Voice Modulator

Adam Rosenfield
Lunduo Ye

6.111 Final Project
Spring 2007
TA: Amir Hirsch

Abstract

We designed, implemented, and tested a voice modulation system, which takes in audio data and modulates the pitch of the data. The modulator can change the pitch of vocal data while preserving vocal formants, maintaining intelligible speech over a wide range of frequencies. The system is implemented on a field-programmable gate array (FPGA) and operates in real time.

Table of Contents

List of Figures

1. Introduction [Adam and Lunduo]

The voice modulator changes the pitch of voice inputs. Users speak or sing into a microphone while playing keys on a keyboard. Real-time visualizations of the waveforms are displayed on a VGA screen. The modulator outputs frequency-shifted copies of the voice data to match the notes selected from the keyboard while preserving vocal formants as much as possible. This device allows users of any musical ability to sing notes or chords perfectly.

The modulator is implemented on a field programmable gate array (FPGA). Inputs are taken via a microphone and a PS/2 keyboard. A VGA monitor is used to display waveforms. MIDI keyboard support was originally planned; however, we could not get it to work in time.

Visualizations for the real-time Fourier transforms of the voice input were also not debugged in time.

The system has two main components, audio and video. Figure 1 shows a high-level overview of the inputs, outputs, and interactions between parts.

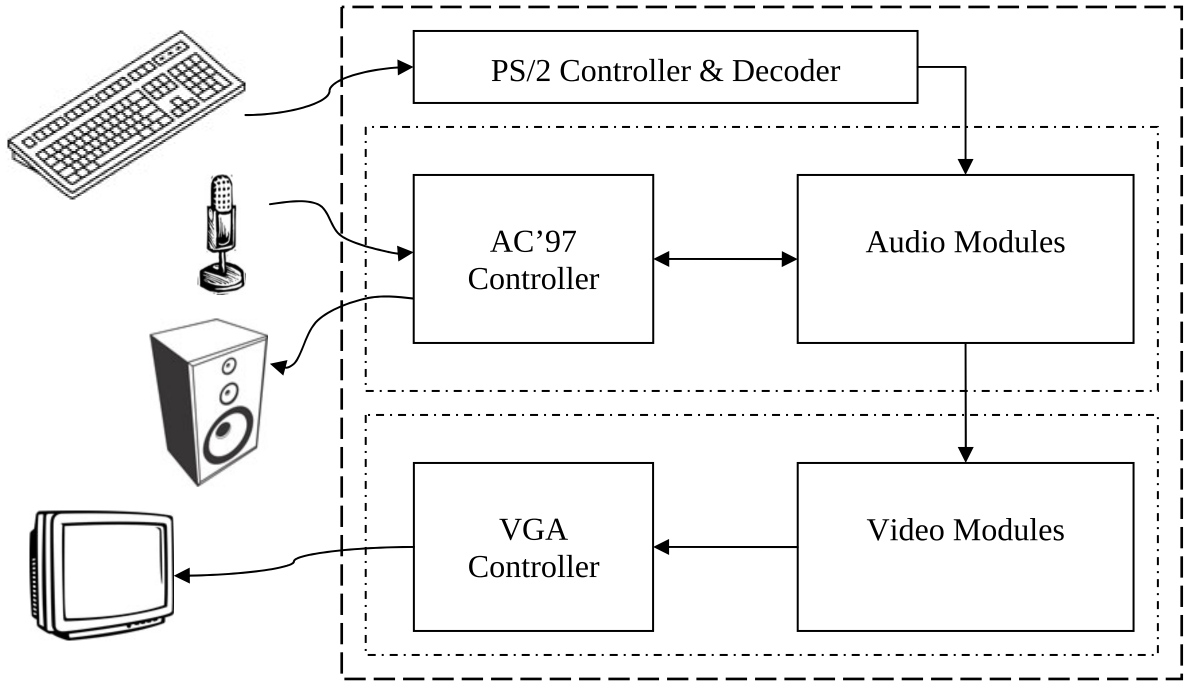

Figure 1: High-level overview of system components.

The audio component of the system consists of an AC'97 audio controller, a fast Fourier transform (FFT) module, a pitch detection module, a frequency modulator, and an inverse fast Fourier transform (IFFT) module. Audio data is continuously sent through the FFT module to compute its frequency spectrum. The Harmonic Product Spectrum (HPS) algorithm is used to determine the input pitch. The modulator shifts frequencies to match those specified from the keyboard, and sends the output to the IFFT module. The resulting waves are buffered, and sent back to the AC'97 at a sample rate of 48KHz. All computations are done on 1024-sample windows.

