fLaCPGA - An FPGA fLaC Implementation

Emmanuel Jacyna - 24227498

October 4, 2016

Contents

List of Figures

List of Tables

Abstract

1 Introduction

This paper describes a hardware implementation of a Free Lossless Audio Codec (fLaC) encoder. The Free Lossless Audio Codec, hereafter referred to as "fLaC", is a popular scheme for losslessly compressing digital audio. The standard, along with an open source reference implementation, is freely available on the internet. It is supported by a large variety of digital playback devices which has led to its widespread popularity. It is the lossless format of choice for digital streaming platforms such as Soundcloud[**?**], Tidal, and Bandcamp. For these reasons the fLaC format was chosen as a target for hardware optimization.

Whilst there are a number of software implementations of the fLaC encoder and decoder, targeted to both traditional microprocessors, digital signal processors (DSPs), and even GPUs, there are no freely available ASIC or FPGA implementations. An efficient hardware implementation would be of great use for a number of reasons. Many nations are now in the process of digitising large national audio archives. These audio archives require data to be compressed losslessly in order to preserve content in a format faithful to the original, however, uncompressed lossless data consumes large amounts of space. Currently, uncompressed formats such as WAVE and BWF_WAVE are quite popular for audio archiving[1]. Consumer trends point to the desire to have the highest audio and video quality possible. Whilst lossy codecs such as MPEG Layer 3 (MP3) are able to satisfy that desire for now, there is a distinct marketing advantage to recording lossless audio in consumer electronics. An efficient and low power hardware implementation will allow devices to very cheaply support beyond real time encoding of large amounts of audio data.

Compressed audio has the major benefit of reducing file size by 50% or more, potentially doubling an archive's potential storage space, and reducing the amount of data transmitted over networks. In order to convert large (terabytes) amounts of audio data to a compressed format, a significant amount of computing power is required. A high throughput hardware decoder would reduce encoding time and potentially reduce power consumed by the encoding process. fLaC is also gaining popularity as a medium for portable audio players. These players are very sensitive to power consumption, thus a hardware implementation would be of great use in reducing the power load of the decoding process. Another potential use case of a hardware encoder would be in a recording studio. Instead of recording audio to an uncompressed format, high quality audio could be encoded in real time as it comes in.

The encoder described within is written in the Verilog hardware description language and is targeted to Altera's Stratix V series of FGPAs. This hardware implementation takes great advantage of the parallelism inherent in the encoding algorithm. The FPGA provides an excellent platform for expressing this parallelism and is able to achieve a speedup of 10x a conventional single core CPU.

2 Audio Compression

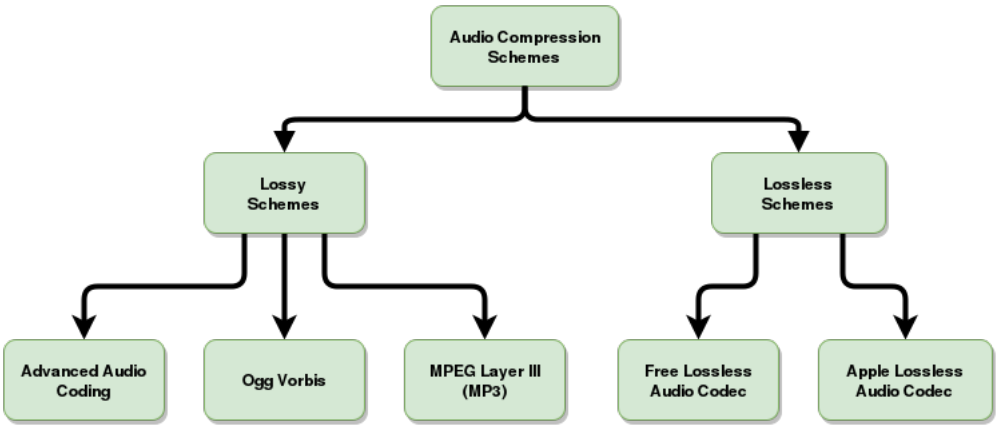

Figure 1: Taxonomy of Audio Compression Schemes

Like any compression scheme, the goal of audio compression is to use the minimum number of bits to represent the maximum amount of data. There are two broad classes of schemes within the audio compression taxonomy that are used to achieve this: lossy and lossless encoding. Lossy encoding reduces the amount of bits used by discarding audio information that is deemed to be irrelevant to the underlying signal. The MPEG-3 [**?**] format is a particularly well known example of a lossy compression scheme.

MP3 uses a psychoacoustic model of the human auditory system to discard certain sounds and frequency bands that are mostly inaudible to an untrained listener. Because it discards data, lossy compression can often achieve a dramatic reduction in file size, potentially compressing a 700MB audio CD into a 70MB MP3 file. However, lossy compression schemes suffer from one major drawback: since they discard information they often suffer from audible artefacts. Whilst this may be adequate for the transmission of speech data or even music, depending on the listener's preferences, there are times when a perfect reconstruction of the original data is required, for example in a recording studio, or when music is played back on a HiFi audio system.

