February 2009

Robotic Bass Player

Adam J. Teti
Worcester Polytechnic Institute

Project Number: MQPWZM-RB08
MQPECC-B708

The Thumper Robotic Bass

A Major Qualifying Project Report

Submitted to the Faculty

Of the

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

_____

Matt Brown, Electrical and Computer Engineering, Class of 2008

_____

Barry Kosherick, Electrical and Computer Engineering, Class of 2009

_____

Adam Teti, Mechanical Engineering, Class of 2008

April 18, 2008

_____

Prof. William R. Michalson, Co-Advisor

_____

Prof. Eben C. Cobb, Co-Advisor

Abstract

The Thumper Robotic Bass MQP is an attempt at creating a self-playing bass guitar that responds to MIDI input as well as input from an electric guitar to provide accompaniment. By bringing together a team of electrical and mechanical engineers, the project was structured to emulate an engineering design as encountered in industry.

Table of Contents

Table of Figures

Introduction

The Robotic Bass project, codenamed The Thumper, is a two major Major Qualifying Project (MQP), involving two Electrical and Computer Engineering (ECE) majors and a Mechanical Engineering (ME) major. The project was originally intended to detect the bass notes of a musical recording and then reproduce these notes by physically playing a standard bass guitar. Due to time constraints, the project was redefined to instead react to input from a human musician playing the guitar and to accompany the operator, and was again redefined to allow remote controlling of the bass. To prepare for the project, the Electrical and Computer Engineering majors completed a Pre-Qualifying Project (PQP), which contained the background research and possible algorithms to be used in the project. This report describes the design philosophy and logical processes used to design, build and test The Thumper.

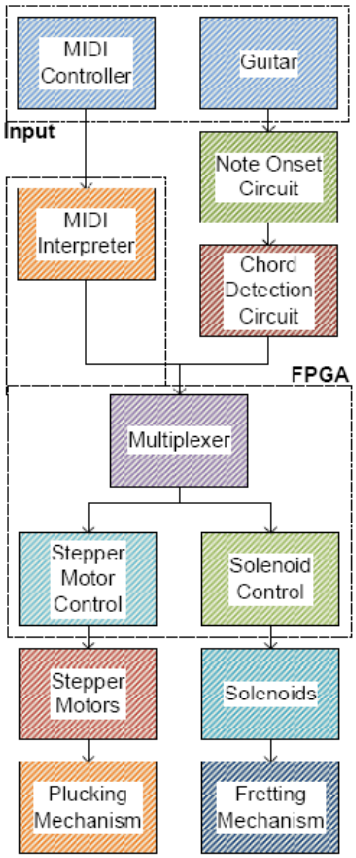

Figure 1 - Top Level Block Diagram

Problem Statement

To design electrical and mechanical systems that can accurately and repeatedly fret notes on a standard electric bass guitar and pluck appropriate strings in a human-like fashion based on a midi or electric guitar input.

Background

The concept of automated musical instruments is not a new concept by any means ranging from the simple music box, to player pianos, to advanced robotic drum kits. The early music boxes used a spring loaded barrel with extrusions on it to strike tuned fingers and play whatever tune was inscribed on the barrel. This concept of using a tuned barrel or roller to strike fingers led to the development of one of the most well known automated musical instruments the player piano. From 1847 through 1876 different designs were created attempting to create an automated piano device. The designs resulted in a multitude of patents and ultimately in 1876 what is accepted as the first player piano, unveiled at an exhibit in Philadelphia, Pennsylvania.

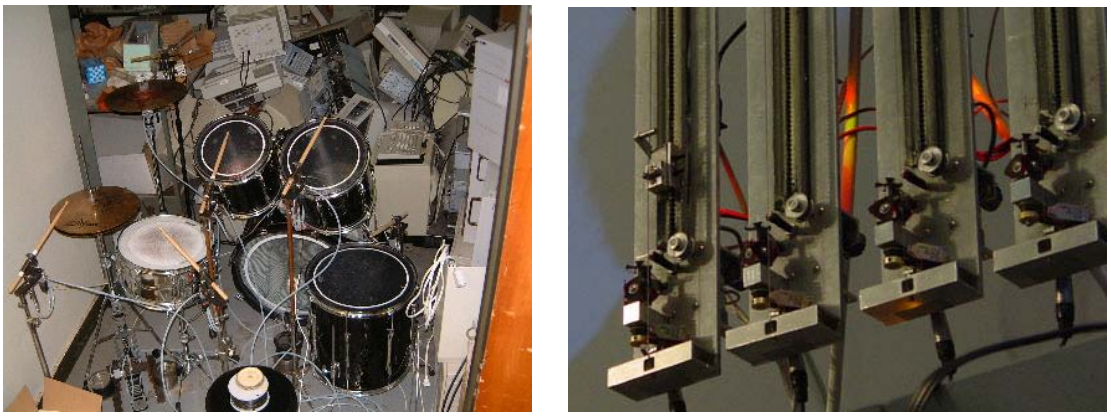

Figure 2 - (left) The P.E.A.R.T. from Univ. of Louisiana and (right) the LEMUR GuitarBOT[1] [2]

The program accepted by the player piano were encoded rolls of paper with holes punched in it one for each note in the song to be played. The holes allowed air valves to be opened or closed which in turn actuated the pianos keys and played the song on the roll. Jump ahead more than 150 years and the paper rolls were replaced with computer memory files programmed with Musical Instrument Digital Interface (MIDI) information where a simple byte of information contains which note is to be played, for how long, and how loud. Using MIDI programs as controls a multitude of automated instruments have been created notably the *Pneumatic and Electronic Actuated RoboT* or P.E.A.R.T., and the GuitarBOT. The P.E.A.R.T. created by four students from the University of Louisiana at Lafayette as a senior design project. The robot was actually a set of pneumatically actuated drum sticks and pedals mounted onto a standard drum kit. The kit is controlled by a computer with a song programmed into it in the MIDI language. The GuitarBOT was created by the League of Electronic Musical Urban Robots and consists of a set of strings with sliders and rotating pick mechanisms controlled via a midi controller operated by the musician. Though it is not an actual guitar the sliders and picks can be used to emulate the sound of a guitar fairly accurately.

[1] Graffagnino, Frank. "Frankie Graffagnino - Projects :: P.E.a.R.T. - the Robotic Drum Machine." Graffagnino.Net. 16 Apr. 2006. University of Louisiana At Lafayette. 2 Apr. 2008 <http://www.graffagnino.net/wwwpeart/>.
[2] Magnusson, Thor. "November 16th 2005: Intelligent Tools in Music by Thor Magnusson." Artificial.Dk. 16 Nov. 2005. 2 Apr. 2008 <http://www.artificial.dk/articles/intelligent.htm>.

Figure 3 - Still from Animusic 2 DVD "Resonant Chamber"[3]

Along with automated instruments of years past much of the initial inspiration for this project came from a computer animated video called *Resonant Chamber* part of the *Animusic 2* DVD. This video features a multi-neck stringed instrument playing all by itself no real control mechanisms can be seen due in part that this is just an animation and therefore did not give much more than just a conceptual idea.

The original concept for this project was not directed at a specific market more just an idea for melding technology and music two passions we all shared. After some research and thought we found that this project does have a market in the musical community. Currently the most automated music comes out as synthesized sound from MIDI devices. With hours of effort this can sound identical to the instruments that would otherwise be playing the selection. To many musical purists this is not quite the same as hearing the real instruments being played. This project along with others like it could create entire concerts with automated musical instruments accepting MIDI input from one centralized MIDI controller. Systems of this size do exist but synthesize the sounds they create notably the Virtual Orchestra created by WPI professor Frederick Bianchi.

Economics and Target Market

The budget for this requirement is something that we have been concerned with for the entirety of the project, as we were given a total of $530.00 from the ECE and ME department shops. The budget required for this project is much greater than that, so alternative funding was required. There were many aspects of this prototype that were more expensive then a full production model. For example in a full production model, the frame would be made out of extruded plastic and not 80/20 aluminum.

[3] Allin, Jeremy. "Animusic 2 (US - DVD) in Reviews." DVDactive. 4 Jan. 2006. 2 Apr. 2008 <http://www.dvdactive.com/reviews/dvd/animusic-2.html>.

80/20 was chosen for a prototype because of its ability to be modular and easily adjusted. Everything aside from the solenoid mounting plate would not be made out of aluminum in a full production model. Aluminum was chosen for the prototype because it was readily available in the Higgins Labs machine shop. The stepper motor plate would not be made out of ¾ inch acrylic, as seen in the prototype. This was used because we found it in a scrap pile in the machine shop.

We intended a full production model to cost much less than the prototype. This was because of the target market. We feel that there would be a small market for this product. We believe that audiophiles and audio purists would much rather this device over standard MIDI devices, because unedited MIDI can sound unnatural especially when the voice is changed. This device plays a physical bass, so the acoustics will be much more accurate than a digital representation. Musicians would also find use for this product because it always shows up to practice on time and does not take up more space in the van. Granted this product does not the creative element that a human musician adds, but it can be used just to provide accompaniment in case the bassist goes missing or continually shows up to practice late.

Knowing our target market we can properly assign a retail value of the product. We feel that a price range between 500 and 1000 dollars would be adequate to cover for the cost of manufacture and materials in addition to being able to turn a profit. We feel that this price range would allow the majority of musicians to be able to afford this, because when compared to the price of other musical equipment, it is comparable to medium quality instruments. While every guitarist may want a 1959 vintage Gibson Les Paul, they easily cost over $4000, and will usually settle for the Fender Stratocaster for $750. If this product was priced similarly we feel it could convince musicians to purchase it. This price range would also be adequate for audio purists and audiophiles, as they have been know to spend large amounts of money to achieve the best sound.

System-Level Requirements

At the beginning of the project, we decided on requirements that this project must complete. The most important requirement is to accurately produce music using this device. This would entail that the sound output be very similar if not indifferent to a human playing a bass guitar. Additionally there are many system requirements including safety, manufacturability, low power draw, and rapid response time. Safety is important because musicians would most likely use the apparatus for things it is not meant for such as a drink holder or support when moving around the stage. Injuring the users or the device would ultimately hurt the sales and marketing of this product. The ability to actually make a design is very important. If you have a great theoretical design, but features impossible angles, cuts or sizes, it is actually a poor design, because it will work in simulation only. Power consumption is important because large current or voltage spikes can cause damage to the components. Rapid response time is important because it will allow us to play notes in rhythm with the user input.

There are many specific requirements for individual systems. The fretting system needs to be able to apply and maintain appropriate force on the strings. This system must travel at least three quarters of an inch in under a hundredth of a second. The fretting system must cover the first four frets

of all strings on a standard electric bass in order offer coverage of at least one full octave. To reduce large current transients, we decided that only one note can be fretted at a time.

The plucking system needs to be able to apply lateral force to one string without contacting adjacent strings. All of the strings need to be able to be plucked at the appropriate times. In order to reduce power consumption, only one string may be plucked at a time. We determined that the fastest we would need to play is four times a second, which is close to the maximum speed of an average bassist. This system needs to be encoded, which would allow us to know exactly where in the rotation the plucking devices are, so that we can accurately play notes in rhythm.

The control system needs to be able to process the input at a rate fast enough so that the output from the system matches the outputs from the musicians. The other systems must be able to properly receive control signals. We must be able to determine when a note starts, so we can allow for adequate time to process the signals. This system must be able to correctly identify the notes that are input to the device, so that we can play the appropriate accompaniment.

Electrical Design

Note Detection

Figure 4 - Note Onset Block Diagram

Knowing when a note begins is of high importance to us, as it gives the rest of the systems the green light. When the note onset detection system would trigger a high, it would send signals to the other systems, telling them to do their function at that time. This system would need a very small latency, in order to play the proper notes in rhythm with the other musicians.

One of the papers they looked at involved a method for finding in recorded music, when in the time domain, a musical note begins[4]. "A Hybrid Approach to Musical Note Onset Detection" by Chris Duxbury, Mark Sandler, and Mike Davies, details the methods that they considered and the approach they eventually decided on to detect the onset of musical notes. They were looking for an algorithm that could accurately detect signals from many types of instruments. A guitar has an onset which appears in a large part of the frequency domain, with components that decay very rapidly at higher frequencies. These wide-band onsets are relatively easy to identify, as the onset information is easily pulled from the signal using some frequency filtering and an appropriate threshold function. On the other hand, the

[4] Chris Duxbury, Mark Sandler, and Mike Davies. "A Hybrid Approach to Musical Note Onset Detection." Proceedings of the 5th International Conference on Digital Audio Effects (2002). 2 Oct. 2007 <A Hybrid Approach to Musical Note Onset Detection>.

authors make it clear they would like to find slower onset notes as well, like those made by a violin (Figure 5 - Comparison of Guitar and Violin Onsets[2]).

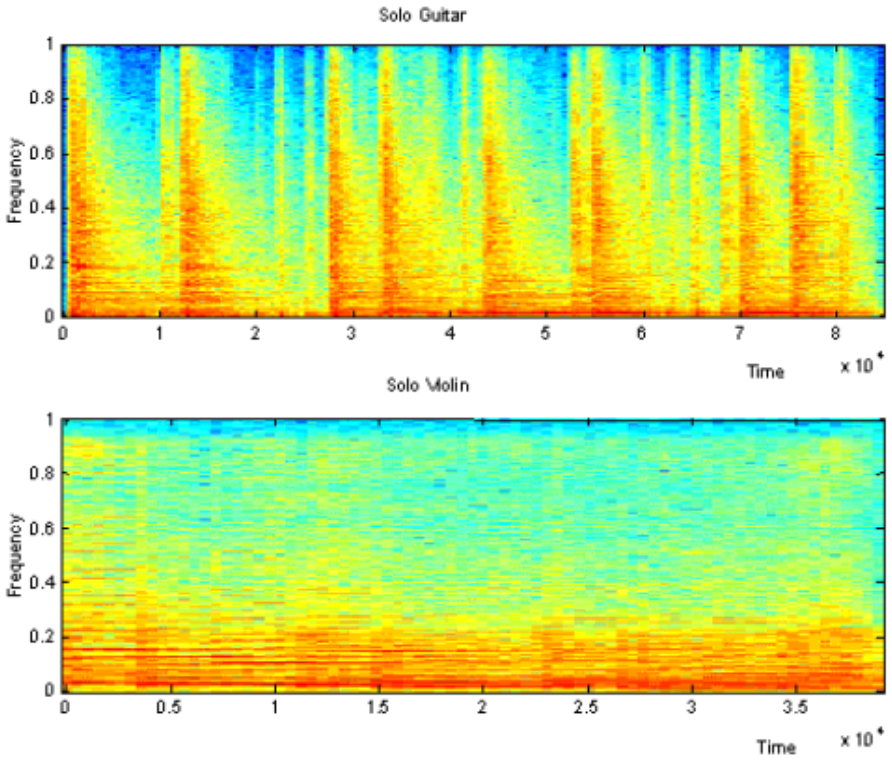

Figure 5 - Comparison of Guitar and Violin Onsets[2]

The authors of the Note Onset Detection paper explain the observations they made while pursuing their goals. They began by breaking the frequency domain they were working with (The general audio range is 20 Hz to 20 kHz.) into 5 sub bands. They found, when looking at the frequency components in each sub band, which the upper sub bands contained most of the onset information, in the form of short bursts of energy that quickly decayed. The lowest sub bands, however, did not contain the same kind of note onset information the higher sub bands did, as it contained most of the steady state portions of the signal. In the highest sub band, the onset information decayed so quickly, the authors decided they could eliminate it to reduce computational cost. This left three sub bands in the range 1.2 kHz and 11 kHz. When looking at signals in the time domain that has been filtered into these sub bands, the effect of the onset of the wide-band onset instruments is quite clear. By looking for significant sudden increases in the energy in these sub bands from sample to sample, many note onsets can be found with acceptable accuracy.

The authors continue to explain that while this works well for the wide-band onset signals, the slower onset instruments will not trigger this type of system as an onset. They propose that a short-time Fourier transform be applied to the sub bands in question, and the comparison of the transient energies of the sub bands can indicate whether or not a note is being played with better detection for slower

onsets. In the lower sub bands (0 Hz – 2.5 kHz), they propose a distance measurement which was determined by measured the distance between the vectors of a fast Fourier transform. Only the positive values are taken to keep the system from triggering for a rapid decay. By normalizing the function, the system reacts better to slow onsets as well as retaining its performance for harder onsets.

After discussing the different types of weighting functions they tested, the authors decide on an exponential weighting function and the final equation is given in equation 1. They account for this by stating that the exponential weighting puts more emphasis on events that have occurred more recently, but still allows past events to affect the results. They felt this gave the best results based on their experimentation.

$$ons(n) = SE(n) - \sum_{a=1}^{A} \frac{SE(n-a)}{a} \quad (Equation\ 1)$$

The writers of the Onset Detection paper also worked on an automatic threshold algorithm for their detection function. They express a belief that while a manual threshold will always perform better than an automatic threshold, there are many applications in which the automatic threshold would be more desirable and, in some cases, necessary. By using probability distribution functions, they attempt to determine when onsets are more likely. By looking at when onsets occur where they are more likely in the probability distribution function, the authors found their ideal threshold. By resetting the threshold every five seconds in their tests, they were able to obtain satisfactory results from the automatic thresholding, and state that the threshold reset can be set for any size window. By weighting the results from each sub band and adding them, the writers achieved success with the detection algorithm over the range of music tests they attempted. Their automatic threshold worked very well for music with sharp wide-band onsets, but did not fare well in music that contained only bowed string instrument.

The ECE majors are intrigued by this paper, and believe that many of the concepts can be adapted and applied to their project. The sub band energy approach to onset detection was determined by the authors of the paper to work well for wide-band onsets, which include the guitar. By redefining the project to react to a guitar player instead of recorded music and using the sub band energy comparisons, we may be able to find relatively accurate timing data. If combined with a way to determine the notes being played by the guitarist, they would have all the information they needed to be able to play the proper notes to accompany the musician.

This system is designed so that once the input goes above a certain value; the note onset circuit will flag a logical high, which will be sent to the controls system. This will then tell the rest of the appropriate systems to fire. Originally we tried to realize this system within MATLAB. The methodology used was that we would record a couple milliseconds on data into an array. The maximum function would then be applied to it, and if the maximum was above a certain value and higher than the previous maximum by a certain percentage, an onset would be flagged high.

The system that we are using to detect onsets is not a digital solution, but an analog one. We were struggling with MATLAB to make this function work in real time, and do so with very little error. After some discussion and simulation we decided that a comparator would be a better choice for implementation. We felt this way because it was a simpler solution, which would flag an onset earlier. In the MATLAB code, we would get the very highest peak of the note, whereas in the comparator we would get the first peak above a certain value. The main problem with using the comparator is that we cannot send only one onset to the controls system; it will flag high every time the signal goes over the specified value. However this can be overcome in the controls system by having it ignore onsets for a certain amount of time after receiving the first onset. This does limit the speed, at which notes can be played, but it only does so to roughly the limit a human can play and only to point were two notes can be identified as two separate notes. The average human cannot distinguish the separation of two notes that are played less than fifty milliseconds apart. We have specified that the fastest we will ever run this machine is at two hundred beats per minute. For comparison, the song "Flight of the Bumblebee" is usually played around this speed of two hundred beats per minute.

Circuit

The input to this system is an electric guitar played through a Marshall Lead 12 amplifier head. This amplifies the signal from the millivolt range to the volt range. In addition it will provide the user the ability to hear what they are playing and condition the signal to optimize its processing. The comparator we chose to use is the LM311, as it has a fairly quick response time of 0.1 microseconds and accurate down to 7.5 millivolts. The output of the comparator is connected to the FPGA through a unity gain buffer, used to protect the FPGA from current spikes coming from the signal or comparator.

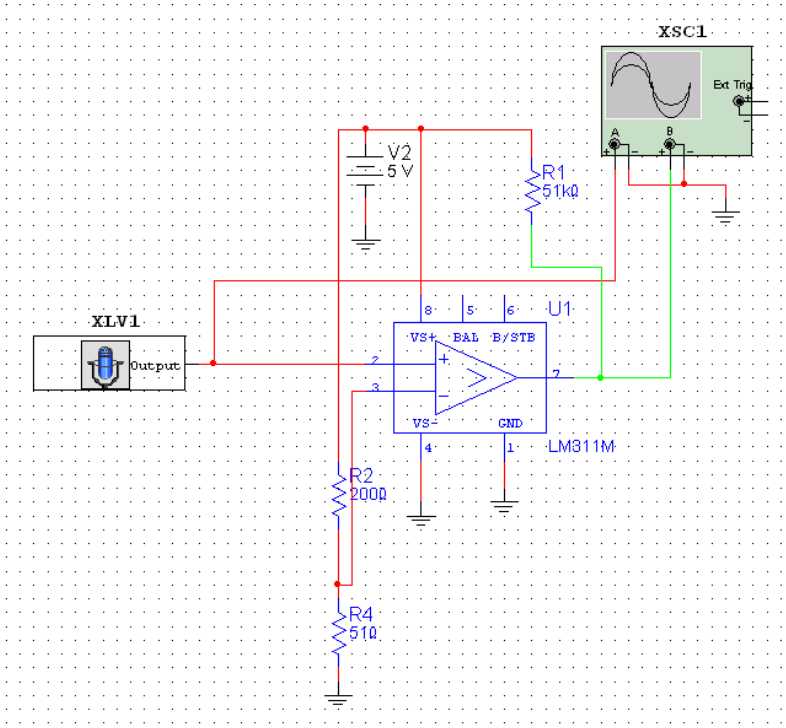

Figure 6 - Note Onset Circuit

Circuit Simulation

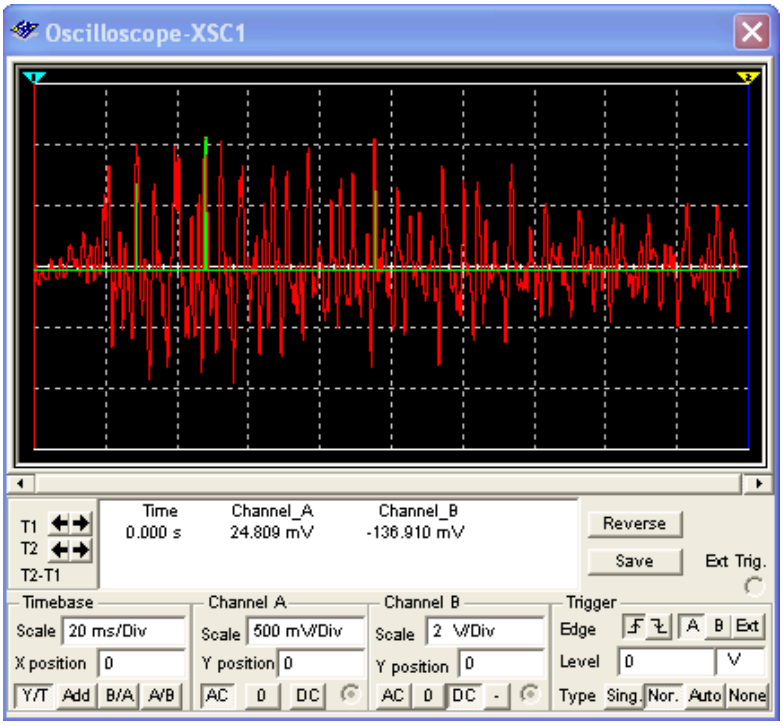

Figure 7 - Note Onset Simulation Results

We tested this system in Multisim, using the National Instruments DAC and Simulink; we were able to properly simulate comparator outputs from a real guitar input through the sound card. When simulated we see the green spike from the output flagging high whenever the input is greater than four volts. We do see that there is more than one output flagged high. They are at most fifty milliseconds apart which approaches the limit where the average human can discern two separate notes.

