# Real-Time Speech Pitch Shifting on an FPGA

Habib Estephan, Scott Sawyer, Daniel Wanninger
{habib.estephan, scott.sawyer, daniel.wanninger}@villanova.edu
http://homepage.villanova.edu/scott.sawyer/fpga

Advisor: Dr. Kevin Buckley
Department of Electrical and Computer Engineering, Villanova University
February 23, 2006

***Abstract* – The objective of this project is to create a real-time speech pitch shifter implemented on a Xilinx Virtex-II Pro Field-Programmable Gate Array (FPGA). FPGAs (programmable devices containing a vast assortment of logic fabric) are becoming increasingly popular for the implementation of digital signal processing (DSP) applications. The project places equal emphasis on proposing a viable DSP algorithm and implementing the technique as a digital system on FPGA hardware. Three pitch shifting algorithms (single-sideband modulation, time-domain processing, and frequency-domain processing) are presented, and the FPGA implementation process for each technique is explored. The report concludes by summarizing deliverables and accomplishments and by analyzing the practicality of performing audio processing on an FPGA. Although the final hardware implementation does not meet all original project specifications, the knowledge gained about speech processing and FPGA development make this project an educational success.**

## TABLE OF CONTENTS



## ACKNOWLEDGMENTS

The project group would like to acknowledge the help and guidance of advisor Dr. Kevin Buckley, as well as the generous contributions of DSP Specialist Sean Gallagher and Xilinx, Inc.

## I. INTRODUCTION

### *A. Background*

Traditionally, real-time digital signal processing (DSP) has been performed using either dedicated microprocessors or custom application-specific integrated circuits (ASICs). Microprocessor-based DSP boasts low design costs, but its performance is limited by the chip's processing capabilities per cycle and maximum clock speed. Because a single microprocessor can only perform one operation at a time, this technology can prove inadequate for applications requiring high-order filters and fast data rates. For increased performance and lower power consumption, DSP engineers have turned to custom ASICs, in which a digital system consisting of logic gates and interconnects is implemented in a tiny silicon chip using very-large scale integration (VLSI) technology and design techniques. The advantage of ASICs is that the chip enables parallel processing by containing many multipliers, adders, and logic functions operating in unison, compared to the single ALU contained in a microprocessor. Additionally, ASICs do not require the operational overhead of fetching and executing instructions. However, ASIC design and fabrication requires a significant initial engineering investment, making the option impractical for low-cost or low-volume applications, and ASICs are impossible to reconfigure once manufactured.

Recent improvements in field-programmable gate array (FPGA) technology and design tools have introduced a new option for DSP applications that require high performance and low development costs. An FPGA is a chip

containing logic fabric in the form of gates, multipliers, logic units, and programmable interconnects. This logic fabric can be configured via high-level design tools or hardware description languages (HDLs) to implement virtually any digital system. Although FPGAs have existed for some time, they have only recently become a viable option for DSP. In the past, these chips required programming using hardware description languages or weak design tools, making the implementation of complex DSP operations, such as transforms and filters, clumsy and tedious. The newest generation of design tools offers libraries of common DSP functions, enabling developers to implement something as complex as a Fast-Fourier Transform (FFT) without writing a single line of code.

The project focuses on audio DSP for its practical applications, manageable data rates, and tangibility in terms of testing and demonstration. Specifically, speech pitch shifting is studied as the DSP task of interest. Pitch shifting is a special effect in which the pitch of a speech or musical audio signal is shifted higher or lower while the duration of the signal remains constant. Applications of pitch shifting include high-end CD players, audio editing software, voice disguising tools, and toys. Furthermore, speech processing was selected as the DSP task because of its current relevance in the communication and multimedia fields.

The purpose of this design project is to implement a real-time speech pitch shifter on an FPGA. Equal focus is placed on developing an effective solution to the non-trivial task of audio pitch shifting and developing a hardware implementation using high-level design tools. We are working with FPGAs in order to gain experience in an area that is attracting significant research and development interest in both academia and industry. We analyze existing pitch shifting techniques and propose two effective algorithms well-suited for FPGA hardware implementation. We then analyze Xilinx System Generator, a leading FPGA design tool for DSP applications, and propose a hardware implementation solution. We present conclusions on the current state of FPGA design tools for DSP and the outcomes of this project.

### *B. Objectives*

The project equally emphasizes pitch shifting DSP techniques and hardware implementation. The project has several objectives in terms of educational aspects and deliverables:

- The students should gain experience in FPGA development.
- Several pitch shifting techniques should be explored and simulated.
- A suitable pitch shifting algorithm should be implemented on hardware and tested for high-fidelity performance.

## II. Design Overview

This part of the report details the entire design process. Section A defines the project specifications. Sections B, C, and D present three iterations respectively of the DSP technique design process. For each technique, we present relevant background and research, a proposed algorithm, and an analysis of the technique's effectiveness and feasibility as a solution based on MATLAB simulation. Section E presents an overview of the FPGA design tools used and progress towards implementation.

### *A. Specifications*

The project deliverable will adhere to the following minimum specifications:

- High-fidelity audio format (44.1 KHz sampling at 16-bits);
- Harmonic-preserving pitch shifting; and
- Algorithm efficiency adequate to enable real-time operation on supplied hardware.

Ideally, the project was intended to fulfill the following design goals (achievements and progress is discussed in the report conclusion):

- Stand-alone FPGA operation;
- Audio D/A and A/D conversion performed on development board; and

- Exploration and analysis of several pitch shifting techniques.

## B. *Frequency Shifting*

The simplest way to shift the frequency content of an audio signal is by single-sideband (SSB) modulation. This process entails eliminating the negative frequency content of a signal, modulating the positive frequencies by multiplication with a complex sinusoid, and finally reconstructing the real signal. Elimination of the lower sideband prevents frequency content from switching sidebands during modulation. Multiplication with a complex sinusoid (as opposed to a real-valued sinusoid) is used to avoid imaging issues introduced by standard modulation.

The lower sideband of a real signal can be removed using the Hilbert transform, a type of all-pass filter easily approximated in hardware or software as a finite-length FIR filter. For any signal $g(t)$, its Hilbert transform, denoted $\hat{g}(t)$ is defined as follows:

$$\hat{g}(t) = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{g(\tau)}{t-\tau}d\tau \qquad (1)$$

For the case of a real input signal, the transform outputs only the upper sideband, producing a signal subsequently referred to as the analytic signal [1]. Appendix A includes a derivation of how we can achieve SSB modulation using the Hilbert transform yielding the following result:

$$\mathrm{Re}\left\{e^{j\omega_c nT_s}x_p[n]\right\} = \cos(\omega_c nT_s)x[n] + \sin(\omega_c nT_s)\hat{x}[n] \qquad (2)$$

where $x_p[n]$ is the analytic signal, $\omega_c$ is the amount of frequency shift in radians per second, $x[n]$ is the input signal, and $T_s$ is the sampling period.

This relatively simple technique lends itself well to hardware implementation. Figure 1 shows a block diagram of the frequency shifter using operations that can be easily implemented using Simulink-based FPGA design tools. A filter is added before the input to of the Hilbert transformer to prevent frequency content at the ends of the spectrum from aliasing when modulated. The filtered signal is then branched to the Hilbert transform filter and a delay line equal in length to the Hilbert filter to ensure that the signal will recombine in phase.

