6.111 Final Project Report: "FPGAuto-Tune"

Ishwarya Ananthabhotla and Trevor Walker

Contents

1 Introduction

In this final paper, we present an overview and the implementation details of "FPGAuto-tune", a real-time audio frequency shifting system designed for an FPGA. The goal of this project was to conceptualize and implement a system that is a more robust and tangible emulation of auto-tuning that is quite common in the music industry, but unique in its use of hardware and capacity to operate in real-time. Typically, the notion of auto-tuning entails shifting the entire frequency spectrum of a sequence of incoming audio samples such that the fundamental frequency is forced to a different location on the spectrum, thereby raising or lowering the pitch to match an intended note. The paper presents the methodology followed during laboratory implementation, focusing on the details of driving and managing sampling rates for the FFT and IFFT components, and the phase vocoder-based system used to perform frequency estimation and translation.

2 Summary

Figure 1 offers a high-level abstraction of our system. The system begins by sampling microphone data from the AC97 Codec built into the course lab kits, and then buffering and supplying this data at the appropriate rate to an FFT algorithm. The FFT then hands off the processed data to a digital signal processing unit which is responsible for (a) fundamental frequency estimation using a phase vocoder algorithm, (b) implementing lookup tables for note and shift factor identification, (c) implementing a "shifter" to perform this translation of frequency in accordance with the outputs of the lookup tables, (d) storing and handing this data back to the input of the IFFT.

The focus of this project was to independently develop and test the two subsystems– the FFT/ IFFT blocks, and the DSP logic blocks– as a proof of concept of the complete

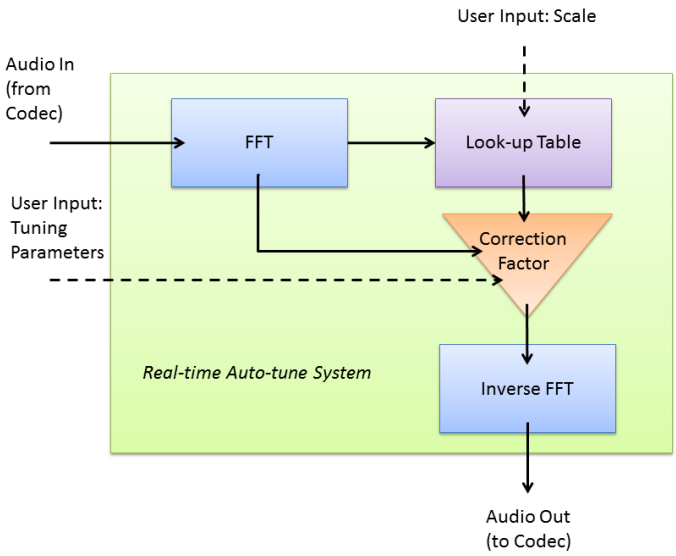

Figure 1: High-level system diagram

end-to-end system. Though full functionality resulting from system integration would have been ideal as a demonstration, it was unachievable in the time frame. However, we believe that the subsystems detailed below are representative of the key design concepts belonging to the system.

The modules that related to handling input and output audio, including the FFT driver, IFFT driver, input and output buffers were written and tested by Ishwarya. The modules pertaining to the signal processing block, including the Cordic, phase vocoder, lookup tables and shifter block were completed by Trevor.

3 Modules

Shown in Figure 2 is a more detailed block diagram representation of the entire system. Each module represented in this diagram will be explained in depth below.

