

# **Digital Pulse Width Modulation for Class-D Audio Amplifiers**

**Deon Jacobs**

**Thesis presented in partial fulfilment of the requirements for the degree of  
Master of Science in Electronic Engineering  
at the**

**University of Stellenbosch**



*Supervisor:* Prof H. dT. Mouton

**April, 2006**

# Declaration

I the undersigned, hereby declare that the work contained in this thesis is my own original work, unless otherwise stated, and has not previously, in its entirety or in part, been submitted at any university for a degree.

.....  
Deon Jacobs  
December 15, 2005

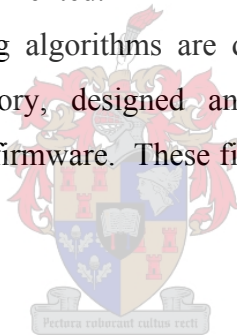


# Abstract

Digital audio data storage mediums have long been used within the consumer market. Today, because of the advancement of processor clock speeds and increased MOSFET switching capabilities, digital audio data formats can be directly amplified using power electronic inverters. These amplifiers known as Class-D have an advantage over there analogue counterparts because of their high efficiency.

This thesis deals with the signal processing algorithms necessary to convert the digital audio data obtained from the source to a digital pulse width modulated signal which controls a full bridge inverter for audio amplification. These algorithms address difficulties experienced in the past which prevented high fidelity digital pulse width modulators to be implemented.

The signal processing algorithms are divided into modular blocks, each of which are defined in theory, designed and simulated in Matlab® and then implemented within VHDL firmware. These firmware blocks are then used to realize a Class-D audio amplifier.



# Opsomming

Digitale oudiodatabergingsmediums is vir 'n geruime tyd al beskikbaar in die verbruikersmark. Vandag, as gevolg van die vooruitgang van verwerker klokfrekwensies en die verhoogte skakelfrekwensie-eienskappe van MOSFET-komponente, kan digitale oudiodataformate direk versterk word deur die gebruik van drywingselektroniese omsetters. Die versterkers staan bekend as klas D-tipes, en is baie meer effektief as analoog klank versterkers.

Hierdie tesis handel oor die seinverwerkingsalgoritmes wat nodig is vir die omskakeling van digitale oudiodata na 'n gemoduleerde digitale pulswydte vir die beheer van 'n volbrug omsetter beheer vir klankversterking. Die algoritmes spreek die struikelblokke aan wat in die verlede verhoed het dat hoëresolusie, digitale pulse wydte modulators geïmplimenteer kon word.

Die seinverwerkingsalgoritmes word verdeel in modulêre blokke. Elk word afsonderlik beskryf in teorie, ontwerp en gesimuleer in Matlab® en geïmplimenteer in VHDL sagteware. Hierdie sagtewareblokke word dan gebruik om 'n klass-D tipe klankversterker te realiseer.



# Acknowledgements

The following acknowledgements are in order:

- Jesus Christ for His salvation.
- My family for their encouragement and prayers.
- Prof. H. dT. Mouton for his guidance.
- Francois Koeslag, Leon de Wit, Neal de Beer and Riaan van den Dool for their friendship, help and support during this project.
- Alfred for all his encouraging words.
- NRF and Thrip for providing funds for this project.



