6.111 Fall 2014 Final Project

Chordination

Real-time harmonization for your improv compositions.

Jacquelyn De Sa
Chris Kevin Ong
Zixi Liu

Mentored by Michael Price

December 10th, 2014

Introduction

Jacqui De Sa

Project Introduction

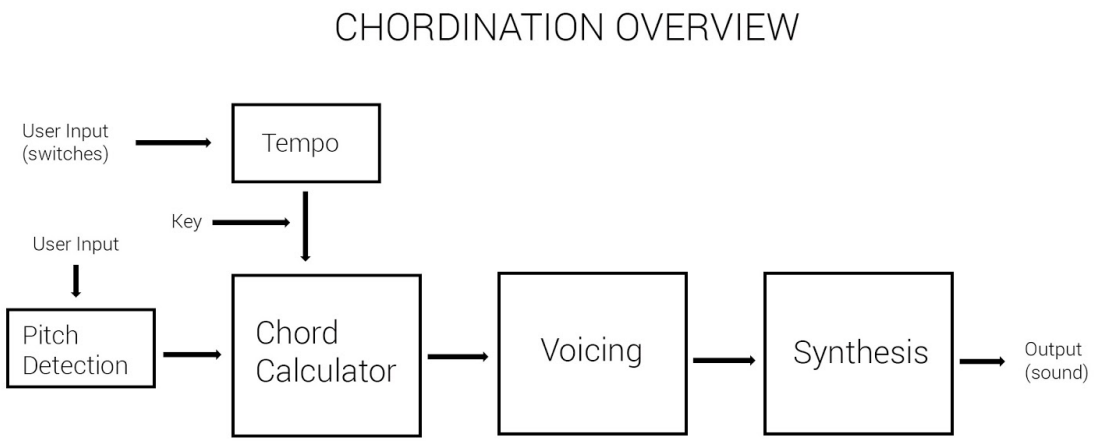
From Top 40 Charts, to classical symphonies, to Jazz Big Bands and rock hits, all successful songs share one common feature -- they never contain just one single line of music, but instead weave together the sounds of different instruments and musicians to produce a sound that is richer and fuller than a single vocal line can ever produce. But producing a harmony -- musical parts accompanying a melodic line allocated to up to hundreds of musicians in an ensemble -- is no simple task. The process of creating a classical harmonization for a melodic line can take years of study; finding musicians to play these lines can be expensive and difficult. Current technology (specifically vocoders) allow musicians to play in their new melodies and creates harmony around them (Write-music.com, 2014). However, this existing process removes the impact of harmonization on the composition process, as the harmonization is synthesized after the fact, and not during the musical composition process. Our 6.111 final project, Chordination, serves to solve this existing problem by producing an FPGA-based real-time harmonization of notes as a user plays them.

Team Introduction & Motivation

The three team members on this project, CK Ong, Jacqui De Sa, and Zixi Liu, are Juniors in the Department of Electrical Engineering & Computer Science at MIT. All three have studied classical music theory through classes at MIT (21M.301 -- Harmony & Counterpoint I) and vocal performance through acapella groups and classical ensembles including the MIT Chamber Chorus, the Chorallaries, the Asymptones, and Techiya. All enjoy organically singing improved harmonies with others and enjoy the process of composing and arranging pieces of music. This project aims to bring the joy of harmonization and composition and make it an accessible and organic process available to all.

Project Overview

Chordination is composed of four major subcomponents: a Pitch Detector, which interfaces with the user to take in information on which pitch the user wants to play at a given time, a Chord Calculator subcomponent, which implements classical music chord progression rules to determine possible chords to be outputted at the next timestep, a Voicing subcomponent which implements classical music theory rules on how individual music lines in a harmonized piece transition from their note on one chord to their note on the next chord, and a Sound Synthesis subcomponent which plays chords back to the user in real-time.



1: Overview of Chordination Subcomponents

Although the Chord Calculator and Voicing Subcomponents were able to be implemented according to their original proposed project specifications, the Pitch Detector and Sound Synthesis modules were implemented through fallback options as the original plan -- to take in a user's sung note and output chords as pitch-shifted versions of the user's own voice -- was out of scope of the project and proved unexpectedly difficult to implement. As a result, the Pitch Detector module was reimplemented with a piano-like interface, in which the buttons of the FPGA were used as keys of a piano and switches were used to select a user's key, tempo, and octave. Additionally, the Sound Synthesis module was implemented using square wave synthesis instead of the original pitch-shifted voice.

Pitch Detector Subcomponent

Zixi Liu

Detect the pitch frequency of the user's singing voice.

Overview of Original Planned Implementation

The composition of a musical piece first begins with a simple melody -- a tune that can then be harmonized, contrasted, and added to over the course of a piece. The pitch detector module interfaces between a user inputting a melody and the remaining submodules, which determine the next possible chord and play it. The pitch detector module aims to achieve this through three steps: down-sample, increment and window then Fourier transform to determine the frequency.

The Pitch Detector Subcomponent was first implemented in MATLAB in order to create a valid software prototype. This software prototype underwent several iterations before being implemented in Verilog. This initial implementation was unable to be finished

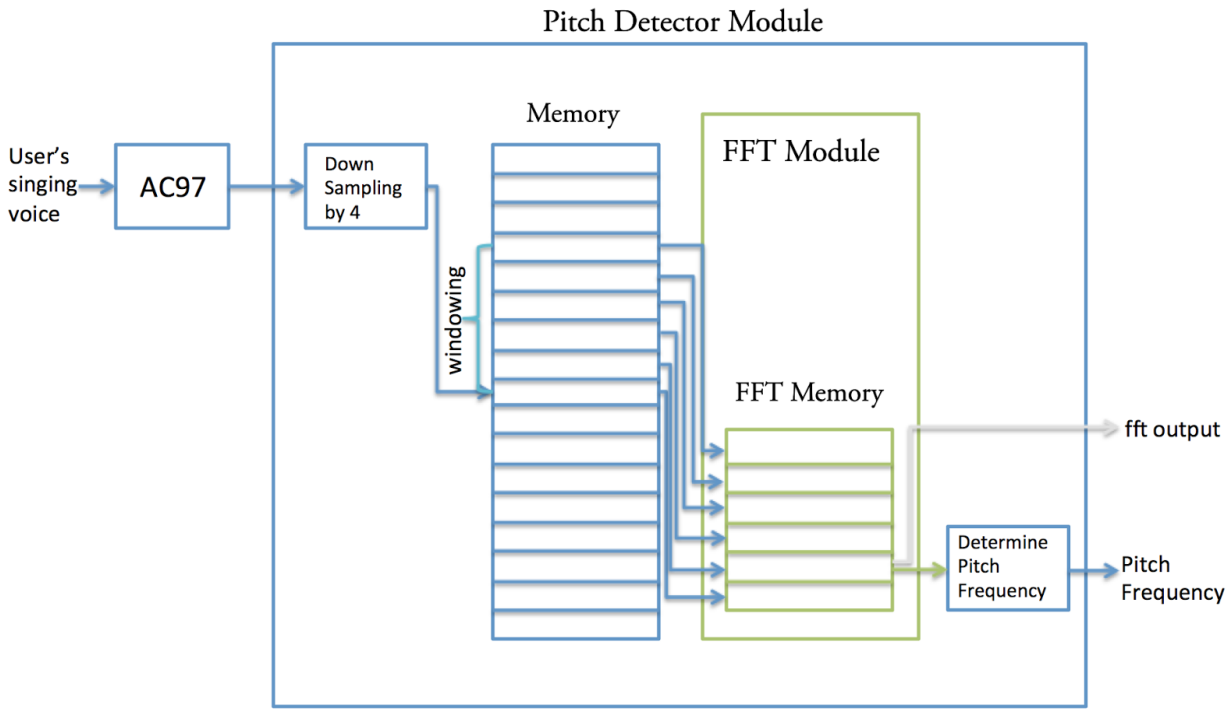


Figure 2: Overview of the Pitch Detection Subcomponent

Step One: Down-Sampling the Input

Chordination interfaces between the user's melodic sound input and the Pitch Detector Subcomponent through the AC97 module, an analog to digital converter, sampled with a sampling frequency of 48kHz. To accommodate the memory size of the Fast Fourier

Transform (FFT) module, which is used in later steps to compute the Fourier transform, digital signal must be downsampled by a factor of four.

Downsampling is triggered by a simple counter which counts from zero to its maximum value, three. When the counter reaches its maximum, the digital signal is stored inside a BRAM (Block RAM) which can exactly accommodate (is equal to) the size of the memory in the FFT module. When the BRAM is full, the new incoming data will directly overwrite the old data.