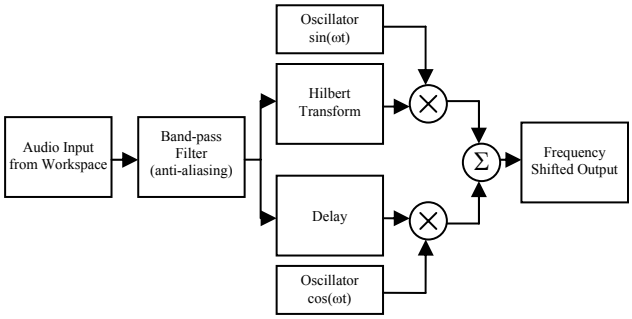


**Figure 1.** Block diagram of SSB frequency shifter

To evaluate the effectiveness of SSB modulation as a pitch shifting solution, we have simulated the algorithm in MATLAB for qualitative and quantitative analysis. As an initial test, classical music is modulated up in frequency by 100 Hz. Qualitatively, it is immediately evident that the algorithm dramatically changes the sound of the input. The original warm sound of the music becomes metallic and dissonant. While the pitch is audibly higher, the shift introduces significant harmonic distortion. Figure 2 shows the signal frequency spectrum before and after the modulation.

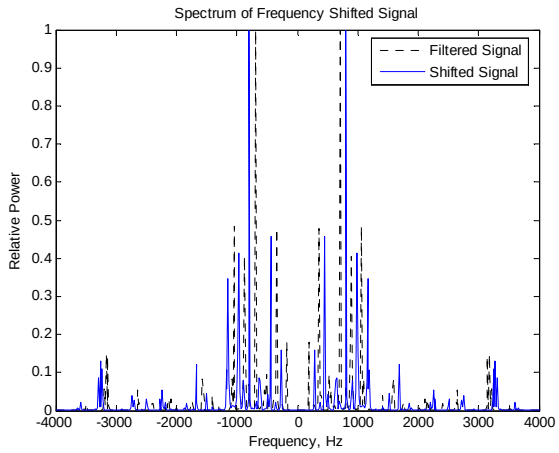


**Figure 2.** Signal spectrum before and after SSB modulation

The output spectrum confirms that the input is linearly shifted along the frequency axis. That is, each frequency component is increased or

decreased by an additive constant, 100 Hz in this example. Linear frequency shifts have many applications; however, human perception of sound relies on the harmonic relationship between frequency components. Modulation does not preserve this harmonic relationship, which results in the perceived degradation of sound quality. Speech input demonstrates that this shifting technique is less problematic for non-musical audio signals. Nonetheless, SSB modulation is certainly not an ideal pitch shifting solution.

*C. Time-Domain Pitch Shifting*

A variety of time-domain pitch shifting algorithms have been developed, the simplest of which involve a modification of the original sampling rate or merely playing back the signal faster or slower. Despite the audible change in pitch, these methods do not represent a true pitch shift because the signal duration is not maintained. The method described in [2] suggests a computationally efficient algorithm for altering the pitch and the length of digitally sampled sounds while maintaining the important characteristics of the original.

It is important to note that we have used our own edited version of the algorithm described in [2], both of which will be described in this section. First, we will describe the original technique, then our proposed modifications. Both algorithms can be divided into three sections: a pitch detector, a time compressor/expander, and a pitch shifter.

The pitch detector dynamically determines the fundamental frequency of the input sound using a crude but effective method. The input signal is passed through a fourth-order band pass IIR filter with a center frequency of approximately 440 Hz and a bandwidth of 200 Hz. The output of this filter is expected to be a single-tone sinusoid representing the fundamental frequency whose period is determined using a zero crossing detector. Since one period of this sinusoid is supposed to cross zero only twice, the period length (and thus the fundamental pitch) can be determined from the location of these zero crossings.

The time compressor/expander will be used to change the duration of the signal. Once the period of the fundamental frequency is identified, a window of the same length is applied to identify the waveform for each cycle. If the desired pitch shift (i.e. an increase in pitch) requires the signal duration to initially be lengthened, individual periods of the input can be repeated in the output signal. To shorten the signal duration (to achieve a decrease in pitch), periods of the signal can be discarded.

The advantage of this pitch shifting method is that the overall spectral envelope doesn't change; hence, we achieve a high quality output. If this algorithm is implemented as it is described, it will shift the pitch while introducing some high frequency artifacts that are primarily due to the discontinuities that would occur from the phase mismatch and also by the choice of the rectangular window used to extract an individual cycle from the input. One can clearly see that if the window is delayed by the right amount we can extract the next period. In other words, the original signal can be recreated by pasting these periods together, while a pitch shifted version of the input can be created by introducing a delay or an overlap between the periods before recombining. When using a rectangular window, we are multiplying the signal by another wave of magnitude equal to one over one period and zero elsewhere. The effect of this window on the signal can be seen by looking at the Fourier transform of the windowed signal that shows a clear harmonic relationship distortion and the presence of some new high frequency components due to the "lumpy" interpolation between the signal's harmonics that occur from rectangular windowing.

To rectify this problem, a Hanning window will be used. Figure 3 shows this window in the time and frequency domain. We can see that the lumpiness in the frequency domain of the windowed signal is decreased and that the interpolation between the harmonics of the original signal is much smoother. This smoothness is especially evident in the higher frequencies. The spectrum of the pitch shifted signal using the Hanning window will show a

very negligible contribution of higher frequencies artifacts.

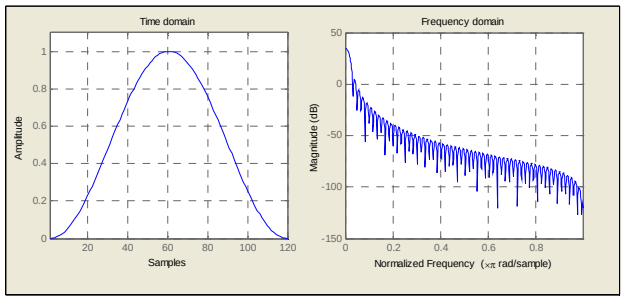


**Figure 3.** Hanning window in both time and frequency domain

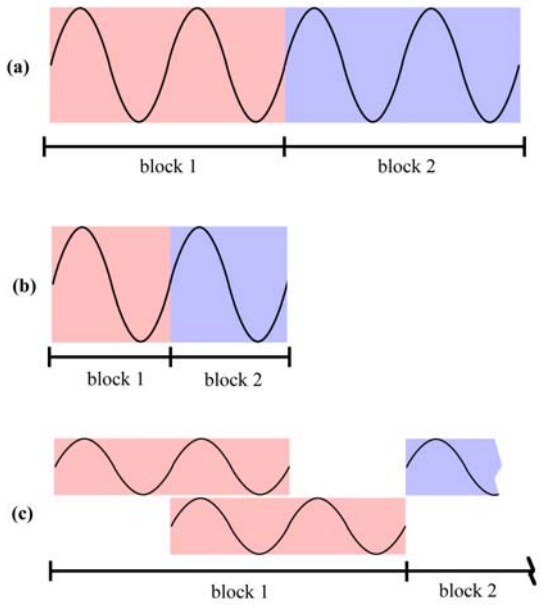


**Figure 4.** Block-based time domain pitch shifting technique; (a) shows input signal split into blocks; (b) for a down shift, input blocks are truncated by an integer number of periods to create a signal of shorter duration; (c) for an up shift, input blocks are overlapped and added to increase signal duration

With the goal of hardware implementation in mind, we have made a few changes to each section of the previously described algorithm for added simplicity. The original pitch tracker dynamically tracks the pitch in each period of the input signal using a band pass filter and a zero-crossing detector. We decided to discard the use of a pitch tracker and replace it with an assumption based on information from [3] specifying the fundamental frequency, $f_0$, ranges for males and females. The typical value of $f_0$ for a male lies between 80 Hz and 200 Hz, and for females it is between 150 Hz and 350 Hz. Since 200 Hz seems to be a common $f_0$ for both males and females, we decided to assume the input signal has a constant pitch of 200 Hz (or a period of 5 ms), bearing in mind that this will introduce some error. Because speech is considered short-time stationary, or periodic over very short periods of time, the error may be small enough to maintain acceptable fidelity. The time compressor/expander is replaced by a process that divides the input signal into data blocks of a length selected to be an integer multiple of the assumed speech period. The block processing approach is employed to reduce computational burden. In order to lower the pitch, we discard an integer number of periods from each data block before transferring it to the output, as shown in Figure 4 (b). The resulting output signal is a shorter version of the input. In order to raise the pitch, the data block is added to a delayed replica of itself, as shown in Figure 4 (c), before transferring it to the output, resulting in an expanded version of the input. At this stage, the input signal has either been expanded or compressed in time to create a signal with a different duration but with the same frequency content.

