Piano Man
FPGA Piano-Playing Robot

Brendan Ashworth, Max Hardy,
& Anthony Nardomarino
12/11/2019

Table of Contents

1 Abstract

A good piano partner is often hard to come by, and with the help of a Nexys 4 DDR FPGA board, we can show the reigns of the keyboard to a robotic companion. Our goal was to build a sound processing system as well as a robotic mechanism, able to be actuated to reproduce the processed sounds on a keyboard. The keyboard is in an anchored location to eliminate the need for vision capabilities, while emphasizing the robot's ability to reproduce operator-played chords and melodies on the same keyboard as an operator. This project will showcase the ability to create a state machine with listening and playing capabilities, involving analyzing playable frequencies and creating a strategy for playing reproducible chords and melodies using controlled motorized fingers.

2 Design

2.0 Block Diagram

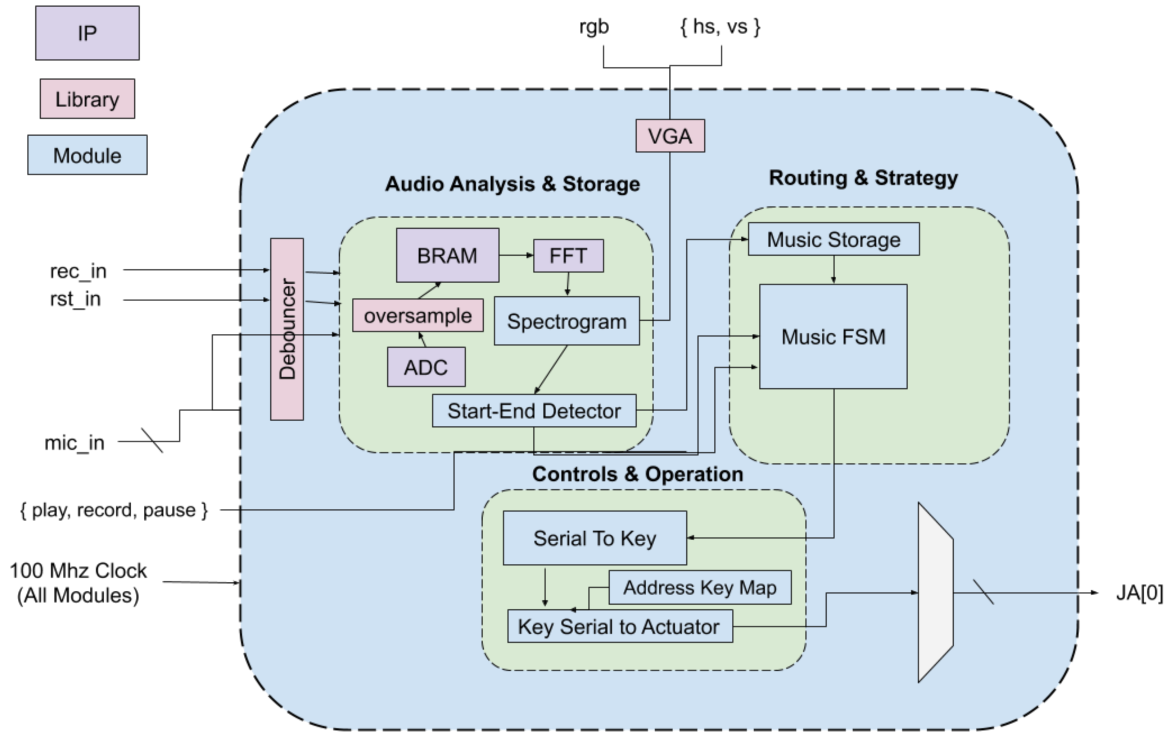

Figure 2.0.a: Piano-Playing Robot FPGA Block Diagram

As seen in Figure 2.0.a, the FPGA Piano-Playing Robot operates under the cooperation of several different modules responsible for different aspects of the system's logic. Debouncing mechanical inputs such as push-buttons is vital for the clean operation of the FSM (2.5) given what they represent.

As soon as the record button is pressed and held, a new state of operation is entered in which the feed from the microphone analog circuit is sampled and stored to BRAM, where windows of the continuous audio feed are independently sent through a Fast Fourier Transform (2.1) operation to later be stitched together in a new state (after detecting peaks with the Spectrogram (2.2)) and to be stored in Music Storage (2.4).

After this process, the Finite State Machine is able to control the release of this data into a feed that can be read according to the rhythm to which it was recorded (within the resolution of a predetermined "beat-clock") (2.5). In this state, the feed is sent on to the controls block in the form of a $12x30n$ bitmap, where n is the frequency of the chosen beat-clock. Each bit in the list of 12-bit values represents the activation of a specific note in the chromatic scale, to then be sent on to the controls block to readjust its current activation state.

The Synchronization & Timing block (2.6) then keeps track of the release of information by keeping its own internal clock consistent with the rate of the serial communication protocol of the ESP32, referencing each bit's relation to a specific motor address, held in the Address Key Map Lookup table, and sending that particular address if that particular motor's note was to be active in that state of operation.

Finally, the ESP32 motor driver manages the received serial information by counting a maximum of 12 bytes as input (according to the 12 possible addresses) before actuating the motors to which those address bytes correspond. During the playback state of the Finite State Machine, those notes are then sent on a loop at the positive edge of the "beat-clock" for the stored time that those notes are to be active.

2.1 Signal Processing Logic (Max)

The signal processing logic required for the project was extensively researched. While the fast Fourier transform was always the backbone of the approach, there were several variations on the downstream analysis of the spectral information which varied in accuracy and complexity.

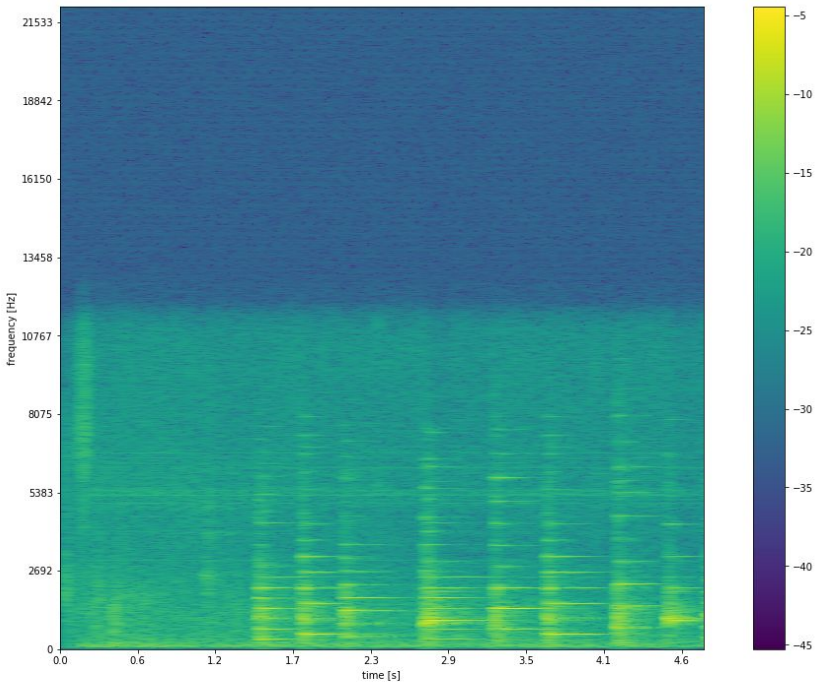

Figure 2.1.a: Generated Spectrogram Example

The figure above is a spectrogram generated during simulations performed in Python. The horizontal axis represents time, while the vertical shows frequency. It is generated by applying FFTs to 'windows' of audio input. This approach is powerful because it gains information in the frequency domain, while retaining information in the time domain - both of which are important in music. Various window sizes were tried, as increasing window size increases frequency resolution at the cost of speed.

Many strategies for note detection were attempted in Python Simulations. Initial approaches included functions which searched for power surges during note onsets, followed by various algorithms for selecting relevant notes at such onsets. Such approaches recorded all audio, performed all FFTs, found all onsets, and found all corresponding notes sequentially.

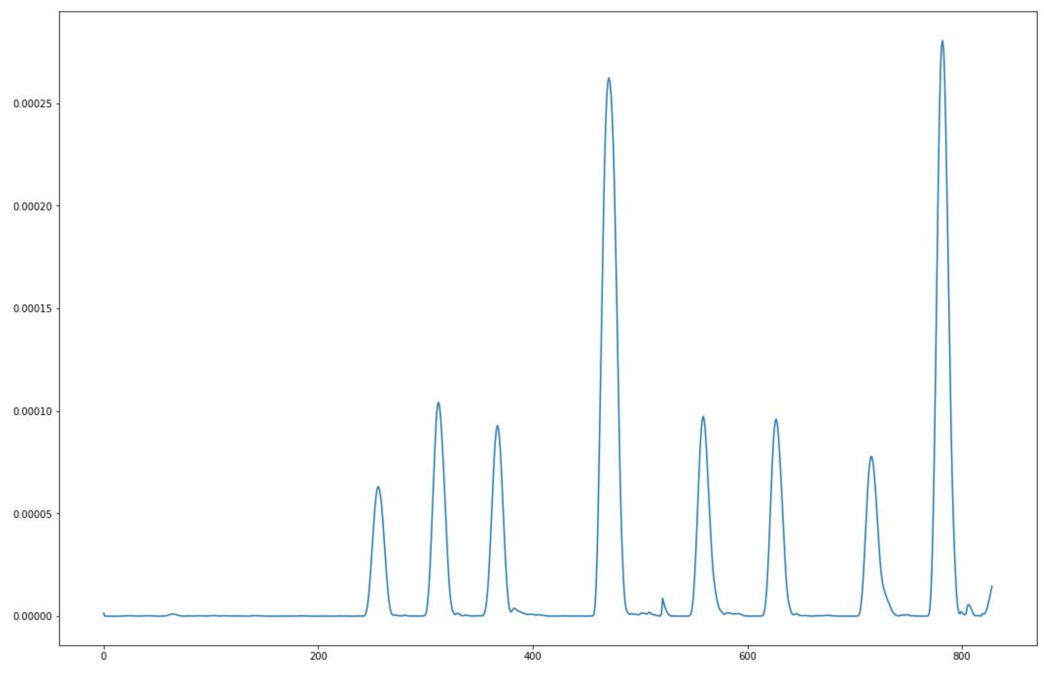

Figure 2.1.b: Onset Peak Detection Example

The above figure illustrates peaks which signify potential note onsets from the previous spectrogram. At such peaks, the relevant time step is recorded, then logic is needed to determine which note or notes were activated at that time step from the stored spectrogram. At this stage, several algorithms were attempted to determine relevant notes. This approach was highly accurate and capable of capturing nuances in music, but cumbersome to implement in Verilog. This approach was effective for slower processing rates and relied heavily on large storage space, which would not utilize the strengths of the FPGA - speed and not storage.

The FPGA was fast enough to produce spectral information in near real time, even with substantial window sizes. In other words, it was fast enough to process all audio as it arrived (not the case in a Python simulation), so there would be little need for large amounts of stored information to process retroactively. Because of this, a more basic approach was chosen in which a simple, customizable power threshold determined activated notes in real time. This was significantly simpler to implement in digital logic.

