fLaCPGA - An FPGA fLaC Implementation

Emmanuel Jacyna

October 17, 2016

Significant Contributions

- Designed a high throughput hardware implementation of the fLaC encoding algorithm

Contents

List of Figures

List of Tables

Abstract

A novel implementation of an end-to-end hardware lossless audio encoder is presented.

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[1], 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. 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 portable and embedded devices to very cheaply support beyond real time encoding of large amounts of audio data. 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.

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 the encoding time and 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 process audio more than ten times faster than a conventional single core CPU.

2 Purpose

The main purpose of this thesis is to provide a case study in implementing software algorithms on FPGA hardware. The thesis demonstrates the impressive performance gains that can be achieved through FPGA optimised implementations.

3 Paper Organisation

This paper is divided into three main sections. First a description of audio compression algorithms and the fLaC algorithm in particular is provided. This includes a review of the theory of linear prediction and entropy coding which are central to fLaC compression. A short review of prior work in implementing audio encoding algorithms on FPGAs is provided. The bulk of the thesis is devoted to a description of the implemented hardware design, providing an overview of each of the main modules. Finally a performance comparison between the hardware implementation and the reference software implementation of the fLaC algorithm is given.

4 Audio Compression

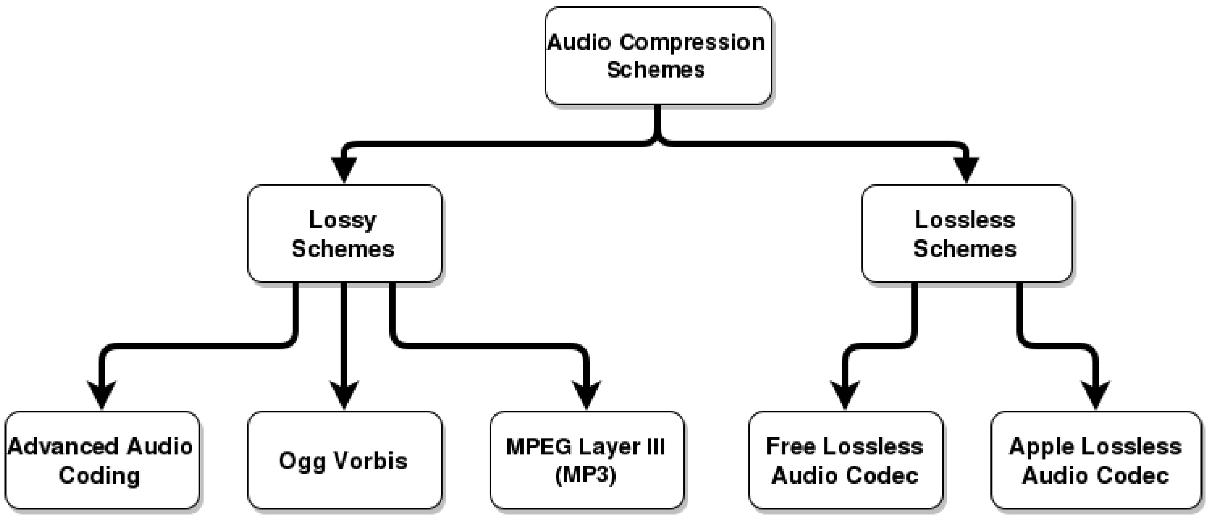

Figure 1: Taxonomy of Audio Compression Schemes

As with 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 a human listener[2]. 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 sound artefacts. Whilst this may be adequate for the transmission of speech 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 30% to 70%[3].

4.1 fLaC Overview

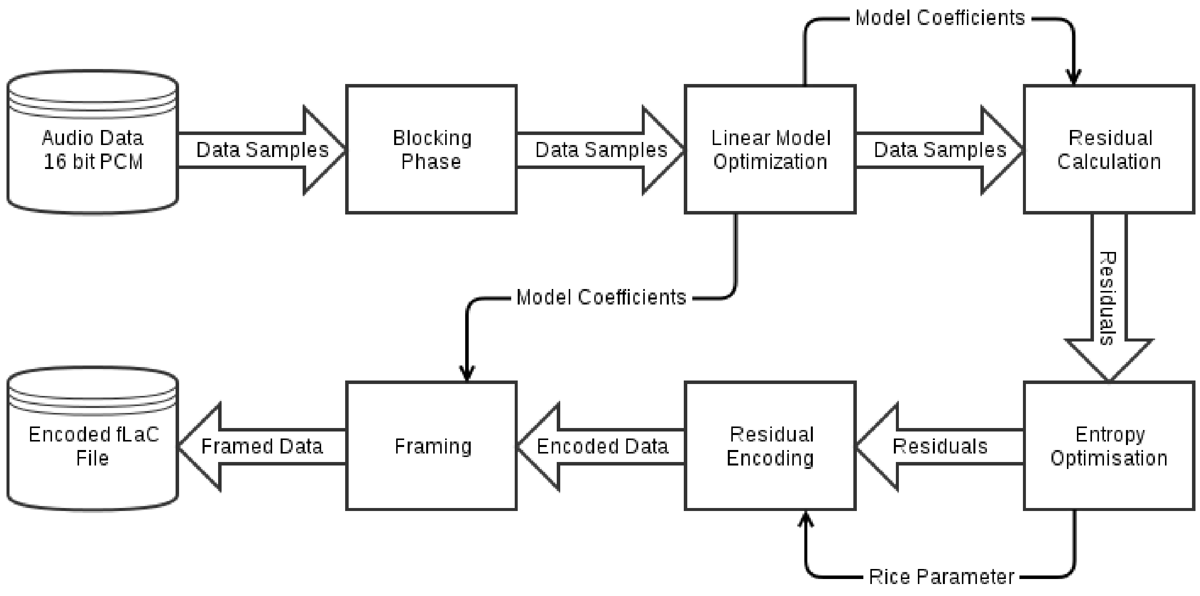

Figure 2: Overview of fLaC Encoding Process

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 much smaller in magnitude than the original signal and in fact will have a different probability distribution to the original signal. This implies that they can be coded using fewer bits than the original audio samples.

The residuals are taken and subjected to entropy encoding. 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 significantly reduced.

The final step is to frame the data. This involves writing metadata such as the sample rate, bits per sample and the model coefficients that allow the decoder to reconstruct the original signal. This information is packed into a *frame header* and written to RAM or to disk, followed by the encoded residuals.