The visual components include a VGA controller, a wave display module, and a (non-functional) FFT display module. By default, the wave display updates continuously as it receives data. The user can freeze the current screen or cause the display to trigger on a rising edge of the waveform. The screen displays both input and output waves. Ideally, the real-time FFT outputs would also be displayed. The VGA runs at a 1024x768 resolution with a 60Hz refresh rate.

All modules are written in Verilog with Xilinx ISE 8. Unit testing was done with ModelSim, although most modules required incremental testing on the FPGA with a Tektronix TLA5202 Logic Analyzer.

2. Module Descriptions and Implementations

The voice modulator was developed in three parts: audio, visual, and keyboard input.

2.1. Audio [Adam]

The audio component is the major component of the project. Its job is to:

1. Sample the microphone data
2. Compute the Fourier transform of each audio frame
3. Analyze the frequency spectrum to determine the fundamental pitch of the input
4. Modulate the spectrum to change the fundamental pitch
5. Synthesize the spectrum back into a new audio frame with the inverse Fourier transform
6. Send the audio data to the headphones

2.1.1. AC’97 Controller

The AC’97 controller (Figure 2) provides a simple audio interface for the rest of the project. On system reset, it initializes the AC’97 by setting the various command registers to appropriate values (e.g. unmuting the headphone and microphone ports). It translates between the AC’97’s bit-serial protocol and a simpler 18-bit parallel protocol, and it also synchronizes from the AC’97’s 12.288MHz bit clock and the FPGA’s 27MHz clock. It provides a 48KHz sync pulse called *frame_enable* every time a new frame of audio data is ready to be sent to the headphones or received from the microphone.

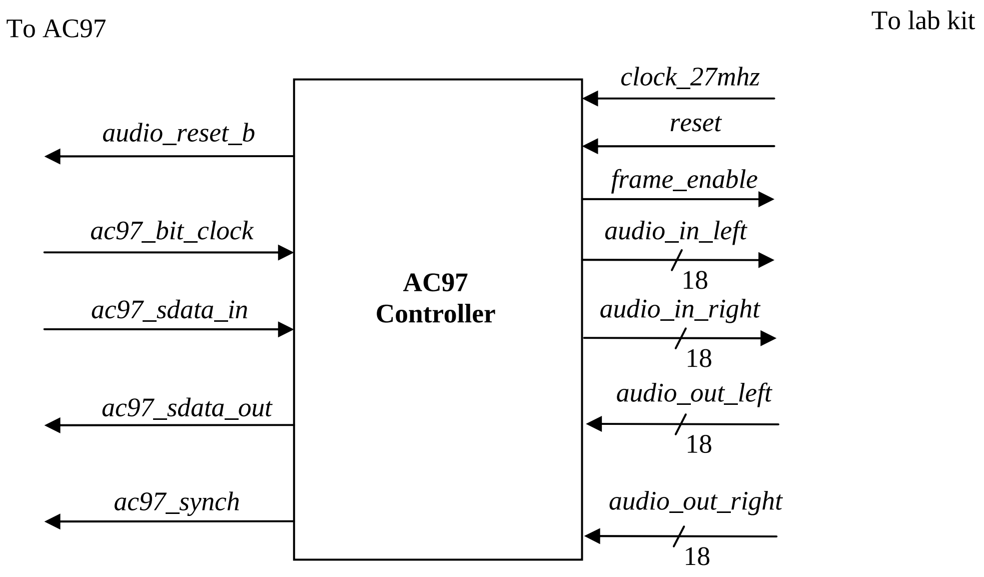

Figure 2: AC'97 Controller

2.1.2. Fourier Transform

The Fourier transform module computes a 1024-point short-time fast Fourier transform of the audio input with a rectangular windowing function. It stores audio samples in block RAM until it acquires 1024 samples, at which point it begins the computation. The FFT is implemented by the Xilinx IP CoreGen FFT, which uses the Cooley-Tukey algorithm.

The entire system works with monaural data, so the stereo inputs are converted to mono by averaging the two channels before they are fed into the FFT. Likewise, the final output signal is copied onto both output channels.

When the FFT has finished computing, it stores the resulting transform in another block RAM and pulses a start signal to the analyzer module, which then reads from that RAM as necessary.

