SNGH's Not Guitar Hero

Rhys Hilter Ruth Shewmon

December 12, 2007

Abstract

We have developed a game, *SNGH*, that teaches users how to play real chords on a standard electric guitar. SNGH is unique because it can successfully identify notes played over the guitar's full frequency range of 82Hz to 1175Hz. Another important feature that sets it apart is the ability to detect chords and their transposes in real time. The game teaches the player chords by stepping them through a preprogrammed sequence, displaying the tabs of the requested chord as well as what is currently being played on the guitar. SNGH also provides quantifiable measures of progress, so players can watch as well as hear themselves improve.

Contents

List of Figures

1 Overview

SNGH is designed to make learning the guitar fun and easy. Only three external devices are required: an electric guitar, a pair of headphones, and a screen.

The game starts off by buffering incoming audio and putting it into an FFT. SNGH uses a 32768-point FFT, which is filled with 4096 audio data points fit to a Hann window and padded with zeros. The output of the FFT is then buffered for display on the screen and detection of peaks.

The peaks are sorted through to find which ones' frequencies are integer multiple of others. If a peak is located at a multiple frequency of another peak, it is not a fundamental. The few peaks that aren't multiples of any other peaks represent the fundamental frequencies of the vibrating strings on the guitar.

The list of vibrating strings is monitored for transient entries, which are discarded. When a stable list of strings has been found, SNGH looks for chords involving that set of frequencies. If it finds a match, the chord is displayed on the screen and compared to the requested chord. Throughout the project, notes are represented as 1's in 12-bit vectors that represent the 12 musical notes in an octave. For example, 12'b100000000000 is a C, 12'b010000000000 is a C sharp, and 12'b100010010000 is a C major triad.

There are two visual display modes for the game. A full-screen FFT debug mode shows the FFT output in real time with peaks and fundamentals highlighted. Markers along the bottom of the screen are helpful in tuning the guitar. The main display mode is a game screen, which contains the fingerings and names of the requested and played chords, a timer bar, a miniature display of the FFT ouput, the name of the next chord in the sequence, and an area for judgement to be displayed after a chord is detected.

2 Description

SNGH's design is broken up into two sections; the frequency-space audio processing written by Rhys Hilter, and the time-space processing and game developed by Ruth Shewmon.

2.1 FFT Handler (Rhys)

The `FFT Handler` takes care of all the audio interfacing required to produce a high-quality FFT. This includes looping the incoming audio back to the user's headphones or speakers, buffering the incoming audio, running the audio through a Hann window, passing it along to an FFT, and buffering the output of the FFT.

2.1.1 Input Buffering, Windowing, and FFT Control

The AC97 codec used for audio input provides 48000 samples per second, so the `FFT Handler`'s 4096 sample input buffer holds about $1/12^{th}$ of a second of sound. However, the audio is run through an FFT about 48 times per second, every time 1024 new audio samples come in.

If we were to use a 4096-point FFT, we'd get about 12Hz resolution ($48000/4096 \approx 11.72$), which is about 2 half-steps at the frequency of the lowest string. In order to detect the frequencies of the low strings reliably, we decided on a 32768-point FFT, which provides resolution of about 1.46Hz, which is about 3 bins for the lowest notes.

To fit the 4096 sample long input data into the 32768-point FFT, we applied a 4096-long Hann window to the data and padded the rest with zeros to reduce ripples in the output. The windowed data was then fed into the FFT.

2.1.2 Output Buffering and Scaling

The `FFT Handler` is responsible for providing data to two modules - the `Peak Detector`, which is the next stage in the audio processing chain, and the `Ruth Conveniencer`, so the FFT can be displayed on the screen.

When the FFT is done processing the audio, it unloads into the buffer. The FFT's output comes in real and imaginary components for each frequency bin, but all that matters for the chord detection is the magnitude. The complex pairs are squared, added, and square-rooted as they come out of the FFT.

