# Bass-Hero
# 6.111 Final Project Report

Humberto Evans Alex Guzman

December 13, 2006

**Abstract**

Our 6.111 project is an implementation of a game on the FPGA similar to Guitar Hero, a game developed by Harmonix. It interfaces a real guitar with the FPGA and the user plays by hitting the notes and chords displayed on the screen. The user gains points by hitting the proper notes, and loses points if he fails to play the correct note. It is essentially DDR with a guitar instead of your feet. The game is implemented with a set of modules that interact with the guitar and drive the VGA display that tells the user what to play. An FFT is used to get frequency data from the guitar and frequency data is mapped to fret numbers on the guitar. The VGA display is used to display scrolling representations of notes on the screen which the user is expected to play. The game logic module compares the note the user is supposed to play with the note the user is playing and awards the user the score which is also displayed on the screen. While all this is going on, the song is being played back from the flash memory through to the speakers.

# Contents

**1 Overview 1**

**2 Description 2**

2.1 FFT Block . . . . . . . . 2
2.2 Note Detector . . . . . . . . 2
2.3 Flash Play . . . . . . . . 4
2.4 Flash Write . . . . . . . . 4
2.5 Note Decoder . . . . . . . . 4
2.6 FSM . . . . . . . . 6
2.7 Game Logic . . . . . . . . 6
2.8 Video Module . . . . . . . . 7
2.9 Hex Convert . . . . . . . . 10

**3 Testing and Debugging 11**

**4 Conclusion 13**

# List of Figures

1 Model of Note Representation on Screen . . . . . . . . . . . . . . . . 1
2 Note Decoder Block Diagram . . . . . . . . . . . . . . . . . . . . . . 4
3 Templatefile note pitch encoding. Chart for the first 5 bits of the templatefile encoding. . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
4 Templatefile note duration encoding. Chart for last 3 bits in templatefile encoding. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
5 FSM State Transition Diagram. The number of each state corresponds to that states game_mode value. . . . . . . . . . . . . . . . . . . . . . 7
6 Game Logic Block Diagram. . . . . . . . . . . . . . . . . . . . . . . . 8
7 Video Module Block Diagram. . . . . . . . . . . . . . . . . . . . . . . 8
8 Sprite Color Code. The color of each sprite tells us which fret we should play on the string that the note sprite is on. . . . . . . . . . . 9
9 Play Mode screen. This is an example of what the Play Mode screen looks like. The Play Now marker is displayed with a yellow box. This screen is composed of pixels from the Note Logic module and the Fretboard module. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
10 Start Screen. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
11 Play Pause Screen. . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
12 Game Over Screen. . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

# List of Tables

1 Mapping of String and Fret Numbers . . . . . . . . . . . . . . . . . . . 3
2 Mapping of Fret and String to FFT bin index . . . . . . . . . . . . . . 3

## 1 Overview

Bass Hero is a game implemented on the 6.111 lab kit. To play Bass Hero the user holds a real electric guitar that is plugged into the lab kit. The idea of bass hero is for the user to play along with the bass line of a real song. While playing the game the user will hear the song and see a visual map of which notes he should play at which time on the VGA display. The notes on the display scroll to the left on a visual representation of the guitar. The color of the square will map to which fret the user should play (See Figure 1).

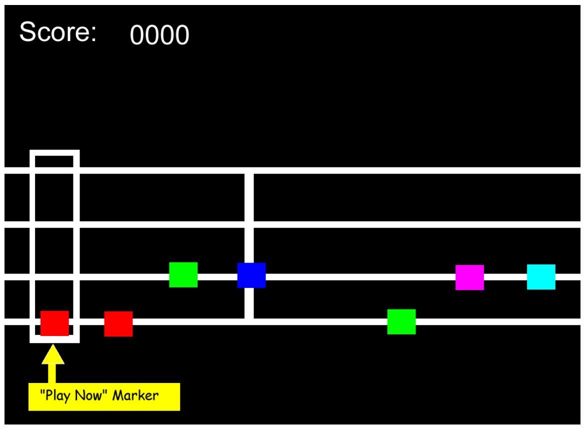

Figure 1: Model of Note Representation on Screen

When the note reaches a marker at the left of the screen, the user should play the note. The object of the game is to keep up with the song and hit as many notes as possible correctly. While playing the user gets continuous feedback about how he is doing with a score that is displayed on the screen. Scoring is based on how close the user was to playing the note note at the right time. The user receives no points for playing an incorrect note. There will be a short period between note transitions where no scoring is tabulated to allow the user enough time to switch finger positions. The guitar is interfaced with the FPGA via the AC97 microphone input integrated into the lab kit.