Step two: Increment and Window

In order to accurately detect pitches, the Pitch Detection Subcomponent samples a pitch from the user audio input every 1/16th of a second. This sampling interval is short enough to detect chord changes, since chord transitions occur at most every half a second according to the specifications of Chordination.

To determine the pitch frequency in the span of 1/16th of a second, Chordination examines audio input data from the last half second and computes the Fourier transform of this data to lock onto the currently-playing pitch. Note that the signal (the data from the most recent half second) is not multiplied by a Gaussian window, which is a common procedure in sound signal processing, because the Gaussian window does not make a significant difference according to our software prototype in MATLAB with our sampling interval being set to 1/16th of a second and our window being set to half a second respectively.

Step three: Take Fourier Transform and Determine the Frequency

As discussed in step two, the pitch frequency is determined by finding the maximum value in the Fourier transform of the windowed data. The detected frequency is also bounded by a range of 130 Hz to 1047 Hz, which is the range of a typical human voice. Any detected frequency outside this range is set to the boundary value.

Figure 3 demonstrates a graph of time vs. magnitude of a signal with frequency 260Hz (a Middle C on a piano), and a frequency vs. magnitude graph of the Fourier Transform of the signal. These graphs are generated in MATLAB using its FFT (Fast Fourier Transform) feature. In this example, the Fourier Transform was computed with data sampled every half a second. As seen from the figure, the correct frequency of 262 Hz in this example is successfully detected.

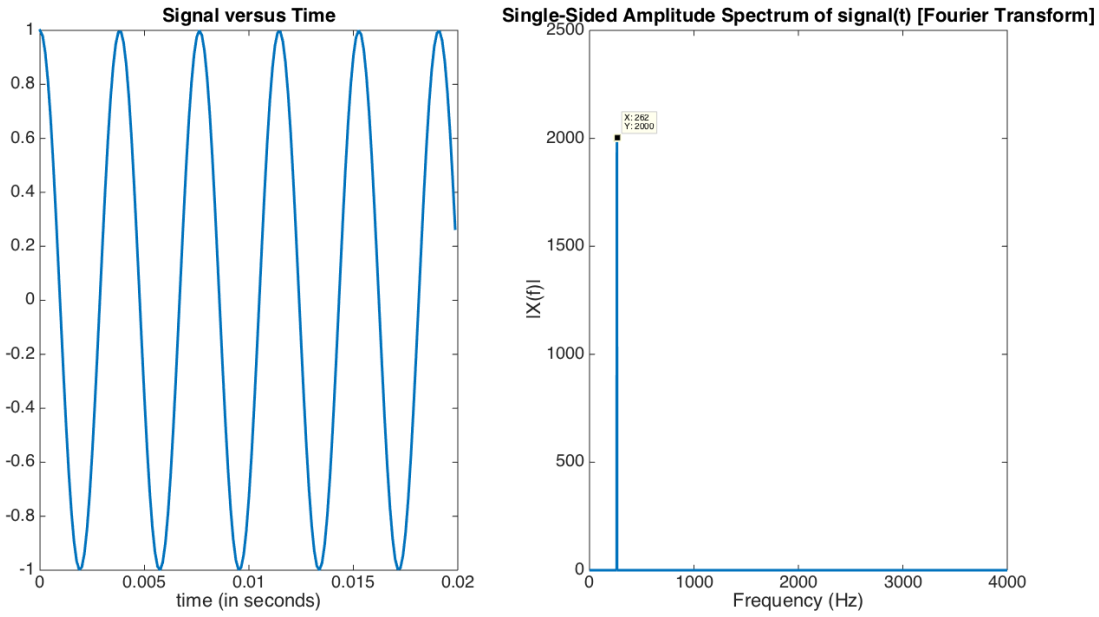


Figure 3: Matlab graphs showing how frequency is detected

Testing Strategy

Within MATLAB

Three files are selected in the MATLAB software prototype test: one is a pure tone of 262 Hz (middle C), the second is the MIDI file of the alto part of a song arranged for acapella, and the third one is generated from a recording of one person singing. For the first two test files, no background noise is present, so the output can be used as a benchmark to compare the impact of background noise on pitch detection.

Test 1 Result:

Test 1 examines whether the module generates the correct output with a pure tone as input. Additionally, it also tests for how delay induced by required calculation. Figure 4 below is a plot of the output. The correct frequency is detected with a delay of a tenth of a second.

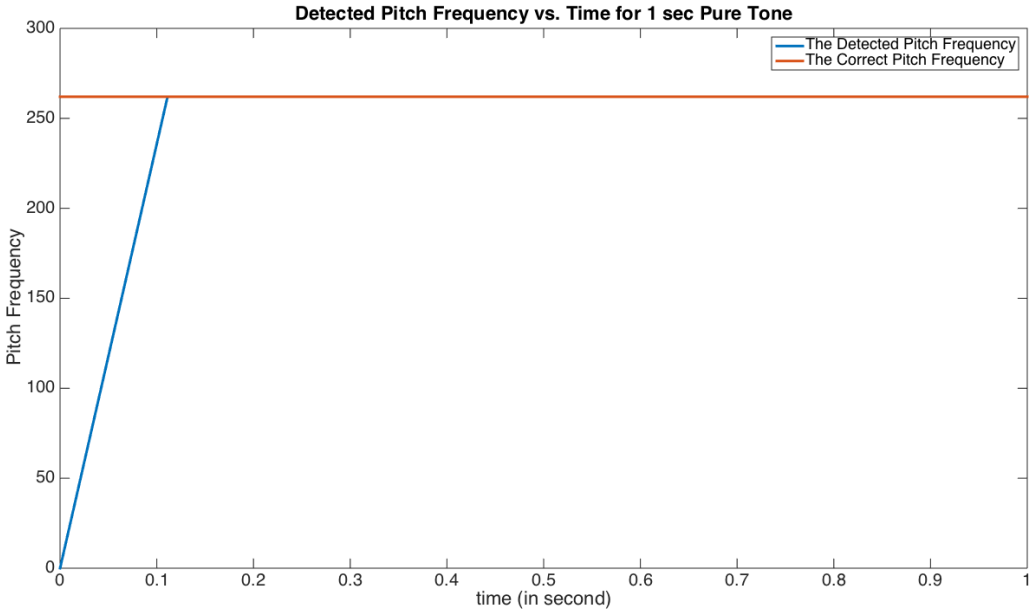


Figure 4: Output for pitch detection of a pure tone

Test 2 Result:

Using the second test validation file, the correct frequency was only detected for the first six seconds. Figure five below is a plot of both the detected pitch and correct pitch for the first six seconds.

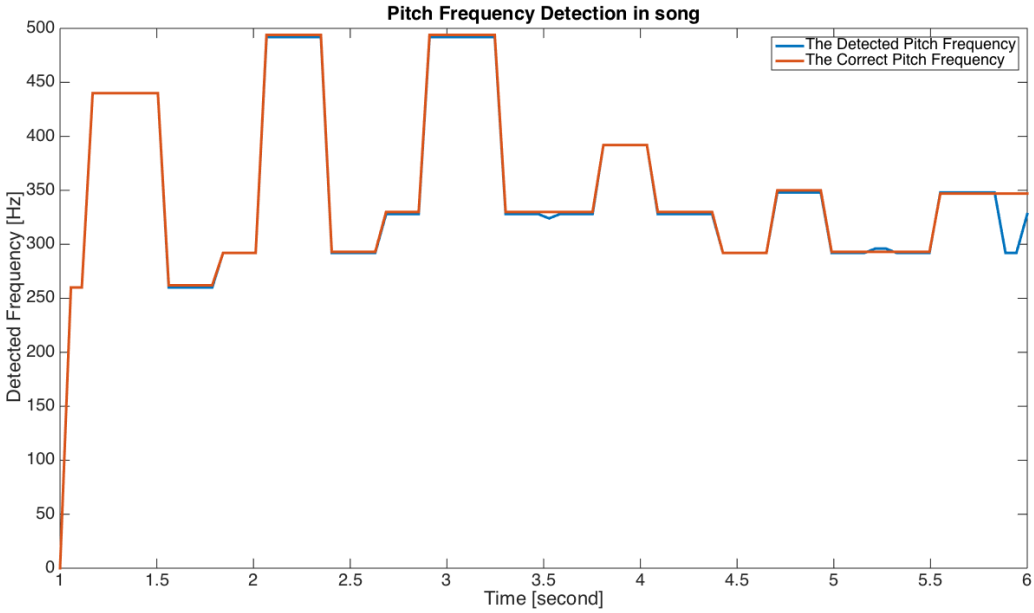


Figure 5: Output for pitch detection during six seconds of one voice part of a real song

Figure 6 below compares the detected pitch and the correct pitch for the entire duration of the music.

