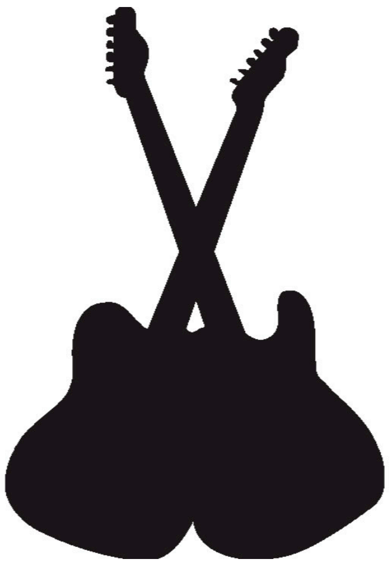


Guitar Commanders:
Final Report

by
Langtian Lang
Shaheen Ranjbaran
Chengjou Liao

CMU ECE 18-545
Professor Ken Mai
December 12, 2007

Background:

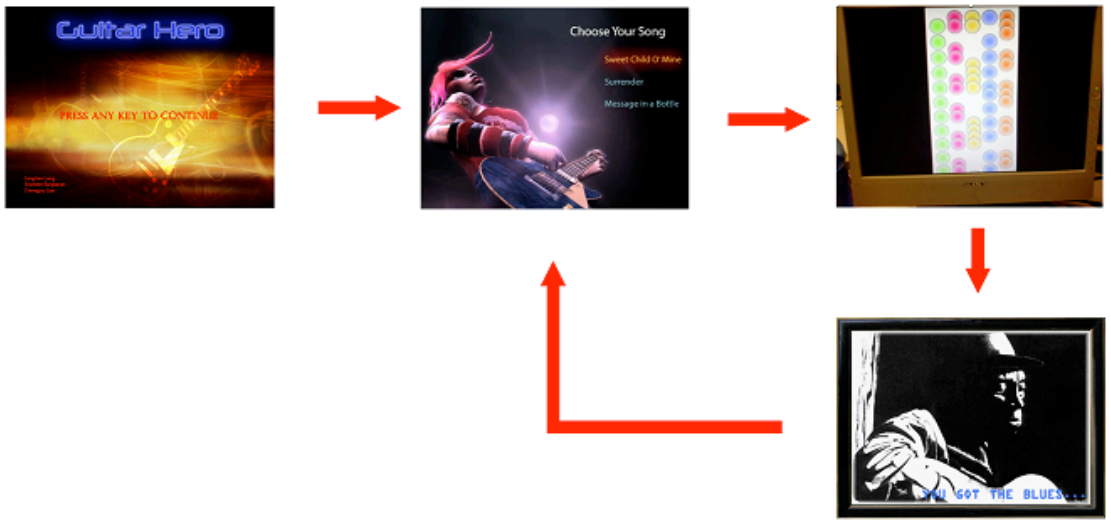
Guitar Hero is a fairly recent music rhythm-based game created by Harmonix Music Systems that has risen to popularity due to its easy-accessibility and innovative control scheme. Using a guitar-shaped controller, players essentially assume the role of the guitarist in a rock band and are given five-note patterns via a sort of piano scroll to play. Of course, even though the initial learning curve of the game is not terribly high, mastering the game and the complex note patterns that appear in many of the songs requires skill and practice.

Because Guitar Hero was released with only a limited number of songs, it wasn't long before an open source version of the game was produced for the PC. Called "Frets on Fire," this open source version was written in Python by Finnish developer Sami Kyöstilä and it allowed the online community to create and/or import their own songs into a Guitar Hero clone and even develop modifications for the software.

Game Description:

Inspired by the open source clone of Guitar Hero, Frets on Fire, we decided to create a Guitar Hero clone of our own implemented on the Xilinx Virtex-II board. Although much software coding has been written for Frets on Fire, the large majority of it has been left undocumented and is specific for a PC application. Therefore, we are largely developing our Guitar Hero clone from scratch, and along the way fine-tuning the balance between the hardware and software partitions in order to optimize our game.