In this case a lossless compression scheme will be more desirable. Instead of discarding information, lossless compression schemes find recurring patterns in the data and exploit these in order to reduce the redundancy in the signal. fLaC in particular operates by mathematically modelling the sound using a technique called *linear prediction*, and then entropy encoding the difference between the model and the actual audio. fLaC is typically able to achieve compression rates of approximately 50% or less.

2.1 fLaC Overview

The fLaC compression scheme supports a wide variety of audio bit rates, sample rates, and channels.

2.1.1 fLaC Encoding Process

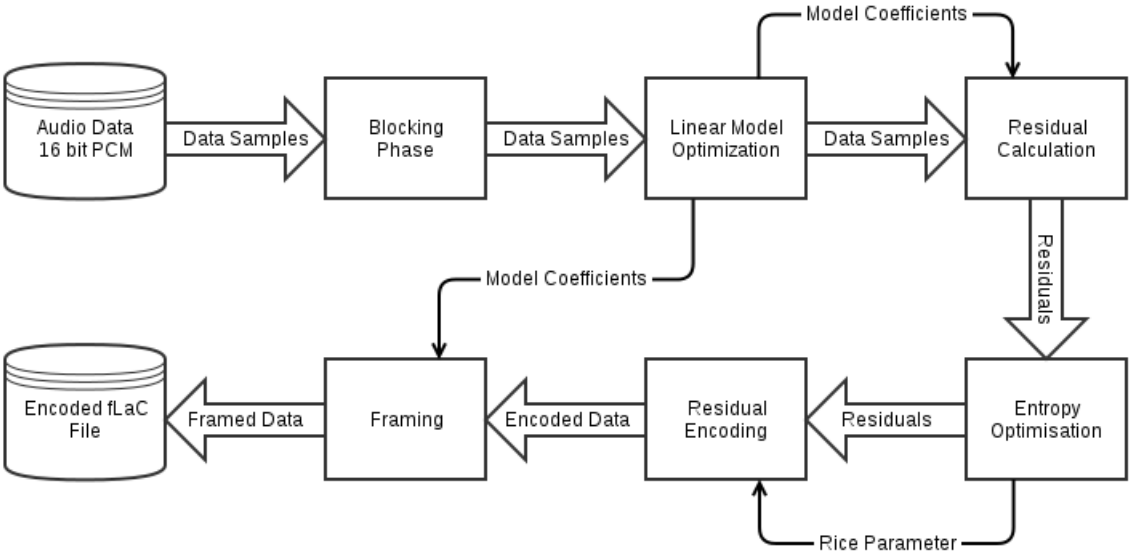

Figure 2: Overview of fLaC Encoding Process

At its core, the fLaC encoding algorithm consists of two steps: linear prediction, and entropy encoding. In the linear prediction step, the encoder analyses a block of audio and creates a mathematical model to predict the audio. The encoder then runs the predictive model and records the error between the predicted data values and the actual data values. This error is known as the "residual", as these are the residual values left over after the predictive step. Assuming that the predictive step was able to adequately describe the signal, the residual values will be smaller in magnitude than the original signal. This implies that they can be coded using fewer bits than the original audio samples.

The residuals are taken and subject to the next step, the entropy encoding step. The entropy encoding scheme used by fLaC is called Rice coding, and functions by taking the lowest k bits of the residual and encoding the higher bits using unary encoding. Since the vast majority of the residuals will fit entirely within the lower k bits, the number of bits used to record the data is reduced.

2.2 Linear Prediction

Given a discretely sampled time domain signal $x[n]$, we wish to predict the value of $x[n]$ as a linear combination of the N most recent prior samples

2.3 Entropy Encoding

3 Hardware fLaC Encoder

The hardware implementation of the fLaC encoder consists of a linear pipeline of four main stages, separated by logical function. Stage one consists of the autocorrelation unit, which calculates twelve lags of autocorrelation of the incoming audio data. Stage two is the most complex stage, and consists of a floating point converter and divider which moves the integer autocorrelation into the floating point domain, the Durbinator, the module that uses the Levinson-Durbin recursion equations to calculate the optimal linear prediction model, and finally a quantiser whhich transforms the model coefficients back into the integer domain. Stage three produces the residuals, and consists of an FIR filter bank that selects the best order encoder, and a variable order FIR filter that produces the final residuals. Stage four performs the entropy encoding of the residuals. This stage consists of a Rice coder, a Rice parameter optimizing module, and finally a module to pack the Rice coded residuals into 16 bit words and write them to a RAM.