# Contents

Declaration .....	i
Abstract .....	ii
Opsomming .....	iii
Acknowledgements .....	iv
List of Figures .....	viii
List of Tables .....	x
List of Abbreviations .....	xi
List of Symbols .....	xiii
Chapter 1 - Introduction .....	1
1.0 Background .....	1
1.1 Research Objectives for Class-D Audio Amplifiers .....	1
1.2 Digital Audio Amplification .....	2
1.3 Defining High Fidelity .....	3
1.4 PWM Difficulties .....	3
1.5 Thesis Objectives .....	3
Chapter 2 - Premodulation Processing .....	4
2.0 Introduction .....	4
2.1 PWM Linearization .....	4
2.2 Clock Speed Reduction .....	4
2.3 Thesis Structure .....	5
Chapter 3 - Interpolation .....	7
3.0 Introduction .....	7
3.1 Choice of Sampling Rate .....	7
3.1.1 What is sampling? .....	7
3.1.2 Nyquist rate derivation .....	8
3.1.3 Reconstructing the continuous input signal .....	13
3.1.4 Reconstruction as an ideal lowpass characteristic .....	15
3.2 Sampling Rate Increase – by Integer Factors .....	16
3.2.1 Sampling rate conversion system .....	17
3.2.2 Ideal digital lowpass filter necessary .....	18
3.2.3 The impulse response of the lowpass filter .....	21
3.3 Digital Interpolation Filters .....	22
3.3.1 Low pass filter characteristic .....	23
3.3.2 Phase characteristic .....	24
3.3.3 Digital filtering methods .....	25
3.4 Design of a Linear Phase Bandpass FIR Filter .....	27
3.4.1 Choice of a FIR filter design method .....	27
3.4.2 Optimum equiripple linear-phase FIR filters .....	28
3.4.3 FIR filter specifications .....	29
3.4.4 Digital filter design characteristics .....	29
3.4.5 Summary .....	29
3.5 Polyphase FIR Structures for Integer Interpolators .....	29
3.5.1 Polyphase FIR structure .....	29
3.5.2 Properties of polyphase filters .....	29

3.5.3 Conversion to the polyphase structure .....	29
3.5.4 Summary .....	29
3.6 Example of the Interpolation Process .....	29
3.6.1 Cosine input signal .....	29
3.6.2 Sample rate expanded signal .....	29
3.6.3 Polyphase filtering .....	29
3.6.4 Summary .....	29
Chapter 4 - Pulse Width Modulation .....	29
4.0 Introduction .....	29
4.1 Pulse Width Modulation Schemes .....	29
4.1.1 Natural and uniform PWM .....	29
4.1.2 Harmonic components of PWM .....	29
4.1.3 Trailing edge naturally sampled modulation .....	29
4.1.4 Trailing edge uniformly sampled modulation .....	29
4.1.5 Leading edge naturally and uniformly sampled modulation .....	29
4.1.6 Double edge naturally sampled modulation .....	29
4.1.7 Double edge uniformly sampled modulation .....	29
4.1.8 Conclusion of PWM schemes studied .....	29
4.2 Pseudo-Natural Pulse Width Modulation .....	29
4.2.1 What is PNPWM? .....	29
4.2.2 What PNPWM scheme should be used? .....	29
4.2.3 PNPWM building blocks .....	29
4.2.4 Numerical root finding algorithms for PNPWM .....	29
4.3 Summary .....	29
Chapter 5 - Noise shaping .....	29
5.0 Introduction .....	29
5.1 Choice of Switching Frequency .....	29
5.2 Noise-Shaping Coders .....	29
5.2.1 Clock speed constraints .....	29
5.2.2 Bit-size reduction through noise shaping .....	29
5.2.3 Recursive noise shaper .....	29
5.2.4 Noise shaping quantizer .....	29
5.2.5 Derivation of the noise transfer function .....	29
5.2.6 Characteristics of the noise shaper .....	29
5.3 Noise Shaping Simulations .....	29
5.3.1 Noise shaping filter .....	29
5.3.2 Noise shaping of the PNPWM output .....	29
5.4 Summary .....	29
Chapter 6 - Firmware Implementation .....	29
6.0 Introduction .....	29
6.1 Hardware Description .....	29
6.2 Firmware Development .....	29
6.2.1 Configuration firmware .....	29
6.2.2 Digital PWM firmware .....	29
6.2.3 VHDL synthesis .....	29
6.3 Fixed Point Arithmetic .....	29
6.4 Cyclone FPGA Resources .....	29
6.5 Implementation Difficulties .....	29
6.6 Summary .....	29
Chapter 7 - Measurements and Results .....	29