Game State FSM:



The opening screen simply consists of the familiar title of our game, fancy guitar graphics we found online that was edited a bit with Photoshop, and our names. While on this screen the user is told to “Press any Key to Continue” which takes the user to the next screen. The purpose of the second screen is to allow the user to choose one of three songs. Based on our decoding of our notes, we were able to get note streams of varying difficulties of each song. The songs we chose are all used in the real version of Guitar Hero. “Message in a Bottle” was our most difficult song, “Sweet Child O’ Mine” was our medium difficulty song, and “Surrender” was our beginner or easy song. The player can use the guitar strum to vertically choose the song to highlight and then can press a fret button in order to pick the song and move to the next screen. Now the user actually begins the game play. The music begins and notes start to scroll down the screen. Their score is simultaneously updated in the top right hand corner of the screen in a scoreboard every time a correct note is hit. After the song is completed the user is taken to one of two post game screens. If the player performed well they are taken to a screen featuring Gene Simmons from Kiss indicating that they “Rocked Out,” otherwise you are taken to a somber “You Got the Blues” page. Both screens show the percentage of notes correctly hit rather than the actual score. Any key pressed at this point takes the player back to the “Choose Song” screen so that the user can choose another song and play again.

<u>Hardware Description:</u>

We implemented 2 main hardware modules: Psx_module and Memclr_module.

Psx_module:

This module implements a FSM to talk to the controller. The ports are connected to a low speed parallel expansion board which connects with the controller. Initially we chose to use a high speed board but it turned out not to work as well, still we used it to power the board. The purpose of this module is to output the controller input to the OPB bus which can be read by the PPC core.

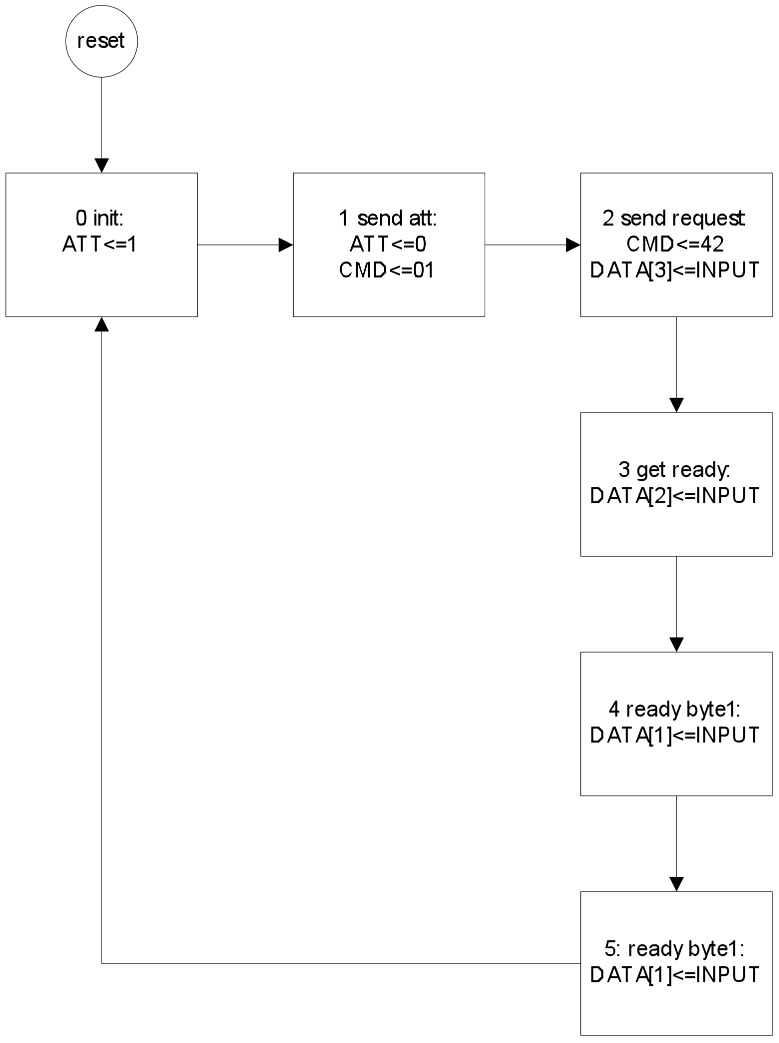
Memclr_module:

The Memclr_module is essentially a hardware buffer clearing module that is a PLB Master which can write directly to the memory allocated for the frame buffers. The PPC can control this module through couple registers and signals start, select (which tells the frame buffer to clear), and background color.

Controller:

The controller is connected to a ps2 adapter which is then soldered onto the parallel connector. This parallel connector is plugged directly into the low speed parallel port. The pin out connection from the guitar connected is ACK, CLK, DATA, CMD, ATT.

Hardware FSM:



1. As one can see from the diagram above we begin with ATT<=1 to wake up the controller.
2. Then we send CMD<=1 in order to poll the controller ID.

3. At this point the controller should send back its ID, and at the same time send CMD<=0x42 to ask for the current state of buttons.
4. The controller will then send back 0x5A, which signals that data ready to send.
5. The first byte is then sent. These bits determine the current state of the buttons.
6. Then the second byte is sent.

