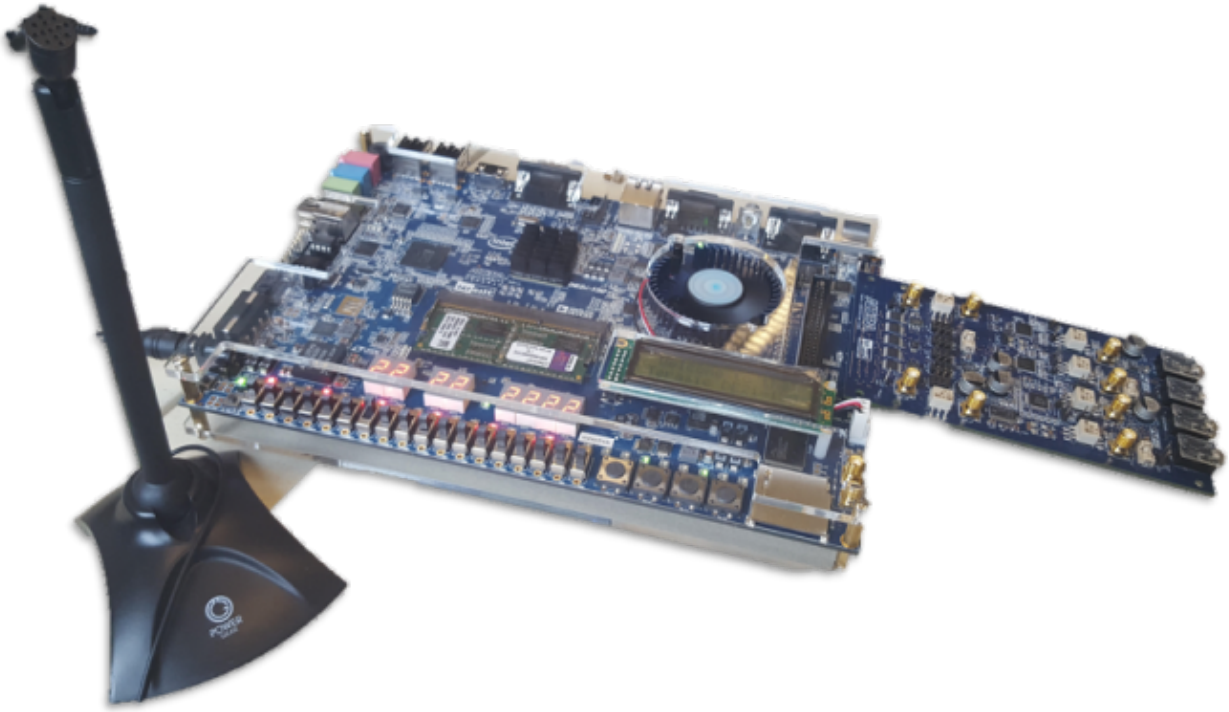
Pitch Shifter

UNIVERSITY of WASHINGTON



Jake Garrison and Jisoo Jung

443 DSP Capstone

6/9/16

Project Formulation

Motivation

We sought out a project that was a both challenging and applicable in a modern market. We were interested in learning new concepts and algorithms rather than purely applying concepts covered in lecture and labs. Real-time pitch shifting is present in most of today's music and films, and also crucial for synthetic speech in the AI revolution as well as voice masking for privacy reasons. Though there is significant research in this area, much of it focuses on higher level Matlab implementations, or use cases on machines with plenty of RAM and CPU power. The lack of preexisting low cost hardware that achieves real-time pitch shifting is a testament for both the demand and challenge of such a concept. Currently, a software pitch shift processor costs around $200 and a hardware version is upwards of $500. Since our solution relies on a standard DSP FPGA, we believe it is a cheaper hardware solution.

Historically, pitch shifting has been used in entertainment but was very limited with analog equipment. The effect dates back to the analog days when records could be played slower or faster, thus changing the pitch. The issue with these transformations is that time is also affected. It was not possible to change the pitch without affecting the time domain, or conversely changing the pitch without affecting time. With digital sound, this is now possible with methods utilizing the time or frequency domain, and it unlocks many new, powerful applications [6].

Our goal is to provide a portable, low power, real-time pitch shift effect processor with additional hardware based effects such as looping and echo, as well as a simple GUI for programming pitch shift melodies, sampling speech, generating tones, and general control.

Features

Hardware FPGA

- **Pitch Shift control**
  - Increment or decrement by semitone or constant factor
- **Melody Mapping**
  - Automatically shifts pitch according to user specified melody (via GUI)
- **Additional effects**
  - ***Echo:*** Repeats buffer with decaying amplitude. User specifies the echo period
  - ***Pitch Echo:*** Same as echo, but pitch changes at constant rate. User specifies the rate
  - ***Loop:*** Buffer is looped indefinitely. User can layer speech with different pitches creating harmony or a chorus effect
- **Input source**
  - Mic or Line in
- **Output mixing**
  - Mix dry and wet to left and right channels

Software GUI

- **Clickable piano keyboard**
  - Three octaves (ranging from C3 to B5)
- **Tone generator**
  - Sine, Square and Sawtooth
  - Maps to the keyboard notes
- **Speech Sampler**
  - User can record a short sample and map it to the keyboard notes
  - User can select sample start and end

    - Uses pitch shift algorithm to map sample to notes
- **Melody Composition**
    - User can record a melody by playing the keys and selecting each note's duration
    - The melody can be played through the tone generator or sampler
    - User can playback or reset melody
    - User can generate code to program the hardware's melody mapping
- **Hardware Settings**
    - GUI displays current hardware configuration to make it easier for the user to see the settings

See the Processing Flow section for specific control instructions

Implementation

Pitch Shifting Algorithms

In order to implement the core pitch shift function, three different relevant algorithms were explored:

i. PSOLA

PSOLA is a time domain method and stands for pitch synchronous overlap and add. Essentially it uses the overlap and add method to stretch the sample. The synchronous aspect is added to provide the most seamless overlapping by utilizing correlation to find the best overlapping coefficients. The pitch component uses resampling techniques to resample the stretched sample such that it restores the original length. Since resampling also changes the frequency proportionally to the magnitude of resampling, pitch shifting occurs after this process. In general, if you stretch the sample by a factor of *n*, you must resample the stretched version by *1/n*, which effectively removes every *nth* sample restoring the original sample length, but shifting the pitch by *n* in the process. Figure 1 below shows how overlap and add is used [7].

Pros:
- Fast to process
- Simple
- Ideal for tones

Cons:
- Reverberation
- Lower Quality
- Resampling algorithm

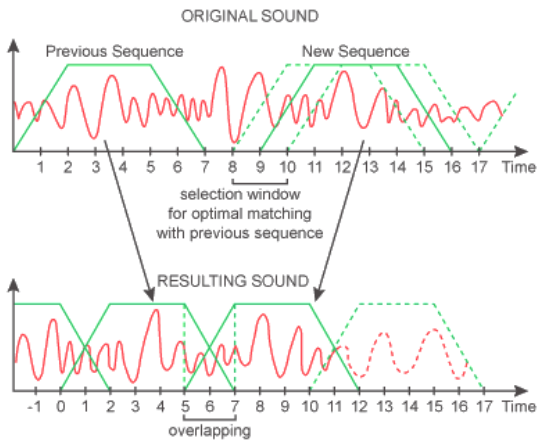


FIGURE 1: OVERLAP AND ADD METHOD

ii. Delay Based

The delay based method works in the time domain and crossfades between two channels with different varying delays and gains to produce a smoothly transitioned pitch shifted signal. To create an upward pitch change, we used a 30-millisecond delay and steadily decrease it at a rate that yields the desired pitch change. As the delay approaches 0, a second delay channel is started at 30 milliseconds and sweeps in a similar manner. A quick crossfade from the first to the second channel is applied, making sure the first channel is completely

faded out before its delay reaches 0. This process is repeated, going back and forth between the delay channels. A downward pitch change is achieved in a similar manner, only the delay channels are started with a near-zero initial delay, and the delay is increased out to around 30 milliseconds, at which time the alternate channel is started and the crossfade performed Figure 2 highlights the changing delays and the crossfading pattern.

