CSEE 4840: Autotune

Fuming Qiu (fq2151)
Luna Ruiz (ler2167)

Table of Contents

Overview

Equipment →

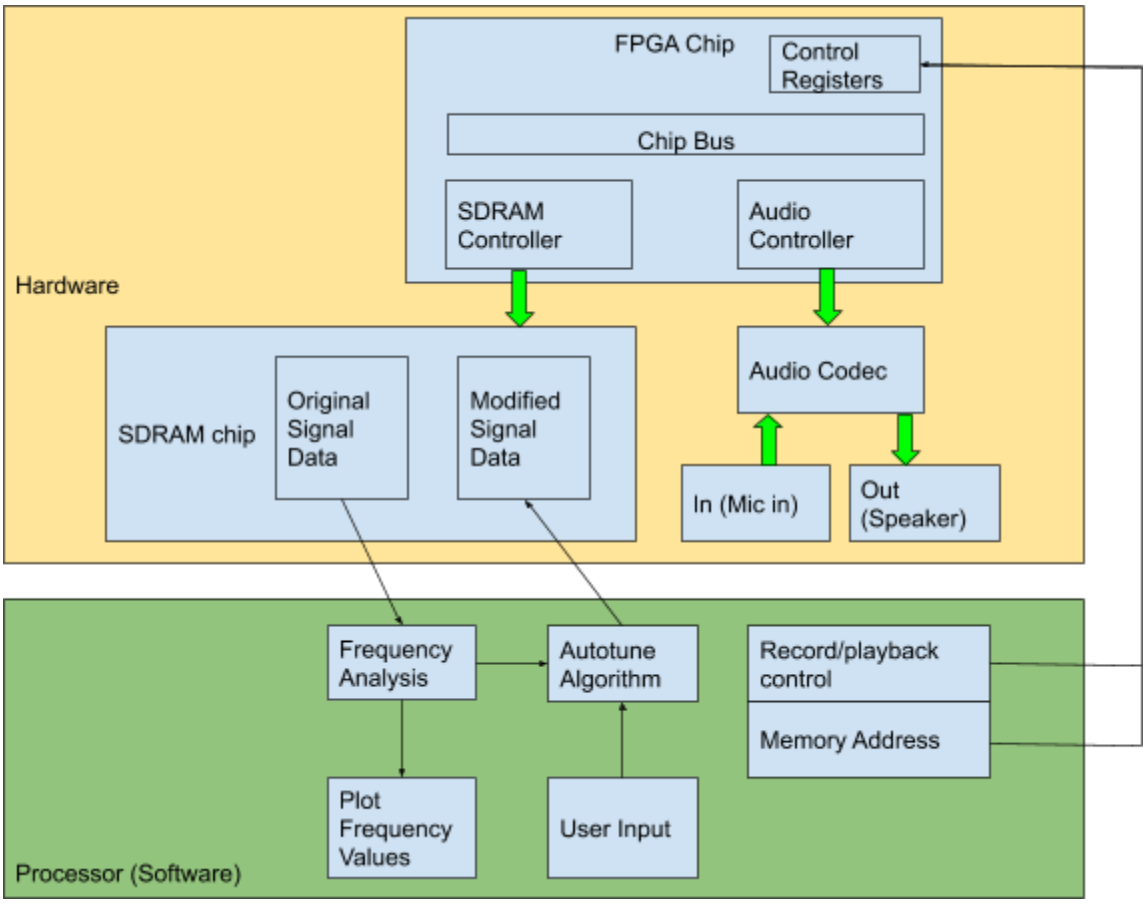
- Terasic DE1-SoC board (FPGA)
- Speaker

How it Works →

Software: Given an input WAV file, the software changes the pitch of WAV file samples based on user input and outputs a new modified WAV file

Hardware: After the software is done processing and modifying the WAV data samples, it is loaded onto SDRAM, where it is extracted and played on the FPGA.

Block Diagram



Software/Hardware Interface

Software

Necessary Files (found in SoftwareBeforeIntegration folder):

- finalProj.c → where signal processing of data can be found
- wave.h → header file with header and sample structs
- Makefile → to run program
- Gnuplot (I used brew install)

After downloading the necessary files, the user can run

./finalProj <in_wave_file> <out_wave_file>

via the command line to start the program. When the user runs this program, the user is given information about the file including sample rate, byte rate, total number of samples, amplitude range, etc. If the WAV file follows the PCM format and is mono, then the program will continue and prompt the user to enter a pitch factor. This pitch factor is used to scale the sample rate, thus changing the pitch of the signal. The out_wave_file with the new sampling frequency is generated along with a plot of the original waveform in the amplitude vs. time plane.