<u>**Software Description:**</u>

Controller:

This portion of the software talks directly with the hardware controller FSM and polls for the controller every frame. It compares the current input with the current note being drawn on the screen near the bottom in order to determine whether the note is correct or not. The score is quickly updated accordingly.

Audio:

The audio codec FIFO is filled with the samples read from the .wav files are added depending on whether the input is correct. They are simply added and divided by two and then copied across the left and right channels. Our audio was not updating enough when we were using our software frame buffer clearing method so we thought one solution would be to enlarge the audio FIFO so that more samples can be included every frame. However, this turned out to be unnecessary since our hardware Memclr_module can clear without delay.

Initially, our wav files were encoded in PCM at 16bit stereo at 44100 Hz. Because of this, every song would total up to be about 60MB worth of data to load. We realized this would take a long time to load, and thus tried to implement some sort of compressed algorithm decoding. However, we instead just down sampled the frequency of the samples until we can hear a noticeable different on the computer speakers. All the files ended up being encoded in PCM at 11025Hz of 16bit samples. The audio files are read in a mono to save even more space and decrease loading time.

Note File:

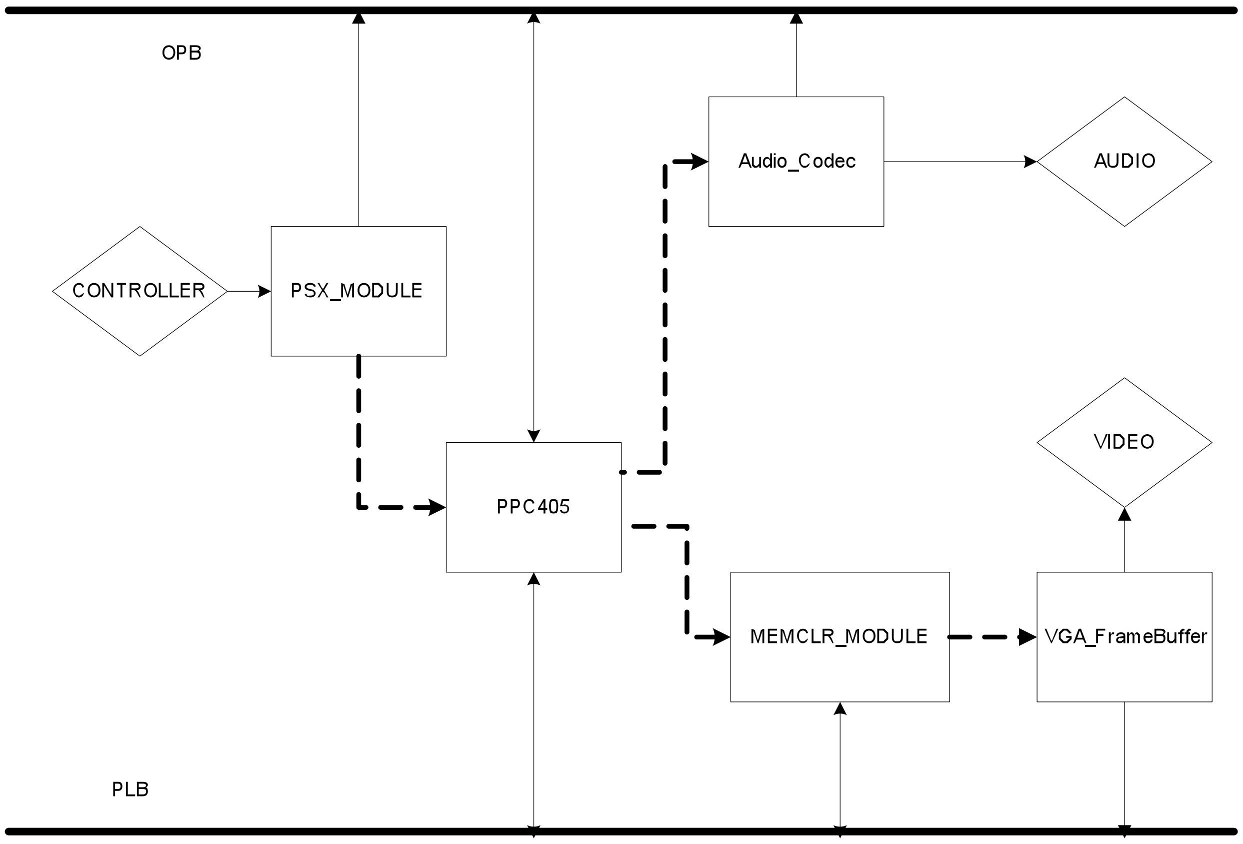
This is a file which is initially for the song and determines which notes appear and for what length. The structure of this file is very simple. Pairs of <8 bit note> <8bit length> are read in and the notes are displayed on the screen. The beginning of the note file lists the length of the entire song which is read by the software. A table is also used to keep track of the current notes being drawn into the frame buffer. Thus for every frame the table is shifted to represent notes falling downwards and a new note is read from the note file and inserted into the table. After the new note is inserted, the code instructs to draw the new up to date table onto the screen depending on whether the input is correct or not.