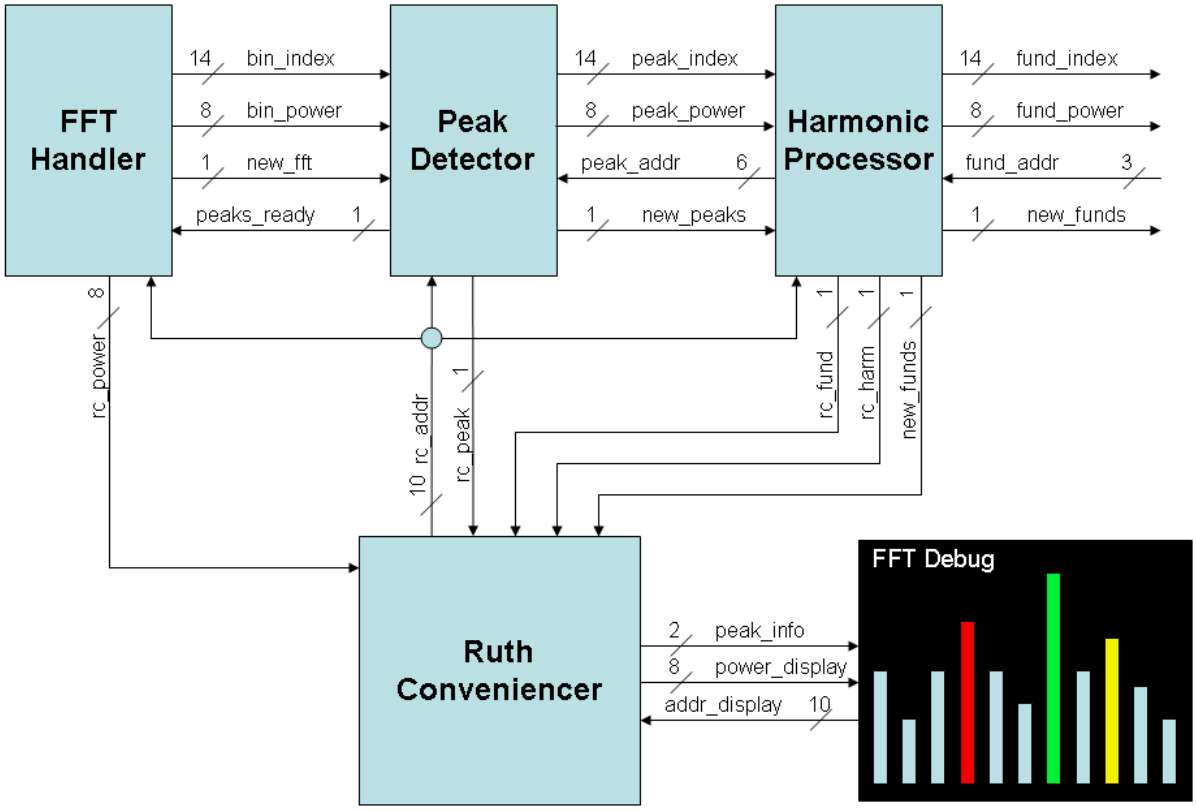

Figure 1: Block diagram of the frequency-space audio processing

When sending data to the `Ruth Conveniencer` for displaying on the screen, the `FFT Handler` sends the first 1024 data points (for the 1024x768 display) with constant scaling, so the display won't jump when the magnitude of the signal changes drastically. The audio processing path needs to be sensitive to changes in the amplitude of the signal, so scaling is required for that path. As the magnitudes of the bins are unloaded from the FFT and stored in the buffer, a register keeps track of the largest value, which is then used to scale the data when it's unloaded from the buffer and sent to the `Peak Detector`.

2.2 Peak Detector (Rhys)

The `Peak Detector` takes the scaled output of the FFT and finds peaks in it. This is an important step for reducing the amount of data that needs to be processed by the rest of the system.

To be considered a peak, the signal must decrease after the current data point and must have been increasing since the last peak. Also, the current value of the signal must be at least $1/16^{th}$ of the maximum value of the signal (to eliminate noise), and at least as large as the data points within 12 samples either direction (to filter out the side lobes of peak caused by the Fourier transform).

The first 64 such data points are declared peaks and their locations and amplitudes are stored in registers to be passed along to the `Harmonic Processor`.

2.3 Harmonic Processor (Rhys)

The `Harmonic Processor`'s purpose is to sort through the list of peaks and identify the ones that represent fundamental frequencies.

When a string vibrates, it vibrates at many frequencies; the fundamental, which fits 1/2 of a wavelength on the string, the 2^{nd} harmonic, which fits $2 * (1/2) = 1$ wavelength on the string, the 3^{rd} harmonic, which fits 3/2 wavelengths on the string, etc. Each string has only one fundamental, and lots of harmonics. Conveniently, the harmonics' frequencies are all integer multiples of their fundamental's.

To figure out which strings are vibrating, it's necessary to come up with a list of the fundamentals. A guitar has six strings, so should only ever have six fundamentals. The `Harmonic Processor` finds up to 8 so that transient false detections don't mess up the detection of real fundamentals in later stages of processing.

