



**18551: Digital Communication and Signal  
Processing Design  
Spring 2001**



**Computer Generated  
Melodies**



**Final Report – May 7, 2001  
Group 7**

**Alexander Garmew (agarmew)  
Per Lofgren (pl19)  
José Morales (jmorales)  
James Mourra (jmourra)**

## **TABLE OF CONTENTS**

### 1. Introduction

- The Problem
- Solution Overview
- Prior 18-551 Work
- Alternative Solutions

### 2. What We Did

- Signal Flow Chart / Outline
- Note Detection Algorithm
- Pitch Shifting / Filtering Algorithms

### 3. Results

- Final Timing Data & Optimizations
- EVM Memory Usage
- Purchases Required

### 4. Summary

### 5. References / Web Links

## INTRODUCTION

### **The Problem:**

As with anything, there is an elemental basis for the composition of music. The material essence of music lies with its melody, harmony, rhythm, and dynamics. Melody gives music soul, while rhythm blends the expression of harmony and dynamics with the tempo of the passage. All are necessary to create a recognizable pattern known as a "song." (1)

Melody is structured by its length and intensity much like sentences in a spoken language. For instance, a phrase in music is a unit of meaning within the larger structure of the song in its entirety. Other examples include the cadence and the climax. A cadence is a final ending to a musical section. A climax is a high point of intensity. (1)

Harmony is the relation of notes to notes and chords to chords as they are played simultaneously. Harmonic "patterns" are established from notes and chords in successive order. Melodic intervals are those that are linear and occur in sequence, while harmonic intervals are sounded at the same time. Whether or not a harmony is pleasing is a matter of personal taste, as there are consonant and dissonant harmonies, both of which are pleasing to the ears of some and not others. (1)

Chords have meaning as they lead to other chords. Certain progressions are encouraged as acceptable in certain styles of music. But basic to all harmony, regardless of style, is the triad. A triad is the most common chord form. It is built on the first, third, and fifth notes of the scale (do, mi, and so) and is symbolized in musical notation by the Roman numeral I. A triad built on the second note of the scale would include the second, fourth, and sixth notes of the scale, still keeping one scale degree between each jump. A triad built on the second note of the scale is written as II. Triad chords may be built on all seven notes of the scale (with the eighth note a

repeat of the first). Chord symbols for the triads built on the third through seventh notes of the scale are as follows: III, IV, V, VI, and VII. The I chord is named the Tonic, and the IV chord is named Sub Dominant. The V chord is the Dominant. The VII chord is referred to as the Leading Tone, as it is often used to change (or "lead") into a new key. This organization around tones is known as "tonality." (1)

In the music industry, often times there are artists out there who try to write their own songs, but unless they're musical geniuses, composing the melody and all the accompanying harmony parts can be quite time consuming especially if you don't know much about music theory. This lack of harmony problem also arises among singers who don't always have a band or the right song to sing along with.

### **Solution Overview:**

Our problem addresses the idea for computers to actually help generate music, given basic structures to work with. By taking any signal from a given instrument, the computer would strip it down to its base components and determine the fundamental structure of the music. It would then create its own music to go along with it. By adding notes above or below the given music, it would give musicians the ability to discover new sounds and textures. By inputting a musical melody from a given instrument into the EVM, we should be able to output the original signal, plus a number of harmonized pseudo-random signals. We say pseudo-random because they will not be of any frequency but "in tune" to the original input. For example, if we input a 220 Hz signal, then certain other signals will sound "consonant" or "dissonant." How consonant or dissonant it sounds is based on the ratio of the input frequency to the new signal. The lower the numbers in that ratio, then the more consonant it will sound, but if the numbers are too high

then it will sound dissonant. Dissonance is usually thought of as sounding "bad," but that is purely up to the listener to decide. So, given the 220 Hz Signal, if we output it along with a 440Hz signal, the ratio is 220/440, which can be reduced to 1/2. This is very consonant and is in fact one octave above the original frequency.

In each octave of music, there are 12 notes, consisting of A, A#, B, C, C#, D, D#, E, F, F#, G, G#. A 220 Hz tone is defined as being a note of "A." One octave above that note is also an A, which is twice the frequency of the lower A, or 440Hz. For each of the other notes in an octave, we can use the following formula to determine their frequency:  $FREQ * 2^{(n/12)}$  where n is a value from 0 to 12.

To represent each note we shall use a burst of a sinusoidal tone followed by a shorter period of silence (a pause). These pauses will allow us to distinguish between two or more separate notes in a row at the same pitch. The duration of each tone burst is determined by whether the note is a whole note, half note, quarter note, eighth note, etc. Obviously, a quarter note has twice the duration of an eighth note and so on. The short pause following each note will be of the same length regardless of the note's duration, unless we include slurs, which are changes of notes without any breaks. (2)

In determining the notes in our sampled signal and also determining when each note begins and ends we can use a process called a time/ frequency representation of the signal, or a spectrograph. This is where time is plotted on the x-axis, frequency on the y-axis, and then color is used to represent the presence or absence of frequencies in the signal during that particular time. We should be able to first do this in MATLAB by breaking the sampled signal into small time segments, of approximately 50 ms in duration, and plotting the energy present at each frequency for that time segment. (2)