Layout and Fabrication

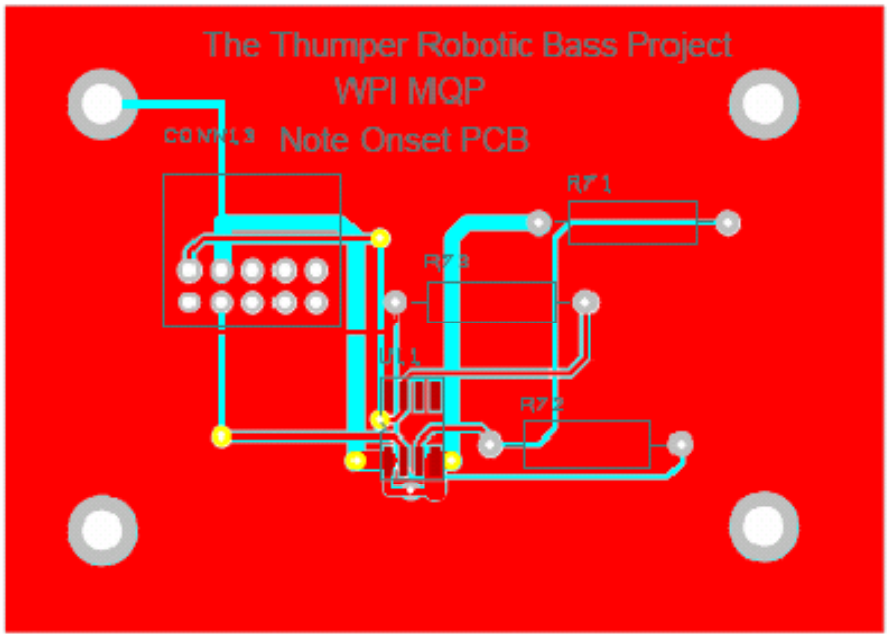

Figure 8 - Note Onset PCB

The layout for this system is very simple. The top layer is a poured copper VCC, the bottom layer is a poured copper ground. All of the stand-off holes are grounded. The comparator chip is a surface mount component; the resistors and connector are through-hole components. The free sample we received of the comparator was a surface mount component, so we decided that it would make the most sense to use the parts we already had, instead of driving up the budget further. We chose to use through-hole components because this board is rather small, and as a result we were not very concerned with making it as small as possible using surface mount components. The cost of through-hole components, especially for resistors is much less than that of the equivalent surface mount component. In addition to the fact that we could receive the components from the ECE shop, we chose to use through-hole components for this board.

Chord Detection

We were planning on developing an algorithm to properly identify notes and chords played by the guitar. We had a few ideas to implement this. The first idea that was suggested was designing and building band pass filters with a bandwidth of a few hertz. The filters would separate each note played by the guitar, and trigger logic high for the specific filter that had a note in its bandwidth, while all of the others would output a logic low. The controls would receive the individual trigger signals from filters, decide which trigger came from the filter with the lowest frequency, and send a signal to the proper stepper motor and solenoid. The lowest filter is all that is required because we are only interested in playing the root note of the chord, because that is what all a simple baseline is. If this theory works, we would implement more filters to cover more input octaves, but remain in the same octave on the bass. It would be possible to cover more octaves on the bass as well, but this would highly increase the price as more solenoids and a larger mounting plate would be required.

The worst case scenario was with the open E string, E 82, and the first fret of the E string, F 87; this filter would have a bandwidth of less than 5 Hz. As the frequency increases, the bandwidth does increase but not by much, as the highest note we decided we would identify would be the forth fret on the G string, or a B 246; the bandwidth of this filter would have to be under 15 Hz. The narrow bandwidth necessary for this filter would require the use of elliptical filters as they have the fastest roll off rate. The less than flat frequency response in the band pass of an elliptical filter would not have much of an effect, because we are just looking for logical high, not a perfect frequency response of the signal.

Upon simulation of filters with such high tolerances, we realized that this would be quite hard to implement as average order for the filters was over thirty. Building such high order filters is close to impossible in the analog domain, and would be highly subject to error from components and the environment. Therefore the filters would have to be implemented in the digital domain. Digital filters have been designed in the MATLAB filter design toolbox, and tested using SIMULINK. The digital filter approach would work, as long as there is a MicroBlaze or PicoBlaze in the FPGA.

The other major idea we thought of for chord detection is using a Fast Fourier Transform or FFT on the input signal on small chunks of time. The FFT would show all of the notes and harmonics played in the small time chunk. Small time samples would be required to reduce the computational speed required for the FFT. After testing the size of the sampling window, we decided that one millisecond is sufficient to provide adequate FFT results, if the sampling frequency is the audio standard of 44.1 KHz. This time window is sufficient because of the acoustic properties of a guitar.

Guitar signals can be split into two separate parts; the steady state response and the harmonics of the fundamental frequency. The steady state response is what is first heard with the pluck of a string. This decays within the first few milliseconds. The steady state response contains the patterns that the harmonics will follow, which allows us to use small time chunks to get an adequate FFT. The majority of what is heard is the harmonics. These are the resonances of the fundamental frequency, which when they are all summed, produce the complete signal. The harmonics are simply an integer multiplied to the fundamental frequency. The relationship between all of the frequencies is what constructs tonality.

The acoustics of the guitar play a large role in the accuracy of the FFT, as they are dependent on many aspects. These include the material the neck, body, fret board and strings are made out of. The general rule is that the as the density of a material increases, the tone gets deeper and resonates for a longer period of time. This is because there is more mass resonating, which means that there is more energy in the system, and so it will take more time for the energy level to return to zero. Another big aspect that controls a stringed instrument's acoustics is the tension of a string. A looser string will vibrate at a lower frequency, and therefore produce a note of a lower pitch. This idea is the reason why it is possible to have the exact same string tuned to different fundamental notes. Age of the string is an aspect that many guitarists know about, but few understand why an older string produces a dead sound when compared to a new one. When a wound string is played, it gets stretched and after many uses the windings will no longer align the same way they did when the string was first wound. This causes the string to no longer move in a perfect sinusoidal wave form, and eventually the string vibrates in a

distorted sinusoidal wave, which causes many more frequencies to manifest themselves in this distortion. The general wave form will still be sinusoidal, but it will no longer be a very precise one.

After the FFT was calculated, the FPGA would filter out everything but the lowest frequency peak. The lowest frequency peak is all we need because we are only interested in playing the root note of the chord. The FPGA would then compare the frequency of the lowest note to a table of the frequencies of known notes. Once the lowest note has been identified, the FPGA would send signals to the appropriate solenoid and stepper motor. This approach would require a decent amount of computation, and a Micro- or Pico- Blaze in the FPGA, but would accurately identify chords and notes played.

Circuit

We have not been able to implement this system because of time constraints. This system was given the least priority for a few reasons. The major reason was that there are commercially available products that convert an electric guitar signal into MIDI signals. Having a MIDI input signal, we can use the MIDI interpreter in the FPGA to properly identify the input and control the output. Knowing that we could buy a solution to this problem, we chose to dedicate our time to systems that we would have to design from the ground up.

Solenoid Driver

The solenoid driver was chosen because it was inexpensive, fit our specifications, and was easily available. By consulting the application circuits given in the manufacturer's documentation, we began investigating the best way to run our solenoids. After some initial tests with a prototype built up on a protoboard, we were able to learn some of the intricacies of the driver. It was determined that the general application circuit given by ST Micro would fulfill all of the needs of the project. Because of the internal configuration of the L294 solenoid driver, it must be started with the input voltage low and the enable must start high. If these conditions are not met, the chip will charge internal capacitances, and will not function as intended. To fix this, we attached a switch and would flip this switch after the power had been turned on. Once the chip had been started in this way, it would easily activate the solenoids with enough force to fret the string underneath it. After the circuit was finalized and the prototype thoroughly tested, we began researching means through which we could create circuit boards.

Solenoid Driver Circuit Layout

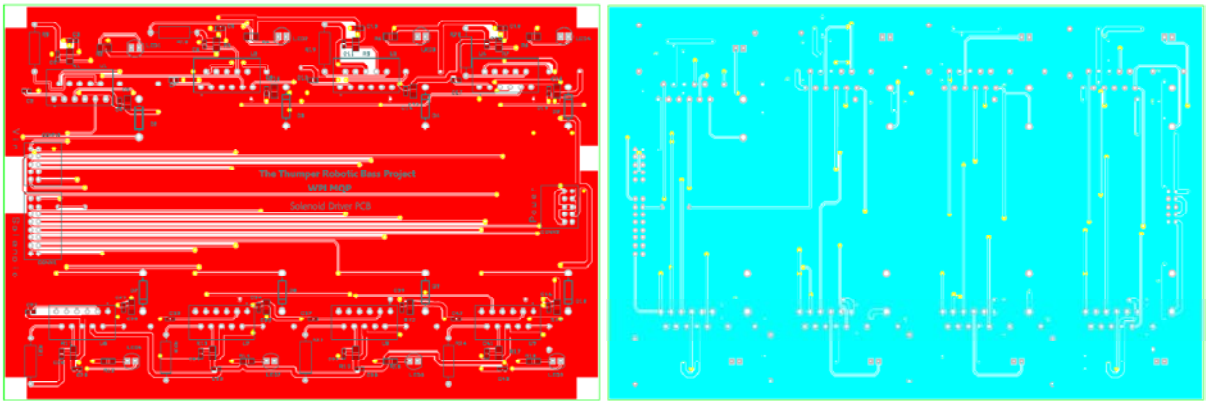

Figure 9 - Solenoid Driver PCB

As we had been able to make this particular driver work, it seemed appropriate to use it on the project. After weighing the options, we decided to use the PCB Artist software from Advanced Circuits and create PCB layouts with eight drivers per board due to the space restrictions defined by Advanced Circuits. Figure 9 - Solenoid Driver PCB shows the finished Solenoid Driver circuit board design from the layout tool. Red indicates the top layer of copper and blue indicates the bottom layer of copper. The board is a two-layer board, and is 5 inches by 8 inches in size. It was designed with a top copper fill that is connected to the highest voltage rail, 24 V. The bottom is also a copper fill and is connected to ground. This causes the resistances to be lower on the higher voltage connections, helps dissipate heat from larger components, and makes routing some of the connections easier. On each board, eight L294 drivers are situated with their supporting circuitry, and all input and outputs are routed to connectors on the edges of the board. The layout was done with an attempt to allow Adam to design and mount heat sinks to the back of the drivers. To reduce the amount of space each circuit would consume on the board, surface mount components were used wherever possible. Several components were required to carry relatively large amounts of current and were therefore through-hole components, such as the power diodes and current sense resistors.

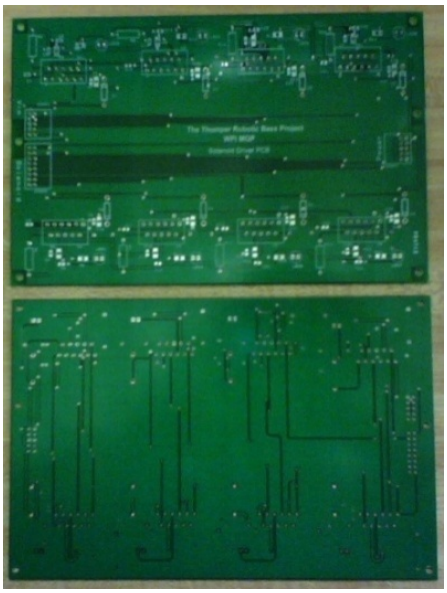

Figure 10 - Solenoid Driver Circuit Boards

After the boards were completed and ordered, we ordered the parts to populate them. When the parts arrived, we had some issues. One of the surface mount components was a size smaller than the footprint on the board, the part being a 0402 package and the footprint being 0603. The current sense resistor also did not fit in the holes provided for them. Either we made an error in checking sizes before ordering parts, or we mistakenly ordered a different suffix than was intended by the layout tool. Because of time constraints, we were unable to order replacement parts that were appropriately sized. With some effort and interesting soldering methods, the boards were populated without any significant setbacks. Because the component that was too small was only slightly so, we were able to solder it into place with a little extra solder at a reflow station. To connect the current sense resistors, we soldered pins into the holes meant for the leads, bent the leads to line up, and soldered the leads to the pins. While this does not leave a good looking board, it is adequate from an electrical stand point, and can still be made to work for our proof of concept. This type of thing is not uncommon in an engineering setting, especially when considering prototypes and proof of concept designs. Many boards undergo “cuts and jumpers,” cutting of the traces and soldering wires between spots on the board to fix errors, before they are considered complete and ready to be produced. Our main limiting factors on PCB development were monetary and time-related.

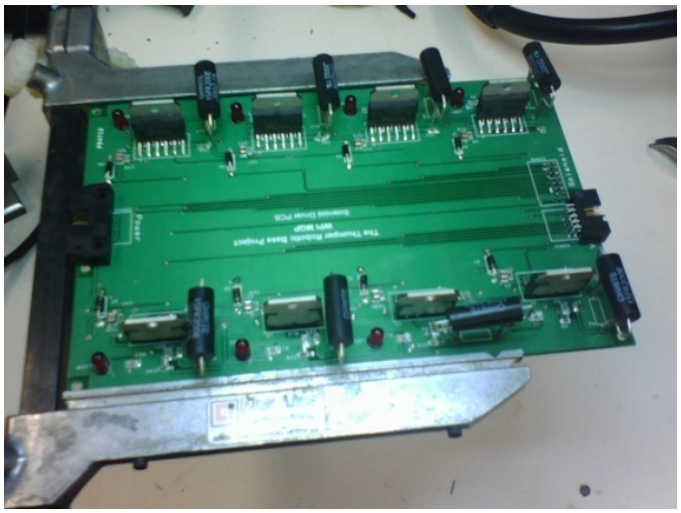

Figure 11 - Solenoid PCB Being Soldered

Stepper Motor Driver Circuit, Layout and Fabrication

When comparing appropriate DC and stepper motors, we found that the DC motors were slightly cheaper than the stepper motors. However if we chose to use DC motors, a much more intensive design would be required to properly control the motors, which would greatly increase the time spent designing, and lessened the time spent building, testing and documenting. As we steadily approached our deadline, we chose to use the stepper motor and driver because it would take much less time to design a control scheme, at the cost of an increased budget.

The stepper motors we chose to use are the 4118S-02RO made by Lin Engineering. These were chosen because they met the torque required to move the linkages at the desired speed, in addition to having the same required current and voltage as the solenoids. These stepper motors also have a footprint of a little under an inch and a half, allowing enough space to put the linkages in the proper places and have all of the motors rotate in the clockwise direction.

This circuit was chosen based on the manufacturer's documentation. We were unable to find simulation models for any of the stepper motor drivers we looked at, so we opted to take advantage of the application circuits given by the manufacturer. The L6228 was chosen because it had the best documentation and description of how to operate out of all of the drivers that would have supplied the appropriate current and voltages. One of the application circuits would provide exactly what we needed, and was documented well enough to give us confidence in its ability to drive our motors. This chip will be operated in the wave drive mode because it gives us the necessary current output with the lowest power dissipated, as it only energizes one coil at a time.

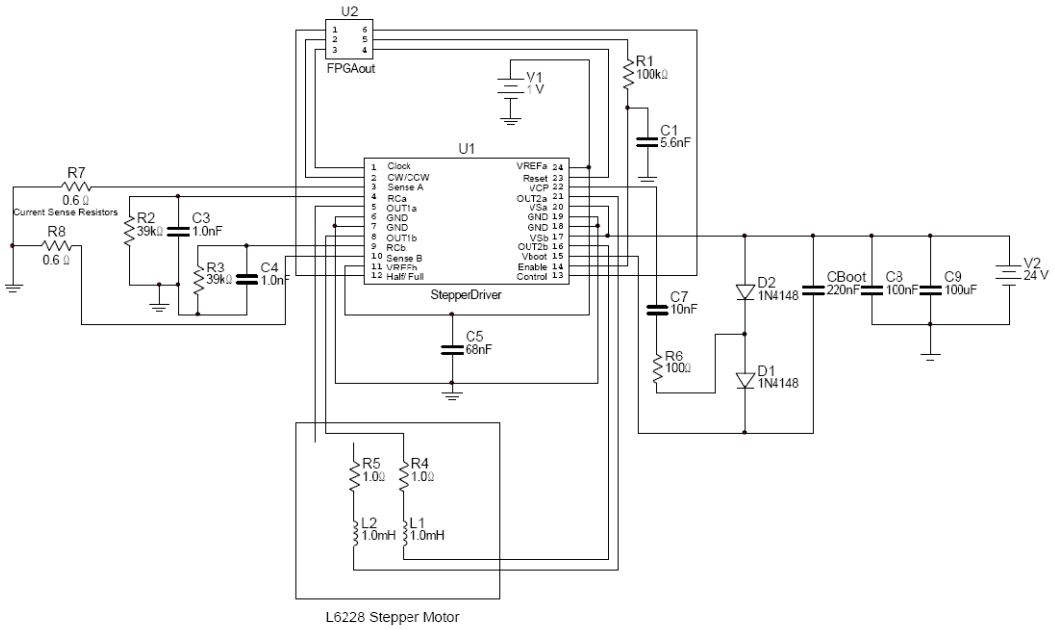

Figure 12 - Stepper Circuit

The PCB has been designed using the software provided by Advanced Circuits. We used their software because they would be the company making our PCBs, and one of their requirements is to use their software. The top layer is a VCC poured copper layer. The bottom is a poured copper ground. The majority of the components used for this board are surface mount. Surface mount components were chosen because this board needs to have four separate circuits on it, and it needs to be fairly compact in order to fit with the other boards in the circuit board housing. The only components that were not surface mount are the diodes and the current sense resistors. The resistors are through hole because surface mount current sense resistors do not exist. The diodes are through hole because of the major difference in price between the through hole and the surface mount diodes of a few dollars.

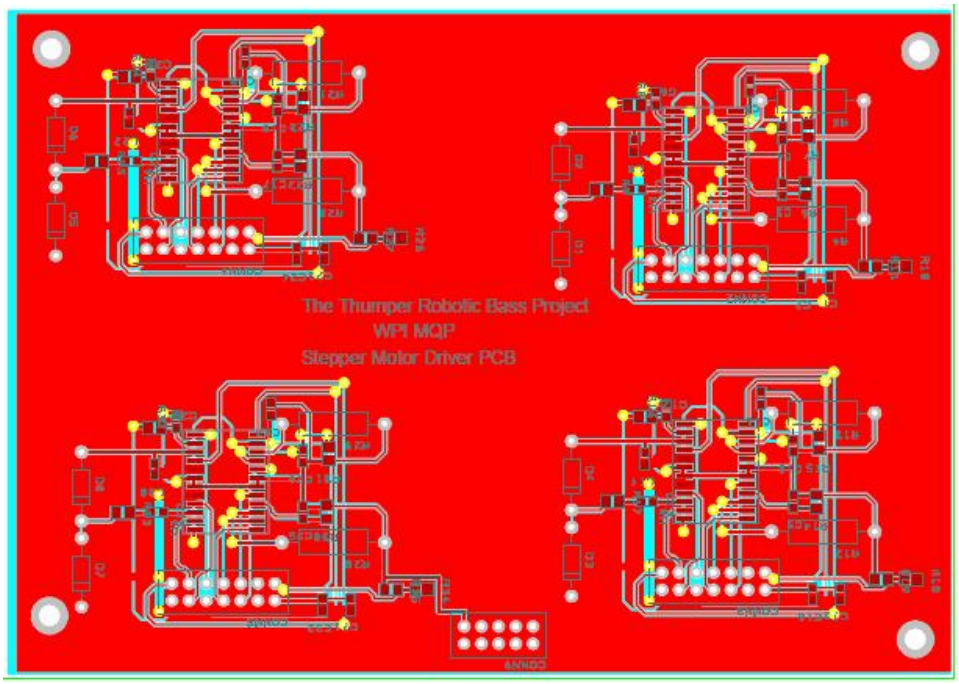

Figure 13 - Stepper Motor PCB

When the parts arrived, we noticed that a few of the components were not the right size. We speculate that this error is caused by the lack of suffixes denoting package size in the software. One of the resistors was larger than the pads and spacing designated for them. To solve this problem we soldered leads onto the resistors, and soldered the leads onto the pads. This problem arose far too late in the building process to allow for new parts to arrive, so we had to make do with the best of what we had. One of the capacitors in the circuit was larger than shown in the footprint in the software and as result this capacitor and the adjacent capacitor do not fit properly on the board. To solve this problem, we soldered standoffs onto the larger capacitor so that is above the smaller one, and separated by an insulator. We realize that these solutions are not the best way to handle these problems, but unfortunately better solutions would require a few weeks to order properly sized replacement parts. We do not have the luxury of a few weeks to wait while parts arrive, so we decided that although our solutions would have to suffice.

Control

To facilitate the calculations and control needed by the Robotic Bass, a digital control unit had to be selected. We had a discussion about whether it was better to use a processor or a field-programmable gate array (FPGA). The decision was narrowed to a Texas Instruments MSP430 microprocessor or a Xilinx Spartan-3E FPGA[5]. Both could easily do the job asked of them, and each had its positive and negative attributes.

