fLaCPGA - FPGA fLaC Implementation

Progress Report

Emmanuel Jacyna - 24227498

May 24, 2016

Contents

Abstract

The purpose of this document is to describe the objectives of the fLaCPGA project, explain the progress achieved to date, and to set out the means by which the project goals will be achieved.

1 Introduction

The goal of my final year project is to investigate hardware optimization techniques through the implementation of a Free Lossless Audio Codec (henceforth, fLaC) decoder and encoder in Verilog. fLaC is a lossless audio compression codec that is very popular as a method of distributing high quality audio recordings and for compressing large audio archives[1].

2 Progress To Date

- ~~Implement a fLaC compatible encoder in C++.~~
- ~~Implement a fLaC compatible decoder in C++.~~
- Implement a fLaC compatible encoder in Verilog.
- ~~Implement a fLaC compatible decoder in Verilog.~~
- ~~Identify areas of the encoding process that can be accelerated using FPGA hardware.~~
- ~~Identify areas of the decoding process that can be accelerated using FPGA hardware.~~
- Optimize the fLaC encoder in the areas identified.
- Optimize the fLaC decoder in the areas identified.

Table 1: List of achieved requirements

2.1 Software fLaC Decoder

A C++ fLaC decoder was completed by week six of the semester. The decoder can successfully decode 16 bit stereo fLaC encoded files. Implementing the decoder provided me with valuable knowledge of the internal workings of the fLaC decoding process. This proved very useful when implementing the hardware decoder.

The fLaC reference code, which is freely available under an open source license, was used as a guide to implementing my version of the decoder. I used a test-driven design methodology when coding; writing unit tests using the GoogleTest C++ testing framework to ensure that my code worked as expected. The most time consuming part of implementing the decoder turned out to be implementing a bit stream reader. Data encoded using fLaC is of course bit packed to achieve compression, so in order to decode it the decoder must be able to access data with single bit precision. There were a large number of various edge cases that needed to be handled by the decoder, but once I had implemented a wrapper for the C++ fstream, I was able to quickly write the code to then read and decode the fLaC data.

2.2 Software fLaC Encoder

A C++ fLaC encoder was also completed during the semester. This was finished by about week ten of the semester. The encoder can successfully encode 16 bit, single channel WAVE format files. It is able to achieve 50% compression on most audio files tested so far. Implementing the encoder was fairly straight forward once the decoder had been completed. Similarly with the decoder, implementing bit packing took up a

bit of time. The encoder was much more interesting to implement, as it involves more algorithmic design than the decoder. A particular achievement was my use of a prefix sum to implement the entropy encoding parameter optimzation in $O(N)$ time.

Having implemented a software encoder proved to be essential to testing the hardware decoder. Using my encoder I was able to output single frames and subframes, which I otherwise would not have been able to do had I only had the reference encoder to work with.

2.3 Hardware fLaC Decoder

A hardware fLaC decoder was written using Verilog and tested using Modelsim. This was the most time intensive task of the semester. Whilst I had initially expected the hardware decoder to be more difficult than the software, I certainly underestimated the size of the task. My main problem was a lack of familiarity with Verilog and the process of writing and testing hardware description code. I also ended up having to rewrite a major portion of the decoder late into the semester due to a poor design earlier on which set me back a bit. Interestingly, the part I struggled with the most was the residual decoding, the part of the decoding process which requires single bit accesses. It took a number of iterations before I settled on an appropriate way of implementing this.

During the process of implementing the hardware decoder, I was able to identify a number of areas that could benefit from optimization. The residual decoding process seems well suited to a FPGA paradigm. Whilst my current method is to decode the residuals by feeding one bit at a time into a state machine, it could be possible to analyze chunks of bits instead, which would greatly speed up the time to decode residuals. Initial investigations into this show that going beyond two bits at once would lead to an explosion in the number of cases to be handled, since the number of different cases is equal to 2^n, where n is the number of bits. This problem seems like it would be more readily tackled by first identifying how to decode each, then using a code generation tool to generate the appropriate hardware description of the logic block.

Another area where optimization would be essential is in the implementation of the linear predictor decoder. The encoding process essentially uses a finite impulse response filter of the form $x[n] = a_0 * x[n-1] + ... + a_k * x[n-k]$ as the linear prediction model. The issue is that this implies that the decoder is in fact an infinite impulse response filter. This has the form $y[n] = a_0 * x[n-1] + ... + a_k * x[n-k] + b_0 * y[n-1]$. Note that $y[n]$ is dependent on $y[n-1]$. This feedback dependence leads to difficulties when pipelining the decoding algorithm, since the pipeline must wait until the appropriate feedback term is available, effectively delaying the pipeline. There are two ways in which this could be improved. One is to use the theory of IIR filters to calculate a zero-cancelling filter to convert the IIR filter into an all-pole FIR filter. This could prove to be an interesting problem in hardware. The other is to somehow precalculate the feedback terms and feed them through the pipeline.

2.3.1 RTL Diagrams

Overall, implementing the hardware decoder provided me with valuable experience with the FPGA design flow and RTL simulation.

3 Plan for Next Semester

The goal for next semester will be to implement and optimize the hardware encoder. Having already completed the hardware decoder, I have a clear idea of how I will go about implementing the encoder. One of the advantages of the encoder, which should make it more straightforward, is that the input to the encoder will simply be 16 bit signed integers. There will not be a need for complex state machines to read the input as there were in the decoder, which I believe is what caused me to struggle so much with it. Having implemented the encoder in software and also having gained more experience with RTL design, I also have a good feel for how the encoder will be designed.

I have already identified a number of areas where optimization can be applied. The process of finding the best model to use can be readily done in parallel, which should provide a major speed up over software. Encoding the residuals can also be done quite easily in parallel. A more interesting problem is implementing the Levinson-Durbin recursion algorithm. In the reference encoder, this algorithm is implemented using floating point arithmetic, whereas fixed point arithmetic is best suited for an FPGA implementation. The process of implementing the algorithm in fixed point should prove to be quite interesting.

References

[1] Kevin De Vorsey and Peter McKinney. Digital preservation in capable hands: Taking control of risk assessment at the national library of new zealand, Spring 2010. Copyright - Copyright National Information Standards Organization Spring 2010; Document feature - Illustrations; ; Last updated - 2015-09-03.