2.3.1 Building the Prime Harmonics Tables

The first step in grouping the fundamentals with their harmonics is to find peaks whose frequencies are prime multiples of other peaks'.

Starting with the peak with the lowest frequency, the `Harmonic Processor` multiplies its frequency by two, three, five, and seven, and generates $\pm 1/16^{th}$ error bounds for catching peaks with frequencies at approximately prime multiples of the base frequency's.

The `Harmonic Processor` then crawls up the list of peaks found by the `Peak Detector` searching for peaks with frequencies within those bounds. When a matching peak is found, it's added to the base peak's list of multiples.

When the `Harmonic Processor` has reached the last peak, it then looks for multiples of the second, third, fourth, etc. peak.

2.3.2 Building the Composite Harmonics Tables

Once the prime multiples of each peak have been found, the `Harmonic Processor` tries to find composite multiples of the peaks. It does this by looking at the lists of prime multiples. To find the fourth harmonic of a peak, the `Harmonic Processor` looks up what twice the peak is, then what twice that peak is, and stores it in the 4^{th} harmonic table.

2.3.3 Identifying Fundamentals

There are three methods employed for filtering the peaks into fundamentals, their harmonics, and noise.

The first method is to require that fundamentals not be harmonics of any other fundamentals. This method is fairly effective, but doesn't work very well against dissonant chords, which have peaks fairly close together. Only one peak that falls in the prime-multiples error bounds gets recorded in the table, so when peaks are close together, one of them often slips through the cracks.

The second method is to require that fundamentals have harmonics and be fairly loud. If a peak is missing its 2^{nd} and 3^{rd} harmonics, it must be at least 1/2 of the maximum volume in the FFT data set. If it has only one of those, it has to be at least 1/4 of the maximum volume, if it has both, it only needs to be $1/8^{th}$ of the maximum. This filter doesn't remove many peaks.

The third and most effective method of finding fundamentals requires that fundamentals not be approximately multiples of other peaks. This is slightly different from the first method in that it doesn't require peaks to be entered into the harmonic tables to be eliminated. It's effective against dissonance because when two peaks are both close to being multiples of another peak, it eliminates both of them, not just the one that happens to get entered into the table.

Through using the third filter, the process of finding fundamentals could be a lot easier, but the `Harmonic Processor` still needs to generate the harmonic tables so it can flag the harmonics for display on the screen.