The MSP430 is an ultra-low power 16-bit processor produced by TI. It is used by the WPI ECE department as a digital platform in some of the digital design courses. This processor was a good

[5] Datasheet – Spartan-3E FPGA Family: http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

candidate for the control unit of this project mostly because it included a universal asynchronous receiver/transmitter (UART) built onto the board it came on. A UART is used to take in serial data and convert it into parallel data that can be used more efficiently by the processor. The drawback of this processor was the set up and use of the UART, which was more complex than necessary for the project. Involving a complicated series of programming steps to activate and configure the UART, it would present a problem in terms of time to development. In addition, we only needed a receive UART (as discussed later), and having to configure a receive section and a transmit section, one of which we would never use, seemed to be a waste of time.

The second possible candidate for the control unit was a Xilinx Spartan-3E FPGA. An FPGA is an integrated circuit that can be used in conjunction with particular pieces of software to define digital designs inside the chip. By using hardware description languages (HDL), an engineer can define a digital design, and then synthesize it using the tools provided by the FPGA manufacturer to configure the chip. This approach was more complicated in some ways, because the chip does not include a UART, therefore we would have to build one ourselves. While we would be able to tailor the UART to our needs, it would increase the time to development of the controls. We found a Spartan-3E Starter Kit[6], which was a pre-built board that had incredible capabilities, but could also be configured to only do what we needed it to do. We were even able to find a UART core[7] on an open source HDL website called Opencores.org, which would negate some of the time to development that would be added by having to create one. VHDL was also an area of particular interest for Matt. This seemed to be the best choice for the control unit.

[6] Spartan-3E Starter Kit Board User Guide:
http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf
[7] Lupas, Ovidiu. "Serial UART: Overview." Opencores.Org. 22 Jan. 2004. 3 Feb. 2008
<http://opencores.com/projects.cgi/web/uart/overview>.

MIDI Interface

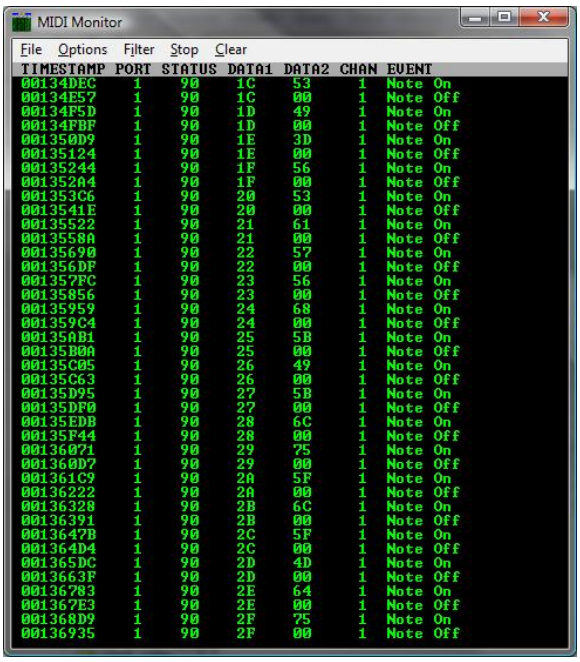

Figure 14 - MIDI Messages

To interface with a MIDI device, we needed to not only understand how MIDI itself operated, but also how to bring it into our FPGA. To do this, we needed a UART as stated earlier, which would take the serial input from the MIDI device and parallelize the data into a useful form. UARTs are used in many applications, such as RS-232 serial ports in a computer. A UART takes in asynchronous data, data that is not clocked, and converts it to parallel, synchronous data. They are usually tailored to whatever protocol they are intended to work with. In our case, we were dealing with the MIDI protocol. To understand how MIDI works, we must have some understanding of serial data transmission.

There are many types of serial protocols, the simplest being 8b/10b, which is eight bits proceeded by a start bit and followed by a stop bit. The 8b/10b stands for every eight bits of data, ten bits are transmitted. The start bit is a particular state that the UART looks for in order to determine when a new byte has started. The stop bit would be opposite. For example, if a protocol required a 0 as a start bit, the stop bit would be a 1. This allows the asynchronous input signal to be chunked properly into bytes. By looking for a particular bit before the data and another after, the receiver has a better chance of not losing its place and grabbing the wrong bits as data. Sometimes, the protocol can be more complicated and can include a header and footer before and after the data, respectively. The header and footer can be several bytes long and can include commands, depending on what is defined by the protocol. The whole transmitted unit (header, data, footer) is often called a packet. The receiver gets the packet, decodes the header, accepts the data, and then decodes the footer. Now the receiver can pass the data on to be worked with.

The MIDI protocol is well defined for the most part. Other than some definitions of commands that rarely get used, it describes the intended workings f the protocol in detail. The transmitter sends its

data in 8b/10b format, using a 0 as a start bit. The speed of the transmission is also explicitly defined in the specification for the protocol. The start bit/stop bit transmission is very common, and we were able to find an open source UART core from Opencores.org, as indicated in a previous section. The core was already setup to deal with 8b/10b, but was intended for use with RS-232. As such, it was intended to work at RS-232 speeds, divided down from a 40 MHz board clock instead of the 50 MHz we had.

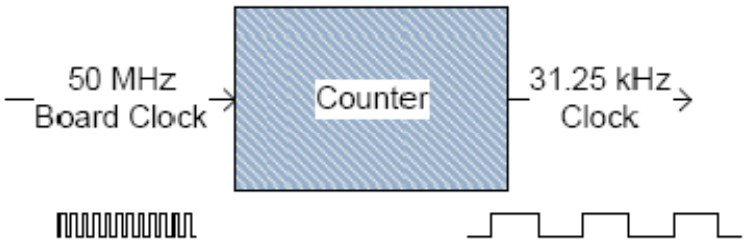

Figure 15 - Dividing the Clock in the UART

By changing the timing of the core to match the 31.25 kbits/sec that is defined in the MIDI specification, we had something that ought to receive the data from a MIDI controller and parallelize the data into bytes. Figure 15 - Dividing the Clock in the UART shows the type of process that sets the receive and transmit rates to meet the specification. These processes count pulses from the 50 MHz board clock, and once it has counted enough of them, outputs a pulse, which looks like another clock, but at a slower speed. In a UART, the receiver runs much faster than the transmitter. This is because the transmitter must send its data out at the rated speed of the protocol, but the receiver has to check the incoming data much faster, to make sure it catches all transitions in the digital signal. If the receiver ran at the same speed as the transmitter and it started checking the data at a time where the incoming signal was in the middle of a transition, the receiver might see a voltage metavalue, something between low and high, which would be undefined. To avoid this, the receiver runs at a much higher speed, to make sure it gets a large enough number of samples to be sure it knows what the incoming waveform looks like.

The protocol states that the first byte has a 1 as its most significant bit, and is a command byte. Command bytes are the only bytes that have this format. The value of this byte tells the receiver how to interpret the data that follows. The following bytes are data, and the byte length of this data can vary, depending on the command received. There are several commands that involve setting up synthesized instruments and configuring the devices. And then there are commands for playing the music. For example, the “Note On” command instructs the listening MIDI devices that a note is to be played, and the data following is two bytes, the first gives the note, and the second gives the velocity. The notes are numbered using the 7 bits of the first data byte (as stated above, only command bytes have a 1 as their most significant bit) and as such, there are 127 possible notes. The scale is defined with middle C as note 60. The velocity bit is how fast the note is played. If someone is playing a piano, and pressed the key slowly, the note will sound softer than if the pianist hits the key rapidly. “Note Off” is a command that has been defined in the MIDI specification, but it is rarely used, in favor of a “Note On” with a velocity of zero.

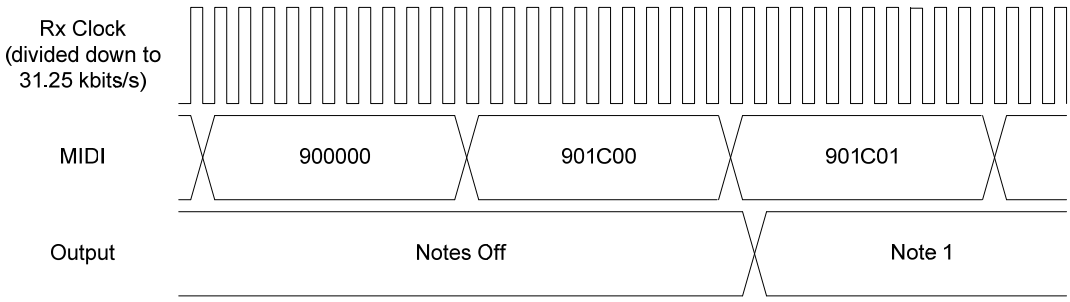

Figure 16 - MIDI Interpreter Timing Diagram

The scope of the Thumper is to play the notes defined as 28 to 47 in the MIDI specification. As such, we needed two commands, a “Note On” and a “Note Off.” The Roland PC-200E mk II we used to test the system uses the “Note On, velocity 0” shorthand instead of a “Note Off”, and as such, our controls interpret it by shutting all solenoids and motors off. The UART takes in the serial MIDI data and an intermediate core places three bytes together to create a 24 bit vector that includes the initial command, the note, and the velocity. We didn’t care about changing the output based on the velocity, and as such, any velocity greater than 0 is treated as a note being played. We also designed the system to only allow one note to be played at a time, to avoid power problems, which will be discussed later. After all of this was built and loaded into the FPGA, we had a system that could receive MIDI information and interpret it in a meaningful way. Figure 16 - MIDI Interpreter shows one of the successful tests run on the MIDI Interpreter core, switching the output note between the lowest note we are interested in, and the highest. Several other tests were done to ensure that that the unit would act as required to properly decode the MIDI input.

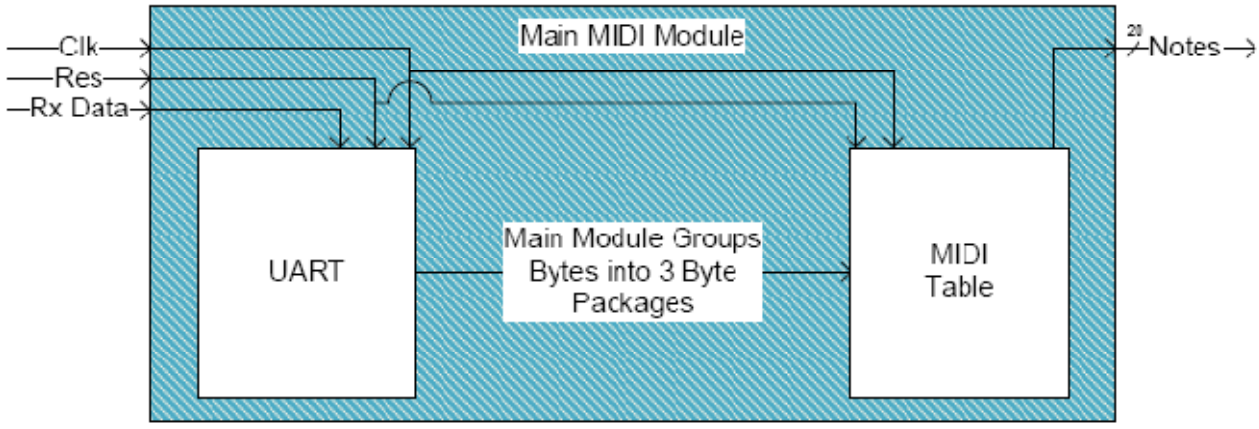

Figure 17 – MIDI Module Block Diagram

The MIDI Interpreter is made up of two major components. The UART, as discussed, bring in the serial MIDI data. These bytes are packaged by the main module into 3 byte units, and then passed to the actual interpreter, the MIDI table. VHDL can communicate with other VHDL files if they are included as components, as in Figure 17 – . In this case, we included the UART core found at Opencores.org and the

MIDI table. This figure shows the higher level workings of the MIDI Module. Figure 18 – shows the logic in the MIDI Table module in terms of a flow chart. First, it would look for the “Note On” command (90 in hexadecimal notation), which would be the first byte in the package the MIDI table received from the main module. It uses this as a reset for the counter that it uses to keep track of which byte it’s currently reading. It then checks the second byte to see which note is being referenced. If it is one of the twenty notes in the scope of our project, in then check the velocity. If the velocity is greater than zero, it plays the note. This is illustrated in Figure 19 - MIDI Packaging State Machine.

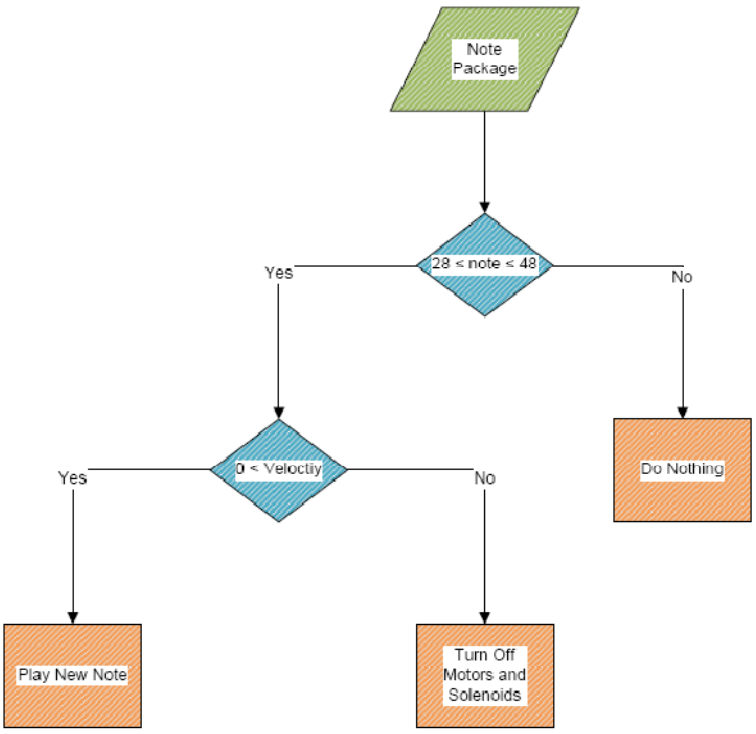

Figure 18 – MIDI Table Logic Flow Chart

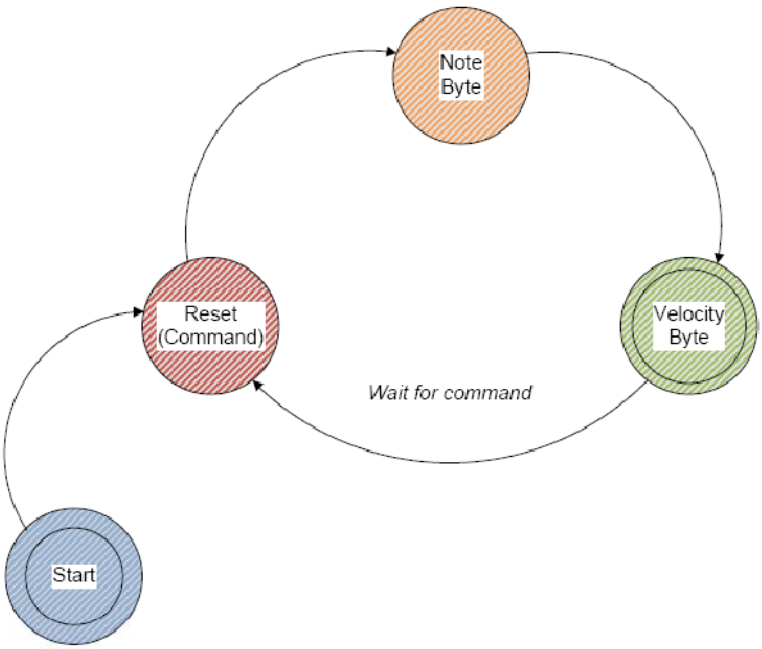

Figure 19 - MIDI Packaging State Machine

Solenoid Control

The solenoid control was done in a simple way. Because the guitar input was not implemented, there was no need to multiplex the solenoid drive signals. Using an L294 solenoid driver from ST Microelectronics, we can output control signals to the fretting heads. The L294 has some eccentricities we were unaware of until we had them in our hands, such as the fact that the enable had to start high, and the input had to start low. To fix this problem, we added a switch to the connector on the solenoid driver board, with the possibility of later connecting these pins to the FPGA and having the switching happen there.

The original plan for the solenoid control was that it would switch between direct MIDI control and guitar input. We had discussions about converting the guitar input into MIDI signals and piping it through the MIDI Interpreter, but this plan was discarded as the hardware components that have the ability (as produced by Roland and other companies) are priced on the order of several hundred dollars, and to design a similar interface would take too much time. The solenoid control would have to be able to be set from guitar input or the MIDI Interpreter. Unfortunately, time constraints eliminated the implementation of the guitar input, but that also removed the need to multiplex the solenoid control. They would be receiving the signals directly from the MIDI Interpreter.

The MIDI Interpreter outputs twenty signals, one for each note to be played. This was done to allow a signal for each solenoid. Because we are playing four open notes, four of the twenty outputs are not used, but they are still needed from triggering the stepper motors. By connecting the signals to be used with solenoids to pins of the FPGA connected to the 100 pin I/O connector, we can get the proper control signals to the solenoid driver boards. Without the multiplexing mentioned above, this is a direct connection. We now had control of our bank of fretting solenoids.

Motor Control

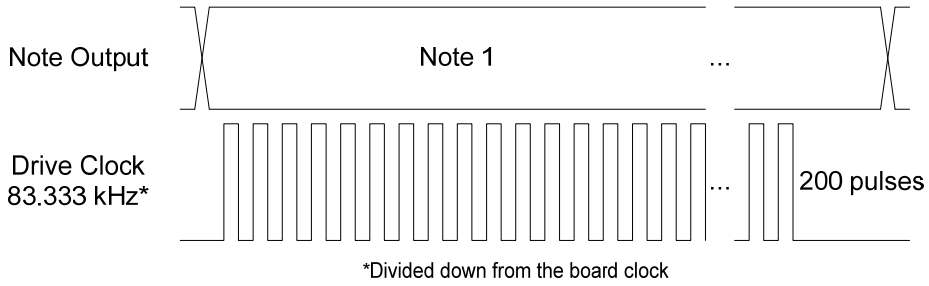

Figure 20 – Stepper Motor Timing Diagram

Using the same output from the MIDI Interpreter, we defined controls for the stepper motors. We broke up the notes into four groups based on the string on which they were to be played. Four pins of the I/O connector were connected to L6228 stepper motor drivers, also from ST Microelectronics. These drivers were chosen because they were capable of handling the motor currents required by the project, and of the components we were looking at, had the simplest interface. The motors were drawing around 1.3 A, and after significant searching, we managed to find the L6228. Most drivers could handle just more than what we needed as a peak, and not continuous. The L6228 could handle the currents, and the interface was much easier than some of the other contenders from a control standpoint. Most of the drivers involved designing a state machine to send the proper signals to the drivers, but the L6228 had its own state machine built in.

The L6228's state machine is designed to be operated in one of four modes: half-step, normal, wave, and micro stepping. Each mode has its advantages and disadvantages, some allowing the motor to exert more force at the cost of additional power consumption. Based on the power curves provided by the data sheet and the desire to keep the operating currents as low as possible, we chose wave drive as the best method. This involves stepping through the internal state machine of the driver in a particular fashion. We hold the HALF/FULL pin high while we reset the chip, apply a pulse to the input, and then drop the HALF/FULL low. This performs a half-step of the motor and puts the state machine into an even numbered state. Because the HALF/FULL pin is now low, the state machine will step a full step to the next even numbered state, and continue this as long as we pulse it. This does have the disadvantage of causing an extra half-step when the system turns on. A reverse half-step would negate this extra half-step, but in the interests of time constraints, has not been included as of yet. After the initial half-step when the system is reset, every time a note is to be played, the system will pulse 200 times, which is a full revolution of the 1.8° stepper motor. The motor should rotate a full 360° every time a note on its string is to be played. As long as the timing between the solenoids and stepper motors is within a few milliseconds, a note will be played.

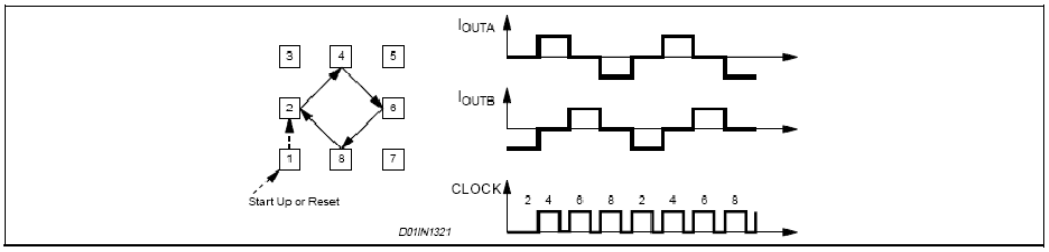

Figure 21 - Diagram of Wave Drive and the Driver's Internal State Machine

The stepper motor control involved logic like that in Figure 22 - Stepper Motor Logic. The notes to be played were checked against which string they would be played on using if-then statements. If the motor had not made two hundred steps, it would step forward. If it had, it would reset its counter to zero and set a flag to 1 so it would not step again until another note was to be played. This flag would only be reset when the MIDI table received another note to play. The clock had to be divided down from the board clock in this module, because the driver could not handle switching speeds greater than 100 kHz.