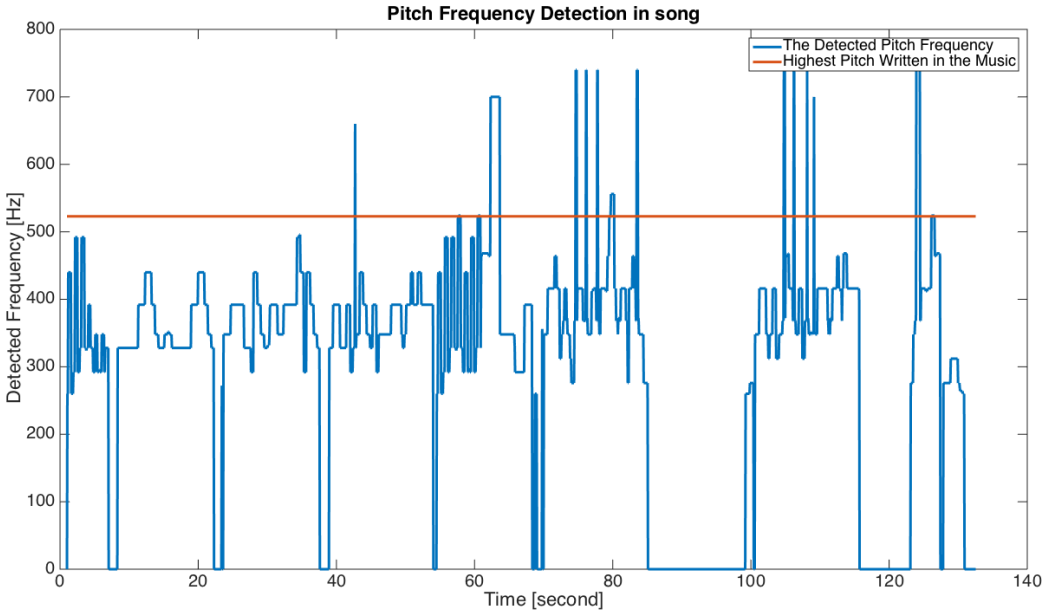


Figure 6: Output for pitch detection for an entire piece of music (one voice part)

The red line is a reference line indicating the highest pitch that is written in the music. As seen in Figure 6, there are a few glitches in the detected pitches. However, the detected pitch is mostly correct as appears in the first figure of Test 2. By manually comparing the detected pitches and the sheet music, the correctness of this module is above 90% in this specific test case. Note that the music used in this test case has pitches shifting relatively quickly compared to average acapella music. Therefore, fewer glitches are anticipated when using with a more realistic input.

Additionally, since new chords are generated at half a second at their most frequent recalculation speed, the overall error caused by the pitch detection module should be minimal.

On ModelSIM

After creating a software prototype, the Pitch Detection subcomponent was then implemented in Verilog and tested within Modelsim. The Modelsim test bench is simple: when given a known signal from a .txt file, an output is generated and checked against the expected output in order to test validity.

Two input files are prepared for the test bench. They are the same file as the first two prepared for the MATLAB prototype test converted to appropriate hex number needed for Verilog Testing.

Unfortunately we didn't finish fully testing the pitch detection module.

Difference between Software Prototype & Verilog Implementation

There are two major differences. One is that the software takes in a pre-set input file, generate the outcome in one array, but the verilog implementation takes input in real time, and generate the output in real time. The other one is that the software prototype determines the frequency by windowing from this moment to 0.5 second forward, but the verilog implementation windows from counting back 0.5 seconds to this moment.

Upon implementing, The software prototype is more straight forward. The Verilog implementation is a little more tedious since the number of inputs needs to be exact when writing to and reading from the fft module, and that the number of bits needs to match.

Pitch Detector: Fallback Implementation

Jacqui De Sa, CK Ong, Zixi Liu

Since the implementation of sung pitch recognition could not be completed by the checkoff time, Chordination implements an alternative user interface simulating a piano keyboard. In this interface, seven buttons of the FPGA are used to represent the seven notes of a Western musical scale (C,D,E,F,G,A,B) with an additional button being able to apply sharps to notes (C#, D#, F#, G#,A#), allowing all twelve notes names of a classical western scale to be produced. Three switches of the FPGA were allocated to allow users to select any of these twelve note names to be the key of their song. Additionally, two switches of the FPGA were allocated to allow users to select which octave (0-7) to play a note in. The currently-playing user note name, selected octave, and key were displayed with three using 7-segment LED displays.

Tempo Subcomponent

Jacqui De Sa
Providing a user-selected interval for chord recalculation

Overview

When harmonizing a piece of music, a musician will often choose not to harmonize every single changing note of the melody -- instead, he or she will harmonize every measure or half-measure of music, changing the chords every several seconds. The tempo module simulates this by allowing users to select one of two tempos which determine a regular interval for chord recalculation. At each interval, the tempo module provides a pulse -- a recalculate signal -- to the Chord Calculator to signal it to calculate a new chord due to changing user input.

Implementation

User selection of tempo

Users are able to select one of several preset tempos through the use of the FPGA switches -- the current implementation of Chordination allows selection between only two tempos due to a need to allocate switches for use with other user inputs such as key and octave. After selecting a tempo, the user is then able to see the interval between chord changes reflected on a flashing LED on the FPGA display and change their tempo choice if their current selection is too slow or too fast for their taste.

Timing the "Recalculate" Signal

The tempo module contains a 32-bit register that serves as a counter that increments every clock cycle and ranges from 0 to the selected preset tempo parameter that the user selects. Upon reaching the preset tempo value, the recalculate signal is asserted and the counter is reset to 0; otherwise, the counter increments until reaching this tempo value.

Testing Strategy

The testing for the tempo module consisted of two parts: a physical check to see the duration and change of the LED representing the "recalculate" signal, and ModelSim testing to determine whether the recalculate signal was indeed being produced regularly.

Chord Calculator Subcomponent

Jacqui De Sa

Calculating new chords from the current note and chord history.

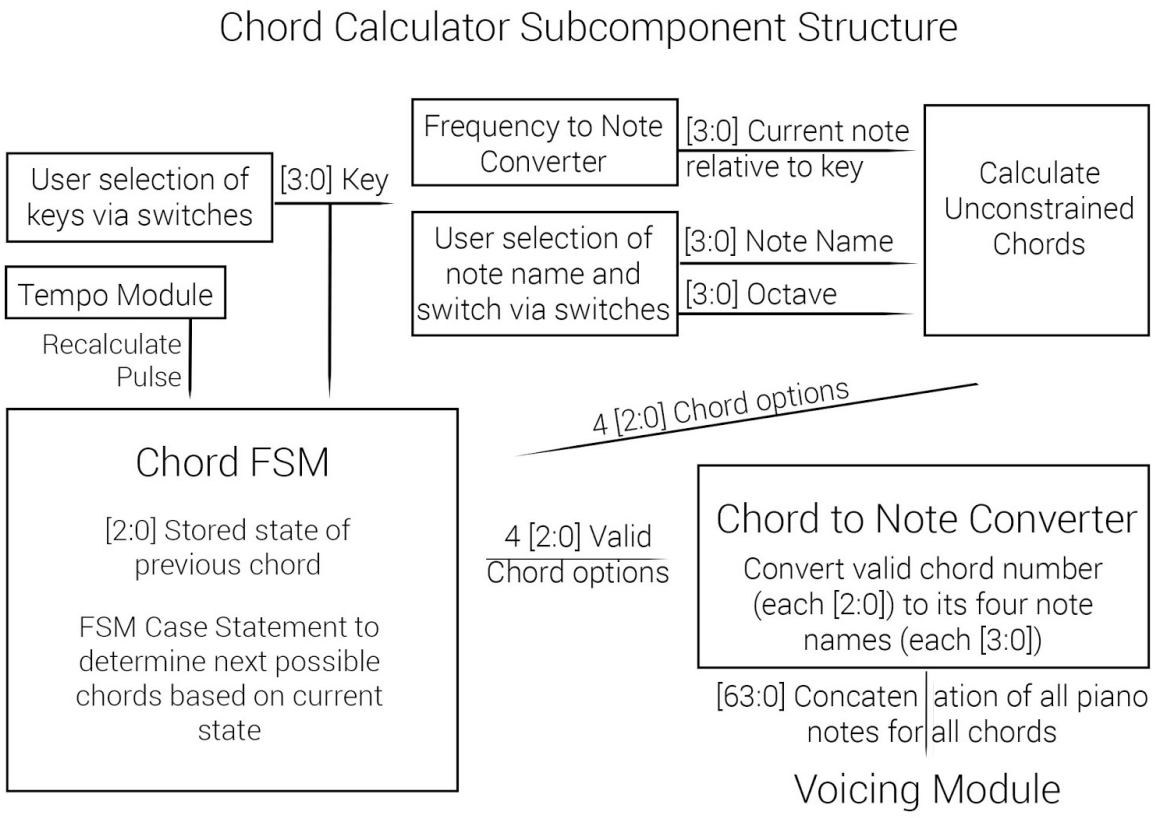


Figure 7: Overview of the Chord Calculator Subcomponent

Overview

The chord calculator module determines the next set of possible playable chords. Generating these possible chords depends on certain inputs: a representation of the currently playing note, a history of previous chords played, the key the song is in, and information on when chords should be recalculated. After calculating the set of possible chords, the Chord Calculator passes this information to the Voicing Module so that chords can be selected and individual notes can be assigned to different Voice Parts.