3.1 Limitations and Assumptions

- One input channel
- Only LPC coding implemented
- Single precision math vs. double precision in software implementation
- No windowing applied during autocorrelation calculation
- Model coefficient quantisation fixed at ten bits, precision fixed at twelve bits
- Fixed Rice partition size (four partitions of 1024 samples)

3.2 Stage 1 - Autocorrelation Calculator

The autocorrelation of the incoming signal needs to be calculated up to the twelfth lag in order to calculate a twelfth order linear predictor. The hardware implementation has an advantage over software in that all twelve lags can be calculated in parallel. The autocorrelation of lag n can be calculated as in Equation 3.1.

$$\gamma(n) = \sum_{i=1}^{m} x[i] \cdot x[i+n] \tag{3.1}$$

Since the autocorrelation of any lag is independent of the other lags, this provides a perfect opportunity for parallel calculation. We can thus perform the multiply and the adds in parallel, which leads to the system diagram in figure 3. This hardware is able to calculate autocorrelation up to the twelfth lag in $4096 + 12$ cycles. Once it has calculated the autocorrelations, they are copied into a parallel in serial out shift register to be passed on to the next component while the autocorrelation hardware resets and calculates the autocorrelation of the next block of audio.

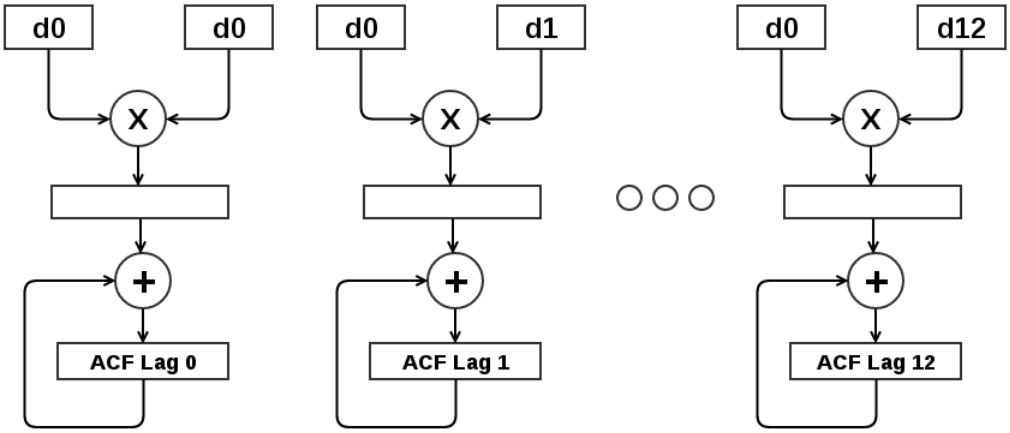

Figure 3: Autocorrelation Module Block Diagram

3.3 Stage 2 - Predictive Model Calculator

The linear prediction calculator implements the Levinson-Durbin recursive method for solving the normal equations of linear prediction. This algorithm and the subject of linear prediction are discussed in Appendix A. The relevant Levinson-Durbin equations are given below. The model coefficients are denoted $a_{m,n}$, where m is the iteration and n is the coefficient order. The reflection coefficient is written as k_m, the iteration error is written as ϵ_m and the iteration reflection update sum is given as α_m. The initial values for the problem are:

$$a_{0,0} = 1 \tag{3.2}$$
$$\alpha_0 = \gamma(1) \tag{3.3}$$
$$\epsilon_0 = \gamma(0) \tag{3.4}$$

Where $\gamma(n)$ denotes the autocorrelation of n lags. In the update step, the values of the reflection coefficient and iteration error are updated according to Equations 3.5 and 3.6.

$$k_m = \frac{-\alpha_{m-1}}{\epsilon_{m-1}} \tag{3.5}$$
$$\epsilon_m = (1 - |k_{m-1}|^2) \cdot \epsilon_{m-1} \tag{3.6}$$

Using these values the mth model can be calculated as:

$$a_{m,n} = a_{m-1,n} + k_m \cdot a_{m-1,m-n+1} \tag{3.7}$$

Finally, the α_m value is given by the dot product of the vector of m autocorrelation lags and the reversed vector of model coefficients denoted by $\bar{A}$.

$$\alpha_m = R \cdot \bar{A} \tag{3.8}$$

Refer to Appendix A for example iterations.

The Levinson-Durbin step is the most difficult to parallelise, since its inductive nature means that each step relies on the data from the step before. However, some parallelism can be extracted within each step of the algorithm. The hardware implementation was divided into modules that each handled a step of the calculation; these modules were then parallelised as necessary.

- `CalculateKAndError`
- `ModelSelector`
- `AlphaCalculator`

3.3.1 Module - CalculateKAndError

The reflection coefficient and error calculator was the most straightforward of the steps to implement. The implementation directly translated the equation into floating point hardware. The block diagram in Figure 4 describes the circuit.