The full song is stored in the flash memory on the lab kit and is played back through the AC97 chip when the user is playing the game. A representation of which notes need to be played is kept on the BRAM memory. While the user is playing, a video module requests the next set of notes to be played. It displays that to the user represented as a set of colored blocks scrolling across the screen. Concurrently, a set of modules will use a Fourier Transform to extract frequency data from what the user is playing on the guitar and map the frequencies to guitar frets. The game logic will compare being played by the user at the time and assign a score based on how well they match. The score is then passed back to the video module and displayed as feed back to the user.

When the user starts the game he will see a start screen. Prompting the user to press the start button, in our case, button3 on the lab kit. (See Figure 2). As soon as the user presses the start button the song will begin playing and the notes will begin to scroll across the screen. The user can pause the game at any time by pressing the start button and resume play by pressing the start button again. Once the music starts grab the guitar and enjoy.

## 2 Description

Bass Hero was developed as a set of modules that interact to bring the user the Bass Hero Experience. The two main modules are the Note Detector and the Video Module. The Note Detector takes frequency data from the FFT block which is computing the Fast Fourier Transform of what the user is playing. It them maps that to which fret on which string the user is playing. The Video Module displays all the information to the user to on the screen, and is in charge of the timing of the game. At designated intervals it retrieves the data for the next note the user is supposed to play from the Note Decoder. It displays the note and scrolls it across the screen until it reaches the scoring area. While the note is in the scoring area, the video module sends to the scoring module which note the user should be playing and how much it would be worth if he played it at that time. In this way, the video module dictates which notes should be played when. The game logic simply takes the score for the note, the current note that is being played, and compares that to the note the user was supposed to play. If they are the same, it awards the points to the user. The Video Module also displays the score to the user.

### 2.1 FFT Block

The FFT block is responsible for extracting the frequency data from the waveform generated by the electric guitar plugged into the labkit. For this block I used a 4096 point FFT generated by the CoreGen Module of Xilinx tools. The module takes as an input the 8 bit data from sampling the microphone input (to which the guitar is plugged in) and performs a fast fourier transform on the data. It outputs one bin of the FFT per sample it is given. The output is three numbers, the index number of the bin, and the real and imaginary parts of the FFT at that index. This module outputs a scaled version of the real and imaginary parts that are scaled by 16. The FFT Block outputs all of the bins before starting again, therefore we get one full sweep of the FFT every 4096 samples we give the FFT.

### 2.2 Note Detector

The note detector is the main part of the back end portion of Bass Hero. The Note Detector receives the frequency information from the FFT Block. Every time it

Table 1: Mapping of String and Fret Numbers

| | Fret1 | Fret2 | Fret3 | Fret4 | Fret5 |
|---|---|---|---|---|---|
| String1 | E 329.63 | F 349.23 | F# 369.99 | G 392.00 | G# 415.30 |
| String2 | A 440.00 | A# 466.16 | B 493.88 | C 523.25 | C# 554.36 |
| String3 | D 587.32 | D# 622.26 | E 659.26 | F 698.46 | F# 739.99 |
| String4 | G 783.99 | G# 830.61 | A 880.00 | A# 932.33 | B 987.77 |

Table 2: Mapping of Fret and String to FFT bin index

| | Fret1 | Fret2 | Fret3 | Fret4 | Fret5 |
|---|---|---|---|---|---|
| String1 | 27 | 29 | 31 | 33 | 35 |
| String2 | 37 | 38 | 41 | 44 | 47 |
| String3 | 50 | 53 | 56 | 59 | 62 |
| String4 | 66 | 70 | 75 | 79 | 84 |

receives a new set of values it processes the the frequency data of the bin numbers that we care about. Since the user is playing on a guitar, and the game is limited to only the first four strings of the guitar the user can only play 20 distinct notes. Therefore, we are only looking for 20 particular notes in a limited frequency range. A Table of the frequencies we are looking at and which notes they are on the guitar can be found on Table 1. A table of which bin index each fret is mapped to can be found in Table 2.

