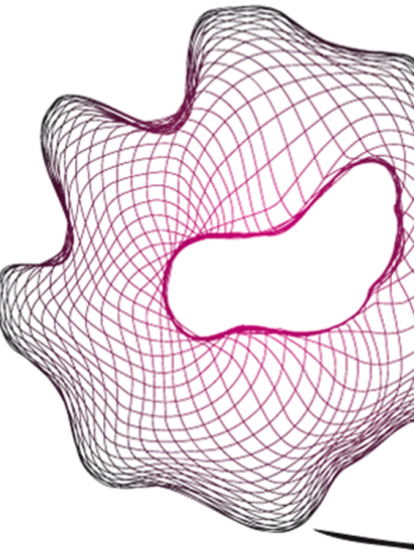
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**



Implementation of Digital Class - D amplifier controller in CλaSH

Anirudh Gottimukkala

M.Sc. Thesis
November 2017

Supervisors:
prof. Dr. ir. J. Kuper
Dr. ir. A. B. J. Kokkeler
Simon - Thijs de Feber (Axign B.V.)
dr. ir. R.A.R. van der Zee

CAES Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

List of acronyms

ADC	Analog to Digital Converter
CD	Compact Disk
CIFF	Cascaded Integrators with Feed Forward summation
DAC	Digital to Analog Converter
DSP	Digital Signal Processing
FIR	Finite Impulse Response
HDL	Hardware Descriptive Language
IIR	Infinite Impulse Response
PCM	Pulse Code Modulated
PWM	Pulse Width Modulator
NPWM	Natural PWM
NTF	Noise Transfer Function
RTL	Register Transfer Level
SNR	Signal to Noise Ratio
STF	Signal Transfer Function
UPWM	Uniform PWM
VHDL	Very High speed IC Hardware Descriptive Language

Acknowledgements

I would like to thank the following people, who had supported me in my course of this research.

- Simon - Thijs de Feber for being my teacher, guide and a patient listener, whose advice on taking steps one at a time was very essential.
- Tim van Doesum for helping me a lot with MATLAB scripts and guiding me with my crucial progress in the final stages.
- Judy and Tim for hosting very nice dinners and for our usual meetings and celebrations.
- Daniel Schinkel for his inputs and patience for pedantic queries.
- Axign, for their continuous trust and support for more than a year.
- Professor Andre Kokkeler and Professor Jan Kuper for their continuous trust and acceptance.

And finally, my parents back in India and my cousin in Eindhoven for having faith in me and for their blessings.

- Anirudh Gottimukkala

Abstract

In this thesis a new way of implementing a Class - D amplifier controller is presented. A Class - D amplifier controller is a system that essentially performs Sigma - Delta modulation to quantize discrete digital input into a 1 - bit Pulse Width modulated signal that is used to drive a power amplifier. The system has internal control loops that perform the necessary noise filtering thus providing quantization noise - free modulated output. This system in a digital implementation traditionally follows digital design flow starting from a VHDL description, which can be time consuming for a controller design that has a greater complexity.

The approach investigated in this thesis involves modeling of the system in a functional programming language Haskell. In the model, elementary mathematical equations that are used to describe the system are modeled in the form of higher order functions. The Haskell code is then converted to a CλaSH (CAES Language for Synchronous Hardware) code, which is a sub-set of Haskell that is used to describe sequential digital hardware. The conversion involves minor changes to the base Haskell model. With its native compiler, CλaSH is used to generate VHDL code from the Haskell modelled system.

Simulations are performed on Haskell and CλaSH models and compared with the simulation of an equivalent Simulink model. The comparison shows that the CλaSH implementation respects the intended functionality of the system.

Contents

Chapter 1

Introduction

Among many applicative areas of signal processing, audio is perhaps one of the most interesting and challenging fields. Right from recording of the source, processing and reproduction of the original audio source, there are many stages and intricate steps involved to achieve a pure audio reproduction. Over time, audio systems dedicated to an efficient audio reproduction moved away from the traditional analog domain to digital, proving faster processing rates and much better elimination of noise in the audio band.

An audio amplifier is an electronic system that reinforces low power auditory signal such that it is strong enough to drive a power load, in this case a loudspeaker. Most audio amplifiers are analog devices, in the sense that the signal is in the from of an analog wave. However, amplifying audio from a digital source requires digital to analog conversion. A class of amplifier suitable for such a task is the Class - D amplifier, which operates on Pulse-Width Modulation (PWM) of audio samples. The amplifier drives a switching power stage to fully ON or fully OFF states, hence giving a theoretical efficiency of 100%. A purely digital solution of this class is therefore faster and efficient.

The design of digital systems start with a general specification of the system with respect to its operational characteristics, energy efficiency, speed of operation and resources consumed, notably among many more. In the next step, the specifications are translated in hardware descriptive languages like VHDL or Verilog. Synthesis tools have been developed that can derive a gate-level hardware implementation of a digital system from a behavioral description in a Hardware Descriptive code (HDL).

The traditional hardware descriptive languages mentioned previously fall short when development time is a crucial factor. Depending on the demands , the system can be quite complex, making the development process much time consuming and error-prone. Hence, the need for a higher level of abstraction is imminent, making the designer focus more on the behavioral aspects and let the tool handle the implementation. To achieve a greater abstraction, functional programming is preferred. In a functional style of programming, the behavior is described as mathematical equations or abstract definitions in the form of basic

functions, and function calls will eventually describe the intended system. The advantage of such abstraction is that the system description is more concise, short and less error prone, and more importantly the development time of complex systems gets much shorter.

A tool that generates fully synthesizeable VHDL code from a specification written in the functional programming language Haskell exists, developed by CAES group in University of Twente. This tool is called *CAES Language for Synchronous Hardware design* (CλaSH), and it gives the opportunity to describe a system at a high abstraction level by describing its mathematical properties using functional expressions. By describing a problem description in a functional programming language, the simulations on the system can be done without any transformations to a programming language with other semantics. The direct transformation from the description in CλaSH to a hardware description in VHDL also eliminates the need for a manual conversion step which can introduce errors in the system.

In this research, a simple Class - D amplifier controller is modeled and implemented in CλaSH. The way to implementation first goes through modeling the required system in Haskell and subsequently implementation in CλaSH. Simulations are performed in their respective native environments to evaluate the correctness and also the performance results, since it will be a point of interest in answering the question:

Is CλaSH a suitable language and environment to design application specific signal processing systems?

1.1 Objectives of research

The primary objective of this research is a demonstration of how a digital Class-D amplifier controller can me implemented in CλaSH. As a start, the system is first modeled in Haskell and eventually translated to CλaSH code. Finally, the extent at which CλaSH can be employed to implement complex architectures can be established.

It is shown that linearity of Class - D operation increases with increasing the sampling rate of the system [1]. However, problems associated with sample rate increase arise during an attempt to obtain a high resolution audio signal. To circumvent this issue, DSP techniques like noise shaping is employed, which is a process by which noise in audio band is reduced while using a low resolution modulator, while using a slower clock speed. This technique is described in Sigma - Delta ($\Sigma\Delta$) Modulation theory.

Concepts of sample rate conversion is quintessential in digital signal processing (DSP) operations, however, these concepts will be briefly described in theory and not as a part of implementation. The research contains following points of discussion:

- Investigating feasibility of modeling DSP structures in a pure functional language.

- Implementation of the architecture in the Functional Hardware Descriptive Language (FHDL) CλaSH.
- Evaluating simulation response of CλaSH with a reference Haskell model, and
- Analyzing synthesis results of CλaSH generated VHDL description of the system.

Analysis of imperfections caused in this system is not discussed as the central point of this research is implementation. However, improvements associated with the system can be a topic of future development.

1.2 Structure of Thesis

The structure of this thesis is presented below.

Chapter 2 gives a brief introduction of digital audio. A general signal chain of audio recording to reproduction is presented. Further explanation is given about Pulse Width Modulator (PWM) amplifiers and their internal workings. Major focus is given on the concept of noise shaping and determination of an optimal noise shaper. The last part of the chapter focuses on modeling the Class - D amplifier controller mathematically and in functional programming language Haskell. This step lays the foundation for implementing the system in CλaSH, which is the primary focus of this research.

Chapter 3 introduces CλaSH as a Functional Hardware Descriptive Language (FHDL). A brief introduction to its similarities with Haskell and advantages of designing digital hardware is presented. Steps necessary to translate Haskell model into CλaSH implementation is laid out. Concluding remarks include the similarities with Haskell and intuitive reasoning of system implementation.

Chapter 4 compares simulation results of Haskell and CλaSH implementations. A quantitative proof is presented and comparisons are made between outputs of each implementation. Furthermore, the results of Haskell and CλaSH models are compared against a Simulink model to establish feasibility and verity of modeling the system in functional environment.

Chapter 5 discusses possible future modifications and directions of improvement for this model. In general, it is to be noted that more efficient higher order functions can be implemented and the system can be greatly improved, for example increasing the order by utilizing the power of higher order formalisms.

Chapter 2

Background

This chapter introduces a brief background on digital audio formats along with a standard digital audio system chain. Furthermore, sections are dedicated to mention the theory of a Class - D amplifier. Also included are fundamentals of Haskell, a functional programming language and CλaSH, which is a functional language for describing synthesizable digital hardware in a functional paradigm.

2.1 Digital Audio

Digital audio is a format that is used for sound recording and reproduction, where analog signals are recored into a digital format. Digital audio over time has replaced analog recording and reproduction techniques in view of its efficiency and speed.

Digital audio processing systems require input in the form of digital encoded values of sampled analog signals. A digital audio system thus starts first with sound converted into an analog signal by a microphone. The analog signal is then encoded into a digital signal by using an Analog - to - Digital converter (ADC). Normally, a digital signal is a Pulse - Code Modulated (PCM) signal that has a resolution depending on the resolution of the ADC which is used. In typical digital audio systems, the resolution used is 16 - bit, and the audio signal is sampled at 44.1 kHz, which gives 44100 samples per second, in case of CD audio. For other formats, different sampling frequencies and resolutions exist. With common digital tools and techniques, the encoded digital values can be stored and/or processed. To output the processed signal, a Digital - to - Analog converter (DAC) is used to obtain an analog equivalent of the digital signal, which is then passed through a power amplifier and given to a loudspeaker or headphones. The entire process of recording to reproducing an audio signal can be visualized in an audio reproduction chain as shown in Figure 2.1.

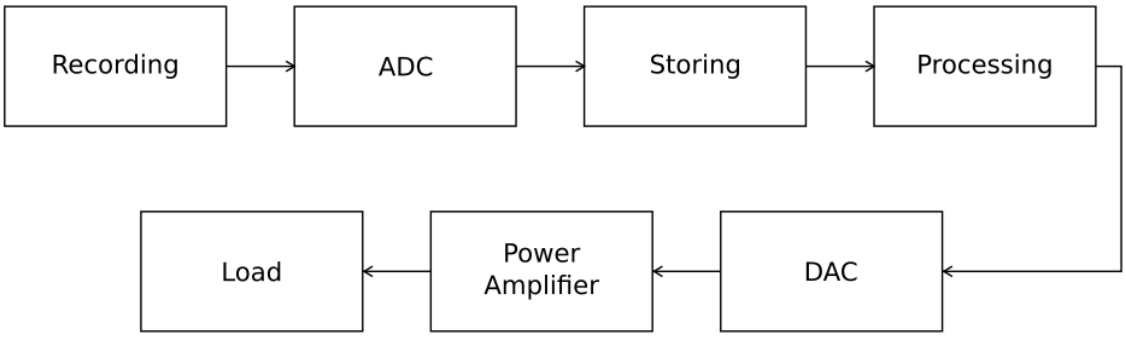


Figure 2.1: Audio recording and reproduction chain

2.2 Introduction to Haskell

A functional programming language, or paradigm, utilizes functions. Executions and calculations are performed by calling functions, that build the required structure. In other words, program execution happens by constant evaluation of expressions that are defined in the said functions. For instance, a function can call another primitive function to evaluate one of its arguments, the result of which is used to evaluate another internally defined operation. One such language is Haskell, which builds the basis for CλaSH in this research.

In imperative languages like C, operations are performed by giving the computer a series of tasks and executions are handled by the computer. Most expressions are evaluated in an 'implied' fashion, that means the program has an internal state and the state can change. For example, a variable can have an initial value, and the same variable can be put into an expression that makes it increment by a value. In a pure functional environment however, the computer or program can only be told what the assignments are. For example, a function defined to calculate a sum of all numbers in a list needs to be told what that list is as an argument, and it results in an evaluated value. Also, the function is guaranteed to return the same value if called with the same argument. This has an advantage, in that the function lacks side - effects. The intended behavior is thus proven at any point of time [2].

A famous attribute of Haskell is it's laziness. A Haskell program does not evaluate functions and provide results unless called otherwise. This is also known as the 'lazy evaluation strategy', which delays the evaluation of an expression until its value is needed. By this strategy, control flow is more abstracted and possibly infinite data structures can be defined. Furthermore, unnecessary calculations can be avoided with lazy evaluation, thus increasing performance of programs [2].

Haskell, combined with its compiler, has more advantages. Most programs during development encounter data type errors. A Haskell program, when compiled, takes care of the data type by type inference, which means that the compiler will determine which variable is of which data type, depending on the assignment. The compiler is also quite extensive in

error reporting, which means debugging is quick as common errors are identified at compile time.

2.2.1 Recursive definitions

An important advantage of describing architectures in functional programming is that functional paradigm supports higher order functions and recursive definitions. From a hardware descriptive point of view, a higher order function could for example be composed of a direct mathematical relation that is instantiated in another function, and so on. A mathematical relation can be anything from a simple adder to modules like full adders that can be called in a carry look-ahead adder structure. With each adder (in this example) having the same mathematical relation inside them, the model becomes more concise and easier to debug. While this is straightforward for combinatorial logic, sequential logic needs to be tackled in a different way in a functional environment, since it does not support loops like imperative environments.

Recursive modeling is the backbone of any Functional Hardware Descriptive Language (FHDL). Most digital designs are evidently sequential, with internal states keeping track of states of data at every clock tick. In an imperative way, they can be modeled simply by formulating a loop with the required number of iterations and the loop takes care of iterations implicitly. Functional languages like Haskell, however, evaluate things differently in this respect, as the evaluation is lazy and functions are pure, i.e, every evaluation with a same input gives the same output. Hence, a sense of previous state has to be introduced. This can be done again with the help of higher order functions, created solely for the purpose of ”storing” a previously calculated value.

A simple example to consider for demonstrating this strategy can be a mealy machine. In a mealy machine, the present output depends on the present input and a previously calculated output, which can be a result of any combinatorial (or mathematical) calculation, for example an accumulator. To model an accumulator in Haskell, two functions are to be declared, one that contains the actual mathematical (combinatorial) addition and the state for one computation, and another to recursively call the previously defined function. Listing 2.1 illustrates the example in Haskell, which models an accumulator.

In this simple code, the function acc is modeled to have a state s and an input a as arguments, and result a tuple of next state s' and present output y as (s', y). Function $simulate$ is also defined that calls any function f, initial state s and an input list as as arguments. A case can now be considered when running $simulate$ with f as acc. It outputs a value y and then calls itself again with arguments acc, updated state s' resulting from function evaluation (single evaluation) of acc, since the tuple (s', y) maps to the resulting tuple evaluation of acc.