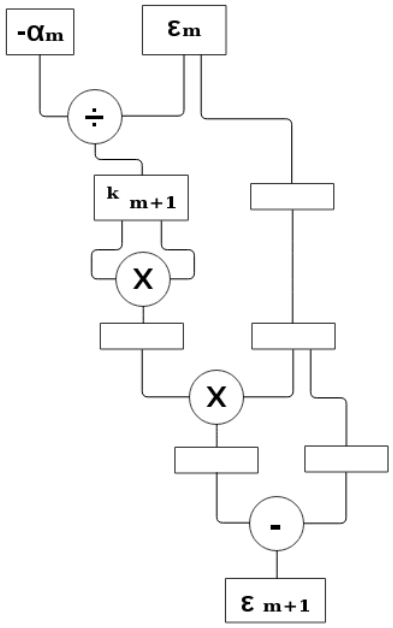

Figure 4: CalculateKAndError Module Block Diagram

3.3.2 Module - ModelSelector

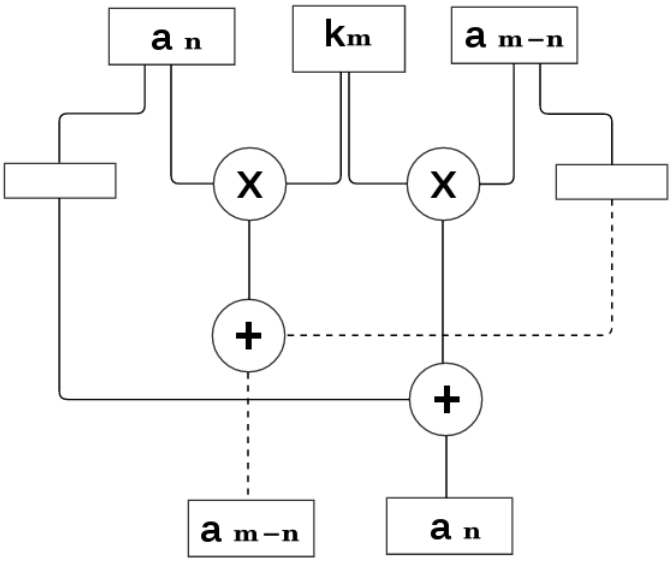

Figure 5: ModelSelector Module Block Diagram

The ModelSelector module implements Equation 3.7. This module is able to take advantage of the parallelism readily apparent in the equation. Clearly each $a_{m,n}$ may be calculated independently of all other model coefficients. Whilst it is technically possible to calculate all model coefficients in one multiply-add step, given that the model is calculated up to the twelfth order, calculating twelve models at a time would only use all resources in the last step. Instead, two model coefficients are calculated simultaneously. An additional optimisation is to reuse the $k_m \cdot a_{m,n}$ product, since Equation 3.7 has symmetry in $a_{m,n}$ and $a_{m,m-n+1}$. This leads to the circuit described in the block diagram in Figure 5.

3.3.3 Module - AlphaCalculator

The AlphaCalculator module implements Equation 3.8. This module calculates the dot product of the autocorrelation and the current model coefficients. It is designed to consume the model coefficients as they are generated, so it operates simultaneously with the ModelSelector model. An example alpha calculation is given below in Equation 3.9.

$$\alpha_3 = \gamma(1)a_{3,3} + \gamma(2)a_{3,2} + \gamma(3)a_{3,1} + \gamma(4)a_{3,0} \tag{3.9}$$

The first stage of the AlphaCalculator module is depicted in Figure 6.

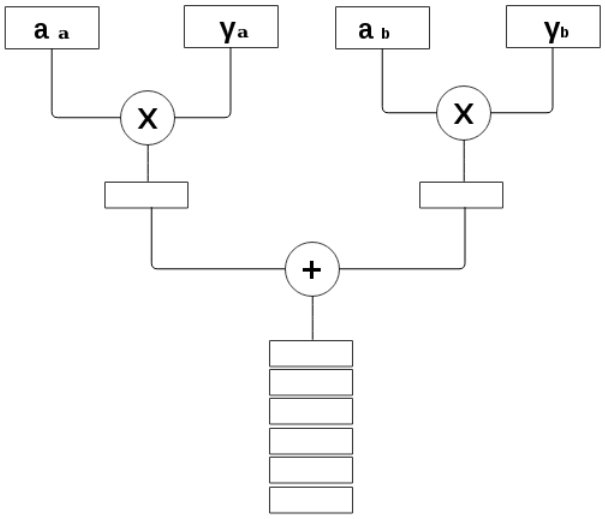

Figure 6: AlphaCalculator stage one Module Block Diagram

As each pair of model coefficients is produced by the ModelSelector, it is fed, along with the respective autocorrelation coefficient pair into the AlphaCalculator. These numbers are then multiplied together and added, then shifted down into a multi-tap shift register. Once all the partial sums have been fed into the shift register, the second stage of the AlphaCalculator begins. Four at a time, partial sums are taken out of the shift register and added together. Depending on the order of the model currently being calculated, the module may exit early and return an alpha value sooner than adding all six partial sums.

