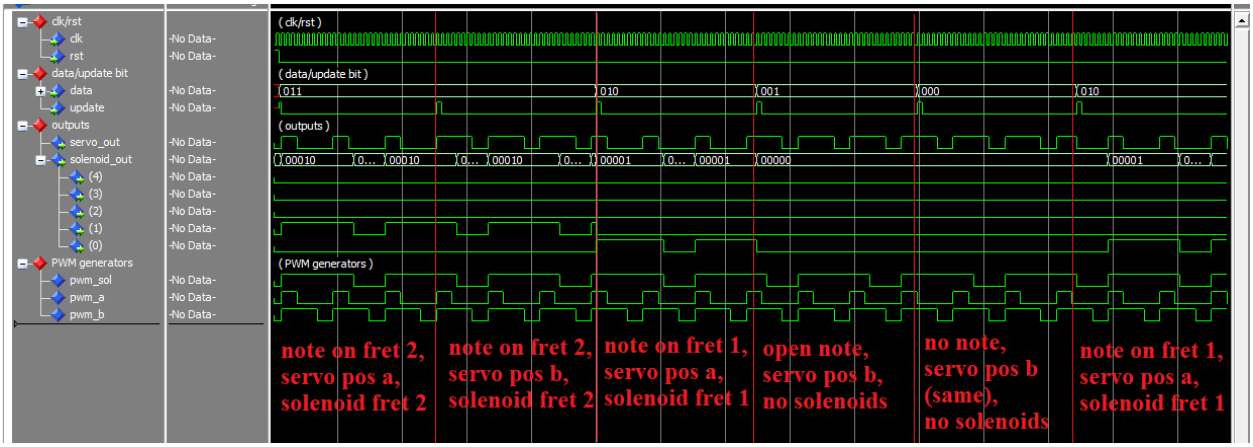


Figure 32: Simplified simulation of FPGA decoding serial input from DSP chip and activating the corresponding solenoids and servo motors for a single string.

Figure 32 illustrates the expected solenoid and servo outputs for different sets of data, using a single string as an example. In this simulation, signals pwm_a and pwm_b control the PWM of the servo motor and signal pwm_sol controls the PWM of the solenoids. Since the solenoids' PWM is used to lower power consumption and should stay the same for each solenoid, the same PWM signal can be muxed to each solenoid output.

This simulation is meant to display the solenoid/servo controllers and not the SPI module. Therefore, it is worth noting that the solenoids and servos update at the beginning of each data update for this simulation. The final operation of the FPGA not pictured in this simulation includes data corruption checking as well as delays for correctly-timed fretting, strumming, and overall operation.

The first set of data sent to the FPGA is "011"; as seen in Figs. 23 and 32, this code corresponds to the second solenoid activating. The next set of data is the same, but the update bit still goes high, indicating the code on this next set still is new information and the solenoids and servos should update correspondingly. Since the code is the same, the previously activated solenoid remains activated but the servo switches from pwm_a to pwm_b to strum the string. The set of data that follows activates the first solenoid and strums the string and the next set activates no solenoids but still changes the position of the servo to strum the string. This action corresponds to an open note, where the string does not need to be fretted—just strummed. The set of data received directly afterwards indicates no note should be played (Fig. 23) and the FPGA refrains from activating any solenoids or strumming the string.

SPI Module:

The SPI module in the FPGA is responsible for recording the codes sent from the DSP chip and checking for signs of data corruption. As seen in Fig. 33, the SPI module takes in the SPI signals for the clock and data, as well as a reset signal, then outputs the data as well as a signal to update other modules that the data register has updated.

```
entity SPI_module is
    generic (
        buffer_width  : positive := 12
    );
    port(
        SCK                 : in  std_logic;
        rst                 : in  std_logic;
        MOSI                : in  std_logic;
        update              : out std_logic;
        buffer_data         : out unsigned(11 downto 0)
    );
end SPI_module;
```

Figure 33: Entity description of the SPI module.