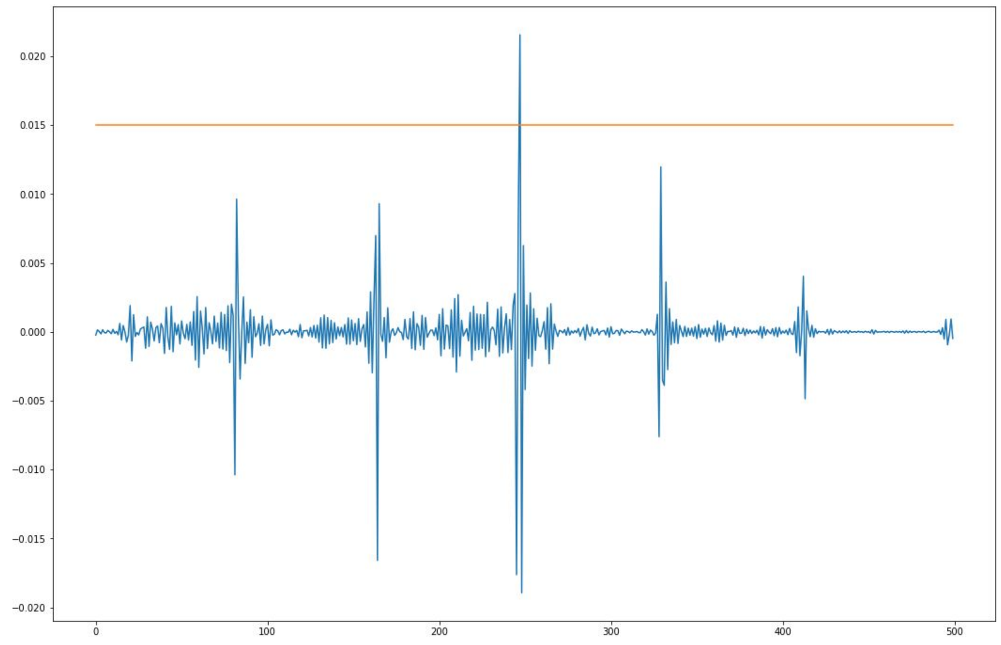

Figure 2.1.c: Frequency Prominence Thresholding

The above image is a simulation of what the FPGA would do, in which the horizontal axis represents frequency, and the vertical axis represents power. Essentially, the FPGA would take a real time FFT and search for any spectral bands that broke a power threshold (orange line). Because spectral power levels tend to oscillate over the course of a ringing note, a refractory period was built in that would prevent a note from activating many times over the course of a single ring. This approach would take advantage of the strengths of the FPGA and be relatively easy to implement.

2.2 Fast Fourier Transform (Max & Anthony)

The FFT was provided by the Verilog IP Core. The team integrated the core with the BRAM such that the FFT operated on audio samples stored in BRAM, and output the spectral information in a different BRAM. An FFT window size of 16384 was used for high resolution spectral information. Larger window sizes require longer computations, but provide higher resolution between notes - i.e. adjacent notes are separated by more bins in the spectral histogram. The output of the FFT was cut down substantially before being stored in the BRAM. Only the lowest 1024 bins were stored, as most of the higher frequency bins were not relevant to music or even in the range of human hearing. FFT window size and stored window size were chosen after repeated testing both in Python simulations, as well as in verilog.

2.3 Spectrogram (Brendan & Max)

The spectrogram displays the current Fourier Transform in the frequency domain on a VGA output monitor attached to the FPGA. The spectrogram also completes the role of fundamental frequency analysis, where it determines the activations of certain notes within in octave.

The VGA spectrogram shows the lowest 256 bins of an 8192-sample Fourier Transform, which has enough information to display the majority of frequencies present along the range of a piano (it loses the top few keys). Colored based on their power activations, with red showing many decibels and yellow showing background noise, one can dynamically see the frequency analysis of the incoming audio input on the VGA screen.

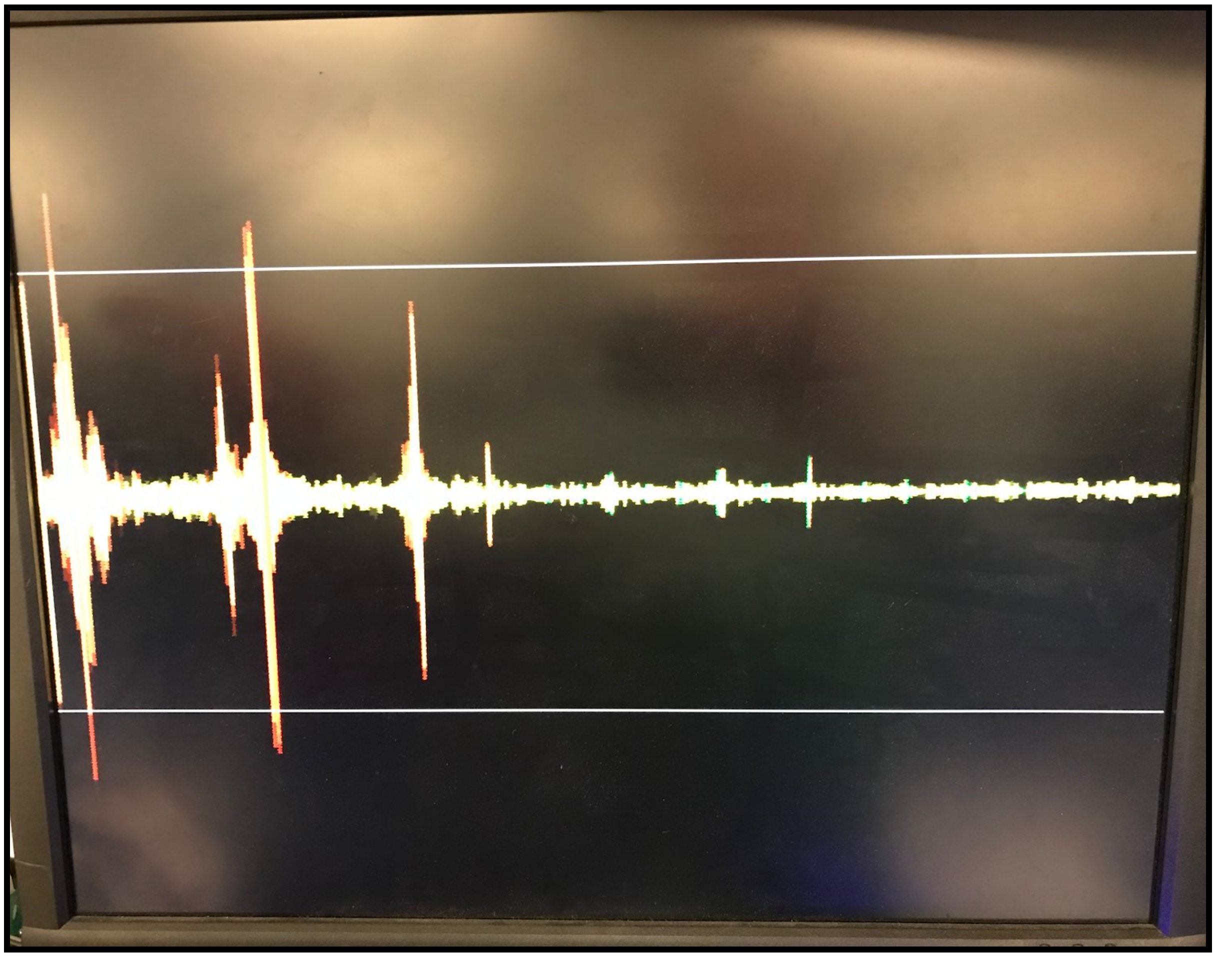

Figure 2.3.a: Implemented VGA Frequency Amplitude Map

On the screen, 12 bins are colored green. These bins are the frequencies of the fundamental notes, keys starting at A, A-sharp, B, ... ending at G and G-sharp. There are two horizontal lines that represent a configurable power threshold. When the green bin activations go above the threshold, they are categorized as ON. As long as they are above the threshold, they continue to be on. When the power goes below the threshold, they will continue to be on for a second, before turning off automatically. This allows both full, rich pianos that can generate enough power above the noise threshold to play well, while also smaller electronic keyboards that are harder to detect.

2.4 Start-End Detector (Brendan)

Storing all of the notes would be too logic-intensive to synthesize in look-up tables on an FPGA. To compress the notes, we take an effective discrete derivative of the note activations. When the beat clock fires, the Start-End Detector calculates the changes between the current note activations and the previous note activations. If these changes are not negligible, the Start-End Detector notes a change and clocks the delta between the current note and the previous note to the Music Storage module.

The Start-End Detector is the crucial step in compressing music before storing it on the FPGA. When it notes a change, it notes the current timer coming from the Finite State Machine and indexes the change based on that timer. This allows the Finite State Machine to find the relevant changes based on the current playback time as opposed to storing many irrelevant, non-changes in idle time.

2.5 Music Storage (Brendan)

The Music Storage module takes clocked input from the Start-End Detector and stores it in a lookup table based on the current count from the Music Finite State Machine. The Music Storage module can store a maximum of 1024 note changes, allowing for a maximum of about 4 minutes of playback on the double-speed FSM clock.

Upon reset (the center button), the Music Storage clears and allows one to re-record from the beginning without reprogramming the FPGA.

2.6 Music FSM (Brendan)

The Music Finite State Machine is the core module of the FPGA side of the Piano Playing Robot. The Music FSM implements a Mealy finite state machine that combines inputs from the buttons (center, left, right, and up) with the beat clock and various states that include PLAYING, PAUSED, RECORDING, ANALYZING, and IDLE.

From these states, it sends environment information including whether it's paused, playing, recording, analyzing, or idle, where each is a glitch-free bit window into the state value. Each state can only transition into a number of states based on the inputs, as is a requirement for Mealy FSMs.

The Finite State Machine handles the counter and length calculates for each recorded song. As the beat clock ticks, the counter increments, and when the FPGA transitions from the recording to the analyzing states, the length is set and the counter reset for playback.

Once recording is done, the FSM transitions to the paused state. Pressing the right button transitions it into the play state, while letting go of it pauses it in time. Pressing the play button again returns it to playing where you left off. When the song is done, it resets to the beginning in a loop. One cannot record over a song without resetting the FSM with the center button. Using the top button, one can fast forward through the song, through both playback and recording. This allows the user to have more fine-grained control over the beat clock frequency, allowing either longer songs to be recorded, or faster songs with better resolution.

2.7 Synchronization & Timing (Anthony)

As explained in Serial Communications (2.7), Synchronization & Timing is of utmost importance in the development of smooth and responsive controls from the FPGA to the ESP32 motor driver. Interfacing directly with the Finite State Machine (2.5), the serialToKey module receives its playback instructions in such a way to maintain rhythm to a resolution that is compatible with both the Nexys 4’s storage and the required time for servo motor actuation.

As the Fundamental Isolator (2.3) interacts with the Finite State Machine to store note changes, we compress the incoming data into a form that can capture both single note presses and chords (multiple notes being activated on the piano at the same time) with a multi-hot 12-bit value with each bit corresponding to a note on the chromatic scale (i.e. A, A#, B, C, C#, D, D#, E, F, F#, G, G#).

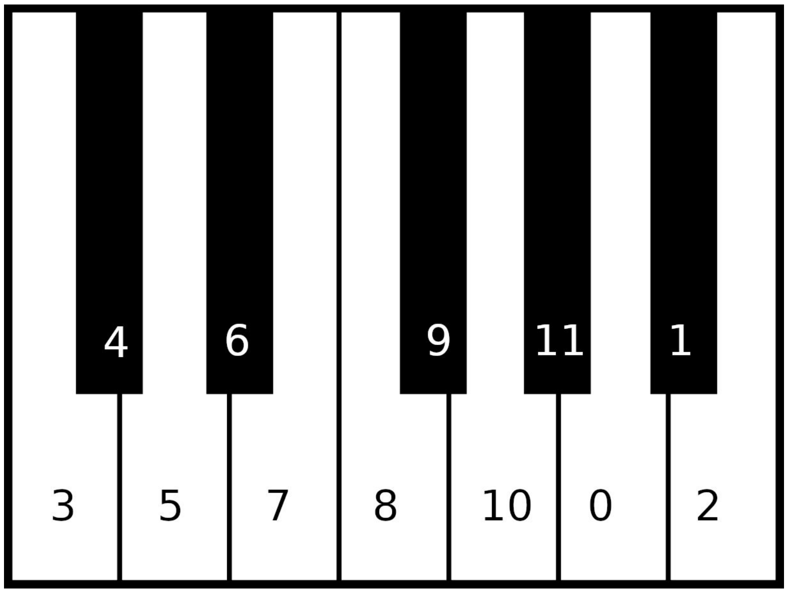

Figure 2.6.a: Bit Mapping of Key Values in the Chromatic Scale

From here serialToKey is able to iterate through each bit in the 12-bit state value to determine which addresses to send serially between each 115200hz cycle. This is done through an interface with the addressKeyMap lookup table that maps each note's activation state with an 8-bit motor address.

Because the serialToKey module's operations are done on a 100Mhz clock, we can take care of any processing between the receipt of the 12-bit state value and trigger of the serial_tx send in ample time. If a bit is high, then the address can be received and sent during the next 115200 baud cycle, or else the line will simply stay high, indicating to the ESP that no new serial stream is available, while it still counts that stream as a blank instruction and thus incrementing its internal counter to figure out if the instruction set is complete and ready for actuation.

2.8 Serial Communications (Anthony)

Serial communications is one of the most important structures to have solidified in the FPGA Piano-Playing Robot because if the notes calculated from the FFT and signal processing blocks can't reach the motors, then there is no bridge between the FPGA processing and the physical output of the robot. This module involves a sensitive translation between the instructions coming from the Finite State Machine

(2.5) and the Motor Controls (2.8), meaning that our compressed playback data needs to be dissected and translated into addresses that the ESP32 will find useful.