Important point to note here is how the next state s' of $simulate$ is mapped to the next state s' of acc. Since the next state s' is simply y, the next state definition becomes $s' = y$. When recursively calling $simulate$ with s', s' takes on the value y and maps it to the present state, and the process repeats. When simulated with a list ranging from 1 to 10, the result is a list with accumulated additions, characteristic of an accumulator. In this way, a state machines can be realized and this fundamental behavior of recursive calling forms an important base for defining $mealy$ function in CλaSH, which will be discussed later.

```
1 acc s a = (s',y)
2   where
3     y  = s + a
4     s' = y
5
6 simulate f s (a:as) = y : simulate f s' as
7   where
8     (s',y) = f s a
```

Listing 2.1: Haskell definition of an accumulator

2.2.2 Functional Hardware Descriptive Languages (FHDL)

Haskell's features like lazy evaluation, recursive definition and polymorphism could be used to describe hardware, as demonstrated in Listing 2.1. This recognition transformed into an idea of developing functional languages around 1980's, where the focus was mainly in reducing the design time by using abstract descriptions of digital hardware. An added advantage of using a functional environment becomes apparent when the process of verifying described hardware is faster and less complex. As a result, over time many functional hardware descriptive languages (FHDLs) emerged which either incorporated syntax and/or semantics of Haskell. There are many such FHDLs existing for a long time, notable among them are ForSyDe and Lava. Of the two mentioned FHDLs, Lava has been used to describe and design hardware for a long time. The fact that Lava can be used not just to describe digital hardware but even implement it by generating a synthesizable VHDL code, made it a long standing choice for FHDL. CλaSH is also an FHDL, possessing the same functionalities as Lava. However, CλaSH directly uses Haskell syntax, whereas Lava has syntax of its own.

2.3 Pulse Width Modulation Amplifier

This chapter describes the concept of a basic Pulse - Width Modulator amplifier, also known as a Class - D amplifier. For brevity, a simple flow of operation is presented to emphasize the workings of the amplifier. In addition, the problems associated with a simple amplifier configuration are mentioned.

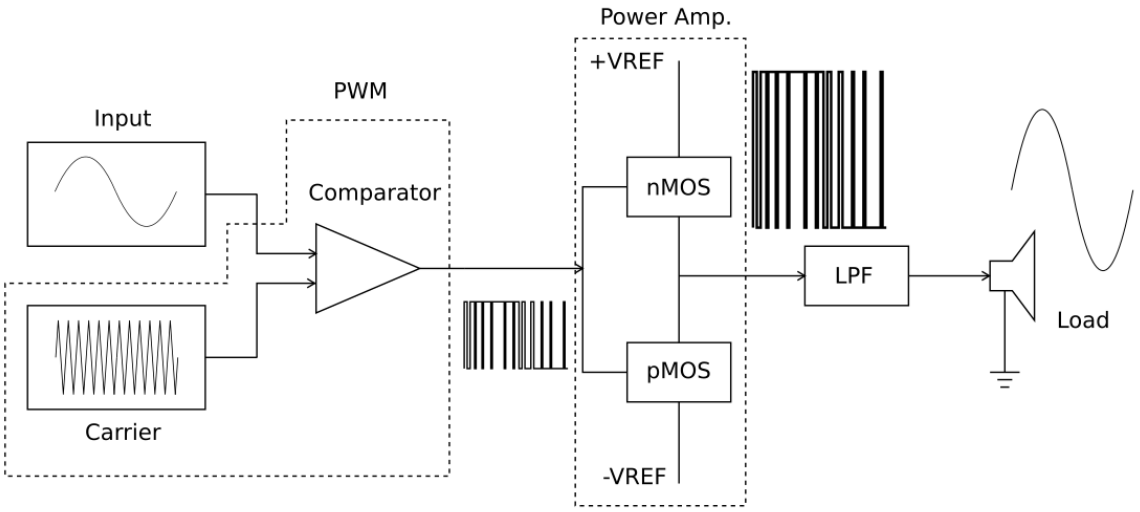


Figure 2.2: Basic Class - D operation

Basic operation

A Class-D amplifier is an electronic system where the amplification devices are, at any particular moment, in fully on or off states. The basic aim of a Class-D operation is to create a train of pulses that is an encoded representation of the input. This is known as Pulse-Width Modulation, where when the pulses are averaged, original data information can be obtained [3]. Operation of a Class - D system can be visualized as shown in Figure 2.2. The system consists of a PWM that encodes incoming signals to two specified levels (usually (-1,1)), by comparing instantaneous levels of the modulating carrier signal and the input signal, at the carrier's sampling rate. The resulting output signal of the PWM is then a rectangular pulse train with instantaneous amplitudes being either of the two specified levels. The pulse train, when driving the amplification devices (power amplifier), produces an amplified version of the PWM output pulse train. This amplified pulse train is used to drive a low pass RLC filter (LPF) to produce an analog equivalent of the input signal.

A digital Class - D amplifier controller is a digital control system that provides in - band noise reduction for a pulse width modulator amplifier. The control system is realized by enclosing the pulse width modulator in a negative feedback loop with a filter structure (loop filter) that is responsible for passing in - band frequency components of the input signal and moving the quantization noise from the PWM out of the bandwidth. A general structure of the controller is shown in Figure 2.3.

The resulting width modulated output consists of just two levels, which appear at a fixed frequency. The time it is high or low states is not always 50%, but it varies according to the instantaneous amplitudes of the signal. This way, when the input signal increases, the high state will be present for longer than the lower state, and vice versa. The mean value of the signal for one cycle is then

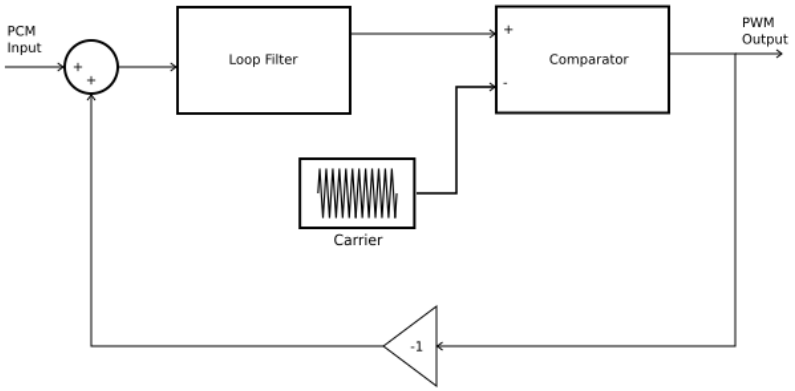


Figure 2.3: Basic digital Class - D amplifier controller

$$V_{avg} = V_h * K + V_l * (1 - K) \tag{2.1}$$

where K is the duty cycle, ratio of ON time and period of the carrier.

As an example, calculation of the mean value of a 50% duty cycle, where both ON and OFF states are present for exactly the same amount of time, with a signal going from +1V to -1V is performed as follows

$$V_{avg} = 1 * 0.5 + (-1) * 0.5 = 0V \tag{2.2}$$

The output of a Class-D amplifier in the absence of input is thus a square signal switching from the positive to the negative rail voltages, with 50% duty cycle. If the input is nearly at the maximum, for example 90%, then

$$V_{avg} = 1 * 0.90 + (-1) * 0.10 = 0.8V \tag{2.3}$$

Pulse-Width Modulation

Pulse - Width Modulation is a method of representing information in the form of a pulse train based on the input's instantaneous amplitude. It is seen as a form of encoding the signal's value in the form of pulses with varying widths. The interpretation of original input value from averaging of pulses is explained in the preceding section. Here, the internal workings of the modulator are presented.

The modulation scheme is detailed by a simple process. Whenever the instantaneous amplitude of the input is larger than the instantaneous amplitude of the carrier, the output of comparator is a high value. Conversely, if the instantaneous amplitude of the input is lesser than that of the carrier, the comparator output is a low value.The carrier can be either a leading-edge or falling-edge sawtooth waveform, or a triangular waveform. Furthermore,

the modulated output depends on the type of the sampling used.

The carrier frequency is to be at least twice the maximum frequency of the input, to satisfy the Nyquist criterion. In practice, however, the carrier frequency is chosen about 10 times the maximum frequency of the input signal bandwidth.

$$f_{carr} \geq 10 * f_{max} \tag{2.4}$$

PWM Sampling

Pulse width modulation sampling is a term representing the kind of sampled modulation that takes place in a pulse width modulator in presence of a sampled input signal. In other words, a type of PWM sampling that depends on how an input reference signal that needs to be modulated is sampled in the first place. Accordingly, there are two types of PWM sampling, one that is based on continuous time and the other based on discrete time reference signals, known as Natural PWM and (NPWM) and Uniform PWM (UPWM) sampling. For analog systems, NPWM inherently occurs in PWM schemes, however for digital systems, UPWM is used as digital systems operate on discrete uniformly sampled quantized levels. NPWM fundamentally differs from UPWM, as illustrated in Figure 2.4.

In Figure 2.4, NPWM and UPWM schemes are presented, where n refers to sampling instants. In an NPWM sampling scheme, the reference signal (shown in red) is sampled at continuous time. In contrast, the reference signal for a UPWM scheme stays at a constant level (shown in blue), since this level is assumed to be a quantized value of the continuous time sampled equivalent. As a consequence, the carrier samples the continuous time reference sample in NPWM earlier, compared to UPWM (in this example), when the carrier is a rising edge sawtooth waveform in this instance. When NPWM and UPWM waveforms are compared against each other, it becomes clear that there are errors present in UPWM scheme (e1 and e2), which contribute to distortion.

From this example, it would seem that NPWM is a better choice to guarantee distortion - free system. However, for a digital system working on quantized values, NPWM cannot be realized and UPWM can, although UPWM gives rise to distortion. To circumvent around this issue, additional processing needs to be performed on the sampled input itself. The idea involves sampling the input at a higher rate such that the presence of higher number of samples will allow more resolution to quantize the input, making the uniformly sampled signal resemble closer to its naturally sampled counterpart. These processing steps are called preprocessing and are discussed in the next section.

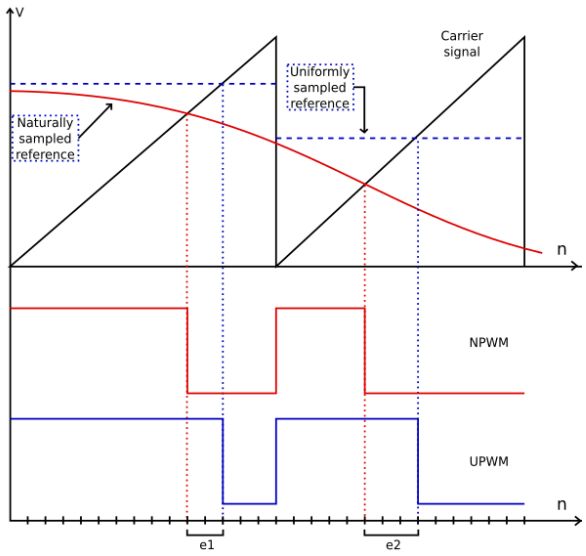


Figure 2.4: Natural and Uniform sampling for PWM

Preproccessing

In the previous section, it has been established that for a purely digital solution, UPWM scheme is realizable. The idea that UPWM operates on discrete samples of data drives a straightforward implementation of the PWM algorithm in the digital domain. However, UPWM suffers from errors as compared to NPWM. The challenge is therefore to design a UPWM scheme that satisfactorily comes close to matching NPWM.

To realize such a UPWM scheme, the incoming audio sample needs to undergo two stages of processing, namely oversampling and interpolation. Oversampling enables the incoming data to be sampled high enough to provide more number of samples than what is required by the Nyquist criteria. Interpolation is a 'reconstruction' of the oversampled data and new data points are extracted from the interpolated signal, at the sample rate matching that of the oversampled data. The presence of a larger number of samples thus makes UPWM operation to be as close as possible to NPWM.

Converting sample rates is quite useful in DSP applications like communications, audio, speech processing and various other multi-rate systems. Sample rate conversions, upsampling and downsampling, exist to increase or decrease the sample rates of a signal respectively. In audio applications, oversampling is quite useful for increasing the frequency bandwidth of the signal, in order to employ Uniform PWM techniques, and eventually certain noise shaping techniques to improve the quality of audio.

In a digital system, upsampling by a factor K can be easily performed by adding K - 1 zero valued samples after each input sample. This is seen as zero-padding, which results in data that can be sampled at a rate K.

Consider an input signal $x(n)$. The upsampling algorithm is then given as

$$w_m = \begin{cases} x(m/K); & m = 0, \pm K, \pm 2K.... \\ 0; & otherwise \end{cases} \tag{2.5}$$

Translating w_m into the z - domain, we get

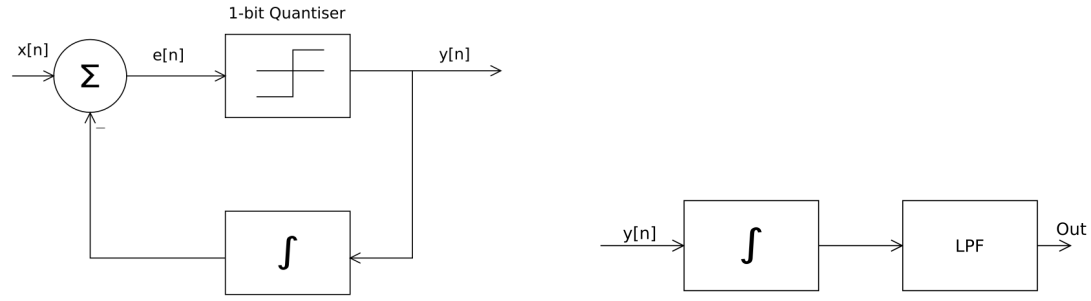
$$\begin{aligned} W(z) &= \sum_{m=-\infty}^{\infty} w(m) z^{-m} \\ &= \sum_{m=-\infty}^{\infty} x(m) z^{-mK} \\ &= X(z^K) \end{aligned} \tag{2.6}$$

This method of increasing the sampling rate is, therefore, much straightforward. However, there are fundamental frequency translated problems associated with it. The effective bandwidth now increases by upsampling, but replicas of the original signal now sit within the expanded bandwidth. This causes aliasing due to the replicas. It becomes evident when spectral content of the upsampled signal is compared to that of the original.

Interpolation in DSP sense is a process of smoothening the areas between two samples. It is a process of constructing a continuous function from discrete points of a signal, or more generally, it is a method of finding missing data within two consecutive samples of sampled data.

Interpolation is a much preferred application in cases where the data's sample rate is to be changed. In the previous section, it has been shown that upsampling data by zero-padding causes replicas of original signal to be present in the expanded bandwidth. The objective is thus to remove those replicas by means of filtering. Filters constructed for this sole application are called interpolation filters.

In DSP theory, many interpolation schemes exist. The basic objective is low pass filtering the upsampled signal, with a passband set to the band of interest, in this case 20kHz for audio, and suppress frequencies beyond that point. Common filter architectures with the said specifications can be employed. FIR filters are easier to implement on hardware where area and resources are not constrained. Recursive filters like IIR can also be employed for the same purpose.



(a) Δ Modulation *(b)* Δ Demodulation

Figure 2.5: Δ Modulation and demodulation chain

Noise Shaping

In previous sections, techniques like oversampling and interpolation of an input signal are used to increase the bandwidth of the audio band. The extended frequency band now enables further signal processing techniques that can be employed to reduce quantization noise by moving the noise outside the audio band, thus improving the quality of the output signal. The technique to perform this operation is known as noise shaping. With this method, it is possible to attain high - resolution audio while running at a moderate bit rate.

Noise shaping is seen as having a high pass characteristic, as it just relocates noise present in the system out of the audio band. This is an important step as the output low pass filter at the power stage filters out noise that has been moved out of the band of interest. The popular method to achieve noise shaping is to enclose the Pulse - Width modulator in a negative feedback loop [4]. The negative feedback enables error correction mechanism that is useful for reduction of noise. This structure is reminiscent of a Sigma - Delta ($\Sigma\Delta$) modulator, where the system can operate on a high sample rate and the quantizer can be 1 - bit.