Theory of Implementation

Representing the Currently-Played Note: FreqtoNote Module

In order to represent the currently-played note, the Chord Calculator Module takes in input from the Pitch Detector module and converts it to a form which it can use.

In the intended implementation of Chordination, the Pitch Detector module consisted of a voice-input interface containing logic to lock-on to the frequency of the user's voice. In order to accommodate this voice-input interface, Chordination supports conversion from frequencies to musical notes in the form of the Frequency to Note submodule. This conversion was implemented by taking an existing frequency chart converting music notes to frequencies (link here) and mapping a continuous range of frequencies to one of the notes on the twelve-tone scale. An alternate implementation, which approximates the note a frequency corresponds to through a formula, was initially tried but discarded since the mapping implementation was both easier to implement (since the formulaic approach requires exponentiation and floating point arithmetic in Verilog) and more efficient.

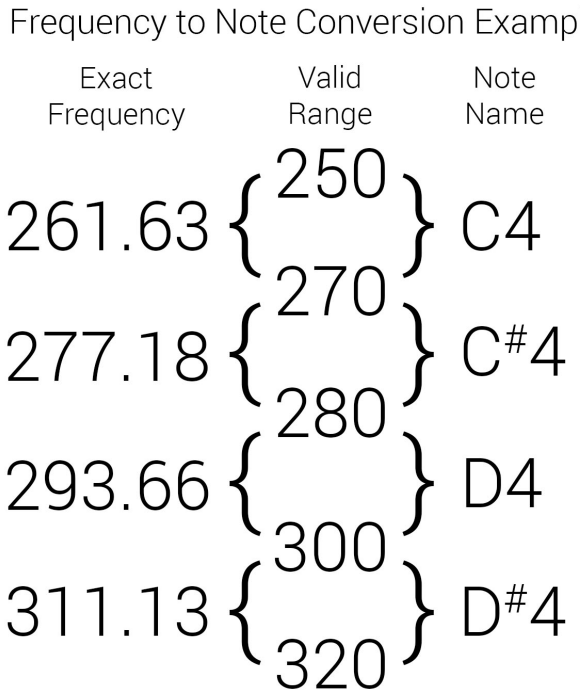


Figure 8: How to convert frequencies into note names

In the final implementation of Chordination, the Pitch Detector Module functions through the implementation of a piano -- a set of buttons representing an octave (C through B), a button allowing for notes to be sharp or natural, and a set of switches to set a note within a given octave. Since this implementation does not require a conversion into frequency, the final version instead takes input from the piano and saves which of the notes on the twelve-tone scale has been selected through the current configuration of buttons and switches.

Determining all possible (unconstrained) chords

After the current note has been determined, the Chord Calculator Module then determines any possible chords that can be formed using the current note. A chord is composed of four different notes: the "root" of the chord, forming its base, the "third" of the chord, which is four half-steps away from the root, the "fifth" of the chord, which is seven half-steps away from the root, and the "seventh" of the chord, which is eleven half-steps away from the root. Possible chords for a currently-played note are determined by figuring out which chords are produced by placing the currently-played note in each of these four slots.

Constraining possible chords: Chord FSM

After determining which chords are possible considering the currently-played note, the Chord Calculator module then constrains which of these chords can next be played by applying the rules of classical music theory chord transitions to constrain chord output depending on a history of previously-played chords. After outputting a set of valid chords to the Voicing Module, the Chord Calculator module then receives back information about which chord the Voicing Module has ultimately selected, and stores this information in its history.

The Chord Calculator selects which chords are valid (or not) by representing the transitions between chords allowed in Classical Music Theory as a Finite State Machine, where the currently playing chord is stored as the state. Transitions between chord states are enacted first by determining which movements are valid from the current chord state and then declaring the output between the unconstrained possible chords and the chords constrained by classical music theory as valid chords.

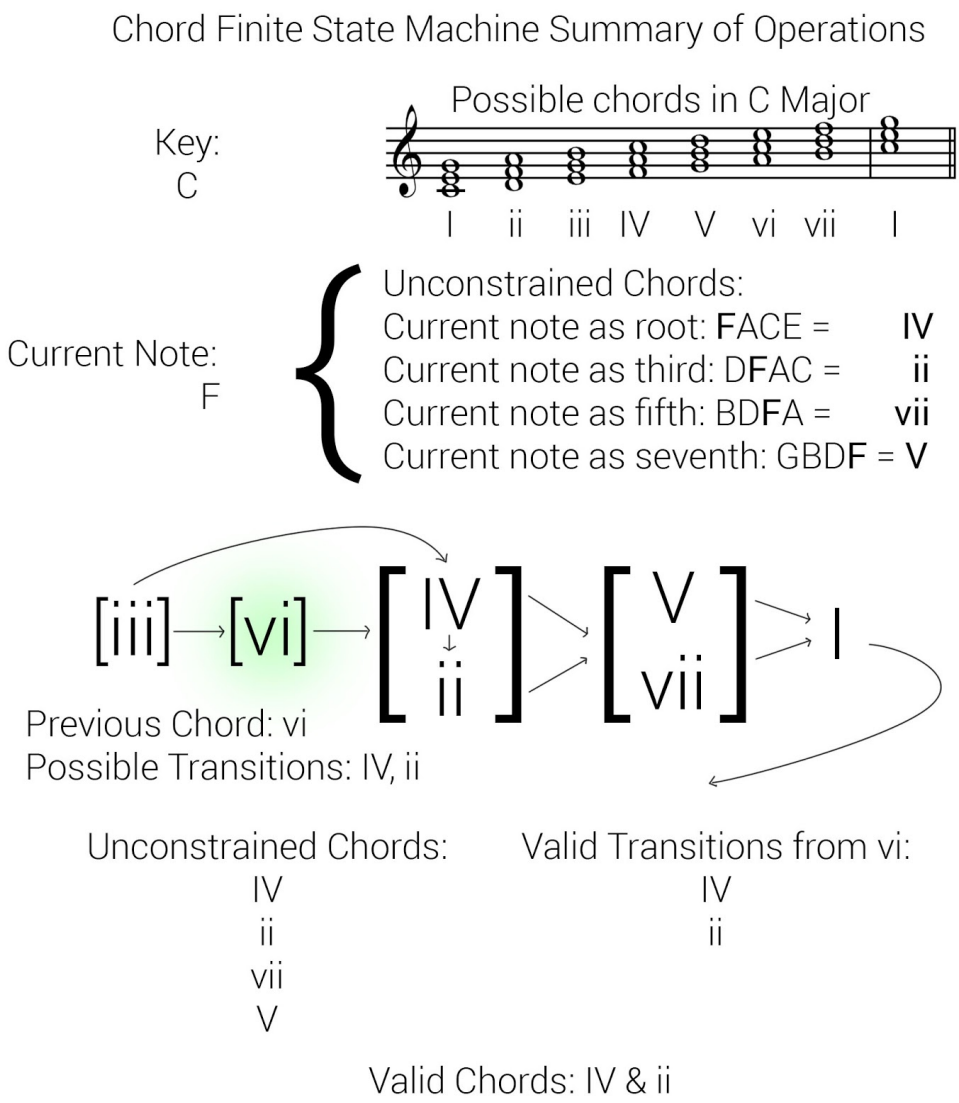


Figure 9: How the Chord Calculator FSM works and chord transition diagram

Converting from Chords to Piano Notes: ChordtoPianoNote Module

After determining the set of possible chords, the Chord Calculator converts each possible chord to a set of piano notes for the Voicing Module to examine. Since the Chord Calculator module internally represents selected chords by their root (in the way they are understood in classical music theory) this part of the Chord Calculator module takes in the current candidate chord and information on what octave the currently-played note is in in order to calculate the piano note corresponding to the root of the chord as follows: [insert equation here]. After finding the root of the chord, the third of the root is always four-half steps away, the fifth of the chord is seven half-steps away, and the fifth of the chord is eleven half-steps

away. The ChordtoPianoNote Module calculates these notes as explained and then exports them to the Voicing Module.