2.1.3. Spectrum Analyzer

The spectrum analyzer module computes the fundamental frequency of the current window of audio data. It does so using the *Harmonic Product Spectrum (HPS)* algorithm

(Figure X). The basic idea behind HPS is that voice data will almost always have strong harmonics above the fundamental at twice, three times, etc. the frequency.

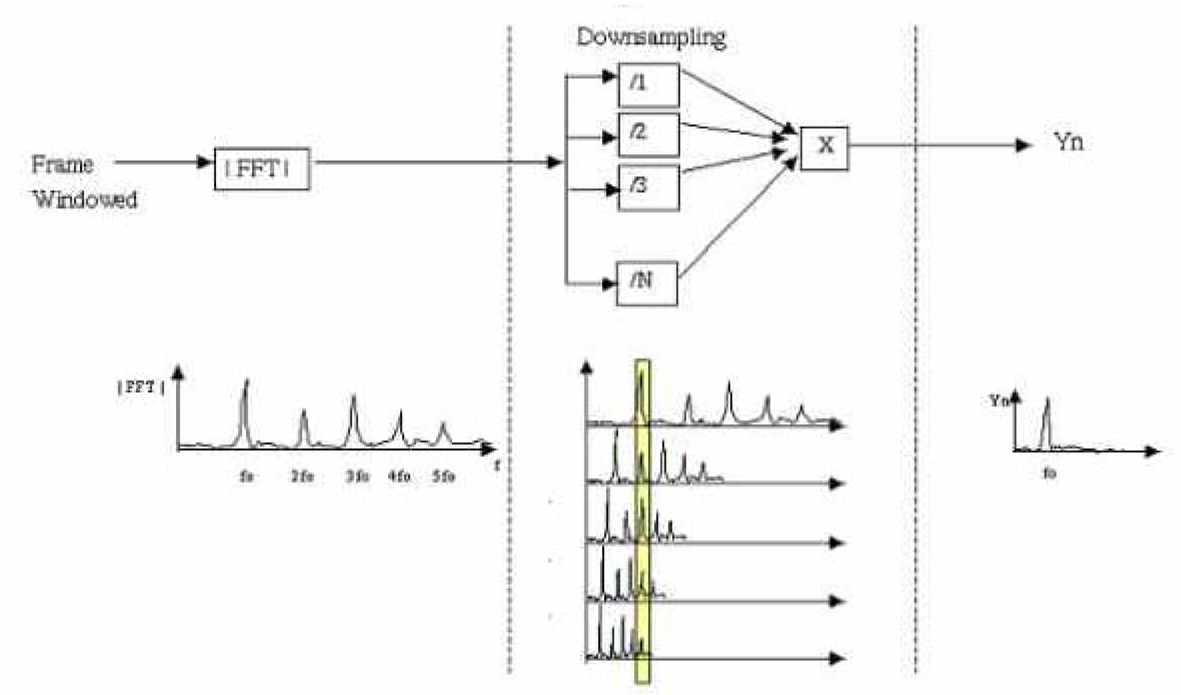

Figure 3: HPS Algorithm [1]

To exploit this, consider the spectra you would get from down sampling the input – they would be contracted by a factor equal to that of the down sampling. Now multiply these spectra together for several down sampling factors. If the original data had strong harmonics, they will line up in the down sampled spectra, creating a strong peak at the fundamental frequency. We chose to down sample by 2x and by 3x, so according to the HPS algorithm, the formula for the fundamental frequency is:

$$f_{fundamental} = \arg\max_{k} \left| X_k X_{2k} X_{3k} \right|$$

Where X_k is the *k*th component of the Fourier transform, and $|\ |$ is the standard complex norm. However, because of the discretized nature of the problem, this formula is flawed, in that it skips over many values of the transform. To rectify this, we modified the formula not to skip any indices as *k* ranges over the indices. Also, since the argmax of $|f(k)|$ is equivalent to the argmax of $|f(k)|^2$, the analyzer avoids square roots and computes squared norms instead. Thus, the formula we use is

$$f_{fundamental} = \arg\max_k \left| X_k \left(X_{2k} + X_{2k+1} \right)\left(X_{3k} + X_{3k+1} + X_{3k+2} \right) \right|^2$$

The spectrum analyzer computes this function as it iterates over the indices *k* for $1 \le k \le \left\lfloor \frac{1024}{6} \right\rfloor = 170$ to avoid aliasing, keeping track of the largest value seen so far. After it finishes, it passes the fundamental frequency onto the voice modulator module, and it pulses a start signal indicating that modulation is to begin.