Typically, when a note is played, the volume rises quickly from zero and then decays over time, depending on how hard the string is struck and how long it is sustained. The variation of the volume over time is divided into four segments: Attack, Decay, Sustain, and Release (ADSR). For a given tone, we can change the volume by multiplying a sinusoid by another function called a windowing function. A decaying exponential is the simplest way to modulate the tone volume. We shall try concatenating different functions to model ADSR. (2)

As the volume of one note is decaying, another note is played. Mathematically this can be accomplished by allowing the time regions occupied by different sinusoids to overlap. This is ultimately what we want to achieve so we would have a much smoother, less staccato sounding piece. (2)

Instruments are usually played in rooms that generate reverberations. Reverberations are reflections off of room surfaces, including walls, ceiling, floors, and obstructions. In discrete time mathematics, this is like generating echoes of the original signal, with decreasing amplitude as the delay time increases. However, a major source (e.g. a back wall), may generate higher amplitudes at specific delays. (2)

Most of the added effects of reverberation, and smooth overlapping notes are things that should be created in the accompanied harmony sounds, but due to time restraints we decided to narrow down our project to output just the basic harmonized tones along with the original input.

### **Prior 18-551 Work:**

The purpose of our project was to implement a musical harmonizer on the EVM C67 board. This program would take guitar inputs and create a harmonious and in-tune accompaniment for that input. The inspiration behind this is to aid the practicing musician. He

or she could record a line of music, perhaps a melody, and then use our program to hear that line of music with a possible accompaniment or harmony attached to it. The program would be able to generate many different kinds of new signals so the user could get ideas for compositions from it. This is the only project of its kind ever attempted in this class, but it can be approached in ways similar to other projects. For example, code for sending information to the EVM was extracted from previous lab code (6), as well as code for performing the FFT. (7), and also code for sending DMA memory transfers. (6)

### **Alternative Solutions:**

Originally we approached the problem from a "time-frequency analysis" point of view. This meant finding the fundamental frequencies and where in the time domain they occurred. There are several different methods in existence for performing this analysis, most notably STFT (Short Time Fourier Transform) and DWT (Discrete Wavelet Transform). Unfortunately we were not able to find any satisfactory C code for these methods. The spring 2000 project concerning music transcription used "pitch estimation analysis" to find the notes of the input. They filtered the signal with a combination wavelet/FIR filter and then measured the period of the signal in the time domain. While this approach seemed logical we decided to try an intuitive approach of our own creation. First, we placed constraints on the content of the input signal. Each input would be a series of "plucked guitar string events" about 3/4 of a second apart. There could be no chords or simultaneously occurring fundamental frequencies. In other words, we would be searching for one frequency at a time. We also would limit the input to a narrow band of frequencies, most likely 1 to 500 Hz or so. The first part of the analysis was to find where in time the individual notes started. In other words find the numbers of the samples in the file