Timing and Clock Cycle Concerns

Since the Chord Calculator subcomponent mainly works through combinational logic, it operates entirely within one clock cycle. However, since Chord Calculator module only recalculates when signalled by the tempo module through a pulse (a “recalculate” signal), the Chord Calculator module only operates on the “recalculate” cycle.

Testing Strategy

In order to verify the correctness of the Chord Calculator subcomponent, individual submodules were tested, then combined together to see if a valid input to the system would still produce a valid output. Since the Frequency to Note Module (FreqtoNote) serves only to map all possible frequencies to piano notes, there was not much testing possible to do within the scope of this module.

Unconstrained Chord Testing Strategy

In order to to test the base case, the Unconstrained Chord test examines whether the note C (index 0) in the key of C (index 0) will produce valid and correct outputs for the representation of the current note as the root, third, fifth, and seventh of the chord (when calculated by hand, these outputs should be I, 6, 4, and 2, respectively). The test also examines the effect of invalid note outputs on the unconstrained chords (such as when the C# is played as a current note within the key of C). It then goes on to test valid notes played within the base key (C), since it may have been possible that an indexing calculation relating to adjusting the currently-playing note relative to the key it is in may occur for keys other than the base case. After calculating this, it then tests unconstrained chords in the general case of keys other than C and with currently-playing notes that are valid within the key.

Example of Unconstrained Chord Testing Strategy

```
//TEST 4: Testing for a valid note input other than
the base case but in the base-case key (C)
        state = 1;
        key = 2; //key of D
        note = 6;  //note is F#
        //outputted chords should be 3, 1, 6, 4

//TEST 5: Testing for a valid note input other than
the base case but in the base-case key (C)
        state = 5;
        key = 0; //key of C
        note = 4;  //note is F#
        //outputted chords should be 3, 1, 6, 4
```

Chord FSM Testing Strategy

The Chord-selecting Finite State Machine (ChordFSM) was tested by partitioning possible inputs into cases and testing inputs within each case. Similar to the Unconstrained Chord Testing Strategy, the Chord FSM Testing Strategy tests the base cases of playing the note of C (index 0) in the key of C (index 0) starting from a I chord (enabling all possible chord

transitions) in addition to testing edge cases and invalid inputs (notes outside of a key) and then testing general case inputs to expect general case outputs.

Successes, Failures, and Future Steps

Unlike the other three main components of Chordination, the Chord Calculation Subcomponent did not produce any major setbacks or require vast changes of implementation. One aspect that did change and adapt over time was the need to interface the Chord Calculation subcomponent with changing needs for inputs in the Voicing Module and changing outputs from the user note input, which meant that time was spent writing modules such as the Frequency to Note conversion module that could have otherwise been spent implementing other aspects.

As a future step, it would be interesting to include minor key chord progressions as an option within Chordination. Adding minor key chord progressions would require implementing a different finite state machine to represent the rules of classic minor key chord transitions. Alternatively, minor key chord progressions could be more easily implemented through use of representing a user's minor key note input in its relative major key, thus allowing the Chord Calculator module to transform its inputs in order to avoid implementing an additional FSM.

Voicing Subcomponent

CK Ong

Selecting an optimal chord from a set of valid chords & allocating notes to voice parts.

The Voicing Subcomponent is responsible for finding out exactly which notes to play (as opposed to the Chord Calculator, which finds out which sets of notes are possible to play considering the currently-playing note and previous chord). As described in Figures 10 and 11, the Voicing Subcomponent takes in chords from the Chord Calculator in the form of grouping of notes. The Voicing Module uses established classical music theory rules in order to constrain the possible piano notes which could be played for any set of chords provided by the Chord Calculator. These rules include rules on jumps (how close or far away two consecutive notes are for the same voice part), motion (whether notes move in the same direction or different directions and to what degree) as well as voice crossing and overlapping (whether a higher voice part sings lower than a lower voice part and vice versa), and the distance in pitch between parts. Following these rules allows voices in chorale music to remain independent and the harmony to sound full.