Because the ESP32 is capable of easily configurable UART 8N1 serial communications protocol, we could structure the serialToKey module in such a way to act as a median between a 12-bit multi-hot encoding scheme and the serial_tx module's translation into an 8-bit address. From the Finite State Machine during playback, states of the notes are received for serial processing in the form of a 12-bit sequence given for each time division (determined according to the beat-clock). For example, if the beat-clock is running at 2hz (resolution of playback is an update every 500ms, which was used in the demo), then 30 seconds of playback would be encoded in a 60x12 array of note states, corresponding to their states every half second.

When this 12-bit binary value is sent to the serialToKey module, the timing is handled to produce a 115200 baud, in which time the 12-bit stream can be iterated over to find out which key addresses should be sent through the serial_tx module. By determining the intended key state for a given value in time, a correspondence with the addressKeyMap lookup table is established to gather the address of that intended motor. Acting for ten cycles to complete the 8N1 stream, the serial_tx stream is given the trigger (dependent on a parallel 115200hz clock in serialToKey) to send the motor address byte until all 12 are sent and the ESP32 can actuate. It does this by setting its input "val" address to a shift register that can then shift in bits according to its own clock compatible with the baud rate, with a default high to provide the end bits, or blank instructions in the case that that particular motor is not to be actuated. However, there may have been a discrepancy in our implementation in how serial communications would work at when taking into considerations the implication of a beat clock, and how instructions are only sent at the rising edge of this clock. This bug caused a failure in simultaneous actuation of keys as well as producing a mechanical error in false instructions, causing certain keys to oscillate between the upward and downward positions.

2.9 Motor Controls (Max & Anthony)

The motor controls are managed through an ESP32 microcontroller acting as both a serial receiver for FPGA motor instructions and a motor driver for the Piano Robot's servo array hands. After the Nexys FPGA generates a bitmap of note activations (the time values for which a given note should be playing in order to replicate the input), that bitmap is sent serially to the ESP32 through a UART 8N1 protocol, controlling the operation of the motors.

On the ESP32 side, actuation of the motors in a stable domain in which chords as well as notes are possible (simultaneous actuation of multiple motors within one beat value). To ensure this, the ESP runs a loop in which it is constantly looking for Serial input, verified by the presence of a serial address being received by a Serial Read function. Because the FSM (2.5) sends a passive high signal over the serial line at 115200 baud in its “play” state, the ESP32 is able to receive these streams and indicate them as an idle stream.

Because of this protocol, the ESP32 is able to detect every 12 adjacent serial transmissions, actuating up to all 12 at the same time. The states of each finger is kept as either a 1 or a 0 (indicating activation and deactivation respectively), and after each serial stream, the old and new states are compared to determine whether a certain finger is required to change states. This ensures that a finger is only pressed when serial instructions are received to keep that finger in the downward position, otherwise it will return back to its raised position.

There are several instances in which the team had to test different methods of keeping finger states, different information being sent serially, and implementations of the motor mount’s interface with the physical keyboard. Firstly, the physical setup is comprised of the motor mount’s securing of 6 servo motors for each hand at about 11mm apart (the rough spacing between keys on the piano). This required precise calculation of angles of depression for actuated fingers as well as lengths of each finger for different key shapes (natural keys and sharps) and their optimal contact locations to reduce stress on the servo motor gears.

The process of measuring resting angles for the two states of the fingers came from trial and error of testing values to ensure that the motors would not stall and draw excess current from pressing the keys, but also sustaining a distance away from the keys in its idle position such that it can actuate within a very short period of time from when it is called to change states. The mount was then mounted to the keyboard with rubber bands to act as a failsafe to motor failure, such that slack can be produced by a sufficiently large angle of depression during the testing phase of the project.

Some of the problems in the design of this system that we would correct upon include the linear nature of servo actuation. For this project, it would be impractical to add series current sensors to detect stall current draw to stop motors from actuating too far given the time we allotted for the mechanical side; however, an alternative method to prevent overactuation would be to use take advantage of the internal Hall outputs of the MG90S servos to determine if the correct position has been reached by gauging the rotor's permanent magnets. If this position does not match a predetermined range of action, this servo motor can promptly be deactivated or moved back to its idle position. Furthermore, the hand case design with 6 fingers was not originally intended to be mounted to the piano statically, but instead attached to a rack and pinion rail to move around the piano freely to its next location. Of course, this implementation was abandoned for the purposes of this project due to time and material constraints, and our design would have to move forward with a mounting technique in which one hand was mounted above the keys, and one was mounted below, as seen in Figure 2.8.a.

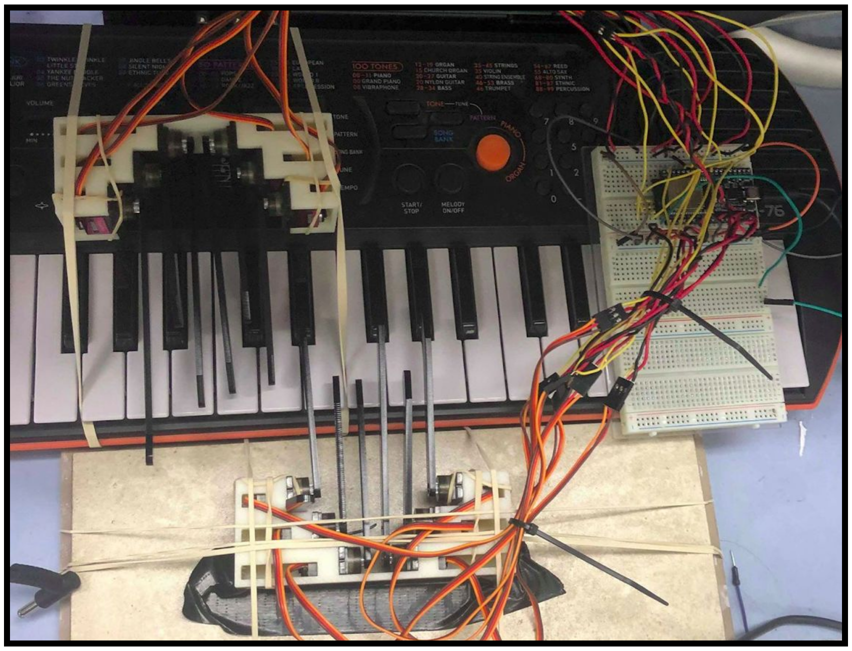

Figure 2.8.a: Mechanical Configuration of Motor Mount Hands

3 Implementation

In implementing the Piano-Playing Robot, our station consists of a playable keyboard for musical input into a microphone, the Finite State Machine controllable from the FPGA's onboard buttons and switches, and the ESP32 driver connected to the servo arrays mounted onto the robot's playback keyboard. Note identification can be controlled with manual switches to set an onset-threshold and tune the identification process to the environment in which the recording is taking place.

In activating the robot, one must press and hold the record button on the FPGA to begin taking in audio data, while there is a musical stimulus to the microphone. During our demonstration, this came from an independant keyboard playing into a microphone. After the preferred recording is complete, the user releases the record button and now has that audio clip saved and processed for playback.

By holding down the playback button, the FPGA will cycle through the divisions in time that it operates under (the beat clock), which can be changed for resolution in rhythm. The notes playing back in this playback mode will be manifested by both mounted LEDs on the FPGA to distinguish notes, as well as a finger from the robot physically pressing the keys of the piano on which it's mounted.

Playback options include fast-forward, pause, and reverse to control the way the robot plays the tune that it just heard. The fingers will move to press keys corresponding to which note should be playing at what time, and release when it is time for the note to end.

4 Review & Future Development

In the product's end phase of testing and demonstration, our system proved its ability to detect notes, determine their activity over a period of time of recording, store their states within a single octave, and enter a playback state in which notes could be correctly sent to the motor driver with accuracy in piano-key activation and in the time domain / rhythm. However, a misconception in the process of serial communication with the controls had prevented the perfected demonstration of the Piano-Playing Robot.

A notable bug in the implementation of the robot during demonstration was that although the correct motor addresses and times of activation were received for the note that was intended to be activated, there was an issue with a finger's ability to properly change states. This problem could be recognized by short taps of the keys as well as a constant oscillation between the upward and downward state transition of keys that were intended to be held. This can be explained by a dissonance in the effective note transfer from the FPGA to the ESP32 motor driver through the beat clock.

Essentially, between every beat clock, there is an idle high signal sent across the line, which had been considered to be a "nonactivity" signal, and that we could count off 12 of those signals and be ready to actuate, resetting the remaining states. However, in the time between each rising edge of the beat clock, those states reset, making it impossible to detect continuous sustains of notes or time-sensitive changes. In the future, we have learned to first seek out any discrepancies in the ways that different devices handle the same serial data, as there are many ways to get the same data across the same wire.

Furthermore, because a complex and robust mechanical structure of the project was not the primary goal, many design choices came from convenience and affordability. Most notably, this manifested in the rubber-band mount for a pseudo-proprioceptive stress on the servo motors, as well as the use of MG90S servo motors for high speed actuation. Problems that arose from these choices included fingers slipping from the keys, and small debugging errors in which the servos were trying to reach an angle far below the physical piano, drawing excess current and causing stress on its internal gears.

Although the rubber bands allowed for some stress relief in lifting the hands instead of allowing the motors to stall for too long, this is a very ineffective solution, and could be solved by implementing failsafe operations for motor actuation that would take intended and actual position into account, ensuring that the motors have safely reached their position in a feasible amount of time, otherwise overriding them and setting them to an idle state. This could be done by streaming the internal Hall feedback from the servo motors and controlling it according to that feed. However, this feed is not perfect considering the design of these servos to be cost effective.

Finally, an important consideration is the necessity of the ESP32 driver. The design choice was made due to its ease of control decisions and changes, while offering a quick method to debug possible note storage and propagation issues that would have been difficult to catch with an Internal Logic Analyzer or testbench. For this reason, the ESP32 microcontroller makes for an excellent prototyping logic device, but in the design of a more refined and final system, we would look into operating the Pulse Width Modulus states of the servo motors from the FPGA directly to decrease wiring required.

5 Parts/Hardware

5.1 Nexys 4 DDR FPGA

The Nexys 4 DDR is the FPGA board at the heart of the 6.111 Digital Systems Lab curriculum, armed with operational tools such as an ADC, VGA capabilities, a seven-segment display, and several switches to debug and operate the Piano Man FPGA Piano-Playing Robot. It is armed with ports for serial communication, specifically used to communicate with the ESP32 as a motor driver and I2C communication device to fully operate the mechanical aspects of the project, as well as being well equipped for temporary Block RAM storage, useful not only for servo motor strategy storage, but also for digital signal processing, at the heart of which lies the Discrete Fast Fourier Transform.

5.2 ESP32

The ESP32 Microcontroller was the obvious choice for an external Servo Motor driver, as it has onboard 16 PWM channels, perfect for operating PWM-controlled Servo Motors. It is capable of instantiating up to 3 hardware UART channels, programmable to many different subprotocols; the Piano Man simply communicates with the ESP32 through a singular 8N1 UART Serial port.

Serial communications to the ESP32 involved an I2C-like addressing function, in which the input byte was indicative of a “change” required for a specifically addressed motor. As 8N1 is the default supported protocol for UART serial communications, 12 motors were programmed to actuate when their 8-bit address is read, and return to an idle state hovering above the keyboard if their address is not read during a 12 byte stream. This simplifies the process of programming finger movements by giving the Nexys 4 FPGA board access to the motor array completely serially.

5.3 Sennheiser e935 Condenser Microphone

The Sennheiser e935 Condenser Microphone is a passive transducer manufactured for vocal and acoustic recording. It offers a clean means of recording audio in a specific direction with minimal outside noise, and provides a reliable feed up to a 24-bit depth. It is connected to the FPGA through the “hot” and ground lines from the microphone’s XLR port.

As the passive peak-to-peak voltage maximum for the purposes of this project is around 2.0mV, there was some modification to be done with the signal so that it provided a readable output to the FPGA's onboard ADC. This is discussed further in 5.5 Input Amplification Circuit.

5.4 MG90S Tower Pro Servo Motors

The MG90S Tower Pro Servo Motors are excellent prototyping motors for their high torque output for such a small size and mass. They were the perfect contenders for our Piano-Playing Robot for their affordability and versatile mounting capabilities. We used two arrays of 6 servo motors each between two 3D printed motor mount "hands" for operation of a keyboard from an ESP32 driver.