4.2 Theory of Linear Prediction

Given a discretely sampled time domain signal $x[n]$, the linear prediction problem is to obtain a prediction of $x[n]$ as a linear combination of the N most recent previous samples. The predictor will have the form:

$$\begin{aligned} y[n] &= a_0x[n] + a_1x[n-1] + a_2x[n-2] \dots a_Nx[n-N] \qquad (4.1)\\ &= \sum_{i=1}^{N} a_ix[n-i] \qquad (4.2) \end{aligned}$$

where the a_k are the *model coefficients* and the number N is the *order* of the predictor. The prediction error is thus

$$\begin{aligned} e[n] &= x[n] - y[n] \qquad (4.3)\\ &= x[n] - \sum_{i=1}^{N} a_ix[n-i] \qquad (4.4) \end{aligned}$$

We can define the optimal predictor as the set of model coefficients that minimize the mean squared error of the prediction error signal. In the audio domain, this is equivalent to saying that we wish to find the set of coefficients that will lead to a low power white noise error signal. Vaidyanathan shows that the optimal model coefficients a_i should result in an error signal $y[n]$ uncorrelated to the input signal $x[n-i]$[4], i.e.

$$E\big(e[n] \cdot x[n-j]\big) = 0, 1 \leq j \leq N \qquad (4.5)$$

By substituting Equation 4.4 into 4.5 and utilising the linearity property of expectation we can obtain

$$E\Big(\big(x[n] - \sum_{i=1}^{N} a_ix[n-i]\big) \cdot x[n-j]\Big) = 0, 1 \leq j \leq N \qquad (4.6)$$

$$E\Big(x[n]x[n-j] - \sum_{i=1}^{N} a_ix[n-i]x[n-j]\Big) = 0 \qquad (4.7)$$

$$E\Big(\sum_{i=1}^{N} a_ix[n-i]x[n-j]\Big) = E\Big(x[n]x[n-j]\Big) \qquad (4.8)$$

Note that $E(x[n]x[n-j])$ is the autocorrelation of lag j of the signal $x[n]$, $\gamma(j)$. By expanding and expressing the above equation in matrix form we obtain a matrix equation of the form $Ra = r$.

$$\begin{bmatrix} \gamma(0) & \gamma(1) & \dots & \gamma(N-1) \\ \gamma(1) & \gamma(0) & \dots & \gamma(N-2) \\ \vdots & \vdots & \ddots & \vdots \\ \gamma(N-1) & \gamma(N-2) & \dots & \gamma(0) \end{bmatrix} \begin{bmatrix} a_{N,1} \\ a_{N,2} \\ \vdots \\ a_{N,N} \end{bmatrix} = \begin{bmatrix} \gamma(1) \\ \gamma(2) \\ \vdots \\ \gamma(N) \end{bmatrix} \tag{4.9}$$

The solution to this matrix equation produces the model coefficients $a_{N,i}$ that form the minimum mean square error predictor for the signal $x[n]$.

This matrix can be solved through the usual methods such as Gaussian elimination, LU decomposition, and other general matrix solvers. These solvers typically run in $O(N^3)$ time, which can be prohibitive for a hardware implementation as the order of the model increases. The structure of the matrix R lends itself to a faster solution, the *Levinson-Durbin recursion.* This algorithm takes advantage of the fact that R is Toeplitz-symmetric, that is, all elements in the diagonals of the matrix have the same value. By utilising this property, the Levinson-Durbin method can compute the solution in $O(N^2$ time, a significant decrease in computational effort. A derivation of the Levinson-Durbin recursion can be found in Vaidyanathan's text[4]. The equations governing the algorithm 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:

$$a_{0,0} = 1 \tag{4.10}$$

$$\alpha_0 = \gamma(1) \tag{4.11}$$

$$\mathcal{E}_0 = \gamma(0) \tag{4.12}$$

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 4.13 and 4.14.

$$k_m = \frac{-\alpha_{m-1}}{\mathcal{E}_{m-1}} \tag{4.13}$$

$$\mathcal{E}_m = (1 - |k_{m-1}|^2) \cdot \mathcal{E}_{m-1} \tag{4.14}$$

Using these values the mth model can be calculated as:

$$a_{m,n} = a_{m-1,n} + k_m \cdot a_{m-1,m-n+1} \tag{4.15}$$

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

$$\alpha_m = R \cdot \bar{A} \tag{4.16}$$

An example iteration of the Levinson-Durbin algorithm for the fourth order is

4.3 Linear Prediction Example

A graphical demonstration of the whitening effect of the optimal linear predictor is given below in Figure 3. A 4096 sample block of the song "Wake Up" by Rage Against the Machine was filtered using a twelfth order linear predictor obtained using the Levinson-Durbin recursion.