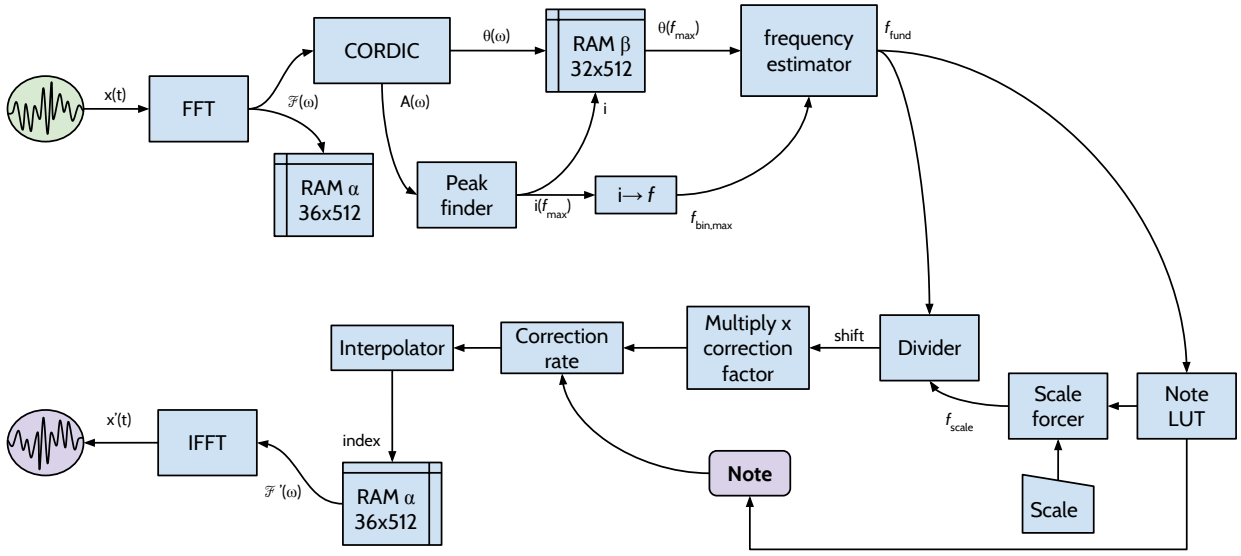

Figure 2: Full system diagram

3.1 FFT/ IFFT and Drivers

Two of the most critical elements of the system are the FFT and its inverse. The implementation for the FFT module itself was provided by the 6.111 staff, along with a simple testbench demonstrating its operation on a single sample consisting of 1024 data points. The fixed-point, low-speed FFT provided meets the following specifications– 18 bit data point precision, capability of operating at a maximum clock speed of 27mhz, and occupying only 4 percent of the total memory on the Xilinx FPGA in the lab. Due to constraints in terms of memory and processing speed, we chose an FFT window consisting of 512 points (logdepth of 9) consistently throughout the system.

In order to operate the FFT module, the testbench was responsible for setting several parameters (such as the direction of operation, the logdepth, whether the input data would be real or complex) and driving the appropriate sequence of write, read, and address signals to correctly process sample generated data from a text file. To be able to incorporate an FFT module into a more complex sequence of blocks, we required formal verilog modules to serve as drivers and maintain the FFT's local three state FSM (writing in data, computing

the data, reading out the data).

Since all of the complexity of operating the FFT was built into the verilog system in the form of drives, a simple test bench could be written to parse sample input data generated by a python script into the input of the driver at a rate that resembled the relative sampling rate of the AC97 audio codec. Several simulations were then run using ModelSim, while fine tuning was performing on the timing of the control signals until the appropriate frequency spectrum could be viewed as an analog waveform at the output of the driver and test bench. Results of these simulations are shown below in Figure 3.

A similar driver and testbench setup was created for the IFFT as well, where the direction of operation and the control signals were altered appropriately. Some data simulations were run here as well, though it was much more difficult to construct sample input data to represent a wide variety of audio signals (apart from simple features such as sinusoidal, complex exponential, and basic DC waveforms).

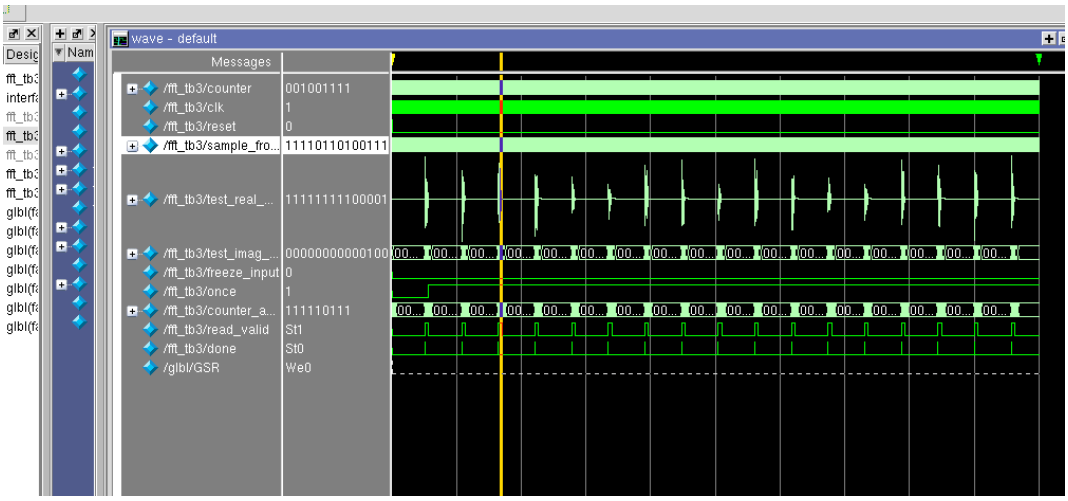

Figure 3: Driver simulation

3.2 Buffers and Sampling Challenges

An important consideration that factored into the design decisions made regarding the system was the timing discrepancy between the codec audio sampling rate and the data throughput