Since our goal is to playback a signal with shifted pitch, the output vector from the previous stage must be resampled to restore the original signal length. Fractional resampling needs to be employed in our case; it consists of interpolation followed by decimation. Figure 5 shows the spectrum of a 1024-sample window taken from the original signal compared to the spectrum of the output signal after raising the pitch. It is clear from those plots that the spectrum did undergo a shift and that some higher frequency components were introduced. This part replaces the pitch shifter section of the algorithm described in [2] and it simply rebuilds a continuous waveform from the current samples and then samples it at the new rate, when the new samples are played at the original rate the audio clip will sound either faster or slower.

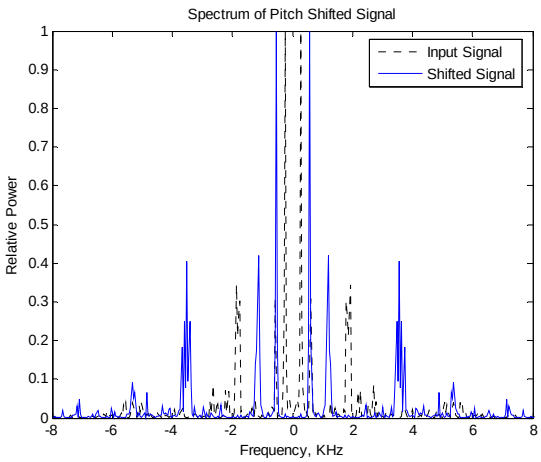


**Figure 5.** Input and output signal frequency spectrum

*D. Frequency-Domain Pitch Shifting*

Like the time-domain technique, the frequency-domain technique is based on shifting small overlapping windowed blocks of data in time and resampling. To shift down in pitch, the overlapping data blocks are shifted closer together to create a signal with the same pitch but shorter duration. The signal is then resampled, expanding to the original duration and stretching the signal in time to achieve a decrease in pitch. To raise the pitch, data blocks are spread further apart in time to create a signal longer in duration. Again, the signal is resampled to restore the original duration, which compresses the signal in time to achieve an increase in pitch.

The frequency-domain technique continues with the assumption that speech is short-time stationary, that is, periodic with relatively constant frequency components over small ranges of time. As such, adjacent blocks will effectively have the same frequency content. However, when these blocks are shifted in time, the block transitions will not be in phase. The resulting phase discontinuities introduce noise, just as we saw in the purely time-domain technique. However, by utilizing frequency domain information, the phase discontinuities can be eliminated.

The phase vocoder algorithm offers an effective solution to the problem of phase discontinuity [4]. It is considered a frequency-domain technique because it utilizes the Short-Time Fourier Transform (STFT), a common audio processing tool that involves taking the Discrete Fourier Transform (DFT) of short, periodic blocks of an audio signal. By properly modifying the phase terms of the STFT and re-synthesizing the time-domain data, it is possible to match the phases of each frequency component across block transitions. Figure 6 demonstrates the phase discontinuity problem introduced by shifting overlapping blocks.

In addition to modeling audio as stationary and periodic over small blocks, the phase vocoder makes several other assumptions. Namely, sound can be modeled as a sum of sinusoids, and each frequency bin in the STFT of an audio block contains no more than one sinusoidal component. In other words, the algorithm operates as if any non-zero frequency bin magnitude is caused a single tone corresponding to the range of frequencies spanning that bin. These assumptions allow us to represent each STFT frequency bin as a single sinusoidal component of the signal. We can then modify the phase offset associated with the sinusoid to ensure continuity when added to the same component of the previous block.

We can summarize the phase vocoder algorithm with the following steps:

1. Calculate the STFT of overlapping signal blocks
2. Modify the phase of each STFT bin
3. Re-synthesize (IFFT) the block and apply the appropriate time shift (see Figure 7)
4. Resample block to restore original duration

To determine how to modify the STFT phase terms, we must derive a phase propagation formula that will accurately predict how the phases of each frequency bin will progress in time. There are several factors that will effect phase propagation: (1) block size; (2) amount of time shift required following re-synthesis; and (3) actual frequency of bin sinusoid (not necessarily the center bin frequency).

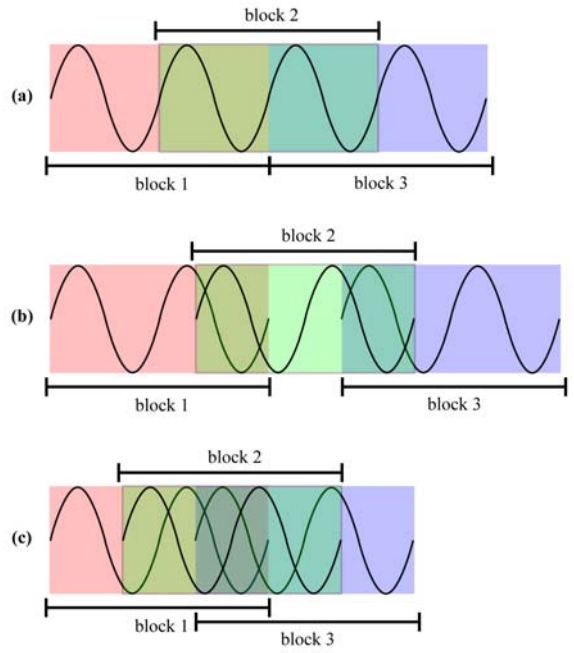


**Figure 6.** Time shifting of overlapping blocks; (a) depicts an input signal split into 3 overlapping blocks; (b) blocks are shifted forward in time to increase signal duration; (c) blocks are shifted back in time to decrease signal duration

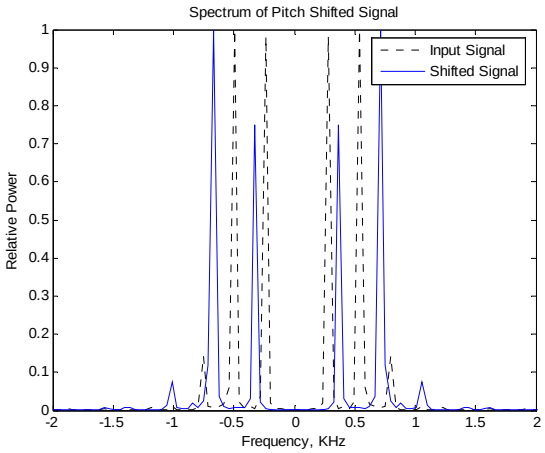


**Figure 7.** Input and shifted frequency spectrum using phase vocoder. Note slight noise introduced around 1.5KHz and higher.

Many implementations of the phase vocoder have been developed in audio processing literature. We will use the symbols and phase propagation formula proposed by Loroche and Dolson [5].