7.0	Introduction.....	29
7.1	Measurement Setup.....	29
7.2	Measurements .....	29
7.2.1	PWM gating signals.....	29
7.2.2	Amplified output measurement.....	29
7.2.3	Frequency response measurement .....	29
7.2.4	THD+N measurement.....	29
7.3	Discussion of Measurement Results .....	29
7.4	Conclusions.....	29
Chapter 8	- Conclusions.....	29
8.0	Overview.....	29
8.1	Fulfillment of Objectives .....	29
8.2	Recommendations and Future Research .....	29
References and Bibliography	.....	29
Appendix A	.....	29
A1.	Interpolation Matlab Code .....	29
Appendix B	.....	29
PWM Spectral Calculation.....		29
B1.	Estimation method.....	29
B2.	Matlab Code for PWM Spectral Estimate.....	29
Appendix C	.....	29
Spectral Estimate Matlab Code for PWM Schemes .....		29
C1.	Trailing edge NPWM.....	29
C2.	Trailing edge UPWM, and PNPWM using Newton's method .....	29
C3.	Trailing edge PNPWM using binary search strategy.....	29
Appendix D	.....	29
DIR 1703.....		29
Appendix E	.....	29
ALTERA CYCLONE (EP1C12Q240C6) Features.....		29
Appendix F	.....	29
VHDL Code.....		29
F1.	Polyphase filtering.....	29
F2.	Polynomial coefficient calculation .....	29
F3.	Binary search .....	29
F4.	Crosspoint calculation.....	29
F5.	Noise shaping.....	29
F6.	PWM generator.....	29



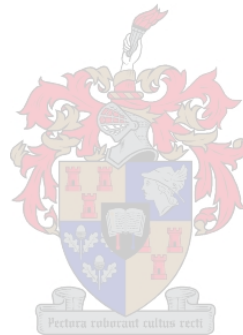
# List of Figures

Figure 1.1: (a) Analogue amplifier topology .....	2
(b) Digital Class-D amplifier topology .....	2
Figure 2.1: Premodulation signal processing blocks. ....	5
Figure 3.1.1: Sampling analogue band-limited signal and aliasing of spectral components altered from [3]. ....	12
Figure 3.1.2: Continuous-time signal generated from discrete-time formula using the reconstruction formula. ....	14
Figure 3.1.3: Frequency representation of perfect reconstruction. ....	15
Figure 3.2.1: Sampling rate conversion system. ....	17
Figure 3.2.2: Block diagram and typical waveforms and spectra for sampling rate increase by a factor of $L$ , altered from [22]. ....	19
Figure 3.3.1: Filter magnitude response with specification parameters altered from [3]. ....	23
Figure 3.4.1: Magnitude response of the equiripple FIR filter. ....	29
Figure 3.4.2: Attenuation of linear FIR filter between 19 kHz and 20 kHz. ....	29
Figure 3.4.3: Impulse response. ....	29
Figure 3.4.4: Phase response. ....	29
Figure 3.4.5: Group delay. ....	29
Figure 3.4.6: Step response. ....	29
Figure 3.4.7: Pole/Zero plot. ....	29
Figure 3.4.8: Zoomed pole/zero plot. ....	29
Figure 3.5.1: Polyphase structures for a 1 to $L$ interpolator. ....	29
Figure 3.5.2: Commutator model for the 1 to $L$ polyphase interpolator. ....	29
Figure 3.5.3: Ideal frequency response of the polyphase networks. ....	29
Figure 3.5.4: Impulse responses of the eight polyphase filters. ....	29
Figure 3.5.5: Magnitude response of polyphase filters. ....	29
Figure 3.5.6: Phase delay of respective polyphase filters. ....	29
Figure 3.6.1: Total interpolation process. ....	29
Figure 3.6.2: 1 kHz sinusoidal input signal $x(n)$ . ....	29
Figure 3.6.3: MTM PSD estimate of $x(n)$ . ....	29
Figure 3.6.4: Sample rate expanded signal $w(n)$ . ....	29
Figure 3.6.5: Zoomed view of $w(n)$ . ....	29
Figure 3.6.6: MTM PSD estimate of $w(n)$ . ....	29
Figure 3.6.8: MTM PSD of output $y(n)$ . ....	29
Figure 4.1: Two-level pulse-width modulator adapted from [17]. ....	29
Figure 4.2: Difference between UPWM and NPWM. ....	29
Figure 4.3: PWM Schemes altered from [17]. ....	29
Figure 4.4: Trailing edge NPWM spectrum. ....	29
Figure 4.5: Trailing Edge UPWM spectrum. ....	29
Figure 4.6: Calculation of PNPWM output signal adapted from [15]. ....	29
Figure 4.7: Building blocks of the PNPWM modulation technique. ....	29
Figure 4.8: Cross Point Derivation. ....	29
Figure 4.9: Newton's method and first two approximations to its zero $\alpha$ . ....	29