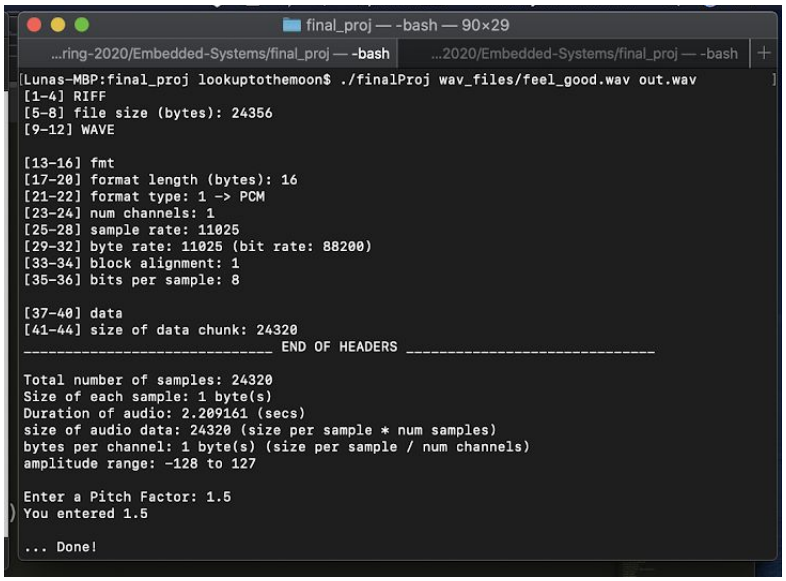
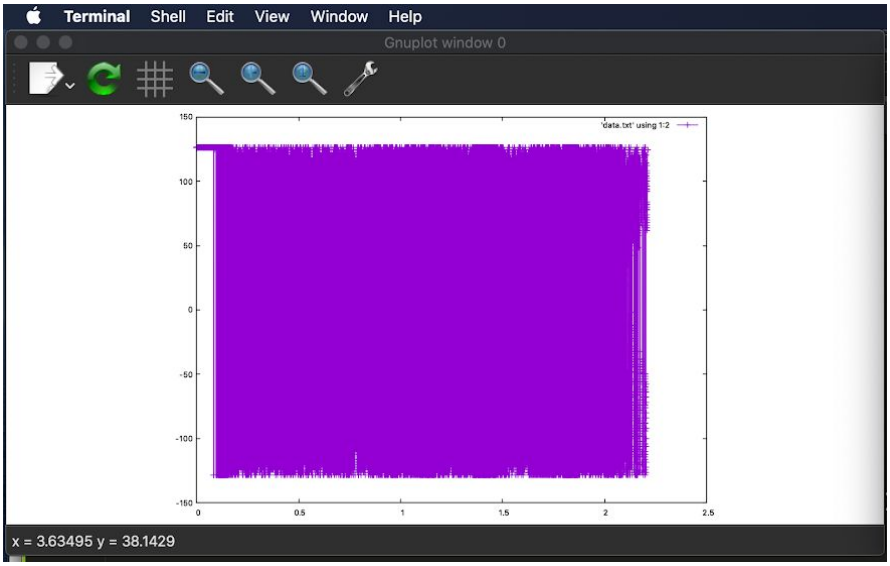


Fig. 1. Example Terminal output



Amplitude vs. Time Graph of Original WAV samples

Our original plan was to create a GUI interface for the user to interact with. We wanted our user to be able to specify a time interval and a desired pitch for that interval. Using this information, we would perform a fourier transform to convert the time domain to the frequency domain, where the pitch scaling would take place. Turns out, it was not this simple. We learned that signals are constantly changing and are hard to make predictions upon. To deal with these uncertainties, it is recommended to perform a short-time fourier transform (stft) on a set of overlapping windows (small signal data chunks) and perform pitch shifting for each window. This meant shifting the indices of the magnitude and frequency components of the calculated fourier transform for each sample in each window. However, shifting the frequency causes the times in the time domain to be out of phase. Due to this, a phase vocoder is used to scale both the frequency and time domains based on phase information. This math behind this step was confusing to understand in addition to the overwhelming data already being tracked throughout the stft.