The process of calculating STFT's of signal blocks will be referred to as the analysis step, while the inverse will be called the synthesis step. During the analysis step, the $u^{th}$ block will start at time $t_a^u = uR_a$, where $R_a$ will be called the analysis hop factor and represent the size of the block. Using lower-case for time-domain signals and upper-case for frequency-domain representations, $x(n)$ and $y(n)$ will be the discrete-time input and output signals respectively. Both analysis and synthesis steps will utilize windows, $h(n)$ and $w(n)$ respectively. We can now define the STFT analysis step and the time domain output following the synthesis step:

$$X(t_a^u, \Omega_k) = \sum_{n=-\infty}^{\infty} h(n)x(t_a^u + n)e^{-j\Omega_k n} \tag{3}$$

$$y(n) = \sum_{u=-\infty}^{\infty} w(n - t_s^u)y_u(n - t_s^u) \tag{4}$$

where,

$$y_u(n) = \frac{1}{N}\sum_{k=0}^{N-1} Y(t_s^u, \Omega_k)e^{j\Omega_k n}\,. \tag{5}$$

Time scaling is effected by shifting the output blocks, such that $t_s^u = uR_s$ when $R_a \neq R_s$. We are left only to define the phase propagation equation by which the phase of the output STFT, $Y(t_s^u,\Omega_k)$, will be altered. Omitting the full derivation, Laroche and Dolson [5] yield the following three equations, (8) representing phase propagation:

$$\Delta\Phi_k^u = \angle X(t_a^u, \Omega_k) - \angle X(t_a^{u-1}, \Omega_k) - R_a\Omega_k \tag{6}$$

$$\hat{\omega}_k(t_a^u) = \Omega_k + \frac{1}{R_a}\Delta_p\Phi_k^u \tag{7}$$

$$\angle Y(t_s^u, \Omega_k) = \angle Y(t_s^{u-1}, \Omega_k) + R_s\hat{\omega}_k(t_a^u)\,. \tag{8}$$

Result (6) estimates phase propagation based on the center bin frequency (i.e. $\Omega_k = k/N$). The actual frequency of the sinusoid contained in the $k^{th}$ bin is estimated by (7) based on the difference in bin phase propagation rate between blocks. We have implemented a phase vocoder according to these results. MATLAB code is included as Appendix D. The final step, resampling, is

performed in the same way as the time-domain technique.

Simulations demonstrate that most of the noise produced by the time-domain pitch shifting technique is eliminating by altering the phases of the STFT. Figure 7 depicts the input and output signal spectrums. However, the frequency-domain method, as described here, is far from perfect. Notably, our simulation suffers from the typical "distance" and "phasiness" issues commonly associated with the phase vocoder. [5] proposes several methods for correcting these phase errors, but they are out of the scope of this project since the simulation produced results of acceptable fidelity.

*E. Hardware Implementation*

The software interface chosen for programming the FPGA was the MATLAB Simulink environment and Xilinx System Generator. Simulink is a real-time simulation platform that provides a comprehensive set of "blocks" that can be used to model a particular system. Each blockset contains a number of elements representing filters, converters, sources, scopes, or other devices that can be interconnected. System Generator, a high-level design utility marketed by Xilinx for high-performance DSP systems, includes additional blocksets specific to Xilinx FPGAs. These blocksets allow for very abstract system design and modeling, and can be used to automatically generate HDL netlists that can be downloaded to an FPGA. This development system greatly simplifies the design process and can drastically reduce the time required when compared to conventional HDL coding.

Another advantage is the ability of System Generator to run a hardware co-simulation. After a design has been laid out in blocks, it can be compiled to a single co-simulation block. A source generated on the computer by Simulink can be sent to the FPGA, processed, and then returned to Simulink for analysis in real-time. In this way, a design can actually be tested with hardware in-the-loop, with input and output from MATLAB, to ensure that it is working [6].

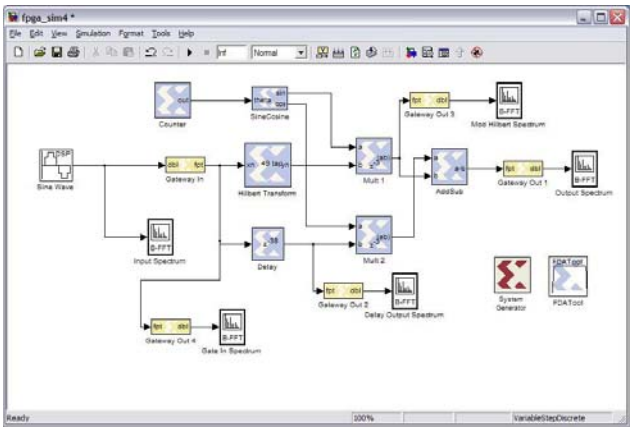

**Figure 8.** Simulink Schematic of a Frequency Shifter

The original SSB modulation frequency shifting algorithm was the first to be designed and laid out in Simulink with the System Generator blocks and is shown above in Figure 8. As can be seen, it is quite similar to the basic diagram of Figure 1, illustrating the advantage of highly abstract programming. Connecting the Xilinx blocks to those provided by Simulink are gateways which provide the conversion from the floating point numerical representation used by MATLAB to the fixed point format which is used by the FPGA. This allows for the easy attachment of input/output Simulink blocks to examine the operation of every aspect of the design. The ability to test in this fashion makes this a very attractive approach.

The design was then compiled and downloaded to a Spartan3 FPGA as a hardware co-simulation on a Digilent development board, provided free of charge by Xilinx. This particular board, however, lacked the required I/O for stand alone operation, and the Spartan3 was insufficient for the extent of the project. (A low pass anti-aliasing filter had to be removed for the design to fit the FPGA.) At the start of the Fall 2005 semester, Xilinx provided a new Digilent development board (the XUPV2P), equipped with an AC-97 audio codec and an audio amplifier which performs 18-bit analog-to-digital and digital-to-analog conversion at rates up to 48kHz, as well as microphone and speaker connectors. The onboard FPGA is a Virtex-II Pro, and with 88 multipliers, and over 20,000 logic cells, it

provides ample capabilities for pitch shifting purposes [7].

*Implementation of Time Domain Algorithm*

After analyzing the two pitch shifting algorithms, the time domain approach was selected for its relative simplicity and reasonably high quality output. Operating in the frequency domain, while certainly feasible, would nonetheless be far more complicated, and it was decided that it could be attempted as the next step after successful implementation of the time domain algorithm.

The challenge of designing this system is that the convenience of MATLAB functions is no longer available and that everything must be built using digital logic elements. Additionally, the system must operate in real-time with little temporal lag. Since the method to shift up is different from shifting down, it is also necessary to create different components to address both cases. Figure 9 shows the Simulink schematic of a working down pitch shifter, with windows of size W truncated by 25%. The real-time specification of the pitch shifter requires that the input and output operate at the same sample time. As such, the primary component of this design is the Xilinx Dual Port RAM block, which allows the simultaneous reading and writing of data to and from a memory bank at two different rates. As an input streams into the system, each window of data is stored sequentially into memory, and each subsequent window replaces the former. At the same time, the first 75% of the data points are read from the memory at a slower rate such that both read and write operators finish at the same time. In this fashion, the incoming and outgoing memory data are perfectly synchronized but at different data rates. The output of this block is then passed through an interpolator followed by a decimator. These cascading elements are necessary to achieve the fractional interpolation required by the algorithm. The final output is a stream of data, shifted lower in pitch but possessing the same sample time.