Digital noise shaping is primarily a Sigma - Delta ($\Sigma\Delta$) modulation by nature. The $\Sigma\Delta$ modulator structure is derived from the established Δ modulator - demodulator structure shown in Figure 2.5 [5]. It consists of a quantizer, loop closed with an integrator in its feedback path. The output of the modulator is then fed to a feed-forward integrator , which is then passed through the usual low pass filter to obtain the filtered output. Delta modulation is based on quantizing the change of the signal at each sample.

Derivation of a $\Sigma\Delta$ modulation from Figure 2.5 is straightforward. The Δ modulation - demodulation process uses two integators. By the property of linearity in integration, the integrator before the output loop filter can be first moved before quantizer at the input of the summation point. After that, the feedback integrator and the integrator at the input can be merged to form a single integrator before the quantizer). The resultant structure is called a $\Sigma\Delta$ modulatior, as shown in Figure 2.6.

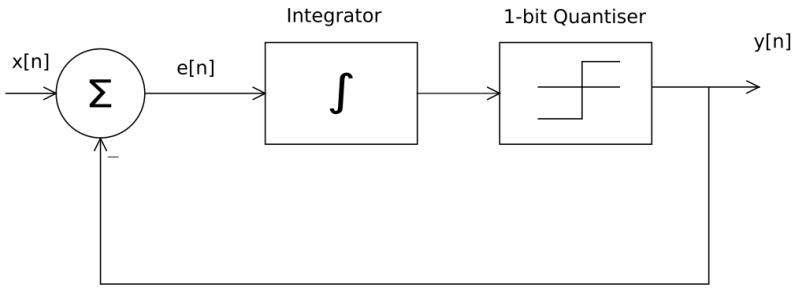


Figure 2.6: 1 - bit $\Sigma\Delta$ modulator

A loop filter is a structure often used in modulators that are required to have a high performance while maintaining a modest resolution. The loop filter replaces the integrator in a $\Sigma\Delta$ modulator by extending its structure to a slightly more complex architecture. There are many standard loop filter architectures for the purpose of noise shaping in a 1 - bit quantizer system. A choice can be made based on requirements like resource utilization or complexity, but for the major part, the design of loop the filter depends fundamentally on deriving transfer functions and extracting the filter specifications from the resulting transfer. Primary requirement for a loop filter is to have a high gain in the bandwidth to ensure a large reduction of error [6].

2.4 Class - D Amplifier Controller

In previous sections, a PWM amplifier was introduced. For this investigation, modeling and implementation of Class - D amplifier controller first starts with formalizing the implementation in terms of a mathematical model, since functional modeling is based on mathematical relations. While mathematical modeling is performed for system analysis, modeling and implementation of the system has a modular approach, wherein each behavior is treated individually. The system primarily consists of a PWM module, which has a comparator and quantizer functionality in - built, along with a loop filter module which is interfaced with PWM in a feedback loop. The translation from mathematical model to architectural model is then demonstrated by performing appropriate simulations on the architectural model.

2.4.1 Mathematical modeling

The mathematical modeling is performed to formulate the intended system behavior in a mathematical description. This step represents the first fundamental point that leads towards developing the Class - D amplifier controller system in a functional environment. The mathematical formulation is presented for describing the relations of the PWM and the loop filter that is included in the closed loop system. The major part of the mathematical formulation rests in describing the loop filter. The Class - D amplifier controller is essentially a 1 - bit

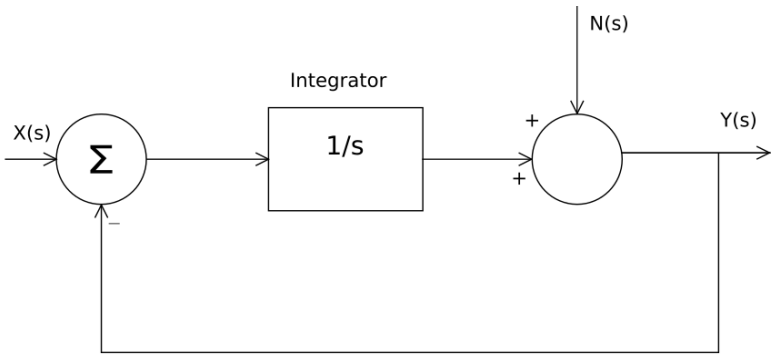


Figure 2.7: 1 - bit $\Sigma\Delta$ modulator s - domain analysis

quantizer with some amount of quantization error reduction due to the inclusion of a filter in the loop, and the architecture is that of a 1 - bit $\Sigma\Delta$ modulator, described in [5]. The loop filter can be designed to be either a simple first order type or a complex n^{th} order type. To achieve a good performance, techniques are described in [7] and [8] which help in understanding the way to design a suitable transfer function behavior for higher order filters. In extension to the above mentioned methods, a loop filter architecture is derived after formulating a loop filter transfer function [9].

In Figure 2.6, a 1 - bit $\Sigma\Delta$ modulator was presented. The 1 - bit quantizer is considered as a noise source that provides the system with quantization noise. The system can then be re-visualized as shown in Figure 2.7, where the quantizer is now replaced with a summing point that adds a noise component $N(s)$. At this stage, the noise shaping mechanism can now be modeled in the s - domain as follows.

$$\begin{aligned} Y(s) &= \frac{X(s) - Y(s)}{s} \\ \frac{Y(s)}{X(s)} &= \frac{\frac{1}{s}}{1 + \frac{1}{s}} = \frac{1}{s+1} \end{aligned} \tag{2.7}$$

$$\begin{aligned} Y(s) &= N(s) - \frac{Y(s)}{s} \\ \frac{Y(s)}{N(s)} &= \frac{1}{1 + \frac{1}{s}} = \frac{s}{s+1} \end{aligned} \tag{2.8}$$

The feedback loop integrates the difference between signal and noise, thereby low - passing the signal and high - passing the noise. This means that the signal is not changed as long as it's frequency is not above the filter's cut - off limits. Equations mentioned above also illustrate the primary characteristic of noise shaping, in that the noise shaper acts as a high pass filter for noise and a low pass filter for signal. Noise shaping, and the modulator in extension, thus requires a wide bandwidth for it to operate, which is only possible by over-sampling the input signal to a high degree.

For formulating in the digital domain, the s domain model of an integrator is translated to

z domain as shown below. The integrator in s domain is an approximation of a continuous time model

$$\begin{aligned} \frac{Y(s)}{X(s)} &= \frac{1}{s} \\ \Rightarrow y(t) &= \int_0^t x(t)dt \end{aligned} \tag{2.9}$$

Realizing an integrator in discrete time is done by considering sampled time kT (T = time period, k = sample) and evaluating integrals over those limits. As a result, $y(t)$ now becomes

$$\begin{aligned} y((k+1)T) &= \int_0^{(k+1)T} x(t)dt \\ &\Rightarrow \int_0^{kT} x(t)dt + \int_{kT}^{(k+1)T} x(t)dt \\ &\Rightarrow y(kT) + \int_{kT}^{(k+1)T} x(t)dt \end{aligned} \tag{2.10}$$

The discrete time approximation can now be translated into the z domain by the Euler integration approximation method. In this approximation, a change of output is calculated over the area under $x(kT)$. This area can also be approximated as a rectangle of total area $Tx(k)$.

$$\begin{aligned} y((k+1)T) &= y(kT) + Tx(k) \\ y((k+1)T) - y(kT) &= Tx(k) \end{aligned} \tag{2.11}$$

After taking z transforms for above equations, the relation now becomes

$$\begin{aligned} zY(z) - Y(z) &= TX(z) \\ \frac{Y(z)}{X(z)} &= \frac{T}{z-1} \\ \frac{Y(z)}{X(z)} &= \frac{Tz^{-1}}{1-z^{-1}} \\ H(z) &= \frac{Tz^{-1}}{1-z^{-1}} \end{aligned} \tag{2.12}$$

Equation 2.12 is the transfer function of a discrete time integrator, and it can be seen as a unit delay with a positive feedback, shown in Figure 2.8. For a single sampling step T, the transfer function simply becomes $z^{-1}/1-z^{-1}$. The Noise transfer function (NTF) for an n^{th} order transfer is given by (from Equation 2.7)

$$\begin{aligned} NTF &= \frac{1}{1+\frac{z^{-1}}{1-z^{-1}}} \\ \Rightarrow NTF &= 1-z^{-1} \\ NTF &= (1-z^{-1})^n \qquad For\ n^{th} order \end{aligned} \tag{2.13}$$

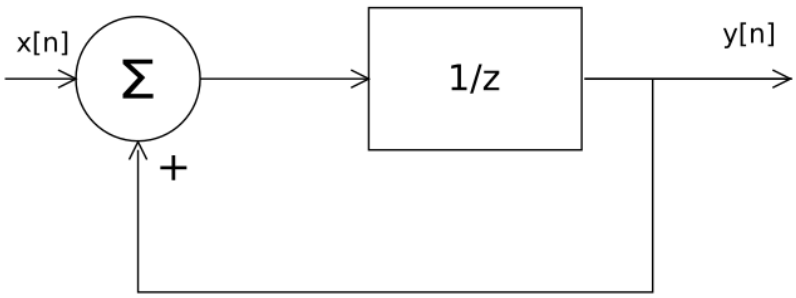


Figure 2.8: Integrator from unit delay

It is known that the higher the order of the noise shaper, the higher will be the SNR, and a modulator which includes an n^{th} order noise shaper in its implementation is known as an n^{th} order modulator. It then would mean that the order could be made sufficiently high, but the straightforward solution is impractical as the size of the filter would be large.

$H(z)$ is seen as a first - order loop filter in the system, but having a single order filter is not sufficient to attain a high signal to noise ratio (SNR) of the system, since the gain provided by a single pole filter (integrator) is not high enough. There are also issues like high frequency components folding back into the system, which are otherwise not filtered out by $H(z)$.

The open loop magnitude response of an integrator is a typical downward sloping response with a slope -20dB/decade. Addition of poles to the transfer function will increase the rate of decay. However, when determining the response, it is desirable to have a flat slope at least in the band of interest. The slope can be flattened out by placing an additional complex pole pair and zero pair at higher frequencies [7] [8]. Addition of complex conjugate pairs of poles can be realized from the transfer function. Any filter with zeros $b_0, b_1, b_2..b_m$ and poles $a_0, a_1, a_2..a_n$ can be realized by the transfer function

$$\frac{Y(z)}{X(z)} = \frac{b_m z^m + b_{m-1} z^{m-1} + .. + b_1 z + b_0}{a_n z^n + a_{n-1} z^{n-1} + .. + a_1 z + a_0} \tag{2.14}$$

Assuming there are no zeros placed, the real poles translate to integrators (first order). Complex poles are then realized as a cascade of two first order sections with negative feedback to each, with feedback gain $(1 - c_1) + (1 - c_2)$, if c_1 and c_2 are the complex pole pairs. The resultant realization is shown in Figure 2.9. This structure is termed as a resonator [9].

The transfer function of a resonator can now be derived as follows. Typically, a resonator is a second order type noise shaper.

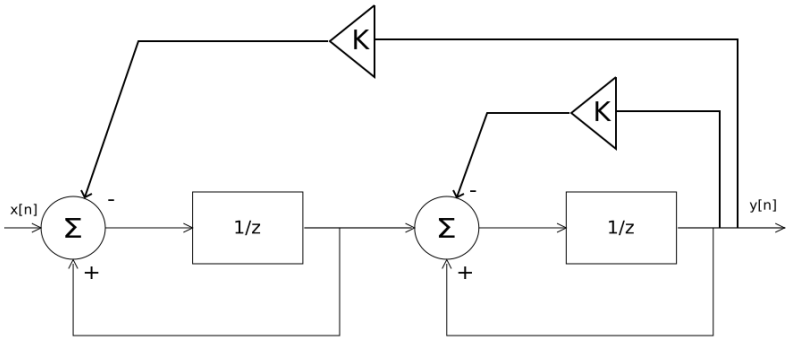


Figure 2.9: Resonator structure

$$\begin{aligned}\frac{Y_r(z)}{X_r(z)} &= \frac{1}{(z-1)((z-1)+K)+K} \\ &= \frac{1}{(z-1)^2+(z-1)K+K} \\ &= \frac{1}{z^2+(K-2)z+1}\end{aligned} \tag{2.15}$$

There exist a variety of filter transfers to implement in the modulator. For a higher order 1 - bit $\Sigma\Delta$ modulator, there are four typical structures of higher order loop filters that can be implemented, varying in complexity and size. Since the focus is on implementing a filter with resonator section, a Cascaded Integrators with Feed Forward summation (CIFF) is chosen. The CIFF structure is realized by interfacing an integrator with a resonator, in that order. This is due to the fact that an integrator can provide the highest dynamic range over the required bandwidth, and a resonator can keep the response of the passband flat. The filter is realized with the following transfer function, and architecture resembling Figure 2.10, which is a third order filter consisting of a first order section (integrator) and a second order section (resonator).

$$H(z) = \frac{1}{z-1} + \frac{1}{z^2-2z+K+1} + \frac{1}{(z^2-2z+K+1)(K-1)} \tag{2.16}$$

2.5 Functional Hardware Descriptions in Haskell

2.5.1 Modeling in Haskell

Modeling in Haskell follows four steps that involves PWM, loop filter, system level modeling and finally simulations for analysis. In subsequent sections, the straightforward implementation of the modules and the system are described with floating point data types, as the focus is primarily on modeling than on performance.

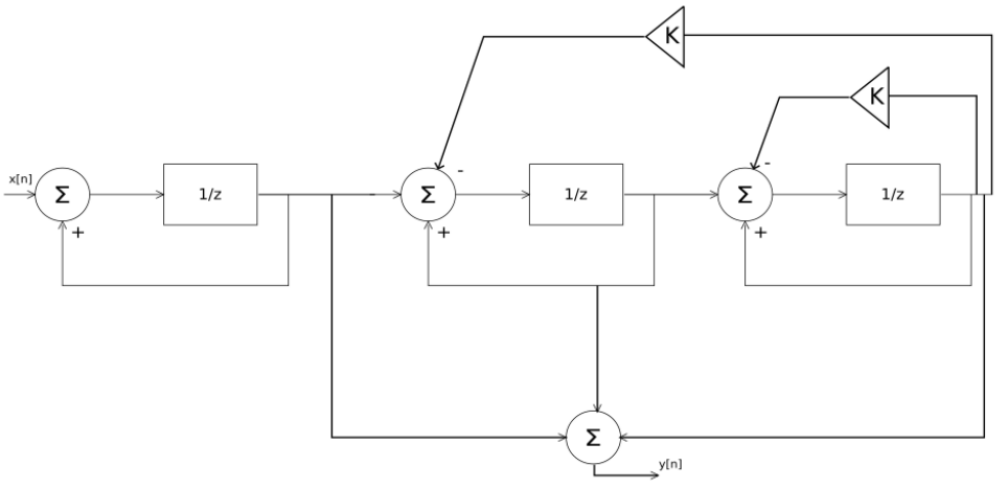


Figure 2.10: 3rd order CIFF structure

Pulse - Width Modulator

The module definition for pulse - width modulator is defined in Listing 2.2. This module contains a triangular wave generator and a comparator that compares instantaneous values of input sample with instantaneous sample of the triangular wave generator's output.

```
1 module Pwm
2 ( pwm      -- main pwm function
3 , Slope
4 ) where
```

Listing 2.2: Module PWM definition

The module exports the function pwm and data type $Slope$ to the top level. The function pwm is responsible for generating a triangular wave, which is based an example presented in [10], which is a Haskell model of an up - down counter. A triangular wave is chosen as the carrier wave, as the objective is to perform double - sided PWM modulation. The definition of pwm is given in Listing 2.3.

```
1  data Slope = Up | Down
2
3  pwm :: (Num a,Fractional a,Floating a,Ord a) => (a,Slope) -> a -> ((a,Slope),a)
4  pwm (v,s) x = ((v',s'),y)
5    where
6      v' = case s of
7             Up   -> v + 1/64
8             Down -> v - 1/64
9
10     s' = case s of
11            Up   | v' < 1.0    -> Up
12                 | otherwise   -> Down
13            Down | v' > -1.0   -> Down
14                 | otherwise   -> Up
15
16     y
17       | (x - v') >= 0.0 = 1.0
18       | otherwise       = -1.0
```

Listing 2.3: pwm definition

The type declaration for pwm is defined so as to make it polymorphic. Polymorphism in Haskell comes from defining function types with type variables. In $triM$, the type variable is considered to be a. The type declaration then states that a can be any number, as long as it is of numerical type Num, which is defined in the standard Haskell library $Prelude$. The type definition further defines variable a to be as $Fractional$ and $Floating$ types, which support fractional and floating point operations. The function $triM$ also involves comparison of arguments, hence the typeclass Ord is also used to declare variable a to support $>$ operation.