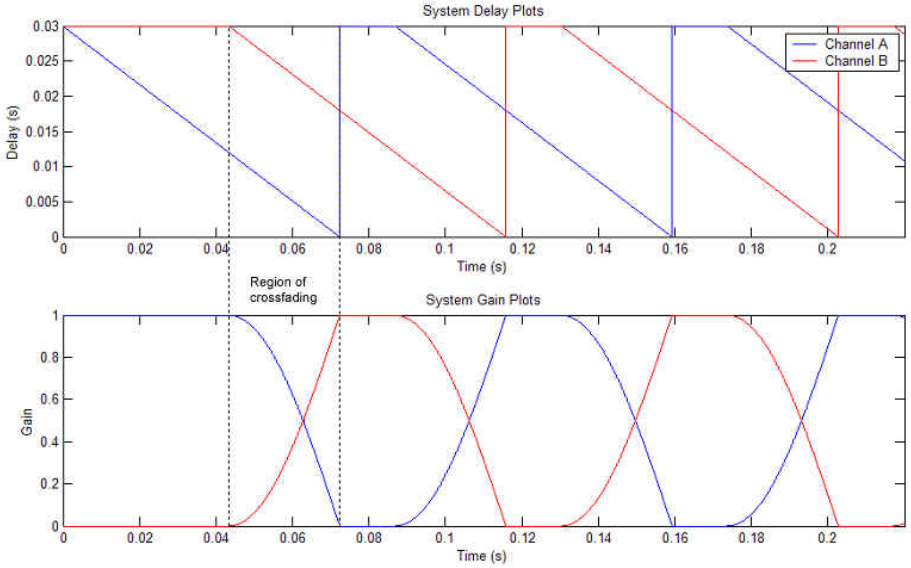


FIGURE 2: DELAY CROSSFADE AND DELAY MODULATION

The method produces more unwanted artifacts as the pitch shift factor is increased. While it is efficient enough to operate in real-time, the method shifts and smears the formant frequencies, which is irreversible. The delay based method is less computationally complex than frequency domain methods such as phase vocoding, and the results have fewer artifacts than PSOLA. As a result of these observations and significant prototype and FPGA testing, this method was chosen to be implemented for this project. Details of how the delay based method works are described in the Processing Flow section [8].

Pros:

- Fast to process
- No filtering
- Better quality than SOLA

Cons:

- Unwanted smeared frequencies
- Unintuitive

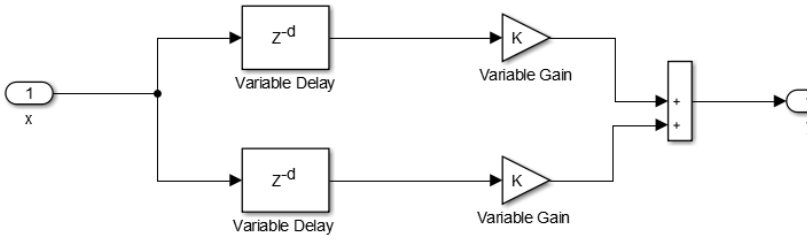


FIGURE 3: DELAY BASED METHOD

iii. Phase Vocoder

The phase vocoder method operates in the frequency domain and heavily relies on the STFT operation to shift the phase of the input. The algorithm can be broken down into an analysis, processing and synthesis operation. The analysis stage applies partitions the buffer, applying a Hanning window, then performs an FFT on each bin. The result is a time-lapse representation of the buffer's FFT, known as an STFT. The processing stage, which occurs on the frequency components, stretches the spectral components based on the requested pitch factor and then adjusts the phase of each bin such that the bins transition seamlessly. Finally, in synthesis, the inverse FFT is performed and Hanning window is applied to each bin, the bins are then overlapped such that the bins are a single buffer. The output buffer is the same length as input, but the frequency components are shifted. The diagram in Figure 4 below summarizes this process [3][4][5].

Pros:

- Best quality
- Customizable
- Ideal for speech

Cons:

- Lots of computations
- Slow
- Complex math

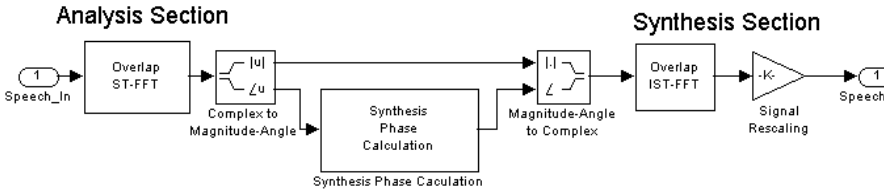


FIGURE 4: PHASE VOCODER METHOD

Design Workflow

In order to learn, and test the different implementations discussed above, we followed a predefined design process shown below. The status update of the process for each algorithm is also shown.