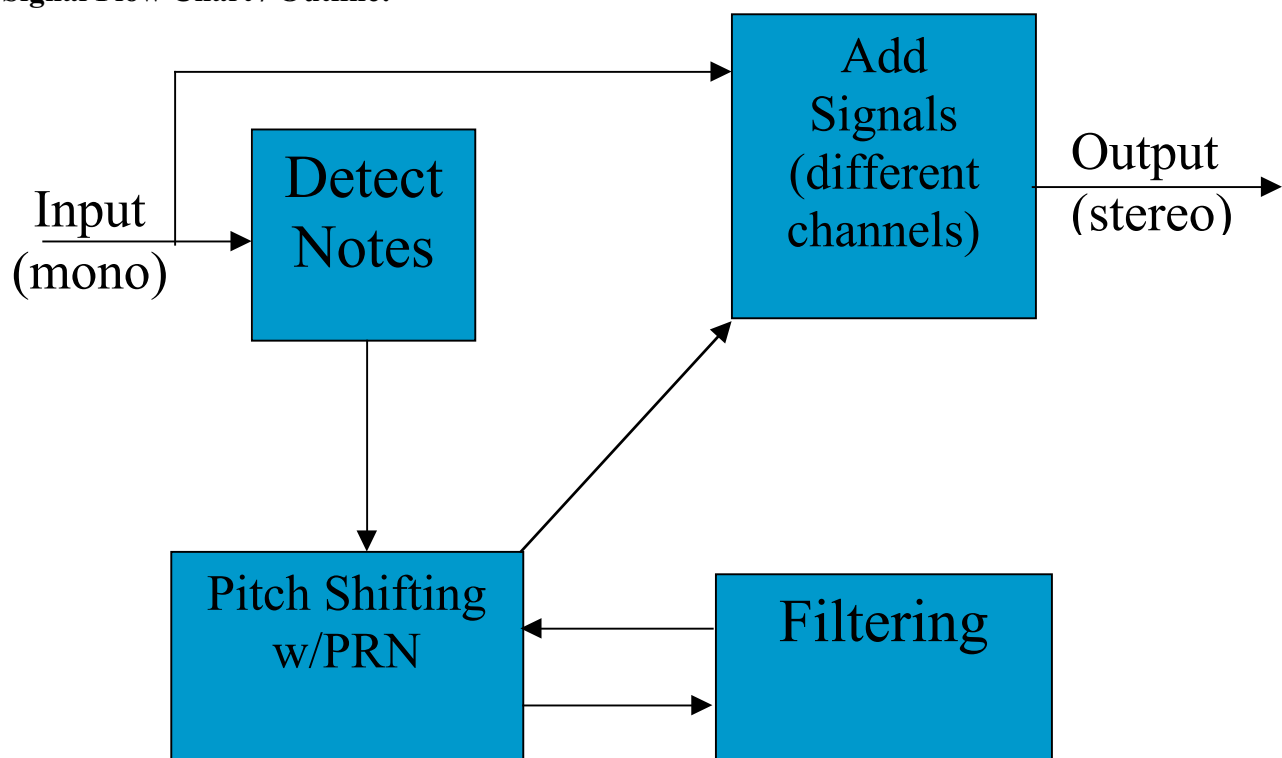
where the "attack", or onset of the note, occurs. Summing up the energies of blocks of samples could do this. For example, take the first 500 samples; add them up, save this number. Take the next 500 samples, do the same thing. Keep doing this until the end of the file. Then look at this list of sums. Where there is a big jump from one sum to the next, a new note has started. Next we would want to know the fundamental frequency of the note. Starting from the attack sample number, take an FFT of the next 1024 samples or so. This would produce a frequency content "picture" of that note with peaks at its fundamental frequencies and its harmonics, which are scaled down integer multiples of the fundamental frequency. The fundamental frequency of the note will have the largest peak, so the energy value of that FFT point will be greater than all the rest, thus making it easy to identify. Once we knew the "what" and "where" of the signal's frequency content, we could then create output signals that have frequencies that are "in-tune" with the input. We were able to make this work in MATLAB, but it was decided that it would be too difficult and inefficient to implement on the DSP. It was decided instead to use pitch shifting to create harmonious accompaniments. The decision to abandon the frequency analysis approach to output generation was based mainly on two factors. First, the difficulty in accurately determining the pitch of the individual notes and second, the occurrence of "false positives" in the attack detection algorithm. Had we continued on with this method we also would have encountered the problem of generating accompaniment signals that sounded "plausible" and "intelligent". What exactly define good and bad music and how to create it are difficult things to discern. For instance, the accompaniment signal would have to be of the same duration as the input, but where in the time domain of this new file would the output notes occur? Would they occur synchronously with the input or in between? Also, what would they sound like? Would they simply be sinusoids played through the sound card? That could scarcely be called music. It



was thus decided that we narrow the scope of our project to simply shifting the pitch of the input signal, the output would then be a "frequency shifted" version of the input. This removes the arduous and precarious task of time-frequency analysis. The question of how good these pitch shifted accompaniment sounds is a matter of personal taste. The fact that it is less error prone than time-frequency analysis makes it very attractive, but the possibilities for the variety of outputs is severely limited. Now the generated signals will have many of the same characteristics as the input. What our pitch shifting essentially does is create chords out of single notes, as opposed to playing a harmony to an inputted melody, so maybe the output is not different enough from the input to lend true insight to the music composition process.

## WHAT WE DID

### Signal Flow Chart / Outline:



As an input signal to the EVM we send a mono .wav file of a guitar melody. Any sampling rate can be used, but the higher the better, which would also take longer to process. Next the signal is then processed to determine when and where in the signal each new note begins. Then each note of the waveform is then pitch shifted by one of five pseudo-randomly chosen scales. The filtering process includes FFT, cropping of excess interpolated zeros, and then IFFT. The output is then written to a stereo .wav file with the shifted signal of the same length of the original signal on one channel, and then the original signal is on the other channel.