Audio Mixing:

We implement the audio mixing in software because it only consists of two tracks: background and guitar. The background music plays throughout the entire game play. The guitar track plays as long as the player continues to hit the notes correctly. The mixing is done by simply adding the two sample values together than dividing by two. Both samples are mono, so we must copy them to both left and right output channels.

Approach, Design Partitioning and Methodology:

After initially looking at the "Frets on Fire" code, we determined that it would be very difficult to port directly. So we wrote basically everything ourselves. The only part that was taken was the notes. Even with that we weren't able to read it with the game directly and had to translate them into our own format.



This is a diagram of our hardware/software split. The single PPC controls all the hardware modules through the dotted lines while data is passed along the OPB and PLB bus with solid lines.

Images:

All images were saved in bitmap format and edited with Adobe Photoshop. This includes the backgrounds, normal and held notes, incorrect note “X”, correct note “flame”, score numbers, and scoreboard.

Frames per Second:

Seeing as that the nature of Guitar Hero calls for a lot of precision between watching the notes scroll down and hitting them correctly, having a fairly high frames per second of our game was essential to the playability of it. The frames per second was initially limited by using the software frame buffer clear. After implementing the hardware clear, we were able get a fast enough frames per second to implement the audio and video in sync.

PPC’s:

Initially, we wanted to use 2 PPC’s in parallel because our initial audio files were so large uncompressed, we wanted to use a compressed format and decode on the fly while the other PPC is used to run all the video, and game code. After talking with a team from last year (the Bomberman group) that tried to do this, we learned that they had failed because of difficulty with synchronization. Thus we decided this was not a good way to proceed and that we should try to use one PPC because we were more certain of it working. Also, we later managed to decrease the audio files tremendously so it was not necessary to keep them encoded and lessened the need for a second PPC.

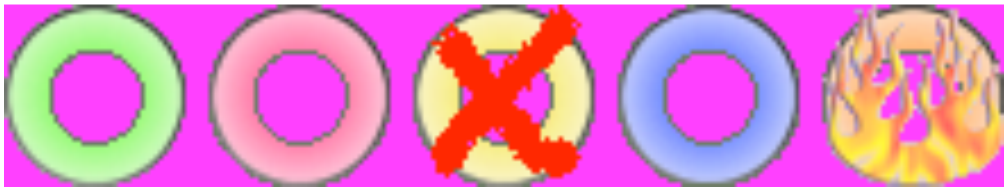
Transfer Rate:

The transfer rate from the compact flash cards is really slow. Our initial bitmap and audio read program given from the labs was very slow because it would read byte by byte and fixed the endianess as it read. For our game, we changed this to read a long range of bytes consecutively so there was not delays and ended up with around 30MB/s transfer rate. This was fast enough with our down sampled audio files and ended up taking about 40seconds to load the two tracks for a given song.

Drawing Notes:

The notes are separated by the number of empty notes and are obviously different for different songs. Since the songs have different lengths and speeds, we had to implement a system where any song can be played; therefore we decided to allow a dynamic amount of blank space between every note. This is very important, because it allows the player better visibility for the notes being played rather than all the notes being squished together. This also decreased the number of notes being drawn per screen which increased the speed and thus creates a higher frame per second. Our ratio was picked to allow the calculation of frame per second to be around 20 and 25.

One problem that we had to overcome was the scrolling of an actual image. Because it was difficult to have the circle itself as the image, we had a background with it which we made transparent. We did this by making the background hot pink, which is a color that will never be used in the game play, and whenever the hexadecimal representation (0xff00ff) of that color was read in, it would jump over it, thus making it transparent. Like the Mario Lab, we also implemented triple buffering so that the actual motion of the circles was not as choppy.