Mentioning a with the additional types $Floating$ and $Fractional$ ensure variable a to be a floating point number and that which can support fractional values and comparison as well. This means that effectively a is constrained to represent a floating point number, and the process of putting such constraining a variable is done by declaring additional typeclasses.

After establishing the type of variables pwm operates on, the type definition of pwm is to be further extended with type behavior. pwm takes a tuple (v, s) containing a floating point number v and a state denoting a switching state (explained later) s, and a value x as arguments, and the evaluation is the resulting tuple $((v', s'), y)$, where y is the present output. The switching state s and s' (next switching state) is defined by a user-defined datatype $Slope$, which can at a time take Up or $Down$. Accordingly, the type definition for tuple argument (v, s) will be $(a, Slope)$, n will be a and finally $((v', s'), y)$ becomes $((a, Slope), a)$.

In Listing 2.3, v is a present initial value, v' is the next step of increment, s is the initial direction of increment and s' is the next direction of increment defined by datatype $Slope$,

which at any point of time can take up either of the fields Up or $Down$, thus signifying the direction of traversal. The step size is an important parameter, as the step size effectively determines the frequency of the resulting wave, as described in Equation 2.17. Here, n refers to number of steps of increment for 0 to maximum amplitude traversal or 0 to minimum amplitude traversal. Hence for a single cycle, there are $4n$ number of traversal steps.

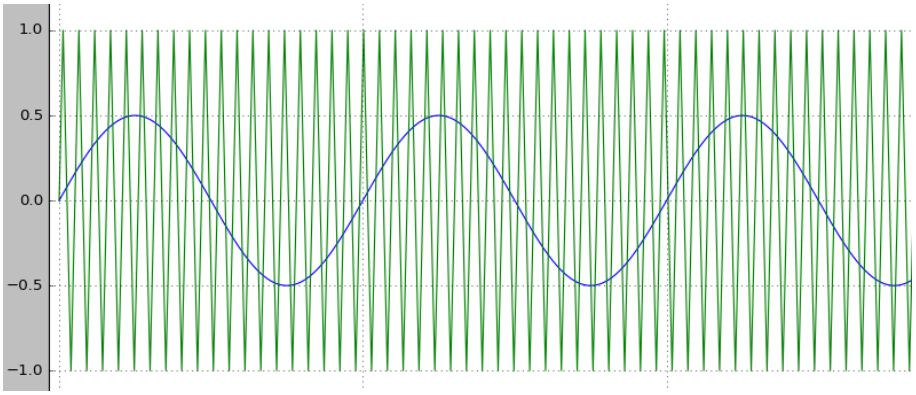
$$f_{tri} = \frac{f_{clk}}{4n} \tag{2.17}$$

The triangular wave is generated as follows. The function is initialized in a predefined start condition, in this case, value 0 and direction of increment as Up. Since the specified incremental direction is Up, the next calculated output will be a positive increment of the present value, with a fraction of step size. In this juncture, it is important to note that the fraction of the step size depends on the final peak amplitude of the required wave. For this research, the requirement is 2 V peak to peak, meaning the maximum traversals of the wave should be +1 and -1 V. Hence, increment step becomes $1/n$. For a peak value k, the increment is k/n.

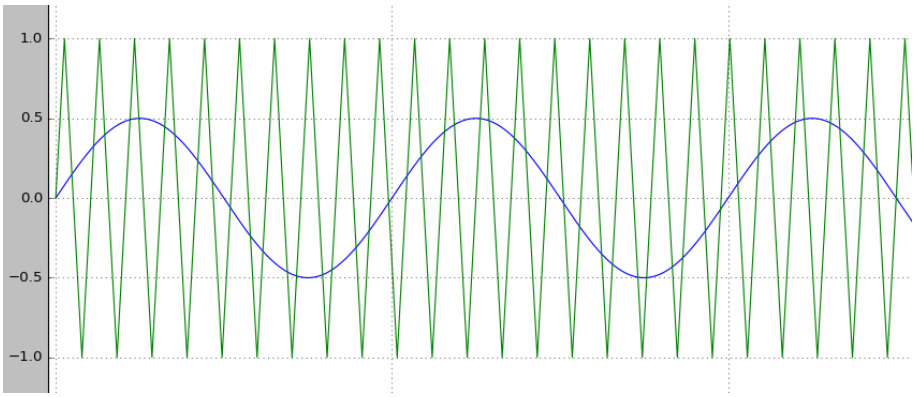
Along with the aforementioned process, pwm also takes care of direction switching. It constantly evaluates the next traversal direction by comparing the output value against the required peak values. If the present traversal is Up and the output is less than the peak, the direction switch stays Up else it switches to $Down$. When the traversal is $Down$, the present output is decremented with the same traversal step until it matches the minimum peak amplitude, and the switch turns to Up again. In this way, a triangular wave is created.

In the introduction for PWM, it is mentioned that in practical applications, the frequency of the carrier should be high enough, about 10 times or more, than the highest frequency component in the bandwidth. For this PWM model, the simulations for triangular wave generation are carried out by considering two values of n as 32 and 64, by providing an input sine wave at 20 KHz, since it is the highest frequency considered in the audio bandwidth. The simulation results are shown in Figures 2.11a and 2.11b respectively. From the simulations, it can be seen that for n as 64, the triangular wave is nearly 10 times the frequency of the input and for n as 32, the triangular wave is about 20 times. For the purpose of implementation, n as 64 is considered sufficient.

Second phase of the PWM module is comparison. This functionality is defined by the declaration for y which compares the input x and current triangular wave sample value v' by first subtracting them and comparing the resultant difference with 0. If the difference is ≥ 0, the output is +1, otherwise -1. The definition of function y is given in Listing 2.4.



(a) 32 steps per rising/falling edge



(b) 64 steps per rising/falling edge

Figure 2.11: Simulation results for n = 32 and n = 64

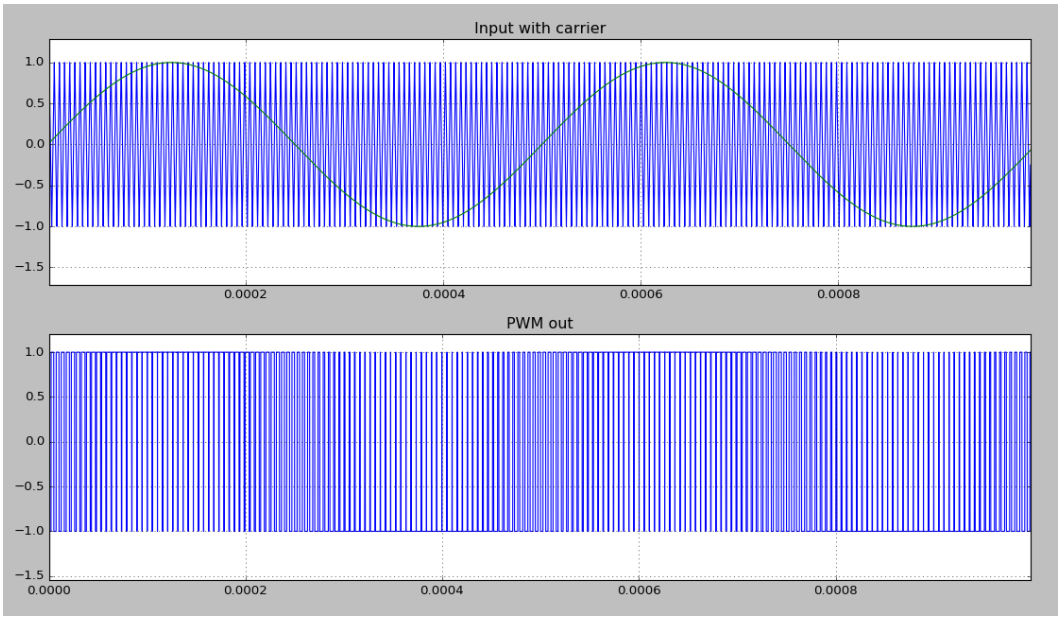


Figure 2.12: PWM output

```
y
  | (x - v') >= 0.0 = 1.0
  | otherwise      = -1.0
```

Listing 2.4: pwm definition

The function pwm is simulated with an input with frequency 2KHz and a triangular wave carrier with 64 steps per rising/falling edge (n), and the result is shown in Figure 2.12. It can be seen that the result of pwm function varies in widths corresponding to the duration of the input staying high or low.

Loop Filter

Module definition of the loop filter is given in Listing 2.5. Similar to PWM, it exports function $lfilter$ to the top level for system integration.

```
module Lfilter
(lfilter
)where
```

Listing 2.5: Lfilter module definition

Listing 2.6 shows the model of the $lfilter$ function. The loop filter architecture is a modified version of the one presented in Figure 2.10, where intermediary gains are inserted to ensure loop stability, resulting in a structure shown in Figure 2.13. Function $lfilter$ models mathematical operations that are essential to the architecture in Haskell definitions. Similarities can be observed between Figure 2.10 and $lfilter$, which is explained below.

In Section 2.3, the theory of loop filter is presented wherein the loop filter that is considered for this case consisted of a first order integrator cascaded with a resonator, and the transfer functions of both sections were derived in the z domain. For realizing the $lfilter$ model from the mathematical descriptions of the loop filter stages, the transfer functions of each stage is first converted into discrete time domain. Equation 2.18 shows the discrete time model of the first order integrator stage, and Equation 2.19 shows the discrete time model of the resonator stage. In Equation 2.18, $y1[n]$ and $x1[n]$ are present state output and input of the first order integrator respectively, and $y1[n+1]$ is the next state output.

$$\begin{aligned} \frac{Y_1(z)}{X_1(z)} &= \frac{1}{z-1} \\ \Rightarrow zY_1(z) - Y_1(z) &= X_1(z) \\ \Rightarrow y1[n+1] &= y1[n] + x1[n] \end{aligned} \tag{2.18}$$

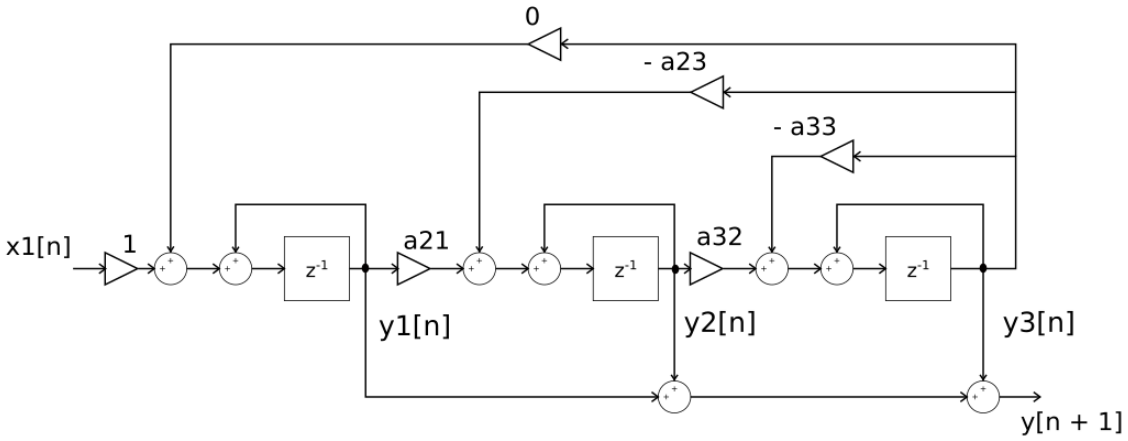


Figure 2.13: Loop filter implementation

Similarly in Equation 2.19, the discrete time equation for resonator is derived by deriving the individual transfer functions of each integrator in the structure, thus resulting in $y2[n+1]$ and $y3[n+1]$ next state outputs for the second and the third integrator respectively.

$$\begin{aligned} y2[n+1] &= y2[n] + x2[n] - a23y3[n] \\ y3[n+1] &= y3[n] + x3[n] - a33y3[n] \end{aligned} \tag{2.19}$$

Thus, the output of the loop filter $y[n+1]$ is given in Equation 2.20. The equation describes the mathematical expression of a third order CIFF loop filter in discrete time domain.

$$\begin{aligned} y[n+1] &= y1[n+1] + y2[n+1] + y3[n+1] \\ y[n+1] &= x1[n] + x2[n] + x3[n] + y1[n] + y2[n] + y3[n] - a23y3[n] - a33y3[n] \end{aligned} \tag{2.20}$$

After including feed-forward gain multiplying factors $a21$ and $a32$, the equation now becomes as shown in Equation 2.21. Also, since the values $x2[n]$ and $x3[n]$ are originating directly from $y1[n]$ and $y2[n]$ respectively, they can be substituted with $y1[n]$ and $y2[n]$.

$$\begin{aligned} y[n+1] &= 1 * x1[n] + a21 * y1[n] + a32 * y2[n] \\ &\quad + y1[n] + y2[n] + y3[n] - 0 * y3[n] - a23 * y3[n] - a33 * y3[n] \end{aligned}$$

$$\begin{aligned} \Rightarrow y[n+1] &= g1 + g2 + g3 \\ Where, g1 &= 1 * x1[n] + y1[n] + 0 * y3[n] \\ g2 &= a21 * y1[n] + y2[n] + (-a23) * y3[n] \\ g3 &= a32 * y2[n] + y3[n] + (-a33) * y3[n] \end{aligned} \tag{2.21}$$

A correspondence is now made between the discrete time formulation derived in Equation 2.21, and the Haskell model of $lfilter$, as shown in Figure 2.14. To model this equation, lists can be employed to enable the use of list specific functions that perform mathematical