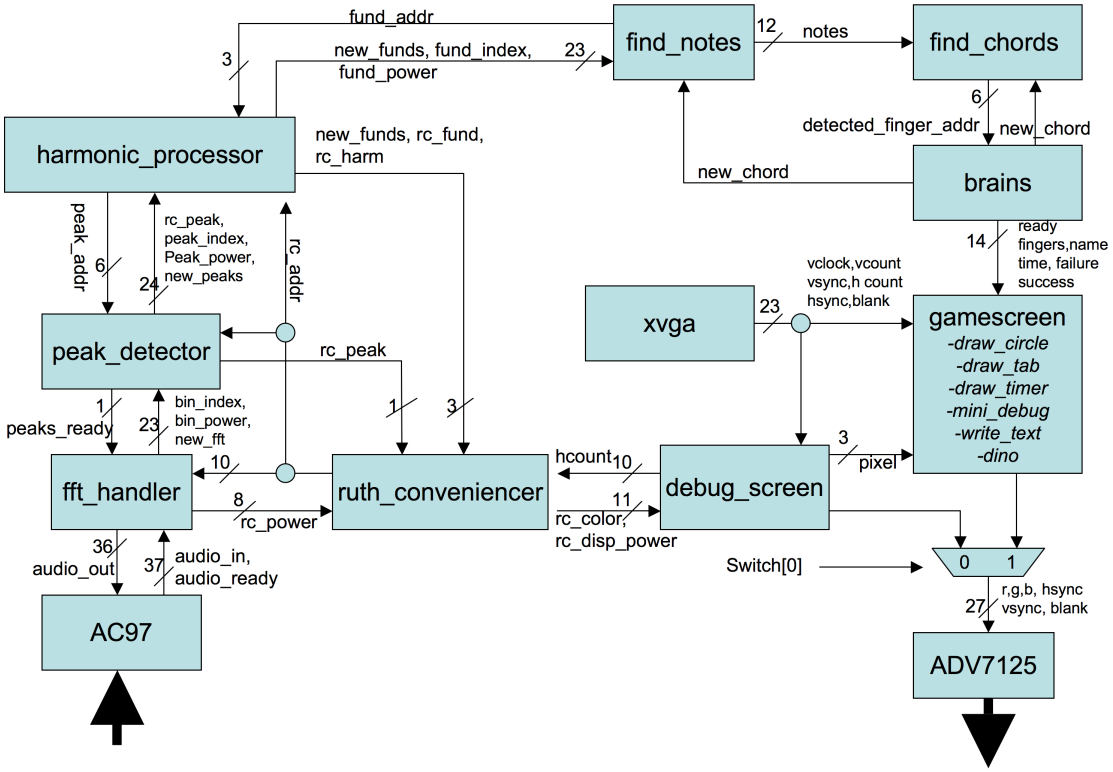

Figure 2: Full block diagram

2.4 Ruth Conveniencer (Rhys)

The `Ruth Conveniencer` is a buffer for displaying an annotated FFT on the screen. It gets the raw FFT from the `FFT Handler`, the `Peak Detector` marks the peaks, and the `Harmonic Processor` marks the fundamentals and their harmonics for display.

The `Ruth Conveniencer` collects data from the three frequency-space processing modules running at 27 MHz when the `Harmonic Processor` finishes and has it ready to display at 65 MHz whenever needed.

2.5 Debug Screen (Rhys + Ruth)

The `debug_screen` module provides a live display of the FFT output, with different colors highlighting the locations of peaks, fundamentals, and harmonics. This display was designed to improve the debugging of the frequency-domain signal processing. It also provides eye candy for the game screen.

One FFT bin, approximately 1.46 Hz wide, corresponds to one of the 1024 columns of pixels on the screen. Because the natural range of a guitar is 82Hz-1175Hz, this scaling for the debug screen allows all possible fundamental frequencies to be displayed. The boundaries in between musical notes are indicated by a scale along the bottom 10 pixels of the screen. The correct open-string frequencies for a guitar are also marked on the screen, which allows the user to tune the guitar by watching the location of the detected peaks.

`Debug_screen` uses *hcount* as an address to retrieve FFT power and peak tagging information from the `ruthconveniencer` BRAM. By comparing *vcount* to the power level for a given FFT bin, `debug_screen` generates *debugpixel[2:0]* for the fullscreen debug display and a 4x vertically scaled *minipixel[2:0]* that fits

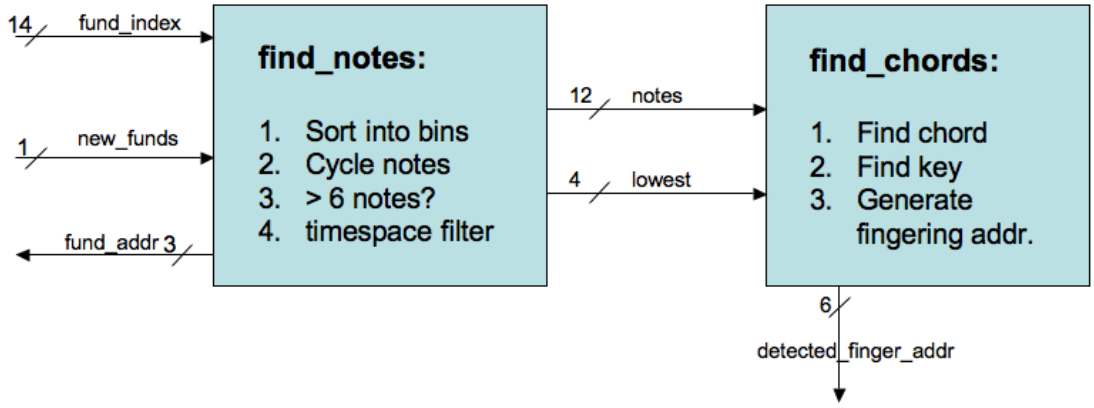

Figure 3: The output of `find_harmonics` is fed to `find_notes`, which sorts peaks into bins and filters out notes of short duration. The remaining notes are sent along to `find_chords` as a 12-bit vector with one entry for each note in a musical octave. Information about the chord is passed to the digital brain, which then tells the player whether they've hit the right chord.

into the bottom of the game screen.

2.6 Time-Domain Filter and Note I.D. (Ruth)

The purpose of the `find_notes` module is to discover what notes have been played on a guitar based on what fundamental frequencies have been detected. Early matlab simulations indicated that the vast majority of noise that survived the peak detection process was transient in nature. Therefore, the find_notes module includes a time-domain filtering step before passing results to the chord processor.