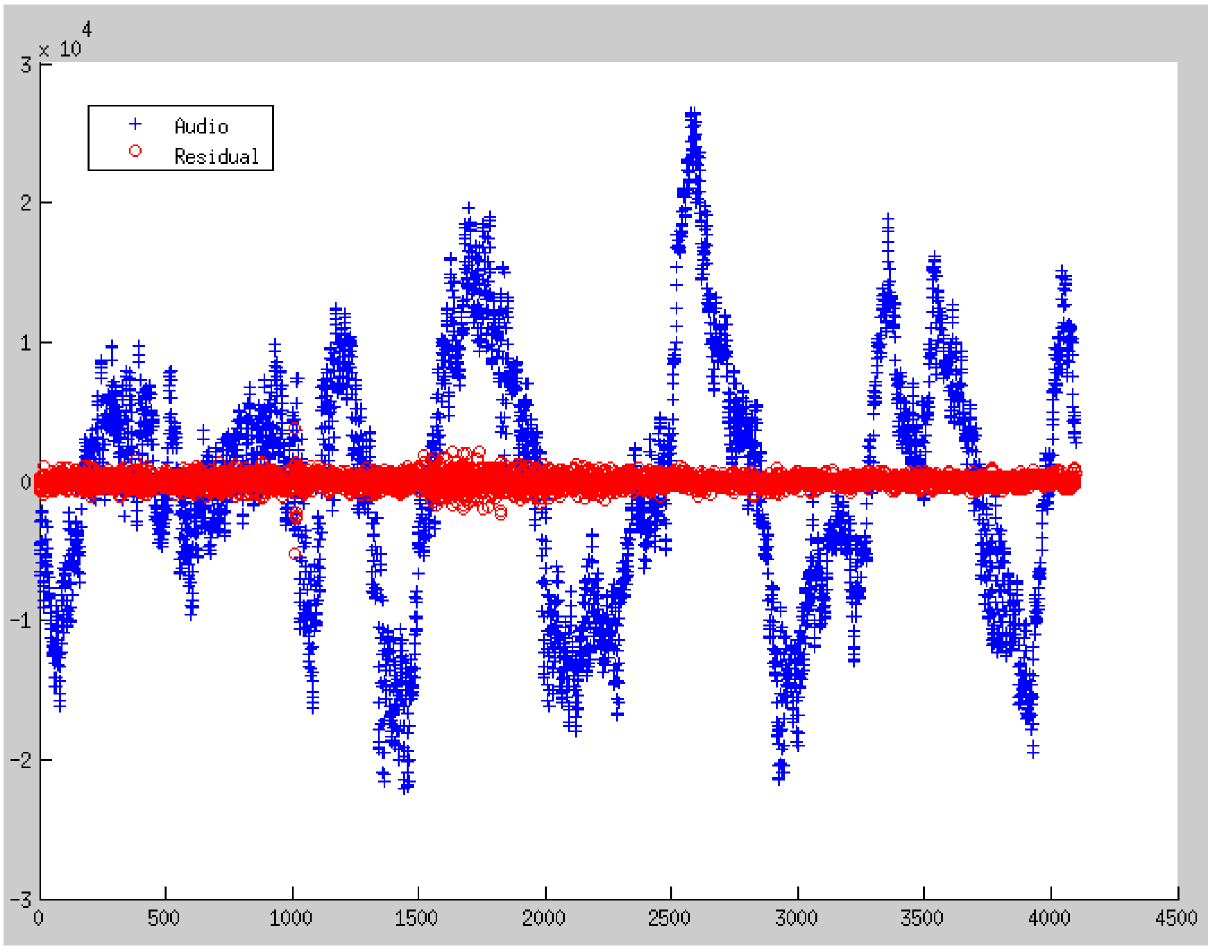

Figure 3: Block of audio filtered through a twelfth order linear predictor

4.4 Theory of Entropy Coding

The Rice codes are a subset of the Golomb codes, a variable length code that is optimal for encoding signals with a one sided exponentially decaying probability distribution[5]. The Golomb code is determined by a parameter k which divides the input integer n into the quotient q and the remainder r. The quotient is encoded in unary and the remainder in binary. The Rice code restricts the parameter k to be a power of two. This simplifies taking the remainder and the quotient to simply taking the lower k bits of n as the remainder and the remaining upper bits as the quotient, which lends itself to a very straightforward hardware implementation. After undergoing the linear prediction step, the resultant residuals will have a two sided exponential distribution due to the negative values in the residual. The residuals are thus remapped into a one sided distribution by the following mapping.

$$M(n) = \begin{cases} 2n, & n \geq 0 \\ 2|n| - 1, & n < 0 \end{cases} \tag{4.17}$$

A histogram of the remapped signal is provided in Figure 4. The signal has an approximately exponential distribution, so is a good candidate for encoding with Rice codes.

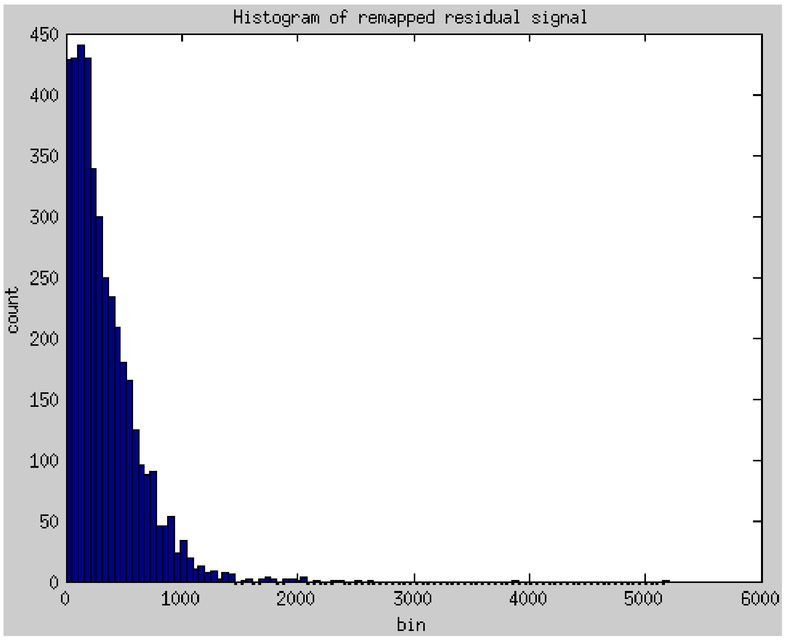

Figure 4: Histogram of remapped residual signal

The Rice codes can be determined by an approximation as given by Weinberger et. al.[6], however, the hardware fLaC encoder simply encodes each residual with all valid fLaC Rice parameters and chooses the parameter that leads to the best encoding.

5 Review of Prior Work

NOTE!! Need to equate the clock speeds and throughputs to MY implementation... remember Mbits and MBytes

The majority of hardware audio encoders are implemented using a System-on-Chip methodology. An embedded CPU performs most of the work, with hardware coprocessors implemented in the FPGA fabric handling the most computationally intensive parts of the process such as parallel filtering or entropy encoding. Because of this it is also relevant to review hardware implementations of algorithms at the core of fLaC, the Levinson-Durbin algorithm, and the Rice entropy encoding method.

Bower[7] provides a typical example of a SoC based audio encoder. His implementation of an encoder for the Ogg Vorbis algorithm, a lossy audio compression algorithm, fits within the framework of a CPU performing the compression with task specific coprocessors. His encoder was able to achieve a 30% performance increase over the software only solution when run with hardware acceleration and was able to achieve real time encoding of 8KHz audio at a clock rate of 25MHz. It is worth noting that the Ogg Vorbis algorithm is a more complex algorithm than fLaC.

Lu et.al.[8] similarly provide an implementation of an AAC encoder, another lossy algorithm, within the SOC framework. Their effort is able to encode eight channels of 44.1KHz audio at real time rates. Similar to Bower, they identify the CPU intensive parts of the AAC algorithm and migrate these to the FPGA fabric. Their design fits into a system with a 160MHz ARM processor.

Xu et. al.[9] implement the Levinson-Durbin algorithm on a Xilinx FPGA as described in. Although the hardware is not comparable, they were able to achieve a clock frequency of 13.4MHz.

Fazlali and Eshghi[10] also provide an implementation of the Levinson-Durbin algorithm focused on reducing resource usage. Because of this they are unable to capitalise on a number of opportunities for parallelism in their implementation of the calculation of the alpha sum and the calculation of the model coefficients. They do not provide any system performance information.

Langi[11] describes a data compression coprocessor using Rice coding. His encoder achieves a throughput of 1.5Mb/s at a clock rate of 10MHz. However his design suffers from a lack of parallelism since it outputs the unary part of the encoded data at a rate of one bit per cycle.

Meira et. al.[12] describe an FPGA implementation of a Golomb-Rice encoder as part of a device to compress electrocardiogram signals. In fact their device also implements a form of linear predictive coding. Since it uses fixed predictors instead of computing the optimum predictor, their predictive coding implementation is not comparable. Similarly to Langi, their implementation of the Rice encoder suffers from a bottleneck due to generating the code word one bit at a time. However they are able to achieve a clock rate of 40MHz, and hence a throughput of 40Mb/s.

The state of the art in FPGA hardware has advanced considerably since the majority of the above works were published. Typical clock rates for FPGAs are now in the hundreds of megahertz. FPGA chips often include dedicated DSP hardware and hundreds of thousands, or even millions of logic elements. Thus a speed focused implementation is likely to achieve significant performance increases in audio encoding.