### **Note Detection Algorithm:**

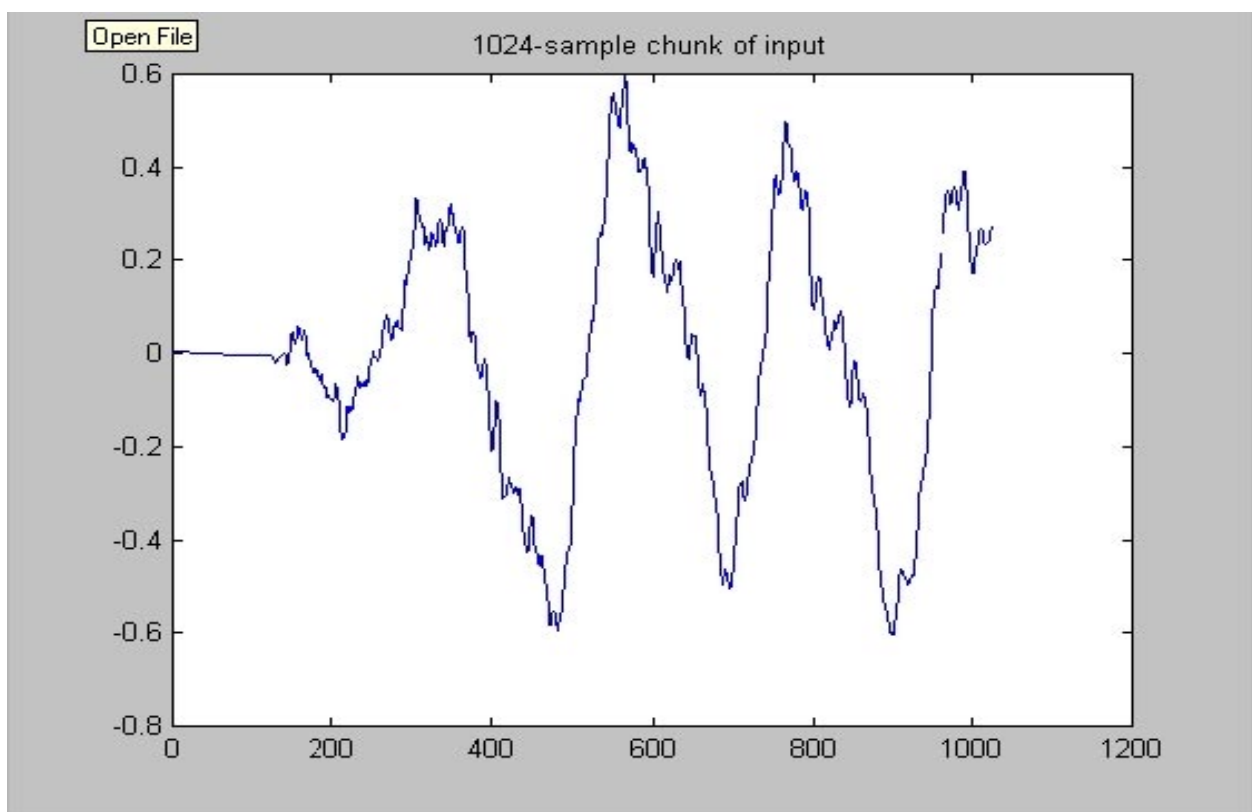
To determine when where in the input sample to start pitch shifting, it was important to first find where each new note begins. To do this, the input waveform signal was divided up into 1024 sample chunks, and then the average of the samples of each chunk was calculated. Then using all the averages you then look for when there's a huge positive jump from one chunk to the next. In our case we used normalized waveforms and determined a difference of greater than 1.5 the previous chunks value, then that's where a new note would begin. This algorithm also works on the assumption that notes will be separated by at least 1024 samples.

### **Pitch Shifting / Filtering Algorithms:**

We create the pseudo-random computer-generated music as follows. First, we get the input .wav file, which contains a string of notes. For the purposes of the lab demo, we chose to use a recording of an acoustic guitar. These recordings consisted of 4-10 notes played individually (i.e., no chords, or overlapping notes), Polyphony, or more than one note being played simultaneously, is beyond the scope of the project.

Taking the input sound file, we first do peak detection, to determine where notes begin and end. We then randomly select a ratio by which to shift each note. We chose to have 5 different ratios of shifting, consisting of the following: Perfect Fifth:  $2/3$ , Perfect Fourth:  $3/4$ , Major Third:  $4/5$ , Major Sixth:  $3/5$ , and Minor Third:  $5/6$ . (8)

These ratios can be thought of in two ways: 1) The ratio of the frequencies of the input to the output, or 2) the ratio of the number of samples that we downsample to from the input to the output. For example, The following shows how we shift a given note from its original frequency up a Fifth. First, we cut up the note, which we shift into useable chunks. We chose 1024 samples per "chunk," as shown in Figure 1 below.