This SPI module functions mainly as a shift register, appending data to the end of an intermediate register on each rising edge of the SPI clock and updating an output register as well as an update signal once 12 bits have been received. However, as seen in the C code for the DSP chip, the chip select signal is set low for each transmission and returns high afterwards. This SPI module also employs the chip select signal in the top level entity by accepting it as the reset signal (Fig. 34). The chip select signal is used in this way to ensure if the SPI clock signal were to experience noise and the bit counter within the SPI module were to desynchronize, the counter would reset on each data transmission, guaranteeing the effects of

data corruption in a transmission are isolated in that transmission. Also seen in the C code for data transmission, the reset signal is set after every track is transmitted as an extra layer of redundancy to ensure if data corruption occurs, it is isolated within the track currently being transmitted.

```
-- spi_module
SPI_MOD : entity work.SPI_module
    port map (
        SCK         => SCK          ,
        rst         => spi_rst      ,
        MOSI        => MOSI         ,
        update      => update_delay ,
        buffer_data => spidata          );

--added to fix spi counting error with 4th string
spi_rst <= rst or ss;
```

Figure 34: SPI module entity defined in the top level entity.

Data corruption checking is achieved in the top level entity. The DSP chip sends each transmission twice to allow the FPGA to check if both transmissions are equal, in turn confirming whether data corruption has occurred (Fig. 35). Since only one SPI module is used, the first send must be stored in a register to compare with the second send. In order to keep track if an incoming transmission is the first or second send, the process statement "SPI_COUNTER" updates signal "handshake_bit," which is used as an enable for the register that stores the first send, "hdata." Then, on each data transmission update, the processes "HANDSHAKE_CHECK" and "COMPARE_FLAG" check if the first send and second send are equal.

```
--spi counter
SPI_COUNTER: process(ss, rst)
begin
    if (rst = '1') then
        handshake_bit <= '0';
    elsif rising_edge(ss) then
        handshake_bit <= not handshake_bit;
    end if;
end process;
n_handshake_bit <= not handshake_bit;

-- spi handshake register
HANDSHAKE_REG: entity work.reg_more
    port map(
    clk    => update,
    rst    => rst,
    enable => n_handshake_bit,
    input  => spidata,
    output => hdata );
```

```
-- xor checker
-- only update on update delay so that it changes
-- after each burst, even if data don't change
HANDSHAKE_CHECK: process(update)
begin
    if ((hdata xor spidata) = "000000000000") then
        HC <= '1';
    else
        HC <= '0';
    end if;
end process;

-- comp_flag for DSP
COMPARE_FLAG: process(update,ss)
begin
    if rising_edge(update) then
        if ((hdata xor spidata) = "000000000000") then
            comp_flag <= '0';
        else
            comp_flag <= '1';
        end if;
    end if;

    if (ss = '1') then
        comp_flag <= '0';
    end if;
end process;
```

Figure 35: Processes and entities allowing to track transmission number and to check for data corruption.

Solenoid Modules:

Once the SPI module outputs the data received from DSP chip, the solenoid modules are ready to control the solenoids. The control over the solenoids is achieved through three main components: the data register, the PWM generator, and the demux entity.

The data register in Fig. 36 employs the HC signal from Fig. 35 as the clock, meaning the register only updates when it is confirmed there is no data corruption for new data; and the handshake_bit as the enable, meaning the contents of the register only update on the second data transmission (the solenoids should not update on the first send of each code). The PWM generator in Fig. 36 passes a 200 Hz PWM signal with a duty cycle of 80% through to activated solenoids. The input generic for the duty cycle requires 10x the duty cycle be entered so that the FPGA can achieve finer duty cycles without floating point operations. The demux component passes in the set of data from the aforementioned data register and outputs a PWM signal with the same width as the number of solenoids, similar to the output signal “solenoid_out” in Fig. 32 but for all 20 solenoids.

```
-- data register
DATA_REG: entity work.reg_more
    port map(
    clk    => HC,
    rst    => rst,
    enable => handshake_bit,
    input  => hdata,
    output => data);
```

```
-- pwm modules
-- pwm for the solenoids
PWM_MOD_SOL : entity work.pwm_mod
    generic map (
            PWM_FREQ => 200, --330 previously
            DUTY_CYCLE => 800 --10x the duty cycle
        )
    port map (
        clk    => clk,
        rst    => rst,
        pwm    => pwm_sol);
```

```
-- solenoid demux
DEMUX : entity work.demux
    port map (
        pwm     => pwm_sol,
        clk     => HC,
        rst     => rst,
        enable  => handshake_bit,
        data    => data,
        output  => solenoid_out);
```

Figure 36: Main entities required for the solenoids.