6 Hardware fLaC Decoder

As part of the fLaCPGA project, a hardware fLaC decoder was implemented prior to the work on the encoder. The hardware architecture of the fLaC decoder is shown in Figure ??. Because of the difficulty of parallelising the decoding of the rice coded residuals, the input fLaC file is fed into the decoder at a rate of one bit per cycle. The decoder consisted of a state machine to parse the incoming data and decode the residuals, and an FIR filter to reconstruct the original audio signal. Most of the hardware design went into the state machine that parsed the fLaC file, as it needs to handle a large number of cases as it analyses the input data. The Rice decoding step proved to the greatest bottleneck in the decoder design. Parallel decoding of the residuals is possible but requires interesting techniques to implement[?]. Even reading one bit at a time the hardware decoder was able to decode much faster than real time. As a result of this the hardware decoder is not able to make much use of the FPGAs parallel architecture. Since the fLaC algorithm is designed to be assymetric, that is, encoding takes much longer than decoding, it was decided to stop work on the decoder and focus on the encoder, as it provides far more opportunities to increase the speed of the design through parallelism and pipelining.

7 Hardware fLaC Encoder

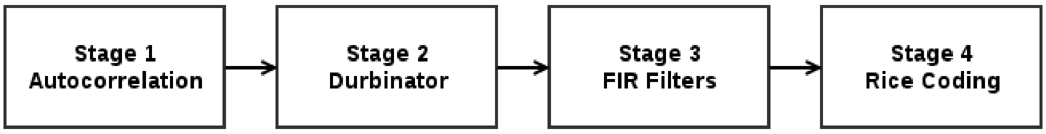

Figure 5: The main stages of the hardware encoder

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 implements 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 difference between the linear prediction model and the actual samples. 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 that packs the Rice coded residuals into 16 bit words and writes them to a RAM.

7.1 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 fifteen bits
- Single Rice parameter over entire block
- Metadata and framing not packed with the residuals

Given the time constraints involved in the project only a subset of the fLaC standard was implemented in the encoder. Inter-channel decorrelation was not implemented. It is possible to encode multiple channels at a time by parallelising the encoder block, but the similarity between channels is not taken into account. Single precision floating point units were used in order to reduce the latency in the Levinson-Durbin module. In hindsight the increased latency and area incurred from double precision math would not significantly affected the performance of the encoder.

Due to the properties of the frequency domain analysis of the autocorrelation signal, the input time domain signal is typically windowed in order to increase the accuracy of the predictor found by the Levinson-Durbin process. Adding Hamming windowing to the autocorrelation step would simply entail an extra multiplication step and a look up table to find the appropriate coefficient. The fLaC standard allows the model coefficient quantization and precision to vary in order to take advantage of the tradeoff between bits used by the model coefficients and the accuracy of the prediction model calculations. These were kept fixed for this implementation as they do not significantly affect the compression ratio achieved and add unnecessary complexity for a first design. Similarly a single Rice parameter was chosen for each block of residuals. It should be fairly trivial to implement Rice parameters that vary over the block, but the gain in compression was not considered worth the extra time expended during the project.

Due to time constraints packing the fLaC framing data with the Rice coded residuals was not implemented. Implementation should be straightforward, but some synchronisation with the Rice packing stage would be required that may take extra time to complete.

7.2 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 7.1.

$$\gamma(n) = \sum_{i=1}^{m} x[i] \cdot x[i+n] \tag{7.1}$$

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 6. 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.

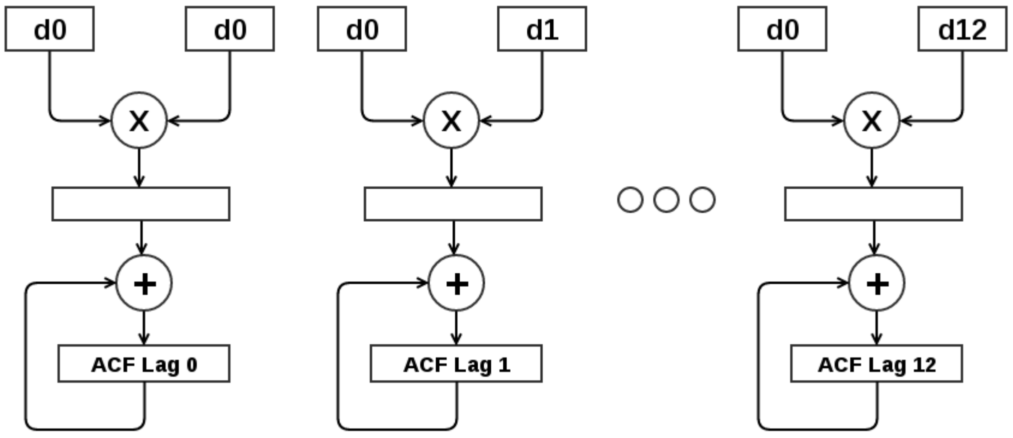

Figure 6: Autocorrelation Module Block Diagram

7.3 Stage 2 - Predictive Model Calculator

The linear prediction calculator implements the Levinson-Durbin recursive method for solving the normal equations of linear prediction. Refer to the relevant Levinson-Durbin equations in Section 4.2.

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`

7.3.1 Module - CalculateKAndError

The reflection coefficient and error calculator is the most straightforward of the steps to implement. The implementation directly implements Equations 4.13 and 4.14 using floating point multiply and subtraction units. The block diagram in Figure 7 describes the circuit. This module suffers the most from a lack of parallelisation as it operates completely serially, with each math operation having to wait for the step before it to complete.

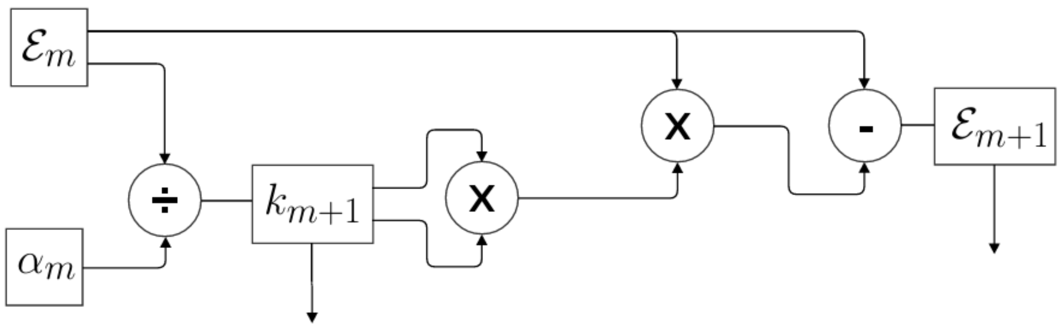

Figure 7: CalculateKAndError Module Block Diagram

7.3.2 Module - ModelSelector

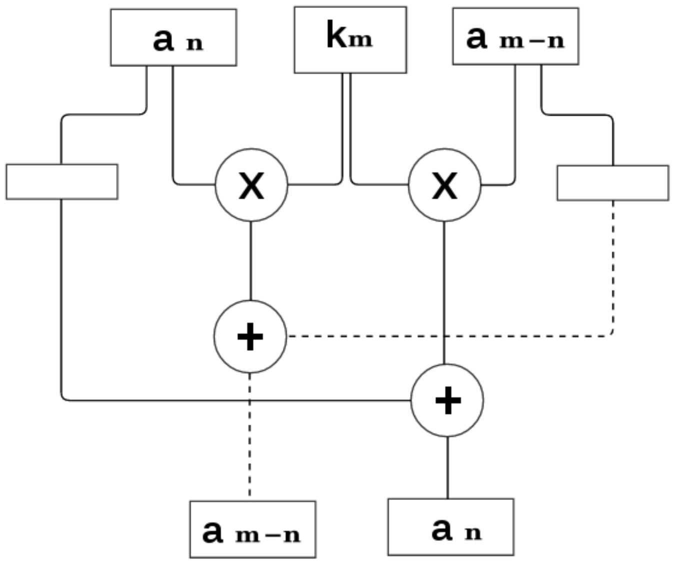

Figure 8: ModelSelector Module Block Diagram

The ModelSelector module implements Equation 4.15. 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 8.