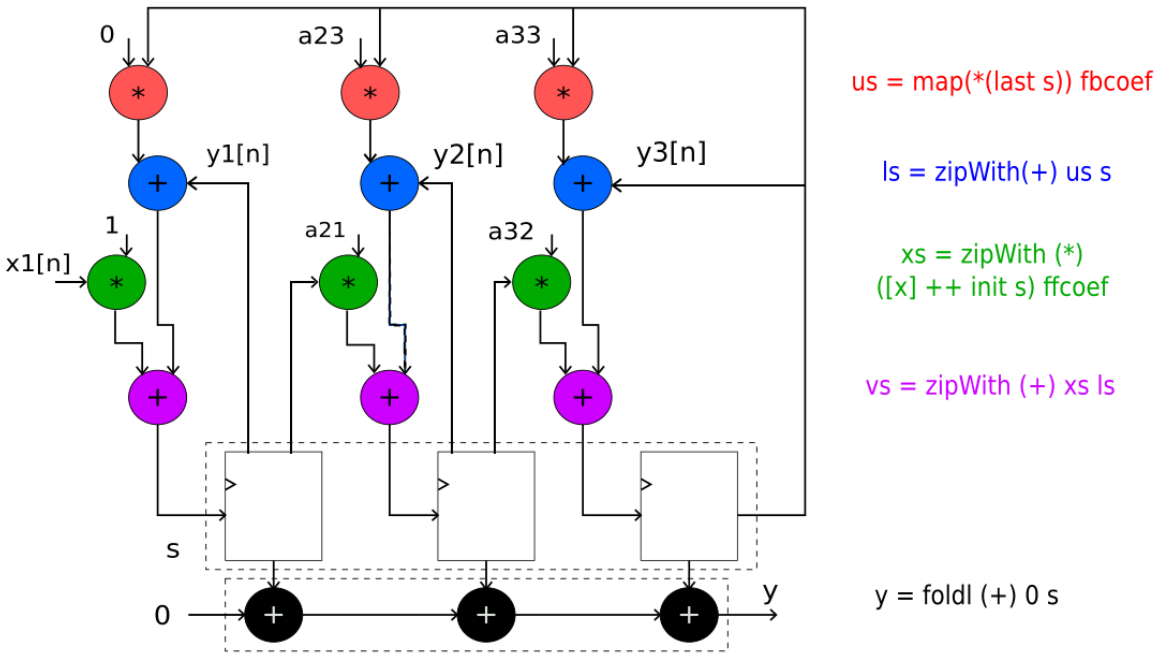


Figure 2.14: lfilter model in Haskell

operations on the elements of those lists. For example, the feed-forward gain factors are defined as *ffcoef*, which is a list containing values (1,a21,a32) and the feedback gain factors are defined by the list *fbcoef* containing the values (0,a23,a33). The present outputs can be represented by a list s of 3 elements that models the present states. The operations in the above defined groups $g1$, $g2$ and $g3$ are carried out by three different functions, which are present in the Haskell $Prelude$ library. To multiply the feed-forward gain factors, the $zipWith(*)$ function is used to multiply *ffcoef* with a list xs composed of $(x1[n], y1[n], y2[n])$ and the feedback gain factors *fbcoef* are multiplied with $y3[n]$ by using the $map(*(last\ \ s))$ function. Results of gain factor multiplications are all lists, and they are added with present states s to essentially form a list vs containing $(g1, g2, g3)$ of Equation 2.21. Finally, the output is obtained by adding the elements $(g1, g2, g3)$ by using the $foldl(+)$ function, that adds all the elements in a list.

```
1  lfilter :: (Num a,Floating a) => [a] -> (a,a) -> ([a],a)
2  lfilter s (x1,x2) = (s',y)
3    where
4      (a21,a23,a32,a33) = ((2.368164*(10**(-2))),(-1.617432*(10**(-3))),
5                           (1.208496*(10**(-2))),(-1.955032*(10**(-5))))
6
7      fbcoef    = [0,a23,a33]
8      ffcoef    = [1,a21,a32]
9
10     x         = x1 + x2
11     us        = map (*(last s)) fbcoef
```

```
12    ls       = zipWith (+) us s
13    xs       = zipWith (*) ([x] ++ init s) ffcoef
14    vs       = zipWith (+) xs ls
15
16    s'       = vs
17    y        = foldl (+) 0 s
```

Listing 2.6: lfilter function definition

Closed loop system

Listing 2.7 details the Haskell model for the Class - D amplifier controller, which is visualized in Figure 2.15. In this function, state s is a list of four values, with the first three values representing states of the loop filter, while the last value represents a state for the entire closed loop system to enable feedback modeling.

```
1  cdAmp :: (Floating a, Ord a,Enum a) => [a] -> a -> ([a],a)
2  cdAmp s x = (s',y)
3    where
4      s1      = init s
5      s2      = tail s
6
7      (s1',u) = lfilter s1 ((-1) * last s) x
8      y       = snd $ pwm (0,Up) u
9
10     s2'     = [y]
11
12     s'      = s1' ++ s2'
```

Listing 2.7: Closed loop model CDAmp

2.6 Conclusions

This chapter explained the introductory theory about a PWM amplifier and associated fundamentals essential to it. Further explanation was given about a 1 - bit $\Sigma\Delta$ modulator. The most important section of a $\Sigma\Delta$ modulation scheme is the loop filter, and the choice of a suitable topology to obtain required characteristics is shown. For this research, a CIFF structure was chosen due to its simplicity.

A brief introduction to functional programming is given by introducing Haskell and its primary features. It is later shown how a Class - D amplifier controller can be modeled in Haskell in the form of functions representing different modules of the system. Representing

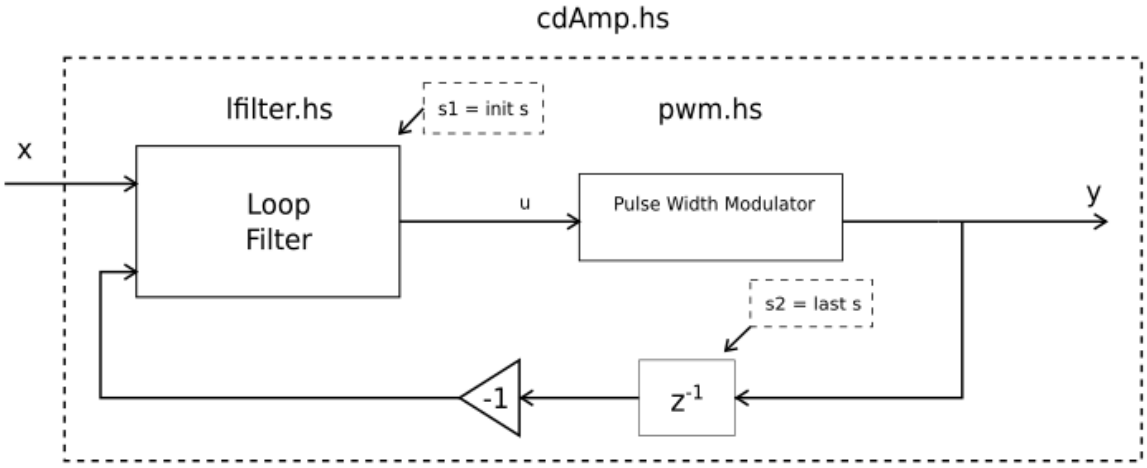


Figure 2.15: System model in Haskell

internal states of loop filter and system state from top level is an interesting step, and gives insight about how high level user defined functions can be used. Further improvements can be sought, however, the central focus of this exercise is to explore the possibility of modeling the system in a functional programming language.

Chapter 3

Implementation in CλaSH

3.1 Introduction

In Chapter 2, the concept of describing digital hardware in functional languages was presented with examples of two existing languages that have been used for a long time. The languages mentioned are called embedded domain specific languages, or EDSLs, which contain pre-defined special functions to simulate a hardware specification. The language CλaSH is a sub-set of Haskell, that borrows syntax and semantics from Haskell. This means that the compiler environment is also the same. Aside from type conversions and rewriting, a CλaSH specification is the same as the Haskell model, and the simulations can be performed by a native Haskell compiler [11]. In addition to that, CλaSH also supports polymorphism and higher order application of functions directly, since it is based on Haskell.

The retyping is done in order to properly convert a Haskell base model into a CλaSH implementation. There are two major type conversions to be done. This is necessary since some dynamic structures like lists and trees in Haskell are not directly realizable in hardware implementation [11]. To overcome this, lists in Haskell are converted to $Vectors$, which have a defined size and are recognized by CλaSH. Another retyping is done for representing integers and floating point values. Currently, CλaSH supports representation of fixed point numbers in both Signed and Unsigned forms. Conversion of floating point types to fixed point types can be made by retyping the function type with appropriate representation formats.

In the beginning of this research, a core reference model of Class - D amplifier was made in Haskell. The following sections describe retyping of the Haskell code to convert it into a CλaSH specification.

3.1.1 Data types and conversions

In Haskell model, a global type that represented values in all phases of operations was floating point. For most designs and even actual DSP processes, floating point arithmetic is preferred over fixed - point, due to its superior precision. However, implementing floating point on hardware is resource intensive, specially in areas where operations are performed with word lengths exceeding 24 bits wide. A trade-off is thus usually considered by selecting fixed - point representation for values.

There are issues working with fixed - point representation. First, there is the issue of integer overflow, which happens when the number of bits used in the format are not enough to represent extreme values. Secondly, care should be taken in selecting a fixed - point format when employing signed values. Fortunately, CλaSH compiler can be used to inspect maximum and minimum bounds of representation with full precision, after which one can select a suitable format.

In this CλaSH implementation, a signed fixed - point format Q6.18 was selected. The decision was made by observing the impulse response of the loop filter, maximum and minimum values of coefficients and the resolution of triangular wave carrier. The impulse response is presented in Figure 3.1, where the maximum amplitude of the response is around 31. Also, the triangular wave (carrier) is implemented to have peak values as -1 and +1. Thus, the format to accommodate the value 31 in signed fixed point needs to contain 6 bits integer bits, since the minimum and maximum bounds for 6 integer bits are (-32,31), whereas for 5 integer bits are (-16,15), and the limits were found by using the $minBound$ and $maxBound$ functions available in the $CLaSH.Fixed$ library. The number of fractional bits can now be anything from 0 to 26. To get a better precision, Q2.26 can be used to utilize a full 32 - bit length, however, keeping in mind standard audio format is 24 bits wide, Q6.18 was chosen instead.

Pulse Width Modulator

Listing 3.1 shows CλaSH implementation of pwm. Similar to Haskell model, the primary function is pwm, and step size is defined with value 0.015625 (requirement 1/64 per step per half edge traversal). The datatype $Slope$ is also retained to switch between Up and $Down$. As discussed in previous section, a type $Sample$ is defined as signed fixed point type in Q6.18 format. Additionally, -1 and +1, being floating point values, need to be defined as $Sample$ as well. For this conversion, $fLit$ function available in CλaSH Prelude library [12] is used, which is a function that converts a signed floating point number to a signed fixed point value. The conversions for both -1 and +1 are stored in two appropriate constants for lower limit and upper limit, denoted by $llim$ and $ulim$ respectively.

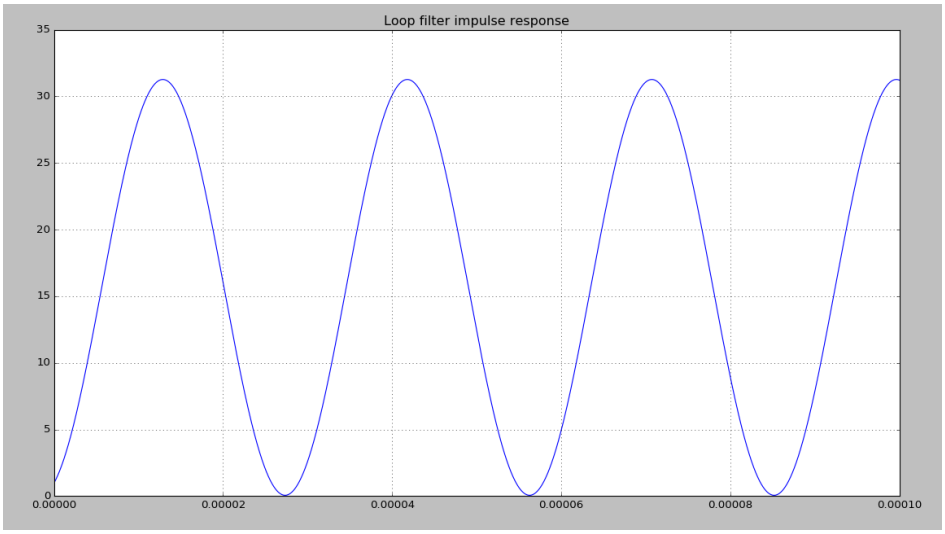


Figure 3.1: Loop filter impulse response

```
1  type Sample = SFixed 6 18
2
3  n    = $$(fLit 0.015625) :: Sample
4  llim = $$(fLit (-1.0)) :: Sample
5  ulim = $$(fLit 1.0)    :: Sample
6
7  data Slope = Up | Down
8
9  pwm :: (Sample,Slope)
10        -> Sample
11        -> ((Sample,Slope),Sample)
12 pwm (v,s) x = ((v',s'),y)
13   where
14     v' = case s of
15       Up  -> v + n
16       Down -> v - n
17
18     s' = case s of
19       Up   | v' < ulim    -> Up
20            | otherwise    -> Down
21       Down | v' > llim    -> Down
22            | otherwise    -> Up
23
24     y
25       | (x - v') >= 0 = ulim
26       | otherwise     = llim
```

Listing 3.1: CλaSH implementation of PWM

Loop Filter

Conversion from lists used in Haskell code to Vectors is perhaps best shown in Listing 3.2, which shows the CλaSH implementation of loop filter. For modeling lists as vectors, a new type $SampleVec3$ is declared, which is a vector of size 3, and each value of that vector is signed fixed - point Q6.18 $Sample$. Another vector operation to note is vector initialization. In CλaSH, it is done with a $:>$ operator, which appends a value left to it to the value on the right. In this implementation for example, vector *ffcoef* is initialized with values $a23$ and $a33$, In this way, list *ffcoef* of Haskell model is modeled in CλaSH, and similarly *fbcoef* as well.