Figure 4.10: PNPWM crosspoint derivation.....	29
Figure 4.11: PNPWM output of Newton’s method .....	29
Figure 4.12: Spectrum of PNPWM using Newton’s method.....	29
Figure 4.13: Spectrum of NPWM using Newton’s method.....	29
Figure 4.14: Bisection method and the first two approximations to its zero $\alpha$ .....	29
Figure 4.15: Binary search method and the first two approximations to its crosspoint $\zeta$ .....	29
Figure 4.17: Spectrum of PNPWM using Binary search strategy. ....	29
Figure 4.18: Spectrum of PNPWM using Newton’s method.....	29
Figure 5.1: Noise-shaper architecture altered from [18].....	29
Figure 5.2: Quantizer modeled as added noise source.....	29
Figure 5.3: Example of a midtread quantizer [Digital signal processing textbook]. ...	29
Figure 5.4: Noise Transfer Function at various orders of $N$ . ....	29
Figure 5.5: Magnitude response of fifth order noise transfer function. ....	29
Figure 5.6: Phase response of fifth order noise transfer function. ....	29
Figure 5.7: Pole/Zero plot of $H(z)$ . ....	29
Figure 5.8: Noise shaped 8-bit PWM output. ....	29
Figure 5.9: Zoomed view of 8-bit PWM output. ....	29
Figure 5.10: Noise shaped 10-bit PWM output. ....	29
Figure 5.11: Zoomed view of 10-bit PWM output. ....	29
Figure 6.1: Signal processing building block for PCM to PWM conversion. ....	29
Figure 6.2: Firmware blocks developed within the FPGA. ....	29
Figure 6.3: Blockdiagram of the interpolation process.....	29
Figure 6.5: Block diagram of polynomial coefficient calculation. ....	29
Figure 6.9: State diagram of the binary search process. ....	29
Figure 6.13 Timing diagram description of firmware.....	29
Figure 6.14: Mealy Machine [6]. ....	29
Figure 6.15: Example of a two-process Moore state machine.....	29
Figure 6.16: A double synchronizer circuit. ....	29
Figure 6.17: Data bus synchronization between asynchronous clock domains. ....	29
Figure 6.18: Quartus II flow summary.....	29
Figure 7.1: Digital modulation measurement setup. ....	29
Figure 7.2: Complete measurement setup. ....	29
Figure 7.3: Zoomed view of digital modulation development board. ....	29
Figure 7.4: PWM gating output signal from FPGA.....	29
Figure 7.5: Single cycle of PWM gating signal. ....	29
Figure 7.6: Single cycle of PWM gating signal. ....	29
Figure 7.7: Amplified 1 kHz sinusoidal output.....	29
Figure 7.8: Amplified 10 kHz sinusoidal output.....	29
Figure 7.9: Amplified 12 kHz sinusoidal output.....	29
Figure 7.10: Amplified 16 kHz sinusoidal output.....	29
Figure 7.11: Frequency response of Class-D amplifier system. ....	29
Figure 7.11: THD+N across the audio band. ....	29
Figure B1: PWM signal. ....	29
Figure B.2: Spectrum estimate calculation. ....	29