These motors operate within a 180° range by PWM signaling. Programmable within a specific range, a specific duty cycle will map the motor to move to a specific angle associated with it. This range will typically be read within an operation cycle of 50hz, where a 1-2ms duty cycle in a 20ms period will cover the entire 180° range. To decrease the time constraints of motor actuation, the ESP32 only drives each motor as a toggling function between two positions in a 90° range.

5.5 Input Amplification Circuit (Max & Anthony)

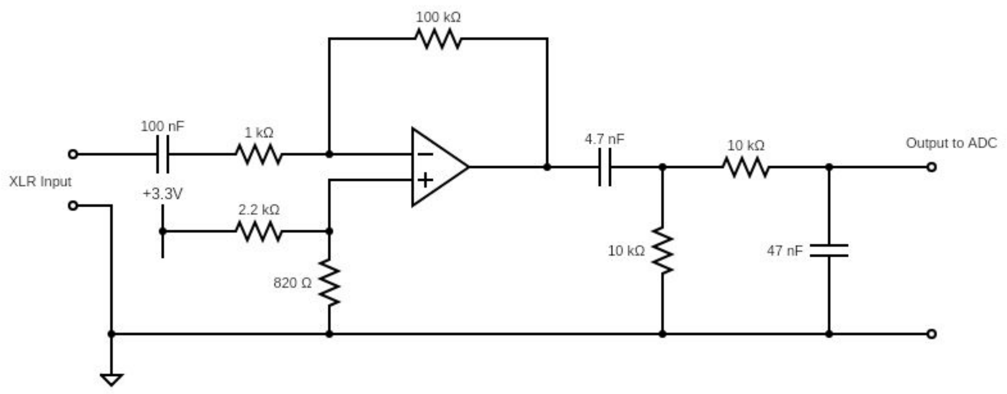

Figure 5.5.a: Input Analog Amplification Circuit

To allow the Microphone we used to be readable by the Nexys 4 DDR on-board ADC, it was important that we did some analog signal processing to not only amplify the incoming hot signal, but to also center it around 0.5V so it may be digitized.

This begins with a capacitive coupler into a LT1632 operational amplifier circuit, offset on the positive port with a rough functional voltage divider to get a 0.5V offset from a 3.3V input from the FPGA. From here, the roughly 2.0mV Pk-Pk input voltage is effectively phase shifted by 180° and amplified to get a 1.0V Pk-Pk amplitude with minimal clipping out of the op-amp. From here, the signal passes through a band-pass filter, allowing frequencies within the frequency response range of the Sennheiser e935 to pass through at full amplitude, while reducing the intensity of frequencies outside of that range. The result is passed on to the Nexys 4 DDR ADC for digitization.

5.6 3D Printed PLA Motor Mount "Hand"

5.6.1 Motor Mount

The Motor Mount "hands" were 3D printed in PLA plastic with an 11mm separation between the servo actuators, measured to a standard MIDI keyboard. A small window behind each motor allows for ease of wiring. Printed in the Cypress Engineering Design Studio.

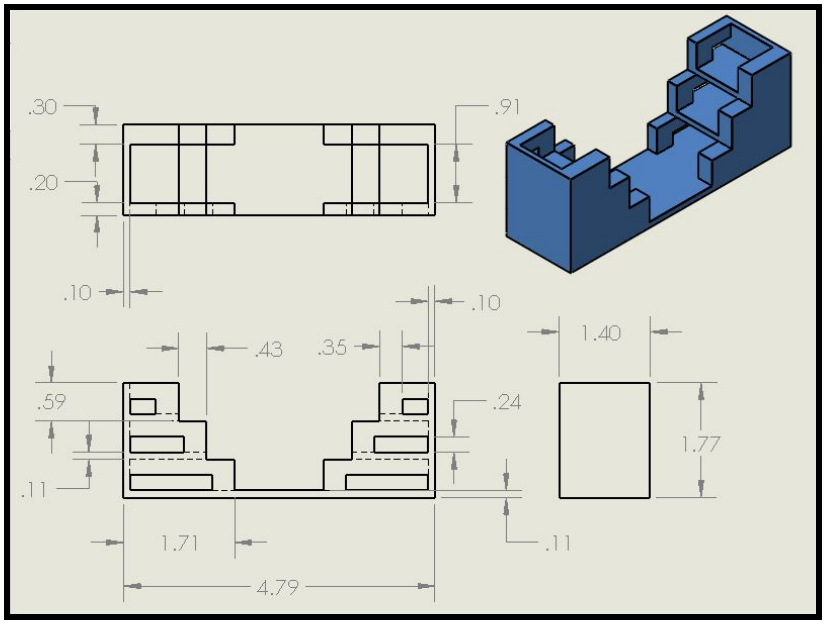

Figure 5.6.1.a: Motor Mount Hand Dimensions (in.)

5.6.2 Acrylic Fingers

Twelve laser-cut black acrylic fingers were used to press the piano keys in our system, attached to the servo arrays with a dowel-fit insertion, allowing them to sustain pressure without slip. Various sizes were used (ranging from 6.5cm to 12cm) in order to reach each key in an octave and reduce the total torque requirement of its attached servo motor. Laser cut in the Cypress Engineering Design Studio.

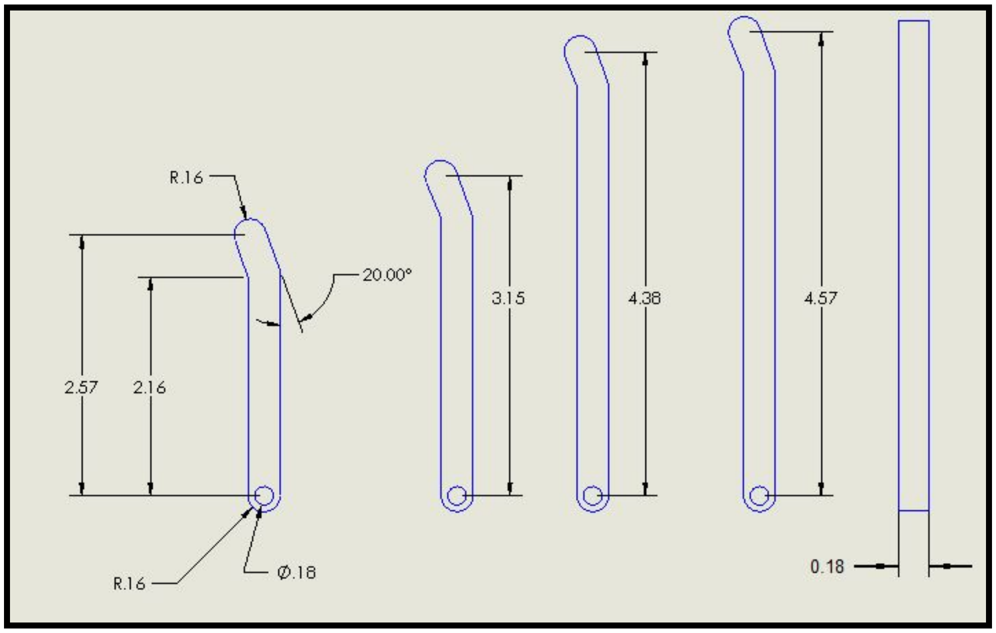

Figure 5.6.2: Acrylic Laser-Cut Finger Dimensions (in.)