Originally, the demux component was entirely combinational but required a synchronous addition with changes in the signal processing in the DSP chip. To reduce spastic behavior in the ukulele, the DSP chip only records one code for every 4 iterations of the FFT, meaning if the FFT captured 4 different codes in a row, only the first code would be kept and the remaining would be replaced with the code for no note. For example, if the following codes were captured in consecutive order, “010,” “011,” “100,” “001,” “110,” “101,” the DSP chip would only keep the first in every sequence of 4 and replace the rest with the code for no note: “010,” “111,” “111,” “111,” “110,” “111.” This implementation, however, would be just as efficient as reducing the FFT rate to a fourth of its current rate, so the DSP chip code is implemented so that the counter for the 4 iterations resets every time no note is captured.

To reflect this adjustment of the DSP chip in the FPGA, the demux component only updates a data register for each string on the first of four iterations (Fig. 37). If the demux component receives a code where no solenoids are activated, however, the iteration count is reset. In other words, the iteration counter is reset each iteration until the demux component receives a code that activates a solenoid, after which the same solenoids stay activated for the next 3 codes received.

This synchronous addition to the demux component also allows the solenoid to be pressed down long enough for the servo motor to strum the string. Before this addition, the solenoids would press down and return almost immediately (FPGA receives new codes at a rate of 10Hz), and the servo motor could not rotate quick enough to strum the string while the solenoid is pressed against the string.

```
process(clk,rst)
    variable counter    : unsigned(3 downto 0) := (others => '0');
begin
    if rst = '1' then
        counter     := (others => '0');
        data1_reg   <= (others => '0');
    elsif rising_edge(clk) then

        if enable = '1' then

            counter := counter + 1;
            --on the first note...
            if (counter = 1) then

                --checking if data uses solenoids...
                if ((data1 = "111") or (data1 = "000") or (data1 = "001")) and
                   ((data2 = "111") or (data2 = "000") or (data2 = "001")) and
                   ((data3 = "111") or (data3 = "000") or (data3 = "001")) and
                   ((data4 = "111") or (data4 = "000") or (data4 = "001")) then
                    --resetting counter if not
                    counter := (others => '0');
                end if;
                --always update solenoids on first go
                data1_reg <= data1;
                data2_reg <= data2;
                data3_reg <= data3;
                data4_reg <= data4;

            elsif counter = 4 then
                --resetting counter on 4th iteration
                counter := (others => '0');
            end if;

        end if;

    end if;
end process;
```

Figure 37: Synchronous section of the demux component for solenoid control.

The combinational part of the demux component can be boiled down to a case statement or mux which decodes the code for each string into whichever solenoid needs to be activated (Fig. 38). Each string has its own case statement and the sensitivity list includes the PWM signal so that the outputs change according to the PWM signal:

```
process(pwm,data1_real,data2_real,data3_real,data4_real)
begin
    -- default output values
    output <= (others => '0');
    lastsol <= "000";
    -- mux for string 1
    case data1_real is
        when "111" => -- no note
            output(4 downto 0) <= (others => '0');
        when "001" => -- open string
            output(4 downto 0) <= (others => '0');
        when "010" => -- fret 1
            output(0) <= pwm;
        when "011" => -- fret 2
            output(1) <= pwm;
        when "100" => -- fret 3
            output(2) <= pwm;
        when "101" => -- fret 4
            output(3) <= pwm;
        when "110" => -- fret 5
            output(4) <= pwm;
        when others =>
            output(4 downto 0) <= (others => '0');
    end case;

    -- mux for string 2
    case data2_real is
```

Figure 38: Combinational section of the demux component for solenoid control.