Frame Buffer:

We implemented a triple frame buffer system to draw the video to the screen, one frame is drawn, one is being edited, and one is being cleared by hardware. The frame buffer is swapped after the frame being edited is complete. Each buffer is 2MB to allow correct alignment however only 640x480 of the pixels are actually displayed onto the screen. After we got the hardware module clear working, we didn't have any other problems with the frame buffers. It was fast enough to properly implement our songs. About halfway through the semester we decided that extending the frame buffer would look nicer and enhance game play. This allowed the notes to scroll on to the top of the screen as opposed to just appearing on the top row, which makes game play less awkward and more like the real Guitar Hero. We implemented this simply by drawing above where the frame buffer actually started.

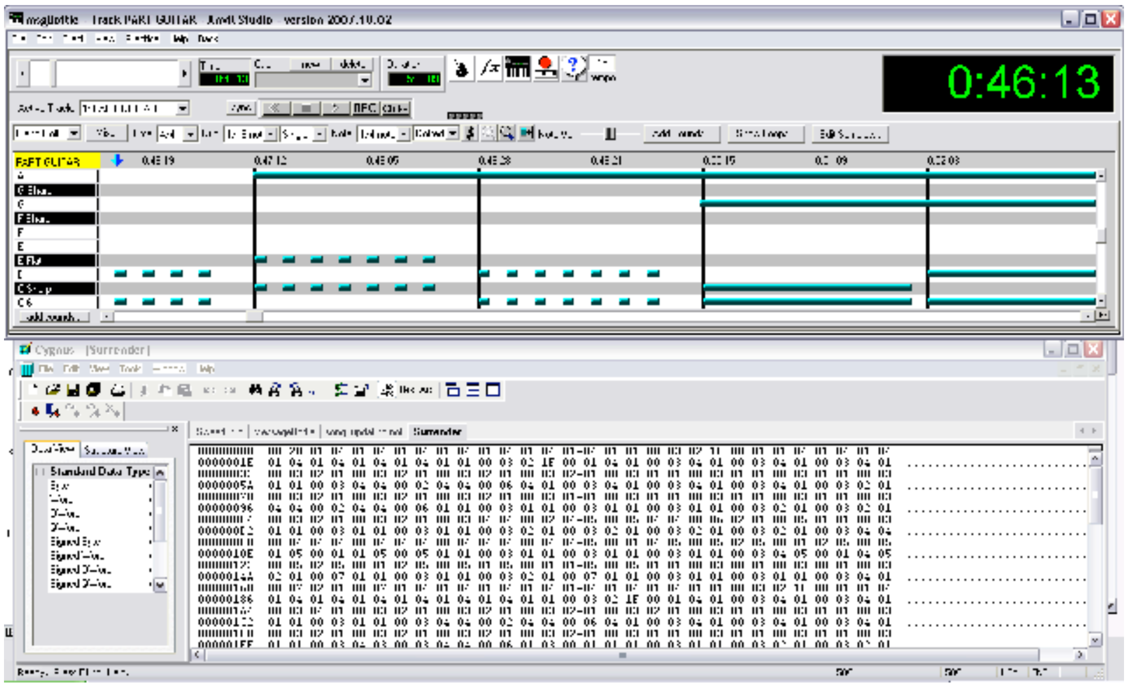
Tools:

We used Anvil Studio to convert the .wav music files to MIDI format. Although we used this program for the majority of the project we found it very difficult to use. The scrolling did not work well at all and graphical way the notes were represented was not as nice as other programs we found later. One such program was "Fruity Loops." This was a high quality, expensive program that more than met our needs for visualizing the notes and note lengths in the MIDI files. Cygnus was a simple hex editor program that we used to write the note stream after manually reading the MIDI files using Anvil Studio or Fruity Loops. All the graphics were edited and resized using Photoshop. All the coding and software testing was done in Xilinx. Some hardware emulation was used for the psx_module and was done in ModelSim. Both the psx and the memclr hardware module were then debugged using ChipScope.

Testing and Verification Methodology:

Writing Note Streams:

This was one of the most tedious parts of the project. In order to write a note stream that the code could read we had to view the MIDI file using Anvil Studio or Fruity Loops and extract the notes and note lengths in a hexadecimal format <8 bits for note> <8 bits for note length>. With each song we were able to choose the difficulty we wanted to use. For easy songs we would start at C5 and read up 4 more notes (to correspond to the 5 frets we had on our guitar, for medium songs C6, and difficult songs C7. Initially we had thought we could write a script to do this for us, but after many attempts and discouragement, we decided that we had to manually do it if we every wanted to have songs before the final project demo. Each song took 3-4 hours to manually decode and with each song there were errors that had to be found by playing the game. Since two members of our team, Langtian and Chengjou, are avid Guitar Hero players they were able to play the song using the note stream that was manually written and determine how in synch it was and at what parts the notes were completely off or lagging.



```
[illegible]
0000001E  01 04 01 04 01 04 01 04 01 01 00 03 02 1E 00 01 04 01 00 03 04 01 00 03 04 01
[illegible]
0000005A  01 01 00 03 04 04 00 02 04 04 00 06 04 01 00 03 04 01 00 03 04 01 00 03 02 01
[illegible]
00000096  04 04 00 02 04 04 00 06 01 01 00 03 01 01 00 03 01 01 00 03 01 01 00 03 02 01
[illegible]
000000D2  01 01 00 03 01 01 00 03 01 01 00 03 02 01 00 03 02 01 00 03 02 01 00 03 04 04
[illegible]
0000010E  01 05 00 01 01 05 00 05 01 01 00 03 01 01 00 03 01 01 00 03 01 01 00 03 04 05
[illegible]
0000014A  02 01 00 07 01 01 00 03 01 01 00 03 02 01 00 07 01 01 00 03 01 01 00 03 04 01
[illegible]
00000186  01 04 01 04 01 04 01 04 01 04 01 04 01 01 00 03 02 1E 00 01 04 01 00 03 04 01
[illegible]
000001C2  01 01 00 03 01 01 00 03 04 04 00 02 04 04 00 06 04 01 00 03 04 01 00 03 04 01
[illegible]
000001FE  01 01 00 03 04 03 00 03 04 04 00 06 01 03 00 01 01 01 00 03 01 01 00 03 02 01
```

This is a screenshot of Anvil Studio above with the note stream of the MIDI file on the top half and the hex editor –Cygnus, we used to record to notes and note lengths on the bottom.

Synchronization:

This is one of the most difficult parts of our project and we did not realize that synchronization required the hardware module to be completed until DR3. We first calculated how many notes were presents in a song and matched that with the length of the song it self. From the length, and the spacing of the notes on the screen, we were able to determine the necessary frames per second required to draw all the notes during the song duration. From this, we calculated two values, one was a time for audio that was determined by the rate of the audio and the offset at the current state audio_time = audio_offset * 11025 Hz. The other was a time for video that was determined by the current frame number, and the frames per second. Video_time = frame_num * fps. The audio_time and video_time must match in order for the video and audio to be in sync. In order to do this, every time the audio_time is ahead of the video_time, the video must skip frames to try to catch up. If the video_time is ahead of audio_time, it must add more delays to slow it down. The controller is synced directly with the video by checking the input every frame. Since it is very difficult to hit the notes on the exact frame, our code allows a tolerance both before and after the correct timing to allow the player more room for error. Also, because the number of notes directly affects the amount of time it took to draw the frames, we had to manually add delay to frames with fewer notes so it would not skip too often.

Miscellaneous Issues:

We had some issues with the held notes (a note that is held for an extended period of time during a song) being unable to determine whether the previous notes were correct/incorrect. Because of this problem, players can re-hit the held notes during the held duration. This problem would have been fixed if we had more time.

Initially we had a noticeable amount of static during our game play because the software frame buffer clear took too long and the FIFO was empty before the next frame started. But after we implemented the hardware module and triple buffering, this problem was resolved.

What We Learned:

We should have spent more time writing a program to decode our notes from the MIDI format for us. This would have saved a lot of time and energy, seeing as that it was such a tedious and cumbersome task to do manually. With an automated system we should also have try to make a program to verify individual note files to their midi counterpart. If we succeeded with this early on we would have been able to add more songs to our game.

Our synchronization strategy could have been better also. If we came up with a better algorithm or something that would allow definite syncing without having to play through and fix individual notes that were out of sync, it would be a lot easier to include more songs that are decoded already.

Words of Wisdom for Future Generations:

Hardware modules take a LONG TIME to debug, so start early. Also talk to the TAs a lot because their opinion is more valuable since they probably had prior experience with most of the problems you are going through.

Website URL:

http://www.ece.cmu.edu/~llang/545

References:

http://www.fretsonfire.net/cgi-bin/ikonboard.cgi?act=ST;f=3;t=7251
http://www.gamesx.com/controldata/psxcont/psxcont.htm
Peter Nelson
Xilinx Documentation

Chengjou Liao's Individual Contribution

1) Note Generation (Time spent: Approximately 7 weeks, 10 hours/week on average)

- Did research on the Frets on Fire open source version of Guitar Hero game, specifically the generation of on-screen notes during game play. As a result of this research, I made the decision to use the MIDI format files existing on the Internet to implement songs into our game.

- Developed proprietary format of note files for our game's usage.

- Initially tried to write a script to translate the MIDI files to our proprietary format, but this method proved to be ineffectual because of the disparities between the original file format and what we needed.