`Find_notes` was implemented as a finite state machine, following the transition diagram shown in figure 4. `Find_notes` becomes active on the rising edge of the *new_funds* signal that is generated every time a set of fundamentals has finished processing. The `SHIFTING`, `BINNING`, and `TRACKING` stages are repeated for each of the eight addresses where detected fundamentals are stored in the `harmonics processor`. Essentially, each peak's frequency is shifted down into the lowest octave of the guitar, compared against a lookup table, and sorted into the *current_notes* vector if it matches the frequency of a musical note.

After all eight peaks have been processed, the 12-bit *current_notes* vector should have a value of 1 at each bit that corresponds to a note that was detected (see figure **??**. The regfile *notes_tally* keeps a running count of how many times each note has been detected out of the past 8 samples. Any note that persists for more than 5 of the last 8 samples is deemed genuine. The `find_notes` module send 12-bit vector of these surviving notes, called *notes[11:0]*, to the `chord_processor` module to see if it matches any known chords.

2.7 Chord Processor (Ruth)

The chord processing subsystem has two steps. First, the `chord_processor` module identifies the key and chord i.d. of the current chord. These two three-bit numbers are concatenated to form a chord address that can be used to look up the chord's fingering in the `fingerrom` memory and the ASCII characters of the chord's name in the `chord_names` module.

2.7.1 Detecting chords

Seven keys (A,B,C,D,E,F,G) and seven chord types (major, major 7, 7, 7sus, minor, minor 7, and 9) have been programmed into the `chord_processor` module. First, the *notes[11:0]* vector is cycled until the lowest

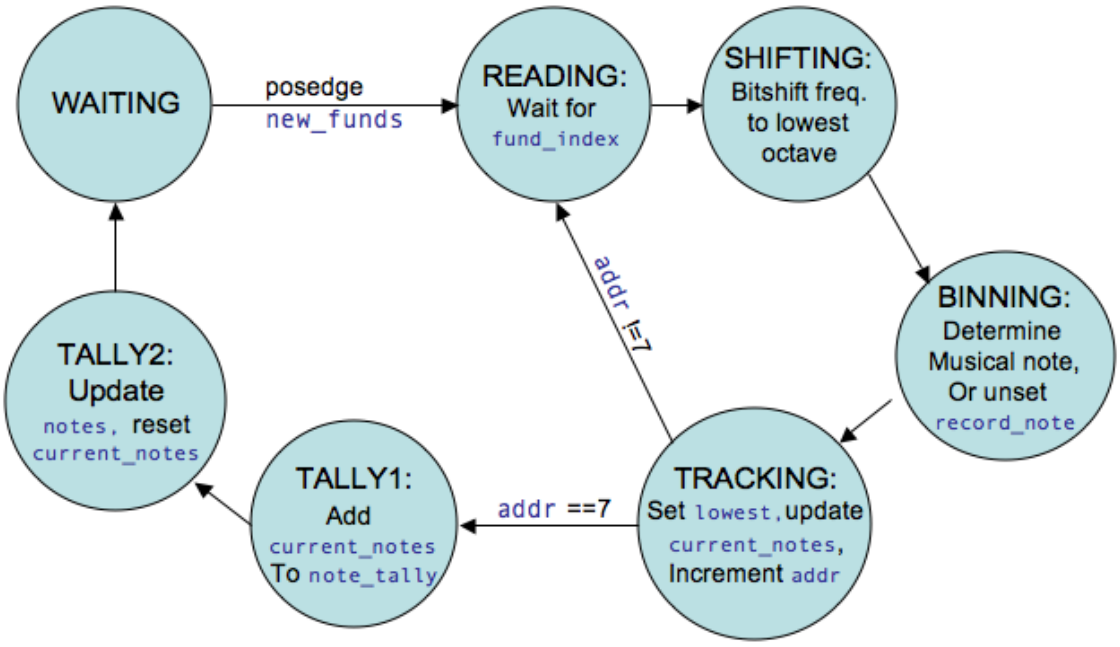

Figure 4: The `find_notes` FSM can be broken down into two big stages: sorting peaks into notes (READING,SHIFTING,BINNING) and filtering out transient notes (TRACKING, TALLY1, TALLY2).

note is in the "C" position. The relative distances between notes is all that matters in determining the chord type, and this first cycling step dramatically reduces the size of the chord lookup table. Once the chord type is determined from the lookup table, the bin of the lowest note can be combined with knowledge of how the chord was transposed to determine what key the chord was in.