Servo Modules:

For correct servo control, the servos need two separate PWM modules for each string and a servo module to decode the data received from the DSP chip. Each servo needs its own dedicated PWM generators because the positioning of each motor requires individual tuning (Fig. 39 shows how the duty cycle for side A of servos 1 and 2 differ by almost 1%) and this is not possible with two global PWM generators:

```
--pwm for servo 1, side a
PWM_MOD_A_S1 : entity work.pwm_mod
    generic map (
            PWM_FREQ => 50, --irl do 50
            DUTY_CYCLE => 76 --10x the duty cycle
        )
    port map (
        clk    => clk,
        rst    => rst,
        pwm    => pwm_a_s1);
```

```
--pwm for servo 1, side b
PWM_MOD_B_S1 : entity work.pwm_mod
    generic map (
            PWM_FREQ => 50, --irl do 50
            DUTY_CYCLE => 90 --10x duty cycle
        )
    port map (
        clk    => clk,
        rst    => rst,
        pwm    => pwm_b_s1);
```

```
--pwm for servo 2, side a
PWM_MOD_A_S2 : entity work.pwm_mod
    generic map (
            PWM_FREQ => 50, --irl do 50
            DUTY_CYCLE => 85 --10x the duty cycle
        )
    port map (
        clk    => clk,
        rst    => rst,
        pwm    => pwm_a_s2);
```

Figure 39: PWM generators for servo 1, sides A and B, and for servo 1, side A.

Each servo also has its own servo control module. Each component takes in two clock signals, one to update the servo's position once data has been verified not to have been corrupted and the other to serve as a delay clock (Fig. 40). The servo controller has 3 main sections: a synchronous register to track the position of the servo, a 0.15s delay, and a combinational case statement which alters the position of the servo based on the registers in the synchronous section.

```
-- servo controllers
-- quarter servo controller 4
SERV_CTRL_Q_1 : entity work.servo_ctrl_q
    port map (
        rst     => rst,
        sys_clk => clk,
        pwm_a   => pwm_a_s1,
        pwm_b   => pwm_b_s1,
        clk     => HC,
        enable  => handshake_bit,
        data    => spidata(11 downto 9),
        output  => servo_out(0)
    );
```

Figure 40: Entity for servo controller 1.

The synchronous section does not require the same iteration counter in the demux component because the servo position already stays constant until the servo control module receives data that indicates a note must be played. Once this data arrives, a signal “switch” is toggled.

```
-- register to track position of servo, triggered by the update from SPI module
process(clk, rst)
begin
    if rst = '1' then
        switch <= '0';
    elsif rising_edge(clk) then
        if enable = '1' then
            if data /= "111" then -- if not no note, strum
                if switch = '0' then
                    switch <= '1';
                else
                    switch <= '0';
                end if;
            end if;
        end if;
    end if;
end process;
```

Figure 41: Synchronous section of the component for servo control.

The delay between the synchronous and combinational sections of the code exists to prevent the servo from strumming before the solenoids fully fret down the string. The earlier tests for the UkeMaster 3000 revealed this behavior and caused muted strings—the addition of the delay removed the issue.

The combinational section of the servo control module refers to the “switch” value in the register to choose the PWM generator to output to the servo (Fig. 42). For example, if the servo control module receives the following sequence of codes: “110,” “111,” “010”, the servo will switch PWM generators on the first code, maintain the same position for the second code since “111” indicates no note and the string should not be strum, and then the servo will update its position for the third send. A more detailed simulation can be viewed in Fig. 32.

```
-- process to change position of servo
process(switch_reg, pwm_a, pwm_b)
begin
    case switch_reg is
        when '0' =>
            output <= pwm_a;
        when others =>
            output <= pwm_b;
    end case;
end process;
```

Figure 42: Combinational section of the component for servo control.