- Instead, manual method of translation was used; this work was shared with Shaheen.

2) Video and game code (Time spent: Approximately 5 weeks, 10 hours/week on average)

- Worked with partners to establish visual game design and structure

- Developed graphics with Shaheen using Photoshop and existing graphics

- Worked with Lang to develop video code and interaction with game code

- Adjusted existing game and video code to properly synchronize for game play

- Debugged video/game code

3) Reports and Presentations (Time spent: Weeks these were due, approximately 3 hours/week)

- Wrote all parts regarding note generation

- Edited and rewrote subsequent iterations of initial report

- Gave Design Review 1 presentation

Class impressions/improvements/complaints

- The amount of structure set in place at the beginning of the semester was just about right for planning and execution of the project

- Instead of having to write a full-on report of the *Pentium Chronicles*, a few short and specific prompts to answer would have been preferable.

- Recurring in-lab demos were crucial for motivation on completing goals

Shawn Ranjbaran
Individual Contribution and Class Impressions:

-Did research for the original game we wanted to create (Tank Commanders) and then more research on Guitar Hero and studied Frets on Fire code. (2 weeks/ 6 hours per week)

-Graphics: Gathered all the graphics we used and used Photoshop to edit them. This was particularly hard for me because I had never used Photoshop before. (4 weeks/ 5 hrs per week)

-Graphics: Worked with Lang to get the note stream to draw correctly on the screen. (6 hours)

-I/O Portion: With Lang implemented the FSM for the controller and the code which gets data from the controller and draws notes to the screen. This was a very long process which included using a high speed then a low speed parallel board and debugging with Chip Scope and ModelSim. (15 hours per week / 2 weeks)

- Helped write and debug a PLB module to clear memory address at a specified value and does this by talking to DRAM (w/ Lang) (3 weeks/ 8 hours per week).

-Helped write some of the game code that read the note file. I ended up doing some calculations, some which helped and some which did not, that attempted to synch up the audio and video. (10 hours)

-Manually decoded 2 of our 3 songs from MIDI to our note structure. (10 hours)

-Helped adjust video and audio code to work in synch with the note steam files we wrote. Before touched the code we tried many ineffectual ways before coming to a reasonable way of testing the note stream versus the video and audio. We ended up using Fruity Loops, an audio timer on the screen with the game running, and an open window the note stream file to catch errors and places where the song lagged as we played it. Discovering that way took a long time and once we realized how to do it, it still took a lot of time to debug. (20 hours)

-Wrote up the Proposal and different parts of DR1, 2, and 3 that I worked on. (8 hours)

-Gave the Proposal Presentation and the DR3 presentation. (prep time 3 hours)

-Wrote the final report. (5 hours)

We wanted to chose a project that would keep our interest and one that we would have a lot of current references to, ie: Frets on Fire. I did not play Guitar Hero before this semester but it really helped that my group members had extensive experience with the game, especially for testing purposes. In the long-run I really liked how the design reviews pushed us to work hard throughout the semester and in the in lab demos made it necessary to have something working rather than to talk about it in high level.

I loved the open-ended nature of the course. I like that no real restrictions or boundaries were in place which motivated us to do something unique and exciting rather than just meet set requirements. The labs were really helpful and the TA's were really great, but I do have one suggestion. Have a pool of different labs and let people choose 3-4 that are pertinent to their project. Still, I understand that it is useful to learn about things that you didn't expect to use, but in some cases it might waste valuable time in a busy schedule. Finally the *Pentium Chronicles* assignment was insightful and interesting, but I would have rather read photocopies of important portions of the book than to buy it and use it once. All in all, I recommend the course as a capstone or even a 2nd capstone for people interested in digital design or hardware and had a lot of fun. It was such a satisfying feeling to know we made a real game that people enjoyed to play, from scratch!

Langtian Lang

Initially I was assigned to do the controller IO and the audio mixing module, both of which were designed to be done in hardware. However throughout the first couple weeks of the course, we did not receive any information on the hardware side of things, and thus delayed the progress of the psx_module until after lab5, which taught as how to implement hardware modules. Before lab5, most of my time was spend researching the FSM needed to talk to the psx controller properly, setting up the physical connection with the board, and helping Shawn with the video drawing part of the project.