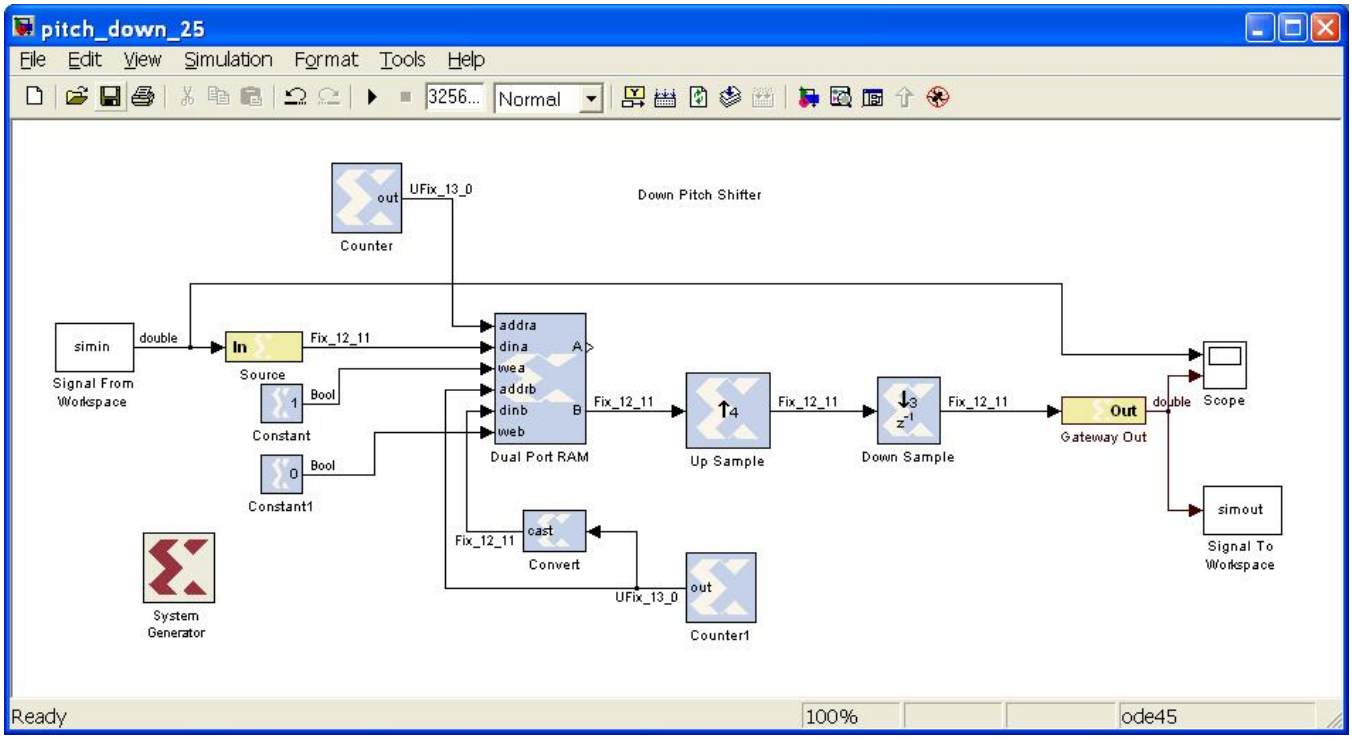


**Figure 9.** Simulink Schematic of a Down Pitch Shifter

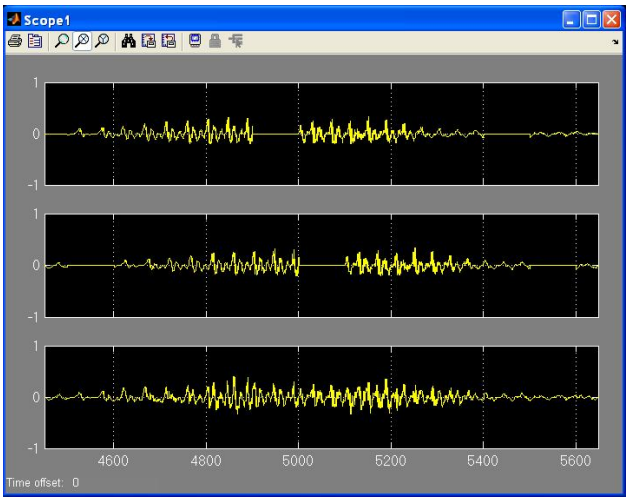


**Figure 10.** Output of the up pitch shifter. The first waveform shows windowed excerpts from an input, padded in between with zeros. The second waveform is a shifted version, with each window overlapping its counterpart by 75%. The third is the down sampled addition and final output.

The implementation of an up shifter (Figure 11) is similar to the down shifter, but requires a few additional elements. Again, a Dual Port RAM block is used, but this time the incoming data is written at a rate slower than the rate at which it is read. Once all data points have been read, zeros are padded at the end of the data stream until the next window has finished being written, and the process begins again. In this way, synchronization is maintained. The windows have now been separated from each other with zeros inserted in between (Figure 10). The data is then sent to an addition block where it is added with a delayed version of itself. For a window sized $W$ and a fractional overlap of $x$, the delay is equal to $W(1-x)$. Following this is an interpolation block and decimation block which convert the signal back to its original sample time.

### *Software/Hardware Issues*

Using Simulink, working simulations of the up and down time domain pitch shifter have been realized. Audio files stored in the .wav format can be processed and written back and the resulting audio file is of the same duration but different pitch. Due to the limitations of Simulink, a real-time demonstration with microphone and speaker cannot be performed; however, this design is perfectly capable of operating in real-time once it has been implemented on the FPGA development board.

The hardware implementation of this system has not yet been realized due to unforeseen difficulties with the software and hardware. The