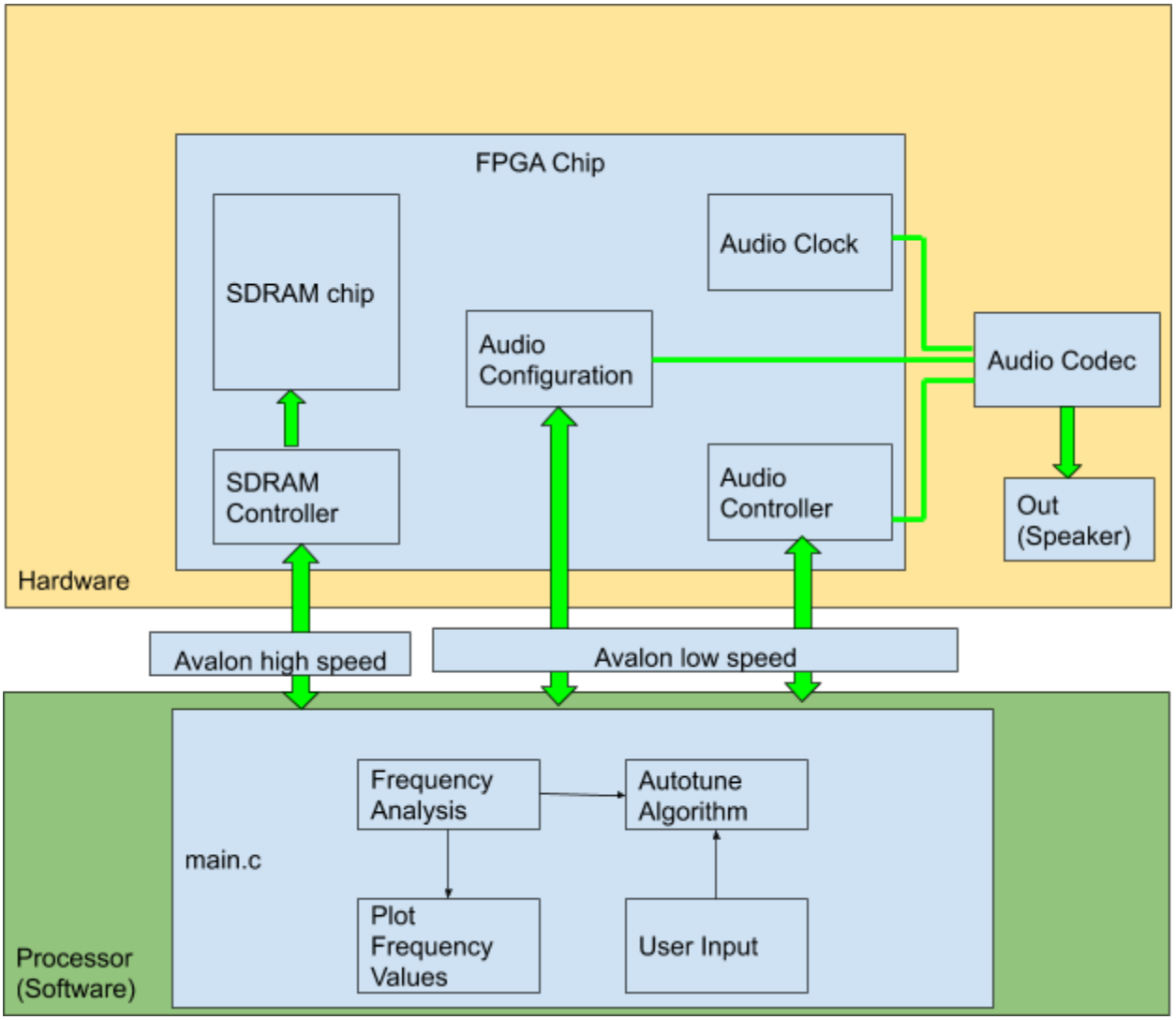
Unfortunately, due to this, we were unable to correctly execute the short-time fourier transform on our data, meaning we were not able to process the signal. Our code attempt is commented in the main method of the finalProj.c file. We also dabbled in other methods such as the synchronous overlap-add algorithm (SOLA), which performs pitch shifting in the time domain. When the sampling rate of a .wav file is changed, the signal not only changes in pitch but also in speed. As a result, the duration of the signal is also changed. The purpose of SOLA is to revert the signal back to the duration and speed originally set. The way this is done is by way of overlapping windows similar to that of the sftf. The windows are shifted based on a scaling factor and the remaining samples in the two overlapping windows are summed together depending on the data in the windows. This seemed like a great idea and much easier than the sftf. However, it was recommended that the windows were centered about the signal's pitch marks. This would require that we knew where the pitch marks were, which we didn't nor did we know how to find them. That idea was short-lived. Overall, we struggled to find a method to

implement on our data so we settled for scaling the sample rate to change the pitch of the signal and dealt with the duration changes. However, we were able to store and play the .wav file audio samples on the FPGA. In fact, the same code used to extract data from a WAV file in finalProj.c was used in the software/hardware interface to do the same thing.