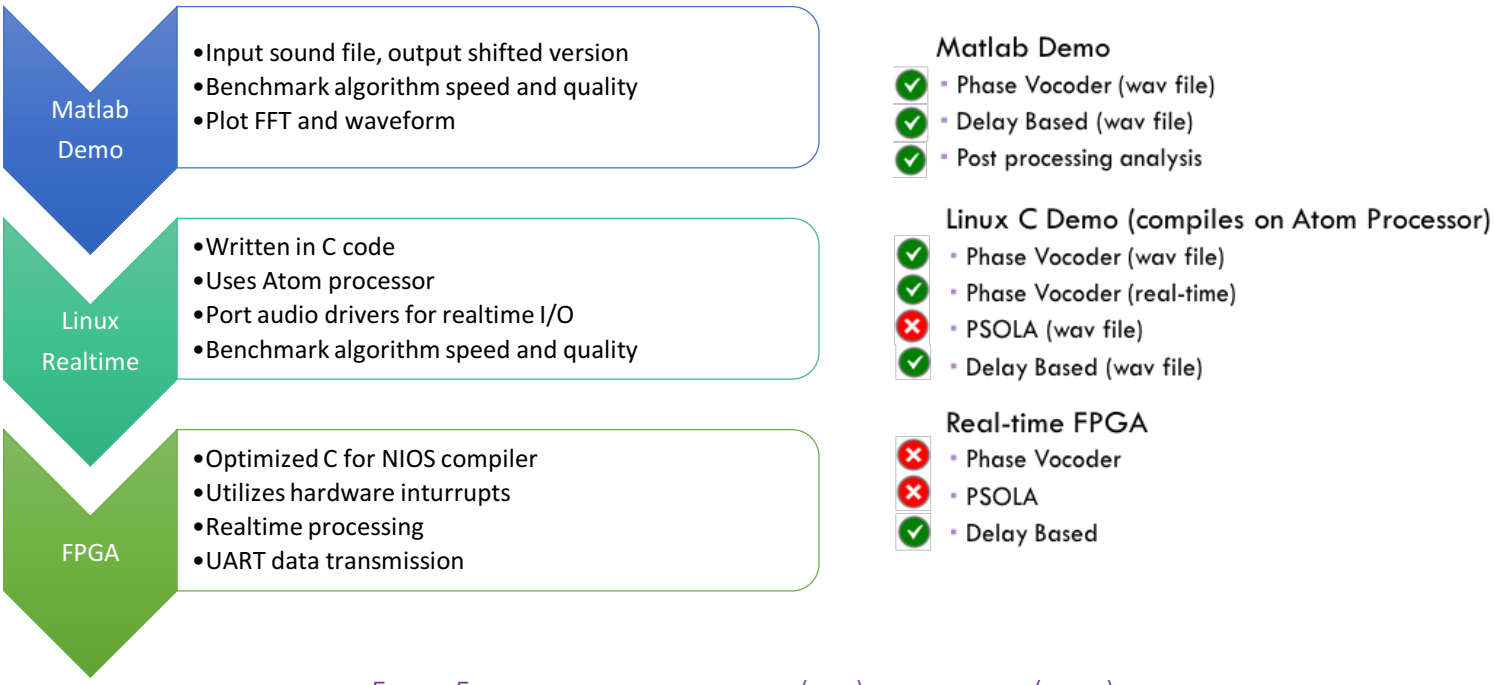


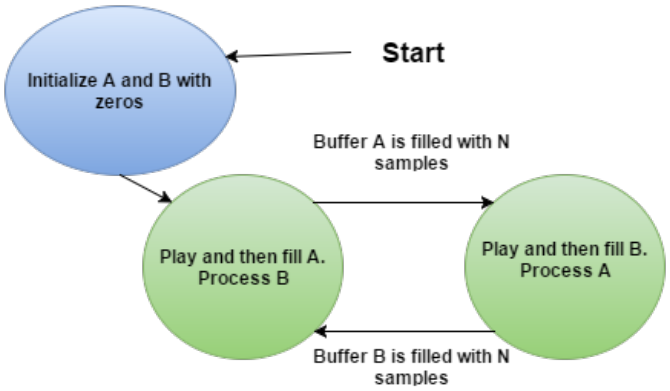
FIGURE 5: ALGORITHM DESIGN PROCESS (LEFT) , FINAL STATUS (RIGHT)

We chose to use the delay based method as it was the only method that succeeded in our design process. Implementing several versions of each algorithm was not all for nothing, we now have a solid understanding of the performance, sound quality and processing steps intrinsic to each method. This helped immensely for debugging and optimizing. The main factor that caused other methods to fail is in the processing time of the algorithm. While the phase vocoder offers superior sound quality, the time to process a buffer, due to the several required FFTs, well exceeded the time to fill/playback a buffer. As a result, there were audible pauses between the playback of the output buffer which caused an undesired stutter effect. While ping pong buffering minimized this, it was still unavoidable due to the processing time of the phase vocoder.

While the delay based method also suffered from this processing bottleneck issue, it was much more minor and could be masked such that it was not as audible to the user. This is explained in the Results section.

Ping Pong Buffering

To minimize memory usage and processing time, ping pong buffering was utilized. In ping pong buffering, two buffers are initialized (A and B) and a pointer alternates between the selection of the two. Before the pointer alternates, one buffer is processed (pitch shifted in this case), while the other buffer plays through the previously processed buffer, while filling it with new samples immediately after a previous sample is played. Once the operations on both buffers are complete, the pointer swaps. Figure 6 demonstrates this logic.




```
      A: buffer PING
      B: buffer PONG
      int *processBuffer     = A
      int *playAndFillBuffer = B
      -------------- SCHEDULE -----------------
t=0   | Start:
      | Initialize A and B with zeros
      |
      | A                      B
      | Start process          Start play and fill
      | .                      .
      | process done           .
      | .                      .
      | .                      play and fill done
      | ----------------------------------------
      |           Swap ptr
      | ----------------------------------------
      | Start play and fill    Start process
      | .                      .
      | .                      process done
      | .                      .
      | play and fill done     .
      | ----------------------------------------
      |           Swap ptr
      | ----------------------------------------
      |           Repeat
      |             .
      |             .
      |             .
t=N   V
```

FIGURE 6: PING PONG BUFFER DIAGRAMS

In our case, the bottle neck was the processing step. Even with a sampling rate of 8khz and optimized pitch shift code, buffer A would play and fill before buffer B finished processing. We attempted to change many of the pitch shift parameters (buffer size, delay time ect.) hoping to overcome this, but the best parameters yielded a processing time approximatley 10% slower than the play and fill buffer time. Our solution was to make a copy of the buffer right after processing the pitch shift, then use the play through this copy while after the play and fill buffer finishes. The result is a brief repeat of the pitch shifted buffer that lasts the duration of the difference between the processing time and play and fill time. This solution performs very well with periodic tones and is hardly noticeable with speech. You can see the resulting waveform in the Results section. Our original solution was to play silence instead, but this resulted in a choppy output as each buffer has a series of zeros outputted before the next buffer was ready.

Pitch Shift Effect

The delay based pitch shift has a few important parameters worth discussing. As mentioned in the Delay Based Algorithm, a delay time for shifting up and shifting down must be specified. These delay values were initialized to be 240 for up and 4 for down. Given that our sampling rate is 8khz, 240/8k is about 30 ms and 4/8k is very close to 0 ms. Additionally, we initialized the buffer size to be 256 samples (~30 ms). Using a larger buffer yields better quality at the expense of longer processing time, while a smaller buffer degrades audio quality. We experimented with several configurations and settled with 256. The crossfading between the independently delayed buffers uses a sine curve mapped to the gains for a smooth, quick crossfade. The two buffers utilize circular buffering to avoid reallocating the buffers when they are full. There is essentially two states that alternate indefinitely to create the pitch shift. State one is playing one of the two buffers while altering the delay at a constant rate, and state two is crossfading between the silent buffer and the one playing in state one.

The PitchShift.h defines the delay based pitch shift function. The inputs are a pointer to the input and output buffer, as well as the pitch shift factor. The hardware KEY1 (increment) and KEY2 (decrement) alter the pitch factor by 0.1 if SW0 is off or a single semitone if SW0 is on. When melody mode is active, the pitch factor is automatically cycled to fit the input melody.

Additional Effects

Melody Mode

If switch 1 is toggled, melody mode is enabled. In this mode, a hardcoded melody is repeated indefinitely such that the pitch shifts to each indexed note for the pitch and duration specified in the melody array. The melody array is of the form [duration (number of samples), pitch factor, …]. A note is defined by the duration and subsequent pitch. To parse the array, one must iterate every two indices, so that a note can be defined. The length of the melody in notes is then half the length of the melody array. The GUI is capable of generating an array that can be copied into the C code and used.

Time Varying Effects

The hardware switches (SW2 to SW4) are utilized to toggle different effects post pitch shift. Three different effects are implemented to process the shifted input and provide the user with more options for controlling the output. The Features section above highlights how these effects work and Results section below shows example oscilloscope screenshots of these effects being applied.

The echo, pitch echo and loop store 5000 samples for repeating each echo with a decreasing amplitude multiplier. For the loop effect, the buffer is repeated and added with incoming samples. There is a pause between each loop to slow down the rate at which the buffer is looped.