Every time the FFT block processes a new sample it sends the bin index, and scaled versions of the real and imaginary parts for that bin. If the bin index falls between 24 and 87 (which includes all the bins we care about See Table 2) then the module begins processing that frequency bin. It begins by finding the magnitude of the FFT by squaring both the real and imaginary parts, multiplying them together and then taking the square root. It then compares the magnitude with the maximum magnitude it has detected thus far from the string on which the fret is located. If the magnitude is bigger it makes that the new maximum and stores which fret has the current running max on each string. In this way the module keeps a running max of which fret has the biggest frequency value associated with it on each string. Since the user can only play one note on each string, and we assume that the fret with the largest magnitude is the one the user is playing, every time the module does a complete sweep of the FFT the module will know which notes the user was playing during that sweep of the FFT. With the AC97 chip running at 48KHz and one full sweep of the FFT requiring 4096 samples a complete sweep is completed about once every 1/12 of a second. It is reasonable to assume that a song will not change notes faster than that.

Every time the FFT cycles through to the same bin index, which means it has completed a full sweep, the module saves the values for the highest bin on each string. It then looks up which fret number that bin belongs to, and presents the information at its output for the video module to read. The output is four three bit numbers each number representing a string, and the value of the number representing the fret number on that string.

## 2.3 Flash Play

The Flash Play module is responsible for playing back to the AC97 chip the PCM data of the song stored on the Flash Memory of the labkit. Every time the ac97 requests a new sample, the module retrieves a new sample from the flash memory. Once the sample is ready, it passes it to the AC97 chip.

## 2.4 Flash Write

The Flash Write module was the hardware used to get the song onto the labkit. The raw PCM data file was first converted into a .coe file using a Java program I wrote. The Flash Write module creates a large BRAM and initializes it to as much of the contents of the song as possible. It then steps through it and writes each location to the flash memory. I then repeated this process until the whole song was on the flash memory. This process was necessary because writing to the flash is a very slow process and recording from live audio was not an option.

## 2.5 Note Decoder

The Note Decoder module is responsible for retrieving data that encodes the string and fret that the user should play, and the interval between notes. The module is composed of two blocks: a BRAM that holds the song data, called the templatefile, and a logic module that decodes the templatefile data and outputs it in a form that can be used by other modules (see Figure 2).

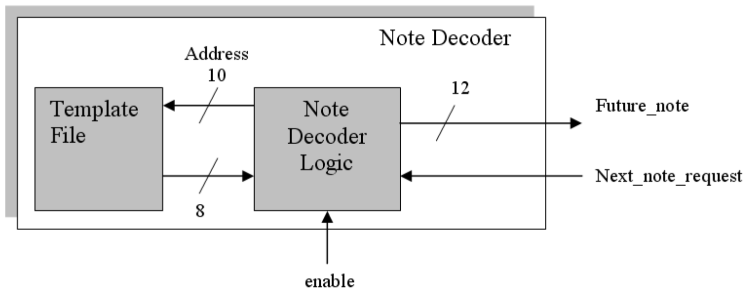

Figure 2: Note Decoder Block Diagram

The templatefile is a BRAM that holds the song note data. The data that it holds was generated by a java program we wrote called templatemaker (see appendix). The templatemaker takes an array of strings (written by the user), each string two words long. The first word tells the program which note the should come next in the song, for example, an open E (1st string open fret) would be written as E and the E an octave above the open E (2nd fret on the 3rd string) would be written as E2. The second word tells the program the duration of that note, for example, whole for a whole note or dhalf for a dotted half note. The templatemaker program can handle notes up to B2 and note durations between a whole and an eight note. The conversion from notes and duration to a binary number to be stored in the BRAM can be seen in figure 3 and figure 4. Each entry is 8 bits long. The first 5 bits represent the note that should be played. The next 3 bits represent the duration of that note. If the last three bits are all true, the file is over. The program will output the contents of the .coe file to the systems console. The data output can be copied into a notepad file and the file extension can be changed from a .txt to .coe. This .coe file was used to initialize the BRAM.

| | Rest | Open | Fret 1 | Fret 2 | Fret 3 | Fret 4 |
|---|---|---|---|---|---|---|
| E String | 11111 | 00000 | 00001 | 00010 | 00011 | 00100 |
| A String | 11111 | 00101 | 00110 | 00111 | 01000 | 01001 |
| D String | 11111 | 01010 | 01011 | 01100 | 01101 | 01110 |
| G String | 11111 | 01111 | 10000 | 10001 | 10010 | 10011 |

Figure 3: Templatefile note pitch encoding. Chart for the first 5 bits of the templatefile encoding.

| | |
|---|---|
| 𝅝 | 000 |
| 𝅗𝅥. | 001 |
| 𝅗𝅥 | 010 |
| ♩. | 011 |
| ♩ | 100 |
| ♪ | 101 |
| end | 111 |

Figure 4: Templatefile note duration encoding. Chart for last 3 bits in templatefile encoding.