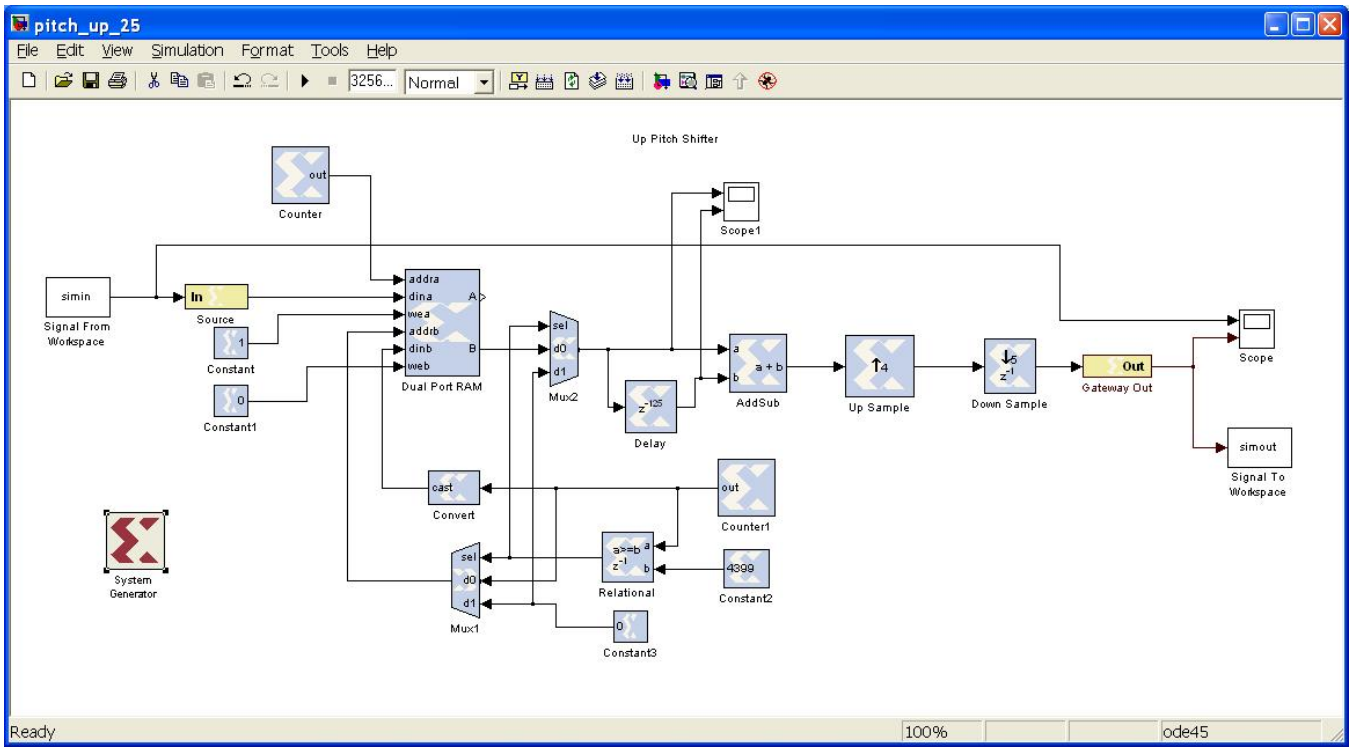


**Figure 11.** Simulink Schematic of an Up Pitch Shifter

beginning of the Fall 2005 semester, Villanova University issued new laptop computers on which we could operate the Xilinx development software. This was not anticipated to be problematic; however, the installation of the software was plagued with errors and proved to be exceedingly time consuming. A greater problem encountered with System Generator was its lack of support for I/O devices. Our original understanding was that the entire design could be built in Simulink using System Generator without having to do any manual coding in VHDL. A Xilinx tutorial offered different methods for incorporating the audio I/O, however, none worked properly, even with the sample designs included with the tutorial. The development board, having recently been released, was poorly supported by the software, and documentation offered little insight into how the operate audio. Ultimately, more time would be required to further research and learn how to utilize the audio hardware on the board before the pitch shifter can be completed.

## III. Project Management

### *A. Schedule & Personnel*

The objective for the Spring 2005 semester was to perform initial research into FPGAs as well as frequency and pitch shifting techniques. Frequency shifting with SSB modulation was successfully implemented as a hardware co-simulation on a Spartan 3 FPGA. Additionally, efforts were made to learn more about the development tools and a project webpage was created (http://homepage.villanova.edu/scott.sawyer/fpga). Work during this semester was generally a combined group effort.

For most of the Fall 2005 semester, each member of the group focused on a particular area. Scott Sawyer researched frequency domain approaches to pitch shifting while Habib Estephan concentrated on pitch shifting in the time domain. Dan Wanninger focused on installing and learning the software and hardware. MATLAB simulations were created for both pitch shifting approaches and the time domain algorithm was selected to be implemented on the FPGA. Unfortunately, because of poor I/O support in the Xilinx development tools, more time was required than what was available. Completion of this project would require roughly an additional month.

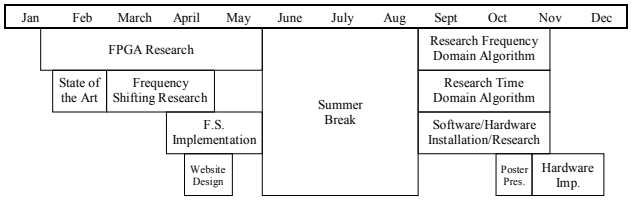


**Figure 12.** Project Schedule Chart

### *B. Budget*

The material and labor required to complete this project include the Xilinx FPGA and development board, development software and tools, and design labor.

Development labor would contribute significantly to production costs. Based on the time invested by our group toward technical research and progress, it is reasonable to estimate that project development would take 200 engineering hours to research and develop this project. Billable at $150 per hour, labor cost would be $30,000.

| | |
|---|---|
| Xilinx Virtex-II Pro Development Board | $1,600 |
| MATLAB with Simulink | $2,000 |
| Labor | $30,000 |
| **Industrial Cost:** | **$33,600** |

**Figure 13.** Estimated Industrial Development Budget

Although industrial development of a hardware pitch shifter implemented on an FPGA may cost upwards of $33,600, there was no cost associated with this project. The College of Engineering provided educational licenses for MATLAB with Simulink, as well as a DSP lab equipped with MATLAB workstations available for use. The FPGA development boards and software were provided by Xilinx

## IV. Conclusions

### A. Summary of Achievements and Deliverables

Due to unforeseen software and hardware issues which resulted in the need for additional time, the final objective of a stand alone unit was not met. However, while pitch shifting was not achieved on the FPGA in real-time, it was successfully implemented in MATLAB with two different approaches, and the SSB modulation frequency shifter was implemented on the FPGA and run through System Generator using hardware co-simulation. Both the time domain and frequency domain algorithms that were developed are capable of high-quality pitch shifting both up and down and can read and write to an audio wave file. Successful implementation of one of these methods on the FPGA development board is not out of reach, but would require an additional time investment.

To summarize, the project has yielded several deliverables:

- Frequency shifting and pitch shifting algorithms including related mathematical derivations;
- MATLAB implementations (.m files) of the frequency shift algorithm and the two pitch shift algorithms;
- System Generator system models (.mdl files) of the frequency shifter and time-domain pitch shifter;
- Xilinx development environments installed, configured and deployed in an engineering lab; and
- A successful implementation of the frequency shifter running on the FPGA in hardware co-simulation mode.

All MATLAB and Simulink models and simulations are available for download on the project website at the following address:
http://homepage.villanova.edu/scott.sawyer/fpga

### B. Project Assessment

As a final assessment, though we did not meet the original project objective of hardware implementation with microphone input and speaker output, several deliverables were successfully produced. Realization of a stand alone unit was impaired by software issues and poor support for the development board, but recently released versions of Xilinx System Generator and improved board support should soon facilitate easy deployment of our existing Simulink simulations.

## References

[1] Smith III, Julius O. “Analytic Signals and Hilbert Transform Filters.” <http://ccrma-www.stanford.edu/~jos/r320/Analytic_Signals_Hilbert_Transform.html>

[2] Lent, Keith. “An Efficient method for pitch shifting Digitally sampled sounds.” *Computer Music Journal*, vol. 13, no. 4, pp. 65-71, 1989.

[3] Filipsson, Marcus. “Speech Analysis Tutorial.” <http://www.ling.lu.se/research/speechtutorial/tutorial.html>

[4] Bernsee, Stephan M. *The DSP Dimension*. <http://www.dspdimension.com/data/html/timepitch.html#pvocform>

[5] Loroche, Jean and Dolson, Mark. “Improved Phase Vocoder Time-Scale Modification of Audio.” *IEEE Trans. On Speech and Audio Processing*, vol. 7, no. 3, May 1999.

[6] “System Generator for DSP.” *Xilinx Inc*. 2006. 10 February 2006. <http://www.xilinx.com/ise/optional_prod/system_generator.htm>

[7] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. Xilinx Inc. 10 October 2005. v4.5.

## APPENDIX A: SSB MODULATION DERIVATION

We will begin by reintroducing the Hilbert Transform. The operation can be applied to any real signal and results in a 90 degree phase shift of every sinusoidal component. Recall from (1), for any signal $g(t)$, its Hilbert transform is denoted $\hat{g}(t)$.