Processing Flow

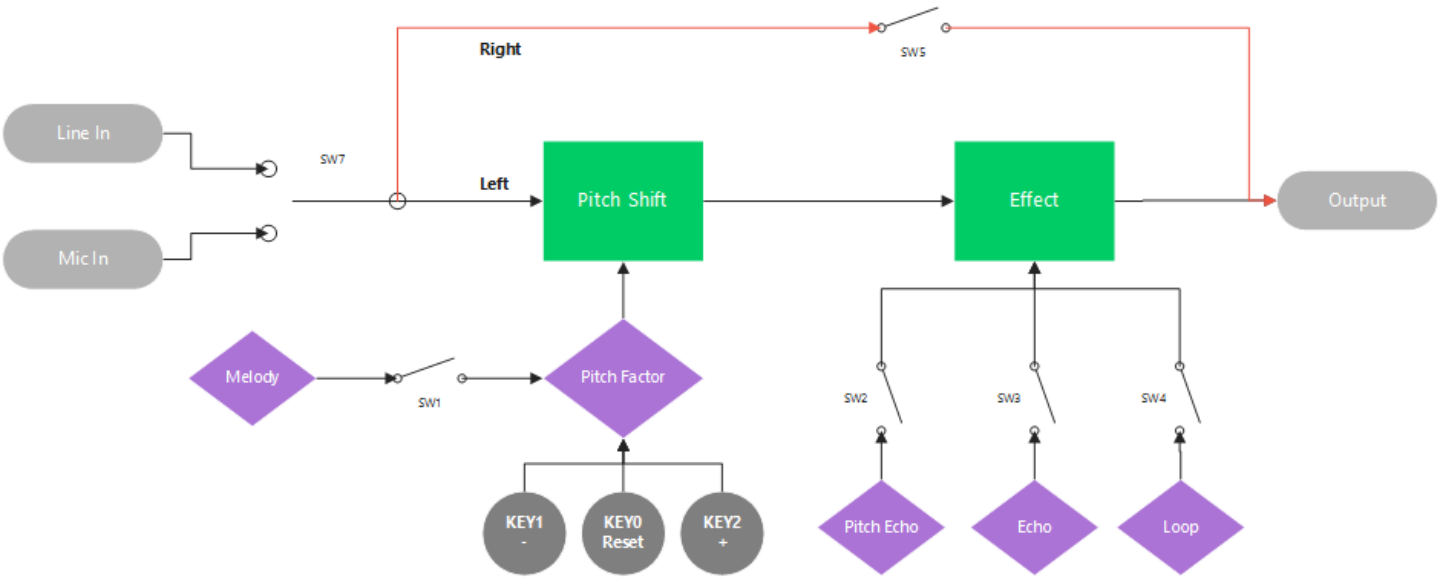


FIGURE 7: PROCESSING DIAGRAM

I/O Control

Input data is sampled at 8khz

- **SW5 ON:** Enable dry input routing to right output channel
- **SW5 OFF:** Disable right output channel (Wet only)
- **SW7 ON:** Microphone input
- **SW7 OFF:** Line Input

Pitch Shift Control

- **KEY0:** Pitch reset (p = 1, semitone = 0, no shift)
- **SW0 OFF:** pitch factor (inharmonic multiplier)
    - KEY1: Pitch shift decrement 0.1
    - KEY2: Pitch shift increment 0.1
- **SW0 ON:** semitone mode (half step mapping)
    - KEY1: Semitone decrement 1
    - KEY2: Semitone increment 1

UART Control

Sends current pitch factor, switch configuration and audio buffer to our GUI

- **KEY3:** Send UART buffer

Effects Control

SW1 – SW4: ON is enabled, OFF is disabled, only one can be enabled

- **SW1 Melody Auto tune:** pitch changes according to user's melody and tempo
    - Keys control pitch shift
- **SW2 Pitch Decay:** Input is echoed, pitch shifting each echo by a given rate
    - KEY0: decrease pitch change rate
    - KEY1: increase pitch change rate
- **SW3 Echo:** Input is echoed with constant pitch and variable decay
    - KEY0: decrease decay rate
    - KEY1: increase decay rate
- **SW4 Loop:** Loop input indefinitely
    - Keys control pitch shift

Matlab GUI

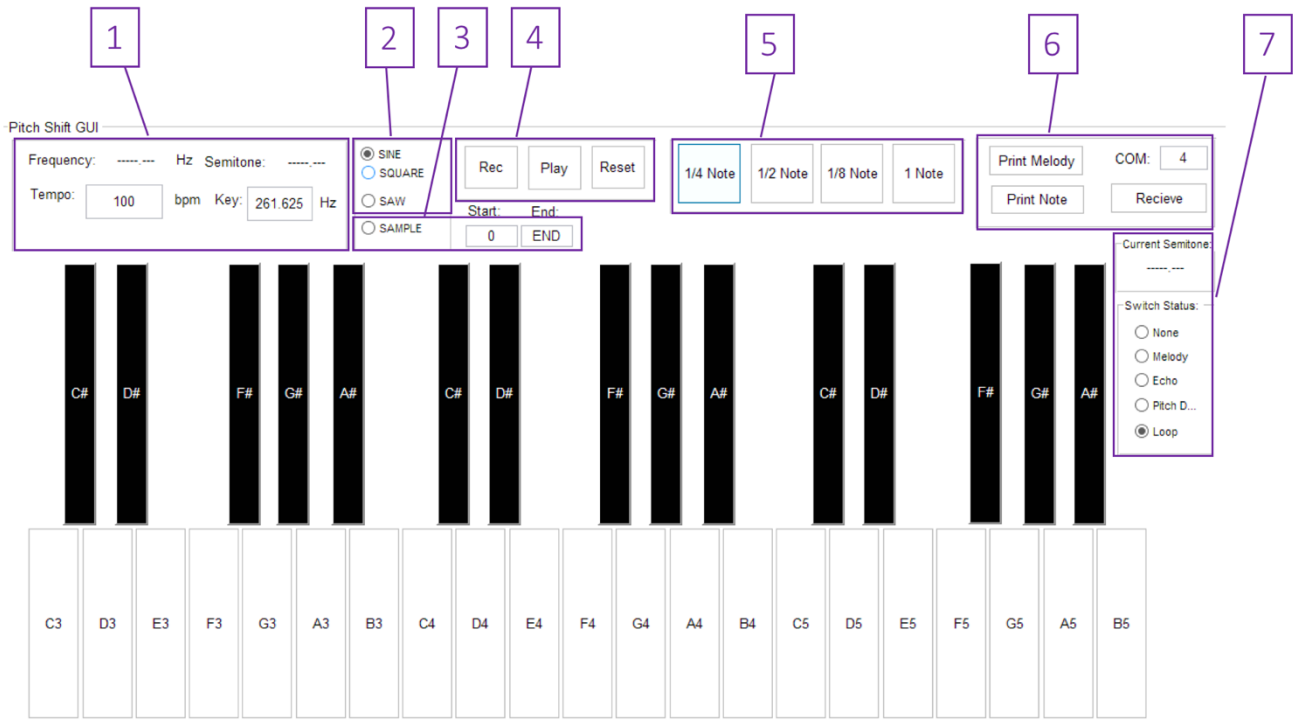


FIGURE 8: MATLAB GUI CONTROLS

1. **Global settings:** Displays current note as well as global tempo and key
    a. User an input tempo and key note frequency
2. **Waveform select:** Choose waveform to map to piano keys
3. **Sample Mode:** Maps sample (from UART) to piano keys using pitch shift algorithm
    a. User can input sample start and end points

4. **Melody Controls:** Play, record or reset a melody
5. **Note length:** Sets duration of next note played
6. **UART Control:** Initialize and receive UART
    a. User can set COM port
    b. Receive, then press KEY3 on hardware to send switch configuration, current note and an audio sample
    c. Audio sample is used when in Sample Mode
7. **Hardware Status:** Indicates current pitch setting and Effects switch configuration

Results and Discussion

Discussion

Struggles

Though we were able to successfully implement real-time pitch shifting with several other features, it did not come without struggles and issues.