The Note Decoder Logic module loads data from the templatefile BRAM and decodes the data into a form that can easily be used by the Video module and the Game Logic module. In particular, the logic module will take the encoding discussed in the templatefile module, and output the data as which fret and which string the user should play. This is encoded as a 12 bit number. Bits 12-10 encode the fret that should be played on the G string, bits 9-7 encode the fret on the D string, bits 6-4 on the A string, and so on. The Note Decoder Logic module receives a next_note signal from the video module (which will occur 4 times a second, one for every eighth note). The Note Decoder Logic module keeps track of the note duration for each note that is fetched from the templatefile using registers. For example, if a whole note is fetched from the templatefile, it will take 7 more next_note signals until the logic module retrieves the next note from the templatefile. The Logic unit takes the one clock cycle delay in the BRAM when trying to read a new output, the Logic module will ask the BRAM for data on one clock cycle, and retrieve/decode it on the next cycle.

## 2.6 FSM

The FSM module is responsible for generating the proper enable signals to other modules and for telling the Video Module which screen to display. There are four states in the FSM: Title Mode, Play Mode, Game Over Mode, and Pause Mode (see Figure 5). The FSM is controlled by Start, Reset and End Game signals. The Start and Reset signals come from the user (except for the startup reset), and the End Game signal comes from the Note Detector module, which is true when song file in the BRAM is finished being read. Enable will only be high when the game is in Play Mode, causing other game modules from running when theyre not supposed to.

## 2.7 Game Logic

The Game Logic module is responsible for comparing the note that the user is currently playing (from the Note Detector module) with the note that the user should be playing (from the Video Module) and increments the total point value if the user is correct (see Figure 6). Since the Note Detector module will also register values from other notes that the user might not be playing, but were excited by playing a certain note (the note an octave above what the user is playing for example), the Game Logic module only checks for the user playing the correct pitch on a certain string, and ignores the other strings that should have been muted. If what the user plays matches what the game expects the user to play (minus the muted strings just described), the value of note_worth, given by the video module, will be added to the total score. A register is used to keep track of whether a point has been awarded for the current note. Once a point value has been awarded for one note, no further points can be awarded for continuing to play the same note. The signal clear_old tells the game logic that a note at the input is new and points can be awarded once again. The total score is output by the Game Logic Module, and is used by the Video Module to display the current score.

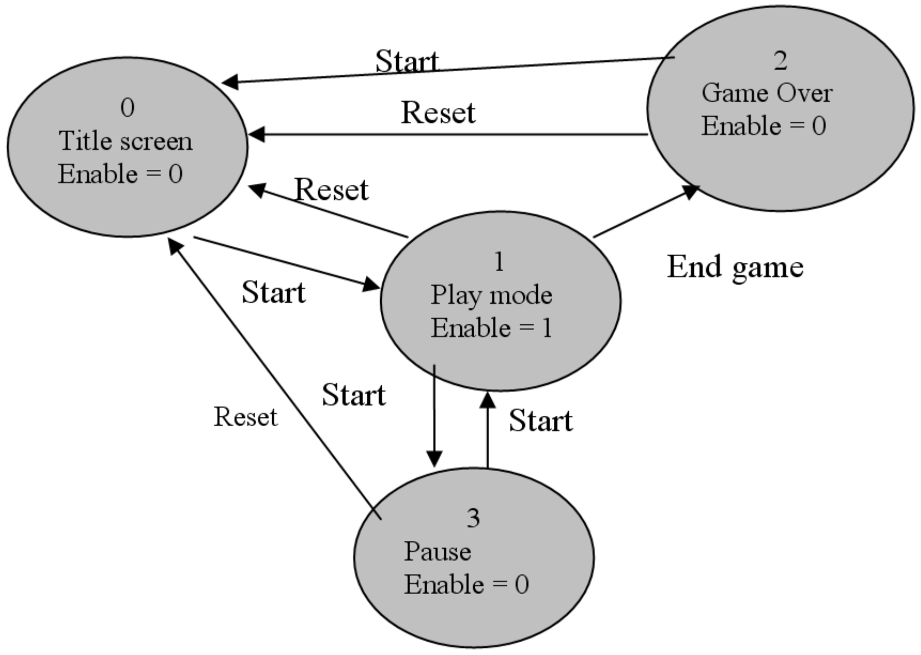

Figure 5: FSM State Transition Diagram. The number of each state corresponds to that states game_mode value.