7.3.3 Module - AlphaCalculator

The AlphaCalculator module implements Equation 4.16. 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.

$$\alpha_3 = \gamma(1)a_{3,3} + \gamma(2)a_{3,2} + \gamma(3)a_{3,1} + \gamma(4)a_{3,0} \quad (7.2)$$

The first stage of the AlphaCalculator module is depicted in Figure 9.

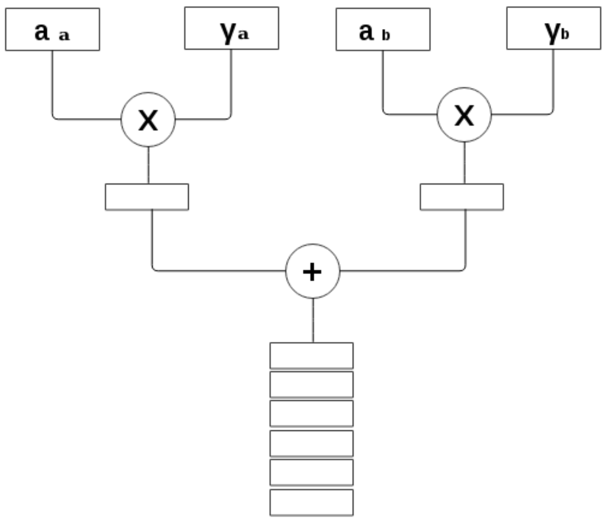

Figure 9: AlphaCalculator stage one Module Block Diagram

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.

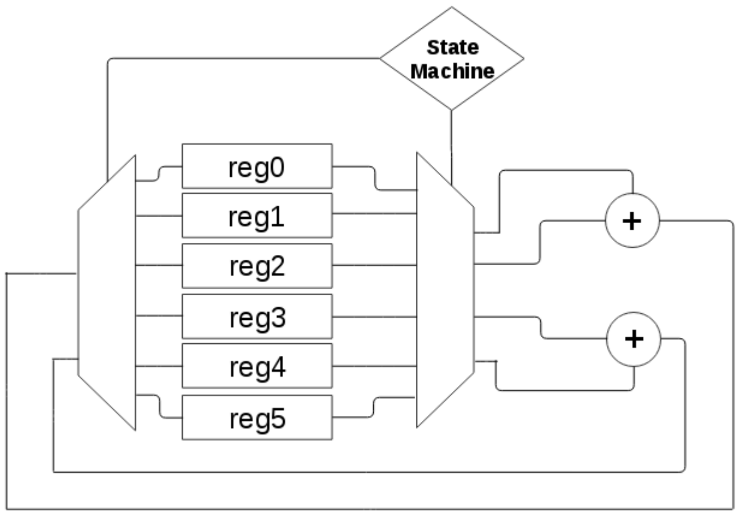

Figure 10: AlphaCalculator stage two Module Block Diagram

7.3.4 Connecting them all together - Durbinator Module

The individual elements that make up the Levinson-Durbin model calculator need to be connected together with support 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 11 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.