Our original plan was to use phase vocoding due to its superior output quality. We spent the first 3 weeks prototyping and optimizing this method and it seemed to work great. Upon porting it to the FPGA, we found the performance suffered drastically. On the Atom processor, the algorithm was able to finish in the time it took to fill a buffer of 64 samples. On the FPGA, this operation took about 5 seconds, which was significantly slower than the time to fill the buffer. Even with parameter optimization and ping pong buffering, this problem did not get much better. With a bit over a week left, we were forced to switch to time domain methods, despite writing a proposal suggesting these methods are inadequate for speech. We spent two very long days researching and writing Matlab, Linux and FPGA code for PSOLA and the delay method and decided the delay method would be best for the FPGA. Following this, we gave our class presentation and finalized the FPGA code. If not for this 100% commitment to the algorithm pivot, we would have never had a working demo, let alone the additional effects and GUI that was added in the last weekend. Having the insight to use the delay method rather than phase vocoding early on would have saved us several days of effort, but in the end, we were able to master three independent methods of pitch shifting rather than one, so perhaps it was not a bad thing.

In addition, we had some issues trying to send and receive UART data to and from the GUI. Ideally, the GUI would simply be a slave to the hardware and all sound processing would occur on the board. Due to the time crunch and UART issues, we were unable to automatically send the melody and other controls over UART and resorted to the code copy and paste method shown in the demo. Additionally, the tone generator and sampler were processed on the Matlab side rather than FPGA.

Finally, the audio quality of the pitch shift is not nearly as clear as the Linux and Matlab implementations. This is mostly due to the algorithm choice, and also the sacrifices made to achieve real-time on the FPGA. All in all, the algorithm works as advertised and is packaged with a powerful GUI and additional effects and features.

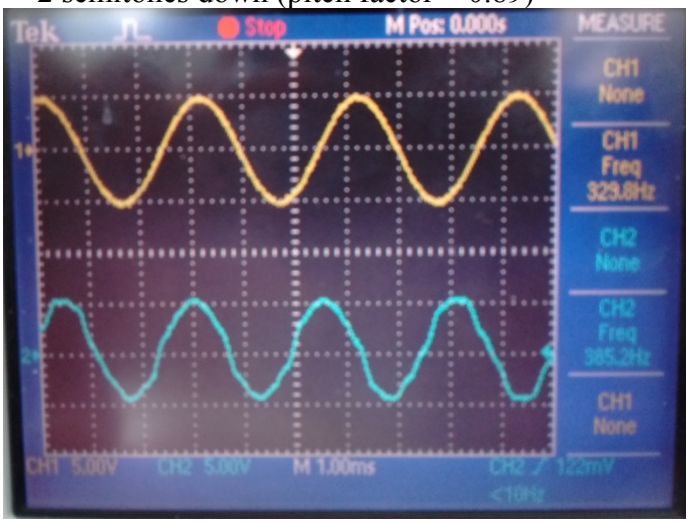
Results

Below are some oscilloscope demonstrating the input output relationship of our pitch shifter processing:
NOTE: The Yellow channel is input signal, Blue is output

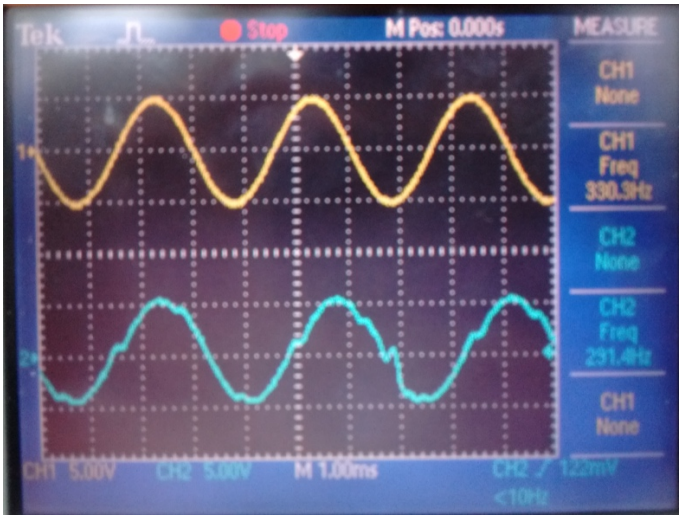
Pitch Shift

Sine and Square Wave:

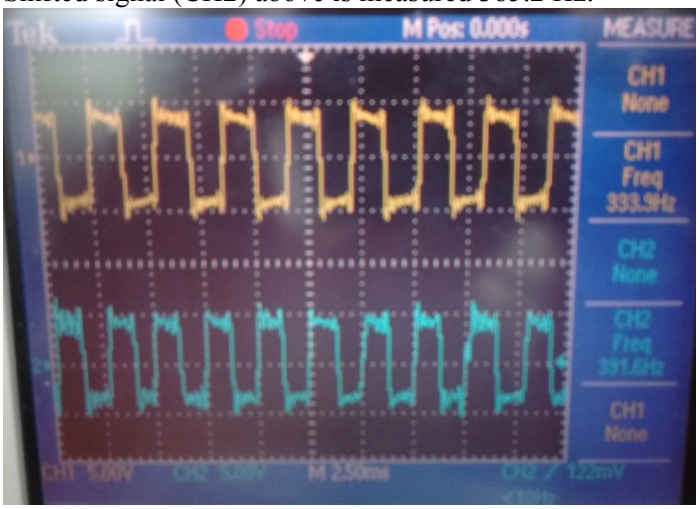
- The note E4 (frequency of 329.63 Hz) was played for below four cases
- 3 semitones up (pitch factor = 1.19)
- 2 semitones down (pitch factor = 0.89)



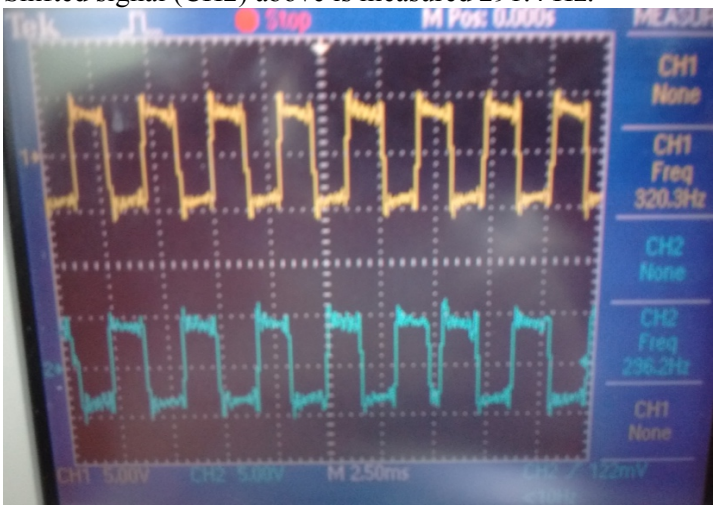
Shift up by 3 semitones. Input is measured to be 329.8 Hz so the expected frequency is 329.8 * 1.19 = 392.1 Hz. Shifted signal (CH2) above is measured 385.2 Hz.



Shift down by 2 semitones. Input is measured to be 330.3 Hz so the expected frequency is 330.3 * 0.89 = 294.3 Hz. Shifted signal (CH2) above is measured 291.4 Hz.