## 2.8 Video Module

The Video Module is responsible for displaying what the user should be playing, displaying a score, displaying title screens, and generating the signals: next_note (15hz), current_note(what the user should be playing), note_worth, and clear_note. The Video module is composed of a Note Logic module (which is responsible for displaying note sprites), a Fretboard Module (responsible for displaying the fret board), a Pause Screen module (responsible for displaying the pause screen), a Game Over module (displays game over screen), a Start Screen module (displays start screen), a HexConvert module (converts a binary 11 bit number into a 32 bit, 4 character ASCII number), and a logic module that selects which pixel should be displayed (see Figure 7).

The Note Sprites module is a simple module that takes in a 10 bit number for the X position of the sprite, a 3 bit number for which string the sprite should be on, and a 3 bit number for the color. The module outputs a pixel value depending on whether hcount and vcount are over the note sprite area.

The Measure Bar module is simply a module that checks to see if the hcount and vcount are within the bounds of a narrow rectangle defined by the X input and internal parameters of the Measure Bar module. It will output a white pixel if hcount and vcount are in the rectangle.

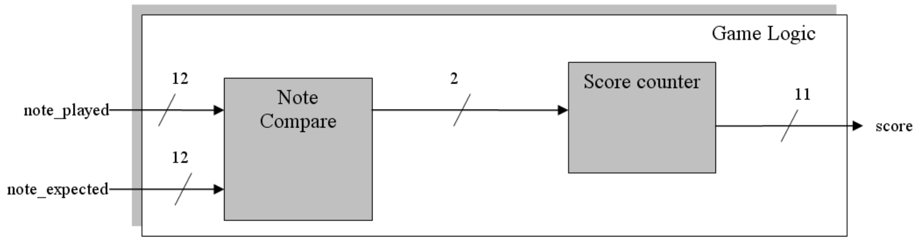

Figure 6: Game Logic Block Diagram.

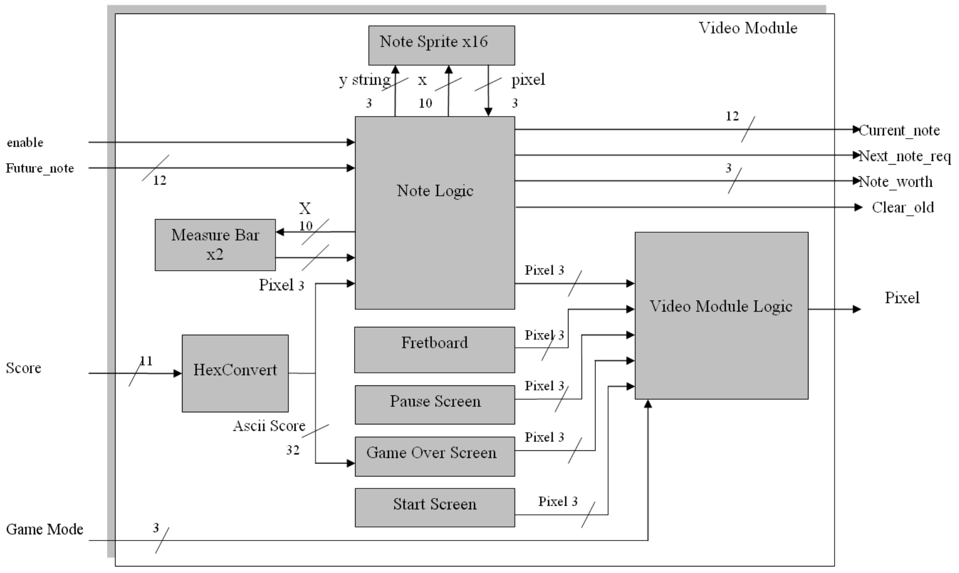

Figure 7: Video Module Block Diagram.

The Note Logic module is responsible for displaying the sprites that the user should play. The module does this by creating 16 note sprites; each sprite is 64 pixels away from the next sprite horizontally. There is a 10 bit register, referenceX, which keeps track of one X position. The value of referenceX is decreased every vsync, creating an X position that scrolls across the screen to the left. The X position of each sprite is referenceX plus an offset (a multiple of 64). This guarantees that each sprite will be evenly spaced horizontally. The color and y position of each sprite is determined by the input future_note from the Note Decoder module. The color of a sprite on a string tells us which fret we should play on the given string (see figure 8). Once a sprite has almost reached the left of the screen (5 vsync signals before), a call is made to the Note Decoder module for the next note (using next_note_req). As soon as a sprite reaches the left of the screen (x position is 0) the Video module will load the value given by future_note and update the sprites registers to save those values until they are no longer needed (when the sprites x position reaches 0 again).