of the FFT and IFFT modules. When clocked at 27mhz, the labkit hardware is limited to a sampling rate of roughly 48Khz, implying that one new data point consisting of 18 bits is recorded at this rate. When the FFT module is clocked at 27mhz, it requires approximately 1000 clock cycles to process 512 such data points, giving it an operational rate of about 27khz. This is about half of the rate of the incoming audio data, which means that several data points would be lost if the audio points were fed directly into a continuously running FFT.

Several approaches were considered to solve this problem, the simplest of which entailed generating a new system clock at 54mhz to operate only the FFT, while the codec and audio sampling mechanisms still operated on a 27mhz clock. This would ensure that the FFT and IFFT internal system were necessarily crunching data points at a rate faster than that at which the codec could sample incoming audio, and samples would not be dropped. However, it was discovered that the staff's FFT implementation was not pipelined, and could not be clocked any faster than the system specification of 27mhz.

The alternative approach was to place an appropriately sized buffer between the ends of the system and the codec to handle these discrepancies. On the input end, a 512 point FIFO buffer was implemented to collect data from the codec, and only when full was the data read into the input of the FFT. Since this FIFO would only fill up at about a frequency of 48khz/512 points = 94Hz, and the FFT can read that data in 27mhz/512cycles = 53khz, there would be no concern of perpetually dropped or overwritten samples during the period in which the data was input from the FIFO to the FFT. A simulation of the FIFO operation can be seen in Figures 4 and 5.

Similarly, a buffer is also required for the data at the output of the IFFT and prior to the data being written out to the codec, but with a slightly different scheme of operation. The IFFT loads the FIFO with data in bursts of 512 points a time, which comprise one sample. The codec simply reads one point at a time from the buffer whenever there is a "ready" signal pulsed, taking place at a rate of 48khz. This ensures that the input and output sampling