# List of Tables

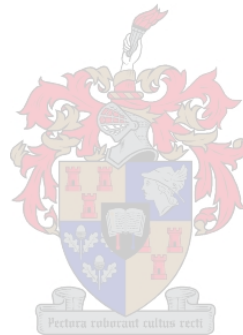
Table 3.3.1: Comparison between digital filtering methods.....	26
Table 3.4.1: Interpolation filter specifications.....	29
Table 3.5.1: FIR filter structures.....	29
Table 3.6.1: Harmonic intervals at which baseband frequencies are centered.....	29
Table 3.6.2: Specifications of low-pass digital filter.....	29
Table 4.1: Polynomial interpolation methods.....	29
Table 4.2: Arithmetic counts of one iteration using Newton’s method.....	29
Table 4.3: Arithmetic counts of one iteration using the binary search method.....	29
Table 4.4: Arithmetic counts of one iteration using the binary search method and lookup table.....	29
Table 4.5: Total arithmetic complexity of the two PNPWM methods.....	29
Table 5.1: Needed clock rates for certain PWM bit resolutions.....	29
Table 6.1: Clock speeds of different processes.....	29
Table 6.2: Firmware development time.....	29



# List of Abbreviations

CD	Compact Disk
CDA	Compact Disk Audio
Class-D	Systems using a digital PWM topology
DAC	Digital to Analogue Converter
DIR	Digital Interface Receiver
DSP	Digital Signal Processing
DUT	Device Under Test
DVD	Digital Video Disc
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
IC	Integrated Circuit
IIR	Infinite Impulse Response
LSB	Least Significant Bit
MAC	Multiply Accumulate
MOSFET	Metal-Oxide Semiconductor Field Effect Transistor
MSB	Most Significant Bit
MTM	Multitaper Thomson Method
NPWM	Natural Pulse Width Modulation
ONS	Oversampling Noise Shapers
PBR	Pass Band Ripple
PCM	Pulse Code Modulation
PLL	Phase Lock Loop
PNPWM	Pseudo Natural Pulse Width Modulation
PSD	Power Spectral Density
PWM	Pulse Width Modulation
RAM	Random Access Memory
ROM	Read Only Memory

SBR	Stop Band Ripple
SNR	Signal to Noise Ratio
SPDIF	Sony Philips Digital Interface
THD+N	Total Harmonic Distortion plus Noise
UPWM	Uniform Pulse Width Modulation
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit



# List of Symbols

$F$	Infinite frequency variable
$f$	Normalized frequency variable
$T_s$	Sampling period
$F_s$	Sampling frequency
$F_c$	Switching frequency
$F_p$	Discrete signal spectrum period
$F_{\max}$	Maximum range of $f$
$B$	Signal Bandwidth
$G$	Interpolation filter gain
$F'$	Upsampled sampling frequency
$T'$	Upsampled sampling frequency
$L$	Upsampling factor
$W(z)$	Sampling rate expander z-transform
$X(z)$	Digital input signal z-transform
$X_a(F)$	Continuous signal spectrum
$X(f) = X(F / F_s)$	Aperiodic Discrete time signal spectrum
$\omega$	Corner frequency in rad/s
$\omega_p$	Passband edge ripple
$\omega_s$	Stopband edge ripple
$\delta_p$	Passband ripple
$\delta_s$	Stopband ripple
dB	Decibel
dBFS	Decibels with respect to digital full scale
dBV	Decibels relative to a reference value of 1.000 Volts
Hz	Hertz
$V_{p/p}$	Volts peak to peak

# Chapter 1 - Introduction

## 1.0 Background

The Class-D mode of operation was originally introduced in 1959 by Baxandall for the potential application in oscillator circuits [20]. Pulse width modulation (PWM) is well established in power electronics as a basis for controlling inverters with sinusoidal output voltages and motor drives [12]. Today PWM is becoming more prevalent in high quality DACs (Digital-to-Analogue Converters), particularly those used in digital audio applications [19]. Audio amplifiers implementing the PWM strategy have emerged on a great scale.