Once a sprite almost reaches the Play now marker (at x position 96, see figure 9), the Video module outputs that sprites string and note value to the game logic module for scoring, along with a high clear_old to let the Game Logic module know that the input is now different from the previous note. In addition, the Video module outputs a note_worth value that represents the points that will be awarded if the player plays the correct note at that moment in time. The note_worth value is a 3 bit number, ranging from 1 to 7. A maximum of 7 points is awarded when the sprite is directly over the play now marker (x position 64), and decreases based on the distance of the sprite from the play now marker.

The module also displays things other than note sprites. The module takes a 32 bit ASCII number from the HexConverter module and uses it to display the current score on the upper left corner of the display screen, and generates vertical Measure Bars at every measure

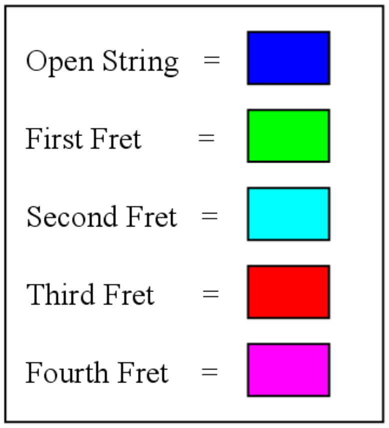

Figure 8: Sprite Color Code. The color of each sprite tells us which fret we should play on the string that the note sprite is on.

The Fretboard Module creates the image of the fret board in the background of the Play Mode and Pause mode Screen. It uses combinational logic to determine if the hcount and vcount are in the location where the fret board should be and assigns white to the pixel at that location (see fig 11).

The Start Screen Module generates the text seen at the start screen (see fig 10).

The Pause Screen Module displays a blue square with the words Paused in the center. The blue square is created by checking the hcount and vcount and checking if they are above and below certain values (see fig 11).

The Game Over Screen Module Displays the Game Over text as well as the final score (see fig 12). The string for the score comes from the HexConverter module.

The pixels from the Note Logic, Start Screen, Pause Screen, Game Over Screen, and Fretboard modules are put together in the Video logic module. This module takes the game_mode from the FSM and decides which screen should be displayed. It also puts the Fretboard pixels behind the Note Logic pixels in Play mode, and

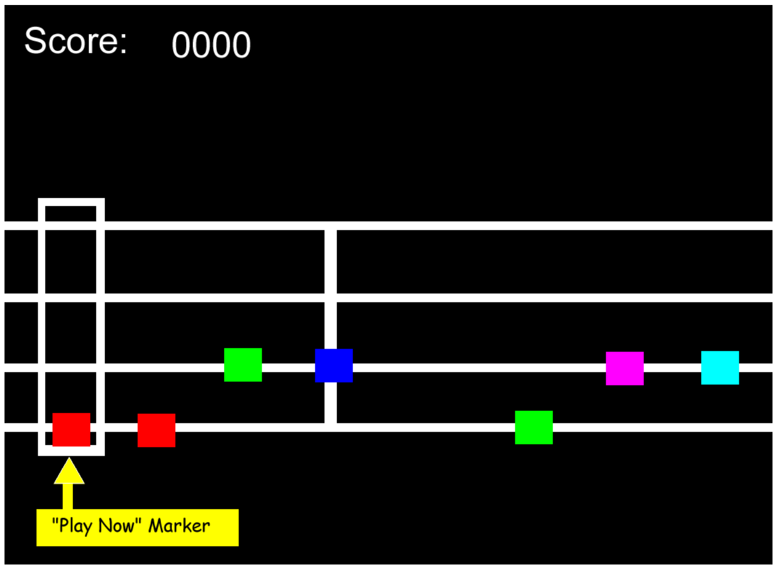

Figure 9: Play Mode screen. This is an example of what the Play Mode screen looks like. The Play Now marker is displayed with a yellow box. This screen is composed of pixels from the Note Logic module and the Fretboard module.

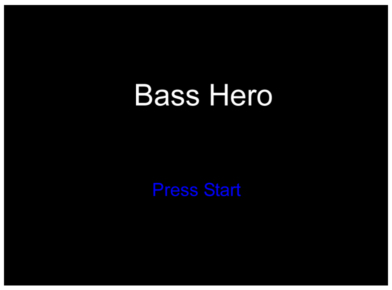

Figure 10: Start Screen.

puts the Pause Screen pixels on top of the Play Mode pixels when the game is in Pause mode. The module is also in charge of starting the HexConverter module at the beginning of every vsync, and saving the old output values of HexConverter in registers for use in the Note Logic and Game Over Screen modules.