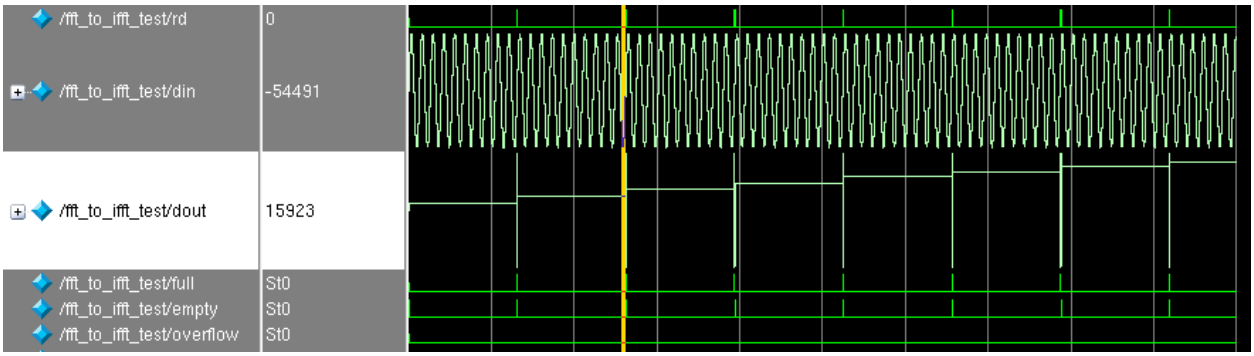

Figure 4: FIFO simulation

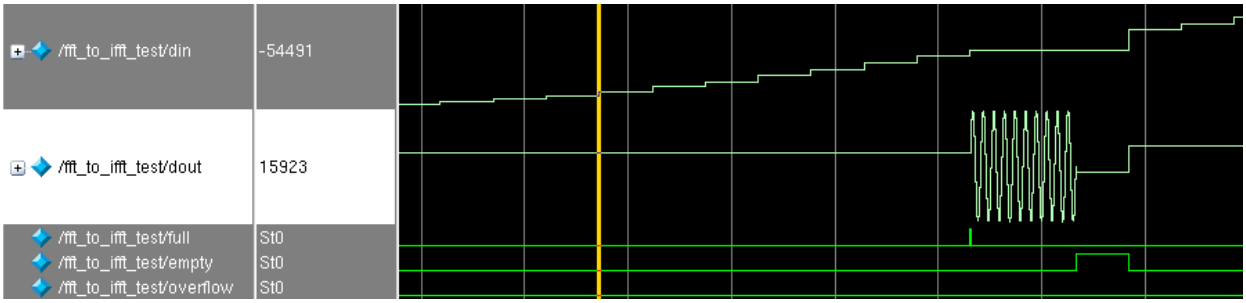

Figure 5: Detail of FIFO simulation

rate are maintained by the internals of the system, allowing the sound to be reconstructed decently well.

3.3 Audio reconstruction with FFT to IFFT as a proof of concept

In order to demonstrate the performance of the buffers, drivers, FFT and IFFT modules, we chose to initially connect the output of the FFT directly to the input of the IFFT and show that the input audio could be reconstructed by the system. We were able to successfully obtain the input audio at the output, and particularly well for pure tones, but there persisted to be several artifacts in the audio. For example, there existed a slight intermittent "clicking" noise, which was a function of the processing of discrete, non-overlapping audio windows. Additionally, there was a high frequency drone that formed a background to the audio– this could have been the result of the lack of complete synchronization between the operation of

the input "write" cycle of the FFT and the output write of the FIFO, resulting in a few data points being dropped or misplaced in every sample processed by the FFT. A simulation of the full FFT to IFFT system can be seen in Figure 6.

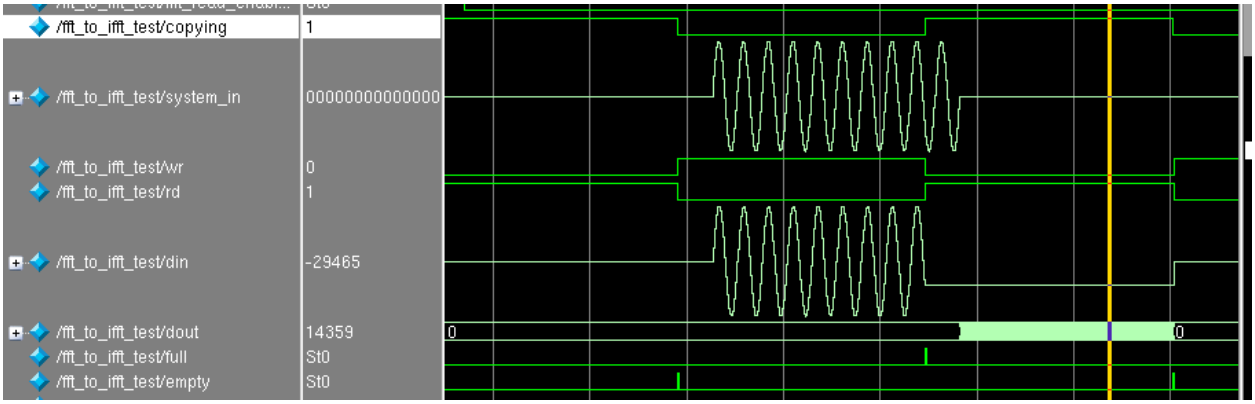

Figure 6: FFT→IFFT simulation

3.4 Preprocessing

3.4.1 CORDIC

Before the FFT result can be used for frequency estimation and correction, several modifications must be applied. In order to ensure that this portion is not reliant on the FFT module holding its data constant for the total duration of all the transformations, the output of the FFT (shown as $\mathscr{F}(\omega)$ in Figure 2) is stored in a RAM (RAMα in the diagram). This isolates subsequent operations on $\mathscr{F}(\omega)$ from changing values at the FFT outputs.

$\mathscr{F}(\omega)$ represents the frequency-domain content of the transformed signal. However, the complex data are represented in Cartesian coordinates; since the latter modules require phasor data the FFT result must undergo a coordinate transformation. This is the function of the COordinate Rotation DIgital Computer, or CORDIC; it implements an iterative algorithm for several trigonometric functions, including Cartesian-to-polar translation, which is fast and takes up little area on the FPGA. Each value of $\mathscr{F}(\omega)$ is passed through the CORDIC

in turn, and the resulting phase $\theta(\omega)$ is stored in RAMβ. The magnitude $A(\omega)$, meanwhile, is passed through a peak-finder module in order to determine which FFT bin contains the most energy.