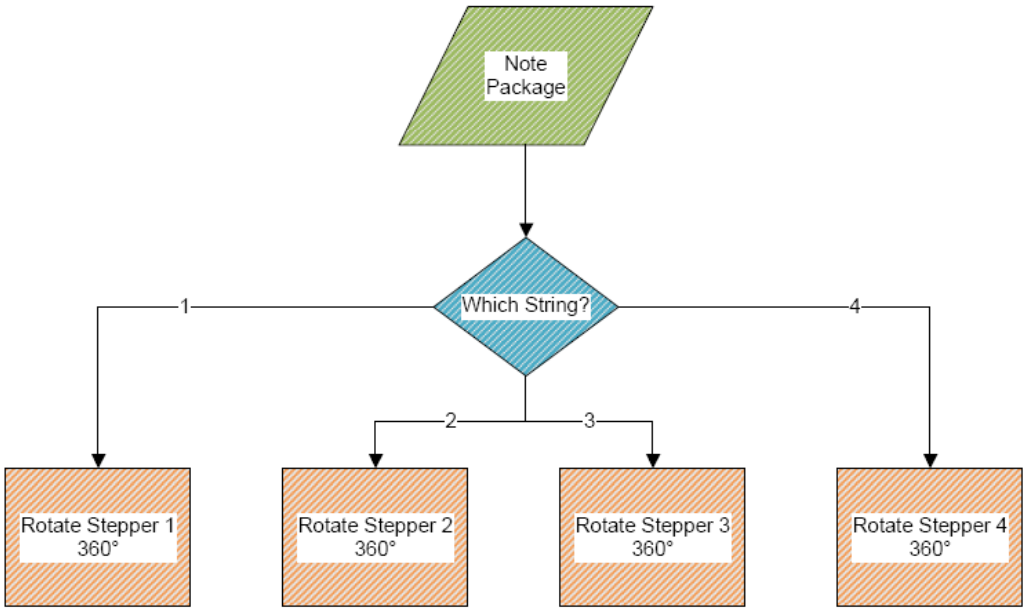

Figure 22 - Stepper Motor Logic Flow Chart

Power

Originally we intended to design our own power supply by means of a transformer to step down the voltage, and step up the current to appropriate values. This would have been fed into a full wave rectifier to change the AC voltage to DC voltage with a small voltage ripple. We also discussed using a DC to DC converter, to supply our power, as finding a transformer that would have the necessary input voltage and output voltage and current proved to be an arduous task with the only viable results costing around 100 dollars. However as we were searching for an appropriate transformer we discovered a power supply that would fit our needs for less than the cost of the transformer. After much discussion about power supplies, we had the epiphany to use a slightly modified computer power supply. The

main modification we would do is put a load on it internally, and add leads to the positive and negative twelve volt rails. Using this option would decrease our budget as each of us has at least 3 old power supplies collecting dust in our apartments. This power supply provides the power to all of the system aside from the FPGA, as the FPGA has an included power supply. The solenoids require 24 volts and 1.6 amps to be driven properly. The stepper motors require 24 volts and 1.3 amps per phase. We decided that having both these systems require similar power would allow for design of the power supply to be much simpler.

Mechanical Design

Fretting

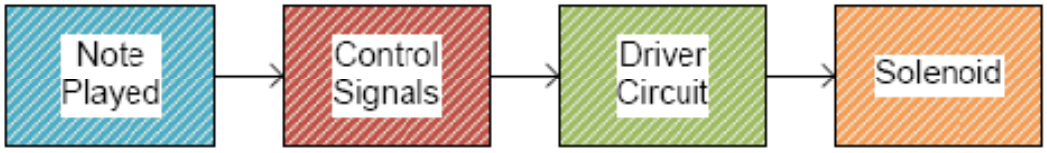

Figure 23 - Fretting Block Diagram

To fret the notes desired by the user three different concepts were created from the simple idea of applying pressure to a string. The first design was simply that a bank of solenoids one solenoid per fret per string each with enough force to fret the note it was assigned to. The second design used the concepts of solenoids from first design and combined them with a sliding mechanism. The third design is based on the motion of a human bass player's left hand with "fingers" that curl around the neck to apply pressure and a sliding mechanism to move the hand up and down the neck.

Design Concepts

Design A

The bank of solenoids approach utilizes a number of solenoids mounted above the neck of the instrument in fixed locations. The number of solenoids required would depend on the number of frets desired to be utilized, for this prototype we have limited our designs to the first four frets of each string, thus requiring 16 linear solenoids to be mounted. This design would require a fixed neck cradle to position the neck in the correct location and not allow it to move, as well as position all the solenoids in the correct position above the frets they are assigned to play.

Advantages	Disadvantages
Straight forward design	lots of force on neck
Fast transitions between notes	large power consumption
Inexpensive	
Easy to control	

Table 1: Solenoid Bank Comparison Table

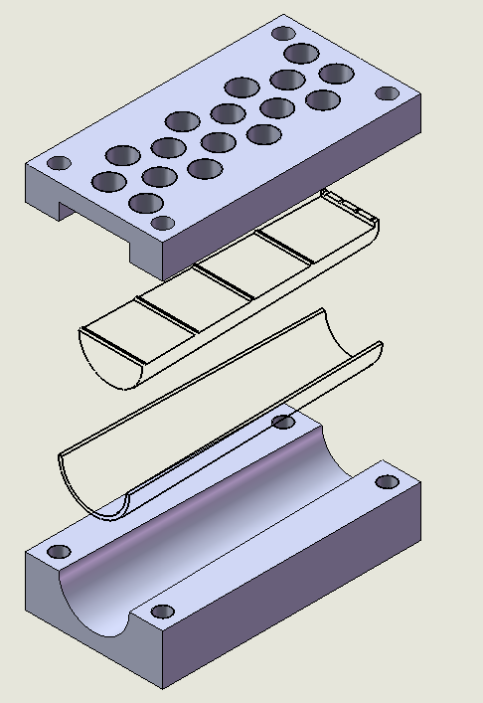

Figure 24: Isometric exploded view of final solenoid mounting system.

Design B

This design is taken in part from the GuitarBOT discussed earlier with a one slider per string designed fret all the notes on that string. This design differs from the GuitarBOT in that the slider is not always in contact with the string but engages and disengages the strings by the use of linear solenoids. This system would require a way to mount the slides to the neck of the guitar without interfering with the strings and each other. This setup not only utilizes four sets of solenoids to engage the strings, but also a motor with encoded movement to control and position the slider above the desired fret.

Advantages	Disadvantages
similar to human motion	complex design
versatility of motion (slides)	slow transitions between notes
smaller power consumption	less stable

Table 2: Sliding Fret Head Comparison Table

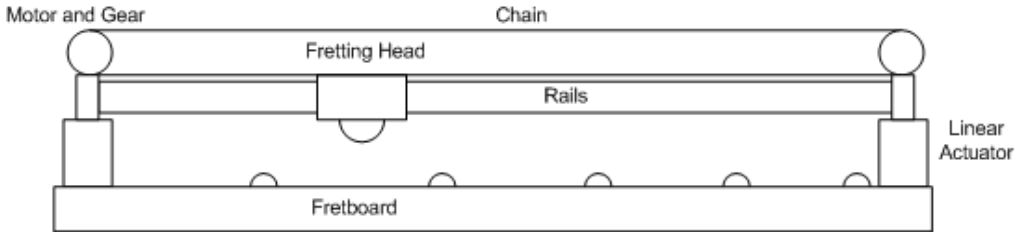

Figure 25: Sliding Fret Head

Design C

The third and final design concept takes its design from the human finger complete with articulating joints. When a person plays a bass guitar their left hand curls around the neck of the instrument and the pads of the fingers fret the notes. This design would create a mechanical human finger analog mounted on a sliding mechanism. The sliding mechanism would allow the finger to reach any fret and the motion of the finger would allow it to reach all the strings. This design would require a similar motor setup to design B as well as whatever actuation would be needed for the finger. There are commercially available robotic finger/hands that use various control setups from pulling wires to air muscles.

Advantages	Disadvantages
Most similar to human motion	complex design
	very slow transitions between notes
	expensive
	high chance of failure of components
	complex controls

Table 3: Human Hand Analog Comparison Table

Figure 26: Commercially Available Human Finger Analog[8]

Final Design

Each design has advantages and disadvantages many of which are different from a production standpoint as compared to the prototype being developed by this project. From a mechanical standpoint the bank of solenoids is the simplest design to design and implement it has the fewest

[8] Worsdall, Mark, Richard Greenhill, Rich Walker, Hugo Elias, Matt Godden, and Jake Goldsmith. "The Finger Unit Pictures." Shadow Robot Company. Shadow Robot Company. 4 Apr. 2008 <http://shadowrobot.com/finger/pictures.shtml>.

moving parts and construction is fairly straightforward, this design does seem to require the most material thus cost and weight would be a factor. Electrically the first design is also the simplest with only having to send a high or low signal to a specific solenoid to accomplish fretting a note, though this design has the potential to draw a lot of current if multiple solenoids are traveling at the same time. Design two adds a level of complexity the first design did not have by utilizing a sliding mechanism this design would require sliding bearings and a fretting head to apply pressure to the strings. This design would however take up less space for the amount of frets it could reach allowing for more notes to be played. Electrically the sliding mechanism would need to be controlled by some form of an encoded motor the control circuitry for which is more complex than that of solenoids alone. The current draw from this design would be comparable to that of the first design even with a reduction in the number of solenoids the addition of motors makes up the difference in current draw.

Solenoids

The decision had been made to use only linear solenoids to actuate the fretting mechanism. The final decision on which specific solenoid to use came from a long list of potential candidates. To start a decision was made to use electric solenoids over their pneumatic or hydraulic counterparts. To control pneumatic or hydraulic solenoids valves need to be actuated which is often accomplished with electric solenoids, this coupled with the complexity and high cost of many hydraulic and pneumatic systems allowed for the exclusive consideration of electric solenoids. Electric solenoids can be classified into push-type or pull-type solenoids based on the direction in which their force is applied. For the implementation of the bank of solenoids a force is needed to be applied down on the string toward the fretboard of the bass. This implies that a push-type solenoid would be necessary for this application and is preferred, in the event that no push-type solenoid is available a system of levers would be designed that could direct the applied force in the appropriate direction.

The amount of force required to fret a note varies with which string as each string has a different stiffness and is under a different tension. Also the closer to the nut at the top of the neck the more force is required to successfully fret the note. A successfully fretted note is one that does not "buzz" when played, the buzz is caused when the string vibrates and bounces off the fret instead of vibrating past the fret. The worst case scenario we are dealing with in this project is the first fret on the g-string. To measure the amount of force required many methods were conceived from hanging known weights off the string until it was fretted successfully to slowing applying a force transducer with a screw mechanism. Ultimately the only test that yielded any usable data was found by applying known masses to the string and seeing how much was required to successfully fret the note. From this test it was found that a 1 kg mass successfully fretted the note, .5 kg mass caused the note to buzz, and a .25 kg mass could not bring the string to the fretboard. This tells us that a force between 9.8 N and 4.9 N is required to fret the note as derived from the definition of a Newton $N = \frac{kg * m}{s^2}$. Many of the solenoids that were considered had their force listed in different units; Table 4 has a listing of the most common conversions.

Newton	gram-force	ounce-force	pound-force
1	101.971	3.5	0.21875

9.8	999.3158	34.3	2.14375
4.9	499.6579	17.15	1.071875

Table 4 - Conversions for forces required to successfully fret a note

Along with having the ability to generate enough force the solenoid must be able to travel far enough to completely depress the string and retract far enough as to not interfere with the string when it is not engaged. This travel depends on how the action, or height of the strings off the fretboard, is set on the instrument being played, many of which have the ability to be adjusted. Our test platform has adjustable action and we have set it to as low as it will go without creating interference during normal playing. The height of the top of the strings to the fretboard is between .255 in and .365 in increasing as the frets get further from the nut. Not only do the solenoids need to travel as far as the strings need to travel they must also travel high enough to not contact the strings when played as this will mute the sound from that string. The amount of travel a solenoid has directly correlates to how much force the solenoid can apply due to the physics behind the magnetic coil acting on the pin within the barrel of the solenoid. As more of the pin is brought inside the barrel more is exposed to the magnetic field of the coil and thus the force increases. This relationship between travel and force exerted is defined by an equation which is different for each solenoid and was examined closely before a solenoid was selected. To optimize the use of the force that whichever solenoid is selected the very end of the solenoids travel should be just slightly more than needed to fret the note resulting in the maximum force being used to fret the note.

With knowledge of how much force and how much travel is required to fret a note and the desired type of solenoid a preliminary search was conducted to many manufactures of linear electric solenoids including Ledex and Guardian Electric. This initial search did not turn up many results that could generate enough force with the length of travel needed by this problem. The few solenoids that were found had barrels with diameters on the order of 1.5 in which brought another major problem to light; how large can the solenoids be and still fit on the neck of the bass. Utilizing simple geometry and the Pythagorean Theorem, as seen in Figure 27, it is possible to get a relationship between *L* the offset distance between the two rows of solenoids and *D* the diameter of the solenoids; the relationship is $D = \sqrt{a^2 + L^2}$. The distance between the strings *increases* further from the nut, the worst case scenario is the first fret with a distance between strings of .466 in. The maximum acceptable *L* has been set at 1 in back from the fret, any further and the string will buzz against the fret not matter how much force is applied. By plugging in 1 for L and using the formula the maximum allowable solenoid diameter is 1.103 in.

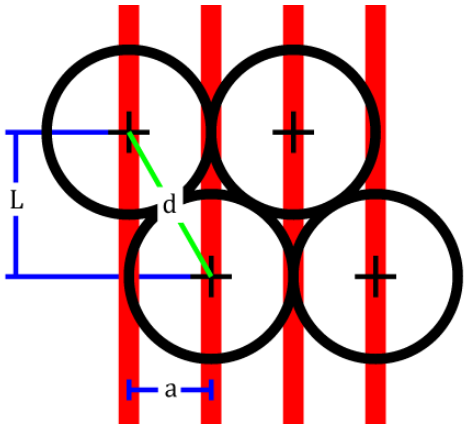

Figure 27 - Calculating Maximum Solenoid Diameter

With this in mind a set of solenoids were selected from Guardian Electric but upon request for a quote it was found that the parts had more than a 4 week lead time. This long lead time on a project with such a short deadline was unacceptable and a different supplier was needed. McMaster-Carr, a large industrial supplier and a common supplier of parts for WPI, has a limited selection of solenoids. Though limited they did in fact have a solenoid that perfectly fit the specifications needed to fret the notes. The solenoid selected exerts 48 ounces of force with a total stroke of .5 in and a diameter of 1 in, all values fit the calculations sufficiently. These solenoids come in two varieties 24v and 12v both require the same amount of power to operate therefore the 24v is desirable as it will draw less current. These solenoids do not have a built in spring return therefore springs needed to be purchased that fit within the barrel of the solenoid and not have excessive force that the solenoid would have to push against.

Solenoid Mounts

To hold the solenoids securely above the neck a mounting plate of some kind had to be designed. This mounting plate needed to work in conjunction with a cradle to hold the neck in the correct location underneath the solenoids. The major concern with this design was to keep the solenoids in position, for this to be accomplished a plate of threaded holes was designed. To locate all the holes correctly all the solenoids would need to pack in as tightly as possible. A rough design of a mounting plate was created in SolidWorks, a rectangular block with a channel cut out for the neck of the bass. Next the holes needed to be located to accomplish this task a set of construction lines were created on the tops of the block in line with the strings of the bass. To have all the holes fit the solenoids needed to be offset from these lines the number that was determined was .117 in and was meant to be split between the top two and bottom two strings. When the design was machined the offset was taken completely by the bottom strings thus placing them just off the strings, this problem was resolved by slightly enlarging the fretting head attached to each solenoid and make the material stiffer. With these construction lines in place the holes could be located, to do this and create the tightest packing arrangement all holes with centered on the construction lines associated with the string it was meant to fret. Then using pure distance relations in SolidWorks all holes we referenced at exactly 1 in apart from each of its neighbors as seen in Figure 28.

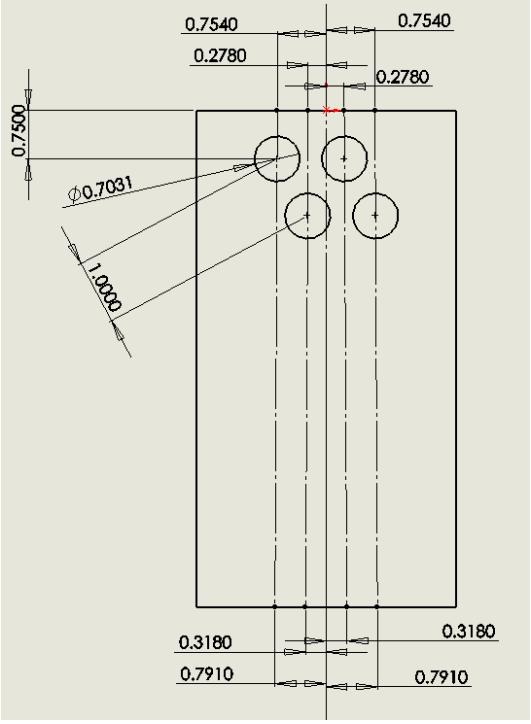

Figure 28: Screenshot from SolidWorks showing construction lines and beginnings of hole layouts.

The plate was manufactured out of aluminum as it was an accessible material known to be able to hold the small threads required by the solenoids. While many plastics would have more than adequately strong for this application the concern for their ability to retain the small UNS 3/4 – 24 threads was in question. The thickness of the plate was also a large concern as the more threads gripped by the plate the more securely the solenoids would be held in place thus the plate is as thick as the threaded portion of the solenoids. The plate was manufactured at WPI in the CNC machining shop in Washburn Labs so that the holes and threading would be as precise as possible. Even with the high tolerances capable on the HAAS VM-3 many of the holes were not tapped perfectly straight thus leading to some difficulty in threading the solenoids in. This could not have been avoided while still being able to machine the parts in house and for an actual model outsourcing the machining would be required.

Plucking

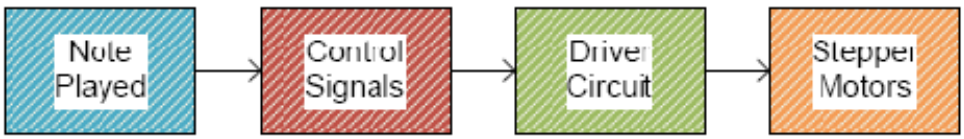

Figure 29 - Plucking Block Diagram

To pluck the appropriate strings at the appropriate times three distinctly different systems were considered. The first system was that of a four or six-bar kinematic linkage with a coupler curve designed to simulate the movement of a human bass player's fingers. The second design was taken from a historical musical instrument, the harpsichord, a keyboard instrument that uses metal picks against strings to produce sound. The third design is similar to the plucking mechanism of the LEMUR

GuitarBOT, with picks mounted on a rotating drum. In all of the following figures the XZ plane is the plane of all four strings of the instrument.

Design Concepts

Design A

A four or six-bar Grashof linkage design would accurately emulate the movement of a human finger plucking a bass string. The linkage would be controlled by an encoded motor which would rotate the crank which in turns drives the rest of the linkage. Linkage coupler curves can be designed to match many complex paths, which would allow for a great deal of freedom for accurate plucking. Linkages provide a large amount of mechanical advantage, which reduces the amount of power needed to pluck the string.

Advantages	Disadvantages
easy to control	Complex Design
most accurate representation of how humans play	Slowest

Table 5: Linkage Comparison Table

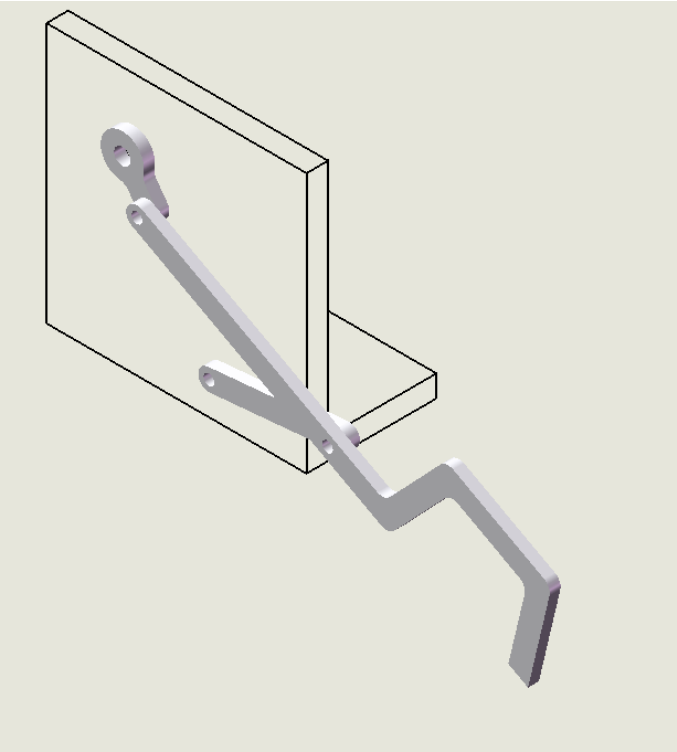

Figure 30: Isometric view of final linkage design.

Design B

A harpsichord produces sound by having a pick hit a string, which causes a sounding board to resonate. When a key is pressed, the pick mount or tongue is forces upwards, causing the pick to hit the string perpendicularly. After the string has been hit, a spring in the tongue forces the pick to change its angle, so that on after the key is released, the pick can travel past the string without hitting it a second

time. Once the tongue falls back into its resting position, a damper stops the string from vibrating any further.

Advantages	Disadvantages
Easy to control	Unnatural Sound
Time tested design	Complex Parts
	High Chance of failure of components

Table 6: Harpsichord Comparison Table

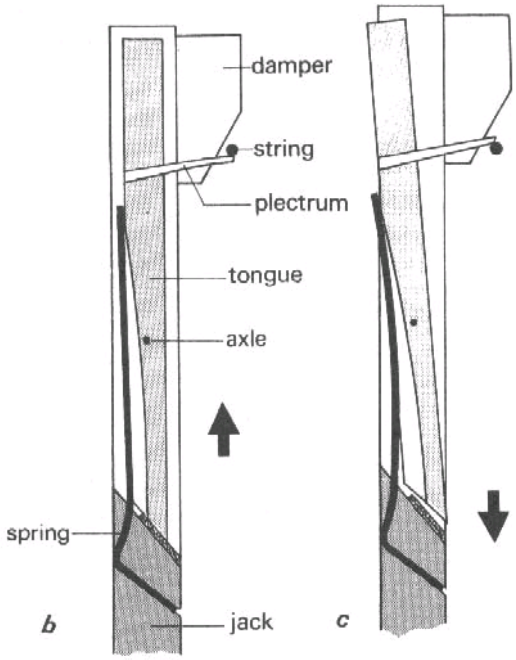

Figure 31: Harpsichord Pick Device[9]

Design C

The rotary pick mechanism has three guitar picks mounted on an equilateral triangle connected to an encoded motor. The motor would spin a third of a rotation for each note that had to be plucked. This design would be mounted above each string at a height so that edge of pick would pluck the string, and so that as the pick passed by other strings, it would not contact any of them.

Advantages	Disadvantages
Easy to control	Least human like sound
Simplest design	

Table 7: Rotary Pick Comparison Table

[9] "Simple Harpsichord Mechanism." Harpsichord.Org.Uk. 1 Oct. 2002. British Harpsichord Society. 4 Apr. 2008 <http://www.harpsichord.org.uk/mech1.htm>.