Shift up by 3 semitones. Input is measured to be 333.9 Hz so the expected frequency is 333.9 * 1.19 = 397.03 Hz. Shifted signal (CH2) above is measured 391.6 Hz

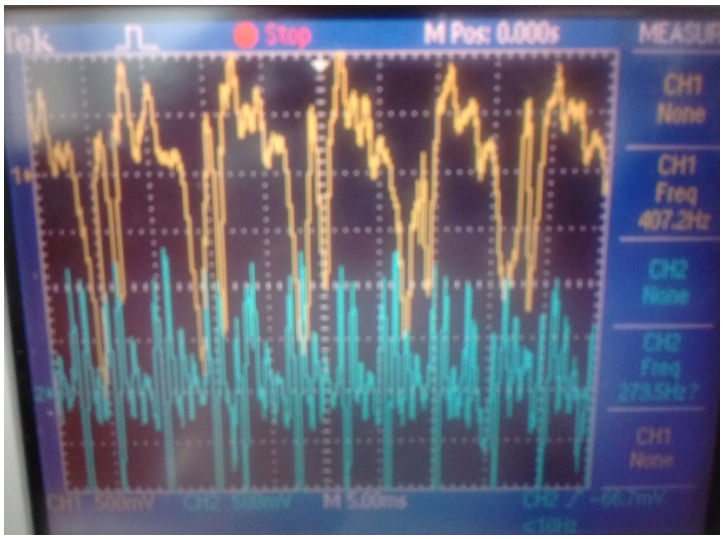


Shift down by 2 semitones. Input is measured to be 320.3 Hz so the expected frequency is 320.3 * 0.89 = 285.39 Hz. Shifted signal (CH2) above is measured 296.2 Hz

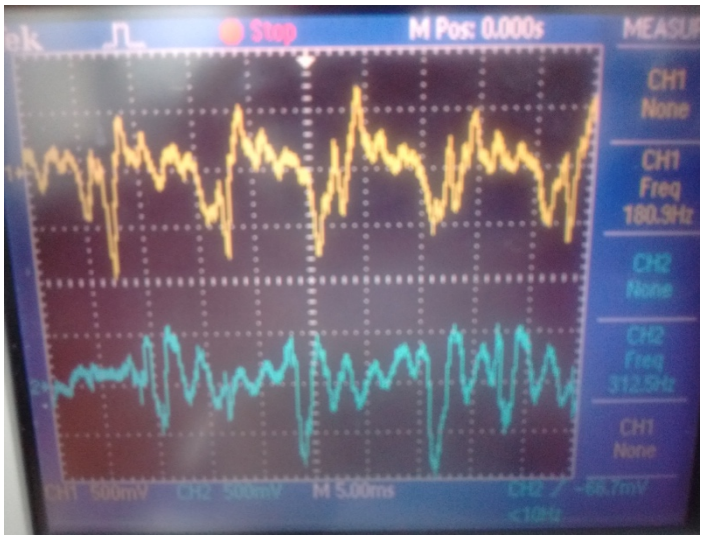
Speech:

Expecting how spoken words should look like is very difficult. So in this experiment sine wave was generated first to verify that the pitch shifting is working and then speech samples were taken. Two characteristics, duration and shifted pitch, were mainly observed in this experiment.

- shift up by 12 semitones (pitch factor = 2)
- shift down by 4 semitones (pitch factor = 0.79)



'Hello' shifted up by 12 semitones. The resulting signal is still audible with higher pitch with the same duration.

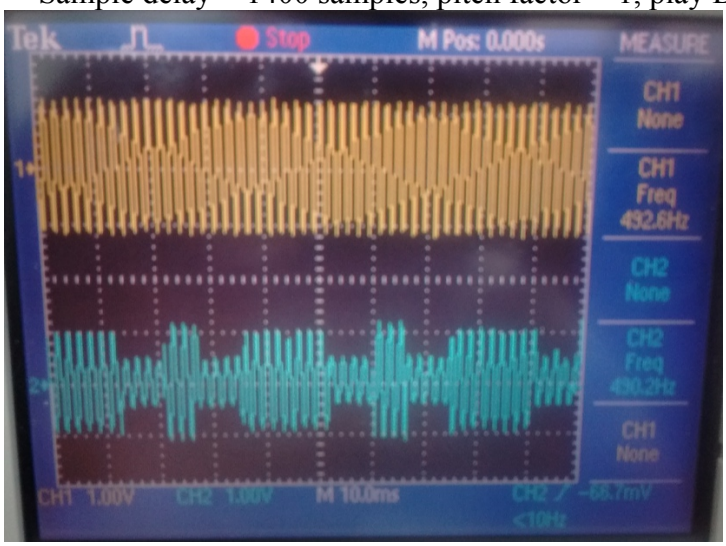


'Hello' shift down by 4 semitones. The resulting signal is still audible with higher pitch with the same duration.

Additional Effects

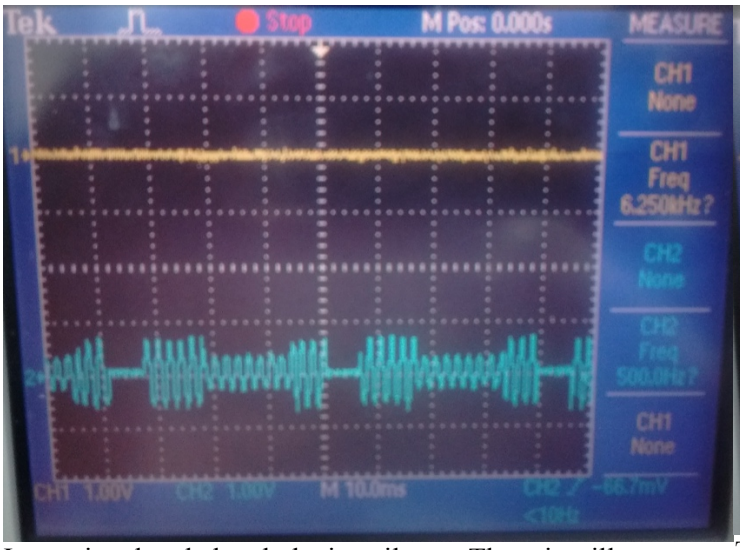
Echo:

- Sample delay = 1400 samples, pitch factor = 1, play B4 (frequency 493.9 Hz)

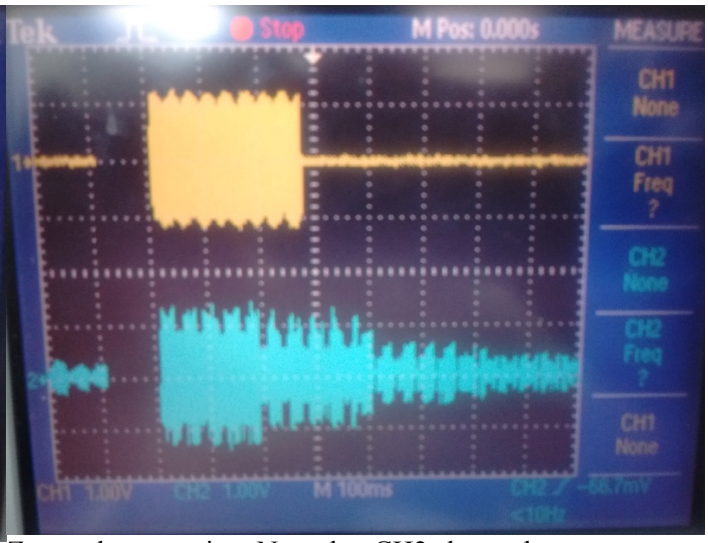


Both input and output are displayed. Fluctuation in the output is the result of decayed sample keep added to the current sample.

Input signal almost finished while echo still is remaining.