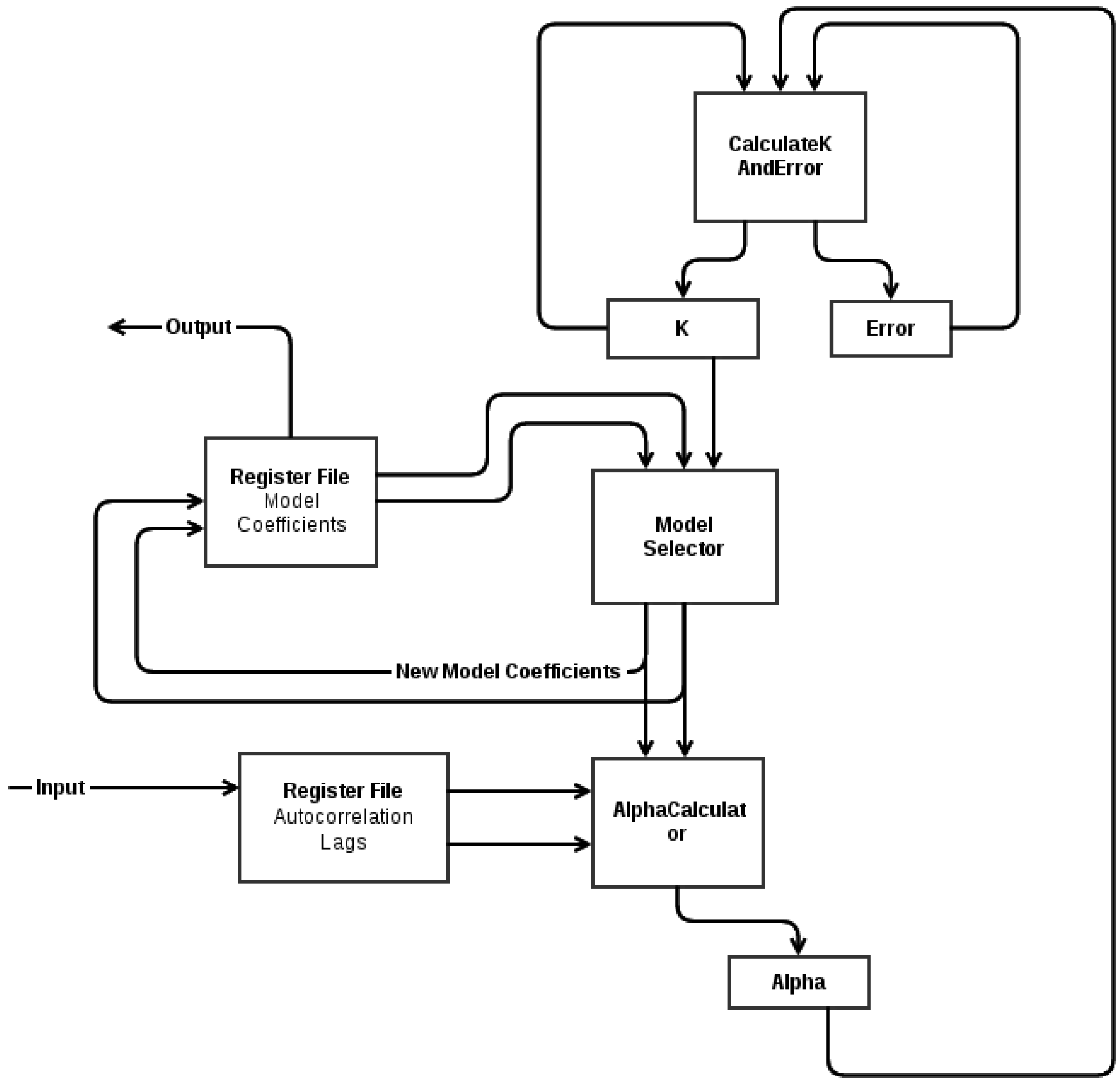

Figure 11: Durbinator Module Block Diagram

7.4 Stage 3 - Residual Calculator

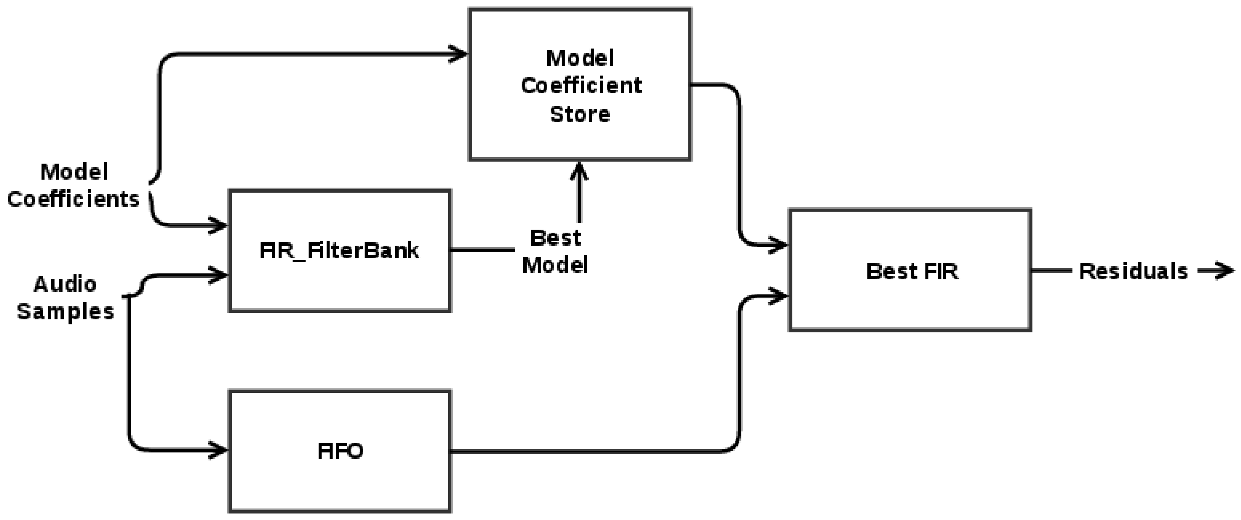

Figure 12: Stage 3 Block Diagram

The residual calculation stage takes the model coefficients generated in Stage 2 and processes the audio samples through each model simultaneously. The final FIR filter stage is then configured with the model coefficients that result in the lowest absolute error. It then processes the audio samples and produces the filtered residuals.

7.4.1 Module - FIR_FilterBank

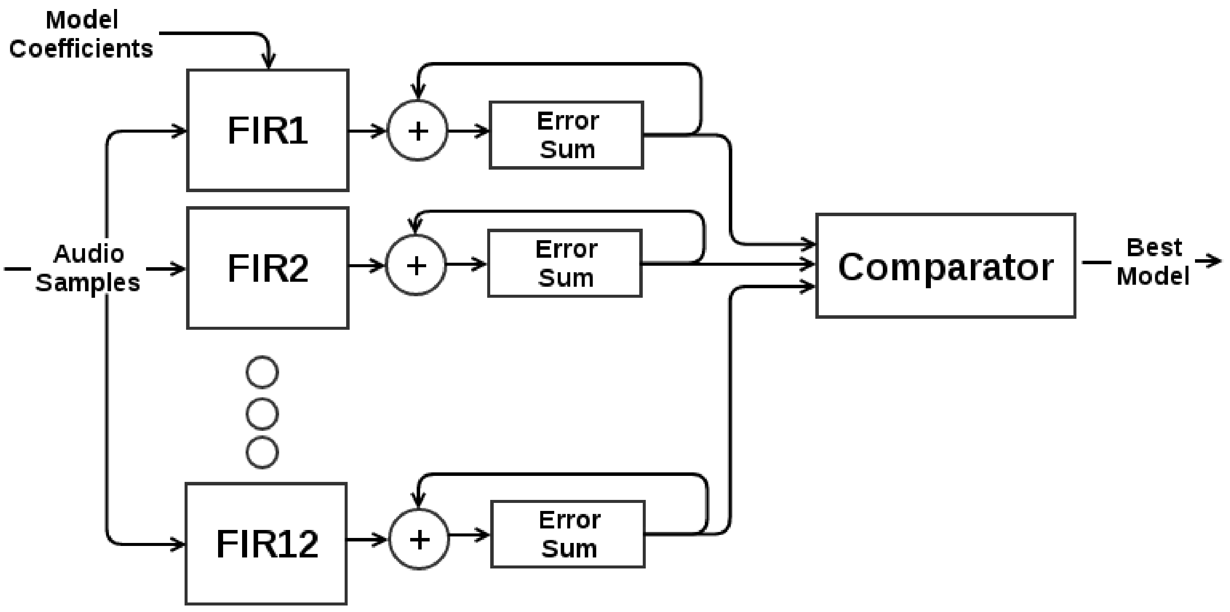

Figure 13: FIR_FilterBank Module Block Diagram

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, they are simultaneously fed with the audio samples. The FIR_FilterBank module keeps a running sum of the absolute error of the residuals produced by each filter and continually compares their magnitudes, selecting the lowest sum to be the best filter. Once the module has processed a full block of audio it outputs the model that corresponds to the best filter.