## 2.9 Hex Convert

Hex Convert was a helper module developed to covert a hex number into an ASCII string that the video module needs to display the score. The Hex Convert received the score, and 11 bit number, from the scoring module. It then entered a loop where the score was divided by 10, the remainder became a digit of the number and the quotient became the new score. When it looped through again the next digit was filled in and the loop would continue until the quotient was zero. This way the hex

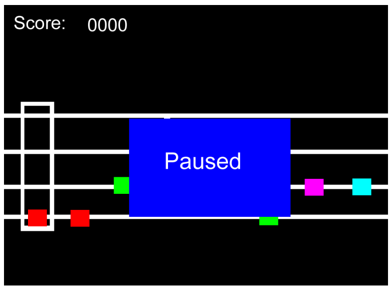

Figure 11: Play Pause Screen.

Figure 12: Game Over Screen.

number was converted to separate decimal digits. After that the module would add 0x30 to convert the digit to ASCII string, present the values of the four digits at the output

## 3 Testing and Debugging

The debugging process for parts of this project were rather tedious while other parts went smoothly. Most of the testing and debugging for the backend of the system was done on the logic analyze, and integration testing was done by play testing the game.

For the backend of the system, most of the problems encountered were not digital design problems. The biggest difficulty was trying to get the analog world to cooperate and interface nicely with the system. The first problem was encountered when the frequency range of a bass guitar was really low, and and the notes themselves were really close together in frequency. This did not allow the system to accurately detect the note that was being played. Also, because the notes were so close together in frequency, the FFT had to be a very large FFT to have enough resolution to tell the difference between two successive notes. Because of the length of the FFT it would take very long to complete the required full sweep and new data about what the user is playing would not be available fast enough. To fix this problem we used a guitar instead of a bass guitar. Since the first four strings of a guitar are the same as those of a bass guitar this did not change the game in any way, but it did make the frequencies fast high enough to work with. To increase the reliability of the game. The user plays starting on the 12 fret of the guitar, again making the frequencies higher and giving the system the ability to run a smaller FFT without losing accuracy. While using the logic analyzer to observe the magnitudes of bins I realized that the first harmonic was usually much cleaner and reliable. The system was switched to listen to the next harmonic up from the 12 fret of the guitar, and the reliability and speed of the system increased dramatically.

Writing to the flash was a relatively straightforward process once I understood how to write to the flash. To figure this out I looked at the flash test module provided by the 6.111 staff. I took the test sequence and modified it to write the data that I wanted to the flash.

Once the note detection system looked clean on the logic analyzer I hooked the output to the hex display to see what the system thought the user was playing. The last phase of testing involved playing on the guitar and observing the correct number appear on the hex display.

The playback from the flash was easy until it came time to synchronize it with the game. The first time the song was played along with the game the song over the headphones was playing much faster than it should have. Eventually I just realized that standard MP3 encoding is done using a 44.1KHz sample rate, and the AC97 chip was playing samples back at 48KHz. Upon realizing this a 44.1KHz enable was generated to drive the samples on the AC97. After that the song lined up perfectly with the game.

The FSM module was one of the easiest modules to write and test. A module called FSMtestSignals was created to generate start, reset, and endgame signals to test the behavior of each state in the FSM. The state of the FSM was output to the logic analyzer ports. The state of the FSM was compared to the input signals to test the module. Debugging was not necessary for this module.

The game logic module compares two signals and adds a value to a running score total. To test this module, we connected the module to a working Video and Note Detector module, and displayed the score on the led lights. We then checked to see if the score incremented by the proper amounts. A bug was found quickly, which was caused by clocking the module on the wrong clock. Originally the module was clocked at 65 MHz, but this clock frequency was too fast and performed operations on data multiple times before that data changed. Logic was added so that the module would still be clocked at 65 MHz, but only performs its scoring operations when vsync changes from 0 to 1. This eliminated unnecessary work, since the input scoring values will only change on vsync.

A module was created, called NDtestSignals, to generate enable, reset, and next_note signals to test the Note Decoder. The Note Decoder modules output was displayed on the logic analyzer along with the input signals. The output of the module was checked for the proper output. Extra attention was paid to make sure that the module output the right note at the right time (for example, making sure that the note is held for the proper number of next_note requests). No debugging was needed for this module.