2.7.2 Chord Names and Fingerings

Fingerings for all 49 available chords are stored in a ROM called `fingerrom`. The address of a chord's fingering is a six-bit number, {`chordtype,key`}. Each fingering is stored as an 18-bit number with three bits dedicated to the finger's position on each of the six strings of a guitar. Position `0` means no finger, `1` indicates a finger behind the first fret, *2* indicates the second fret, etc. A value of `7` indicates that the string should not be plucked.

The same address that marks a chord's fingering is also used by the `chord_name` module to generate the ASCII codes for the chord's name. The game screen calls `char_string_display` to print the text on the screen.

2.8 Digital Brain (Ruth)

The digital brain subsystem controls gameplay. A main `brains` module calls on `clkdivide2` to generate the enable signals that drive a timer. The game works as follows: a chord's name and fingering are drawn on the screen, and the user has a limited amount of time to play the correct chord in response. If the timer runs out or they play something wrong, then a red square is shown on the screen. When the user plays the correct chord on time, a green square and a bitmap of a dancing dinosaur are displayed on the screen. After a brief pause, the timer is reset and the user is asked for a new chord.

Chord sequences are written into the `songs` module, which is essentially a large case statement with the chords listed in order. Rhys programmed in the codes for "A Hard Day's Night" by the Beatles for our demonstration, but this module is ready to be extended to larger numbers of songs.

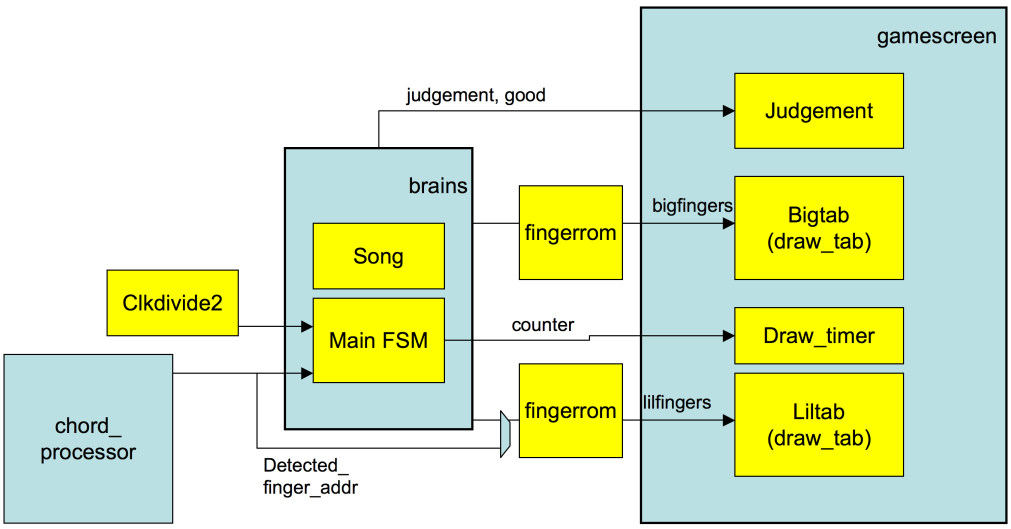

Figure 5: The interface between the digital brain and gamescreen.

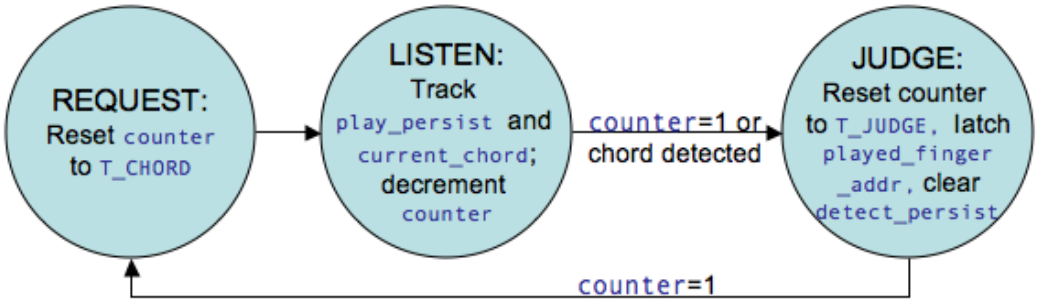

Figure 6: The digital brain, which controls gameplay, has two main states: `JUDGE` and `LISTEN`.