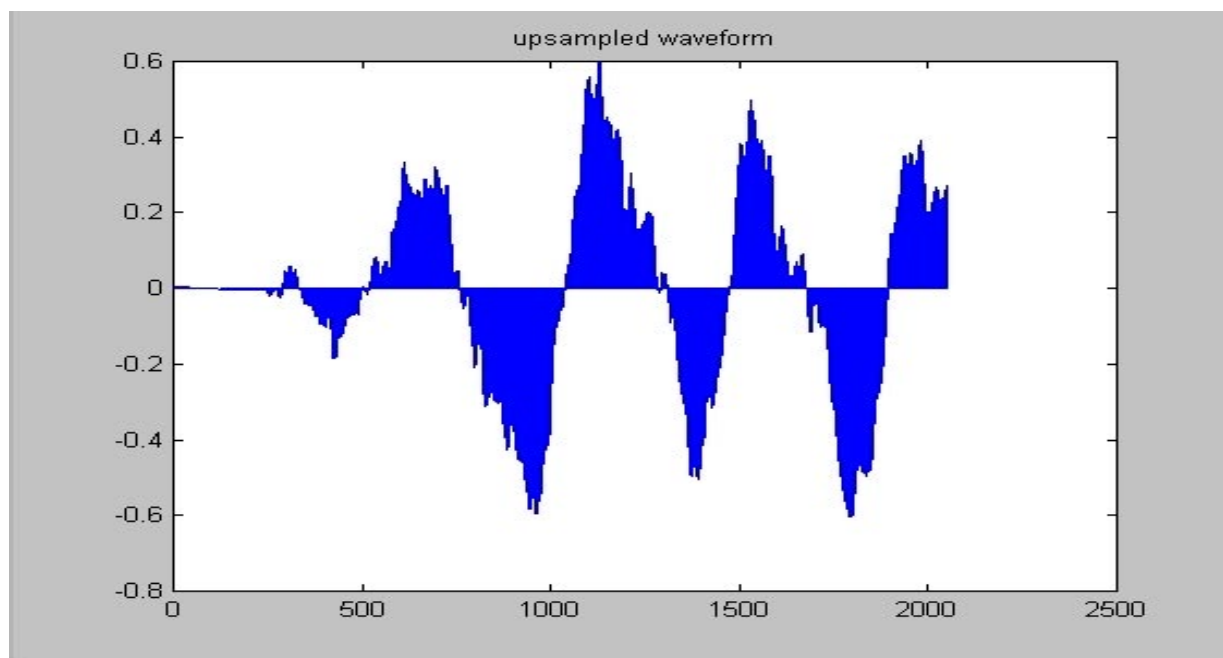


**Figure 1**

The reason we chose this number is that the frequency range on a guitar is approximately 80-1500Hz. The maximum frequency reproducible by a 22050Hz input signal is approximately

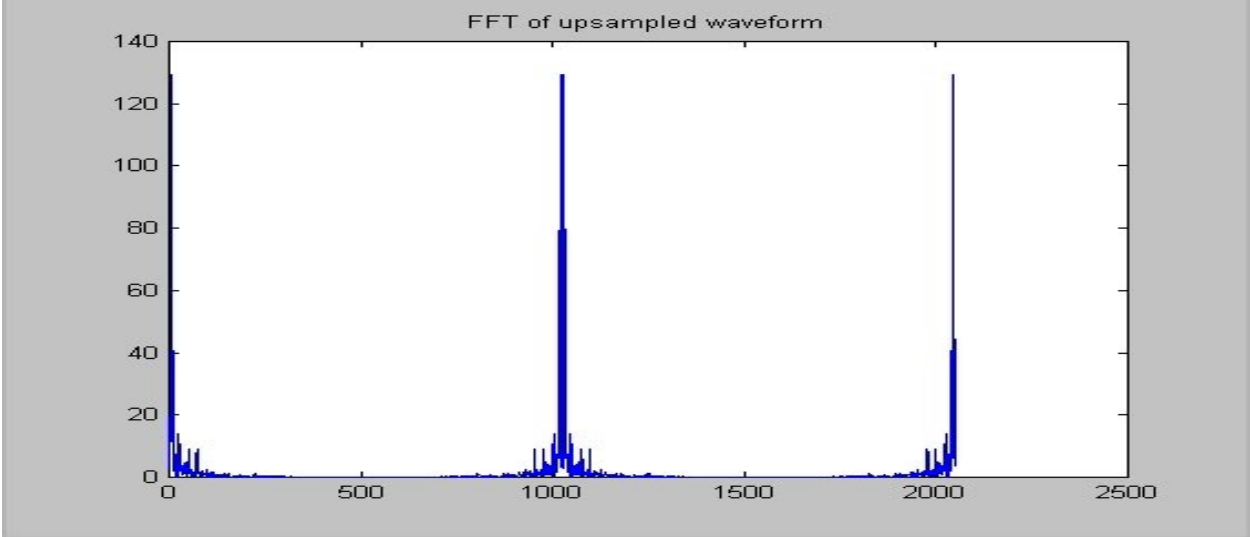
11025Hz. If we divide 22050Hz by 1024Hz, we get approximately 21 chunks per second. Each one of these chunks can represent a frequency as high as 512Hz. This frequency is high enough to get a decent representation of the input and allow us to shift it reasonably as long as we stay in the lower frequencies of the guitar. We may lose some harmonics on the shifted output notes, but if we use chunks that are much larger, we will lose accuracy, since notes can be quite short. Also, the FFT we implement needs to use a power of 2, so 1024 was an ideal choice. We ran into a small problem with some shifts, which I will go into detail after the explanation.

Once we have the 1024-sample chunk, we must shift it. Still using the example of a Fifth, we want to end up with a sample that is  $\frac{2}{3}$  the size of the original sample. This means that we want to "shrink" the 1024 sample chunk down to a 683-sample chunk. We cannot just take 2 samples, skip one, take two, etc... because this will change the waveform. We must first upsample the waveform by taking the original 1024-sample chunk. Next we interpolate zeros as follows and then create a 2048-sample vector as shown in Figure 2 below.