Timing:

While not included in the list of main sections required for the FPGA to operate, the accurate timing of the FPGA's components is essential to its correct operation. The attention to timing mainly concerns the "update_delay" signal output from the SPI module. Since the data register output from the SPI module changes on the same edge as the update signal it outputs, if other registers outside the SPI module update their data on the rising edge of the "update_delay" signal, they will update with the old data before the update. Therefore, most operations in the FPGA rely on the "update" signal, which is the "update_delay" signal with a delay of one cycle. This one-cycle delay allows data registers to update with the new data available in the system.

```
-- delay for update bit
DELAY : entity work.reg
    port map(
    clk    => clk    ,
    rst    => rst    ,
    input  => update_delay  ,
    output => update );
```

Figure 43: One cycle delays for update signal.

Obstacles Faced:

DSP Programming:

FFT Implementation

The FFT was hard to implement at first due to it being done on a separate controller. The CLA that the FFT is computed by executes C code like the main processor, but the debugger connected to the processor does not connect to the CLA. This made debugging the FFT when it was not working more difficult. Additionally, there were several issues in Code Composer Studio early on with the linker file as well as other files included in the project that took time to solve in order to get the FFT working properly.

LED Strip Implementation

While Using the Configurable Logic Block to send data to the LED strip was the simplest solution hardware wise, it was not simple to implement. Configuring the CLB took a lot of time researching in the manual and testing with example code from TI to understand how to program to it. Additionally, the SPI CLK and PWM signals had to be synchronized which was challenging because the PWM would always lead the SPI CLK and there was no way to shift this in the PWM configuration. To solve this, the PWM has a higher duty cycle than it needs so that the overlap between the SPI CLK and PWM signal is what it should be.

DSP PCB:

There were several errors made when designing the DSP PCB that would have to be corrected after the fact on the final board. First, the DSP chip is on a TI launchpad that is powered with 5 V, so the push buttons and ADC input from the mic had peak voltage levels of 5 V. This was a mistake however and the DSP chip itself runs off of 3.3 V, not 5 V. Additional resistors were added to the circuit for the push buttons to divide the voltage seen by the GPIO pins to 3.3 V instead of 5 V, and the power rails for the op amps in the Mic amplifier circuit were changed from 5 V to 3.3 V.

Additionally, some components were incorrectly mapped to power and reset pins, so traces had to be cut and those components had to be remapped to other GPIO pins.

UkeMaster 3000 Housing:

The Housing for the UkeMaster went through 2 iterations. The first was almost what we needed, but there were a few mistakes made in the dimensions for the wood cuts in the UI. Additionally, the slots that hold the solenoid holders were made slightly larger and fillets were added to the top of the slots to make the holders go in more easily.

Mounting the wooden pegs used to extend the solenoid plungers down to the Ukulele strings was also challenging. We used hot glue to stick them onto the plungers, but the pegs would often fall off due to how hard and fast the plungers moved up and down. To solve this we drilled holes into the pegs that could fit the plungers and added extra glue to the pegs. These two changes maximized the surface area that the glue contacted between the pegs and the plungers and the pegs now hold well.

Signal Processing:

The latter two processes within the “Signal Processing” section were a result of troubleshooting and tuning the UkeMaster 3000. The methods used in removing repeat codes and erratic behavior aided the UkeMaster 3000 in accurately playing back the recorded input audio and can be explored in greater detail in those respective sections of the report.

FPGA VHDL:

The main obstacles faced with the FPGA programming lie in timing issues as well as dealing with data corruption. The timing issues were mostly resolved by adding delays, whether it be for the update signals in the “Timing” section or with servo and solenoid timing in sections “Solenoid Modules” and “Servo Modules.” The major obstacle in dealing with data corruption resided in understanding what the issue was, why it was happening, and how to fix it. Once the issue was understood, the process of rewriting code for both the FPGA as well as the DSP chip to send each code twice and compare the output brought about only minor obstacles.

Solenoid Drivers:

Originally, all testing on the operation of the UkeMaster 3000 was done for 4 solenoids and 1 servo on the same string. After adding the remaining 16 solenoids and 3 servos, the system began experiencing unusual behavior, activating multiple solenoids and pulling a very large current when neither should be possible. This obstacle was understood after conducting an extensive and thorough series of tests, pinpointing the issue to the removal of power while the system was still running and solenoids remained active. The theory is that since the solenoid driver circuit basically consists of 20 large inductors, if the power is unexpectedly pulled while some of those solenoids are active, the circuit will be resistant to the change in current and cause unexpected behavior. Therefore, only removing power while the system is idle prevents the issue from repeating.