6 Appendix

6.1 Associated FPGA SystemVerilog

6.1.1 fft.sv (Max, Anthony, Brendan)

```
// The main module for the Piano Playing Robot
// built on an FPGA.
// Team: Brendan Ashworth, Max Hardy, Anthony Nardomarino
// December 9 2019
module piano_playing_robot (
        CLK100MHZ,
        // ADC and VGA.
        VGA_R, VGA_B, VGA_G,
        VGA_HS, VGA_VS,
        AD3P, AD3N,
        // Inputs
        SW, BTNC, BTNU, BTNL, BTNR, BTND,
        // 8-segment display
        SEG, AN,
        // Lights
        LED16_B, LED16_G, LED16_R,
        LED17_B, LED17_G, LED17_R,
        LED,
        // Outputs to microcontroller
        JA
        );

        // inputs
        input logic CLK100MHZ;
        input [15:0] SW;
        input logic BTNC;
        input logic BTNU;
        input logic BTNL;
        input logic BTNR;
        input logic BTND;
        input logic AD3P;
        input logic AD3N;

        // outputs
        output logic [3:0] VGA_R;
        output logic [3:0] VGA_B;
        output logic [3:0] VGA_G;
        output logic VGA_HS;
        output logic VGA_VS;

        output logic LED16_B, LED16_G, LED16_R;
        output logic LED17_B, LED17_G, LED17_R;
        output [15:0] LED;
        output [7:0] SEG;
        output [7:0] AN;
        output [7:0] JA;

        // Split the 100mhz clock into lower and higher frequencies.
        // ADC takes 104mhz, VGA takes 65mhz.
        logic clk_104mhz, clk_65mhz;
        clk_wiz_0 clockgen(
        .clk_in1(CLK100MHZ),
        .clk_out1(clk_104mhz),
        .clk_out2(clk_65mhz));

        logic hsync, vsync, blank;
        logic [10:0] hcount;
        logic [9:0] vcount;
        xvga xvga1(
        .vclock(clk_65mhz),
        .hcount(hcount),
        .vcount(vcount),
        .vsync(vsync),
        .hsync(hsync),
        .blank(blank));

        logic BTNC_clean, BTNU_clean, BTND_clean, BTNL_clean, BTNR_clean;
        debounce #(.COUNT(5)) db0 (
        .clk(clk_104mhz),
        .reset(1'b0),
        .noisy({BTNC, BTNU, BTND, BTNL, BTNR}),
        .clean({BTNC_clean, BTNU_clean, BTND_clean, BTNL_clean, BTNR_clean}));

        // Full system reset.
        logic system_reset;
        assign system_reset = BTNC_clean;

        logic [15:0] adc_sample;
        logic eoc;
        // Sample audio input from the ADC.
        // Taken with inspiration from the nexys_fft_ddr demo.
        xadc_demo xadc_demo (
        .dclk_in(clk_104mhz),  // Master clock for DRP and XADC.
        .di_in(0),            // DRP input info (0 becuase we don't need to write)
        .daddr_in(6'h13),     // The DRP register address for the third analog input register
        .den_in(1),           // DRP enable line high (we want to read)
        .dwe_in(0),           // DRP write enable low (never write)
        .drdy_out(),          // DRP ready signal (unused)
        .do_out(adc_sample),   // DRP output from register (the ADC data)
        .reset_in(system_reset),
        .vp_in(0),            // dedicated/built in analog channel on bank 0
        .vn_in(0),            // can't use this analog channel b/c of nexys 4 setup
        .vauxp3(AD3P),        // The third analog auxiliary input channel
        .vauxn3(AD3N),        // Choose this one b/c it's on JXADC header 1
        .channel_out(),       // Not useful in sngle channel mode
        .eoc_out(eoc),        // Pulses high on end of ADC conversion
        .alarm_out(),         // Not useful
        .eos_out(),           // End of sequence pulse, not useful
        .busy_out()           // High when conversion is in progress. unused.
        );

        // Increase sampling fidelity by oversampling 16x.
```

```
	// Taken from the FFT demo.
	logic [13:0] osample16;
	logic done_osample16;
	oversample16 osamp16_1 (
	.clk(clk_104mhz),
	.sample(adc_sample[15:4]),
	.eoc(eoc),
	.oversample(osample16),
	.done(done_osample16));

	// This BRAM stores the FFT frame that was last generated.
	// This is before we process via FFT and then take the usable part of the spectrogram.
	logic [13:0] frame_head = 0; // Frame head - a pointer to the write point, works as
circular buffer
	logic [13:0] frame_addr;     // Frame address - The read address, controlled by
bram_to_fft
	logic [15:0] frame_data;     // Frame data - The read data, input into bram_to_fft
	bram_frame bram1 (
	.clka(clk_104mhz),
	.wea(done_osample16),
	.addra(frame_head),
	.dina({osample16, 2'b0}),
	.clkb(clk_104mhz),
	.addrb(frame_addr),
	.doutb(frame_data));

	// On every sample, increase the frame head pointer.
	// It'll overflow back when we take enough samples.
	always @(posedge clk_104mhz) if (done_osample16) frame_head <= frame_head + 1;

	// The FFT can be taken much faster than we take it, but because we only care about
	// visible and human readable frequencies, let's use one thats more reasonable.
	// The VGA vsync happens about every 60hz so we can use that.
	// This logic converts it to a single clock pulse.
	logic vsync_104mhz, vsync_104mhz_pulse;
	synchronize vsync_synchronize0(
	.clk(clk_104mhz),
	.in(vsync),
	.out(vsync_104mhz));

	level_to_pulse vsync_ltp0(
	.clk(clk_104mhz),
	.level(~vsync_104mhz),
	.pulse(vsync_104mhz_pulse));

	// Read frames from the BRAM and pipe it to the FFT.
	logic last_missing; // All these are control lines to the FFT block design
	logic [31:0] frame_tdata;
	logic frame_tlast, frame_tready, frame_tvalid;
	bram_to_fft bram_to_fft_0(
	.clk(clk_104mhz),
	.head(frame_head),
	.addr(frame_addr),
	.data(frame_data),
	.start(vsync_104mhz_pulse),
	.last_missing(last_missing),
	.frame_tdata(frame_tdata),
.frame_tlast(frame_tlast),
.frame_tready(frame_tready),
.frame_tvalid(frame_tvalid)
);

// We take an FFT with 16384 samples each of 16 bit depth. This logic
// sends it to the FFT IP to get the magnitudes of each frequency.
logic [23:0] magnitude_tdata;
logic [13:0] magnitude_tuser;
logic magnitude_tlast, magnitude_tvalid;
fft_mag fft_mag_i(
.clk(clk_104mhz),
.event_tlast_missing(last_missing),
.frame_tdata(frame_tdata),
.frame_tlast(frame_tlast),
.frame_tready(frame_tready),
.frame_tvalid(frame_tvalid),
// Scaling isn't necessary.
.scaling(12'b0000_0000_0000),
.magnitude_tdata(magnitude_tdata),
.magnitude_tlast(magnitude_tlast),
.magnitude_tuser(magnitude_tuser),
.magnitude_tvalid(magnitude_tvalid));

// We use a huge number of samples into our FFT (~16k), but we only care
// about the lowest of bins. This is because they contain the fundamental
// frequencies of piano notes spaced far enough apart to be distinguishable.
// This only store to BRAM when it's the lowest eighth.
logic fft_is_lowest_chunk;
assign fft_is_lowest_chunk = ~|magnitude_tuser[13:10];
// Store a single window of FFT in BRAM, with 1024 bins x 16 bit depth.
// log2(1024) => 10 bit address space
logic [9:0] fft_bin_index;
// 16 bit depth describes the magnitude of the frequency in that bin
logic [15:0] fft_magnitude;
bram_fft fft_windows (
.clka(clk_104mhz),
.clkb(clk_104mhz),
.wea(fft_is_lowest_chunk & magnitude_tvalid),
// Writing port.
.addra(magnitude_tuser[9:0]),
.dina(magnitude_tdata[15:0]),
// Reading port.
.addrb(fft_bin_index),
.doutb(fft_magnitude)
);

// These are the currently activated notes, updated every 60hz.
logic [11:0] notes;

// We allow for a range of thresholds for the audio analysis because
// of the wide variability in piano volume and outside noise. Take
// it from the switches.
logic [15:0] threshold;
assign threshold = SW[15:0];

// The spectrogram both shows the spectrogram on the
// VGA port and also isolates fundamental notes from it
// to pass to the finite state machine and start end detector.
fft_spectrogram spectrogram(
.vclk_in(clk_65mhz),
.hcount(hcount), .vcount(vcount), .blank(blank),
.hsync(hsync), .vsync(vsync),
.rgb({VGA_R, VGA_G, VGA_B}),
.fft_bin_index(fft_bin_index),
.fft_magnitude(fft_magnitude),
.hsync_out(VGA_HS),
.vsync_out(VGA_VS),
.notes(notes),
.threshold(threshold)
);

// Beat clock. How quickly the notes can change.
logic music_clk;

// Assign RGB LEDs
assign {LED16_R, LED16_G, LED16_B} = {2'b00, music_clk};

// The number of note changes we need to uniquely identify
// to play the entirety of the song.
parameter NUM_CHANGES = 1024;

logic is_recording;
logic fast_forward;
assign fast_forward = BTNU_clean;

mega_clk_div mega_clk_div0(
.fast_forward(fast_forward),
.clk_in(clk_65mhz),
.clk_out(music_clk));

logic backwards;
assign backwards = BTND_clean;

// interop between the start_end_detector
// and the music_storage.
logic [11:0][1:0] note_change;
logic [$clog2(NUM_CHANGES)-1:0] note_change_index;
logic note_clk;

// Playback on the LEDs.
start_end_detector detector0(
.rst_in(system_reset),
.activated_notes_in(notes),
.note_clk_in(music_clk & is_recording),
.note_change_out(note_change),
.note_change_index_out(note_change_index),
.note_change_clk_out(note_clk)
);

// Counter is the time, seq_ptr is the index.
logic [$clog2(NUM_CHANGES)-1:0] counter;
logic [$clog2(NUM_CHANGES)-1:0] ptr;
```

```
display_8hex display(
.clk(clk_65mhz),
.data({6'b0, ptr[9:0], 6'b0, counter[9:0]}),
.seg(SEG[6:0]),
.strobe(AN));
assign SEG[7] = 1;

// Assign RGB LEDs

// Print debugging states to the LEDs.
logic is_playing;
logic paused_out;
assign {LED17_R, LED17_G, LED17_B} = {is_recording, is_playing, paused_out};

// For displaying the current state on the FPGA.
logic [3:0] state;

fsm fsm0(
.clk_in(music_clk),
.rst_in(system_reset),
.fft_done(1'b1),
.state_out(state),
.counter(counter),
.recording_in(BTNL_clean), .playing_in(BTNR_clean),
.playing_out(is_playing), .recording_out(is_recording), .paused_out(paused_out));

logic [NUM_CHANGES-1:0][$clog2(NUM_CHANGES)-1:0] notes_indexes;
logic [NUM_CHANGES-1:0][11:0] notes_stored;
logic [NUM_CHANGES-1:0][11:0] notes_on;

logic note_clk_with_reset;
assign note_clk_with_reset = note_clk | system_reset;

music_storage music_storage0(
.rst_in(system_reset),
.notes_on(notes_on),
.note_change_in(note_change),
.note_change_index_in(note_change_index),
.note_change_clk_in(note_clk_with_reset),
.notes_stored(notes_stored),
.notes_indexes(notes_indexes));

logic [15:0] led_proxy;
assign LED[11:0] = led_proxy[11:0];
assign LED[15:12] = state[3:0];

logic fft_clk;
assign fft_clk = music_clk | system_reset;

always_ff @(posedge fft_clk) begin
if (system_reset) begin
       ptr <= 0;
       led_proxy[11:0] <= 12'b0;
// If recording.
end else if (is_recording) begin
       led_proxy[11:0] <= notes[11:0];
// If playing.
```

```
        end else if (is_playing) begin
                // Play out the notes onto the LEDs.
                led_proxy[11:0] <= notes_on[ptr];

                // allow for playing backwards
                if (backwards) begin
                // backwards lookup
                // Check the next pointer.
                if ((notes_indexes[ptr - 1] >= counter) & (notes_indexes[ptr - 1] > 0))
                        ptr <= ptr - 1;
//              else if (counter < ptr)
//                      ptr <= 0;
                end else begin
                // regular order
                // Check the next pointer.
                if ((notes_indexes[ptr + 1] <= counter) & (notes_indexes[ptr + 1] > 0))
                        ptr <= ptr + 1;
                else if (counter < ptr)
                        ptr <= 0;
                end
        // If paused.
        end else if (paused_out) begin
                // Keep the notes.
                led_proxy[11:0] <= notes_on[ptr][11:0];
        end else begin
                led_proxy[11:0] <= 12'b0;
        end
        end

        assign JA[7:1] = 7'b0;

//      logic [11:0] interest_note;
//      assign interest_note = notes_stored[ptr];

        logic data_output_pin;
        assign JA[0] = ~(~data_output_pin & is_playing);

        serialToKey hands (.clk_100mhz(CLK100MHZ), .rst(system_reset),
.data_out(data_output_pin),
                .beat_clk(fft_clk), .pb(is_playing), .interest_note(notes_on[ptr]));

endmodule
```

6.1.2 fft_spectrogram.sv (Brendan)

```
// Generates a visual spectrogram that contains
// VGA-generating display logic. This augments
// the audio output to provide another way to
// look at what the robot is doing.
// This also contains the fundamental note isolator/
// note recognition module. I merged the two to
// simplify note identification.
// Author: Brendan Ashworth
module fft_spectrogram(
        vclk_in,

        // VGA output
        hcount, vcount, blank,
        hsync, vsync,
        rgb,
        hsync_out, vsync_out,

        // note identification
        fft_bin_index,
        fft_magnitude,
        notes,
        threshold
        );

        // Fixed screen parameters.
        parameter SCREEN_HEIGHT = 767;
        parameter SCREEN_HEIGHT_HALF = 383;
        parameter SCREEN_WIDTH = 1023;

        input logic vclk_in;

        input [10:0] hcount;
        input [9:0] vcount;

        input logic blank;
        input logic hsync;
        input logic vsync;

        output logic [11:0] rgb;

        output logic hsync_out;
        output logic vsync_out;

        input logic [15:0] threshold;

        output [9:0] fft_bin_index;
        input [15:0] fft_magnitude;

        // old
        logic [9:0] hheight;
        logic [9:0] vheight;
        logic [1:0] intensity;

        parameter [27:0] NOTE_ACTIVATION = 28'd33_500_000;

        // COLORS is the color spectrum that we use to generate the spectrogram.
        // This makes it visually pleasing but not computationally or space
        // intensive.
        parameter [11:0] COLOR_RED = 12'hD11;
        parameter [11:0] COLOR_ORANGE = 12'hF84;
        parameter [11:0] COLOR_YELLOW = 12'hFD0;
        parameter [11:0] COLOR_BLACK = 12'h000;

        logic [27:0] A = 0;
        logic [27:0] As = 0;
        logic [27:0] B = 0;
        logic [27:0] C = 0;
        logic [27:0] Cs = 0;
        logic [27:0] D = 0;
        logic [27:0] Ds = 0;
        logic [27:0] E = 0;
        logic [27:0] F = 0;
        logic [27:0] Fs = 0;
        logic [27:0] G = 0;
        logic [27:0] Gs = 0;

        output logic [11:0] notes;
        assign notes = {A > 0, As > 0, B > 0, C > 0, Cs > 0, D > 0, Ds > 0, E > 0, F > 0, Fs >
0, G > 0, Gs > 0};
```

```
        // cooldown of 9.75*10^7

        always_ff @(posedge vclk_in) begin
        // We pipeline to allow for some computation, so
        // delay all logics equally.
        hheight <= fft_magnitude >> 7;
        vheight <= SCREEN_HEIGHT - vcount;
        // Intensity depends on the FFT magnitude.
        intensity <= {hheight > 210, hheight > 20};
        {hsync_out, vsync_out} <= {hsync, vsync};

        rgb <= blank ? COLOR_BLACK :
                (vheight == SCREEN_HEIGHT_HALF + threshold) ? 12'hFFF :
                (vheight == SCREEN_HEIGHT_HALF - threshold) ? 12'hFFF :
                // First, filter out what shouldn't be colored.
                // Center the spectrogram on half screen height.
                (vheight > (SCREEN_HEIGHT_HALF + hheight)) ? COLOR_BLACK :
                (vheight < (SCREEN_HEIGHT_HALF - hheight)) ? COLOR_BLACK :
                // Now do it based on intensity.
                // Color our fundamental frequencies specially.
                ((fft_bin_index == 210) | (fft_bin_index == 187) | (fft_bin_index == 166) |
(fft_bin_index == 147) | (fft_bin_index == 131) | (fft_bin_index == 117)) ? 12'h0F0 :
                ((fft_bin_index == 222) | (fft_bin_index == 197) | (fft_bin_index == 175) |
(fft_bin_index == 155) | (fft_bin_index == 139) | (fft_bin_index == 125)) ? 12'h0F0 :
                // Otherwise, just color according to how strong it is.
                (intensity[1]) ? COLOR_RED :
                (intensity[0]) ? COLOR_ORANGE :
                COLOR_YELLOW;

        // If a note is on, we "empty the bucket" progressively,
        // until the note turns off.
        if (A > 0)
                A <= A - 1;
        if (As > 0)
                As <= As - 1;
        if (B > 0)
                B <= B - 1;
        if (C > 0)
                C <= C - 1;
        if (Cs > 0)
                Cs <= Cs - 1;
        if (D > 0)
                D <= D - 1;
        if (Ds > 0)
                Ds <= Ds - 1;
        if (E > 0)
                E <= E - 1;
        if (F > 0)
                F <= F - 1;
        if (Fs > 0)
                Fs <= Fs - 1;
        if (G > 0)
                G <= G - 1;
        if (Gs > 0)
                Gs <= Gs - 1;

        // If a note is above our threshold, we reset
        // its activation, "filling the bucket".
        if (fft_bin_index == 117 & A == 0)
                A <= (hheight > threshold) ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 125 & As == 0)
                As <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 131 & B == 0)
                B <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 139 & C == 0)
```

```
                C <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 147 & Cs == 0)
                Cs <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 155 & D == 0)
                D <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 166 & Ds == 0)
                Ds <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 175 & E == 0)
                E <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 187 & F == 0)
                F <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 197 & Fs == 0)
                Fs <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 210 & G == 0)
                G <= hheight > threshold ? NOTE_ACTIVATION : 0;
        if (fft_bin_index == 222 & Gs == 0)
                Gs <= hheight > threshold ? NOTE_ACTIVATION : 0;
        end

        // We fit 256 bins on the screen. This makes each bin
        // wide enough to see, but small enough that we can fit
        // our octave of interest on the screen.
        assign fft_bin_index = hcount[9:0] >> 2;

endmodule
```