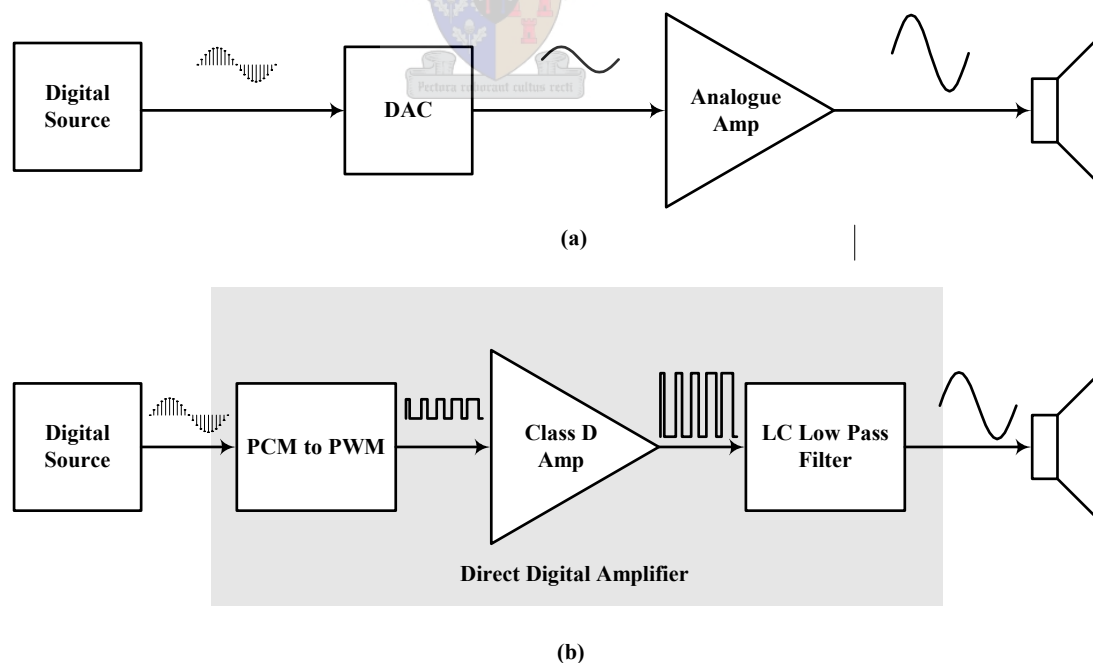
## 1.1 Research Objectives for Class-D Audio Amplifiers

Two primary objectives drive the research within digital Class-D audio amplifiers. The primary objective of Class-D amplifiers is high efficiency. Conventional audio amplifiers (analogue) rarely exceed 20 % efficiency in use. An amplifier based on a PWM inverter, in contrast, can reach 90 % efficiency or more [12]. The lower power losses therefore decrease or even eliminate the use of heat sinks. It is thus evident that the higher levels of efficiency translate into smaller, lower cost designs. The potential efficiency improvement for battery-powered applications or for miniature amplifiers has played a large part in driving the study of advanced PWM amplification techniques [12].

The second research objective of these amplifier topologies is to amplify digital audio data directly. This need has arisen because of the growing use of digital audio in compact disks (CDs), DVDs (Digital Video Disc), movie soundtracks, broadcasting, and computer applications. Most amplifiers today firstly need to convert these digital sources to small voltage analogue signals before amplification can be performed. It is desired that the audio data remain in the digital domain through the amplification process, and only be converted to the analogue domain at the output stage.

## 1.2 Digital Audio Amplification

PWM provides a medium for digital audio amplification. It encodes a signal into two discrete levels, with the information represented in pulse duty ratios. This coding characteristic enables energy to be delivered to the output by switching power transistors which are either fully ON or fully OFF. The gating signals fed to these transistors represent the encoded signal, while the high voltage ON and OFF outputs represent the discrete and therefore digital amplified output. An advantage of this modulation is its ability to recover the amplified discrete-level form with a passive output filter. When the discrete power sources for the power transistors are generated efficiently, PWM provides the basis for highly efficient signal delivery, especially to loads with low-pass characteristics [12]. Thus a digital PWM signal prepared from an audio input can be used as a switching function for a full bridge, half bridge inverter, where a low pass filter extracts the audio and delivers it to a loudspeaker. Figure 1.1 compares the analogue amplifier topology with the proposed digital Class-D amplifier topology.



**Figure 1.1: (a) Analogue amplifier topology  
(b) Digital Class-D amplifier topology.**



### 1.3 Defining High Fidelity

With the extensive growth of digital audio, the digital characteristics of a signal provide a basis for defining high fidelity. For example the 16-bit signal from a CDA (Compact Disc Audio) source has 1-bit quantization error as the lower bound on noise and distortion. This is one part in  $2^{16}$ , or -96 dB. A 24-bit audio sampling range corresponds to a lower bound of -144 dB. An amplifier that can reach these low levels is effectively perfect by comparison with the audio signal quantization error [12].