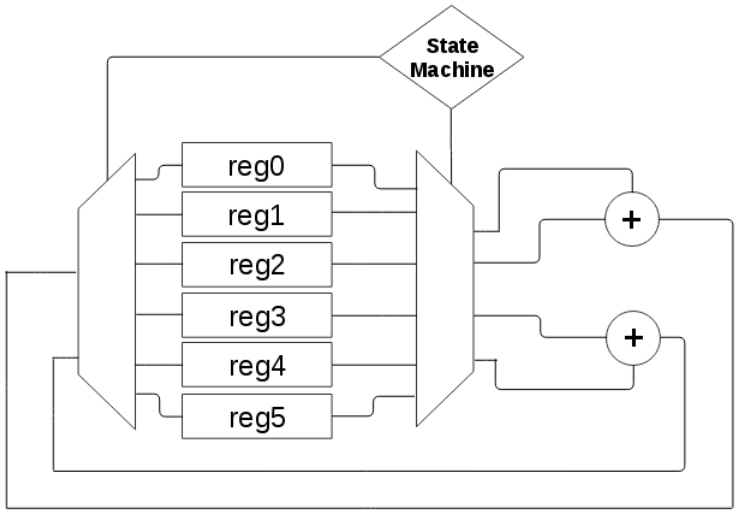

Figure 7: AlphaCalculator stage two Module Block Diagram

3.3.4 Connecting them all together - Durbinator Module

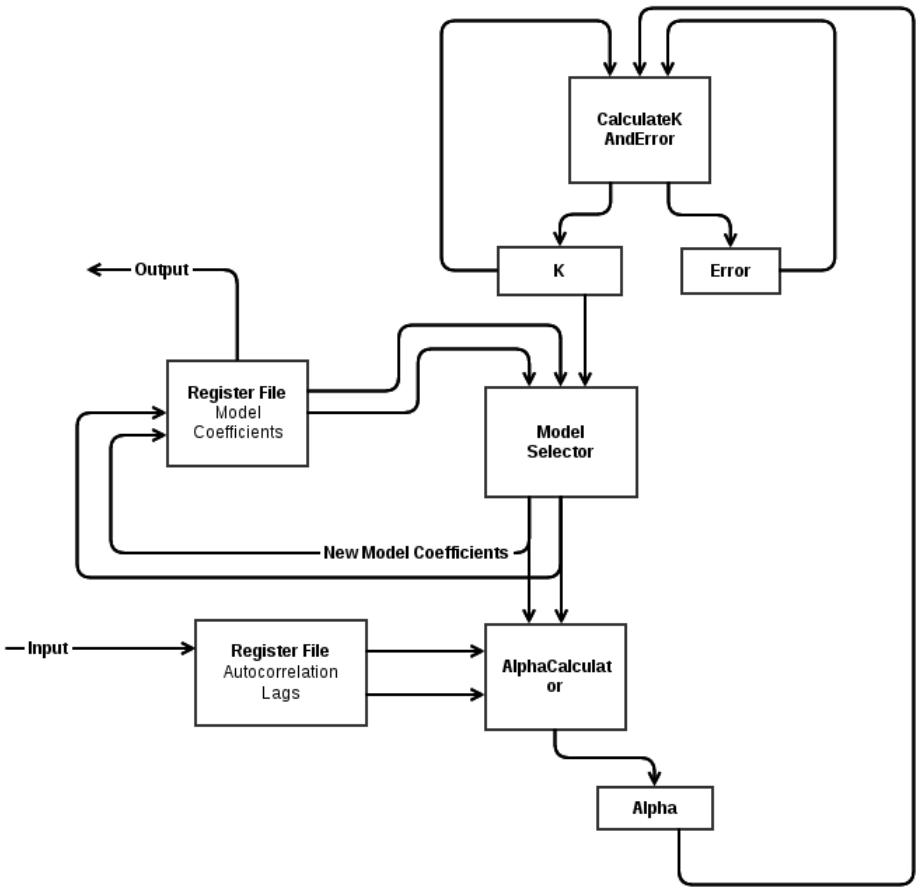

Figure 8: Durbinator Module Block Diagram

The individual elements that make up the Levinson-Durbin model calculator need to be connected together with some glue logic in order to function correctly. The Durbinator module consists of a state machine that controls the flow of data from one module to another in order to correctly sequence the inputs and outputs of each module. Refer to the diagram in Figure 8 for an overview of this process. In addition to connecting the modules together, the Durbinator module also stores and outputs the calculated model coefficients using an addressable register file.