6.1.3 freqKeyMap.sv (Anthony)

```
///////////////////////////////////////////////////////////////////
//
// freqKeyMap.sv
// Anthony Nardomarino
// FPGA Piano Playing Robot
// MIT 6.111 Digital Systems Lab
// 11-15-2019
//
///////////////////////////////////////////////////////////////////

module addressKeyMap(
        countIn, addrOut
);

        input   logic   [3:0] countIn;
        output  logic   [7:0] addrOut;

        //              i2c Address Key:
        //
        // 0100_0000 - A      0101_0000 - C#   0100_1000 - F
        // 0010_0000 - A#     0011_0000 - D  0010_1000 - F#
        // 0110_0000 - B      0111_0000 - D#   0110_1000 - G
        // 0001_0000 - C      0000_1000 - E  0001_1000 - G#

        always_comb begin
        case(countIn)
        4'd0:           addrOut = 8'b0100_0000;
        4'd1:           addrOut = 8'b0010_0000;
        4'd2:           addrOut = 8'b0110_0000;
        4'd3:           addrOut = 8'b0001_0000;
        4'd4:          addrOut = 8'b0101_0000;
        4'd5:          addrOut = 8'b0011_0000;
        4'd6:          addrOut = 8'b0111_0000;
        4'd7:          addrOut = 8'b0000_1000;
        4'd8:          addrOut = 8'b0100_1000;
        4'd9:          addrOut = 8'b0010_1000;
        4'd10:         addrOut = 8'b0110_1000;
        4'd11:         addrOut = 8'b0001_1000;
        default:       addrOut = 8'b1111_1111;
        endcase
        end
endmodule // freqKeyMap
```

6.1.4 mega_clk_div.sv (Brendan)

```
// The beat clock generator.
// Allows for fast-forward with the fast_forward
// input, taken from button up.
// Author: Brendan Ashworth
module mega_clk_div(
        fast_forward,
        clk_in,
        clk_out);

        logic [23:0] mid = 0;

        input logic fast_forward;

        input logic clk_in;
        output logic clk_out;

        // mid[22] will be twice the speed
        assign clk_out = fast_forward ? mid[22] : mid[23];

        always_ff @(posedge clk_in) begin
        mid <= mid + 1;
        end

endmodule
```

6.1.5 music_storage.sv (Brendan)

```
// The music storage module stores all recorded note changes
// as a two-dimensional logic array. It's from this music storage
// module that the music FSM reads and loads notes to send
// to the controls modules.
// Author: Brendan Ashworth
module music_storage(
        rst_in,
        note_change_in,
        note_change_index_in,
        note_change_clk_in,
        notes_on,
        notes_stored,
        notes_indexes);

        input logic rst_in;
        // Clocked in when a note change is found by the start_end_detector.
        input logic note_change_clk_in;
```

```
// This is the number of fundamental notes / frequencies
// that the piano can play. This is a result of applying
// the fundamental isolator on the activated frequencies in
// the Fourier transform.
parameter NUM_NOTES = 12;

// The number of note changes we need to uniquely identify
// to play the entirety of the song. Identical to:
// The number of sequences we need to uniquely identify
// to play the entirety of the song. NUM_CHANGES * sample_duration
// gives the maximum length of a song
parameter NUM_CHANGES = 1024;

// The delta functions that represent changes in
// the activated frequencies. This also includes
// information about the current activated frequency;
// i.e., given a single note_change_out, you can
// start playing a song at that point.
// For some given frequency, note_change_out is:
// 00 = no change in activation, off
// 11 = no change, on
// 10 = note turns off
// 01 = note turns on
// note_change_out is designed such that
// note_change_out[0] ^ note_change_out[1]
// signifies that the motors must be actuated in some
// direction specified by note_change_out[0].
input logic [11:0][1:0] note_change_in;

// Starting at 0, this is the index for a note change.
// It will increment with every clock and can be used
// to uniquely identify the activated frequencies in
// a song at any point in time.
input logic [$clog2(NUM_CHANGES)-1:0] note_change_index_in;

// The actual storage array. We need to store information
// about the notes that change at what time periods.
// This necessitates two logical 2d arrays: one for notes,
// one to index the time periods.

// Activations for each note at a certain time index.
// This is effectively the LSB of note_change_in across the notes.
output logic [NUM_CHANGES-1:0][11:0] notes_stored;

output logic [NUM_CHANGES-1:0][11:0] notes_on;

// Time indexes for each stored note. For each entry in notes_stored[i], there
// is an equivalent entry in notes_indexes[i] that describes the time period at
// which this note change occurs.
output logic [NUM_CHANGES-1:0][$clog2(NUM_CHANGES)-1:0] notes_indexes;

// index_pointer contains a time index that indicates at which notes location
// we should store the next note. Storing this prevents traversing the logic
// until the notes_index[j] == 0, which is conceptually the same.
logic [$clog2(NUM_CHANGES)-1:0] index_pointer;
```

```
	always_ff @(posedge note_change_clk_in) begin
	integer i;
	if (rst_in) begin
		notes_stored <= '{default:'0};
		notes_on <= '{default:'0};
		notes_indexes <= '{default:'0};
		// Begin writing at t=1.
		index_pointer <= 1'b1;
	end else begin
		// Store the new note.
		notes_indexes[index_pointer] <= note_change_index_in;

		// Store each note activation individually because it comes
		// in as a [1:0] but we only care about the LSB.
		for (i=0; i < 12; i++) begin
		notes_stored[index_pointer][i] <= note_change_in[i][0] ^ note_change_in[i][1];
		notes_on[index_pointer][i] <= note_change_in[i][1];
		end

		// Increment the pointer.
		index_pointer <= index_pointer + 1'b1;
	end
	end

endmodule
```

6.1.6 serial_tx.sv (Anthony)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// serial_tx.sv
// Created by: Anthony Nardomarino
// 6.111 - Digital Systems Laboratory
// Piano Playing Robot
// 12-02-2019
//
// Based off of 6.111 Lab 2, in which Serial communications are established
// between a serial python reader and the FPGA. This module takes instructions
// sent by the serialToKey.sv module and sends them via 8N1 UART to an ESP32
// with a precalculated baud of 115200 (100Mhz clock / 868).
//
////////////////////////////////////////////////////////////////////////////////


module serial_tx(
	clk, reset, trigger, val, data_out, is_sending//, counter_serial
  );
	input   logic         clk;
	input   logic         reset;
	input   logic         trigger;
	input   logic [7:0] val;
	output  logic         data_out;
	output  logic         is_sending;

//	output logic [7:0] counter_serial;

	parameter DIVISOR = 868; //4.  115.2 kbps baud divisor from 100Mhz
	logic [7:0]	shift_buffer;
	logic [31:0]	counter;
//	assign counter_serial = counter[7:0];

	logic old_trigger;

	always_ff @(posedge clk) begin
	old_trigger <= trigger;
	if(reset)begin
		is_sending	<= 0;
		counter <= 32'd0;
		shift_buffer <= 8'b11111111;
	end else begin
		if (trigger & ~old_trigger) begin
		// take the first edge of the trigger
		shift_buffer[7:0] <= val[7:0];
		counter <= 32'd0;
		is_sending	<= 1;
		end else if (is_sending) begin
		counter <= counter + 1;
		case(counter[14:0])
			DIVISOR*0: data_out <= 1'b0;
			DIVISOR*1: data_out <= shift_buffer[0];
			DIVISOR*2: data_out <= shift_buffer[1];
			DIVISOR*3: data_out <= shift_buffer[2];
			DIVISOR*4: data_out <= shift_buffer[3];
			DIVISOR*5: data_out <= shift_buffer[4];
			DIVISOR*6: data_out <= shift_buffer[5];
			DIVISOR*7: data_out <= shift_buffer[6];
			DIVISOR*8: data_out <= shift_buffer[7];
			DIVISOR*9: begin
				data_out <= 1'b1;
				is_sending	<= 0;
				end
		endcase
		end
	end
	end

endmodule
```

6.1.7 serialToKey.sv (Anthony)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// serialToKey.sv
// Created by: Anthony Nardomarino
// 6.111 - Digital Systems Laboratory
// Piano Playing Robot
// 12-02-2019
//
// Interface for reading motor strategy BRAM and sending instructions
// via the serial_tx.sv module to an ESP32, which uses an I2C protocol
// to address motor arrays (consisting of 12 motors). Operation uses
// a 8N1 UART serial communication protocol between FPGA and ESP32,
// necessitating an 8 bit address for the motors (5 bits followed by
// 3 zeros)
//
////////////////////////////////////////////////////////////////////////////////


module serialToKey(
       clk_100mhz, rst, data_out,
       beat_clk, pb, interest_note
//     debug, counter_out, counter_serial
       );

       parameter DIVISOR = 868;

       // INPUTs
       input logic           clk_100mhz;
       input logic           rst;
       input logic           beat_clk;   // slow beat clock from pb (min 2hz, max 10hz)
       input logic           pb;            // playback wire

       // note of interest (active note according to beat clk)
       input logic [11:0]  interest_note;

       // OUTPUTs
       output logic data_out;       // serial out

//     output logic [24:0] debug;
//     assign debug = {motor_addy, key_addy, motor_count, trigger, is_sending, sending,
curr_beat, old_beat};
//     output logic [9:0] counter_out;

       // Internal logic
       logic [7:0] motor_addy;      // 8 bit address of the motor to be actuated
       logic [7:0] key_addy;        // 8 bit address of motor of interest,
                                    // specified by value of motor_count
       logic [3:0] motor_count;     // counter to 12 motor streams
       logic          trigger;              // trigger to send serial info
       logic          is_sending;                  // trigger from serial to indicate full
serial stream sent
       logic          sending;
       logic          curr_beat;
       logic          old_beat;

       logic [31:0] counter;
//     assign counter_out = counter;

//     output logic [7:0] counter_serial;

       serial_tx fingers (.clk(clk_100mhz), .reset(rst), .trigger(trigger),
                     .val(motor_addy), .data_out(data_out), .is_sending(is_sending)/*,
                     .counter_serial(counter_serial)*/);

       addressKeyMap motorKey (.countIn(motor_count), .addrOut(key_addy));

       always_ff @(posedge clk_100mhz) begin
       curr_beat      <= beat_clk;
       old_beat       <= curr_beat;
       if (rst) begin
```

```
                trigger       <= 0;
                sending       <= 0;
                motor_count <= 4'd0;
                counter       <= 0;
        // onset of beat clock: send notes
        end else if (curr_beat & ~old_beat & pb) begin
                sending       <= 1;
                motor_count <= 4'd1;
                counter       <= 0;
                motor_addy  <= interest_note[11] ? 8'b0100_0000 : 8'b1111_1111;
        end else if (motor_count == 12) begin
                sending       <= 0;
                counter       <= 0;
        end else if (sending & (counter==32'd8680)) begin
                counter       <= 0;
                motor_count <= motor_count + 1;
                trigger <= 1;
//              motor_count <= ~is_sending ? (motor_count + 1)  : (motor_count);
//              trigger       <= is_sending;
                motor_addy  <= interest_note[11 - motor_count] ? key_addy : 8'b1111_1111;
        end else if (sending) begin
                trigger <= 0;
//              trigger       <= is_sending;
                counter       <= counter + 1;
        end
        end

endmodule // serialToKey
```