7.4.2 Module - FIRn

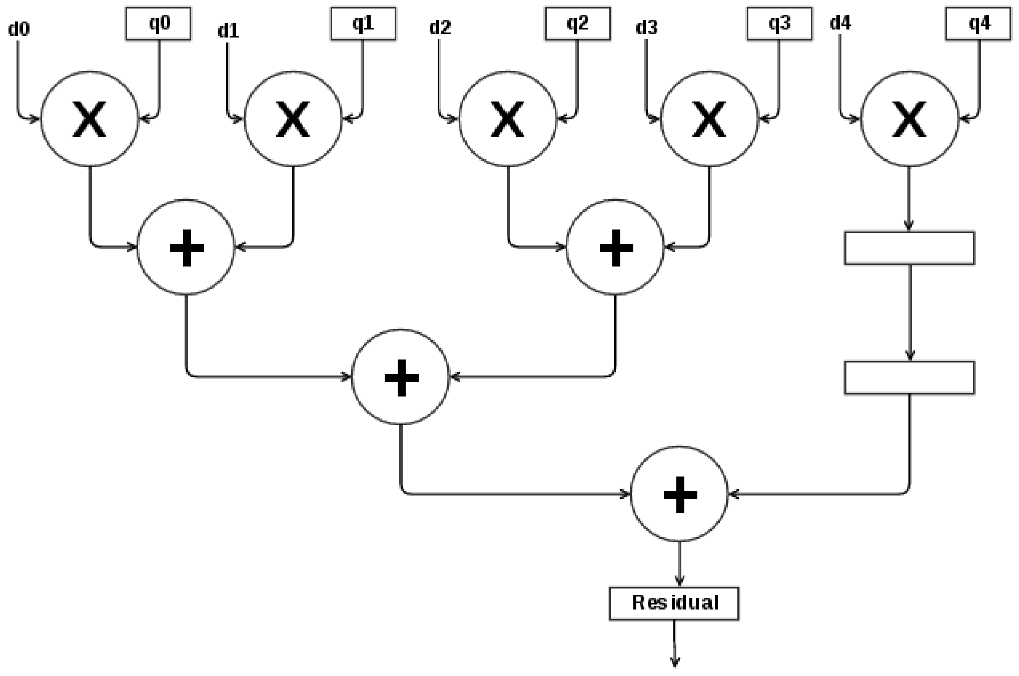

Figure 14: Fifth order FIR Module Block Diagram

A Python script was written that autogenerates the pipelined FIR filters up to the twelfth order. An example FIR filter of order five is given in Figure 14. 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 resource usage of the design. Integer arithmetic also results in a much greater operating frequency as compared to the floating point units in Stage 2.

7.5 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 parameters 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.

7.5.1 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.

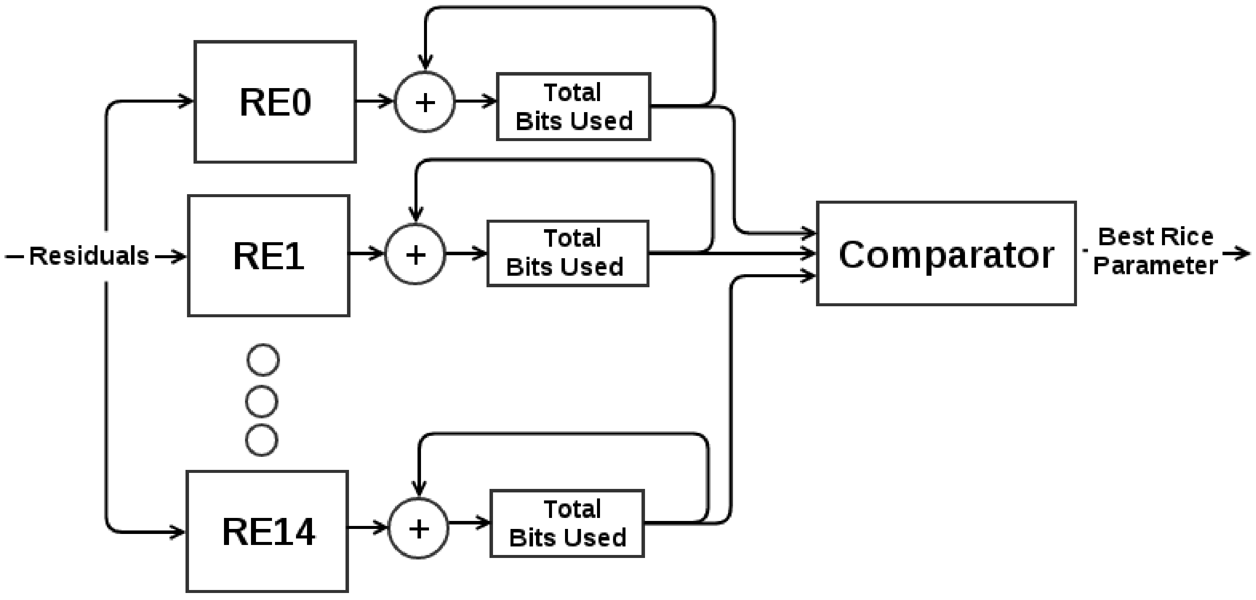

Figure 15: RiceOptimizer Module Block Diagram

7.5.2 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.

7.5.3 Module - RiceWriter

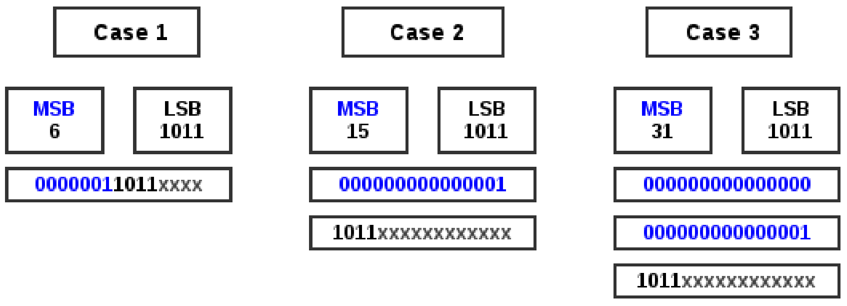

Figure 16: The three possible input cases

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. The 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 16. Given that code words can potentially overflow any 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 is maintained.

7.6 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 multiplexes those signals into two RAM modules. Whilst one RAM is being written to by Stage 4, the output stage is reading data from the second RAM and resetting it to zero. Double buffering provides enough time to both read the data and clear the RAM before it is written to again by Stage 4. The final output of this module is the 32 bit packed Rice coded residuals.

7.7 fLaC Encoder

All these modules are joined together within the fLaC encoder module. This module provides a black box implementation of the fLaC encoder. Audio samples enter, and model coefficients, model order, warmup samples, and the encoded and packed residual data are output. Due to time constraints, full framing of the fLaC data was not implemented, so the fLaC Encoder module outputs the framing data separately from the encoded residuals. Packing the framing data with the encoded residuals is possible but would simply take more time than the project had in hand.

8 Verification

Table 1: Verification Results

Audio	**Blocks Encoded**	**Verified**	**fLaC Compression Ratio**	**fLaCPGA Compression Ratio**
`Wake Up`	252	Yes	73%	74%
`White Noise`	252	No	101% (*incompressible*)	N/A
`Ravel's Pavane`	252	Yes	28%	29%

Since the final output file is not a true fLaC file, verification could not be performed by decoding using the fLaC reference encoder. A testbench was written that feeds the fLaC Encoder module with 16 bit PCM data. The testbench then writes out the model coefficients, warmup samples, and the encoded residuals to separate files. A series of python scripts uses this data to reconstruct the original signal in the same way that fLaC would. The final script saves the reconstructed signal to a file and compares it to the original input signal, flagging an error if the output differs from the input. Finally, the compression ratio achieved by fLaCPGA was calculated, as was the compression ratio achieved by fLaC run with the same parameters that fLaCPGA uses.