2.1.4. Voice Modulator

The voice modulator module takes the Fourier transform of the audio, the computed fundamental frequency, and the desired output frequency, and it produces a new Fourier transform with a shifted fundamental frequency. It does this by scaling the transform according to the ratio of the input and output frequencies. For example, if the desired output is twice as high as the input, the transform gets stretched out by a factor of two.

The desired output frequency can actually be a whole set of frequencies, e.g. a chord. The keyboard interface provides a 48-bit vector corresponding to which of the 48 musical notes are currently being pressed. For each key, the voice modulator performs the modulation and adds all of the results together.

The first step in the modulation process is that the output transform is initialized to all zeroes. Then, for each output frequency, the ratio *r* of the output to input frequencies is computed by a fixed-point division. Next, each index *k* of the input FFT is mapped to the index *rk* of the output. To avoid losing information and energy, the new value of *out_fft*[*rk*] gets set to *out_fft*[*rk*] + *in_fft*[*k*]. The new FFT is stored in another block RAM. When the modulation has finished, the voice modulator module pulses a start signal to the inverse Fourier transform module, indicating that the audio synthesis is ready to begin.

2.1.5 Inverse Fourier Transform

The inverse Fourier transform module synthesizes a window of audio data from the transform produced by the voice modulator. It uses the same CoreGen module as the forward transform. After the transform has finished computing, it stores the audio data in a block RAM.

The audio data is then passed back to the AC'97 controller as needed according to the *frame_enable* signal, which is pulsed at 48KHz

Ideally, the inverse transform will finish computing each window just as the last sample from the previous window is being fed to the AC'97 controller. Although this does not occur in practice, it does not produce any noticeable effect of having a small number of frames from one audio window appear in the next or previous window due to timing differences between windows.

2.2. Video [Lunduo]

Figure 4 shows the modules involved in the video component. The *audio_xxx* and *fft_xxx* signals are from the AC'97 controller and inverse FFT modules respectively. The *mod_xxx* and *fft_xxx* come are from the modulator and FFT modules respectively.