Hardware

The original plan for the hardware was to expose a set of control registers and the SDRAM to the software so that once the SDRAM was populated with the audio data, the software could initiate playback. This setup would require a way for the SDRAM data to be streamed in hardware to the audio controller. Such a system would provide a clean abstraction of the hardware layer so the software would only need to access two control points.

A modified hardware interface that simplified the design with a focus on getting an end-to-end system working was adopted as time ran out. The modified hardware interface relied heavily on hardware components found in QSYS that were accessible through the Avalon bus protocols. These components exposed registers that could easily be accessed by software, as most of the difficulty in communication was abstracted by the Avalon interface. These components, implemented in the FPGA, interface to other chips on the board. This modified system is shown below.



Memory access to the off chip SDRAM was provided by the SDRAM controller, which was connected to the high speed Avalon bus master of the ARM processor. The audio codec on the DE1-SOC is provided by the on board WM8731. This codec requires a specific clock speed depending on the sample rate of the audio to be played, data lines and a data clock to transfer DAC or ADC data, and an I2C connection to configure the codec. These signals are provided by three hardware components. The data connections and data clock are provided by the audio core component, which consists of registers accessible over the Avalon bus and a 128 entry by 32 bit FIFO between the bus and the codec. The codec is configured by the audio configuration component, which outputs configuration data to the codec over I2C. This component can configure the codec using values from the Avalon bus or preset values that it stores. Finally, the codec clock is provided by an audio pll component that can be set to output the clock frequency corresponding with the specific sample rate of the audio data. The modified design has the

audio configured to support only 16 bit samples at a sample rate of 48 KHz, which is defined in the stored preset values of the audio configuration component. .

Registers:

The intended system required one 8 bit control register and a 16 bit address register. Neither of those registers are seen in the modified product. The single small control register was replaced by the registers to the various audio components, since the software needed to play a much more direct role in moving the data around the system. The 16 bit address register was unnecessary as this functionality was provided by the SDRAM controller, and it did not make sense to wrap the controller inside another module. In the modified system, the audio core component's registers were the only ones directly used. These registers are shown below (image from Altera's University program publication Audio Core for Altera DE Boards):

Table 1. Audio Core register map

Offset in bytes	Register Name	R/W	Bit Description									
			31…24	23…16	15…10	9	8	7…4	3	2	1	0
0	control	RW	*(1)*			WI	RI	*(1)*	CW	CR	WE	RE
4	fifospace	R	WS LC	WS RC	RA LC			RA RC				
8	leftdata	RW *(2)*	Left Data									
12	rightdata	RW *(2)*	Right Data									

The control register was used to clear the internal FIFOs, the fifospace register was polled so that the software knew when the FIFOs were full, and the audio data output was written to the leftdata and rightdata registers. Since only single channel audio was used, the same value was written to both data registers.

Interface

In order to access the registers and the SDRAM, the physical device memory (/dev/mem) was mmap-ed into the program's virtual space and bound to pointers. These pointers were cast to the correct data width. By dereferencing these pointers and setting their values, the registers were written with new data. This is how the 4 registers of the audio core were accessed. This system did not work well for the SDRAM, but memcpy was able to properly write data to the SDRAM. To read from the registers, the pointers were simply dereferenced.

The final modified system's software reads audio data from a .wav file, parses out the relevant header information, and copies all the sample data from the file into SDRAM. Then, a while loop keeps track of the number of samples written and each sample is written in turn by writing to the audio core registers until the FIFOs are full, which is determined by accessing the contents of the fifospace register. The data is automatically played by the codec as soon as it is written.

Future Endeavors

Endeavor 1: Successfully perform short-time fourier transform and pitch shifting on audio samples based on user input

Endeavor 2: Play any WAV audio file at any sample rate on the FPGA