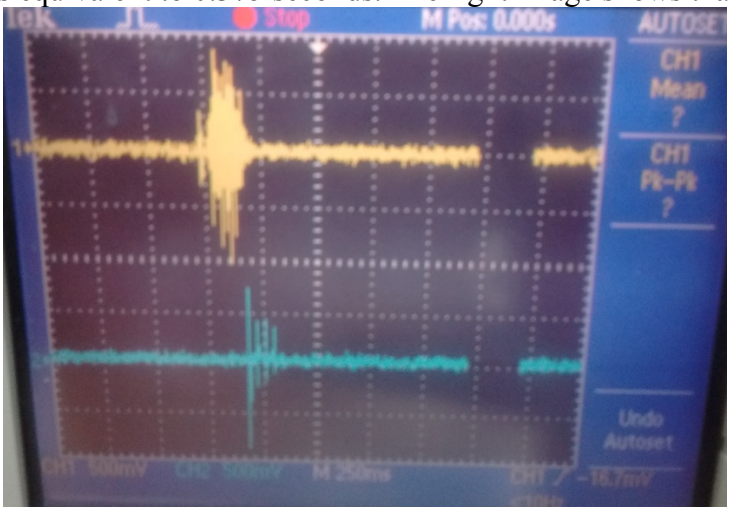
Input signal ended and playing silence. There is still an echo played in CH2 with reduced amplitude.



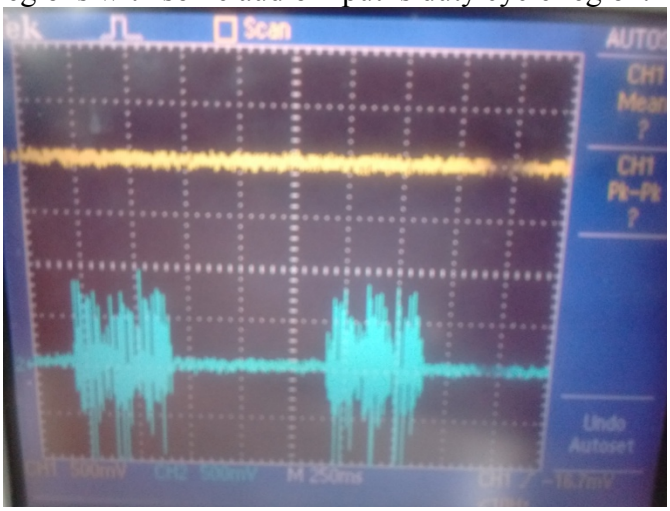
Zoomed out version. Note that CH2 shows decay over time.

Loop:

NOTE: Buffer that stores previous samples has a length set to be 5000, which in 8000 Hz sampling frequency, is 5000/8000 = 0.625 seconds worth of samples. In this test case, duty cycle of the loop is 3000 samples, which is equivalent to 0.375 seconds. The right image shows that regions with some audio input is duty cycle region.



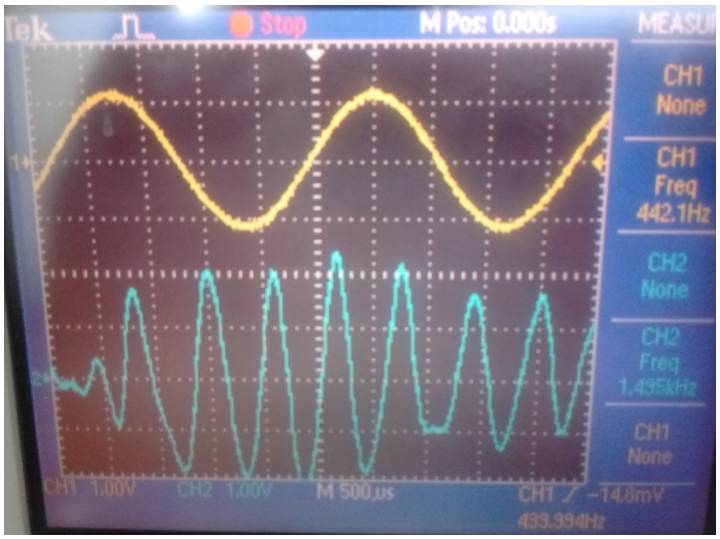
CH1 shows a signal to be repeated.



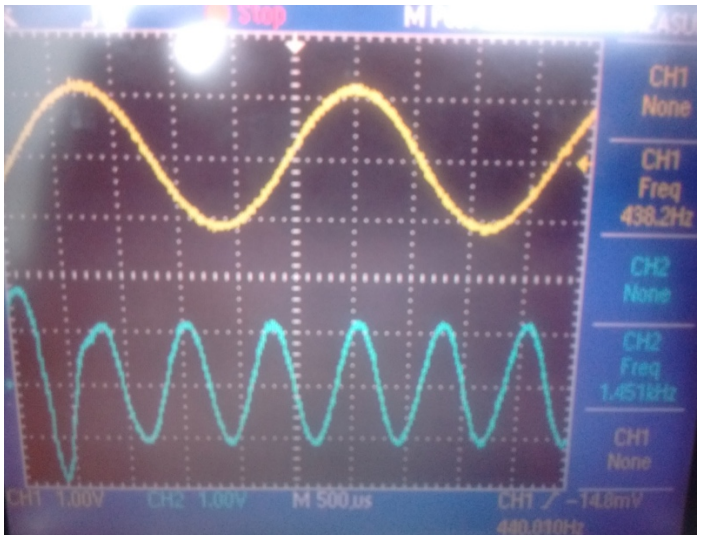
CH2 looping previously inputted signal. Note that noise is added in between the first and second capture.

Pitch echo:

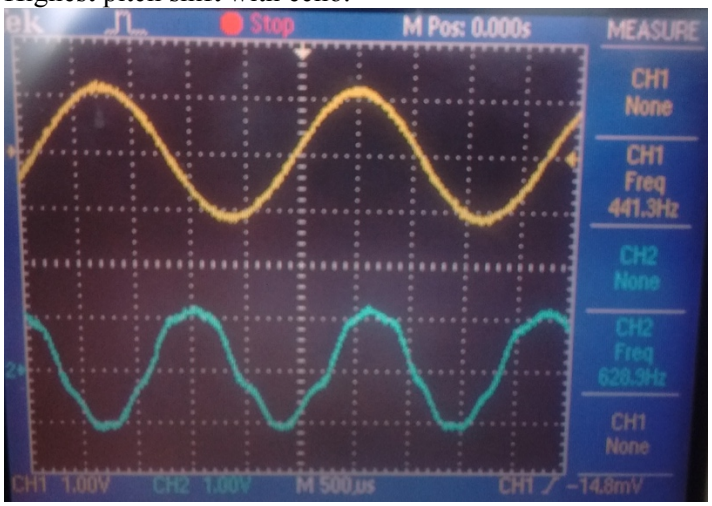
- Sine wave of frequency 440 Hz played.
- NOTE: This feature combines echo and pitch shifting at the same time. Plots above does not show consistent frequency because of echoing effect.



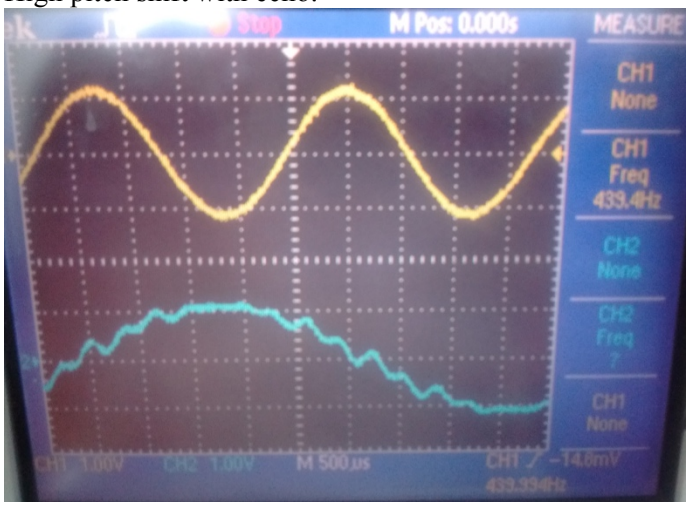
Highest pitch shift with echo.



High pitch shift with echo.