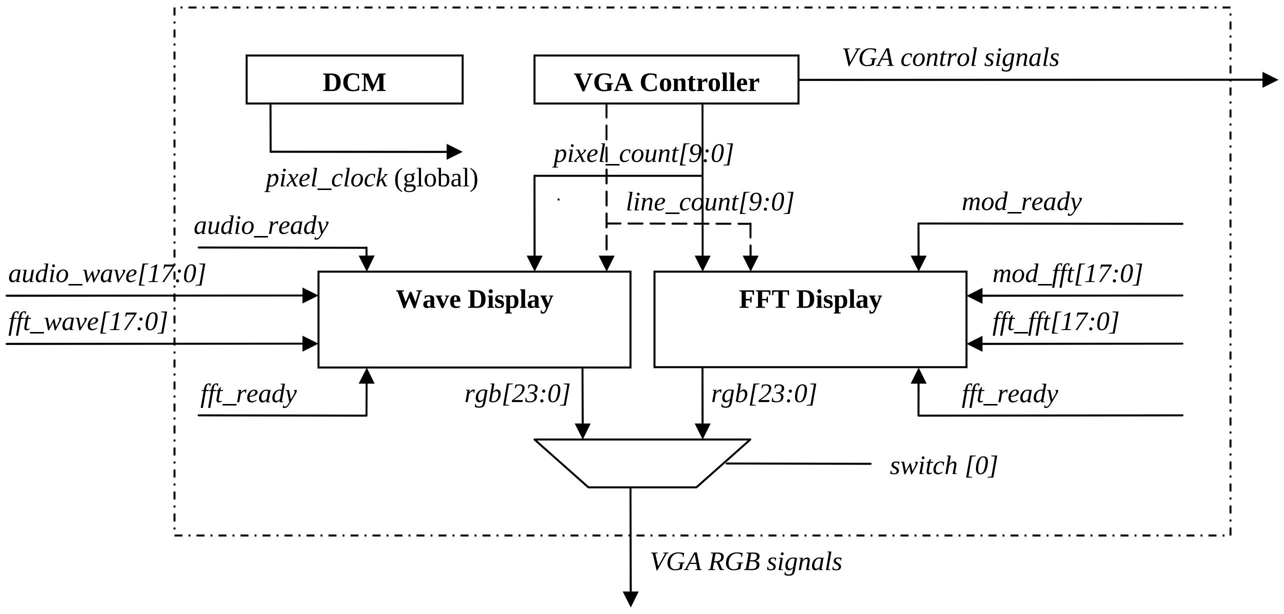

Figure 4: Video Component

2.2.1. Wave Display

The wave display module takes a *ready* signal, a wave input, and current pixel coordinates as inputs. The module samples the wave only when *ready* is high. It keeps three buffers: a *display buffer* for the current VGA frame, a *wave buffer* to hold new audio data, and an *other buffer* containing enough audio data for the next VGA frame. Buffers are numbered 0, 1, and 2. Any buffer number can be derived by exclusive-nor'ing the other two. Each buffer slice stores the minimum and maximum y-coordinates for a given x-coordinate.

The module keeps a pointer to the current rendering frame called the *wave pointer*. Because the pixel moves horizontally across lines, the wave pointer traverses the entire display buffer once per line. At the end of each line, the pointer is reset to 0. The module outputs color if the y-coordinate is between the minimum and maximum values for the given x-coordinate, or black otherwise. The wave buffer keeps its own pointer. The module continually writes to the active wave frame. When the *ready* signal is high, the wave pointer is incremented and the audio data is latched.

When the wave buffer is complete, it swaps pointers with the other buffer. When the pixel reaches the end of the VGA frame, the display buffer swaps pointers with the most recently completed buffer. Rendering a VGA frame takes much longer than filling a buffer, so the two non-Display buffers are overwritten several times before another frame can be displayed. VGA frames are necessarily discontinuous snapshots in time; however, the human brain simply interprets the discontinuity as moving forms.

In addition to data inputs, the user controls the *hold* and *trigger* signals to the module. When *hold* is high, the display buffer remains constant. The user sees the same data until he releases *hold*. The timing is illustrated in Figure 5. When *trigger* is enabled, the wave data must trigger before starting to write a new wave buffer. Triggering occurs when a positive sample value follows a zero value. Once triggered, all subsequent wave data is written until the wave buffer is complete.

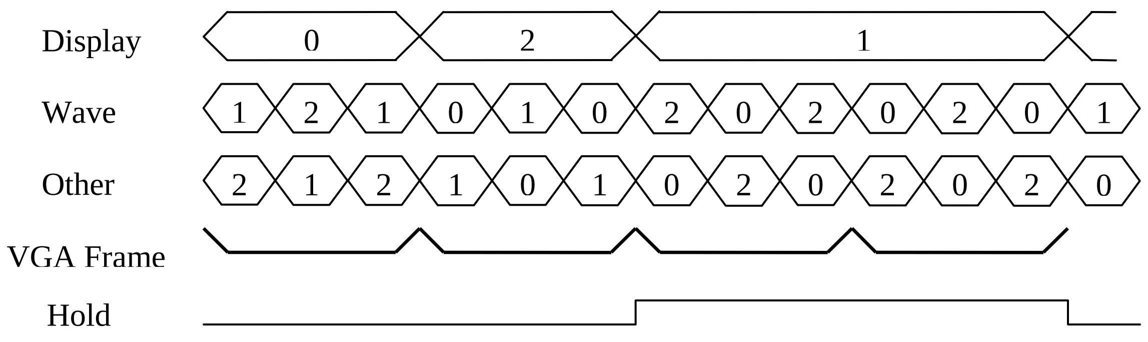

Figure 5: Timing Diagram of Buffer Swaps

There were several challenges in implementing the wave display module. The VGA display runs on a 31.5-MHz pixel clock, but audio data is updated on an internal 27-MHz clock. It would be dangerous to update buffers on different clocks, as the display and wave pointers might be assigned to the same buffer. Therefore, all buffers are updated only on the 27-MHz clock. The display buffer is updated only once during the vertical blanking period and never conflicts with the wave buffer.

Trigger mode causes a problem when the data does not cross the zero-line. “Zero-line” refers to any value whose 7 most significant value bits are zero. When data never triggers, the display and other buffer alternate while the wave buffer remains constant. As a result, the screen flickers between the most recent triggered data and an older snapshot. To fix this issue, a flag can be set once a buffer is overwritten. For example, if display is buffer 0, *flags* should be 3’b110 when display is ready to switch buffers. If wave is stuck on buffer 1, display will first switch to buffer 2. When it is done rendering, *flags* will be 3’b010. Display will then remain at buffer 2 because buffer 0 contains old data.