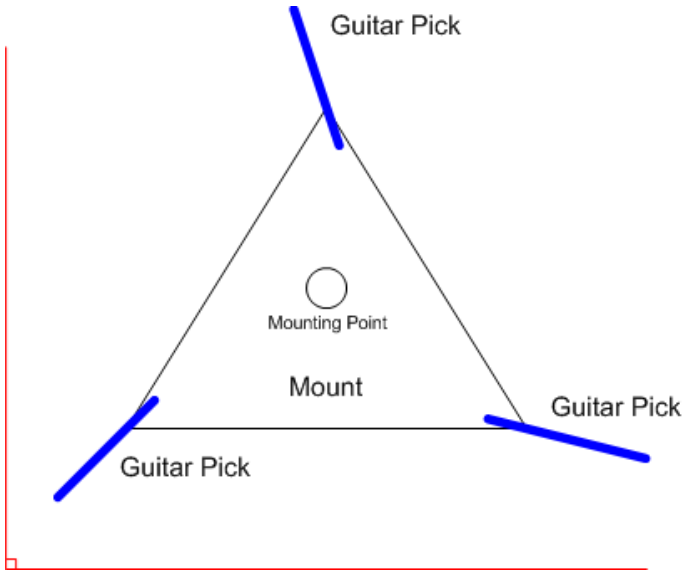

Figure 32: Rotary Pick Device

Final Design

We spent a great deal of time discussing the merits of each possible design. We concluded that one of the most important details was the quality of the sound that each design would produce. Plucking a string with a finger is viewed as the most traditional way to play a bass, and produces the least distortion of the note. When a pick is used to pluck the strings, there is more noticeable distortion to the note. This increased distortion is caused by the rapid change in position that the string undergoes. When a string is plucked with a finger, it has a much slower change in position because the finger is in contact with the string for a much larger time. Both the rotary pick and the harpsichord use a pick to vibrate the string, but the linkage uses motion similar to that of a human bassist.

The linkage idea was considered because it would be the most accurate representation of natural bass playing, in addition to group member's fascination with them. The harpsichord idea was first considered because it is one of the oldest mechanical methods of playing a stringed instrument, and it would be fairly easy to control. The rotating pick idea was discussed because it is an easy and simple solution to design.

We chose to use the linkage because it has the best musical characteristics, even though from a pure engineering point of view it would be the worst choice. Accurate musical representation is the paramount idea that this project is based around. Having poor acoustical qualities is something that we would not tolerate, similar to that of most musicians. In the case that we could not get the linkage working in time we would have chosen to use the harpsichord method. This was chosen as the backup plan over the rotating pick, because of the simplicity to control, even though it would be harder to make. We are confident in our manufacturing and machining skill, so we feel that a harder to manufacture, but easier to control solution would be better than the opposite.

Linkage

Motors

To drive the linkages, we will be using a stepper motor for each linkage. Stepper motors were selected because of the ease of stopping. We need to be able to stop quickly and easily because in order to properly and accurately pluck the string on time, and not pluck more or less than intended. We need the linkage to start and stop in known locations. There was discussion as to using a dc motor with a limit switch to stop the motor after the string has been plucked. Originally it was thought that the high cost of a stepper motor and driver would exceed the budget of this project, as the most exact drivers cost around five hundred a piece. However after much searching through suppliers we found sufficient stepper drivers that cost less than ten dollars apiece.

Design

Once the linkage design concept had been selected the process of selecting a suitable linkage began in the *Analysis of the Four-Bar Linkage* to select a linkage with the correct coupler curve. The focus of the curve was on the cusp at bottom of the curve which is where the string will be engage by the plucking finger. The rest of the curve is not as important but a quick return linkage is preferable as it give the bass the ability to play faster notes on the same string. The original idea for an extending linkage was found to be too complex at this point due to the changing nature of the coupler curve and the precision at which the link lengths would need to change. This would require a multitude of linear actuators electric, pneumatic, or hydraulic and would increase the minimum time between notes on separate strings as well as increase the complexity of the control circuitry and logic by a substantial amount. With a simpler linkage to design a four-bar linkage was selected. The coupler curve and be seen the Figure 33 note that the ground link is parallel with the x-axis in the final design this will be elevated at some angle as to place the cusp at a location ideal for engaging the strings.

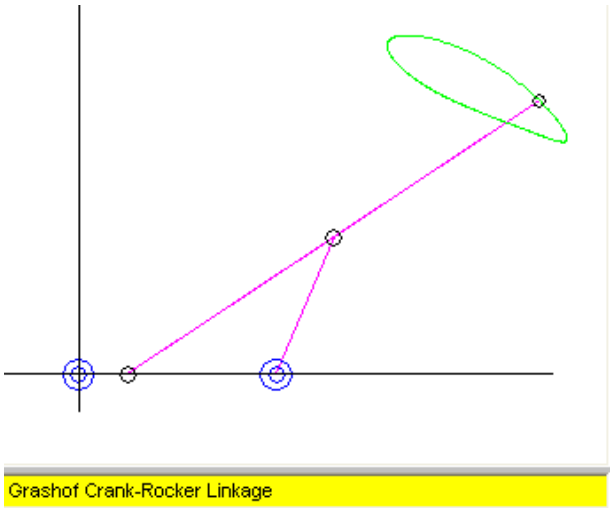

Figure 33: Screenshot from FOURbar depicting the coupler curve of the linkage.

Kinematic Analysis

Once the linkage design concept had been selected the process of selecting a suitable linkage began in the *Analysis of the Four-Bar Linkage* to select a linkage with the correct coupler curve. The focus of the curve was on the cusp at bottom of the curve which is where the string will be engage by the plucking finger. The rest of the curve is not as important but a quick return linkage is preferable as it give the bass the ability to play faster notes on the same string. The original idea for an extending linkage was found to be too complex at this point due to the changing nature of the coupler curve and the precision at which the link lengths would need to change. This would require a multitude of linear actuators electric, pneumatic, or hydraulic and would increase the minimum time between notes on separate strings as well as increase the complexity of the control circuitry and logic by a substantial amount. With a simpler linkage to design a four-bar linkage was selected. The coupler curve and be seen the Figure 33 note that the ground link is parallel with the x-axis in the final design this will be elevated at some angle as to place the cusp at a location ideal for engaging the strings.

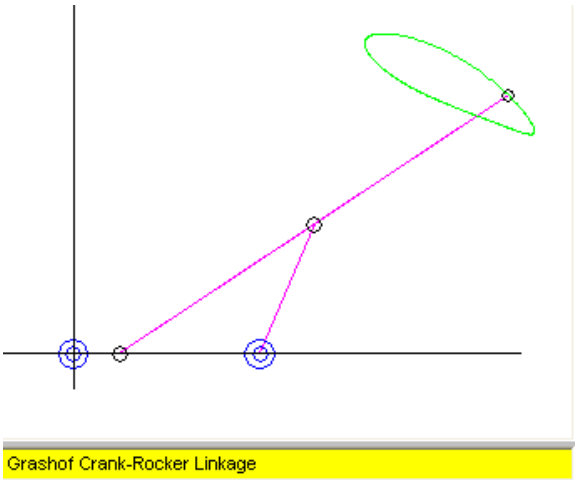

Figure 34: Screenshot from FOURbar depicting the coupler curve of the linkage.

Dynamic Analysis

The initial analysis of the linkage was done using the FOURbar software part of the Design of Machinery software package created by WPI professor Robert Norton. The software is designed to calculate different kinematic aspects of the linkages movement. This data was not as important for this project as for others but with this data calculated the software can then do a dynamic analysis on the linkage calculating forces and torques on the individual links. For this analysis the software requires information about the weight and moments of inertia of the links, as well as any outside forces acting on the links.

Link	Mass	Moment of Inertia
	(lbs)	(in-lbs-s^2)
Crank	0.00335	0.000284
Rocker	0.0064	0.002141
Coupler	0.0152	0.03168

Table 8: Masses and moments of inertial for linkage links.

Utilizing solid models created in SolidWorks and utilizing the materials database and mass properties functions within the program the weights and moments of inertia were calculated and can be seen in Table 8**.** The only outside force acting on the linkage would be the coupler engaging with the string to pluck the note. FOURbar can simulate forces acting in different positions but cannot simulate forces that vary over time. For the majority of a revolution there would be no outside force on the link but during the engagement with the string a force would develop until the string disengages to pluck the note. The maximum this force would be is 2 lbf measured just above the basses pickups with a spring scale. Using FOURbar the linkage was simulated as if this 2 lbf force was applied constantly at the coupler point. Form these calculations the torque requirement of the linkage was well beyond anything a reasonably economic motor could produce. The torque required is proportional to the size of the crank driving the system based on the equation of a moment arm of $T = F * l$ with T being the torque, F the force applied and, l the length of the moment arm. To reduce the torque requirement the length of the crank was reduced to 50% of its original size and to maintain a similar coupler curve the crank was reduce to 75% of its original size. The comparison of the two coupler curves and torques can be seen in Figure 35.

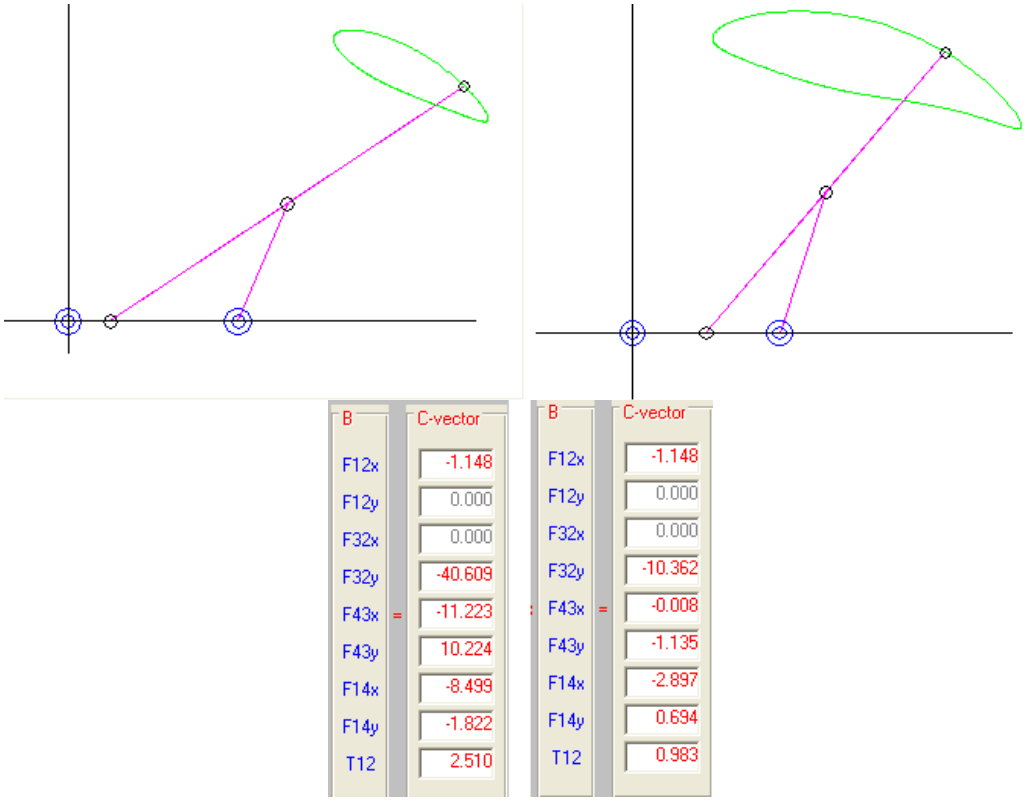

Figure 35: Coupler curve and torque comparison between initial linkage (left) and the final linkage (right).

All of the torque calculations were done at a rotational speed of 200 RPM which translates to an equal number of notes per minute being able to be played. A goal of at least 180 notes per minute was desired as this is a reasonable maximum for an above average human bass player. The torque from stepper motor decreases with increased rotational speed and a graph or formula is often provided by the manufacturer. Using this information and data calculated from FOURbar two lines were plotted on

the graph in Figure 36 - Torque - Speed Characteristics of the Stepper Motors and using this information a motor and rotational speed could be selected. The linkage torque requirements were doubled on the graph to include a factor of safety. A speed of 260 RPM would be ideal; this was determined by where the motor torque capabilities and the linkage torque requirements intersect. This speed would produce enough force to result in successful string plucking without applying a force too strong that would break the strings.

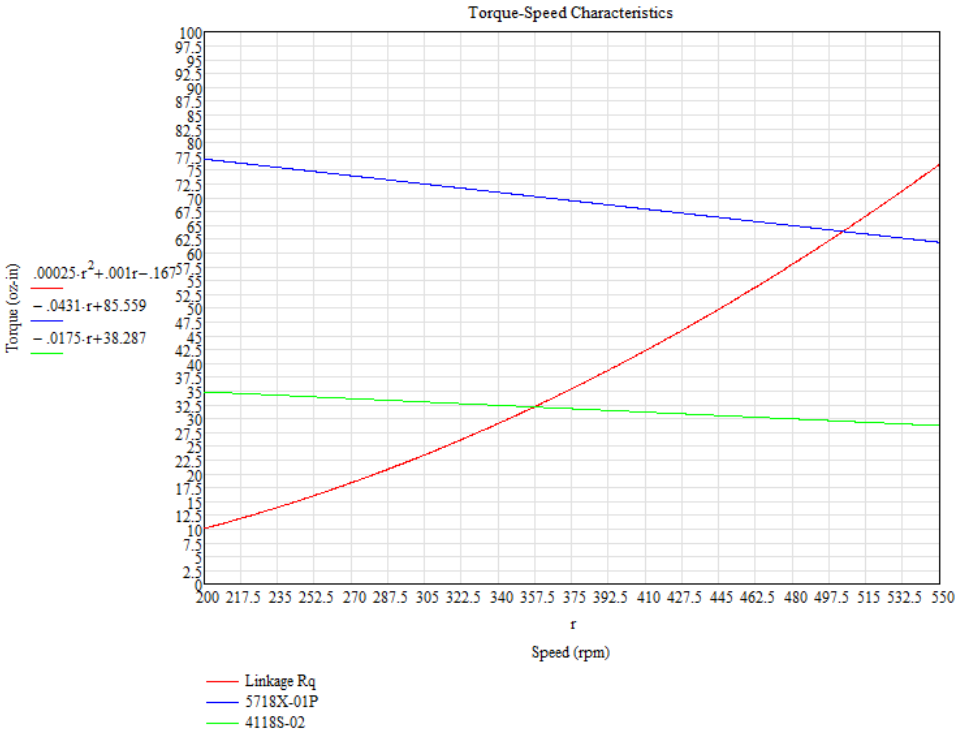

Figure 36 - Torque - Speed Characteristics of the Stepper Motors

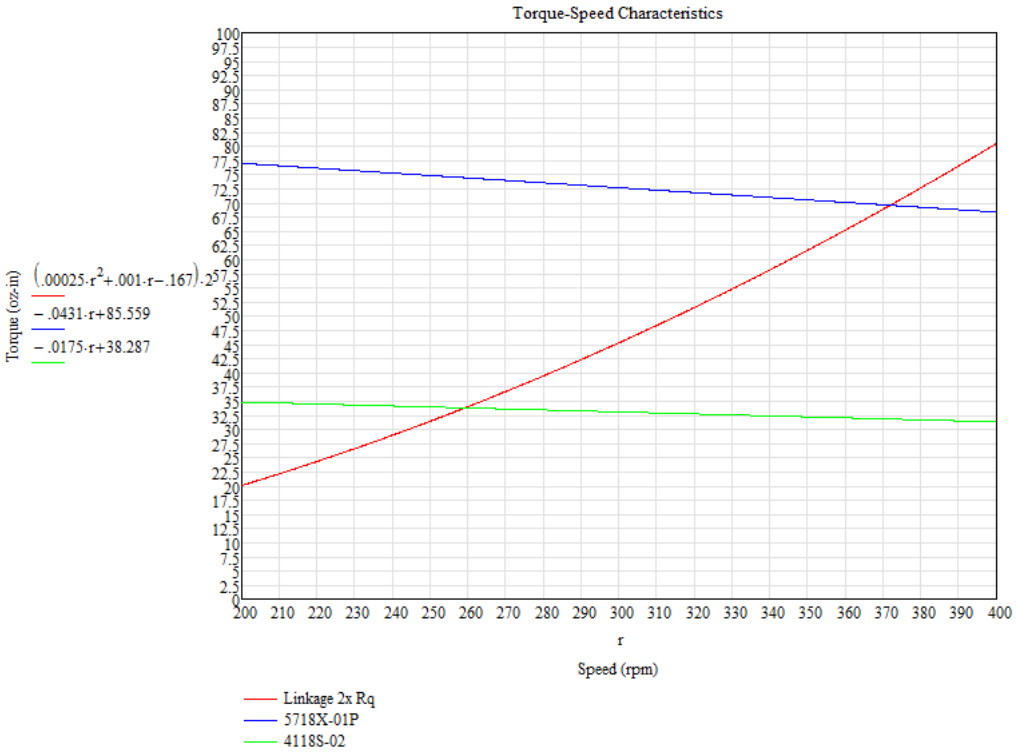

Figure 37 - Torque-Speed Characteristic (2x Requirement)

Stress and Deflection Analysis

Once the linkage was proven to work in calculations the links needed to be analyzed to see if they can survive the forces. To begin a machine design analysis a material had to be selected for this analysis both aluminum and acrylic were selected to be tested. Utilizing singularity equations the forces, stresses, slope, and deflection can be found. As can be seen in the graphs in Figure 38 if acrylic were used when the coupler was fully engaged with the string the acrylic would be deflecting almost 0.4 in. Due to the amount of deflection acrylic needed to be ruled out and further analysis of aluminum was conducted. Checking the ultimate tensile strength for static loading, dynamic loading, and for fatigue. Assuming a design for infinite life aluminum has factors of safety ranging from static: 17 to fatigue: 8.5, this shows that aluminum is overly designed for this project. The reason aluminum was ultimately selected was its ease of machining and ease of acquisition. Other materials with elastic modulus and ultimate tensile strengths between that of aluminum and acrylic would be better suited for use in this application but are not as readily available for use at WPI.

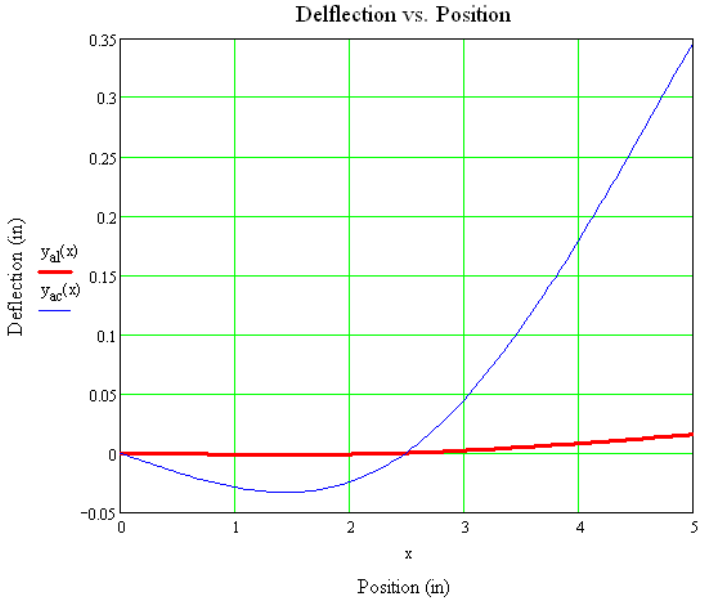

Figure 38: Plot of deflection vs. position along the coupler link for aluminum (red) and acrylic (blue)

Frame

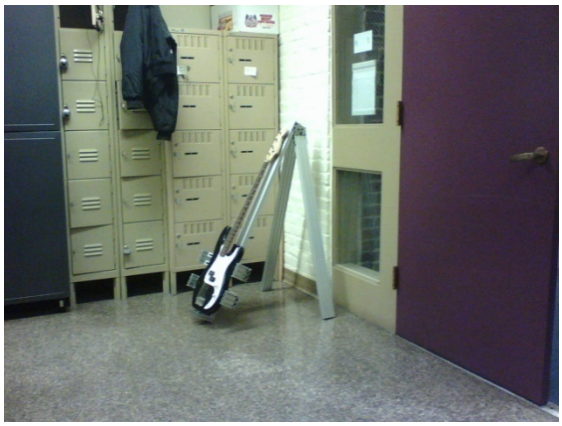

Figure 39 - The Frame Holding the Bass

For all of the fretting and plucking components to work consistently they must be held in a fixed position above the bass. To have these components remain in the proper location the bass must be held fixed position as well. A frame of some kind needed to be developed to mount both the bass and the fretting/plucking components. This frame had to be sturdy enough to support the weight of the bass and all of the components used to play it

Design


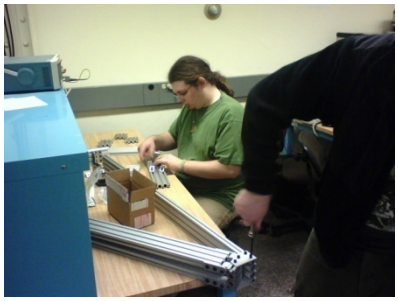


Figure 40 - The Frame Being Assembled

A general "spine" was designed with places to mount the solenoid mounting system for the fretting, and the motor and linkage for plucking. This spine was designed to position the components correctly above the bass this spine was then extrapolated into an instrument stand similar to the look of commercially available stands. The main difference with the frame used for the project and a commercial stand is that a commercial stand is not designed to support anything other than an instrument. The design for the frame as seen in Figure 41 - Isometric View of Final Frame Design**Error! Reference source not found.** with the fretting assembly mounted on the main spine of the frame and the plucking assemblies mounted to the large plate near the base of the structure.