3.4 Stage 3 - Residual Calculator

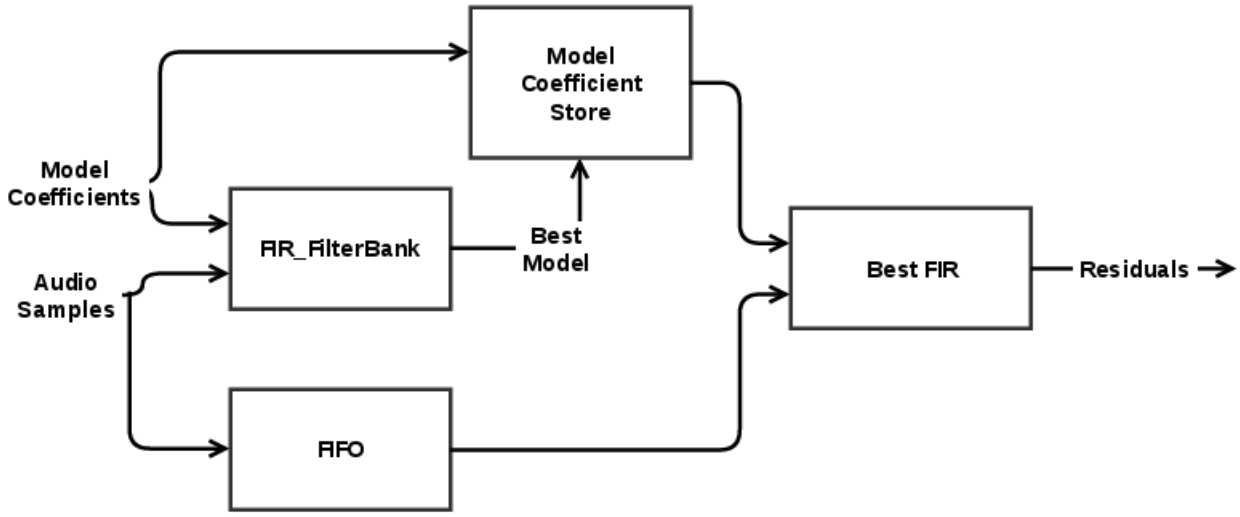

Figure 9: Stage 3 Block Diagram

The residual calculation stage takes the model coefficients generated in Stage 2 and processes the audio samples through each model simultaneously. The model that results in the lowest absolute error is then fed into the final residual calculation stage and generates the residuals.

3.4.1 Module - FIR_FilterBank

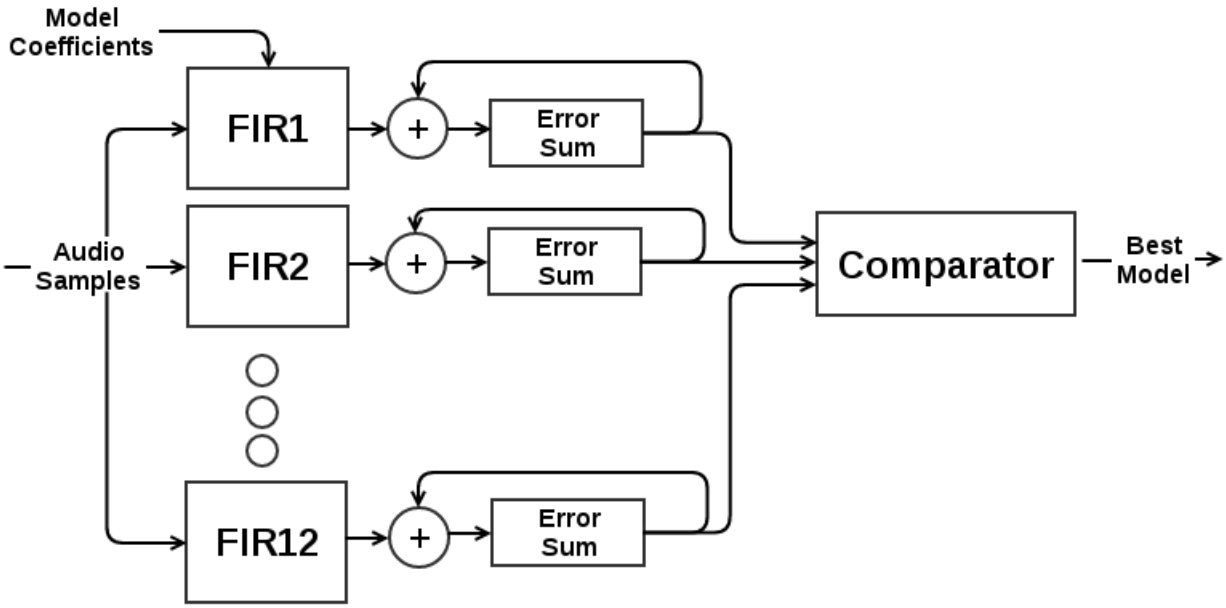

Figure 10: FIR_FilterBank Module Block Diagram

The FIR_FilterBank module consists of twelve pipelined fixed order FIR filters. These filters are loaded with the appropriate model coefficients as Stage 2 generates them. Once

all filters have been loaded, the audio samples are fed into each of them simultaneously. The FIR_FilterBank module keeps a running sum of the residuals produced by each filter and continually compares their magnitudes, selecting the lowest sum to be the best filter.