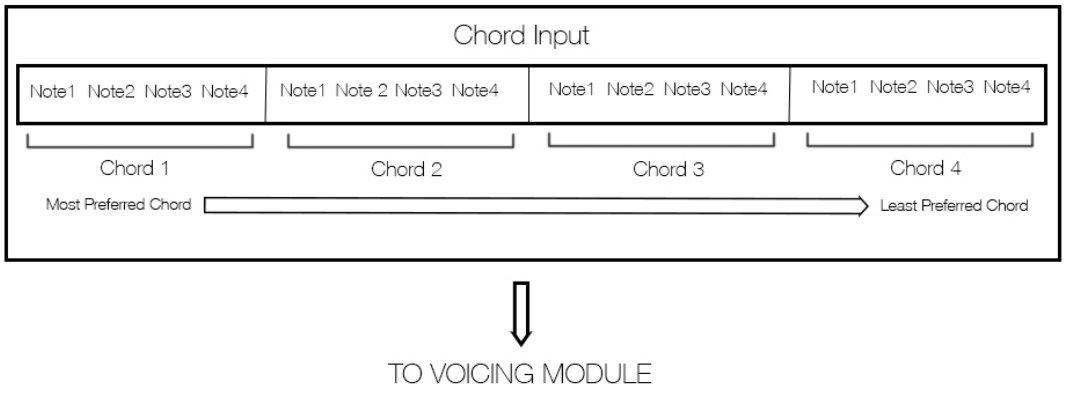


Figure 10: Visualizing the input into the voicing module

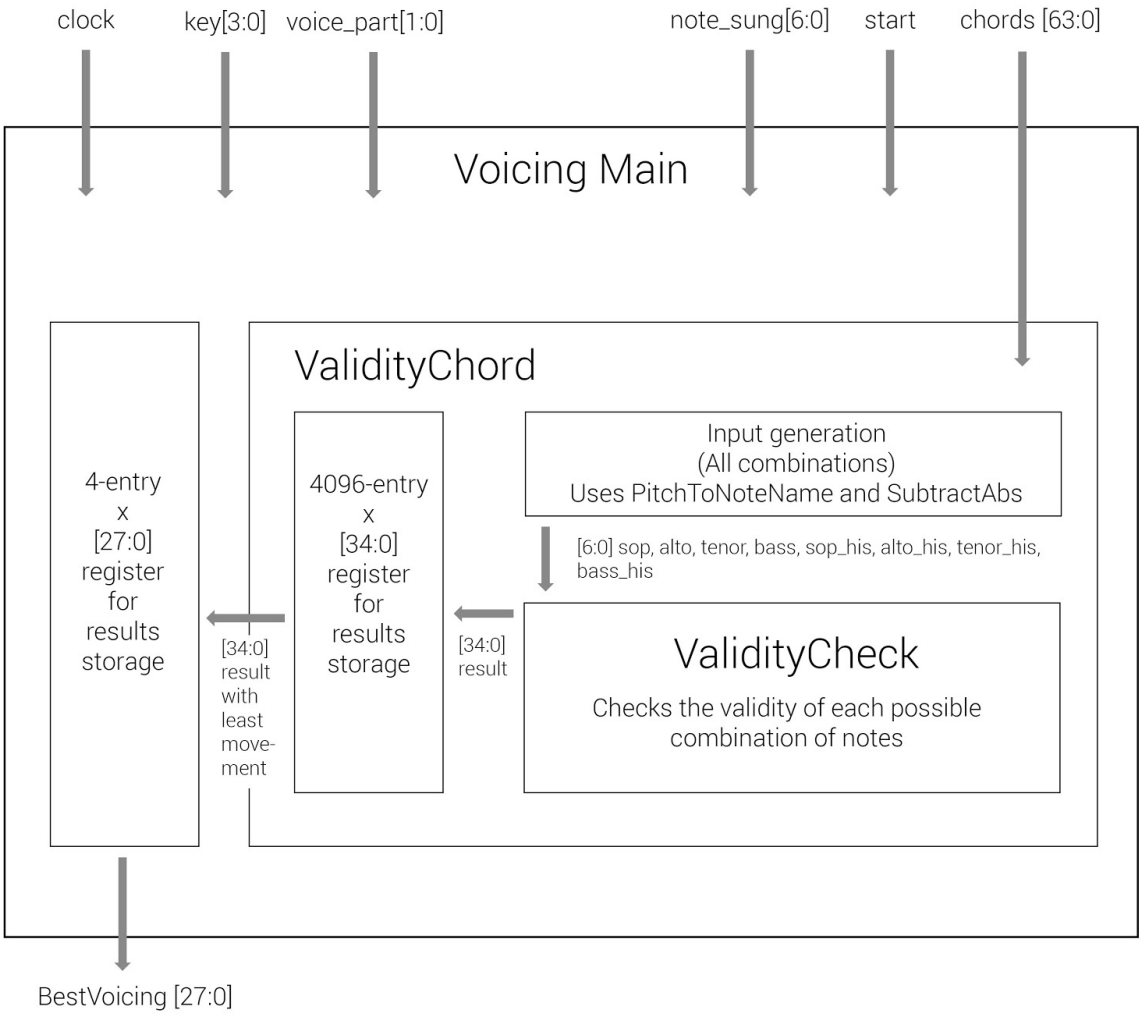


Figure 11: Overview of the Voicing Subcomponent

Another important constraint to take into account is the user's voice part, which determines how many synthesized voices will sing higher and lower than the main voice. The user enters their voice part via switches on the labkit. For a four-part harmony, the voice parts are usually soprano, alto, tenor and bass, in decreasing order of how high each part sings. This constraint is necessary since if that the voicing module always assumes that a soprano is singing, if a bass sings into the devices all the harmonized notes will be even lower than the existing low bass line and as a result, unrecognizable.

The voicing module uses these rules to calculate the most ideal combination of notes to play, which is any combination that does not break any of the voicing constraints imposed by classical music theory.

SUBMODULES

VoicingMain

VoicingMain is the top level module that receives chords from the Chord Calculator and outputs the optimal voicing. It does this by having an instance of ValidityChord and feeding in each chord as an input, storing each of the four outputs in registers.

ValidityChord

This module takes a chord (made of four note names) from the VoicingMain module and using four different counters, feeds every single voicing possibility into the ValidityCheck module. There are many manners in which the notes could potentially be arranged. Each voice part could move to the root, third, fifth or seventh of the new chord, which makes for four possibilities per voice part. Additionally, in music theory, excessive movement of a vocal line -- large jumps from one note to another -- is highly discouraged. Therefore each part can only move to any of the four chord notes directly above it or below it, adding a factor of two to the number of possible voicings. Figure 12 illustrates the 8 possibilities per voice part.

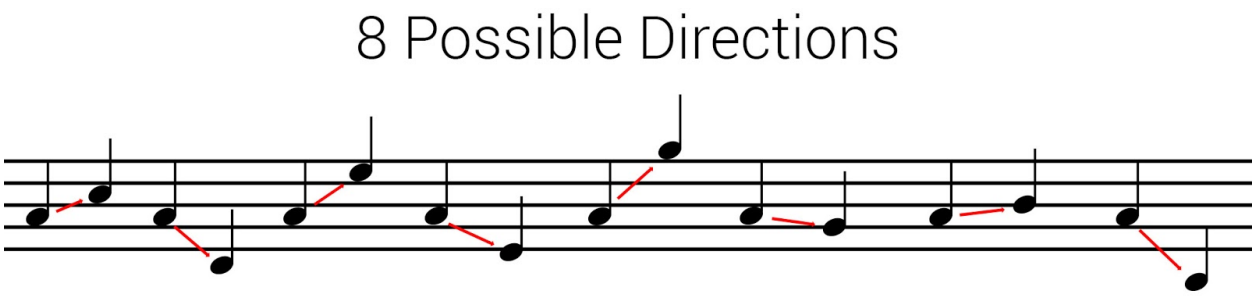


Figure 12: Different ways a note (A) can go with C major 7 chord (C, E, G, B)

In order to produce all possible inputs (for the combination of all voice parts), all possibilities need to be iterated through. Four counters, one for each voice part, count from zero to seven, which accounts for eight possible moves. The logic is then set up such that the each higher voice counts a factor of 8 slower than the lower voice. The bass counter counts up every clock cycle, the tenor counter every 8, the alto counter every 8^2 and the soprano counter

every 8^3, just like one would see in a nested “for” loop. As the numbers change, the input to ValidityCheck change and the output is recorded.

At each clock cycle, the output of ValidityCheck is stored in a 2D array. After all the values have been found and stored, a process is started that iterates through the results stored in the 2D array and outputs the piano note configuration with the least movement associated. If there is no valid voicing for this chord, zero is returned. “Done” is asserted when the process is completed.

ValidityCheck

The ValidityCheck module is the core of the Voicing module; it is responsible for telling whether or not particular voicing of a chord (passed into it) is valid. It also determines how much movement is present in this particular voicing, where movement is the summed difference between each part’s current note and potential next note:

```
movement = |sop_cur - sop_new| + |alto_cur  - alto_new| + |tenor_cur
                - tenor_new| + |bass_cur - bass_new|
```

This module checks the input’s adherence to several music theory voicing rules, including: voice crossing, voice overlapping, voice distance, parallels (unisons, fifths, octaves), hidden parallels, and the resolution of 7ths and the leading tone.

At the end of this process, ValidityCheck outputs a 35-bit value (shown in Figure 13) corresponding to the amount of movement concatenated with the input piano note configuration.

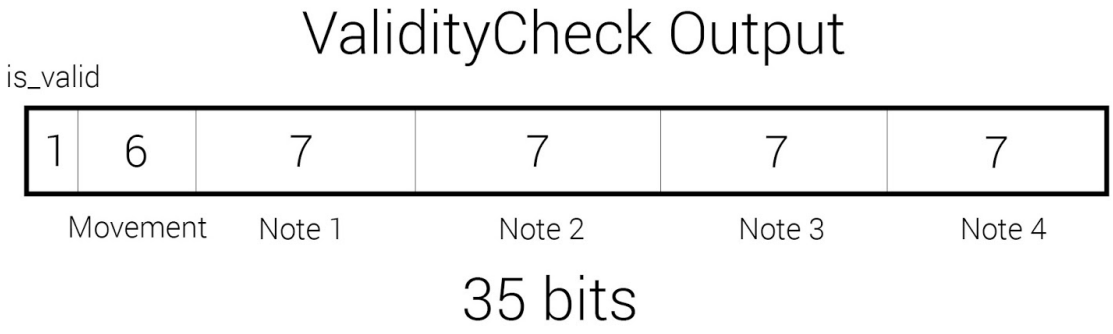


Figure 13: Visualizing the output of ValidityCheck

SubtractAbs

In order to do calculations on the distance between piano notes many times the absolute value of the difference between the piano notes needs to be taken (depending on which pitch is higher). To simplify the operation, the SubtractAbs module does exactly this. It takes in two 4-bit numbers and assigns the difference between the two to a signed wire to preserve the negative sign if it's present. Then, it assigns the output to the difference if it's greater than zero and assigns it to the negative of the difference if it's less than zero.

PitchToNoteName

PitchToNoteName is a module that takes in a 7-bit pitch (piano key index) and turn it into a 4-bit note name (note from C to B). It does this by implementing a mod-12 function and taking the modulo 12 of the pitch. The result is the note name. One of the challenges to making this module was being able to do a modulo 12 operation, since the synthesis tools only support the modulo operator with powers of two. In order to do modulo 12, the following equation was used:

```
mod12 = input - (input*int(1024/12) /1024)*12
mod12 = input - (input*85 /1024)*12
```

Voicing Subcomponent Testing Strategy

The Voicing Subcomponent was subject to extensive testing, especially the brain of the module: the ValidityCheck submodule. Therefore, two types of tests were performed: one type for making sure the ValidityCheck module was always correct and one that made sure the module as a whole returned appropriate values.

ValidityCheck Module Testing Strategy

The ValidityCheck module was tested by attempting each of the voicing rules and determining from simulation whether or not the voicing was considered valid. Several control test cases were created, cases where the module should return that the voicing is valid. Then, each of the voicing rules are broken individually to determine whether the module detects these individual types of voicing errors.

```
// ValidityCheckTest
sop = 48; // C5
alto = 43; // G4
tenor = 40; // E4
bass = 24; // C3
sop his = 47; // B4
alto his = 43; // G4
tenor his = 38; // D4
bass his = 31; // G3
seventh label = 4; // No 7th
leading_tone = 11; // B is the leading tone
(because the chord is G)


// G7 -> C wrong (7th doesn't resolve down)
alto_his = 41; // F4
seventh_label = 1; // 7th is in the alto
#100;

// G7 -> C correct
alto = 40; // E4
tenor = 36; // C4
#100;
```

Below is an example where the module is being tested for the seventh of the chord resolving down:

VoicingMain Module Testing Strategy

A similar approach was taken for the module as a whole. A pre-made note history as well as possible chords were initialized as inputs in the test bench. The optimal voicing outcome was calculated by hand and compared to the output of the Voicing Module. The other case that needed to be tested was one where there is no valid voicing, in which case the output should be whatever is currently in the note history (does not change).