3.5 Finding the fundamental

Next, RAMβ is indexed by the index of the highest-energy bin, producing $\theta(f_{max})$, the phase of the maximum bin. We make the assumption that the highest-energy bin contains the fundamental frequency - this always holds for pure sinusoids, and for our purposes should be good enough for most vocal content. $\theta(f_{max})$, along with the frequency of the maximum bin (determined by multiplying the index by $\frac{44.1\,\text{kHz}}{512}$, i$\rightarrow$f in Figure 2) is passed into the frequency estimator module.

3.5.1 Frequency estimator

This module, implementing a phase vocoder, is essential to nearly all implementations of auto-tune algorithms. The frequency resolution of a 512-point FFT at a sampling rate of 44.1 kHz is theoretically only $\frac{44.1\,\text{kHz}}{512} \approx 86\,\text{Hz}$. The phase vocoder is able to increase that resolution by a factor of 1000 or more by incorporating information from the time domain. Specifically, the fact that the phase of a sinusoid is the derivative of its frequency means that by dividing the change in phase between adjacent windows by the time difference between then gives a very good estimate of the true value of the fundamental frequency. However, there is another wrinkle: the phase may change enough between cycles to exceed 2π radians, possibly several times over. Since the phase is always represented within a range of 2π radians, it is actually the phase module 2π that is being calculated. To account for this, the numerator

of the algorithm has a $2\pi n$ term added; n is incremented until the calculated frequency is closest to the center frequency of the highest-energy bin. The full equation is as follows:

$$f_{fundamental} = \frac{\theta_2 - \theta_1 + 2\pi n}{t_2 - t_1}$$

3.6 Correction

Once the phase vocoder has determined the fundamental frequency, further operations are possible. The first of these is determining which note the user sang into the input; this is the function of the "Note LUT" module. It takes as input the fundamental frequency, and outputs a note from 0 to 11 (C to B) as well as the octave. Figure 7 shows the output of our Verilog testbench for the note lookup table; it sweeps through three octaves of frequencies.

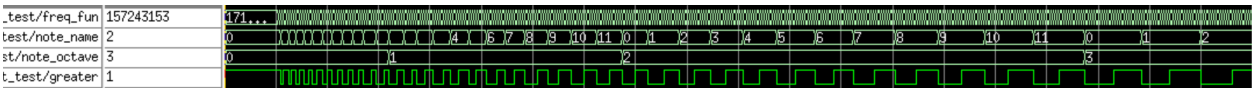

Figure 7: Output of Note LUT testbench

After the note is determined, the closest note on the user-provided scale is determined, and the fundamental frequency is divided by the closest scale note to produce the "shift factor". This is then modulated by the other user-controlled parameters, degree of correction and correction rate, and the result fed into the interpolator. This module's function is to interpolate the FFT results (still in RAMα, by multiplying the indices by the scale-factor. In this case we used nearest-neighbor interpolation, simply rounding the scaled indices to the nearest integer. This produces a resultant transform $\mathscr{F}'(\omega)$ which can be passed to the IFFT for resynthesis as audio.

4 Implementation Process

Implementation was conducted mostly in parallel - Ishwarya implemented the FFT/IFFT and their drivers, while Trevor focused on the other modules. The process consisted of writing the Verilog module, producing a testbench to evaluate it, and debugging iteratively until the module passed all tests. In this way we tried to ensure that there would be no surprises when the modules were connected. Furthermore, all of the non-FFT modules were connected together and encapsulated in an FSM; this FSM connected to the FFT and one end and the IFFT at the other, simplifying the integration process.

5 Review and Recommendations

Unfortunately, integration posed unforeseen problems. When we attempted to combine the two portions of the project, we obtained unexpected results from the integrated system. This is discussed further in Section 6. In the future, we recommend that integration be spread throughout the process of development; this would permit the identification of problems like the ones we encountered earlier in the process and enable us to correct them in advance of the final construction of the system.

6 Conclusion

In conclusion, we have presented the implementation details of the FPGAuto-tune system, a concept and partial physical implementation of a system that performs frequency detection, estimation, and shifting in real-time. While the project was advantageous in that it required very little physical hardware, and the two subsystems could be developed and

tested independently, we underestimated the challenges that would arise with operating the staff-provided FFT implementation and the resulting complexity of the entire system. Since we have demonstrated that our components are each functional on their own, a completely functional system is only a matter of forward extension by a short period of time.