3.4.2 Module - FIRn

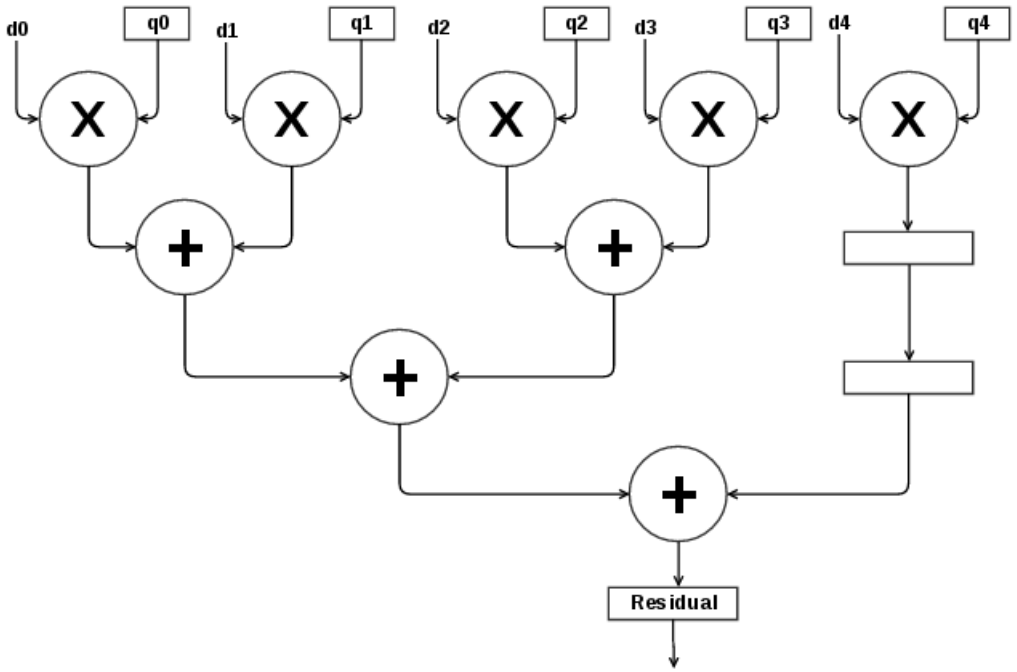

Figure 11: Fifth order FIR Module Block Diagram

A Python script autogenerates the pipelined FIR filters up to the twelfth order. An example FIR filter of order 5 is given in Figure 11. The filters are loaded with the model coefficients into the qx registers. The incoming audio data is then pushed through a shift register $d[order]$ and the dx registers are fed through the multiply accumulate blocks each cycle. Since the model coefficients at this stage are now quantized, integer addition and multiplication modules can be used which greatly reduce the latency and increase the frequency of the design.

3.5 Stage 4 - Rice Coding

Stage 4 takes the residuals generated in Stage 3 and compresses them using the best rice coding parameter. This stage consists of a module similar to FIR_FilterBank that processes the residuals using all rice parameter options and chooses the best one, a module that then processes the residuals using the best rice parameter, and finally a module that packs the encoded residuals into a RAM.

3.5.1 Module - RiceOptimizer

The RiceOptimizer module performs a very similar function to the FIR_FilterBank module; it runs the residuals through each rice parameter and calculates the rice parameter that leads to the lowest number of bits used in the output. Whilst there are methods

to estimate the optimal rice parameter, the hardware cost of simply calculating the best parameter is small enough that it is reasonable to do so. The RiceOptimizer module consists of fifteen parallel RiceEncoder modules that each output the total bits used per residual. The total bits used are then passed through a comparator that selects the lowest total and outputs it as the best rice parameter.