It is possible and simpler to display waveforms with only two buffers, one for rendering and the other for loading data. However, triggering the data would be difficult. If the sampling rate, pixel clock speed, and how often data triggers were known, one could design the module to update the wave buffer a known number of times during VGA blanking. However, this design would require a constant sampling rate and knowledge of the nature of wave inputs.

2.2.2. FFT Display

The FFT display module is similar to the wave display module, except that it always triggers when the FFT index is zero. Inputs to the FFT display include the FFT index, the real and imaginary values, and a *ready* signal. A buffer slice stores the absolute value of the FFT vector for a given index. Unfortunately, we were unable to fully debug the module. The final system does not contain FFT visualizations.

2.2.3. VGA Controller

The VGA controller takes only the *reset* and *clock* signals as inputs. It outputs sync and blank signals to the VGA, as well as current pixel and line counts. The module uses an internal pixel counter, incremented at each clock edge. The pixel counter is reset when it reaches the pixel limit for each line. The line count is incremented once a line. It also rolls over once per frame. The VGA control signals are generated with combinatorial logic according to Figure 6.

All signals are active low. Sync signals pulse low during the horizontal and vertical sync periods. Blank signals are high during active video periods and low otherwise. This implementation ties the composite sync signal to 1, as it is not used in most modern VGA displays.

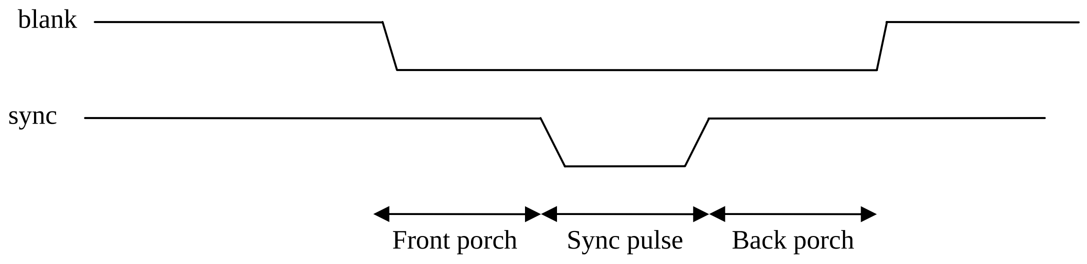

Figure 6: Timing Diagrams for VGA Control Signals

Two wave display modules for the input and output waves run simultaneously. Both modules feed RGB outputs into a multiplexer that selects the VGA signals, depending on the current area being displayed. With the current implementation, it would also be possible to simply output the bitwise OR of all display outputs. Display modules simply output black (RGB 0) when the pixel is outside of the module's area parameters.

2.3. **Keyboard** [Lunduo]

A PS/2 keyboard is used to input selected pitches. The lower two rows [**Z-comma**] and [**S-J**] represent the white and black piano keys respectively for one octave. The upper two rows [**Q-I**] and [**2-7**] represent the set one octave higher. [**F1-F3**] change the base octave (0 – 2), thus allowing the user to input 49 different pitches.

PS/2 allows for both device-to-host and host-to-device communication. This project only uses device-to-host communication. The keyboard provides the clock signal at 10 – 16.7 kHz. PS/2 device-to-host is a serial protocol consisting of 11-bit frames (Figure 7). Each frame contains a start bit, a data byte in little endian, and parity and stop bits. Data is written on the rising edge of the clock, and sampled on the falling edge.

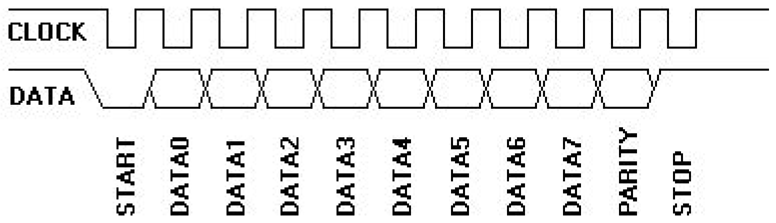

Figure 7: Device-to-Host Communication for PS/2 [2]

When a key is pressed, the keyboard generates a "make code" containing the corresponding scan code. Scan codes are generally one byte long, although some keys have extended codes prefixed with 0xE0. When a key is released, the keyboard generates a "break code" consisting of 0xF0 and the scan code for the key. The data line is held high until the keyboard has data to send.