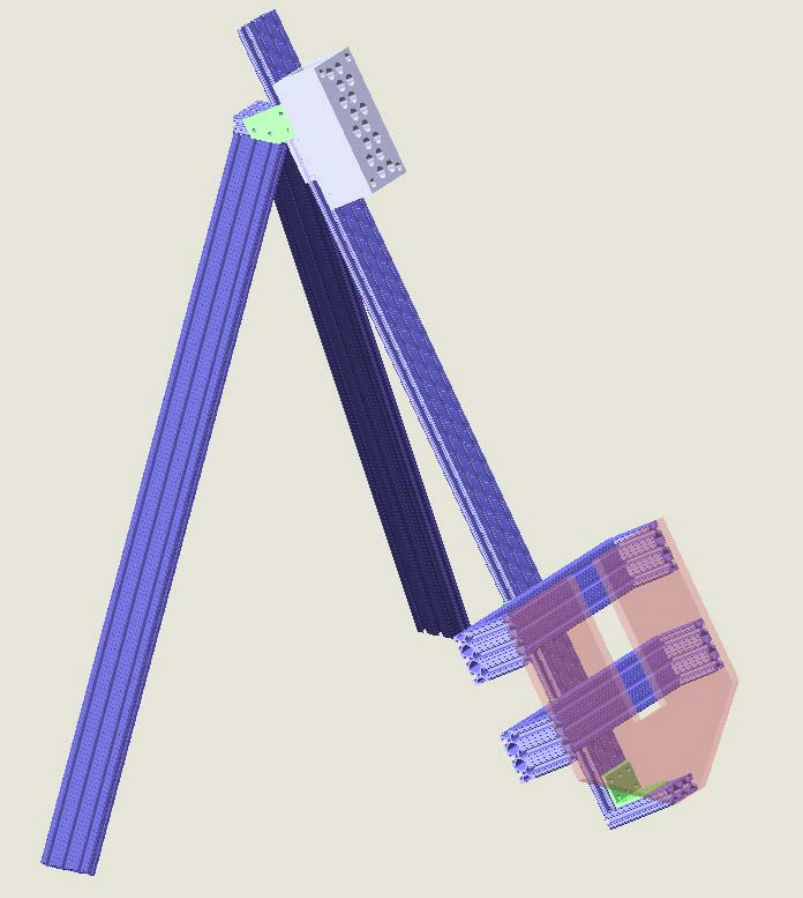

Figure 41 - Isometric View of Final Frame Design

The material selected to create the frame was 80/20 extruded aluminum, a commercially available system designed for rapid prototyping and modular construction. Both the rapid prototyping and modular properties of the material were desirable for this project. The modular design is necessary to allow the fretting head be located properly as the tight packing of the solenoids the tolerance for placement is small.

Analysis

Figure 42 - Adam Enjoying His Accomplishments

Strength of the frame while necessary was not the key element in its design, after selecting a design for its rapid prototyping and modular abilities it needed to be analyzed to see if it would hold up to the weight of the bass, mechanical, and electrical components. A basic static analysis included the weight of the instrument, weight of the solenoids and mounts, and the weight of the motors, the weight of the linkages were not included as they are of an order of magnitude less than all the other components. Utilizing two, 2-dimensional, static analysis the reaction forces at the joints of the frame were determined and then utilizing machine design singularity functions deflection plots could be created due to the complex nature of the cross section of 80/20 the functions could not be fully defined and therefore not plotted. When assembled the frame is exceedingly stable and as the frame was not designed for its strength or cost, but for rapid prototyping and modular nature this is not a reflection of a production model frame.

Safety

Safety is an often neglected yet very important aspect of designs. We are concerned with safety in respect to the user and in respect to the components. The foreseen risks associated with user safety are pinch points with the solenoids and the linkages, sharp edges, and falling over. We have calculated that the motors will stall before they can cause damage if something got in the path of the linkage. While it may not be pleasant if a finger was pinched with the linkage, because of the motor stalling, the worst that will happen is some discomfort, but not permanent damage. The solenoids also have a pinch point, but after some testing we have concluded that they will also only cause some minor discomfort but no permanent damage. We tried to smooth all of the possible sharp edges so that people can not injure themselves if they run their hand over it. The biggest safety hazard is if it fell over, as it weighs approximately 50 pounds with a bass in it. If the whole thing fell onto something or someone, it could cause some permanent damage or at least a large amount of pain. Although the frame is stable, accidents happen, so we designed cross bracing to reduce the chance of the frame falling over.

Component safety is an aspect that we spent a lot of time on when choosing parts. All of the electrical components have been chosen so that the components maximum allot able power is more than 150% of the maximum power it should receive. All of the connections have been isolated from each other to prevent them from causing a short circuit. All of the mechanical parts have been selected with a factor of safety of at least one and half and in most cases greater than two. This was done so that any forces sustained by them would be no where near their respective breaking points.

Recommendations for Further Work

Even if we have a functional project, there will always be a place for improvement. One the biggest areas that will need improvement and much further work are the chord detection system. We were not able to get this system off the ground, but we feel that given more time it would be possible to accurately identify input from a guitar. We believe that the best way to do this would be to implement filters with bandwidths of a single note. We feel that this approach will give fast and accurate results, and at the same time remain eloquently simple.

Simplicity is an aspect that we feel will help further work on this project. We saw many times how a more complex solution interfered with other systems of the project, in addition to leading to lengthy discussions of how feasible they were. Keep It Simple Stupid or K.I.S.S. was a working motto of out project, and we feel keeping it will allow for more features and functionality to be added in working order.

We would also like to increase the octaves that this device could play. This would allow for The Thumper to be a better bassist with increased expression ability. This can be implemented by increasing the number of frets that could be possibly played. However the approach we used of using solenoids to fret the string would become harder and harder the higher you wanted to play on the fret board.

Another idea that we would have liked to implement is increased musical knowledge. We discussed having two different modes that this project would work in; a rehearsal mode and a performance mode. In the rehearsal mode, the user would play the pieces of music they intended to use for an actual performance. The system would record what was played; figure out the chord progressions and key and time signatures used. Knowing ahead of time what would be played could allow The Thumper to provide more exciting accompaniment, as it would have better understanding of music theory. Ultimately we would love for The Thumper to be able to act just like another musician including being able to solo and jam.

Chord Detection should be completed. We researched several possibilities, but were unable to do much work on any of them due to time constraints. The chord detection, identifying input from an electric guitar, and controlling the robot bass to act as accompaniment would be an excellent opportunity to improve the design. Adding capability to receive some information from a MIDI pickup on a guitar, through the MIDI Interpreter, would be another option, although it would require more in-depth knowledge of the working of the pickup. A controls design that incorporates the MicroBlaze or

PicoBlaze soft-core processors, to allow for the later addition of the chord detection algorithms, would allow less time to completion.

The solenoid driver board could use some modification to be made more compact, and to ensure that all components fit as intended. Parts need to be checked more thoroughly before they are ordered to prevent the types of size issues we had. Research could be done with an effort to find better or more efficient drivers. Creating a more compact and less complicated circuit would add value to project in terms of ease of assembly and, if the project were to be marketed, ease of production. As the board was our first printed circuit board layout, there were some mistakes made, and it could be greatly improved by someone with some experience in that area. More research should be done on possible solenoids. The ones used for this build were larger than we would have liked, but it seemed we could not find one smaller that produced enough force to fret the string.

The area of controls for this project is an excellent opportunity for more work. The current controls system was designed with utilitarian purpose. It was meant to perform its function, but at the same time have as little impact of the overall schedule as possible. Creating a more capable MIDI interface would be excellent starting point, such as a system that could respond meaningfully to some of the other commands defined in the MIDI protocol. The UART used in the design could be improved, or a better one created. A faster and smaller UART or one that included only a receiver could decrease the number if blocks used in the FPGA, meaning smaller chips could be used. Stepper motor drivers like any of those made by Lin Engineering, or possible the use of their SilverPak series of integrated stepper motors would streamline the project significantly. It would also eliminate the need for one of our PCBs. Also, while the FPGA was definitely the proper choice for our build, the particular one we chose was much more than we needed. A smaller one could be used, such as the Spartan-3 kit, also available from Xilinx. It has a smaller FPGA and less capability, but the control system could be modified to work with it.

There is a possibility that more suitable stepper motors exist, but we were unable to find many that met our requirements while still being available from normal distributors. Newark seemed to be the only distributor that carried steppers large enough. They had the 4118S-02 and one larger motor, both of which were larger than necessary for the task at hand. More suitable drivers, as mentioned above, would also help, but they were prohibitively expensive for the scope of the current build.

Much of the material selected to be used in the project was overly strong or expensive for a production level project. The 80/20 extruded aluminum for the frame in particular was selected because of its rapid prototyping capabilities and modular design, without which this project would not have fit together correctly. Many of the other parts of the bass were made from machine aluminum many of which could have been cast aluminum or even plastic.

Conclusion

This is a really good example of how the real engineering world differs from the ideal engineering world. In the perfect engineering world there is an unlimited budget and no deadline. We soon discovered how different the real world is from the ideal case, as the budget is an aspect that has

been looming over our heads for the majority of this project. The deadline of project presentation day has been exponentially increasing our awareness of time lines, as we approach it. In the real engineering world, compromises need to be made between time and money. If the budget must remain static, then in most cases the deadline must be lengthened to achieve the design goals without exceeding the budget, as earlier designs may have been easier, but more costly. In our case of a university mandated deadline, the budget would have to increase in order to accomplish our goals on time.

Although all of the components work in theory, the project does not yet fully function because of issues with differences in connector types, but we have replacement connectors on the way. Once the connectors have arrived we will be able to connect all of the components together. Barring any unseen issues, all of the systems should work together, as they all work independently. We may need to alter the control code if the components are not exactly as stated in the manufacturer's data sheets. This is an example of one the biggest lessons we learned from this project; check part sizes and packaging. There was a large amount of errors made by choosing the wrong part size or packaging, especially with surface mount components on all of the printed circuit boards. In future designs we will defiantly check to ensure that all part sizes match the footprint, and that connectors are of the appropriate type.

This lesson is part of a bigger lesson known as Murphy's Law; anything that can go wrong, will go wrong at the worst time. This can be designed for by using Ockham's razor; if all other things are equal, the simplest choice is the best. There are always setbacks, but with a simple design, setbacks have a smaller impact as it is easier to overcome them, allowing less time to be lost solving problems. Both of these principles have been quite prevalent in this project. We analyzed each design for areas of possible failure and chose the simplest solution that would fit the requirements.

A very unique aspect of this project was the integration of different majors, something not done on MQP's. For this project to be successful the mechanical and electrical aspects needed to work together in harmony which means they needed to be designed together. Many compromises had to be made on both sides of the project for systems to work properly. Many deadlines needed to be changed to accommodate different stages of the design process for each engineering discipline. This synergy of different engineering groups can be seen as a good representation of engineering in industry where designs groups will be interdisciplinary. Throughout this project our advisors continually stated how much like industry this project was when in weekly design review/progress reports we brought up issues of the mechanical and electrical designs conflicting.

Appendix – VHDL Code

The following is the VHDL code used to generate the controls of the project.

UART Core

This is the modified UART core from OpenCores.org

uart_lib.vhd

This VHDL file is a definition core for the miniUART VHDL module. It defines the functions used to create the UART that are not included in the standard IEEE libraries.

```
--
------------------------------------------------------------------------------
--
--  S Y N T H E Z I A B L E    miniUART   C O R E
--
--  www.OpenCores.Org - January 2000
--  This core adheres to the GNU public license
--
-- Design units   : UART_Def
--
-- File name      : uart_lib.vhd
--
-- Purpose        : Implements an miniUART device for communication purposes
--                  between the OR1K processor and the Host computer through
--                  an RS-232 communication protocol.
--
-- Library        : uart_lib.vhd
--
-- Dependencies   : IEEE.Std_Logic_1164
--
--
------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--
-- Revision list
-- Version   Author                 Date                         Changes
--
-- 0.1      Ovidiu Lupas     15 January 2000                  New model
--        olupas@opencores.org
--------------------------------------------------------------------------------
--
--------------------------------------------------------------------------------
---
-- package UART_Def
--------------------------------------------------------------------------------
---
library IEEE,STD;
use IEEE.Std_Logic_1164.all;
use IEEE.Numeric_Std.all;
--**--
package UART_Def is
      --------------------------------------------------------------------------
------
```

```
      -- Converts unsigned Std_LOGIC_Vector to Integer, leftmost bit is MSB
      -- Error message for unknowns (U, X, W, Z, -), converted to 0
      -- Verifies whether vector is too long (> 16 bits)
      ------------------------------------------------------------------------
------
      function  ToInteger (
         Invector : in  Unsigned(3 downto 0))
       return     Integer;
end UART_Def; --==================== End of package header
=====================--
package body UART_Def is
  function  ToInteger (
       InVector : in Unsigned(3 downto 0))
      return  Integer is
    constant HeaderMsg   : String          := "To_Integer:";
    constant MsgSeverity : Severity_Level  := Warning;
    variable Value       : Integer         := 0;
  begin
    for i in 0 to 3 loop
      if (InVector(i) = '1') then
         Value := Value + (2**I);
      end if;
    end loop;
    return Value;
  end ToInteger;
end UART_Def; --================ End of package body ================--
```

clkUnit.vhd

This is the clock unit for the miniUART core. This takes in a board clock and divides down to the appropriate speed. We modified this to use the 50 MHz board clock that the Spartan-3E Starter Kit comes equipped with and had it divide down to the 31.25 kHz required by the MIDI specification.

```
--=============================================================================--
--
--  S Y N T H E Z I A B L E    miniUART   C O R E
--
--  www.OpenCores.Org - January 2000
--  This core adheres to the GNU public license

-- Design units   : miniUART core for the OCRP-1
--
-- File name      : clkUnit.vhd
--
-- Purpose        : Implements an miniUART device for communication purposes
--                  between the OR1K processor and the Host computer through
--                  an RS-232 communication protocol.
--
-- Library        : uart_lib.vhd
--
-- Dependencies   : IEEE.Std_Logic_1164
--
--=============================================================================--
```

```
--------------------------------------------------------------------------------
--
-- Revision list
-- Version   Author              Date                Changes
--
-- 1.0     Ovidiu Lupas      15 January 2000         New model
-- 1.1     Ovidiu Lupas      28 May 2000     EnableRx/EnableTx ratio
corrected
--      olupas@opencores.org
--------------------------------------------------------------------------------
--
-- Description    : Generates the Baud clock and enable signals for RX & TX
--                  units.
--------------------------------------------------------------------------------
--
-- Entity for Baud rate generator Unit - 9600 baudrate
--
--------------------------------------------------------------------------------
--
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
library work;
   use work.UART_Def.all;
--------------------------------------------------------------------------------
--
-- Baud rate generator
--------------------------------------------------------------------------------
--
entity ClkUnit is
  port (
     SysClk   : in  Std_Logic;  -- System Clock
     EnableRx : out Std_Logic;  -- Control signal
     EnableTx : out Std_Logic;  -- Control signal
     Reset    : in  Std_Logic); -- Reset input
end entity; --================== End of entity
==============================--
--------------------------------------------------------------------------------
--
-- Architecture for Baud rate generator Unit
--------------------------------------------------------------------------------
--
architecture Behaviour of ClkUnit is
  --------------------------------------------------------------------------------
--
  -- Signals
  --------------------------------------------------------------------------------
--
  signal ClkDiv10a  : Std_Logic;
  signal tmpEnRX    : Std_Logic;
  signal tmpEnTX    : Std_Logic;
begin
  --------------------------------------------------------------------------------
--
  -- Divides the system clock of 40 MHz by 26
  -- Modified to divide 50 MHz by 10    --MB
```

```
  ------------------------------------------------------------------------
--
  DivClk10a : process(SysClk,Reset)
     constant CntOne : unsigned(3 downto 0) := "0001";
     variable Cnt10a  : unsigned(3 downto 0);
  begin
     if Rising_Edge(SysClk) then
        if Reset = '0' then
           Cnt10a := "0000";
           ClkDiv10a <= '0';
        else
           Cnt10a := Cnt10a + CntOne;
           case Cnt10a is
              when "1010" =>
                  ClkDiv10a <= '1';
                  Cnt10a := "0000";
              when others =>
                  ClkDiv10a <= '0';
           end case;
        end if;
     end if;
  end process;
  ------------------------------------------------------------------------
--
  -- Provides the EnableRX signal, at ~ 155 KHz
  -- Still divides by 10. Results in 500 kHz  --MB
  ------------------------------------------------------------------------
--
  DivClk10 : process(SysClk,Reset,ClkDiv10a)
     constant CntOne : unsigned(3 downto 0) := "0001";
     variable Cnt10  : unsigned(3 downto 0);
  begin
     if Rising_Edge(SysClk) then
        if Reset = '0' then
           Cnt10 := "0000";
           tmpEnRX <= '0';
        elsif ClkDiv10a = '1' then
           Cnt10 := Cnt10 + CntOne;
        end if;
        case Cnt10 is
             when "1010" =>
                tmpEnRX <= '1';
                Cnt10 := "0000";
             when others =>
                tmpEnRX <= '0';
        end case;
     end if;
  end process;
  ------------------------------------------------------------------------
--
  -- Provides the EnableTX signal, at 9.6 KHz
  -- Still divides by 16. Results in 31.25 kHz  --MB
  ------------------------------------------------------------------------
--
  DivClk16 : process(SysClk,Reset,tmpEnRX)
     constant CntOne : unsigned(4 downto 0) := "00001";
     variable Cnt16  : unsigned(4 downto 0);
```

```
  begin
     if Rising_Edge(SysClk) then
        if Reset = '0' then
           Cnt16 := "00000";
           tmpEnTX <= '0';
        elsif tmpEnRX = '1' then
           Cnt16 := Cnt16 + CntOne;
        end if;
        case Cnt16 is
           when "01111" =>
                tmpEnTX <= '1';
                Cnt16 := Cnt16 + CntOne;
           when "10001" =>
                Cnt16 := "00000";
                tmpEnTX <= '0';
           when others =>
                tmpEnTX <= '0';
        end case;
     end if;
  end process;

  EnableRX <= tmpEnRX;
  EnableTX <= tmpEnTX;
end Behaviour; --==================== End of architecture
===================--
```

RxUnit.vhd

This is the receive unit of the miniUART core. It is given serial data which it turns into parallel bytes which can be used in the system.

```
--=============================================================================--
--
--  S Y N T H E Z I A B L E    miniUART   C O R E
--
--  www.OpenCores.Org - January 2000
--  This core adheres to the GNU public license
--
-- Design units   : miniUART core for the OCRP-1
--
-- File name      : RxUnit.vhd
--
-- Purpose        : Implements an miniUART device for communication purposes
--                  between the OR1K processor and the Host computer through
--                  an RS-232 communication protocol.
--
-- Library        : uart_lib.vhd
--
-- Dependencies   : IEEE.Std_Logic_1164
--
--
=============================================================================--
-------------------------------------------------------------------------------
--
-- Revision list
-- Version   Author                 Date                          Changes
```

```
--
-- 0.1      Ovidiu Lupas     15 January 2000                    New model
-- 2.0      Ovidiu Lupas     17 April   2000  samples counter cleared for bit
0
--        olupas@opencores.org
-------------------------------------------------------------------------------
--
-- Description    : Implements the receive unit of the miniUART core. Samples
--                  16 times the RxD line and retain the value in the middle
of
--                  the time interval.
-------------------------------------------------------------------------------
--
-- Entity for Receive Unit - 9600 baudrate                                    -
-
-------------------------------------------------------------------------------
--
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
library work;
   use work.UART_Def.all;
-------------------------------------------------------------------------------
--
-- Receive unit
-------------------------------------------------------------------------------
--
entity RxUnit is
  port (
     Clk    : in  Std_Logic;  -- system clock signal
     Reset  : in  Std_Logic;  -- Reset input
     Enable : in  Std_Logic;  -- Enable input
     RxD    : in  Std_Logic;  -- RS-232 data input
     RD     : in  Std_Logic;  -- Read data signal
     FErr   : out Std_Logic;  -- Status signal
     OErr   : out Std_Logic;  -- Status signal
     DRdy   : out Std_Logic;  -- Status signal
     DataIn : out Std_Logic_Vector(7 downto 0));
end entity; --================== End of entity
==============================--
-------------------------------------------------------------------------------
--
-- Architecture for receive Unit
-------------------------------------------------------------------------------
--
architecture Behaviour of RxUnit is
  -----------------------------------------------------------------------------
--
  -- Signals
  -----------------------------------------------------------------------------
--
  signal Start     : Std_Logic;             -- Syncro signal
  signal tmpRxD    : Std_Logic;             -- RxD buffer
  signal tmpDRdy   : Std_Logic;             -- Data ready buffer
  signal outErr    : Std_Logic;             --
  signal frameErr  : Std_Logic;             --
  signal BitCnt    : Unsigned(3 downto 0);  --
```

```
  signal SampleCnt : Unsigned(3 downto 0);  -- samples on one bit counter
  signal ShtReg    : Std_Logic_Vector(7 downto 0);  --
  signal DOut      : Std_Logic_Vector(7 downto 0);  --
begin
  ------------------------------------------------------------------------
  -- Receiver process
  ------------------------------------------------------------------------
  RcvProc : process(Clk,Reset,Enable,RxD)
      variable tmpBitCnt    : Integer range 0 to 15;
      variable tmpSampleCnt : Integer range 0 to 15;
      constant CntOne       : Unsigned(3 downto 0):="0001";
  begin
     if Rising_Edge(Clk) then
        tmpBitCnt := ToInteger(BitCnt);
        tmpSampleCnt := ToInteger(SampleCnt);
        if Reset = '0' then
           BitCnt <= "0000";
           SampleCnt <= "0000";
           Start <= '0';
           tmpDRdy <= '0';
           frameErr <= '0';
           outErr <= '0';

           ShtReg <= "00000000";  --
           DOut   <= "00000000";  --
        else
           if RD = '1' then
              tmpDRdy <= '0';      -- Data was read
           end if;

           if Enable = '1' then
              if Start = '0' then
                 if RxD = '0' then -- Start bit,
                    SampleCnt <= SampleCnt + CntOne;
                    Start <= '1';
                 end if;
              else
                 if tmpSampleCnt = 8 then  -- reads the RxD line
                    tmpRxD <= RxD;
                    SampleCnt <= SampleCnt + CntOne;
                 elsif tmpSampleCnt = 15 then
                    case tmpBitCnt is
                         when 0 =>
                                if tmpRxD = '1' then -- Start Bit
                                   Start <= '0';
                                else
                                   BitCnt <= BitCnt + CntOne;
                                end if;
                                SampleCnt <= SampleCnt + CntOne;
                         when 1|2|3|4|5|6|7|8 =>
                                BitCnt <= BitCnt + CntOne;
                                SampleCnt <= SampleCnt + CntOne;
                                ShtReg <= tmpRxD & ShtReg(7 downto 1);
                         when 9 =>
                                if tmpRxD = '0' then  -- stop bit expected
                                   frameErr <= '1';
                                else
```

```
                                    frameErr <= '0';
                                 end if;

                                 if tmpDRdy = '1' then --
                                    outErr <= '1';
                                 else
                                    outErr <= '0';
                                 end if;

                                 tmpDRdy <= '1';
                                 DOut <= ShtReg;
                                 BitCnt <= "0000";
                                 Start <= '0';
                           when others =>
                                 null;
                     end case;
                  else
                     SampleCnt <= SampleCnt + CntOne;
                  end if;
               end if;
            end if;
         end if;
      end if;
   end process;

   DRdy <= tmpDRdy;
   DataIn <= DOut;
   FErr <= frameErr;
   OErr <= outErr;

end Behaviour; --==================== End of architecture
====================--
```