Conceptually, the Hilbert transform is nothing but a convolution of the signal $g(t)$ with the time function $(1/\pi t)$ [A. 1]. The convolution in time of two signals is equivalent to the multiplication of their Fourier transforms in the frequency domain. Consider $G(f)$, the Fourier transform of the signal, and the corresponding Fourier transform of the time function $(1/\pi t)$ is defined below:

$$\Im\left(\frac{1}{\pi t}\right) = -j\,\mathrm{sgn}(f) \tag{A. 1}$$

where sgn($f$) is called the signum function and is defined as follows:

$$\mathrm{sgn}(f) = \begin{cases} 1, & f > 0 \\ 0, & f = 0 \\ -1, & f < 0 \end{cases} \tag{A. 2}$$

Now the Fourier transform of $\hat{g}(t)$ becomes,

$$\hat{G}(f) = -j\,\mathrm{sgn}(f)G(f) \tag{A. 3}$$

To recap, in order to obtain the Hilbert transform of a signal we need a linear filter whose frequency response is equal to $-j$ sgn($f$), which will result in a +90 degree shift for all negative frequencies and a -90 degree shift for all positive frequencies. The amplitudes of all frequency components are not affected by this type filter, so all the frequencies are passed with an equal gain of 0 dB.

With the Hilbert transform defined and its effect on a signal explained, the transform can be applied to SSB modulation. The purpose of the Hilbert transform is to remove a sideband (that is, either the positive or negative frequency components) of a signal. In the case where only positive frequencies are passed, the resulting single-sideband signal is referred in the time domain as the analytic signal. For every real signal there is a corresponding analytic complex signal that is obtained with the Hilbert transform using the following formula:

$$a(t) = g(t) + j\hat{g}(t) \tag{A. 4}$$

To see why that works, consider the effect of this addition in the frequency domain:

$$A(f) = G(f) + j\hat{G}(f) \tag{A. 5}$$

where $\hat{G}(f)$ is defined in (A. 3). Substituting (A. 3) into (A. 5),

$$A(f) = G(f) + \mathrm{sgn}(f)G(f) \tag{A. 6}$$

from which we can directly conclude:

$$A(f) = \begin{cases} 2G(f), & f > 0 \\ G(0), & f = 0 \\ 0, & f < 0 \end{cases} \tag{A. 7}$$

Thus, the analytic signal has a spectrum consisting of just positive frequencies whose magnitude is amplified by a factor of two. The factor of two can be eliminated by making the filter's magnitude response equal to one-half instead of one. The analytic signal can now be shifting up or down in the frequency domain by a time domain multiplication with a complex exponential. Because negative frequencies have been eliminated, no distortion is caused by negative frequencies crossing over into the positive half of the spectrum. However, the remaining modulated signal is still complex.

$$e^{j2\pi f_c t}a(t) = e^{j2\pi f_c t}g(t) + je^{j2\pi f_c t}\hat{g}(t) \tag{A. 8}$$

It is readily shown that a SSB can be reflected in the frequency domain by simply taking the real part of the sideband. For proof, consider the

complex conjugate symmetry in the DTFT of a discrete-time signal, $x[n]$. Let $X_p(e^{j\pi})$ be the positive part of the DTFT $X(e^{j\pi})$,

$$X(e^{j\omega}) = X_p(e^{j\omega}) + X_p^*(e^{j\omega}) \quad \text{(A. 9)}$$

According to the conjugate property of the DTFT,

$$x^*[n] \xleftrightarrow{\Im} X^*(e^{-j\omega}) \quad \text{(A. 10)}$$

We can take the inverse DTFT of (A. 9) using the conjugate property (A. 10) and linearity,

$$x[n] = x_p[n] + x_p^*[n] \quad \text{(A. 11)}$$

where $x_p[n]$ is the inverse DTFT of $X_p(e^{j\pi})$. Adding a signal to its conjugate cancels the imaginary part, and we can conclude,

$$x[n] = 2\,\mathrm{Re}\{x_p[n]\} \quad \text{(A. 12)}$$

Now sample the modulated signal from (A. 8) to yield the discrete-time case,

$$e^{j2\pi f_c nT_s} x_p[n] = e^{j2\pi f_c nT_s} x[n] + je^{j2\pi f_c nT_s} \hat{x}[n] \quad \text{(A. 13)}$$

Applying the result from (A. 12) to reflect (A. 13) in the frequency domain,

$$\mathrm{Re}\{e^{j2\pi f_c nT_s} x_p[n]\} = \cos(2\pi f_c nT_s)x[n] + \sin(2\pi f_c nT_s)\hat{x}[n] \quad \text{(A. 14)}$$

which is equivalent to result (2) used in our SSB modulation algorithm.

## REFERENCE

[A. 1] Haykin, Simon. *Communication Systems*, Fourth Edition.

## APPENDIX B: FREQUENCY SHIFTING MATLAB SIMULATION

```
clear; clc;

%%% Define input signal and settings

load hilbertfilter;
input = wavread('1min.wav');
fs = 8000;                              % Audio sampling rate
Ts = 1/fs;                              % Audio sampling period
winlength = 3;                          % Window size in seconds
winstart = 20;                          % Location (in seconds) to begin
hilorder = 64;                          % Order of Hilbert FIR filter
hildelay = 32;                          % Hilbert filter delay/latency
bpforder = 64;                          % Order of Bandpass Filter
bpfdelay = 32;                          % BPF delay/latency
shift = 100;                            % Desired frequency shift (in Hz)
fftorder = 500;                         % FFT length for signal analysis

%%% Prepare and analyze audio input

window = input(winstart*fs+1:winstart*fs+winlength*fs);
len = length(window);
X1 = fft(window, fftorder);
spec1 = abs(fftshift(X1)).^2';
spec1 = spec1./max(spec1);
freq = linspace(-fs/2, fs/2, fftorder);
stop1 = (abs(shift)*2)/(fs/2);
stop2 = ((fs/2)-abs(shift)*2)/(fs/2);
bandpass = fir1(bpforder, [stop1 stop2]);
bpwin_ = conv(window, bandpass);
bpwin = bpwin_(bpfdelay+1:bpfdelay+len);
bpHf = abs(freqz(bandpass, 1, fftorder/2)').^2;
bpneg = zeros(1,fftorder-length(bpHf));
for n=1:length(bpHf)
bpneg(length(bpneg)-(n-1)) = bpHf(n);   % Fold LPF frequency response
end
bpspec = [bpneg bpHf];
X2 = fft(bpwin, fftorder);
spec2 = abs(fftshift(X2)).^2';
spec2 = spec2./max(spec2);
figure;
plot(freq, spec1, 'r:'); title('Spectrum of Input Audio Signal');
xlabel('Frequency, Hz'); ylabel('Relative Power');
hold on;
plot(freq, bpspec, 'g--'); plot(freq, spec2, 'b');
legend('Original Signal','Bandpass Frequency Reponse','Filtered Signal');
hold off;

%%% SSB modulation using Hilbert filter

mhat = conv(bpwin,hilxformer);
Mr = bpwin';
Mi = mhat(hildelay+1:hildelay+len)';

% Create analytic signal and plot DFT

ssb = Mr + j*Mi;
X3 = fft(ssb, fftorder);
spec3 = abs(fftshift(X3)).^2';
spec3 = spec3./max(spec3);
hilHf = abs(freqz(hilxformer, 1, fftorder/2)').^2;
```

```
hilspec = [zeros(1, fftorder-length(hilHf)) hilHf];
figure;
plot(freq, spec2, 'r:');
title('Spectrum of Upper Sideband');
xlabel('Frequency, Hz');
ylabel('Relative Power');
hold on;
plot(freq, hilspec, 'g--');
plot(freq, spec3, 'b');
legend('Filtered Signal','Hilbert Frequency Reponse','Upper Sideband');
hold off;

%%% Modulate with "local oscillators"

n = 1:len;
sign = shift/abs(shift);
localosc = cos(2*pi*abs(shift)*n*Ts);
localosc90 = sin(2*pi*abs(shift)*n*Ts);
freqshift = localosc.*Mr - (sign)*localosc90.*Mi;
X4 = fft(freqshift, fftorder);
spec4 = abs(fftshift(X4)).^2';
spec4 = spec4./max(spec4);
figure;
plot(freq, spec2, 'k:');
title('Spectrum of Frequency Shifted Signal');
xlabel('Frequency, Hz'); ylabel('Relative Power');
hold on;
plot(freq, spec4, 'b');
legend('Filtered Signal','Shifted Signal');
hold off;

%%% To listen to your signals (at reduced volume):
% sound(window./2, fs); % orignal signal
% sound(bpwin./2, fs); % filtered signal
% sound(freqshift./2, fs); % frequency shifted signal
```