6.1.8 start_end_detector.sv (Brendan)

```
// The start end detector effectively takes the
// derivative of an array of activated frequencies,
// giving the time indices at which they change in value
// (turn off or on). This transforms a step function in
// both directions to delta functions towards negative and
// positive infinity.
// Brendan Ashworth
module start_end_detector(
        rst_in,
        activated_notes_in,
        note_clk_in,
        note_change_out,
        note_change_index_out,
        note_change_clk_out
        );

        // This is the number of fundamental notes / frequencies
        // that the piano can play. This is a result of applying
        // the fundamental isolator on the activated frequencies in
        // the Fourier transform.
        parameter NUM_NOTES = 12;

        parameter NUM_CHANGES = 1024;

        input logic rst_in;
```

```
// An array of activated notes - a 0 signifies
// the note is off, a 1 is on.
input [11:0] activated_notes_in;
// note_clk_in pulses when a new sequence of activated notes
// are introduced.
input logic note_clk_in;

// The delta functions that represent changes in
// the activated frequencies. This also includes
// information about the current activated frequency;
// i.e., given a single note_change_out, you can
// start playing a song at that point.
// For some given frequency, note_change_out is:
// 00 = no change in activation, off
// 11 = no change, on
// 10 = note turns off
// 01 = note turns on
// note_change_out is designed such that
// note_change_out[0] ^ note_change_out[1]
// signifies that the motors must be actuated in some
// direction specified by note_change_out[0].
output logic [11:0][1:0] note_change_out;

// Starting at 0, this is the index for a note change.
// It will increment with every clock and can be used
// to uniquely identify the activated frequencies in
// a song at any point in time.
output logic [$clog2(NUM_CHANGES)-1:0] note_change_index_out;

// The activated notes on the last clock pulse.
// The same value as the LSB of note_change_out for
// all notes.
logic [11:0] last_activated_notes;

// Whether or not that specific frequency has a change.
// OR ing this entire logic will provide a logical 1 if
// there is a note change detected.
logic [11:0] note_has_change;

// This clk pulses any time there is a new note_change_out
// available to process.
output logic note_change_clk_out;

always_ff @(posedge note_clk_in) begin
if (rst_in) begin
       for (integer i = 0; i < 12; i++) begin
       note_change_out[i] <= 2'b00;
       last_activated_notes[i] <= 0;
       end

       note_change_index_out <= 0;
       note_change_clk_out <= 0;
end else if (note_change_clk_out) begin
       // Turn off the clock out if it was on.
       // Technically speaking this disallows the start end detector
       // from detecting changes that happen immediately after another
       // change, but this is short-lived for the duration of one
```

```
            // sample. In other words, for the real piano this doesn't
            // actually matter, and if anything will improve performance
            // by smoothing the outcoming notes.
            note_change_clk_out <= 0;

        end else begin
            // If we just clocked the pulse, unpulse.
            // Go through each individual note.
            for (integer i = 0; i < 12; i++) begin
            // Let [1] be the last note.
            // Let [0] be the current activated note.
            note_change_out[i] <= {last_activated_notes[i], activated_notes_in[i]};

            // Store the old notes so we don't reactivate on the same one.
            last_activated_notes[i] <= activated_notes_in[i];
            end

            note_change_clk_out <= (last_activated_notes[0] ^ activated_notes_in[0])
                    | (last_activated_notes[1] ^ activated_notes_in[1])
                    | (last_activated_notes[2] ^ activated_notes_in[2])
                    | (last_activated_notes[3] ^ activated_notes_in[3])
                    | (last_activated_notes[4] ^ activated_notes_in[4])
                    | (last_activated_notes[5] ^ activated_notes_in[5])
                    | (last_activated_notes[6] ^ activated_notes_in[6])
                    | (last_activated_notes[7] ^ activated_notes_in[7])
                    | (last_activated_notes[8] ^ activated_notes_in[8])
                    | (last_activated_notes[9] ^ activated_notes_in[9])
                    | (last_activated_notes[10] ^ activated_notes_in[10])
                    | (last_activated_notes[11] ^ activated_notes_in[11]);

            // Increment the index.
            note_change_index_out <= note_change_index_out + 1'b1;
        end
        end

endmodule
```

6.1.9 fsm.sv (Brendan)

```
// The finite state machine. This represents the different possible
// states the piano playing robot can enter, as a superposition
// of possible environment observables.
// Author: Brendan Ashworth
module fsm(clk_in, rst_in,
        fft_done,
        counter,
        state_out,
        recording_in, playing_in,
        playing_out, recording_out, paused_out);

        // The number of note changes we need to uniquely identify
        // to play the entirety of the song.
        parameter NUM_CHANGES = 1024;

        // System
        input logic clk_in;
        input logic rst_in;
```

```
// Signifies that the FFT is done with processing
// data.
input logic fft_done;

// User inputs
// recording_in is a switch
input logic recording_in;
// playing_in is a switch
input logic playing_in;

// We output information about the state as information
// about the environment.
output logic playing_out;
output logic recording_out;
output logic paused_out;

// State storage.
logic [3:0] state;

output logic [3:0] state_out;
assign state_out = state;

// Counter for song playback.
// Each tick (+1) in this counter represents
// a window length for the FFT.
output logic [$clog2(NUM_CHANGES)-1:0] counter;
logic [$clog2(NUM_CHANGES)-1:0] length;

// Each state can be broken down into environment states.
const int ENV_PAUSED = 4'b0010;
assign paused_out = (state & ENV_PAUSED) == ENV_PAUSED;

const int ENV_PLAYING = 4'b0001;
assign playing_out = (state & ENV_PLAYING) == ENV_PLAYING;

const int ENV_RECORDING = 4'b0100;
assign recording_out = (state & ENV_RECORDING) == ENV_RECORDING;

const int ENV_IDLE = 4'b1000;

// Represent each state as a combination of those
// environments.
const logic [3:0] STATE_PLAY = ENV_PLAYING;
const logic [3:0] STATE_PAUSED = ENV_PAUSED;
const logic [3:0] STATE_RECORDING = ENV_RECORDING;
const logic [3:0] STATE_ANALYZING = 0; // analyzing represents none of the environments
const logic [3:0] STATE_IDLE = ENV_IDLE;

always_ff @(posedge clk_in) begin
if (rst_in) begin
       // If reset is high, transition to the idle state.
       state <= STATE_IDLE;
       // Reset the counter, length.
       counter <= 0;
       length <= 14'b11_1111_1111_1111; // max value
end else begin
```

```
            // Handle state transitions based on inputs.
            if (state == STATE_PLAY) begin
            // If we stop playing transition to paused.
            if (!playing_in)
                   state <= STATE_PAUSED;
            // The counter reaching the song length means we're done with playback.
            else if (counter == length) begin
                   // Go back to the beginning.
                   counter <= 0;
                   state <= STATE_PAUSED;
            end else begin
                   // We're currently playing the song,
                   // increment the counter.
                   counter <= counter + 1'b1;
            end
            end else if (state == STATE_PAUSED) begin
            // If we hit the play button continue/start.
            if (playing_in)
                   state <= STATE_PLAY;
            end else if (state == STATE_RECORDING) begin
            // Increment the counter so long as we're recording.
            // This allows us to find the length of the song.
            counter <= counter + 1'b1;

            // If the user stops recording, transition to finish analyzing.
            if (!recording_in)
                   state <= STATE_ANALYZING;
            end else if (state == STATE_ANALYZING) begin
            // Reset the counter, set the length accordingly.
            length <= counter + 1'b1;
            counter <= 0;

            // Analyzing continues as the FFT finishes in the pipeline.
            // However long FFT takes, we wait for it to finish before allowing
            // the user to continue with playback.
            // FFT signifies it's done with analyzing with an input signal of
            // fft_done. fft_done is low when it's processing real audio data.
            if (fft_done)
                   state <= STATE_PAUSED;
            end else if (state == STATE_IDLE) begin
            // Resetting the system automatically transitions us
            // to the idle state, and so does a song finishing playback.
            // The user can begin recording by hitting the recording switch.
            if (recording_in)
                   state <= STATE_RECORDING;
            end
       end
       end

endmodule
```

6.1.10 onset_detector.sv (Max)

Note: 6.1.10 was not in synthesis on the FPGA but was adopted as we got closer to the deadline to be simpler. Merged with the spectrogram.sv.

```
// Max Hardy
module H(input reg [15:0] x,
       output reg [31:0] h
              );

       reg [15:0] abs_x;


       abs my_abs(.x(x),.abs(abs_x));

       always @* begin
       // rejects decreasing peaks
       // allows increasing peaks to have non-zero value
       h =(x+abs_x)/2;

       end

endmodule
```

6.1.11 peak_detector.sv (Max)

Note: 6.1.11 was not in synthesis on the FPGA but was adopted as we got closer to the deadline to be simpler. Merged with the spectrogram.sv.

```
// Max Hardy
module peak_detector( input logic clk_in,
                      input logic rst_in,
                      input logic frame_done,
                      input logic [31:0] dif,
                      input logic [15:0] threshold,
                      output logic max_flag,
                      output logic peak
       );

       logic [21:0] cool_off;

       always_ff @(posedge clk_in) begin

       if (rst_in) begin

              peak<=0;
              cool_off<=0;

       end else if (frame_done & cool_off == 0) begin

              if (dif > threshold) begin

              peak<=1;
              cool_off<=threshold[3:0] << 18;

              end else begin

              peak<=0;

              end

       end else if (frame_done & cool_off > 0) begin
```

```
               cool_off<=cool_off-1;
               peak<=0;
       end
       end
endmodule
```

6.2 Associated ESP32 C++

```
#include <ESP32Servo.h>
#include <HardwareSerial.h>

#define RXP   26             // rx pin for MySerial reading
#define TXM   17             // misc tx pin for MySerial interface
#define BAUD  115200         // Serial baud rate for FPGA interfacing
#define IDLE  0
#define PLAY  1
#define IDLE_TIME 2000       // milliseconds until idle transition
#define ACT_DELAY 100        // actuation delay for servos to reach desired position
#define NUM_MOTORS 12        // motors cover one octave

/* pianoMan.ino
  @author: Anthony Nardomarino
  6.111 Digital Systems Laboratory
  12-01-2019
  ESP32 Interface with FPGA Piano Playing Robot Hands
*/

HardwareSerial MySerial(1);

//             Key:
//
// 0100_0000 - A      0101_0000 - C#   0100_1000 - F
// 0010_0000 - A#     0011_0000 - D  0010_1000 - F#
// 0110_0000 - B      0111_0000 - D#   0110_1000 - G
// 0001_0000 - C      0000_1000 - E  0001_1000 - G#

Servo fgA, fgAs, fgB, fgC, fgCs, fgD;
Servo fgDs, fgE, fgF, fgFs, fgG, fgGs;

Servo keys[NUM_MOTORS] = {fgA,  fgAs, fgB, fgC,  fgCs, fgD,
                          fgDs, fgE,  fgF, fgFs, fgG,  fgGs};

// 8 bit addresses for each motor
const byte  addresses[NUM_MOTORS] = {64, 32, 96, 16, 80, 48, 112, 8, 72, 40, 104, 24};

// measured resting high angles for servo fingers
const int __upStrats[NUM_MOTORS] = {700, 700, 650, 2200, 2250, 2150, 2200, 2300, 2200, 700,
700, 700};

// measured low angles for activated servo fingers
const int downStrats[NUM_MOTORS] = {840, 820, 810, 2000, 2000, 1850, 2050, 2000, 2000, 900,
900, 900};

const byte  servoPins[NUM_MOTORS] = {4, 5, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22};
```

```
// strategies for all motors
byte dirs[NUM_MOTORS] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};     // 0 for up, 1 for down
                                                                  // old state of motors to be
compared

byte buttons[NUM_MOTORS] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // onset control
                                                                  // new state of motors
received serially
int state;                   // PLAY or IDLE hands

int timer;                   // idle timer

int   serialIn;              // Serial input from FPGA
int   counter;               // counts 12 serial inputs before actuation
byte  needDel = 0;           // dynamic delay logic

void setup() {
  // Setup serial channels
  Serial.begin(BAUD);
  MySerial.begin(BAUD, SERIAL_8N1, RXP, TXM);
  // attach and program rest state servos
  for (int i = 0; i < NUM_MOTORS; i ++){
       keys[i].attach(servoPins[i]);
  }
  for (int i = 0; i < NUM_MOTORS; i ++){
       keys[i].writeMicroseconds(__upStrats[i]);
  }
  delay(100);
  counter = 0;
  // detach to prevent current flow from skipping motors
  for (int i = 0; i < NUM_MOTORS; i ++){
       keys[i].detach();
  }
  state = IDLE;
  timer = millis();
}

void loop() {
  // onset reads
  // Read Serial and update onset control buttons
  serialIn = MySerial.read();
  if(serialIn != -1){
       counter ++;
       timer = millis();
       if (state == IDLE){
              state = PLAY;
              for (int i = 0; i < NUM_MOTORS; i ++){
                     keys[i].attach(servoPins[i]);
              }
       }
       for (int i = 0; i < NUM_MOTORS; i ++){
              if(serialIn == addresses[i]){
                     buttons[i]     = 1;
                     needDel               = 1;
              }
       }
  }
  // Check for state change
  if(millis() - timer > IDLE_TIME && state == PLAY){
       state = IDLE;
       for (int i = 0; i < NUM_MOTORS; i ++){
              keys[i].writeMicroseconds(__upStrats[i]);
       }
       delay(100);
       for (int i = 0; i < NUM_MOTORS; i ++){
              keys[i].detach();
       }
  }

  // Actuate activated motors
  if(counter >= 12 && state == PLAY){
       for(int i = 0 ; i < NUM_MOTORS; i ++){
              // onset of play
              if (dirs[i] == 0 && buttons[i] == 1){
              keys[i].writeMicroseconds(downStrats[i]);
              // onset of release
              } else if (dirs[i] == 1 && buttons[i] == 0) {
              keys[i].writeMicroseconds(__upStrats[i]);
              }
       }

       if(needDel){
              delay(ACT_DELAY);
       }
       needDel = 0;
       // reset serial received states and update old states
       for(int i = 0; i < NUM_MOTORS; i ++){
              dirs[i] = buttons[i];
              buttons[i] = 0;
       }
  }
}
```