TxUnit.vhd

This is the transmit section of the UART core. It is unnecessary for this project, but must be included for the miniUART core to compile and build without severe modification.

```
--
------------------------------------------------------------------------------
--
--  S Y N T H E Z I A B L E    miniUART   C O R E
--
--  www.OpenCores.Org - January 2000
--  This core adheres to the GNU public license
--
-- Design units   : miniUART core for the OCRP-1
--
-- File name      : TxUnit.vhd
--
-- Purpose        : Implements an miniUART device for communication purposes
--                  between the OR1K processor and the Host computer through
--                  an RS-232 communication protocol.
--
-- Library        : uart_lib.vhd
--
```

```
-- Dependencies   : IEEE.Std_Logic_1164
--
--
================================================================================--
--------------------------------------------------------------------------------
--
-- Revision list
-- Version   Author                 Date                          Changes
--
-- 0.1      Ovidiu Lupas       15 January 2000                 New model
-- 2.0      Ovidiu Lupas       17 April   2000    unnecessary variable
removed
--  olupas@opencores.org
--------------------------------------------------------------------------------
--
-- Description    :
--------------------------------------------------------------------------------
--
-- Entity for the Tx Unit
--
--------------------------------------------------------------------------------
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.Uart_Def.all;
--------------------------------------------------------------------------------
--
-- Transmitter unit
--------------------------------------------------------------------------------
--
entity TxUnit is
  port (
     Clk    : in  Std_Logic;  -- Clock signal
     Reset  : in  Std_Logic;  -- Reset input
     Enable : in  Std_Logic;  -- Enable input
     Load   : in  Std_Logic;  -- Load transmit data
     TxD    : out Std_Logic;  -- RS-232 data output
     TRegE  : out Std_Logic;  -- Tx register empty
     TBufE  : out Std_Logic;  -- Tx buffer empty
     DataO  : in  Std_Logic_Vector(7 downto 0));
end entity; --================== End of entity
==============================--
--------------------------------------------------------------------------------
--
-- Architecture for TxUnit
--------------------------------------------------------------------------------
--
architecture Behaviour of TxUnit is
  ------------------------------------------------------------------------------
--
  -- Signals
  ------------------------------------------------------------------------------
--
  signal TBuff    : Std_Logic_Vector(7 downto 0); -- transmit buffer
  signal TReg     : Std_Logic_Vector(7 downto 0); -- transmit register
```

```
  signal BitCnt   : Unsigned(3 downto 0);         -- bit counter
  signal tmpTRegE : Std_Logic;                    --
  signal tmpTBufE : Std_Logic;                    --
begin
  ------------------------------------------------------------------------
--
  -- Implements the Tx unit
  ------------------------------------------------------------------------
--
  process(Clk,Reset,Enable,Load,DataO,TBuff,TReg,tmpTRegE,tmpTBufE)
      variable tmp_TRegE : Std_Logic;
      constant CntOne    : Unsigned(3 downto 0):="0001";
  begin
     if Rising_Edge(Clk) then
        if Reset = '0' then
           tmpTRegE <= '1';
           tmpTBufE <= '1';
           TxD <= '1';
           BitCnt <= "0000";
        elsif Load = '1' then
           TBuff <= DataO;
           tmpTBufE <= '0';
        elsif Enable = '1' then
           if ( tmpTBufE = '0') and (tmpTRegE = '1') then
              TReg <= TBuff;
              tmpTRegE <= '0';
--              tmp_TRegE := '0';
              tmpTBufE <= '1';
--           else
--              tmp_TRegE := tmpTRegE;
           end if;

           if tmpTRegE = '0' then
              case BitCnt is
                  when "0000" =>
                          TxD <= '0';
                          BitCnt <= BitCnt + CntOne;
                  when "0001" | "0010" | "0011" |
                       "0100" | "0101" | "0110" |
                       "0111" | "1000" =>
                          TxD <= TReg(0);
                          TReg <= '1' & TReg(7 downto 1);
                          BitCnt <= BitCnt + CntOne;
                  when "1001" =>
                          TxD <= '1';
                          TReg <= '1' & TReg(7 downto 1);
                          BitCnt <= "0000";
                          tmpTRegE <= '1';
                  when others => null;
              end case;
           end if;
        end if;
     end if;
  end process;

      TRegE <= tmpTRegE;
      TBufE <= tmpTBufE;
```

```
end Behaviour; --=================== End of architecture
====================--
```

miniUART.vhd

This is the main module of the miniUART core that was downloaded from Opencores.org. It takes in serial data and passes it to the receiver section. It also takes serial data from the transmitter section and sends it to whatever its talking to. In this project, however, the later is not needed, and will be synthesized out during building of the core in the Xilinx tools.

```
--
=============================================================================--
--
--  S Y N T H E Z I A B L E    miniUART   C O R E
--
--  www.OpenCores.Org - January 2000
--  This core adheres to the GNU public license
--
-- Design units   : miniUART core for the OCRP-1
--
-- File name      : miniuart.vhd
--
-- Purpose        : Implements an miniUART device for communication purposes
--                  between the OR1K processor and the Host computer through
--                  an RS-232 communication protocol.
--
-- Library        : uart_lib.vhd
--
-- Dependencies   : IEEE.Std_Logic_1164
--
-- Simulator      : ModelSim PE/PLUS version 4.7b on a Windows95 PC
--
=============================================================================--
-----------------------------------------------------------------------------
--
-- Revision list
-- Version   Author                 Date           Changes
--
-- 0.1      Ovidiu Lupas     15 January 2000       New model
-- 1.0      Ovidiu Lupas     January  2000         Synthesis optimizations
-- 2.0      Ovidiu Lupas     April    2000         Bugs removed - RSBusCtrl
--          the RSBusCtrl did not process all possible situations
--
--        olupas@opencores.org
-----------------------------------------------------------------------------
--
-- Description    : The memory consists of a dual-port memory addressed by
--                  two counters (RdCnt & WrCnt). The third counter (StatCnt)
--                  sets the status signals and keeps a track of the data
flow.
-----------------------------------------------------------------------------
--
-- Entity for miniUART Unit - 9600 baudrate
--
```

```
--------------------------------------------------------------------------------
--
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
library work;
   use work.UART_Def.all;

entity miniUART is
  port (
     SysClk   : in  Std_Logic;  -- System Clock
     Reset    : in  Std_Logic;  -- Reset input
     CS_N     : in  Std_Logic;
     RD_N     : in  Std_Logic;
     WR_N     : in  Std_Logic;
     RxD      : in  Std_Logic;
     TxD      : out Std_Logic;
     IntRx_N  : out Std_Logic;  -- Receive interrupt
     IntTx_N  : out Std_Logic;  -- Transmit interrupt
     DRdy_out : out Std_Logic;
     Addr     : in  Std_Logic_Vector(1 downto 0); --
     DataIn   : in  Std_Logic_Vector(7 downto 0); --
     DataOut  : out Std_Logic_Vector(7 downto 0)); --
end entity; --=================== End of entity
==============================--
--------------------------------------------------------------------------------
--
-- Architecture for miniUART Controller Unit
--------------------------------------------------------------------------------
--
architecture uart of miniUART is
  --------------------------------------------------------------------------------
--
  -- Signals
  --------------------------------------------------------------------------------
--
  signal RxData : Std_Logic_Vector(7 downto 0); --
  signal TxData : Std_Logic_Vector(7 downto 0); --
  signal CSReg  : Std_Logic_Vector(7 downto 0); -- Ctrl & status register
  --              CSReg detailed
  -----------+--------+--------+--------+--------+--------+--------+--------+
  -- CSReg(7)|CSReg(6)|CSReg(5)|CSReg(4)|CSReg(3)|CSReg(2)|CSReg(1)|CSReg(0)|
  --    Res   |  Res   |  Res   |  Res   | UndRun | OvrRun |  FErr  |  OErr  |
  -----------+--------+--------+--------+--------+--------+--------+--------+
  signal EnabRx : Std_Logic;  -- Enable RX unit
  signal EnabTx : Std_Logic;  -- Enable TX unit
  signal DRdy   : Std_Logic;  -- Receive Data ready
  signal TRegE  : Std_Logic;  -- Transmit register empty
  signal TBufE  : Std_Logic;  -- Transmit buffer empty
  signal FErr   : Std_Logic;  -- Frame error
  signal OErr   : Std_Logic;  -- Output error
  signal Read   : Std_Logic;  -- Read receive buffer
  signal Load   : Std_Logic;  -- Load transmit buffer
  signal DRdy_buff : Std_Logic;
  --------------------------------------------------------------------------------
--
  -- Baud rate Generator
```

```
  ------------------------------------------------------------------------
--
  component ClkUnit is
   port (
     SysClk   : in  Std_Logic;  -- System Clock
     EnableRX : out Std_Logic;  -- Control signal
     EnableTX : out Std_Logic;  -- Control signal
     Reset    : in  Std_Logic); -- Reset input
  end component;
  ------------------------------------------------------------------------
--
  -- Receive Unit
  ------------------------------------------------------------------------
--
  component RxUnit is
  port (
     Clk    : in  Std_Logic;  -- Clock signal
     Reset  : in  Std_Logic;  -- Reset input
     Enable : in  Std_Logic;  -- Enable input
     RxD    : in  Std_Logic;  -- RS-232 data input
     RD     : in  Std_Logic;  -- Read data signal
     FErr   : out Std_Logic;  -- Status signal
     OErr   : out Std_Logic;  -- Status signal
     DRdy   : out Std_Logic;  -- Status signal
     DataIn : out Std_Logic_Vector(7 downto 0));
  end component;
  ------------------------------------------------------------------------
--
  -- Transmitter Unit
  ------------------------------------------------------------------------
--
  component TxUnit is
  port (
     Clk    : in  Std_Logic;  -- Clock signal
     Reset  : in  Std_Logic;  -- Reset input
     Enable : in  Std_Logic;  -- Enable input
     Load   : in  Std_Logic;  -- Load transmit data
     TxD    : out Std_Logic;  -- RS-232 data output
     TRegE  : out Std_Logic;  -- Tx register empty
     TBufE  : out Std_Logic;  -- Tx buffer empty
     DataO  : in  Std_Logic_Vector(7 downto 0));
  end component;
begin
  ------------------------------------------------------------------------
--
  -- Instantiation of internal components
  ------------------------------------------------------------------------
--
  ClkDiv  : ClkUnit port map (SysClk,EnabRX,EnabTX,Reset);
  TxDev   : TxUnit port map
(SysClk,Reset,EnabTX,Load,TxD,TRegE,TBufE,TxData);
  RxDev   : RxUnit port map
(SysClk,Reset,EnabRX,RxD,Read,FErr,OErr,DRdy,RxData);
  ------------------------------------------------------------------------
--
  -- Implements the controller for Rx&Tx units
```

```
  ------------------------------------------------------------------------
--
  RSBusCtrl : process(SysClk,Reset,Read,Load)
     variable StatM : Std_Logic_Vector(4 downto 0);
  begin
     if Rising_Edge(SysClk) then
        if Reset = '0' then
           StatM := "00000";
           IntTx_N <= '1';
           IntRx_N <= '1';
           CSReg <= "11110000";
        else
           StatM(0) := DRdy;
           DRdy_out <= DRdy;
           StatM(1) := FErr;
           StatM(2) := OErr;
           StatM(3) := TBufE;
           StatM(4) := TRegE;
        end if;
        case StatM is
             when "00001" =>
                  IntRx_N <= '0';
                  CSReg(2) <= '1';
             when "10001" =>
                  IntRx_N <= '0';
                  CSReg(2) <= '1';
             when "01000" =>
                  IntTx_N <= '0';
             when "11000" =>
                  IntTx_N <= '0';
                  CSReg(3) <= '1';
             when others => null;
        end case;

        if Read = '1' then
           CSReg(2) <= '0';
           IntRx_N <= '1';
        end if;

        if Load = '1' then
           CSReg(3) <= '0';
           IntTx_N <= '1';
        end if;
     end if;
  end process;
  ------------------------------------------------------------------------
--
  -- Combinational section
  ------------------------------------------------------------------------
--
  process(SysClk)
  begin
     if (CS_N = '0' and RD_N = '0') then
        Read <= '1';
     else Read <= '0';
     end if;
```

```
      if (CS_N = '0' and WR_N = '0') then
         Load <= '1';
      else Load <= '0';
      end if;

      if Read = '0' then
         DataOut <= "ZZZZZZZZ";
      elsif (Read = '1' and Addr = "00") then
         DataOut <= RxData;
      elsif (Read = '1' and Addr = "01") then
         DataOut <= CSReg;
      end if;

      if Load = '0' then
         TxData <= "ZZZZZZZZ";
      elsif (Load = '1' and Addr = "00") then
         TxData <= DataIn;
      end if;
  end process;
end uart; --===================== End of architecture
=======================--
```

MIDI Interpreter

These are the VHDL files used for the MIDI Interpreter

miditable1.vhd

This is the decoder for the MIDI Interpreter. It accepts input from the MIDI Interpreter main module and decides what actions should be taken.

```
-- Filename          : miditable1.vhd
-- Modelname         : MIDI Table
-- Title             : Specialized MIDI Decoder Module for MIDI Interpreter
-- Purpose           :
-- Author(s)         : Matt Brown, Barry Kosherick
-- Comment           :
-- Assumptions       :
-- Limitations       :
-- Known errors      :
-- Specification ref :
--------------------------------------------------------------------------------
--
-- Modification history:
--------------------------------------------------------------------------------
--
-- Version  | Author | Date       | Changes made
--------------------------------------------------------------------------------
--
-- 1.0      |   BK   | 15.11.2007 | initial version
--------------------------------------------------------------------------------
--
-- 1.3      |   MB   | 07.02.2008 | initial comnpleted version
--------------------------------------------------------------------------------
--
-- 1.6      |   MB   | 13.02.2008 | successful test and revised comments
```

```
------------------------------------------------------------------------------
--

library ieee;
use ieee.std_logic_1164.all;

entity miditable1 is
  port (
    in1    : in  std_logic_vector(23 downto 0);  -- input from MIDI
controller
    clk    : in  std_logic;                      -- clock signal
    reset  : in  std_logic;                      -- reset signal
    m_done : out std_logic;                      -- MIDI Decoder done
    notes  : out std_logic_vector(19 downto 0);  -- output to solenoid
drivers
    m_en   : in  std_logic);                     -- Enable for the MIDI
Decoder
end miditable1;

architecture miditbl of miditable1 is

begin  -- miditbl
  process (m_en, reset, clk)
  begin  -- process
    -- Reset
    if rising_edge(clk) then
      if (reset = '0') then
        m_done                              <= '0';
        notes                               <= (others => '0');
        -- if enabled
      elsif (m_en = '1') then
        -- Check to see if the first byte is a Note On
        -- The Roland PC-200E mk II we are using uses a Note On to
        -- turn a note on, and a Note On with a velocity of 0 to turn off the
        -- note.
        if (in1(23 downto 16) = x"90") then
          case in1(15 downto 8) is      -- Check to see if the note a target
note
            when "00011100" => if (in1(7 downto 0) /= x"00") then
                                 notes     <= (others => '0');
                                 notes(0)  <= '1';
                                 m_done    <= '1';
                               else
                                 notes     <= (others => '0');
                               end if;
            when "00011101" => if (in1(7 downto 0) /= x"00") then
                                 notes     <= (others => '0');
                                 notes(1)  <= '1';
                                 m_done    <= '1';
                               else
                                 notes     <= (others => '0');
                               end if;
            when "00011110" => if (in1(7 downto 0) /= x"00") then
                                 notes     <= (others => '0');
                                 notes(2)  <= '1';
                                 m_done    <= '1';
                               else
```

```
                    notes     <= (others => '0');
                  end if;
when "00011111" => if (in1(7 downto 0) /= x"00") then
                    notes     <= (others => '0');
                    notes(3)  <= '1';
                    m_done    <= '1';
                  else
                    notes     <= (others => '0');
                  end if;
when "00100000" => if (in1(7 downto 0) /= x"00") then
                    notes     <= (others => '0');
                    notes(4)  <= '1';
                    m_done    <= '1';
                  else
                    notes     <= (others => '0');
                  end if;
when "00100001" => if (in1(7 downto 0) /= x"00") then
                    notes     <= (others => '0');
                    notes(5)  <= '1';
                    m_done    <= '1';
                  else
                    notes     <= (others => '0');
                  end if;
when "00100010" => if (in1(7 downto 0) /= x"00") then
                    notes     <= (others => '0');
                    notes(6)  <= '1';
                    m_done    <= '1';
                  else
                    notes     <= (others => '0');
                  end if;
when "00100011" => if (in1(7 downto 0) /= x"00") then
                    notes     <= (others => '0');
                    notes(7)  <= '1';
                    m_done    <= '1';
                  else
                    notes     <= (others => '0');
                  end if;
when "00100100" => if (in1(7 downto 0) /= x"00") then
                    notes     <= (others => '0');
                    notes(8)  <= '1';
                    m_done    <= '1';
                  else
                    notes     <= (others => '0');
                  end if;
when "00100101" => if (in1(7 downto 0) /= x"00") then
                    notes     <= (others => '0');
                    notes(9)  <= '1';
                    m_done    <= '1';
                  else
                    notes     <= (others => '0');
                  end if;
when "00100110" => if (in1(7 downto 0) /= x"00") then
                    notes     <= (others => '0');
                    notes(10) <= '1';
                    m_done    <= '1';
                  else
                    notes     <= (others => '0');
```

```
                    end if;
when "00100111" => if (in1(7 downto 0) /= x"00") then
                      notes     <= (others => '0');
                      notes(11) <= '1';
                      m_done    <= '1';
                    else
                      notes     <= (others => '0');
                    end if;
when "00101000" => if (in1(7 downto 0) /= x"00") then
                      notes     <= (others => '0');
                      notes(12) <= '1';
                      m_done    <= '1';
                    else
                      notes     <= (others => '0');
                    end if;
when "00101001" => if (in1(7 downto 0) /= x"00") then
                      notes     <= (others => '0');
                      notes(13) <= '1';
                      m_done    <= '1';
                    else
                      notes     <= (others => '0');
                    end if;
when "00101010" => if (in1(7 downto 0) /= x"00") then
                      notes     <= (others => '0');
                      notes(14) <= '1';
                      m_done    <= '1';
                    else
                      notes     <= (others => '0');
                    end if;
when "00101011" => if (in1(7 downto 0) /= x"00") then
                      notes     <= (others => '0');
                      notes(15) <= '1';
                      m_done    <= '1';
                    else
                      notes     <= (others => '0');
                    end if;
when "00101100" => if (in1(7 downto 0) /= x"00") then
                      notes     <= (others => '0');
                      notes(16) <= '1';
                      m_done    <= '1';
                    else
                      notes     <= (others => '0');
                    end if;
when "00101101" => if (in1(7 downto 0) /= x"00") then
                      notes     <= (others => '0');
                      notes(17) <= '1';
                      m_done    <= '1';
                    else
                      notes     <= (others => '0');
                    end if;
when "00101110" => if (in1(7 downto 0) /= x"00") then
                      notes     <= (others => '0');
                      notes(18) <= '1';
                      m_done    <= '1';
                    else
                      notes     <= (others => '0');
                    end if;
```

```
            when "00101111" => if (in1(7 downto 0) /= x"00") then
                                 notes     <= (others => '0');
                                 notes(19) <= '1';
                                 m_done    <= '1';
                               else
                                 notes     <= (others => '0');
                               end if;
            when others => notes <= (others => '0');
                           m_done <= '1';
          end case;
        end if;
      else
        m_done <= '0';
      end if;
    end if;

  end process;

end miditbl;
```