Three files were verified, the song "Wake Up" by Rage Against the Machine, a white noise track, and Ravel's "Pavane pour une infante d'efunte". The results of the verification process are shown in Table 1. In all cases fLaCPGA achieved a similar compression ratio to fLaC. The white noise data was, as expected, incompressible by both fLaC and fLaCPGA. fLaC raises an error and warns that the file is incompressible. fLaCPGA in its current form is unable to detect this and will overflow the internal RAM buffers. Since this is a preliminary version this behaviour was considered acceptable.

Overall fLaCPGA performs equivalently to fLaC run with similar parameters. fLaCPGA still needs the logic to detect when compression has failed and write the data out verbatim. As a proof of concept fLaCPGA achieves its goal in producing losslessly compressed audio output.

9 Performance

Table 2: Performance summary for each stage of encoding

Module	Latency	Clock (MHz)	Logic(ALMs)	Memory Bits
`Stage 1`	4096	393	2398	65728
`Stage 2`	1182	253	8790	28530
`Stage 3`	8192	345	13815	65536
`Stage 4`	5120	320	5819	16384
`fLaC Encoder`	18590	248	15,721	225298
`Stratix V Specs`	N/A	1300 (Max)	135,840	19,599,360

The fLaC hardware encoder is able to run at a frequency of 248MHz on the Stratix V model 5SGSMD4E1H29C1. It can process one 16 bit audio sample per clock cycle, thus achieving a throughput of 500MB/s. Whilst the final implementation only processes one channel at a time, the design is very easily parallelisable since blocks of audio can be compressed independently. Adding extra parallelism would simply involve instantiating multiple instances of the fLaC Encoder module and synchronising their output. Ultimately, the design is limited by the I/O bandwidth of the targeted device, as it is able to produce outputs as fast as it receives inputs.

This hardware implementation focused mainly on speed; resource usage was a secondary consideration. Due to the relaxed area requirements, a large amount of parallelism was achieved through duplication of resources. Parallel computation was performed wherever possible. In the autocorrelation module, each lag of autocorrelation is calculated in parallel. In the Levinson-Durbin module parallel adds and multiplies are used wherever possible. The FIR filter optimizer runs each LPC order in parallel in order to choose the best model, gaining a twelve times speed up. The Rice encoder similarly runs through each rice parameter in parallel, achieving fifteen times speedup. A significant consequence of this is that the hardware implementation is able to perform high quality compression just as fast as it performs low quality compression. The choice of implementing high quality compression is simply a question of resource usage. The hardware encoder is able to perform a model search up to the twelfth order with no performance penalty, as testing an extra model is simply a matter of running another filter stage in parallel, whereas in software each filter stage must be run sequentially.

A large portion of the performance increase over the software implementation was achieved by implementing a very deep pipeline. At no stage through the datapath is a module not processing data. The deep pipeline of the system means that while Stage 1 is calculating the autocorrelation, Stage 2 calculates the linear predictor coefficients, Stage 3 is optimizing the model order and also filtering the audio signal, and Stage 4 is continuously producing Rice coded residuals. This kind of parallelism allows a very significant speedup when compared to the reference CPU encoder, as the reference software runs entirely sequentially.

Table 3: Comparison of hardware encoder to software decoder

Processor	**Release Date**	**Clock (GHz)**	**Time (seconds)**	**Throughput (MB/s)**
`Intel Xeon E3-1260L v5`	9/2015	2.9	5.10	45.9
`Intel Core i5-6600K`	9/2015	3.5	5.15	45.4
`Intel Core i5-3210M`	3/2012	2.5	9.15	25.6
`5SGSMD4E1H29C1`	3/2012	.25	.46	500

Software implementation performance data was taken from OpenBenchmarking.org[13]. The benchmark for fLaC consists of three encodes of a 78.1MB WAVE file. The respective throughputs of the software benchmarks were thus calculated as:

$$throughput = \frac{78.1 \times 3}{time} \tag{9.1}$$

Benchmark results are given in Table 3, along with the performance of the hardware encoder. The Xeon is a high performance server processor and the Core i5 is a high performance consumer processor. The release dates of each CPU are given in the table to provide a reasonable comparison to the FPGA. The hardware design is approximately ten times as fast as the software implementation.

10 Conclusion

References

[1] Tidal. How good is the sound quality on TIDAL? `https://support.tidal.com/hc/en-us/articles/201594722-How-good-is-the-sound-quality-on-TIDAL`, 2016. [Online; accessed 10-October-2016].

[2] K Pohlmann. *Principles of digital audio (6th ed.).* McGraw-Hill, New York, 2011.

[3] Josh Coalson. FLAC Format. `https://xiph.org/flac/format.html`, 2014. [Online; accessed 23-May-2016].

[4] PP Vaidyanathan. The theory of linear prediction. *Synthesis lectures on signal processing*, 2(1):1–184, 2007.

[5] R. Gallager and D. van Voorhis. Optimal source codes for geometrically distributed integer alphabets (corresp.). *IEEE Transactions on Information Theory*, 21(2):228–230, Mar 1975.

[6] Gregory K. Wallace. The jpeg still picture compression standard. *Commun. ACM*, 34(4):30–44, apr 1991.

[7] J. Bower. A system-on-a-chip for audio encoding. In *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, pages 149–155, Nov 2004.

[8] Yan-Chen Lu, Chun-Fu Shen, and Chi-Kuang Chen. A novel hardware accelerator architecture for mpeg-2/4 aac encoder. In *Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on*, volume 2, pages 1139–1142 Vol.2, June 2004.

[9] J. Xu, A. Ariyaeeinia, and R. Sotudeh. Migrate levinson-durbin based linear predictive coding algorithm into fpgas. In *Electronics, Circuits and Systems, 2005. ICECS 2005. 12th IEEE International Conference on*, pages 1–4, Dec 2005.

[10] B. Fazlali and M. Eshghi. A pipeline design for implementation of lpc feature extraction system based on levinson-durbin algorithm. In *2011 19th Iranian Conference on Electrical Engineering*, pages 1–5, May 2011.

[11] A. Z. R. Langi. An fpga implementation of a simple lossless data compression coprocessor. In *Electrical Engineering and Informatics (ICEEI), 2011 International Conference on*, pages 1–4, July 2011.

[12] M Meira, J de Lima, and L Batista. An fpga implementation of a lossless electrocardiogram compressor based on prediction and golomb-rice coding. In *Proc. V Workshop de Informática Médica*, 2005.

[13] Phoronix Media. FLAC Audio Encoding Performance Showdown, Automated Performance Comparison. `http://openbenchmarking.org/showdown/pts/encode-flac`, 2016. [Online; accessed 3-October-2016].