The PS/2 controller module is a simple FSM that reads data from the keyboard and outputs a key number, state (on/off), and octave. For simplicity, the module only handles one-byte scan codes. The user is limited to using the aforementioned keys. The PS/2 decoder module then converts the key number and octave into a single key index (0 – 48).

For unknown reasons, pressing too many keys simultaneously or pressing the same key too quickly sometimes causes codes to be dropped. The space bar is the “clear all” key, which resets all key states to off.

<u>3. Testing and Debugging</u> [Adam and Lunduo]

Testing and debugging a large, complex system like this one is a daunting task. Nothing ever works the first time, and the process is slowed even further by very long compilation times that only get longer as the project grows.

ModelSim was an invaluable resource for testing and debugging. We wrote test benches for every major component of the system to test them in controlled, isolated environments, and then we ran these in ModelSim to verify the correct outputs. As we wrote and tested components, we started connecting them to each other and ensuring they could work together. Finally, we wrote a test bench which simulated the entire lab kit to make sure the entire pipeline of modules worked together.

One of the biggest challenges in testing was simulating the audio data. At 48KHz, one would have to simulate for roughly 20ms to get one 1024-sample window of audio data. Typical simulation lengths are on the order of several hundred microseconds. To get around this, we put a switch in the system that would generate false audio data in the form of a sawtooth wave at a much higher frequency, but it still had to be slow enough to allow all of the computations to be performed during one window.

Code that works in simulation will often not work on hardware for a variety of reasons. These situations were the hardest to debug. The logic analyzers were a very valuable resource for this, but they are limited to 32 bit lines, when frequently several hundred data bits need to be watched at once to get a full picture of all of the system’s internal state. To get around this, we had a switch cycle through states where different signals were muxed into the analyzer probes. This still has its limitations, however, and every recompile took over 30 minutes with all of the modules in place.

Most video data required full recompiles to test, as timing issues were not present in simulations. Once video and FFT modules were present in the same top-level labkit module, Xilinx would generate hold errors for the internal 27-MHz clock. Although we were unable to find the cause of this problem, we managed to solve it by buffering all internal clock signals.

4. Results and Conclusions [Adam and Lunduo]

Time proved to be the ultimate enemy in this project. The very long compile times made testing and debugging excruciatingly painful, and although almost every module seemed to work perfectly four times out of five, we had a difficult time getting everything to work together.

Pitch detection proved to be the most difficult part. The output of the HPS algorithm was not as accurate was we had hoped. As a result, the output frequency was all over the place and did not resemble voice at all. The best results we got were when we ignored the output of the pitch detection and hard-wired the detected frequency to a fixed value. The voice could then be modulated up or down by a fixed ratio, and it sounded really cool and was even intelligible.

One reason for this is probably the low frequency resolution – at a sampling rate of 48Khz and a window size of 1024, the frequency resolution is about 48Hz. Two ways to increase the resolution are to decrease the sample rate by down sampling or to increase the window size. In a test where we down sampled, the pitch detection was significantly better, but the modulation would no longer work. In a test where we increased the window size, the pitch detection accuracy did not improve.

Future work would obviously be to improve the pitch detection algorithm. One idea we had but did not have a chance to implement and test was to have two separate forward Fourier transforms – one on the regular audio data and one on down sampled data. The transform of the down sampled data would be used to compute the detected frequency according to HPS, and that value would be fed to the voice modulator, but that would work with the original transform data, not the down sampled data.

Other future work would be to finish and integrate all of the modules which we could not successfully debug in time to work with the rest of the system, specifically the MIDI controller and the FFT VGA display. A MIDI keyboard provides a much better musical input mechanism and is also more reliable when a large number of keys are pressed. Being able to visualize the FFT would also help immensely in debugging.

All in all, our Voice Modulator system can provide cool effects by changing someone's voice up or down in pitch while preserving vocal formants and intelligible speech, but it could have been even cooler if the pitch detection worked better.

5. References

[1] Garreth Middleton. "Pitch Detection Algorithms". <http://cnx.org/content/m11714/latest/>

[2] Adam Chapweske. "The PS/2 Mouse/Keyboard Protocol.
<http://www.computer-engineering.org/ps2protocol/>

Nathan Ickes. "Audio Input and Output".
< http://www-mtl.mit.edu/Courses/6.111/labkit/audio.shtml>