MIDI_Interpreter.vhd

This is the main core of the MIDI Interpreter. It takes information in from the UART and passes it to the decoder to determine what is to be done with the input.

```
-- Filename          : MIDI_Interpreter.vhd
-- Modelname         : MIDI Interpreter
-- Title             : Top Level Module for MIDI Interpreter
-- Purpose           :
-- Author(s)         : Matt Brown
-- Comment           :
-- Assumptions       :
-- Limitations       :
-- Known errors      :
-- Specification ref :
--------------------------------------------------------------------------------
--
-- Modification history:
--------------------------------------------------------------------------------
--
-- Version  | Author | Date       | Changes made
--------------------------------------------------------------------------------
--
-- 1.0      |   MB   | 15.11.2007 | inital version
--------------------------------------------------------------------------------
--
-- 1.3      |   MB   | 07.02.2008 | initial complete version
--------------------------------------------------------------------------------
--
-- 1.6      |   MB   | 13.02.2008 | Successful test and revised comments
--------------------------------------------------------------------------------
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
library work;
use work.all;

-- MIDI Interpreter Main Module
entity MIDI_Interpreter is
  port (
    SysClk  : in  std_logic;            -- System Clock
    Reset   : in  std_logic;            -- Reset input
    RxD     : in  std_logic;            -- Serial In Data
    IntRx_N : out std_logic;            -- Receive interrupt
    notes   : out std_logic_vector(19 downto 0));  -- Output to Sol and
Steppers
end MIDI_Interpreter;

architecture BEHAV of MIDI_Interpreter is

  signal   byte_me   : unsigned(1 downto 0);          -- Byte counter
  constant byte_plus : unsigned(1 downto 0) := "01";  -- Counter Increment

  ---------------------------------------------------------------------------
--
  -- Declare Components
  ---------------------------------------------------------------------------
--

  -- UART Module used for formatting the serial in data
  component miniUART
    port (
      SysClk   : in  std_logic;
      Reset    : in  std_logic;
      CS_N     : in  std_logic;
      RD_N     : in  std_logic;
      WR_N     : in  std_logic;
      RxD      : in  std_logic;
      TxD      : out std_logic;
      IntRx_N  : out std_logic;
      IntTx_N  : out std_logic;
      DRdy_out : out std_logic;
      Addr     : in  std_logic_vector(1 downto 0);
      DataIn   : in  std_logic_vector(7 downto 0);
      DataOut  : out std_logic_vector(7 downto 0));
  end component;

  -- MIDI Interpreter Table
  component MIDITABLE1
    port(IN1    : in  std_logic_vector (23 downto 0);
         clk    : in  std_logic;
         Reset  : in  std_logic;
         M_DONE : out std_logic;
         NOTES  : out std_logic_vector (19 downto 0);
         M_EN   : in  std_logic);
  end component;

  -- Signals for UART
  signal CS_N     : std_logic;
  signal RD_N     : std_logic;
  signal WR_N     : std_logic;
```

```
  signal TxD      : std_logic;
  signal IntTx_N  : std_logic;
  signal Addr     : std_logic_vector(1 downto 0);
  signal DataIn   : std_logic_vector(7 downto 0);
  signal DataOut  : std_logic_vector(7 downto 0);
  signal W_IN1    : std_logic_vector (23 downto 0);
  signal m_done   : std_logic;          -- MIDI Decoder Done
  signal MIDI_EN  : std_logic;          -- Enable for MIDI Decoder
  signal DRdy_out : std_logic;
  signal flag     : std_logic;

begin  -- BEHAV

  ---------------------------------------------------------------------------
--
  -- Instantiate Components
  ---------------------------------------------------------------------------
--
  UART : miniUART
    port map (
      SysClk   => SysClk,
      Reset    => Reset,
      CS_N     => CS_N,
      RD_N     => RD_N,
      WR_N     => WR_N,
      RxD      => RxD,
      TxD      => TxD,
      IntRx_N  => IntRx_N,
      IntTx_N  => IntTx_N,
      DRdy_out => DRdy_out,
      Addr     => Addr,
      DataIn   => DataIn,
      DataOut  => DataOut);

  MIDI : MIDITABLE1
    port map (
      IN1    => W_IN1,
      clk    => SysClk,
      Reset  => Reset,
      M_DONE => M_DONE,
      NOTES  => NOTES,
      M_EN   => MIDI_EN);

  process (Reset, SysClk)
  begin  -- process

    if rising_edge(SysClk) then
      -- Reset
      if (Reset = '0') then
        W_IN1                 <= (others => 'Z');
        MIDI_EN               <= '0';
        CS_N                  <= '1';
        RD_N                  <= '1';
        byte_me               <= "00";
      else
        CS_N                  <= '0';
        RD_N                  <= '0';
```

```
        Addr                    <= "00";
      end if;
      -- Read from serial data
      if (DRdy_out = '1') then
        if (DataOut(7) = '1') then
          -- Counter reset, interpretation disabled, set first
          -- byte to input bit string, increment counter
          byte_me               <= "00";
          MIDI_EN               <= '0';
          W_IN1                 <= (others => '0');
          W_IN1(23 downto 16)   <= DataOut(7 downto 0);
          byte_me               <= "01";
        elsif (m_done = '0') then
          if (byte_me = "01") then
            -- set second byte to input bit string
            W_IN1(15 downto 8) <= DataOut(7 downto 0);
            byte_me            <= byte_me + byte_plus;
            MIDI_EN            <= '0';
          elsif (byte_me = "10") then
            -- set third byte to input bit string and enable
            -- interpretation
            W_IN1(7 downto 0)  <= DataOut(7 downto 0);
            MIDI_EN            <= '1';
            byte_me            <= byte_me + byte_plus;
          end if;
        end if;
      else

      end if;
    end if;
  end process;

end BEHAV;
```

miditbl_TB.vhd

This is the test bench for the MIDI Interpreter, including the UART. The UART was also tested separately to ensure it would work as intended.

```
--
================================================================================-
-
-- Design units   : TestBench for MIDI Interpreter.
--
-- File name      : miditbl_TB.vhd
--
-- Purpose        : Implements the test bench for miniUART device.
--
-- Library        : uart_Lib.vhd
--
-- Dependencies   : IEEE.Std_Logic_1164
--
--------------------------------------------------------------------------------
---
--------------------------------------------------------------------------------
---
```

```
-- Revision list
-- Version   Author             Date          Changes
--
-- 0.1      Matt Brown        2/6/2008        New model
--------------------------------------------------------------------------------
---
--------------------------------------------------------------------------------
--
-- Clock generator
--------------------------------------------------------------------------------
--
library IEEE,work;
use IEEE.Std_Logic_1164.all;
--
entity ClkGen is
   port (
      Clk     : out Std_Logic);   -- Oscillator clock
end ClkGen;--==================== End of entity
==============================--
--------------------------------------------------------------------------------
---
-- Architecture for clock and reset signals generator
--------------------------------------------------------------------------------
---
architecture Behaviour of ClkGen is
begin --========================== Architecture
==============================--
  --------------------------------------------------------------------------------
---
  -- Provide the system clock signal
  --------------------------------------------------------------------------------
---
  ClkDriver : process
    variable clktmp : Std_Logic := '1';
    variable tpw_CI_posedge : Time := 10 ns; -- ~50 MHz
  begin
     Clk <= clktmp;
     clktmp := not clktmp;
    wait for tpw_CI_posedge;
  end process;
end Behaviour; --=================== End of architecure
======================--


--------------------------------------------------------------------------------
--
-- LoopBack Device
--------------------------------------------------------------------------------
--
library IEEE,work;
use IEEE.Std_Logic_1164.all;
--
entity LoopBack is
   port (
      Clk     : in  Std_Logic;   -- Oscillator clock
      RxWr    : in  Std_Logic;   -- Rx line
      TxWr    : out Std_Logic);  -- Tx line
```

```
end LoopBack; --==================== End of entity
==========================--
-------------------------------------------------------------------------------
---
-- Architecture for clock and reset signals generator
-------------------------------------------------------------------------------
---
architecture Behaviour of LoopBack is
begin --========================== Architecture
============================--
  -------------------------------------------------------------------------------
---
  -- Provide the external clock signal
  -------------------------------------------------------------------------------
---
  ClkTrig : process(Clk)
  begin
      TxWr <= RxWr;
  end process;
end Behaviour; --=================== End of architecure
====================--


-------------------------------------------------------------------------------
---
-- Testbench for UART device
-------------------------------------------------------------------------------
---
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.Uart_Def.all;

entity miditbl_tb is
end miditbl_tb;

architecture stimulus of miditbl_tb is
  ---------------------------------------------------------------------
  -- Signals
  ---------------------------------------------------------------------
  signal Reset   : Std_Logic;  -- Synchro signal
  signal Clk     : Std_Logic;  -- Clock signal
  signal DataIn  : Std_Logic_Vector(7 downto 0);
  signal DataOut : Std_Logic_Vector(7 downto 0);
  signal RxD     : Std_Logic;  -- RS-232 data input
  signal TxD     : Std_Logic;  -- RS-232 data output
  signal CS_N    : Std_Logic;
  signal RD_N    : Std_Logic;
  signal WR_N    : Std_Logic;
  signal IntRx_N : Std_Logic;  -- Receive interrupt
  signal IntRx_N_M : Std_Logic;
  signal IntTx_N : Std_Logic;  -- Transmit interrupt
  signal Addr    : Std_Logic_Vector(1 downto 0);
  signal notes : Std_Logic_Vector(19 downto 0);
  signal tb_test_name_v     : string(1 to 15) := "  Initialize   ";
```

```
  ------------------------------------------------------------------------
--
  -- MIDI Interpreter
  ------------------------------------------------------------------------
--
  component MIDI_Interpreter
    port (
      SysClk  : in  std_logic;                      -- System Clock
      Reset   : in  std_logic;                      -- Reset input
      RxD     : in  Std_Logic;                      -- Serial Input
      IntRx_N : out Std_Logic;                      -- Rx Interrupt
      notes   : out Std_Logic_Vector(19 downto 0));  -- Output to solenoids
  end component;
  ------------------------------------------------------------------
  -- Clock Divider
  ------------------------------------------------------------------
  component ClkGen is
   port (
      Clk     : out Std_Logic);   -- Oscillator clock
  end component;
  ------------------------------------------------------------------
  -- LoopBack Device
  ------------------------------------------------------------------
  component LoopBack is
   port (
      Clk     : in  Std_Logic;   -- Oscillator clock
      RxWr    : in  Std_Logic;   -- Rx line
      TxWr    : out Std_Logic);  -- Tx line
  end component;
  ------------------------------------------------------------------
  -- UART Device
  ------------------------------------------------------------------
  component miniUART is
  port (
     SysClk   : in  Std_Logic;  -- System Clock
     Reset    : in  Std_Logic;  -- Reset input
     CS_N     : in  Std_Logic;
     RD_N     : in  Std_Logic;
     WR_N     : in  Std_Logic;
     RxD      : in  Std_Logic;
     TxD      : out Std_Logic;
     IntRx_N  : out Std_Logic;  -- Receive interrupt
     IntTx_N  : out Std_Logic;  -- Transmit interrupt
     Addr     : in  Std_Logic_Vector(1 downto 0); --
     DataIn   : in  Std_Logic_Vector(7 downto 0); --
     DataOut  : out Std_Logic_Vector(7 downto 0)); --
  end component;
begin --======================== Architecture ========================--
  ------------------------------------------------------------------------
  -- Instantiation of components
  ------------------------------------------------------------------------
  Clock       : ClkGen port map (Clk);
  LoopDev     : LoopBack port map (Clk,TxD,RxD);
  miniUARTDev : miniUART port map (Clk,Reset,CS_N,RD_N,WR_N,RxD,TxD,
                                   IntRx_N,IntTx_N,Addr,DataIn,DataOut);
  DUT : MIDI_Interpreter
    port map (
```

```
      SysClk  => Clk,
      reset   => reset,
      RxD     => RxD,
      IntRx_N => IntRx_N_M,
      notes   => notes);
  ------------------------------------------------------------------------
  -- Reset cycle
  ------------------------------------------------------------------------
  RstCyc : process
  begin
     Reset <= '1';
     wait for 5 ns;
     Reset <= '0';
     wait for 250 ns;
     Reset <= '1';
     wait;
  end process;
  ------------------------------------------------------------------------
  -- Main
  ------------------------------------------------------------------------
  -- Modified from the UART Test Bench, as the UART is being used as a
  -- loopback to take in a byte and feed serial data into the internal
  -- UART of the MIDI Interpreter
  ------------------------------------------------------------------------
  ProcCyc : process(Clk,IntRx_N,IntTx_N,Reset)
      variable counter : unsigned(3 downto 0);
      constant cone : unsigned(3 downto 0):= "0001";
      variable temp : bit := '0';

      variable l_test_name_v : string(1 to 15);  -- local testname variable
      variable l_test_index_v : integer := 0;  -- local test index

      -- purpose: assign test name variable and signal
      procedure p_test_name (
        test_name : in string) is
      begin  -- p_test_name
        l_test_name_v := "               ";
        if (test_name'length < 15) then
          l_test_name_v  := test_name(1 to test_name'length) &
l_test_name_v((test_name'length + 1) to 15);
        else
          l_test_name_v := test_name(1 to 15);
        end if;
        tb_test_name_v <= l_test_name_v;
        report "Started " & l_test_name_v(1 to 15) & " test." severity note;
      end p_test_name;

  begin
     if Rising_Edge(Reset) then
        counter := "0000";
          WR_N <= '1';
          RD_N <= '1';
          CS_N <= '1';
     elsif Rising_Edge(Clk) then
        if IntTx_N = '0' then
           if temp = '0' then
              temp := '1';
```

```
                case counter is
                   when "0000" =>
                        p_test_name("Lowest Note");
                        Addr <= "00";
                        DataIn <= x"90";
                        WR_N <= '0';
                        CS_N <= '0';
                        counter := counter + cone;
                   when "0001" =>
                        Addr <= "00";
                        DataIn <= x"1C";
                        WR_N <= '0';
                        CS_N <= '0';
                        counter := counter + cone;
                   when "0010" =>
                        Addr <= "00";
                        DataIn <= x"01";
                        WR_N <= '0';
                        CS_N <= '0';
                        counter := counter + cone;
                   when "0011" =>
                        p_test_name("Highest Note");
                        Addr <= "00";
                        DataIn <= x"90";
                        WR_N <= '0';
                        CS_N <= '0';
                        counter := counter + cone;
                   when "0100" =>
                        Addr <= "00";
                        DataIn <= x"2F";
                        WR_N <= '0';
                        CS_N <= '0';
                        counter := counter + cone;
                   when "0101" =>
                        Addr <= "00";
                        DataIn <= x"01";
                        WR_N <= '0';
                        CS_N <= '0';
                        counter := "0000";
                   when others => null;
                end case;
             elsif temp = '1' then
                temp := '0';
             end if;
          elsif IntRx_N = '0' then
             Addr <= "00";
             RD_N <= '0';
             CS_N <= '0';
          else
            RD_N <= '1';
            CS_N <= '1';
            WR_N <= '1';
            DataIn <= "ZZZZZZZZ";
          end if;
       end if;
    end process;
end stimulus; --================== End of TestBench ==================--
```

Stepper Motor Control

These files are the VHDL files used to create the core that controls the stepper motors, and test that the design will work as planned.

stepper_drive.vhd

This is the control unit for the stepper motors used to pluck the strings. See Motor Control for information and descriptions of how the control block was designed.

```
--------------------------------------------------------------------------------
--
-- Stepper Motor Controller
--------------------------------------------------------------------------------
--
-- Controls the stepper motors via L6208 stepper motor drivers based on input
-- from the MIDI Interpretation and Chord Detection* modules.
--     *Chord Detection control of motors will be added if the algorithm is
--      finished before the end of the project
--------------------------------------------------------------------------------
--
-- filename: stepper_drive.vhd
--------------------------------------------------------------------------------
--
-- Version | Author |    Date    | Changes
--------------------------------------------------------------------------------
--
--   1.2   |   MB   | 14.02.2008 | Initial
--------------------------------------------------------------------------------
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity step is

  port (
    notes         : in    std_logic_vector(19 downto 0);  -- the notes of
interest
    clk           : in    std_logic;    -- Clock
    Reset         : in    std_logic;    -- Reset
    HALF_FULL     : out   std_logic;
    CW_CCW        : out   std_logic;
    RESET_DRIVE_N : out   std_logic;
    EN_DRIVE      : out   std_logic_vector(3 downto 0);
    DRIVE_CLK     : inout std_logic);   -- Motor Out

end step;

architecture behav of step is

  signal   flag_reset    : std_logic            := '0';
  signal   reset_done    : std_logic;
  signal   flag_step     : std_logic            := '0';
```

```
  signal   notes_place   : std_logic_vector(19 downto 0);
  -- Counter for stepping cycles
  signal   step_count    : unsigned(8 downto 0) := "000000000";
  -- Counter for dividing clock by 600
  -- Divides by 300 to create 50/50 clock at 83 kHz
  signal   clock_div_300 : unsigned(8 downto 0) := "000000000";
  constant count_up      : unsigned(8 downto 0) := "000000001";
  constant clk_div_up    : unsigned(8 downto 0) := "000000001";

begin  -- behav

  CW_CCW <= '1';                        --Motors go clockwise

  Pluck_string : process (reset, clk)
  begin  -- process Pluck_string
    if (reset = '0') then
      HALF_FULL          <= '1';                -- Hi means Half-step the
motor
      RESET_DRIVE_N      <= '1';                -- Reset is active low
      EN_DRIVE           <= "0000";             -- Enable is active High
      DRIVE_CLK          <= '0';                -- Drive pulse starts low
      flag_reset         <= '0';
      flag_step          <= '0';
      clock_div_300      <= "000000000";
      step_count         <= "000000000";
      --if (flag_reset = '0') then
      RESET_DRIVE_N      <= '0';
      --HALF_FULL     <= (others => '1');
      EN_DRIVE           <= (others => '1');
      --end if;
    elsif (rising_edge(clk)) then
      if (reset = '1' and clock_div_300 = "100101100") then
        EN_DRIVE         <= "0000";
        flag_reset       <= '0';
        RESET_DRIVE_N    <= '1';
        DRIVE_CLK        <= '1';
        reset_done       <= '1';
        HALF_FULL        <= '0';
        clock_div_300    <= "000000000";
        if (notes /= notes_place) then
          flag_step      <= '0';
          notes_place    <= notes;
        end if;
        if (notes < "00000000000000100000" and notes /=
"00000000000000000000") then
          EN_DRIVE       <= (others => '0');
          EN_DRIVE(0)    <= '1';
          --EN_DRIVE(0)    <= '1';
          if (flag_step = '0') then
            DRIVE_CLK    <= not DRIVE_CLK;
            --HALF_FULL(0) <= '0';
            if (step_count = "110010000") then  -- if we've had 200 steps
              step_count <= "000000000";
              flag_step  <= '1';
            else
              step_count <= step_count + count_up;
            end if;
```

```
          end if;
        elsif (notes < "00000000010000000000" and notes >=
"00000000000000100000") then
          EN_DRIVE       <= (others => '0');
          EN_DRIVE(1)    <= '1';
          --EN_DRIVE(0)    <= '1';
          if (flag_step = '0') then
            DRIVE_CLK    <= not DRIVE_CLK;
            --HALF_FULL(1) <= '0';
            if (step_count = "110010000") then  -- if we've had 200 steps
              step_count <= "000000000";
              flag_step  <= '1';
            else
              step_count <= step_count + count_up;
            end if;
          end if;
        elsif (notes < "00001000000000000000" and notes >=
"00000000010000000000") then
          EN_DRIVE       <= (others => '0');
          EN_DRIVE(2)    <= '1';
          --EN_DRIVE(0)    <= '1';
          if (flag_step = '0') then
            DRIVE_CLK    <= not DRIVE_CLK;
            --HALF_FULL(2) <= '0';
            if (step_count = "110010000") then  -- if we've had 200 steps
              step_count <= "000000000";
              flag_step  <= '1';
            else
              step_count <= step_count + count_up;
            end if;
          end if;
        elsif (notes     <= "10000000000000000000" and notes >=
"00001000000000000000") then
          EN_DRIVE       <= (others => '0');
          EN_DRIVE(3)    <= '1';
          --EN_DRIVE(0)    <= '1';
          if (flag_step = '0') then
            DRIVE_CLK    <= not DRIVE_CLK;
            --HALF_FULL(3) <= '0';
            if (step_count = "110010000") then  -- if we've had 200 steps
              step_count <= "000000000";
              flag_step  <= '1';
            else
              step_count <= step_count + count_up;
            end if;
          end if;
        elsif (flag_step = '1') then
          EN_DRIVE       <= (others => '0');
        end if;
      else
        clock_div_300    <= clock_div_300 + clk_div_up;
      end if;
    end if;
  end process Pluck_string;

end behav;
```

stepper_drive_tb.vhd

This is the test bench for the stepper motor control unit. This was used to make sure that there were no logic errors in the stepper motor control block, and to test different signal situations to ensure safe operation.

```
--------------------------------------------------------------------------------
--
-- Stepper Motor Testbench
--------------------------------------------------------------------------------
--
-- Test Bench for the stepper motor driver module.
--     Set up for testing the MIDI controlled module only
--------------------------------------------------------------------------------
--
-- filename: stepper_drive_tb.vhd
--------------------------------------------------------------------------------
--
-- Version | Author |    Date    | Changes
--------------------------------------------------------------------------------
--
--   1.2   |   MB   | 21.02.2008 | Initial
--------------------------------------------------------------------------------
--
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.all;

entity step_tb is
end step_tb;

architecture TB of step_tb is
--------------------------------------------------------------------------------
--
-- Component Declaration
--------------------------------------------------------------------------------
--
  component step
    port (
      notes       : in  std_logic_vector(19 downto 0);
      clk         : in  std_logic;
      Reset       : in  std_logic;
      HALF_FULL   : out std_logic;
      CW_CCW      : out std_logic;
      RESET_DRIVE_N : out std_logic;
      EN_DRIVE    : out std_logic_vector(3 downto 0);
      DRIVE_CLK   : inout std_logic);
  end component;

--------------------------------------------------------------------------------
--
-- Signal declarations
--------------------------------------------------------------------------------
--
  signal tb_test_name_v     : string(1 to 15) := "  Initialize   ";
```

```
  signal notes : std_logic_vector(19 downto 0);
  signal clk : std_logic := '0';
  signal reset : std_logic := '1';
  signal HALF_FULL : std_logic;
  signal CW_CCW : std_logic;
  signal RESET_DRIVE_N : std_logic;
  signal EN_DRIVE : std_logic_vector(3 downto 0);
  signal DRIVE_CLK : std_logic;
-------------------------------------------------------------------------------
--

begin  -- TB

  clk <= not clk after 10 ns;

  tested : step
    port map (
      notes       => notes,
      clk         => clk,
      Reset       => Reset,
      HALF_FULL   => HALF_FULL,
      CW_CCW      => CW_CCW,
      RESET_DRIVE_N => RESET_DRIVE_N,
      EN_DRIVE    => EN_DRIVE,
      DRIVE_CLK   => DRIVE_CLK);

  -------------------------------------------------------------------------------
--
  -- purpose: Top level control
  -- outputs:
  -------------------------------------------------------------------------------
--
  top_control : process

    variable l_test_name_v : string(1 to 15);  -- local testname variable
    variable l_test_index_v : integer := 0;  -- local test index

    -- purpose: assign test name variable and signal
    procedure p_test_name (
      test_name : in string) is
    begin  -- p_test_name
      l_test_name_v := "               ";
      if (test_name'length < 15) then
        l_test_name_v  := test_name(1 to test_name'length) &
l_test_name_v((test_name'length + 1) to 15);
      else
        l_test_name_v := test_name(1 to 15);
      end if;
        tb_test_name_v <= l_test_name_v;
      report "Started " & l_test_name_v(1 to 15) & " test." severity note;
    end p_test_name;


  begin  -- process top_control

    p_test_name("Reset");
    reset <= '0';
```

```
    notes <= "00000000000000000000";
    wait for 12 us;
    reset <= '1';
    wait for 12 us;
    ------------------------------------------------------------------------
--
    p_test_name("Low OOR");
    notes <= "00000000000000000000";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    p_test_name("Low Bin 1");
    --wait for 100 ns;
    notes <= "00000000000000000001";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    p_test_name("High Bin 1");
    notes <= "00000000000000010000";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    p_test_name("Low Bin 2");
    notes <= "00000000000000100000";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    p_test_name("High Bin 2");
    notes <= "00000000001000000000";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    p_test_name("Low Bin 3");
    notes <= "00000000010000000000";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    p_test_name("High Bin 3");
    notes <= "00000100000000000000";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    p_test_name("Low Bin 4");
    notes <= "00001000000000000000";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    p_test_name("High Bin 4");
    notes <= "10000000000000000000";
    wait for 1300 us;
    ------------------------------------------------------------------------
--
    wait for 100 ns;
    report "S i m u l a t i o n   C o m p l e t e d !!" severity failure;

  end process top_control;
```

```
end TB;
```

Appendix – Mechanical Drawings

The rest of this page intentionally left blank.