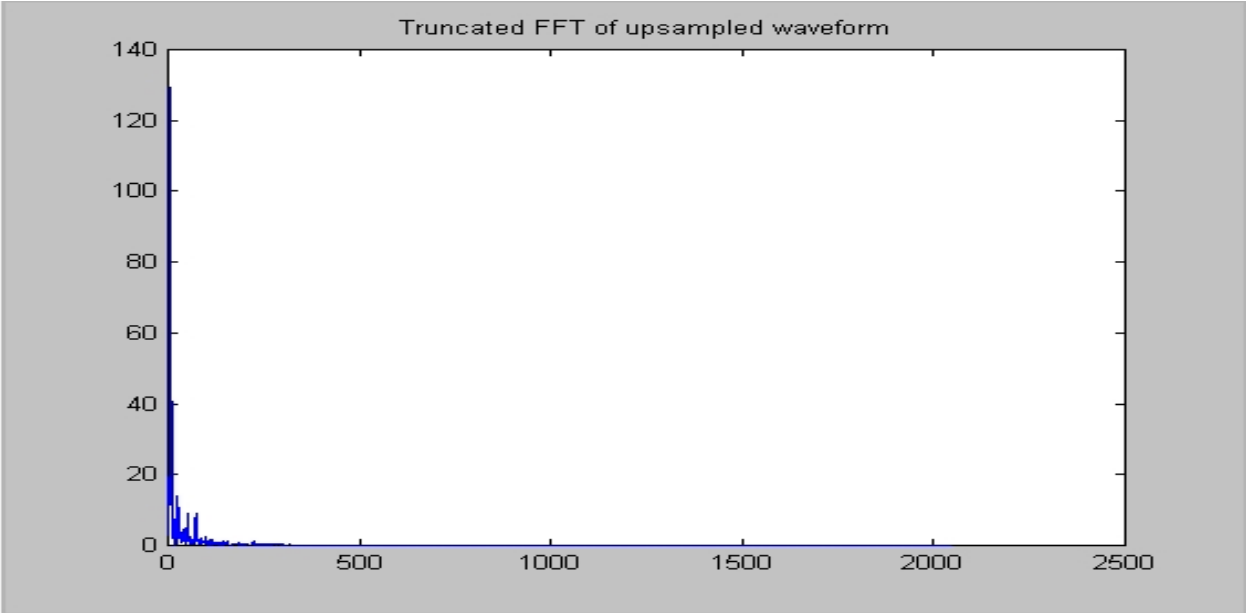


**Figure 2**

Once this is complete, we need to low-pass filter the upsampled waveform to interpolate the zeros with values. We did this simply by taking a 2048-pt FFT of the vector as shown in Figure 3 below.

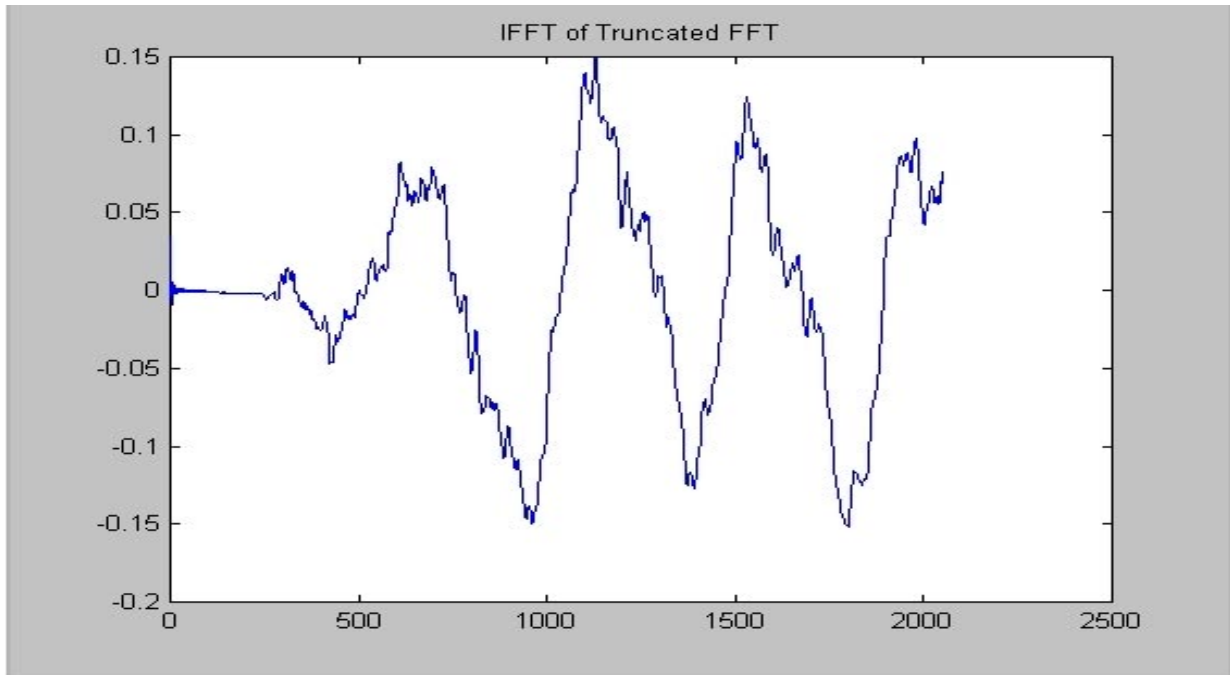


**Figure 3:** As you can plainly see, the FFT has two sets of peaks. To low-pass filter, we simply zero all values above 512 samples.