Bill of Materials:

Several of the materials used to construct the UkeMaster 3000 were lent to us or already in our possession from previous classes, however, the following list summarizes the approximate cost to build this project from scratch.

Component	Quantity	Price	Total Cost
PCBs	2	$23.48	$23.48
LED strip	1	$18.68	$18.68
LT1632 Op Amp	2	$10.12	$20.24
Solenoids	20	$5.50	$110.00
Servos – 4pk	1	$10.98	$10.98
1N4004 Diode	21	$0.10	$2.10
IRLZ44N MOSFET	21	$1.55	$32.55
Ukulele	1	$35.00	$35.00
DE-10 Lite - FPGA	1	$82.00	$82.00
F28379D LaunchPad – DSP Chip	1	$39.00	$39.00
NE555 Timer	1	$0.43	$0.43
UI Buttons	4	$4.98	$19.92
LCD Screen	1	$10.68	$10.68
Corner Brackets	2	$4.98	$9.96
Wooden Base and Supports	1	$11.98	$11.98
Microphone	1	$0.78	$0.78
			Total: $427.78

Division of Labor:

Nikodem Gazda	Ryan Simoneau
- FPGA VHDL	- FFT Implementation
- DSP Signal Processing	- DSP Signal Processing
- Solenoid/Servo Driver PCB	- User Interface/LED Strip
- Microphone Amplifier/Filter	- DSP Board PCB
- Physical Device Construction	- Buck Converter
	- Physical Device Construction
	- CAD Design

Project Schedule:

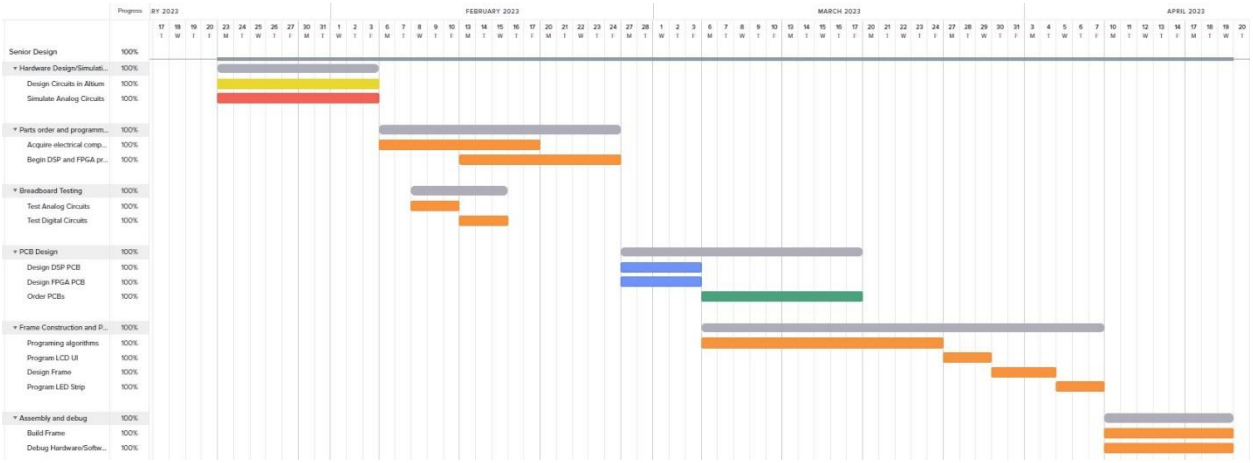


Figure 44: UkeMaster 3000 Design Schedule

Collaboration and Teamwork:

We worked well together as a team. We were able to brainstorm effectively with each other and come to the best solutions when it came to our biggest design decisions like the apparatus that frets the strings and the how the housing would be constructed. We planned and divided the project in the beginning and both of us were able to work independently when we needed to get our parts of the project done.