Challenges

The main challenge with the voicing module was the synthesis time required in its initial implementation. The previous approach to the ValidityChord module was to implement it with as few clock cycles as possible. To do this, we considered that out of the 4096 voicing possibilities, only 767 of them have a greater than zero probability of being valid, so the ValidityChord module contained a case statement with 767 cases to feed into ValidityCheck. This setup was entirely fine during testing and development using ModelSim. However, when it came time to finally synthesize it for the labkit it took three hours to compile. This posed a significant roadblock in the development of Coordination, since our ability to iterate quickly was constrained by our ability to compile and test on the labkit.

This problem was finally resolved after reading ISE's console output as it synthesized. It seemed that ValidityChord was taking by far the longest to synthesize because the synthesizer was treating the case statment like a Finite State Machine and was optimizing its encoding. We were able to optimize the compile time by changing this implementation to an alternate implementation that Verilog could better understand and optimize. Instead of hardcoding 767 inputs into a case statement, the above method of iterating through all 4096 possibilities was used and compile time dropped from three hours to three minutes. While it is true that the latter method takes more than four times as many clock cycles as the former, even 4096 clock cycles is a small amount of time compared to the 27 million clock cycles that happen in one second, so performance is not affected. On the other hand, iteration speed was drastically increased.

Synthesis Subcomponent

Zixi Liu

Pitch-shifting the user audio input to pitches generated by the voicing module.

The synthesis module takes the four target pitches generated by the voicing module and outputs the harmonizing chord signal. The output signal is then fed into the ac97 module,

which converts the digital signal to an analog signal. The generated chord is played by a speaker attached to the FPGA. To achieve the desired output, the synthesis module divides the task into 3 steps: reorganizing the order of the FFT (Fast Fourier Transform), computing the inverse Fourier transform of the reordered FFT, and adding/zero-padding.

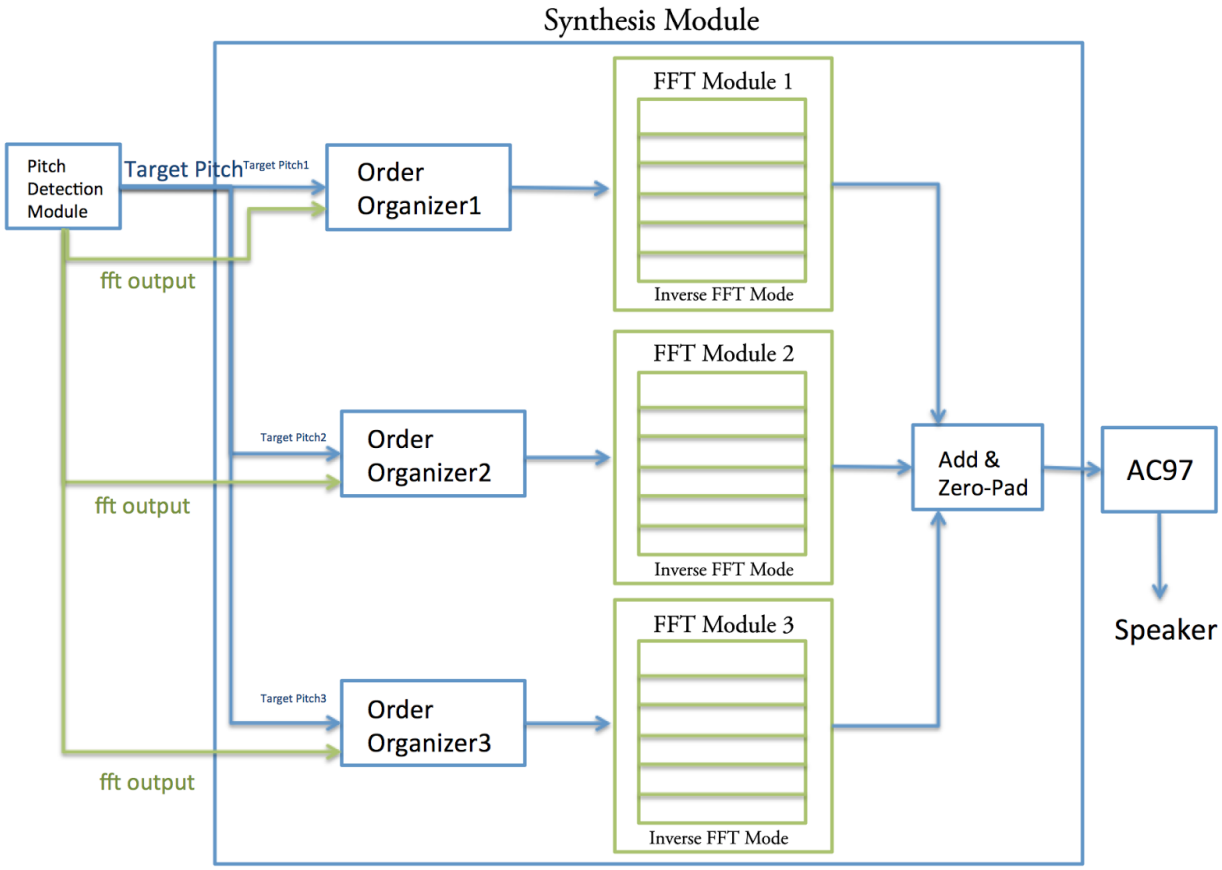


Figure 14: Overview of the Synthesis Subcomponent

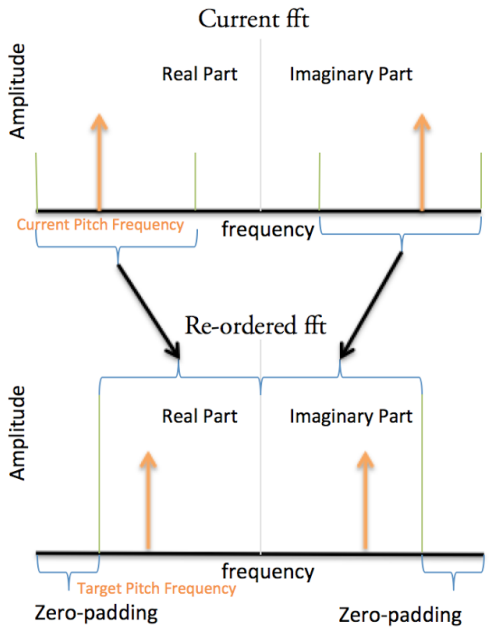
Step One: Reorganize the Order of the FFT (Fast Fourier Transform)

To shift a pitch, we re-order the numbers of the Fourier transform of the input signal, i.e. the digital signal of the user's singing voice. There are two possibilities in tempting to re-organize the numbers in the Fourier transform: when a target pitch is higher than input pitch, i.e. the user's singing pitch, and when a pitch is lower than the input pitch.

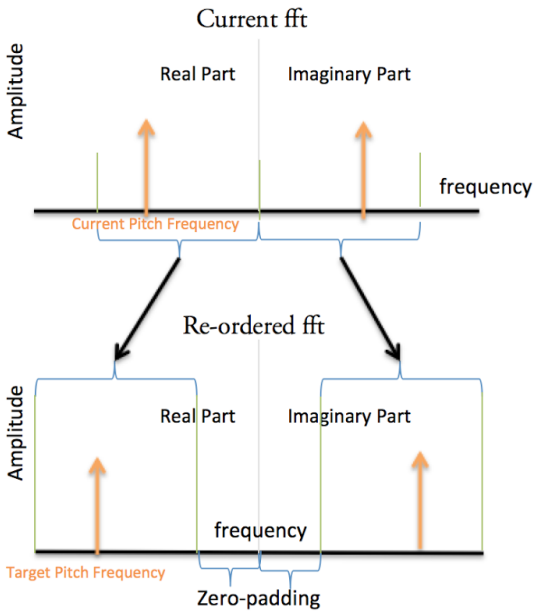
First of all, there are two parts in the Fourier Transform's real part: the positive part, and the negative part. They are mirror images to each other when plotted.

When the target pitch is higher than the input pitch, for the positive part of the Fourier transform, we would zero-pad the beginning of the Fourier transform, and then fill the rest of the spots with the Fourier transform of the input data until the positive part is filled, such that the maximum value associates with the target pitch. For the negative part of the Fourier transform, we would leave the first input Fourier transform's negative part that is the same number of the zero-pads in the real part, and take the rest of it. Then we would fill the rest with zeros. As a result, the re-organized Fourier transform will also have real and imaginary parts that are mirror to each other, and has maximum values associated with the target pitch frequency.
When the target pitch is lower than the input pitch, we will reverse what we do for the other case. For the real part, we will take the last parts of the input Fourier transform's real part, and then zero-pad to fill the rest; and for the imaginary part, we will first zero-pad, and then take the first part of the input Fourier transform's imaginary part, such that the maximum value is associated with the target frequency.