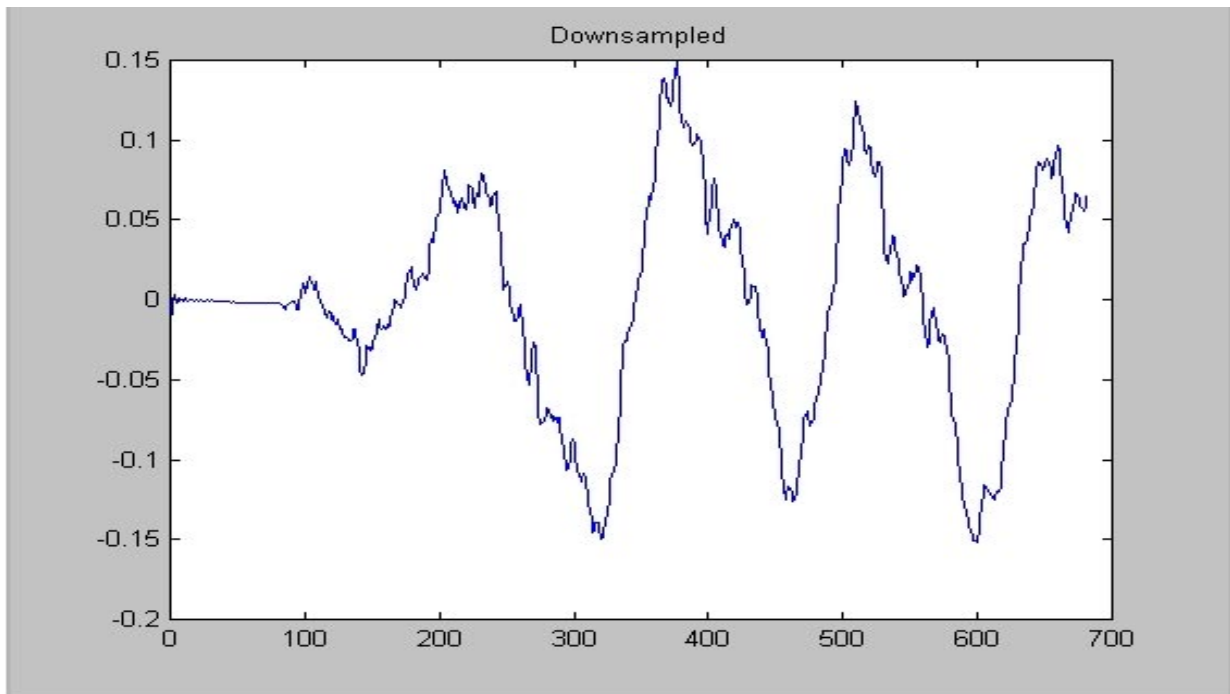


**Figure 4**

We then IFFT this truncated vector to get the interpolated 2048-sample waveform as shown in Figure 4 above.



**Figure 5:** Comparing this waveform to the original, we see that it is exactly the same, except for the higher resolution. Now we just need to take every third sample.



**Figure 6**

We now have a downsampled version of the input chunk as shown in Figure 6 above. To get a 1024 sample chunk from this 683-sample chunk, we simply cut and append the last third of the shifted waveform to the end. We then send this shifted waveform back to the output in the same location as the original chunk so that the two notes are played simultaneously.

Some problems encountered with this method were quite devious. When shifting by other ratios we begin by interpolating 2, 3, or 4 zeros instead of just one, which creates vectors of length 3072, 4096, or 5120 samples. Since our FFT cannot deal with vectors whose length is not a power of two, we had to find ways around this. So we simply adjusted the ratios and simply rounded when necessary. For a Forth, for example, we changed the ratio  $3/4$  to  $4/5.3333333$ . These new ratios worked perfectly. Another problem was that as we interpolated more zeros, we have to take bigger FFTs, and hence use up more CPU time.

One alternative approach we considered was to take 1024 sample chunks with an offset. We could overlap over chunks, so as not to need to append the last \_ or \_ to create the illusion of a 1024-sample chunk; however, this had a number of drawbacks such as increased CPU usage since we would be doing  $1/3$  more FFTs to determine the output. Also that we would have to determine the ratio prior to cutting the input into chunks in order to determine the number of samples to overlap. In the end, our method can sound clipped, but mostly retain the sound of the original guitar.

## RESULTS

### **Final Timing Data & Optimizations:**

Memory was a big concern; however, we were still able to keep the program in on-chip data. The big concern with memory was how to do the pitch shifting efficiently with all the

samples on off-chip memory. All of the input samples are located in SDRAM. We do this so we can store 1,045,504 floating-point samples. This is done to minimize the amount of times data is transferred between the EVM and the PC. The problem is that SDRAM is too slow, so we page 1024 samples into SBSRAM memory. Our program shifts the pitch of every note in the input. We do this using a for loop and an array that tells where each note starts. In this for loop we shift the pitch of 1024 sample chunks for as many samples as a note lasts. Since we only needed 1024 samples at a time we attempted paging for optimizations. We tried paging the samples into on-chip data, but we did not have enough memory left over on-chip. We paged both the input as well as the output.