6.3 Associated Signal Processing Simulation Python (Max)

```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov  9 19:38:16 2019

@author: mhare
"""

import pyaudio
import wave
import numpy as np
import wav_utils
import pandas
from numpy.fft import fft as nfft
from numpy.fft import ifft as nifft
import time

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from math import e, pi, sin, cos, log
j = 1j

def fft(x):
    return (nfft(x)/len(x)).tolist()

def ifft(x):
    return (nifft(x)*len(x)).tolist()

def mic_in(name,seconds,rate):
    FORMAT = pyaudio.paInt16
    CHANNELS = 1
    RATE = rate
    CHUNK = 1024
    RECORD_SECONDS = seconds
    WAVE_OUTPUT_FILENAME = name

    audio = pyaudio.PyAudio()

    # start Recording
    stream = audio.open(format=FORMAT, channels=CHANNELS,
                    rate=RATE, input=True,
                    frames_per_buffer=CHUNK)
    print("recording...")
    frames = []

    for i in range(0, int(RATE / CHUNK * RECORD_SECONDS)):
        data = stream.read(CHUNK)
        frames.append(data)
    print("finished recording")


    # stop Recording
    stream.stop_stream()
    stream.close()
    audio.terminate()

    waveFile = wave.open(WAVE_OUTPUT_FILENAME, 'wb')
    waveFile.setnchannels(CHANNELS)
    waveFile.setsampwidth(audio.get_sample_size(FORMAT))
    waveFile.setframerate(RATE)
    waveFile.writeframes(b''.join(frames))
    waveFile.close()

def read_in(name):
    file=Wave.from_file(name)
    file.plot()
    samples=file.samples
    return samples

def stft(x, window_size, step_size, sample_rate):
    nsteps = (len(x)-window_size)//step_size + 1
    return [fft([a*b for a,b in
zip(x[i*step_size:i*step_size+window_size],np.hanning(window_size))]) for i in range(nsteps)]

def k_to_hz(k, window_size, step_size, sample_rate):
    return k*sample_rate/(window_size)


def hz_to_k(freq, window_size, step_size, sample_rate):
    return round(freq*window_size/(sample_rate))


def timestep_to_seconds(i, window_size, step_size, sample_rate):
    return round(i*step_size/sample_rate,2)


def transpose(x):
    return [[i[j] for i in x] for j in range(len(x[0]))]

def spectrogram(X, window_size, step_size, sample_rate):
    return [[abs(i)**2 for i in j] for j in transpose(X)]


def plot_spectrogram(sgram, window_size, step_size, sample_rate):
    width = len(sgram[0])
    height = len(sgram)//2+1  # only plot values up to N/2

    plt.imshow([[log(i) if i !=0 else -30 for i in j] for j in sgram[:height+1]],
aspect=width/height)
    plt.axis([0, width-1, 0, height-1])

    ticks = ticker.FuncFormatter(lambda x, pos: '{0:.1f}'.format(timestep_to_seconds(x,
window_size, step_size, sample_rate)))
    plt.axes().xaxis.set_major_formatter(ticks)
    ticks = ticker.FuncFormatter(lambda y, pos: '{0:.0f}'.format(k_to_hz(y, window_size,
step_size, sample_rate)))
    plt.axes().yaxis.set_major_formatter(ticks)

    plt.xlabel('time [s]')
    plt.ylabel('frequency [Hz]')

    plt.colorbar()
    plt.show()

def H(x):
    return (x + abs(x))/2

def spectral_difference(X):
    N = len(X[0])
    out = []
    for ix in range(len(X)):
        o = 0
        for k in range(N):
            if ix == 0:
                o += abs(X[ix][k])**2
            else:
                o += H(abs(X[ix][k]) - abs(X[ix-1][k]))**2
        out.append(o)
    return out

def find_peaks(x, threshold, min_spacing):
    x = x[:]
    out = []
    while True:
        p = max(range(len(x)), key=lambda i: x[i])
        if x[p] <= threshold:
            break
        out.append(p)
        for i in range(min_spacing):
            for index in {p-i, p+i}:
                if 0 <= index < len(x):
                    x[index] = 0
    return sorted(out)

def k_at_time(X, n, music_dictionary):
    valids=[j[1] for j in music_dictionary.values()]
    return max([i for i in range(len(x[n])) if i in valids], key=lambda ix: abs(X[n][ix])**2)

def mode(x):
    return max(x, key=lambda i: x.count(i))

def k_for_note(X, n_start, n_stop, music_dictionary):
    return mode([k_at_time(X, i, music_dictionary) for i in range(n_start+1, n_stop)])


def notes_ref_table(window_size, step_size, sample_rate):
    ref_table=pandas.read_excel('C:\\Users\\mhare\\Desktop\\Life\\MIT\\Course
6\\6.111\\Musical_Frequencies.xlsx',sheet_name='Sheet1')
    frequencies=list(ref_table.iloc[:,4].dropna())[1:]
    names = list(ref_table.iloc[:,2].dropna())[1:]
    octaves = list(ref_table.iloc[:,3].dropna())[1:]
    simple_names = [i.split(' ')[0] for i in names]
    full_names=[simple_names[i]+str(octaves[i]) for i in range(len(octaves))]
    music_dictionary = dict(zip(full_names, frequencies))


    for i in music_dictionary.keys():
        k=hz_to_k(music_dictionary[i], window_size, step_size, sample_rate)
        music_dictionary[i]=[music_dictionary[i], k]

    return music_dictionary

def instructions(x, peaks, music_dictionary, window_size, step_size, sample_rate):
    notes=[]
    for i in peaks:
        try:
            notes.append(k_for_note(x, i, i+20, music_dictionary))
        except IndexError:
            cap=len(x)-i-1
            notes.append(k_for_note(x, i, i+cap, music_dictionary))

    instructions=[]
    for i in range(len(notes)):
        for j in music_dictionary.keys():
            if music_dictionary[j][1]==notes[i]:
                onset=timestep_to_seconds(peaks[i], window_size, step_size, sample_rate)
                instructions.append([j,onset])
    return instructions
```

```
def play_back(music_dictionary, ins, samples, name):
    sr=44100
    notes=[i[0] for i in ins]
    times=[i[1] for i in ins]
    freqs=[music_dictionary[i][0] for i in notes]
    ns=[int(sr*i) for i in times]
    ns.append(len(samples))
    amps=[]
    for i in range(0,len(ns)-1):
        t=np.arange(ns[i],ns[i+1])/44100
        amps.extend(np.cos(freqs[i]*t*2*np.pi))
    w=Wave(amps,sr)
    w.save(name)

    chunk=1024
    wf = wave.open('C:\\Users\\mhare\\Desktop\\Life\\MIT\\Course 6\\6.111\\'+name, 'rb')

    p = pyaudio.PyAudio()

    stream = p.open(format =
                    p.get_format_from_width(wf.getsampwidth()),
                    channels = wf.getnchannels(),
                    rate = wf.getframerate(),
                    output = True)


    data = wf.readframes(chunk)
    start=time.time()
    while data != '':
        stream.write(data)
        data = wf.readframes(chunk)
        end=time.time()
        if (end-start>=5):
            break


    stream.close()
    p.terminate()

def highest(y):
    m=0
    for i in y:
        for j in i:
            a=j
            if (a>m):
                m=a
    return m

def fundamentals(music_dictionary,y, window_size, step_size, sample_rate):
    m=highest(y)
    ks=[i[1] for i in music_dictionary.values()]
    ks=list(set(ks))
    ks.sort()
    all_notes=[]
    for k in ks:
        times=y[k]
        note=[]
        for t in times:
            if (t>=.02*m):
                note.append(1)
            else:
                note.append(0)
        all_notes.append(note)

    whole=[]
    for n in range(len(all_notes)):
        sub=[i for i in music_dictionary.keys() if ks[n]==music_dictionary[i][1]]
        note = all_notes[n]
        subsub=[]
        for i in range(1,len(note)):
            if note[i]-note[i-1]==1:
                subsub.append(timestep_to_seconds(i, window_size, step_size, sample_rate))
            elif note[i]-note[i-1]==-1:
                subsub.append(timestep_to_seconds(i, window_size, step_size, sample_rate))
            if len(subsub)==2:
                sub.append(subsub)
                subsub=[]
        whole.append(sub)

    return whole

def chords(music_dictionary,y, window_size, step_size, sample_rate, peaks):
    valids=[j[1] for j in music_dictionary.values()]
    for time in peaks:
        powers={}
        for k in valids:
            power=0
            for i in range(20):
                power = power+abs(x[time+i][k])**2
            powers[(time,k_to_letter(music_dictionary, k))]=power/20
        high=max(powers.values())
        for key in powers.keys():
            if powers[key]>=.1*high:
                print(key, powers[key])
        #print(sorted(powers.items(), key = lambda kv:(kv[1], kv[0])))
        print('************************')

def k_to_letter(music_dictionary, k):
    for i in music_dictionary.keys():
        if music_dictionary[i][1] == k:
            return i

window_size = 8192
step_size = 256
sample_rate = 44100
name = 'test20.wav'
mic_in(name,5,44100)
samples=read_in(name)
x=stft(samples, window_size, step_size, sample_rate)
y=spectrogram(x, window_size, step_size, sample_rate)
plot_spectrogram(y, window_size, step_size, sample_rate)

music_dictionary=notes_ref_table(window_size, step_size, sample_rate)
```

```
print(music_dictionary)

#ins2=fundamentals(music_dictionary,y, window_size, step_size, sample_rate)

dif=spectral_difference(x)
plt.plot(dif)
plt.show()
peaks=find_peaks(dif,.00005,40)
print(peaks)

ins=instructions(x, peaks, music_dictionary, window_size, step_size, sample_rate)

chords(music_dictionary,y, window_size, step_size, sample_rate, peaks)

#play_back(music_dictionary, ins, samples, 'testback20.wav')


#print(instructions)
print('done')
#
#def main():
#    window_size = 8192
#    step_size = 256
#    sample_rate = 44100
#    name = 'test10.wav'
#    mic_in(name,5,44100)
#    samples=read_in(name)
#    x=stft(samples, window_size, step_size, sample_rate)
#    y=spectrogram(x, window_size, step_size, sample_rate)
#    plot_spectrogram(y, window_size, step_size, sample_rate)
#
#    music_dictionary=notes_ref_table(window_size, step_size, sample_rate)
#
#
#    print(music_dictionary)
#
#    instructions=fundamentals(music_dictionary,y, window_size, step_size, sample_rate)
#    print(instructions)
#    print('done')
#
#if __name__=="__main__":
#    main()
```