CλaSH implementation of $lfilter$ differs from its Haskell counterpart in some ways. While most of the algorithm remains same, multiplication (in function $fpmult$) is realized in CλaSH implementation by using the function '$times$' after which the result is resized to the required datatype using '$resizeF$', which is a resizing function for fixed point numbers. Both '$times$' and '$resizeF$' functions are available in 'Prelude.Fixed' library of CλaSH.

```
1  type Sample     = SFixed 6 18
2  type SampleVect3 = Vec 3 Sample
3
4  a33 = -1.955032e-5 :: Sample
5  a23 = -1.617432e-3 :: Sample
6  a21 = 2.368164e-2 :: Sample
7  a32 = 1.208496e-2 :: Sample
8
9  fbcoef = 0 :> a23 :> a33 :> Nil
10 ffcoef = 1 :> a21 :> a32 :> Nil
11
12 fpmult :: Sample -> Sample -> Sample
13 fpmult a b = c
14   where
15     c = resizeF (a `times` b) :: Sample
16
17 lfilter :: SampleVect3 -> (Sample,Sample) -> (SampleVect3, Sample)
18 lfilter s (x1,x2) = (s', y)
19   where
20     x = resizeF (x1 `plus` x2) :: Sample
21     us = map (fpmult (last s)) fbcoef
22     ls = zipWith (`plus`) us s
23     xs = zipWith (fpmult) ([x] ++ init s) ffcoef
24     vs = zipWith (`plus`) xs ls
25
26     s' = vs
27     y  = foldl (`plus`]) 0 s
```

Listing 3.2: CλaSH implementation of Loop filter

3.1.2 Closed loop system

Listing 3.3 shows the CλaSH implementation of Class - D amplifier. The type $CDSample$ is also Q6.18 since the type is propagated to other modules. Afterwards, a function $bundle$ is used, which takes two values of type $Signal$ and merges them into a tuple of the type $Signal$ that is synchronous with system clock. Top entity annotations are defined for creating an RTL code that acts as a wrapper for actual $topEntity$ declaration.

```
{-# ANN topEntity
(defTop
{ t_name    = "cdamparchM"
, t_inputs  = ["I_in"]
, t_outputs = ["O_out"]
}) #-}

type CDSample    = SFixed 6 18
type CDSampleVect4 = Vec 4 CDSample

cdAmp :: CDSampleVect4 -> CDSample -> (CDSampleVect4,CDSample)
cdAmp s x = (s',y)
  where
    s1 = init s
    s2 = last s

    (s1',u) = lfilter s1 (x,(last s))
    y       = snd $ pwm (0,Up) u

    s2'     = y :> Nil
    s'      = s1' ++ s2'

cdamparchM = mealy cdAmp (repeat 0)

topEntity :: Signal CDSample -> Signal CDSample
topEntity = cdamparchM
```

Listing 3.3: CλaSH closed loop model CDAmp

In Haskell models, a notion of state was modeled in the pwm and $lfilter$ functions. However, in CλaSH, sequential designs which contain a notion of state need to be declared with an initial state. This is done by using the $mealy$ function for the $cdAmp$ function. In CλaSH, the function type of $mealy$ is described as shown in Listing 3.4 [12]. $mealy$ takes a function having the signature of the type $s \rightarrow i \rightarrow (s', o)$, where s is present state, i is input and the tuple (s', o) is the next state and output. A second argument is input i of the type $Signal$, which is used in functions that are translated to top level entities. The output is denoted by $Signal\ o$. Since $cdAmp$ has a type definition similar to $s \rightarrow i \rightarrow (s', o)$, the function $cdAmp$

can be converted to mealy state machine function, given by $cdamparchM$.

```
mealy :: (s -> i -> (s, o))
        -> Signal i
        -> Signal o
```

Listing 3.4: CλaSH mealy function type description

3.2 Conclusions

In this chapter, modeling and implementation of a basic Class - D amplifier has been carried out. In Haskell, modeling was performed in a modular fashion to accentuate the structural design aspect of design in a functional semantic environment, and to also to make the model easier to read. Implementation in CλaSH was done by transforming the Haskell code with modifications, without sacrificing the original approach.

In the next chapter, results of Haskell and CλaSH models are shown and compared against each other. As an additional exercise, both models are also compared against a Simulink model. The reason behind this exercise is to show how in a basic sense Haskell itself can be used to model digital systems in the first place, and then how CλaSH implementation agrees to both models.

Chapter 4

Results and Comparisons

In the previous chapter, modeling and implementation of a Class - D amplifier controller in Haskell and CλaSH were discussed respectively. In this chapter, simulation results of both models are presented and compared.

For both Haskell and CλaSH simulations, a simulation flow was built, as shown in Figure 4.1. In case of Haskell simulations, primary inputs were given to the top level function, and results were written to a text file by using IO functionality. The text file was transformed to a comma - separated values (.csv) file from which values were extracted by a MATLAB script (see **Appendix B**) that calculates Power Spectrum Density (PSD) and performance figures [13]. CλaSH simulation differed from Haskell in that the input was defined in top level CλaSH code, using $simulate$ function provided in the CλaSH library.

After performing the required simulations on Haskell and CλaSH models, comparisons are made on an existing Simulink model of the system. This model was built to represent the intended system as a basic reference. It is speculative at this point as to whether the Haskell model itself could represent as reference, however, since the CλaSH implementation was done by translating Haskell code itself, the Simulink model was also brought into this

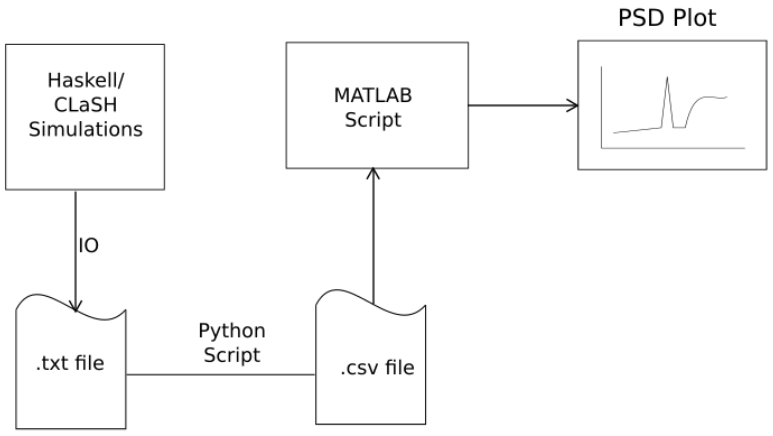


Figure 4.1: Simulation Environment flow

perspective to illustrate how the Haskell model agrees to its implementation in the first place. The Simulink top level is shown in Figure 4.2. Here, the subsystem blocks are connected as per the Class - D amplifier controller architecture.

The simulation results of the Simulink model, the Haskell model and the CλaSH implementation include two points of comparison. The first subject of comparison is the intended behavior of the system. It has been discussed in Chapter 2 that the Class - D amplifier controller system should exhibit a noise shaping behavior. This means that the noise, which is the quantization noise occuring around the frequency of the carrier signal and its integer multiples, should be moved outside the band of interest, which is from 0 Hz to 20 KHz, and the behavior is viewed in the form of power spectral density (PSD) plots. In the obtained PSD plots, the points of interest are the central frequency gain occuring at the frequency of the applied input signal, and the quantization noise out of the band exhibiting a high pass behavior. The second subject of comparison is done from the obtained signal to noise ratio (SNR) values from the system's response. However, most of the interest is on evaluating the noise shaping behavior of the system.

4.1 Haskell Simulation

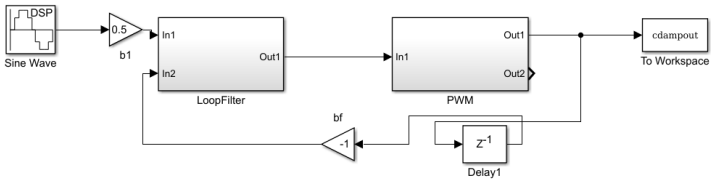
Simulation is performed by passing a list containing sampled values of a sine function. The sine wave has a frequency of 6 kHz, with sampling frequency 49.152 MHz. This frequency was chosen keeping in view that this frequency was required to generate a triangular carrier of 192 kHz and that which contains 256 steps in one cycle, since the PWM sampling for one cycle contains 256*192 K = 49152 K samples. Also, the amplitude of the input was kept 0.5.

Figure 4.3 shows the pulse spectrum density (PSD) plots of Simulink model and Haskell model output. It clearly shows the expected noise shaping that happens out of the audio band. The first carrier frequency component occurs at 192 kHz and its copies are at integral multiples of its base frequency.

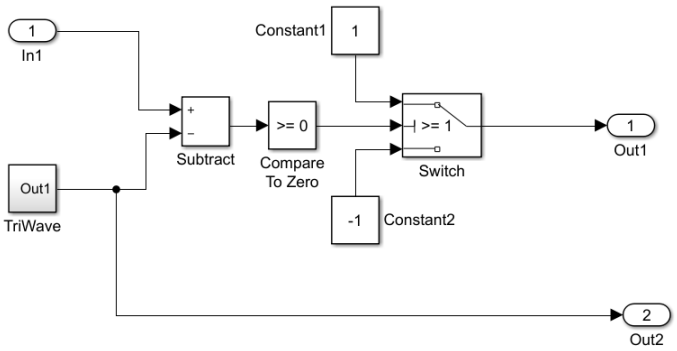
4.2 CλaSH Simulation

The CλaSH simulation follows the Haskell simulation in terms of parameters, whereas the method is different. As shown in Listing 4.1, the keyword $simulate$ is used which evaluates the function cdAmp with the input list $inpdata$. Just like with other values in the code, the input should also follow the same format, which is signed fixed - point Q6.18.

Figure 4.4 shows the PSD plots of Simulink and CλaSH simulation results. Here too, it can be seen that the system performs noise shaping as expected, with PWM carrier's fundamental and higher frequency components outside the audio band. There is also a high gain around the input frequency.

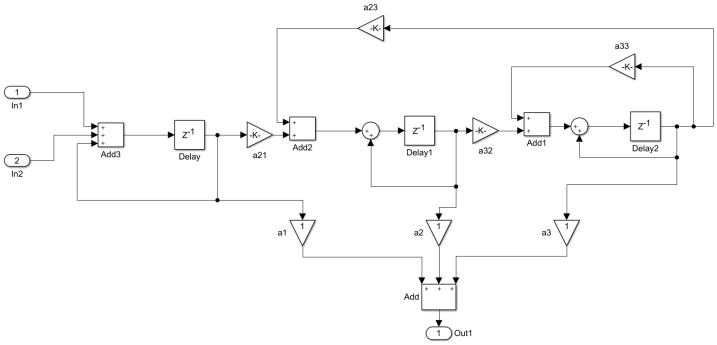


(a) Top level Simulink model



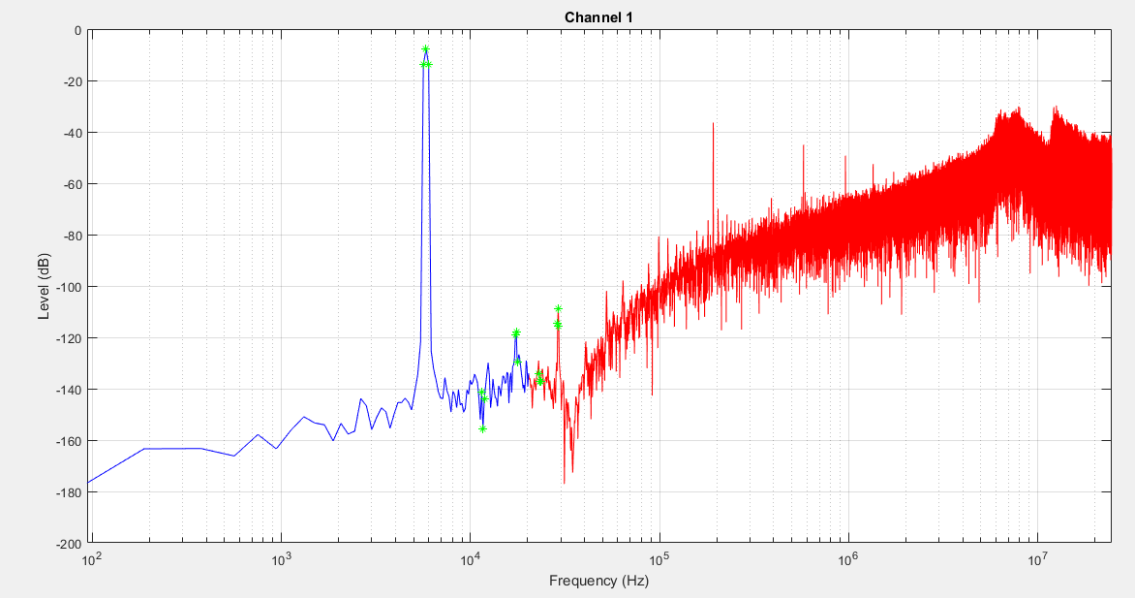
(b) Pulse Width Modulator

(c) Triangular wave generator (carrier)