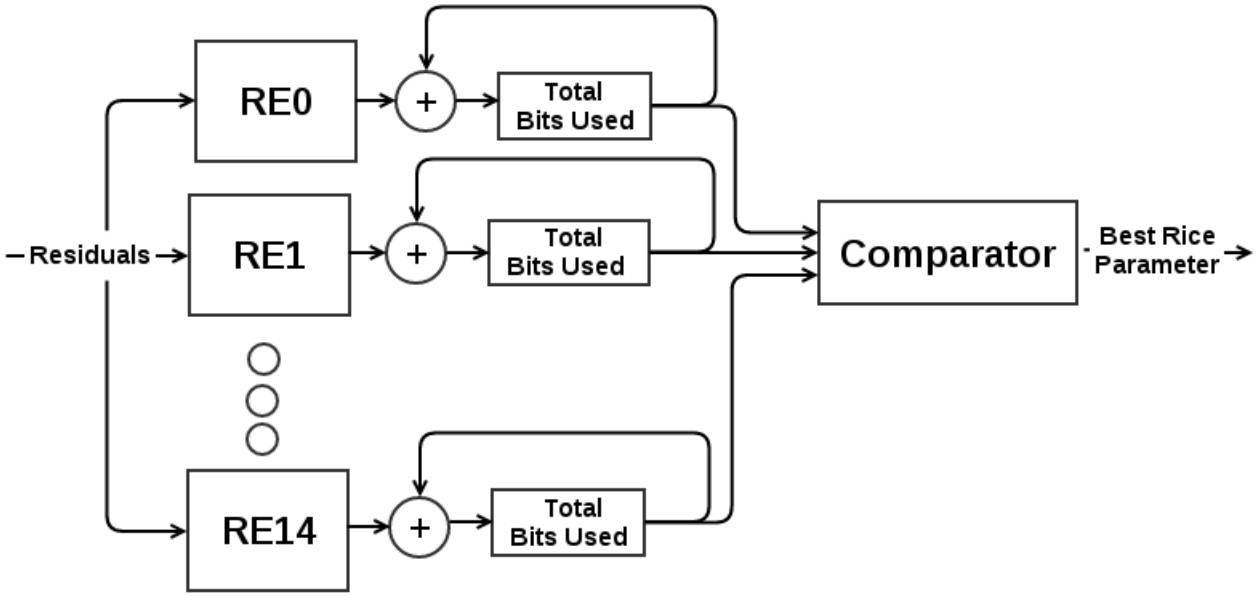

Figure 12: RiceOptimizer Module Block Diagram

3.5.2 Module - RiceEncoder

The RiceEncoder module performs the rice encoding. Given a residual and a rice parameter k it first inverts the sign of the residual and encodes it in the first bit of the unsigned representation. The LSB and MSB of the encoded residual are then output as the lower k bits of the residual and the upper $16 - k$ bits respectively.

3.5.3 Module - RiceWriter

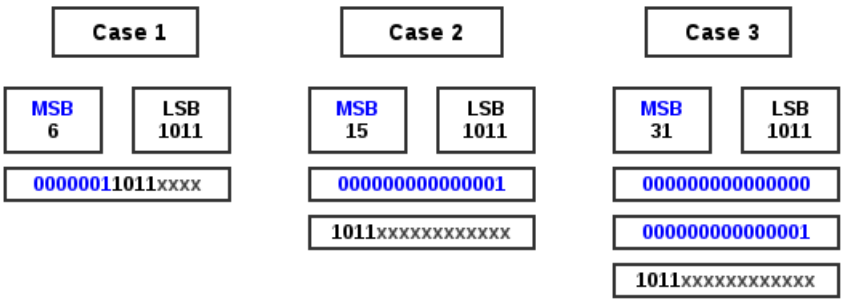

Figure 13: The three possible input cases

The RiceWriter module packs the encoded residuals into sixteen bit numbers and outputs the encoded residuals to a RAM. In order to maintain a throughput of one sample per cycle, the RiceWriter module requires a RAM with dual write ports. The module

functions by packing encoded residuals into a 16 bit buffer and writing out to RAM when the buffer is full. There are three possible cases when processing an encoded residual.

1. There code word fits into the current 16 bit buffer
2. The code word overflows into the next 16 bit buffer
3. The code word overflows multiple buffers

These cases are illustrated in Figure 13. Given that code words can potentially overflow a large number of buffers, it would seem impossible to limit the number of writes to RAM to two per cycle. Is is important to note that in general, the buffer can only overflow more than two buffers due to a large unary part of the code. This means that the buffers that will be overflowed will be all zeros. Thus if the output RAM is assumed to be zeroed, instead of writing each buffer to RAM, buffers that overflow due to the unary part can be skipped over and only the first and last buffers need to be written to RAM. By handling each case in one cycle, high throughput can be maintained.

3.6 Stage 5 - Output

The output stage controls the final output of the compressed data. It takes as inputs the RAM control signals from Stage 4 and uses a double buffered RAM to combine the compressed data with framing metadata as specified in the fLaC format.

4 Performance

Table 1: Performance Summary for each stage

Module	Latency	Frequency (MHz)	Registers	Memory Bits
`Stage 1`	4096	375	1853	65728
`Stage 2`	x	244	8607	28530
`Stage 3`	x	342	12877	65536
`Stage 4`	x	x	x	x

5 Conclusion

References

[1] Library of Congress. Wave audio file format, 2013.

6 Appendix A - Levinson-Durbin Algorithm Derivation