### 1.4 PWM Difficulties

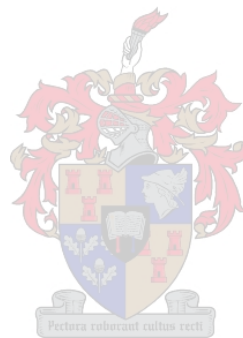
Two main difficulties however continue to be associated with PWM based conversion systems, these are practicality and performance. Excessive modulator clock speeds are required to resolve  $2^{16}$  or  $2^{24}$  distinct pulse widths per pulse interval. Moreover, harmonic distortion inherent to the uniform sampling modulation processes makes 16-bit performance very difficult to achieve [17]. Fortunately, methods have been developed within the digital signal processing field which successfully solves these shortcomings associated with uniform sampled modulation. These methods are known as premodulation, predistortion signal processing linearization algorithms, and have been described in [19], [17], [15].

### 1.5 Thesis Objectives

This thesis addresses these PWM difficulties of performance and practicality by using the premodulation, predistortion algorithms mentioned above. It does this by:

- Identifying how these algorithms address these difficulties.
- Dividing these algorithms into appropriate blocks.
- Sufficiently describing each block in theory.
- Presenting a design solution for each block.
- Simulating these designs in Matlab®.
- Developing VHDL firmware of the simulated designs.

- Attempting to realize a practical Class-D amplifier using the developed firmware.



# Chapter 2 - Premodulation Processing

## 2.0 Introduction

Chapter 1 introduced the difficulties associated with digital PWM, these are practicality and performance. Chapter 1 also introduced a solution of these difficulties which are premodulation precompensating linearization algorithms. Here these algorithms are outlined briefly and it is shown which of these algorithms addresses which digital PWM difficulty. After the brief outline, a description of the thesis structure follows.

## 2.1 PWM Linearization

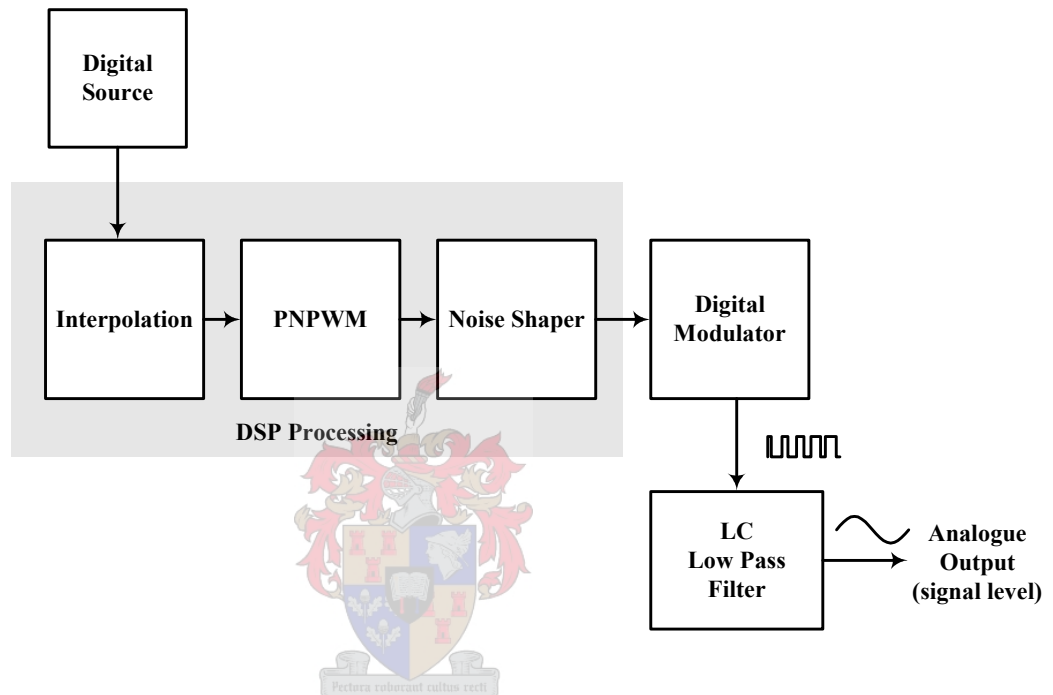
The fundamental problem of PWM based open-loop digital Class-D audio power amplifiers are the inherent nonlinearity of the PWM process, this necessitates the application of linearization algorithms.