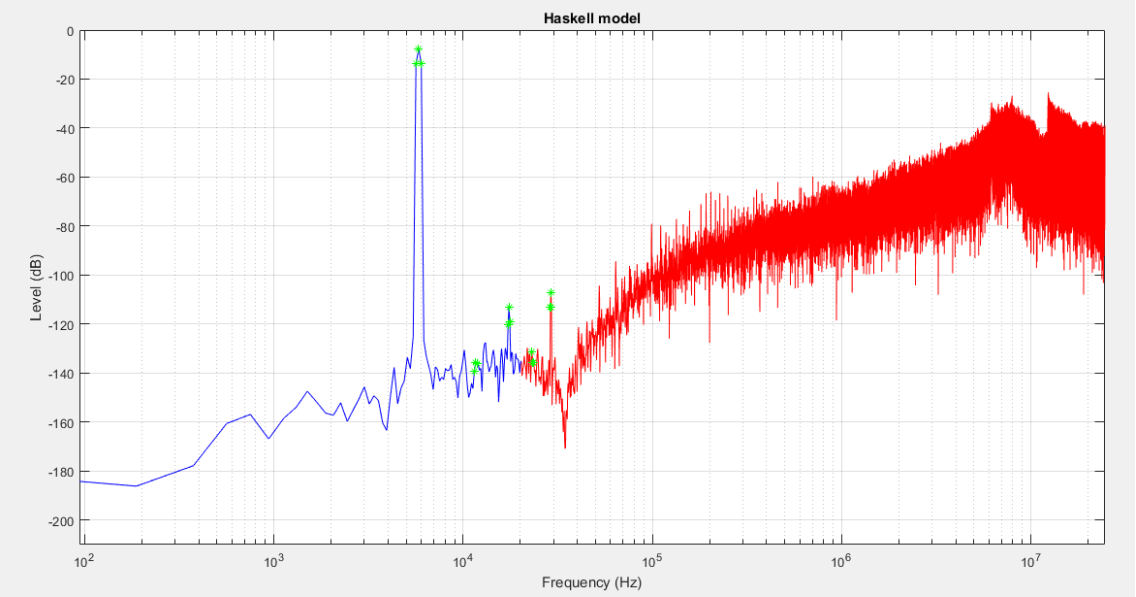


(d) Loop filter

Figure 4.2: Simulink model top view and subsystem

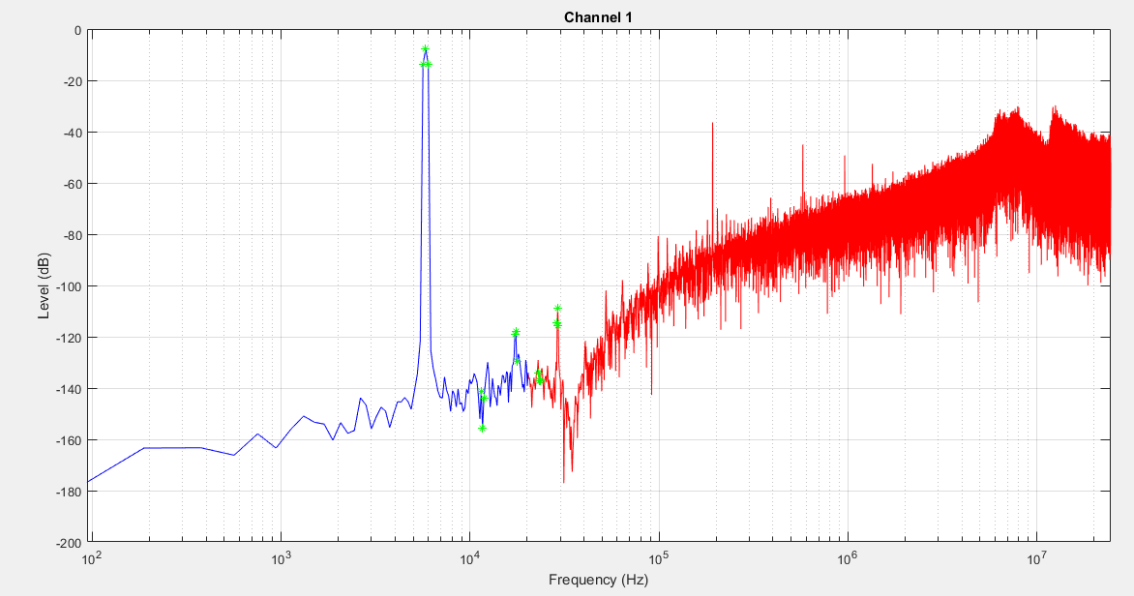


(a) Simulink Simulation

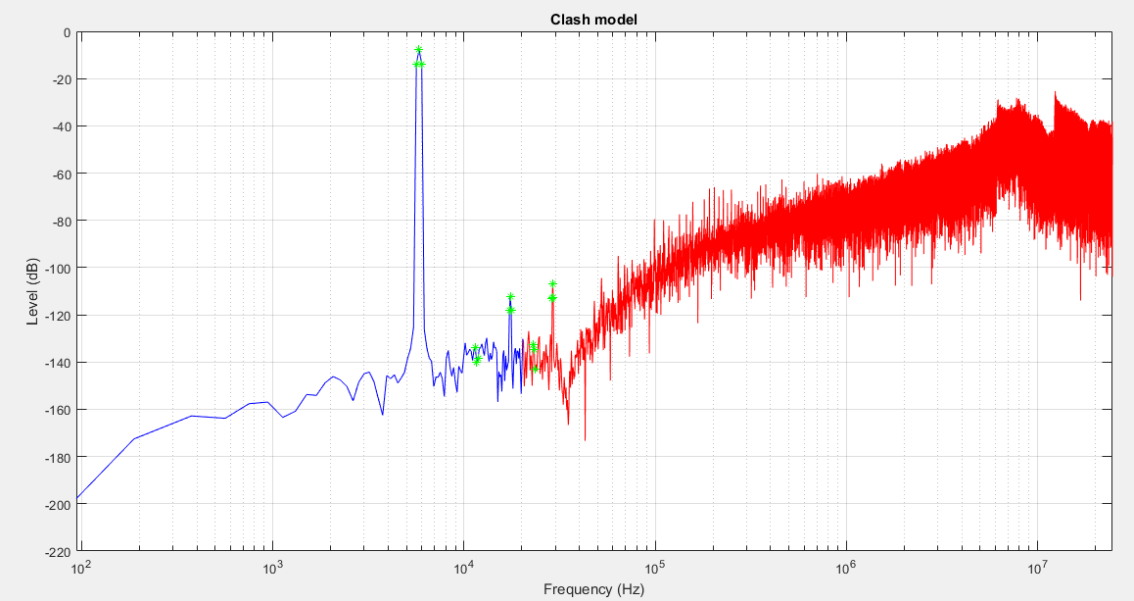


(b) Haskell Simulation

Figure 4.3: Simulation results of Simulink (a) and Haskell (b) models



(a) Simulink Simulation



(b) CλaSH Simulation

Figure 4.4: Simulation results of Simulink (a) and CλaSH (b) models

```
f6 = 5812.5
w6 t = 0.5*(sin(2*pi*f6*t))
st = [0,(1/(1024*48000))..1]

inpdata = L.map fLitR (L.map w6 st) :: [CDSample]

res = simulate cdAmp inpdata
```

Listing 4.1: CλaSH simulation of Class - D amplifier

The MATLAB script also calculated signal parameters like Signal - to - Noise Ratio (SNR) and Total Harmonic Distortion plus Noise (THDN). Table 4.1 gives an overview of the calculated values for both Haskell and CλaSH simulations.

Simulation	**SNR (dB)**	**THDN (dB)**
Simulink	109.06	-106.14
Haskell	109.75	-104.13
CλaSH	110.88	-103.19

Table 4.1: Simulation comparisons of Simulink, Haskell and CλaSH

From the values obtained, it is clearly seen that the Signal to noise ratio is well above 100 dB for all three models, which is usually a characteristic of a 1 - bit Class - D modulator. The difference of about 1 dB between the Haskell model and the CλaSH implementation can be because of the differences between floating point and fixed point formats.

4.3 Synthesis

Any RTL code that needs to be realized as physical hardware, should be synthesizable. In the beginning of this research CλaSH was introduced along with a feature that enables it to generate VHDL code that is synthesizable. When developing a design in bare VHDL, one must follow certain coding guidelines in order to make the design synthesizable. CλaSH generated code however is readily synthesizable on to hardware.

The generated VHDL top entity code is an entity with port descriptions mentioned in the top level ANN annotations. In addition, CλaSH also generates other RTL files on which the top level design description depends. To see how the design is synthesized, the generated top entity $cdAmp.vhdl$ is synthesized by Quartus with target device set to Cyclone II EP2C20F484C7. The synthesis results are shown in Figure 4.5.

The RTL view of $cdAmp$ is also presented in Figure 4.6. In this RTL view, the block *cdamp_lfilter* is the loop filter and the block *cdamp_pwm* is the PWM block. The system's input is denoted as $x[23..0]$ which represents the 24 bit Q6.18 value. Similarly the output is

Top-level Entity Name	cdAmp
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	577 / 18,752 (3 %)
Total combinational functions	560 / 18,752 (3 %)
Dedicated logic registers	88 / 18,752 (< 1 %)
Total registers	88
Total pins	50 / 315 (16 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	15 / 52 (29 %)

Figure 4.5: Synthesis result of cdAmp.vhdl

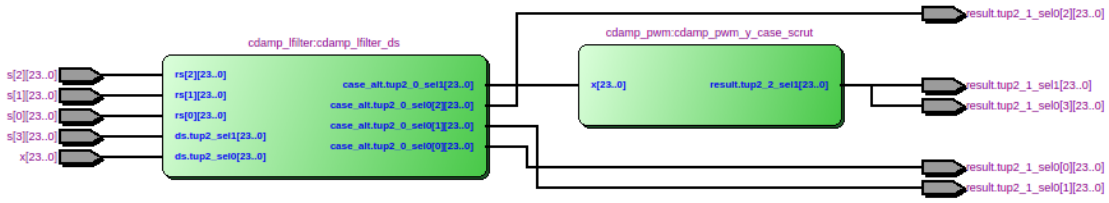


Figure 4.6: RTL view of cdAmp.vhdl

denoted as *result.tup2_1_sel1[23..0]* which at top level is connected to $y[23..0]$ and it follows the same datatype as the input.

4.4 Conclusions

In this chapter, simulations were first performed on a Haskell model and a CλaSH implementation of a Class - D amplifier controller. Since the CλaSH implementation was derived from Haskell, the Haskell model's correctness was first established by comparing its simulation results with a Simulink model. The simulation verification was performed on two fronts, obtaining the PSD response and a quantitative comparison of obtained performance figures. Haskell model's results closely resembled that of Simulink variant.

Simulation results obtained from the CλaSH model reflected the resemblance with it's Haskell counterpart. The results compared in a similar way with Simulink results confirmed the similarities with the CλaSH implementation and the Haskell model. Furthermore, the performance figures recorded by both CλaSH and Haskell were comparable and very less difference could be seen between them.

Synthesis was performed on the top level entity VHDL description that was generated by the CλaSH compiler. The results show that the design is indeed synthesizable, with the design occupying just about 3% of the logic elements available in the FPGA. The RTL view generated by Quartus indicated that the synthesized model has a structure resembling that of the theory and the Simulink implementation.

Chapter 5

Conclusions

In this thesis, a method to design hardware that uses functional semantics was investigated. The method uses an FHDL called CλaSH, which is a functional programming language used to describe and implement digital hardware, and was sued to design a Class - D amplifier controller system.

The research begins with introducing a Class - D amplifier system and how the system performance can be improved by the means of digital control. The system was analyzed by formulating the system in mathematical form. A notable observation from this formulation is the derivation of the structure of a disctrete time integrator, which forms an important part of the control system. Further analysis showed that the integrator, when extended to formulate a resonator structure, introduces more stability in the system. Combining the integrator structure with the resonator structure resulted in a well known loop filter structure from $\Sigma\Delta$ theory, called the Cascaded Integrators with Feed Forward summation (CIFF). The basic system design involving a loop filter and a pulse width modulator resembles a 1 - bit $\Sigma\Delta$ modulator.

The next step taken was to model the system in Haskell. This step is crucial and beneficial to this research, as the Haskell model forms a basis for a CλaSH implementation, and also to prove the correspondence between implementation and modeling. The Haskell model was then simulated with a sampled input of a fixed frequency to establish the initial performance of the system. Performance figures like signal power, SNR and noise power were recorded.

Implementation of the system in CλaSH required an established Haskell model, which meant verifying the Haskell model with a reference. The simulations performed in Haskell matched with a Simulink reference model with respect to the noise shaping behavior and performance figures. The next step then involved translating Haskell definitions to CλaSH. This step majorly involved changing the floating point datatypes used in Haskell model to a signed fixed point representation, keeping in mind that the system needs to be hardware efficient. This meant sacrificing performance of the system, but since the system that

is described and implemented in this research is a basic 1 - bit Class - D amplifier controller, the focus rested solely on obtaining a comparable performance and an expected noise shaping behavior. Other modifications included transformation of lists to vectors and introducing mealy definitions to implement subsystem modules as sequential designs. The final CλaSH model resembled the base Haskell model and simulation comparison with the Simulink model again proved its correctness. Performance figures obtained from simulations closely matched with those of Haskell model, further strengthening the claim that the CλaSH implementation respects the system behavior.

The RTL code generated from CλaSH description was synthesized to evaluate resource usage. It was found that the design used only upto 3% of resources on the chosen FPGA Cyclone II EP2C20F484C7. The RTL views generated in Quartus also respect the overall system structure. This strengthens the validity of the CλaSH implementation.

In conclusion, CλaSH is a viable option to design systems involving audio signal processing applications. The design steps were direct and the obtained performance figures were comparable to what is expected of a Class - D amplifier controller system. Further improvements can be made on the system design with higher order function applications, for example implementing a higher order loop filter by using the $lfilter$ implementation, which will improve the system's performance and greatly reduce development time.

Chapter 6

Future Work

The previous chapter summarizes simulation results of Class - D amplifier controller models described in Haskell and CλaSH and how they compare against each other. When observing the general noise shaped PSD and even performance figures of both implementations, compared against their Simulink counterpart, it can be immediately deduced that describing the Class - D system in CλaSH is feasible. However, when designing systems that focus on DSP applications, there is a lot of room for improvement. A first important area to improve the implementation is performance. Class - D designs aim to get as low a noise floor as possible and as high SNR as possible. A lot of factors come into play that determine the resultant performance of this system; factors like choice of fixed point representation, type of loop filter, coefficient values and such.

The implemented system is a basic Class - D system for a single stream of output. In audio applications, for example, there is a requirement to serve at least two channels at the system's output for two respective input streams. The CλaSH implementation of the function $cdAmp$ could then be used as a higher order function to implement a system that involves two instances of $cdAmp$. This higher order application saves a lot of design time when implementing a Class - D controller system that is required drive 4 channels or more.

In a similar approach, increasing the order of the loop filter using higher order application can also be investigated. Typically, the state of the art Class - D controller systems [Source Aixgn] use loop filters of upto 7th order per channel. Also, each channel loop filter is serially connected with loop filters of further stages for a much better noise reduction. Furthermore, it can also be highly desirable to have the system extensively configurable. Configurability can be implemented for programming gains and coefficients used in the loop filters, and the programming could be from system level. This level of programmable system design enables great flexibility in tuning the performance of the system from system level.

Describing digital hardware has come a long way with regards to working with CλaSH. Cuurently, CλaSH is a highly stable FHDL to develop sequential designs in Haskell, a highly advantageous fact for developers familiar with Haskell. Designs based on pure digital oper-

ations can be easily described and implemented in this environment, however for designs targeting both digital and analog domains (mixed signal), the language does not yet support analog modeling. The idea comes from looking at HDLs like Verilog - AMS which can support analog primitives for behaviorally modeling continuous time systems [14]. Introducing analog primitives in CλaSH can be an ambitious task, but can be very beneficial in cases where verification that involves interaction of the system with the analog world needs to be locally performed in the CλaSH tooling environment.

Bibliography

[1] J. M. Goldberg and M. B. Sandler, “Noise shaping and pulse-width modulation for an all-digital audio power amplifier.” *Audio Eng. Soc.*, vol. 39, no. 6, pp. 449–460, 1991.

[2] “Learn you a haskell (PDF).” [Online]. Available: http://learnyouahaskell.com

[3] D. G. Holmes and T. A. Lipo, “Pulse width modulation for power converters.” *Wiley-IEEE*, p. 96, 2003.

[4] M. Hawksford, “A tutorial guide to noise shaping and oversampling in adc and dac systems.” in *Proc. Institute of Acoustics*, vol. 39, no. 6, 1989, pp. 289–302.

[5] S. Park, *Principles of Sigma - Delta Modulation for Analog to Digital Converters*. Motorola.

[6] L. Risbo, “Discrete-time modeling of continuous-time pulse width modulator loops.” in *Audio Engineering Society Conference: 27th International Conference: Class D Audio Amplification.*, 2005.

[7] T. Mouton and B. Putzeys, “Digital control of a pwm switching amplifier with global feedback.” in *Audio Engineering Society Conference: 37th International Conference: Class D Audio Amplification.*, 2009.

[8] R. Schreier and G. C. Temes, “Understanding delta-sigma data converters.” *IEEE press.*, vol. 101, pp. 115–123, 2005.

[9] D. Schinkel, “High accuracy and high resolution sigma delta converters, MSc.Thesis,University of Twente,” 2004.

[10] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards, “Cλash: Structural descriptions of synchronous hardware using haskell.” in *Proceedings of the 13th Conference on Digital System Design (DSD), Lille, France.*, 2010, pp. 714–721.

[11] J. Kuper, C. P. R. Baaij, M. Kooijman, and M. E. T. Gerards, “Exercises in architecture specification using cλash.” in *Proceedings of Forum on Specification and Design Languages (FDL), 2010, Southampton, England.*, 2010, pp. 178–183.

[12] “CLaSH.Prelude : CAES Language for Synchronous Hardware - Prelude Library.” [Online]. Available: http://hackage.haskell.org/package/clash-prelude-0.11.2/docs/CLaSH-Prelude.html

[13] T. v. Doesum, “MATLAB Spectrum analysis toolbox, Axign B.V., Enschede.”

[14] *Cadence - Verilog AMS Language Reference*. Cadence, 2005.