A state transition diagram is shown in figure 6, and a block diagram of the communication between the gamescreen and digital brain, is shown in figure 5.

2.9 Gamescreen (Ruth)

The `gamescreen` module draws the screen by coordinating several smaller modules that have each been assigned their own drawing tasks.

2.9.1 Drawing the Tabs

A tab is a picture of the neck of a guitar, with six vertical lines to represent the strings and a few horizontal lines to show the locations of frets. The `Draw Tab` module called the `Draw Circle 2` module each time a finger was to be indicated on the tab. Each chord fingering contains 3 bits of information per string that encode the vertical position of the finger on the string. A value of zero indicates that a finger should not be drawn.

The `Draw Tab` module was also useful for debugging when its fingering was set to display the raw output of the `Chord Processor`.

2.9.2 Displaying Text (From Fall 2005 website)

A module called `Char String Display` and an ascii table coe file were taken from the Fall 2005 6.111 website. The code was modified so that the fontrom would exist in the game module, allowing multiple instances of `Char String Display` to share the same ROM. The current, played, and next chords' names were displayed by this module.

2.9.3 Mini-Debug Screen

The `Mini Debug` Screen is a scaled version of the FFT debug screen that is generated at the same time as the full debug screen. It adds a very dynamic feel to the user interface, and is useful for tuning the guitar.

2.9.4 Judgement (Ruth + Rhys)

The `Judgement` module takes two inputs, *judgement* and *good*, from the `Digital Brain`. If *judgement* is 1 and *good* is zero, then the user has played the wrong chord or run out of time an a red rectangle is displayed. If the judgement is favorable, then the user sees a green rectangle as well as a bitmap of a dinosaur with an electric guitar that was programmed by Rhys.

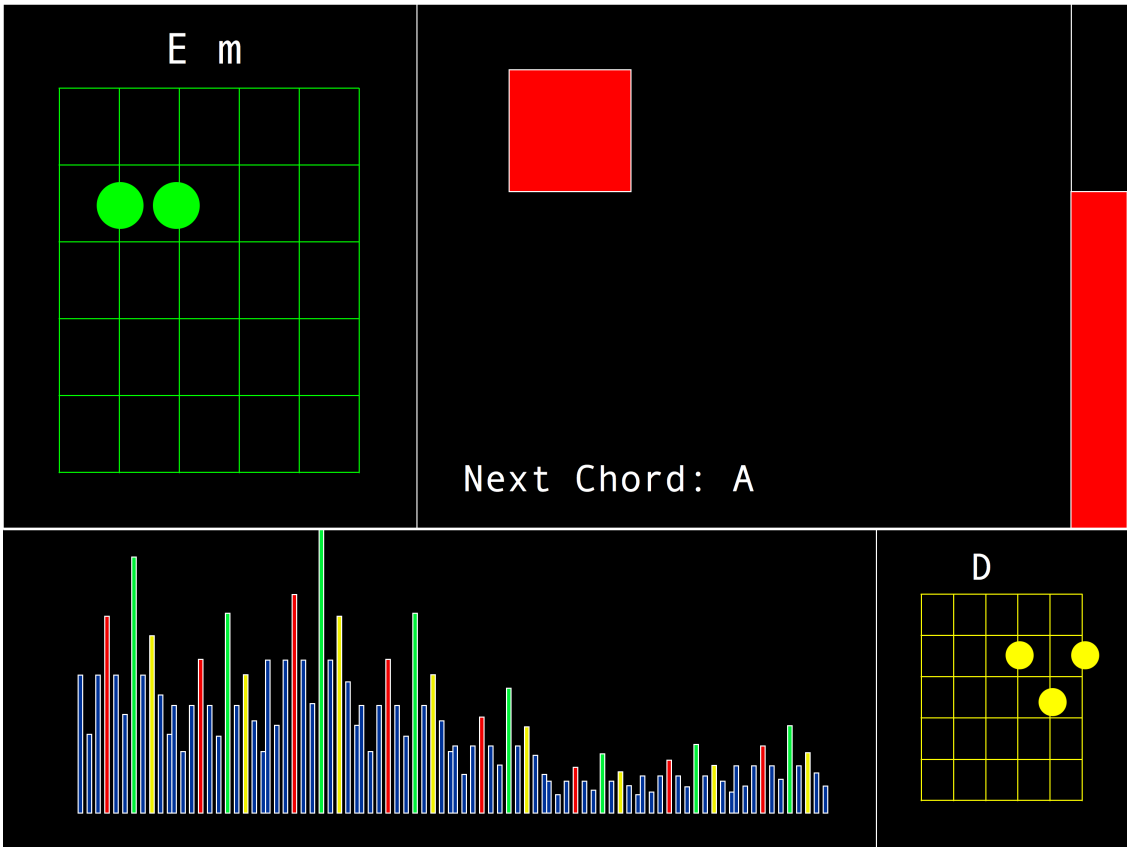

Figure 7: An artist's rendition of the gamescreen. See the demo video for a live picture.