## APPENDIX C: TIME-DOMAIN PITCH SHIFTING MATLAB SIMULATION

### (UPSHIFTER.M)

```
clear
clc

Nin=wavread('freddie_mono.wav');%input wave
SampleFreq=44100;%Sampling Frequency of the input
windowsize=1200;%(100e-3)*SampleFreq;
overlap=0.25;                                %Specifies the amount of overlap between
                                             %two blocks before summing them
numsections=floor(length(Nin)/windowsize); %Specifies the total number of blocks in
                                             %the input speech signal
shiftval=round(windowsize*overlap);        %Calcultes the the amount by which we need to
                                             %shift our block before we add
                                             %it to itself in order to
                                             %achieve the desired overlap
Nin=Nin(1:numsections*windowsize);         %adjusts the length of Nin to make it an integer
                                             %multiple of the block size
Nin=Nin';%Input's transpose to make it a 1xM array
adjust=windowsize*(2*overlap-1)+1;          %adjustment variable
x=round(length(Nin)/windowsize);
Nout=(windowsize+shiftval)*x+(-shiftval);
Output=zeros(1,Nout+shiftval);              %Initiates an empty output variable

for n=1:round((length(Nin)/windowsize))

section=Nin(windowsize*n-(windowsize-1):windowsize*n);%.*hann(windowsize)';
                                             %Section contains the corresponding block
                                             %from the input vector
start=(windowsize+shiftval)*n+(1-windowsize-shiftval);
                                             %points to the insertion location of a block
                                             %in the output vector
last=(windowsize+shiftval)*n+(-shiftval);  %points to end location of a block in the output
Output(start:last)=section;
Output(start+shiftval:last+shiftval)=Output(start+shiftval:last+shiftval)+section;
end
Output=Output(1:last);
ratio=length(Output)/length(Nin);
[t,d]=rat(ratio);                           %returns the the ratio as a fraction,
                                             %where the Numerator is stored in t
                                             %and the denominator is stored in d.

Out1=resample(Output,d,t);                  %in order to play the outputed signal at the
                                             %same rate we have resample it at
                                             %a rate that d/t the current sampling rate
                                             %in order to rescale the output vector and
                                             %make it the same length as the
                                             %input
Sound(Out1,SampleFreq)
```

(DOWNSHIFTER.M)

```
clear; clc

Nin=wavread('freddie_mono.wav'); %input wave
Nin=Nin';                         %Input's transpose to make it a 1xM array
SampleFreq=44100;                 %Sampling Frequency of the input
windowsize=1800;                  %Specifies the Size of the block
estimperiod=(5e-3)*SampleFreq;    %Calculates the number of samples in 5 ms of signal
                                  %which is the typical period of human's
                                  %speech fundemental frequency
maxnumperiod=floor(windowsize/estimperiod)  %Calculates the total periods in one block
numperiod=2                       %Specifies the number of periods taken out of each block

if numperiod>maxnumperiod         %this if statement gurantees that the number of truncated
periods is less
                                  %than the total number of periods in that
                                  %specific block
    nunmperiod=maxnumperiod-1;
end
x=round(length(Nin)/windowsize);
Nout=floor(windowsize*x-numperiod*estimperiod);

Output=zeros(1,100);              %Defines the output vector that will contain the truncated
blocks
                                  %from the input vector
for n=1:round((length(Nin)/windowsize))
    start=windowsize*n+(1-windowsize)   %points to the beginning of the next block from the
input
    last=floor(windowsize*n-numperiod*estimperiod)  %points to the end of the next block from
the input
    section=Nin(start:last);      %Section will contain the current block through each
iteration
    l=length(section);            %Length of Section
    st=n*l+1-l;                   %points to the location of insertion of the truncated block
into the output vector
    lst=n*l;                      %points to the final location of the truncated block into the
output vector
    Output(st:lst)=section;       %Output vector
end
ratio=length(Output)/length(Nin);
[t,d]=rat(ratio);                 %returns the the ratio as a fraction, where the Numerator is
stored in t
                                  %and the denominator is stored in d.


Out1=resample(Output,d,t);        %in order to play the outputed signal at the same rate we
                                  %have resample it at
                                  %a rate that d/t the current sampling rate
                                  %in order to rescale the output vector and
                                  %make it the same length as the
                                  %input
Sound(Out1,SampleFreq)
```

## APPENDIX D: FREQUENCY-DOMAIN PITCH SHIFTING MATLAB SIMULATION

```
clear; clc;

%%% Define input signal and sampling rate

input = wavread('freddie_mono.wav')';    % input WAV
fs = 44.1e3;                             % input sampling rate

%%% Define global constants

alpha = 13/12;                           % pitch-shift factor
N = 512;                                 % frame length
overlap = .75;                           % overlap fraction
window = hanning(N)';                    % input window

%%% Calculate working variables

input_length = length(input);            % length of input signal
frame_count = floor((input_length-2*N)/(N*(1-overlap)));
                                         % number of frames in input
Ra = floor(N*(1-overlap));               % analysis time hop
Rs = floor(alpha*Ra);                    % synthesis time hop
Wk = (0:(N-1))*2*pi/N;                   % center bin frequencies
output = zeros(1, input_length*alpha);   % output signal initialization

%%% Process input frames

Xu_current = fft(window.*input(1:N));    % analyze initial frame
PhiY_current = angle(Xu_current);        % initial frame output phases

for u=1:frame_count
    Xu_prev = Xu_current;                % store last frame's STFT
    PhiY_prev = PhiY_current;            % store last frame's output phases
    Xu_current = fft(window.*input(u*Ra:u*Ra+N-1));
                                         % analyze current frame
    DPhi = angle(Xu_current) - angle(Xu_prev) - Ra*Wk;
                                         % unwrapped phase change
    DPhip = mod(DPhi+pi, 2*pi) - pi;     % principle determination (+/- pi)
    w_hatk = Wk + (1/Ra)*DPhip;          % estimated "real" bin frequency
    PhiY_current = PhiY_prev + Rs*w_hatk;
                                         % Phase propagation formula
    Yu = abs(Xu_current).*exp(j*PhiY_current);
                                         % output STFT
    output(u*Rs:u*Rs+N-1) = output(u*Rs:u*Rs+N-1) + real(ifft(Yu));
                                         % add current frame to output
end

norm_output = output./max(output);       % normalize the output amplitude
[t,d]=rat(alpha);                        % determine integer shift ratio
shifted = resample(output,d,t);          % resample for pitch shift

% sound(input,fs);
% sound(output,fs);
% sound(shifted,fs);
```