By DR1, I had very little success with the controller. By the demo day, Shawn and I had a FSM designed in verilog and was ready to be simulated in Modelsim. The physical PSX extension cable was stripped down and plugged into one of the expansion boards attached to the High Speed I/O port of the Xilinx board. After simulating what seemed to be a correct output wave form in both ModelSim and Chipscope, the FSM was still unable to talk with the controller properly. At some point, I believe Peter Nelson suggested that we try one of the low speed parallel boards, similar to how he implemented his controller module. I changed rewired the controller and connected the software ports, and everything magically worked. We believe the problem might have been caused by the fact that the connections on the parallel board included the necessary resisters build in. By DR2, the controller was able to send signals across the bus and read by the PPC. Shawn and I implemented the necessary software to grab the data from the controller and was able to display the notes properly on the screen. The controller was in sync with the video by around DR3.

The next big obstacle for us was the audio. The wave files that we had were CD quality at 44,100 Hz 16bit stereo. This created files that were 3-4 minutes long with size over 40 MB. We knew this would be a big problem and started researching other ways to implement the audio files. My initial idea was to try to find a hardware implementation of a decoder of MP3 or Ogg Vorbis. This search turned out fruitless. The second plan was to try to use the other PPC to implement a software decoder that can decode on the fly while the other PPC controlled the rest of the game code. However, I was unable to get the linker properly setup for both PPC projects to run in parallel. The big breakthrough in audio came when I realized we can down sample the existing wave file. CD quality is definitely note a necessity so I messed with the frequency and eventually brought it down to 11024 Hz, which is only barely noticeable. The stereo tracks were also unneeded and would save up half the space after I simply removed the right track. At this point, the files were about 2-3MB each, and were much more reasonable.

While writing the software for loading the audio, I noticed that Sound reader program given by the lab was very inefficient in that it loaded single bytes at a time and implemented endianess swapping as it loaded. I changed the code to load all the bytes sequentially and this increased the throughput by 200-300%. Endianess fixing would be done as the samples are played, which did not affect performance much. I implemented a hardware sample adder but that did not seem to make a big difference simply because we had only two tracks to mix. In fact it is probably faster to do a single software add than sending the data across the bus, wait for the hardware to finish adding, and then reading it back. By the DR3, audio was working properly in parallel with the video by filling the codec FIFO before every frame that is drawn, however there was noticeable static. I tried to fix this problem by increasing the length of the audio FIFO that came with the Xilinx however, later on I realized, with the help of the TAs, that I can fix this by simply making the frame draw faster by implementing a hardware frame clear module.

Video implementation was started by Shawn and me very early on. We had a simple note drawing system by DR1 that used a table of notes that represented the screen. By shifting the table, and redrawing the frame, the notes appear to move down wards. At this point, we had used static delays between each frame to allow an almost static frame rate. By DR3, I had implemented the extension of the frame buffer to allow notes to more smoothly appear near the top of the screen. At some point after DR3 I realized we had to implement a hardware module for frame clearing in order to allow a more reasonable frame rate and remove the audio statics. I spent about 3-4 weeks creating a master PLB module that can talk directly with the DRAM to clear the memory address with a specified value. This hardware module took a long time because debugging it was very painful, Shawn assisted with it as well. It was hard to find all the signals in the documentation to be able to read the output from the Chipscope PLB debugger. Eventually, I got this working, and implemented a triple frame buffer system so as one frame buffer is being shown, another is being drawn, and the 3rd is being cleared by the hardware memclr module. After this fix, the audio problem seemed to be fixed, and also I was able to increase the rate of the drawing tremendously. A little bit of delay was included to keep the hardware memclr module from flooding the bus and causing a little bit of jitter.

Game code was first started around DR1 with Shawn and consisted of reading of the note files. I implemented the software to read the note file and draw them consecutively to the screen based on the length of each note. After the final DR, I realized, with the help of TAs, that in order to keep the audio and video in sync, calculations must be made every frame to make sure they're not speeding up past the other. I implemented the synchronization by calculating the time of the frames which was note by the number of frames drawn and the rate of the frames being drawn, and also the number of samples played by the audio, and the rate the audio is played. If the two were to ever get out of range of each other, the video would delay, or skip frames until it matched up. This system worked fairly well. The only problem was the preciseness of the frames per second. I had to manually tweak this and the note files in order to sync the notes up completely. This took a long time and in the end I was only able to finish 3 songs.

Overall this class was a lot of fun, although a large percent of it were the moments when I realized I did something totally wrong. Without the help of the TAs, we would not have been able to complete the game. The labs were tremendously useful. I understand this year they were being written up as we were assigned them, but I feel all of them should be opened to the students immediately so they can start implementing hardware without a lot of delay. The amount of problems I had on writing the few hardware modules showed how little I know of actually writing physical hardware modules, rather that something Modelsim can simulate. I wish our previous hardware classes like 240 and 447 were able to better prepare me for this class.