Most of the testing for the video module was done on the VGA display. After code for the module was written, the project was compiled and the video display was seen. The signals for current_note, note_worth and clear_old were observed using the logic analyzer. Many bugs were observed in the Video module. First, the video module was not displaying anything. The problem was traced back to a bug in the way that pixels were put together, in particular, using AND gates instead of OR gates. The next bug occurred assigning the proper values to the note sprites. Some note sprites were not being assigned properly. This problem was traced back to a bug in how fast the value in the registers holding the X reference position decreased. The X value decreased too fast, causing the sprites to miss the queue for receiving a new value assignment. After some math, it was concluded that each sprite should move 64 pixels in 15 vsync signals. This meant that we needed to decrease X by 4 for 11 vsync signals and by 5 on the other 4 vsync signals. Some problems arose in displaying the score on the Play mode screen and in the Game Over screen. The score was not being displayed correctly. The issue was caused by the ASCII signal encoding the score not being ready in time for the Video modules to use. This was fixed by using registers to hold the previous value of the ASCII string stable for the video module to use while the HexConverter module computed the next ASCII value.

## 4 Conclusion

This entire project was a very enjoyable experience. Working long hours towards a goal that you are really excited about is more than worthwhile. This particular system is unique because the idea of a game that interfaces with a real guitar gives the user the ultimate feeling of reality. It is more than a simulation, you are actually playing along with on a bass with the song. Its even better than virtual reality, its actual reality. The process of building, and playing the game was really fun.

The design of the system required it to extract frequency information from the guitar. An interesting thing that we learned during the project is that the harmonics on a guitar are usually much cleaner and free of noise. Upon coming to this realization I was able to listen for the harmonics of the note being played, and the reliability of the system increased dramatically. This does however present a problem in actual note detection. Since we are listening to harmonics, we lose the knowledge of which actual note it is the user is playing; all we know is the letter of the note. It basically becomes impossible to tell the difference between the same note on different octaves that the user is playing. Luckily for our game this was not an issue since we only had to make sure the user was playing one note at a time we deemed it was ok if the user played the right note on the wrong octave.

The note detection system was overall very robust and quite reliable. It virtually always detected what note the user was playing, however it did often report notes that were not being played on other strings. Again, for the purpose of our game this was not a problem. Since we were only rewarding the user for playing the correct note, and not penalizing for the wrong notes the system merely ignored all the superfluous notes the note detector was reporting, and only made sure that the note that was supposed to be played was in fact played at the right time.

In the end the note detection system worked quite well and I was very pleased with its reliability.

More in depth research about the acoustics of guitars would have likely been very helpful and made the project go faster, since alot of my time was used up tweaking frequency bins and remapping when I realized a different scheme would work better.

The note sprites are constantly moving across the screen at the correct tempo. Once a sprite reaches the end of the screen, it will return the beginning of the screen and have new values assigned to its registers according to the input from the Note Decoder module. To solve the problem of having rests, where no note should be played, rested notes are displayed off the fret board as a black sprite, causing it indistinguishable from the black background.

Due to the way that the Video module was implemented, the song is only able to be played at 120 bpm. A good addition to the project would be to have adjustable song tempos. This would involve major changes to the Video module to ensure that the sprites move at the correct tempo.

The game currently does not support holding notes, it just checks in the duration of an eighth note so see if the user played the correct note. Adding this feature would improve the game. To do this, many major changes would need to take place. The Note Decoder module would need to be able to differentiate between a held note and many short notes played in sequence (for example, an E for a whole note, with four quarter note Es). The video module would need to be able to display the longer notes and output new note_worth values that takes the longer notes into account.

Glitching in the note sprites occurred once all of the video module pieces were put together. The problem occurred because the time it took for the proper pixel value for each note sprite to settle was too large. Originally each note sprite was a circular colored object. This involved taking the squares of differences in the sprites X and Y value with hcount and vcount values, can comparing it to an internal variable. This computation took too long. The problem was solved by using square sprites instead of circular ones, since square sprites do not need to compute any multiplication. The square sprites might not be as aesthetically pleasing as the circular ones, but solving the glitching problem was more important.

The question on how to move the note sprites was a tricky one to solve. Each note X position needed to decrement by a certain number in order to keep the correct tempo for the song. The vsync frequency of 60 Hz turned out to be perfect for out song which was played at 120 beats per minute. This allowed us to set one eighth note every quarter of a second, or 15 vsync cycles. Dividing the video monitor into

16 equal pieces, we set each sprite 64 pixels apart, leaving each note sprite to move 64 pixels in 15 vsync cycles in order to keep the correct tempo. This was achieved by decreasing the X position by 4 during 11 or the 15 vsync cycles, and by 5 for the other 4 cycles.

Bass Hero turned out to be a successful project. The game was playable and even enjoyable. It was almost as fun to build it as it was to play.