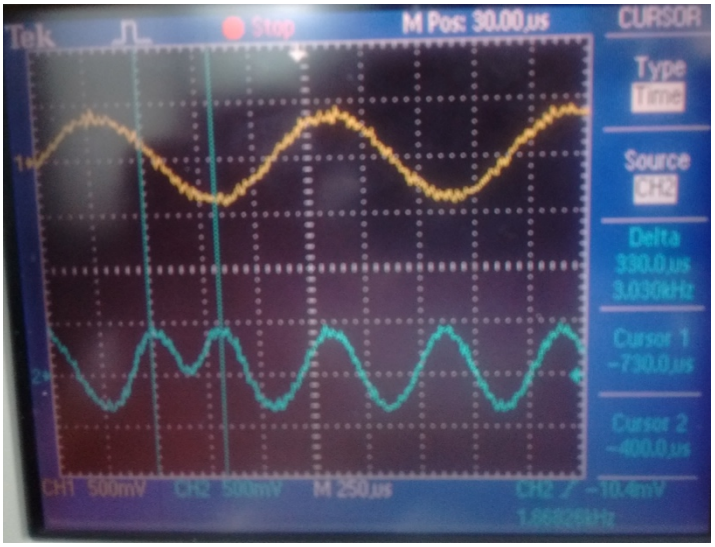


Mid pitch shift with echo.



Low pitch shift with echo. Ripples are created by echoing effect.

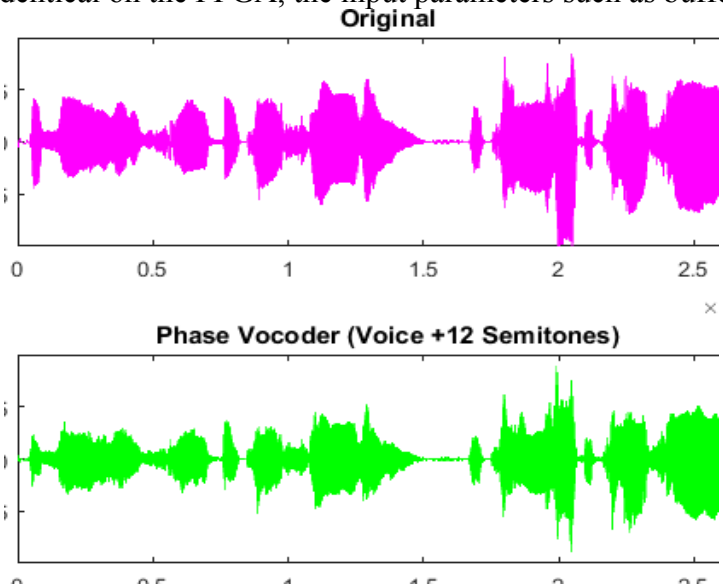
Ping Pong Buffer Latency



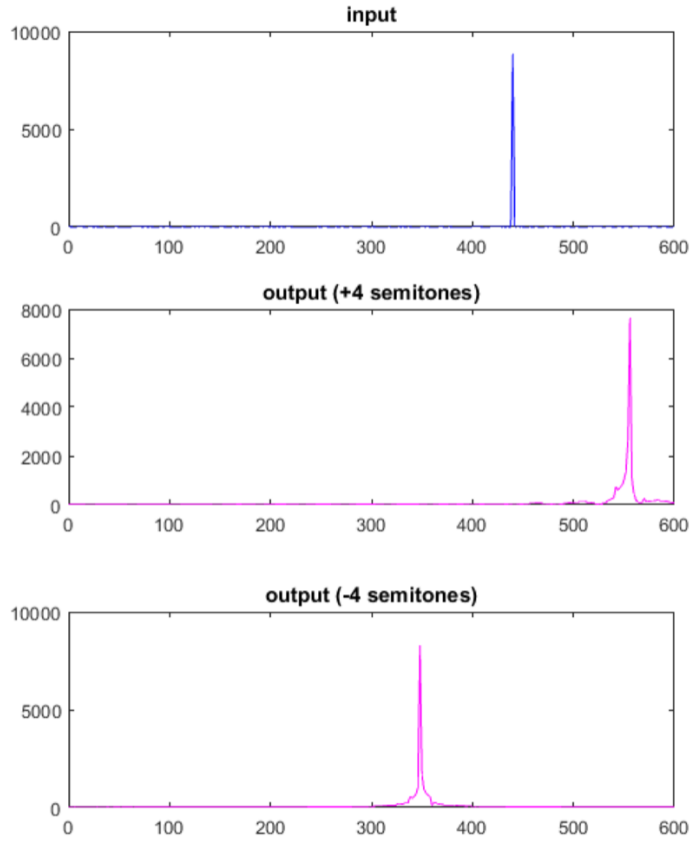
Due to hardware limitation, there are times when delay based algorithm is not yet processed in buffer A when buffer B already finished playing its samples to line out. When this issue occurs, last processed buffer is played back again to provide better sound quality. Two cursors in the oscilloscope image below shows a time when the last-processed buffer is being played back again. It is approximately a third of a wavelength, or 330 micro seconds or 3 samples at 8khz sampling rate.

Matlab Implementation:

The following plots came from our Matlab implementations of pitch shift. While the algorithm is almost identical on the FPGA, the input parameters such as buffer size differ.



Since this pitch shift is done in Matlab, phase vocoder technique can be used, hence, the result is much cleaner and higher quality.



Matlab FFT of our delay based pitch shift. Notice the imperfect frequency smearing that takes place.

References

[1] O. Parviainen, "article", *Surina.net*, 2016. [Online]. Available: http://www.surina.net/article/time-and-pitch-scaling.html. [Accessed: 05- May- 2016].

[2] "Pitch Shifting and Time Dilation Using a Phase Vocoder in MATLAB - MATLAB & Simulink Example", *Mathworks.com*, 2016. [Online]. Available: http://www.mathworks.com/help/audio/examples/pitch-shifting-and-time-dilation-using-a-phase-vocoder-in-matlab.html. [Accessed: 05- May- 2016].

[3] "Time Stretching And Pitch Shifting of Audio Signals – An Overview | Stephan Bernsee's Blog", *Blogs.zynaptiq.com*, 2007. [Online]. Available: http://blogs.zynaptiq.com/bernsee/time-pitch-overview/. [Accessed: 05- May- 2016].

[4] J. Laroche and M. Dolson, "NEW PHASE-VOCODER TECHNIQUES FOR PITCH-SHIFTING, HARMONIZING AND OTHER EXOTIC EFFECTS", 2016. [Online]. Available: http://www.ee.columbia.edu/~dpwe/papers/LaroD99-pvoc.pdf. [Accessed: 05- May- 2016].

[5] D. Ellis, "A Phase Vocoder in Matlab", *Labrosa.ee.columbia.edu*, 2016. [Online]. Available: http://labrosa.ee.columbia.edu/matlab/pvoc/. [Accessed: 05- May- 2016].

[6]"Time Stretching And Pitch Shifting of Audio Signals – An Overview | Stephan Bernsee's Blog", *Blogs.zynaptiq.com*, 2007. [Online]. Available: http://blogs.zynaptiq.com/bernsee/time-pitch-overview/. [Accessed: 10- Jun- 2016].

[7]"Real-time FPGA Pitch Shifter | EEWeb Community", *Eeweb.com*, 2016. [Online]. Available: https://www.eeweb.com/project/dylan_hughes/real-time-fpga-pitch-shifter. [Accessed: 10- Jun- 2016].

[8]"A Real", *Cnel.ufl.edu*, 2016. [Online]. Available: http://cnel.ufl.edu/~harpreet/projects/speech_website/main.htm. [Accessed: 10- Jun- 2016].