First we placed the samples into SBSRAM memory using for loops. Then, in order to optimize the transaction, we unraveled the loop to take advantage of the EVM's parallelism. The chart on the next page will show an increase in performance of 6,320.3 cycles on average. Next we tried to get rid of the for loops used for paging, by using DMA transfers. DMA transfers were used for paging the input and the output. The chart will show an increase in performance, from the unraveled loops, by 8,169,222 cycles in total. We then tried to initiate two DMA transfers at once. That is, we used two DMA transfers to bring in two pages of samples. The chart will show an increase in performance from the previous DMA transfer by 43,819,184 cycles in total. There is an increase in number of cycles on average, but that is because we are shifting two pages within each iteration of the loop, which means that throughput goes up.

That's all we did for paging the samples. Our next optimization step was to look in every pitch shifting function we have and unravel each loop. This showed an increase in performance of 7,356,064 cycles in total when using one DMA transfer to page the input, and it showed a similar increase when using two DMA transfers to page the input. The only reason the increase



in performance would be different for the two is because of the overhead of using profile points. When optimizing the code for pitch shifting, DMA transfers were not considered because inside each loop there was some scaling of the index. This made DMA transfers impossible because DMA transfers copy one continuous piece of memory onto another.

One thing that we could not do was to place the paged input into on-chip memory. We had to place the stack on the SBSRAM memory, which was faster than the SDRAM, and that helped significantly. But if we could have kept the stack size down we could have made the program run a lot faster. Another thing we could have done was check if the FFT code we downloaded from the Internet was the fastest we could find. We should have looked more rigorously for a good FFT. (7)

<b>Data from Profile points</b>			
Testing	Count	Average cycles	Total number of cycles
For loops to page memory	131	43,947,882.5	5,757,172,605
Parallelism in for loops	131	43,941,562.2	5,756,344,651
One DMA trans. for input	131	43,879,201.7	5,748,175,429
Two DMA trans. for input	65	87,759,326.8	5,704,356,245
One DMA trans. w/ para.	131	43,823,048.6	5,740,819,365
Two DMA trans. w/ para.	65	87,679,561.4	5,696,998,989

<b>Profile times for each Pitch Shifting function</b>		
Shifting function	Count	Total number of cycles
Fifth	1	43,795,094
Forth	1	43,806,728
Major Third	1	43,802,780
Major Sixth	1	43,803,915
Minor Third	1	43,798,244

### **EVM Memory Usage:**

<b>Table of memory placement</b>			
	Origin addr.(in hex)	Length	Comments
Program text	260	47.5 Kb	Fast memory
Program stack	440140	UNINITIALIZED	SBSRAM
Far memory	3000000	UNINITIALIZED	SDRAM0 for input samples
Global Variables	2000000	UNINITIALIZED	
Global Const.	440000	315 bytes	SBSRAM

### **Purchases Required:**

The only additional equipment used was a guitar, microphone, and speakers. All equipment was acquired from personal owners, so no purchases were required.

### **SUMMARY**

In summary, we accomplished a lot in the amount of time given to work on this project. All of our algorithms worked to make a harmonizer. We got everything to fit into memory and also to work on the EVM. For our lab demo, we successfully inputted a mono 22 kHz .wav file of a guitar melody into the EVM for processing. After running our program we outputted a stereo signal of the same length with the pitch-shifted signal on one channel and the original signal on the other. The pitch-shifted signal wasn't quite as smooth as a guitar should sound, but they were good enough and in tune to act as a sufficient harmonizer to the original sound.

Given more time, we think we could have implemented harmonizing tunes with a different beat than that of the original signal. Also, we might have found a faster way to implement our algorithms so that we could process input signals on the fly. And we could add together more than one harmonized signal with the original for more complex chords. Another improvement would be to add more scales for a wider range of frequencies. But all of these improvements would require many years to accomplish.

## REFERENCES / WEB LINKS

- (1) [http://nd.essortment.com/elementsmusic\\_rllc.htm](http://nd.essortment.com/elementsmusic_rllc.htm)
- (2) Stonick, Virginia (1996). Digital Music Synthesis. Labs for Signals and Systems: Using MATLAB, pp. 25-32. Boston: PWS Publishing.
- (3) [http://ccrma-www.stanford.edu/~pdelac/220c/pitch\\_detection/img2.htm](http://ccrma-www.stanford.edu/~pdelac/220c/pitch_detection/img2.htm)
- (4) Stereo Wave File Format: <http://www.borg.com/~jglatt/tech/wave.htm>
- (5) Wave File Format:  
<http://www.technology.niagarac.on.ca/courses/comp630/WavFileFormat.html>
- (6) Lab 3 Code: <http://www.ece.cmu.edu/~ee551/Labs.html>
- (7) FFT Code: <http://www.mathcs.carleton.edu/faculty/jgoldfea/cs395/FFT>
- (8) Pitch Shifting Music Ratio Knowledge:  
[http://home.datacomm.ch/straub/mamuth/mamufaq.html#Q\\_ntet](http://home.datacomm.ch/straub/mamuth/mamufaq.html#Q_ntet)