Target Pitch Frequency > Current Pitch Frequency



Target Pitch Frequency < Current Pitch Frequency



Since there are three target pitches at any given time, if any, we have three order-reorganizers and three instances of the FFT module, which will compute the inverse Fourier transform of the data. The order-organizers will send rearranged, zero-padded the data one by one to the according FFT module.

In order to feed the data in the corrector order to the FFT modules, the order-reorganizer needs to store the FFT module output in memories and select where to read from the memory and feed it to the FFT modules. Each order-reorganizer has its only local memory that is used to store the input FFT output and to read from the local memory. These memories are also *bram*s that are instances of the mybram module that comes with the labkit. Three different local memories are necessary although the same information is stored because the three sub-parts functions simultaneously in time, and one can only read from a bram at one address at any given time.

We figure out to re-order the fft by following the following steps:
Step 1, figure out the offset number that we use to zero pad. The offset number is found by finding out the difference between two adjacent bins in the frequency spectrum, and multiply that number by the difference in pitch. The difference between two adjacent bins in the frequency spectrum can be found by the multiplying the number of bins in the real part of the fft spectrum and the sampling frequency divided by 2. That is because, the maximum possible frequency in the frequency spectrum is the Nyquist frequency, which is the sampling frequency divided by 2.
Step 2, if the pitch is either smaller that 130 hz, or 1050 hz, we determine that, the detected frequency is not in reasonable human singing voice frequency range and choose to not modify, or shift it.
Step 3, if the pitch is within the normal range, determine if the detected pitch is less than, or greater than the target pitch. If it is less than the current pitch, we create our new FFT in the following order: offset number of zeros, original FFT data from 1 to half of it minus the offset number*, the original FFT data from half of it plus the offset number*, and offset number zeros again. If the detected pitch is greater than the current pitch, we create our new FFT by the following order: original fft from the offset number to half of it*, offset numbers of zeros, another offset numbers of zeros, original fft from half to the offset number before the end*.
Note that the re-ordering is very tedious, one must make sure that the total length of the re-ordered fft equal to to total length of the original fft. To do so, we need to keep track of if the total length of the original fft is even or odd to determine exactly how many zeros to pad into our re-ordered fft. With even one extra or less zero, the fft might not be able to generate a valid output, specifically when we just started running the module.
(*) Let me explain what I mean by the original fft from a to b. The original fft is an array of data. When I say the original fft from a to b, I mean the sub-array that consists from the a-th to the b-th data in the original fft.

Step Two: Compute the inverse Fourier transform of the reordered the FFT

The FFT module is used to compute the inverse Fourier transform (iFFT)of the re-ordered data in the frequency dimension (the Fourier transform). By computing the iFFT of the re-organized data, we will get the user's singing voice shifted to the target pitches.
The three identical instances of the FFT modules computes the shifted voice signal simultaneously. Upon finish computing the iFFT of the re-ordered FFT data, we have three separate parts of shifted voice being output from the FFT module by clock cycle.

Step Three: Add and Zero-Pad

To integrate the three shifted voices together, we simply add the three signals of shifted voice together. To output the data to a speaker, we need to zero-pad the signal because when taking in the signal, we have downsized the signals by 4. That is, after outputting each added data, we need to send 3 zeros in a row to the ac97 module before we can output another added data.

Testing Strategy

Within MATLAB

We first used generated a shift_pitch function within MATLAB that functions in exactly the same way as described above, except that it is not in real time. We pre-generated a sine wave that is a 262 hz (middle C on a piano) pure tone when played on a speaker. The sine wave file is used as an input to generated an output that is shifted to another frequency. Pure tones pitches can be easily found by computing the Fourier transform and find the frequency that's associated with the maximum value/amplitude. By detecting the pitch as described above, we were able to verify that the function indeed generates the correct result.
Then, we recorded us singing for 1 minute and attempted to shift the entire minute to 262 hz (middle C on a piano). We found that the output accompanied with some noise that is of the sample amplitude scale as the ideal output file.
We later realized that it was because we did not handle the overlapping in windowing very well. The noise was significantly removed by simply using a sine window or Gaussian window instead of a square window (since we are simply using the 0.5 second most recent data).

Within Verilog

Unfortunately the process of the implementation of the synthesis module stopped after we have only set up the necessary components of the module. That is when we shifted back to improving the software prototype to remove the noise. When we have solved the noise issue, we decided that we do not have enough time to complete the synthesis module because in doing that we need to figure

Challenges

On MATLAB

We prototyped the described algorithm in MATLAB. We had three separate files, one taking input data and shift the input to a preset target pitch frequency, one adapted from the pitch detection module prototype, and one that integrates them together to generate the outcome. To simulate the input, we preset targeted pitch frequencies and inputted them to the integration file. Although the voice is shifted to correctly to the target pitch, unfortunately there is noise accompanying with the generated output.

One possible explanation is that we have only considered the real part of the Fourier transform of the input data. The information in the imaginary part is being ignored. The imaginary part of the fourier transform is an important part, as we learnt later on, that it contains the phase information of the data.

Sound Synthesis: Fallback Implementation

Jacqui De Sa, CK Ong, Zixi Liu

Since the pitch-shifted Sound Synthesis approach was unable to be implemented within the timeframe, Chordination uses a square-wave synthesis to output pitches. This alternate implementation interfaces between the four piano notes outputted by the Voicing Modules which correspond to indices of notes on a piano (0 through 96 -- a 7 bit input), converts these indices to their corresponding musical frequency, then generates a square wave corresponding to the chord needed.

Convert Piano Notes to Frequencies

After receiving the final selected notes (in 7-bit piano note index form) for the new chord, the Sound Synthesis Subcomponent converts these piano notes to frequencies through the Note to Frequency (NotetoFreq) Module through mapping each piano index to a preset frequency as described by a note to frequency conversion chart (Phy.mtu.edu, 2014). These frequencies were stored as 12-bit values (since frequencies range from 0 to approximately 8,000 through 8 octaves), and then passed to the Sound Generator Module for synthesis.

Sound Generator Module

The Sound Generator module takes in the 12-bit frequency values for the soprano, alto, tenor, and bass lines. It initializes four 21-bit regs representing counters for each voice part and single bit regs representing when a tone should be sounded for each part. At each clock cycle, these counters increment until reaching the frequency value for their respective voice parts. Upon reaching the desired frequency, the tone for that part is sounded -- set to one --

and the counter is reset. A 20-bit reg, representing the chord tone, was set to the sum of the individual tones for each of the four voice parts as described above. This tone was then outputted to the ac97 for sound output.

Conclusion

Jacqui De Sa

Lessons Learnt

Integration vs Progressive Iteration

In addition to the challenges and proposed solutions discussed within the individual subcomponents, there were some overarching challenges that caused project setbacks. One of these challenges was the decision to focus on horizontal subcomponent integration, where all subcomponents would be combined together at one step, rather than a progressive iterative vertical integration, where the project would first be implemented through a bare-bones, minimalistic approach with limited functionality which could then be added to. During the week intended for group integration, we spent most of our time using the former approach. However, since some subcomponents encountered setbacks which affected compile time or overall functionality, it would have been better instead to start with the latter integration approach, as time was spent attempting to interface with subcomponent approaches that ultimately had to be abandoned. After switching to the vertical integration approach, we were able to produce a working product much more quickly and then focus on adding new features without worrying that the product may be unable to come together.

Future Extensions

In the future, we would like to continue the Chordination Project and either implement the original voice input and pitch-shifted output that we originally suggested or provide an interface for MIDI synthesis as an output channel. This interface would allow users to employ pre-existing MIDI instrument synthesizers to save and modify their Chordination improv harmonies and have them be played by a variety of pre-exisitng MIDI instruments. This functionality would allow Chordination to be even more accessible to composers and musicians alike.

Bibliography

1. Phy.mtu.edu, (2014). Frequencies of Musical Notes, A4 = 440 Hz. [online] Available at: http://www.phy.mtu.edu/~suits/notefreqs.html [Accessed 17 Nov. 2014].

2. Write-music.com, (2014). Music Software Ludwig: Automatic Arranging, Composing and Songwriting. Royalty Free Music.. [online] Available at: http://www.write-music.com/ [Accessed 17 Nov. 2014].