Appendix A

A.1 VHDL Code for cdAmp entity

```
entity cdAmp is
port(I_in      : in signed(23 downto 0);
-- clock
system1000     : in std_logic;
-- asynchronous reset: active low
system1000_rstn : in std_logic;
O_out          : out signed(23 downto 0));
end;
```

A.2 VHDL Code for cdAmp top level

```
-- Automatically generated VHDL-93
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.ALL;
use std.textio.all;
use work.all;
use work.cdamp_types.all;

entity cdamp_cdamp is
port(s      : in cdamp_types.array_of_signed_24(0 to 3);
x      : in signed(23 downto 0);
result : out cdamp_types.tup2_1);
end;

architecture structural of cdamp_cdamp is
signal app_arg     : cdamp_types.array_of_signed_24(0 to 3);
signal y           : signed(23 downto 0);
```

```
19 signal case_alt   : cdamp_types.array_of_signed_24(0 to 3);
20 signal y_case_scrut : cdamp_types.tup2_2;
21 signal ds         : cdamp_types.tup2_0;
22 signal app_arg_0  : cdamp_types.array_of_signed_24(0 to 0);
23 signal s1         : cdamp_types.array_of_signed_24(0 to 2);
24 signal y_app_arg  : signed(23 downto 0);
25 signal y1         : signed(23 downto 0);
26 signal ds_app_arg : cdamp_types.array_of_signed_24(0 to 2);
27 signal ds_app_arg_0 : cdamp_types.tup2;
28 signal ds_app_arg_1 : signed(23 downto 0);
29 signal u          : signed(23 downto 0);
30 begin
31 result <= (tup2_1_sel0 => app_arg
32 ,tup2_1_sel1 => y);
33
34 app_arg <= case_alt;
35
36 y <= y1;
37
38 case_alt <= cdamp_types.array_of_signed_24'(cdamp_types.array_of_signed_24'(s1) &
       cdamp_types.array_of_signed_24'(app_arg_0));
39
40 cdamp_pwm_y_case_scrut : entity cdamp_pwm
41 port map
42 (result => y_case_scrut
43 ,x      => y_app_arg);
44
45 cdamp_lfilter_ds : entity cdamp_lfilter
46 port map
47 (case_alt => ds
48 ,rs       => ds_app_arg
49 ,ds       => ds_app_arg_0);
50
51 app_arg_0 <= cdamp_types.array_of_signed_24'(0 => y);
52
53 s1 <= ds.tup2_0_sel0;
54
55 y_app_arg <= u;
56
57 y1 <= y_case_scrut.tup2_2_sel1;
58
59 -- init begin
60 ds_app_arg <= s(0 to s'high - 1);
61 -- init end
62
63 ds_app_arg_0 <= (tup2_sel0 => x
64 ,tup2_sel1 => ds_app_arg_1);
```

```
65
66 -- last begin
67 ds_app_arg_1 <= s(s'high);
68 -- last end
69
70 u <= ds.tup2_0_sel1;
71 end;
```

A.3 VHDL Code for lfilter

```
1  -- Automatically generated VHDL-93
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5  use IEEE.MATH_REAL.ALL;
6  use std.textio.all;
7  use work.all;
8  use work.cdamp_types.all;
9
10 entity cdamp_lfilter is
11 port(rs      : in cdamp_types.array_of_signed_24(0 to 2);
12 ds      : in cdamp_types.tup2;
13 case_alt : out cdamp_types.tup2_0);
14 end;
15
16 architecture structural of cdamp_lfilter is
17 signal x2         : signed(23 downto 0);
18 signal x1         : signed(23 downto 0);
19 signal result     : signed(23 downto 0);
20 signal ws1        : cdamp_types.array_of_signed_24(0 to 2);
21 signal ws         : cdamp_types.array_of_signed_24(0 to 3);
22 signal ws1_app_arg : cdamp_types.array_of_signed_24(0 to 2);
23 signal app_arg    : cdamp_types.array_of_signed_24(0 to 2);
24 signal app_arg_0  : cdamp_types.array_of_signed_24(0 to 2);
25 signal app_arg_1  : cdamp_types.array_of_signed_24(0 to 2);
26 signal app_arg_2  : cdamp_types.array_of_signed_24(0 to 1);
27 signal app_arg_3  : signed(23 downto 0);
28 signal app_arg_4  : cdamp_types.array_of_signed_24(0 to 2);
29 signal app_arg_5  : cdamp_types.array_of_signed_24(0 to 1);
30 signal app_arg_6  : cdamp_types.array_of_signed_24(0 to 1);
31 signal ds1        : signed(24 downto 0);
32 signal result_0   : signed(23 downto 0);
33 signal shiftedl   : signed(24 downto 0);
34 signal case_alt_0 : signed(23 downto 0);
35 signal case_alt_1 : signed(23 downto 0);
```

```
signal case_scrut : boolean;
signal case_alt_2 : signed(23 downto 0);
signal case_scrut_0 : boolean;
signal app_arg_7  : signed(24 downto 0);
begin
x2 <= ds.tup2_sel1;

x1 <= ds.tup2_sel0;

case_alt <= (tup2_0_sel0 => app_arg
,tup2_0_sel1 => result);

-- last begin
result <= ws(ws'high);
-- last end

-- zipWith begin
zipwith : for i in ws1'range generate
begin
cdamp_lfilter_specf_0 : entity cdamp_lfilter_specf
port map
(result => ws1(i)
,x => rs(i)
,y => ws1_app_arg(i));
end generate;
-- zipWith end

ws <=
    cdamp_types.array_of_signed_24'(signed'(shift_left(to_signed(0,24),to_integer(to_signed(18,64))))
    & ws1);

-- init begin
ws1_app_arg <= ws(0 to ws'high - 1);
-- init end

-- zipWith begin
zipwith_1 : for i_0 in app_arg'range generate
begin
cdamp_satplus_1 : entity cdamp_satplus
port map
(result => app_arg(i_0)
,a => app_arg_4(i_0)
,b => app_arg_0(i_0));
end generate;
-- zipWith end

-- zipWith begin
```

```
81  zipwith_3 : for i_1 in app_arg_0'range generate
82  begin
83  cdamp_satplus_2 : entity cdamp_satplus
84  port map
85  (result => app_arg_0(i_1)
86  ,a => app_arg_1(i_1)
87  ,b => rs(i_1));
88  end generate;
89  -- zipWith end
90
91  app_arg_1 <=
        cdamp_types.array_of_signed_24'(signed'(shift_left(to_signed(0,24),to_integer(to_signed(18,6
        & app_arg_2);
92
93  -- map begin
94  map_r : block
95  signal vec_1 : cdamp_types.array_of_signed_24(0 to 1);
96  begin
97  vec_1 <= cdamp_types.array_of_signed_24'(-to_signed(424,24),-to_signed(5,24));
98  map_r_0 : for i_2 in app_arg_2'range generate
99  begin
100 cdamp_fpmult_3 : entity cdamp_fpmult
101 port map
102 (result => app_arg_2(i_2)
103 ,a => app_arg_3
104 ,b => vec_1(i_2));
105 end generate;
106 end block;
107 -- map end
108
109 -- last begin
110 app_arg_3 <= rs(rs'high);
111 -- last end
112
113 app_arg_4 <= cdamp_types.array_of_signed_24'(signed'(result_0) & app_arg_5);
114
115 -- zipWith begin
116 zipwith_5 : block
117 signal vec1_2 : cdamp_types.array_of_signed_24(0 to 1);
118 signal vec2_2 : cdamp_types.array_of_signed_24(0 to 1);
119 begin
120 vec1_2 <= app_arg_6;
121 vec2_2 <= cdamp_types.array_of_signed_24'(to_signed(6207,24),to_signed(3167,24));
122 zipwith_6 : for i_3 in app_arg_5'range generate
123 begin
124 cdamp_fpmult_4 : entity cdamp_fpmult
125 port map
```

```
126 (result => app_arg_5(i_3)
127 ,a => vec1_2(i_3)
128 ,b => vec2_2(i_3));
129 end generate;
130 end block;
131 -- zipWith end
132
133 -- init begin
134 app_arg_6 <= rs(0 to rs'high - 1);
135 -- init end
136
137 cdamp_fextendingnumfixedfixed_cminus1_ds1 : entity
        cdamp_fextendingnumfixedfixed_cminus1
138 port map
139 (result => ds1
140 ,ds     => x1
141 ,ds1    => x2);
142
143 result_0 <= case_alt_0 when case_scrut else
144 case_alt_1;
145
146 shiftedl <= shift_left(ds1,to_integer(to_signed(0,64)));
147
148 with (app_arg_7) select
149 case_alt_0 <= case_alt_2 when "00000000000000000000000000",
150 signed'(0 => '0', 1 to 24-1 => '1') when others;
151
152 case_alt_1 <= case_alt_2 when case_scrut_0 else
153 signed'(0 => '1', 1 to 24-1 => '0');
154
155 case_scrut <= ds1 >= to_signed(0,25);
156
157 case_alt_2 <= resize(shiftedl,24);
158
159 case_scrut_0 <= app_arg_7 = (not (resize((signed'(0 => '0', 1 to 24-1 =>
        '1')),25)));
160
161 app_arg_7 <= shiftedl and (not (resize((signed'(0 => '0', 1 to 24-1 => '1')),25)));
162 end;
```

A.4 VHDL Code for pwm

```
1 -- Automatically generated VHDL-93
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
```

```
4  use IEEE.NUMERIC_STD.ALL;
5  use IEEE.MATH_REAL.ALL;
6  use std.textio.all;
7  use work.all;
8  use work.cdamp_types.all;
9
10 entity cdamp_pwm is
11 port(x      : in signed(23 downto 0);
12 result : out cdamp_types.tup2_2);
13 end;
14
15 architecture structural of cdamp_pwm is
16 signal app_arg                                : cdamp_types.tup2_4;
17 signal app_arg_0                              : signed(23 downto 0);
18 signal app_arg_1                              : unsigned(0 downto 0);
19 signal case_scrut                             : boolean;
20 signal v                                      : signed(23 downto 0);
21 signal case_scrut_0                           : boolean;
22 signal app_arg_2                              : signed(23 downto 0);
23 signal v_case_alt                             : signed(23 downto 0);
24 signal v_case_scrut                           : std_logic_vector(0 downto 0);
25 signal v_case_alt_0                           : signed(23 downto 0);
26 signal v_case_alt_1                           : signed(23 downto 0);
27 signal v_app_arg                              : std_logic_vector(0 downto 0);
28 signal r                                      : std_logic_vector(23 downto 0);
29 signal v_case_scrut_app_arg                   : cdamp_types.tup2_3;
30 signal v_case_scrut_app_arg_0                 : std_logic_vector(0 downto 0);
31 signal v_case_alt_0_case_scrut_app_arg        : std_logic_vector(0 downto 0);
32 signal v_case_alt_0_case_scrut_app_arg_app_arg : std_logic_vector(0 downto 0);
33 signal v_case_alt_0_case_scrut_app_arg_app_arg_0 : std_logic_vector(0 downto 0);
34 signal r_case_scrut_app_arg                   : cdamp_types.tup2_3;
35 begin
36 result <= (tup2_2_sel0 => app_arg
37 ,tup2_2_sel1 => app_arg_0);
38
39 app_arg <= (tup2_4_sel0 => v
40 ,tup2_4_sel1 => app_arg_1);
41
42 app_arg_0 <= to_signed(262144,24) when case_scrut else
43 to_signed(-262144,24);
44
45 app_arg_1 <= to_unsigned(0
46 ,1) when case_scrut_0 else
47 to_unsigned(1,1);
48
49 case_scrut <= app_arg_2 >=
       (shift_left(to_signed(0,24),to_integer(to_signed(18,64))));
```

```
50
51 -- split begin
52 split: block
53 signal bv : std_logic_vector(24 downto 0);
54 begin
55 bv <=
       (std_logic_vector((resize((shift_left(to_signed(0,24),to_integer(to_signed(18,64)))),25)
       + resize(to_signed(4096,24),25))));
56 v_case_scrut_app_arg <= (bv(bv'high downto 24)
57 ,bv(24-1 downto 0)
58 );
59 end block;
60 -- split end
61
62 v <= v_case_alt;
63
64 case_scrut_0 <= v < to_signed(262144,24);
65
66 cdamp_satmin_app_arg_2 : entity cdamp_satmin
67 port map
68 (result => app_arg_2
69 ,a      => x
70 ,b      => v);
71
72 with (v_case_scrut) select
73 v_case_alt <= v_case_alt_1 when "0",
74 v_case_alt_0 when others;
75
76 -- msb begin
77 msb : block
78 signal bv_0 : std_logic_vector(24 downto 0);
79 begin
80 bv_0 <=
       (std_logic_vector((resize((shift_left(to_signed(0,24),to_integer(to_signed(18,64)))),25)
       + resize(to_signed(4096,24),25))));
81 v_case_scrut_app_arg_0 <= bv_0(bv_0'high downto bv_0'high);
82 end block;
83 -- msb end
84
85 v_case_scrut <= v_case_scrut_app_arg_0 xor v_app_arg;
86
87 -- msb begin
88 msb_0 : block
89 signal bv_1 : std_logic_vector(23 downto 0);
90 begin
91 bv_1 <=
       (std_logic_vector((shift_left(to_signed(0,24),to_integer(to_signed(18,64))))));
```

```
92  v_case_alt_0_case_scrut_app_arg_app_arg <= bv_1(bv_1'high downto bv_1'high);
93  end block;
94  -- msb end
95
96  -- msb begin
97  msb_1 : block
98  signal bv_2 : std_logic_vector(23 downto 0);
99  begin
100 bv_2 <= (std_logic_vector(to_signed(4096,24)));
101 v_case_alt_0_case_scrut_app_arg_app_arg_0 <= bv_2(bv_2'high downto bv_2'high);
102 end block;
103 -- msb end
104
105 v_case_alt_0_case_scrut_app_arg <= v_case_alt_0_case_scrut_app_arg_app_arg and
        v_case_alt_0_case_scrut_app_arg_app_arg_0;
106
107 with (v_case_alt_0_case_scrut_app_arg) select
108 v_case_alt_0 <= signed'(0 => '0', 1 to 24-1 => '1') when "0",
109 signed'(0 => '1', 1 to 24-1 => '0') when others;
110
111 v_case_alt_1 <= signed(r);
112
113 -- msb begin
114 v_app_arg <= r(r'high downto r'high);
115 -- msb end
116
117 -- split begin
118 split_0: block
119 signal bv_4 : std_logic_vector(24 downto 0);
120 begin
121 bv_4 <=
        (std_logic_vector((resize((shift_left(to_signed(0,24),to_integer(to_signed(18,64)))),25)
        + resize(to_signed(4096,24),25))));
122 r_case_scrut_app_arg <= (bv_4(bv_4'high downto 24)
123 ,bv_4(24-1 downto 0)
124 );
125 end block;
126 -- split end
127
128 r <= r_case_scrut_app_arg.tup2_3_sel1;
129 end;
```

Appendix B

B.1 MATLAB script to calculate system performance

The following script is credited to Tim van Doesum of Axign B.V. This script calculates FFT of the pwm samples recorded in ”cdampout”, csv_haskell and csv_clash obtained from Simulink, Haskell and CλaSH models respectively. This main script is a part of a spectrum analysis toolbox and for general description only the main script is presented here.

```
figIndex = 3;
nfft = 2^18;
fs = 1024*48e3;
bandwidth = [0 20e3];
window = 'hann';


% Settings for generated signal:
ampDB = -20;
amplitude = 10^(ampDB/20);
findesired = 5.8125e3; % near to 6kHz
offset = 0;
Anoise = 1e-6;

fin = calcFFTFreq(findesired, fs, 'nsamples', nfft);

inS = cdampout;
inH = csvread('csv_haskell.csv');
inC = csvread('csv_clash.csv');

% Calculate the FFT for Simulink
[psdS, freqS] = calcFFT(inS, fs, window, nfft);

% Calculate the FFT for Haskell
[psdH, freqH] = calcFFT(inH, fs, window, nfft);

```

```
27 % Calculate the FFT for Clash
28 [psdC, freqC] = calcFFT(inC, fs, window, nfft);
29
30 % function performanceFigures calculates and outputs SNR, Signal power, THD and
       noise power
31 %performanceFigures(psdS, fin, fs, nfft, window, 'bandwidth', bandwidth);
32 %performanceFigures(psdH, fin, fs, nfft, window, 'bandwidth', bandwidth);
33 %performanceFigures(psdC, fin, fs, nfft, window, 'bandwidth', bandwidth);
34
35 % plotting
36 figure(figIndex);
37 title('Simulink model');
38 plotSpectrum(freqS, psdS, nfft, fs, window, 'bandwidth', bandwidth, 'inputFreqs',
       fin);
39
40 figure(figIndex+1);
41 plotSpectrum(freqH, psdH, nfft, fs, window, 'bandwidth', bandwidth, 'inputFreqs',
       fin);
42 title('Haskell model');
43
44 figure(figIndex+2);
45 plotSpectrum(freqC, psdC,freqH, psdH, nfft, fs, window, 'bandwidth', bandwidth,
       'inputFreqs', fin);
46 title('Clash model');
```