2.9.5 Timer Bar

3 Testing and Debugging

The logic analyzer was invaluable for debugging the first half of the audio processing chain. Gim provided extra dongles for connecting all 64 data lines to the logic analyzer, and we could use switches to change the set of signals that got sent to the analyzer. This was very useful for examining the control signals for the FFT as well as the data flowing into and out of it, watching the various conditions for peaks getting detected change as the FFT marched past, looking at the tables of harmonics get built, and seeing the modules flag peaks, harmonics, and fundamentals for the `Ruth Conveniencer`.

The most next most useful tools for debugging the first half of the audio processing chain were the `Fake FFT Handler`, which was a direct drop-in replacement for the `FFT Handler`, but output fake data from a lookup table, and the `Fake Harmonic Processor`, which was a direct drop-in replacement for the `Harmonic Processor`, but did absolutely no processing of harmonics. These enabled extremely quick compiles and predictable data for testing the `Peak Detector`, `Ruth Conveniencer`, and other modules.

For Ruth's time-domain signal processing, outputs of the note tallies of the `find_notes` module were quite useful. The logic analyzer was also extensively used for debugging purposes. The video gamescreen was most easily tested and debugged by observing what was drawn on the screen. Most of the switches were in constant use for turning filters on and off, moving the system between different states, or switching the display mode.

Dummy modules were also extremely useful for debugging the time-domain signal processing chain. The `test_cases` module accepted addresses 0-7 and returned the frequencies of a guitar playing a g7 or a c chord, mimicking the behavior of the `Harmonic Processor`. An older version of the `test_cases` module would also mimic the `Find Notes` module so that the chord By the time both developers had finished early testing versions of their code, the integration only took a few hours because all modules had been tested with well-specified dummy modules.

4 Conclusion

SNGH is special because it can resolve individual notes over the full frequency range of the guitar. This is made possible by the excellent resolution of the FFT of 1.47 Hz. Instead of filling the FFT with $2/3^{rds}$ of a second of audio data, which would result in unacceptable delays in chord detection, we used a Hann window and zero padding to put $1/12^{th}$ of a second of audio into the FFT at a time. This gave us good resolution in frequency and time.

SNGH is also set apart by its ability to detect chords. This functionality relies on the identification of multiple fundamental frequencies, requiring advanced signal processing. First, a list of peaks is generated from the FFT output. Those peaks are sorted through to find which ones' frequencies are integer multiple of others. If a peak is located at a multiple frequency of another peak, it is not a fundamental. The few peaks that aren't multiples of any other peaks represent the fundamental frequencies of the vibrating strings on the guitar. The list of vibrating strings is monitored for transient entries, which are discarded. When a stable list of stings has been found, SNGH looks for chords involving that set of frequencies. If it finds a match, the chord is displayed on the screen and compared to the requested chord.

One of our best design decisions was to make our interfaces as simple as possible. Our modules ran on two different clock domains (65 MHz for Ruth's modules, 27 MHz for Rhys's), but it was never a problem because our interfaces were effectively BRAMs. The interface to display the FFT on the screen was a 10x1024 BRAM, and the interface for passing the fundamentals was a 22x8 regfile.

Tests of SNGH's operation were mostly successful, but there were a few areas that could have been more polished and reliable in their operation. This project was fun to develop, and we plan to continue making improvements over IAP. This may include: displaying an x over the strings that aren't supposed to be strummed, an better programming interface for entering new songs, and optimization of the frequency-domain signal processing so that SNGH will fit onto a smaller FPGA that we can keep. Some work remains

to be done on the user interface: a score should be displayed on the screen, and the transitions between states of the digital brain weren't very intuitive to some of the users who tested the game.

Neither of us had played guitar before the project started, but after two days of playing the game in lab, Rhys knew enough chords do demonstrate "A Hard Day's Night," by the Beatles. We are curious to see how well this game can be used for learning the tabs to our favorite songs over IAP.