Non-linearity of the PWM process is reduced through increasing the sampling rate of the digital audio signal applied to the modulator input. Unfortunately this increased rate has no effect on the in band harmonic distortion resulting from the uniform PWM process. However, a DSP technique has been proposed which has the ability to imitate the natural PWM process digitally resulting in negligible in band harmonic distortion [19]. This idea is called 'pseudonatural PWM (PNPWM)'. Interpolation (upsampling) and PNPWM therefore address the problem of performance through linearizing the digital PWM process.

## 2.2 Clock Speed Reduction

The second problem of practicality is addressed through a DSP process known as noise shaping. This technique reduces the word length of the PWM output since high fidelity PWM outputs of 16-bits or more cannot be realized. Excessively high clock speeds are necessary to output these high fidelity signals which cannot be attained even by today's advanced digital signal processing devices. Noise shaping has the ability to therefore reduce the word length of the PWM output signal but still

attain negligible loss in baseband signal quality. The word length is reduced to such an extent that current processor clock speeds suffices. The resulting digital DAC which consist of these algorithms are shown in Figure 2.1. This figure shows the extent to which this thesis investigates digital Class-D amplifiers. It should be noted that the amplifier stage and low pass output filter is overlooked in the investigation, therefore falling outside the scope of this thesis.



**Figure 2.1: Premodulation signal processing blocks.**

## 2.3 Thesis Structure

The precompensation linearization algorithms are the main focus of this thesis and are represented by the different blocks illustrated in Figure 2.1. Each of these blocks is dealt with separately within its own chapter, and after its description and design, simulations are given which prove their functionality.

The first block known as interpolation is presented in Chapter 3. It starts with the description of sampling a continuous signal which leads to the relation that exists between the continuous and digital domains. This relation provides insight into digital filtering which is the fundamental concept in the implementation of the interpolation process. After this, a digital filter is designed for interpolation and an efficient structure is described for its implementation.

Chapter 4 defines the pulse width modulation process completely. It investigates and compares various PWM schemes using a two dimensional Fourier analysis. From these comparisons a desired PWM scheme is identified. The idea of PNPWM is then described, consisting of calculating the crosspoint between the audio input signal and a carrier wave through polynomial interpolation, linear interpolation and iterative root finding algorithms. Two numerical methods for calculating the crosspoint in the PNWPM scheme are compared and a choice between these is then made for practical implementation.

Chapter 5 describes the noise shaping coder which has the ability to reduce the resolution of a digital signal but still retain a certain SNR within a specific band. Within this chapter 5<sup>th</sup> order 8-bit and 10-bit noise shaping coders are considered.

Chapter 6 uses the knowledge gained from the previous three chapters which describe all the precompensation linearization algorithms, in theory, and in simulation to develop VHDL firmware for a practical implementation within a FPGA. The developed firmware is described through relevant block, time and state diagrams.

Chapter 7 gives relevant measurements concerning the implementation of the VHDL firmware, and then interprets them.

Chapter 8 summarizes and concludes the thesis by providing an overview of the work done and discusses possible future research opportunities.

# Chapter 3 - Interpolation

## 3.0 Introduction

Before any modulation can be performed on the digital audio input signal, its sampling rate ( $F_s$ ) firstly needs to be converted to that of the PWM switching frequency ( $F_c$ ). The switching frequency is at a higher rate than the audio input sampling frequency to increase the linearity of the PWM process (described in Chapter 4), and for use by the noise shaping process (described in Chapter 5). This process of sample rate conversion needs to retain all of the audio information since any loss would result in some form of distortion which will then be reflected in the modulated PWM signal.

Increasing the sampling rate implies that a certain number of equidistant samples are placed between the original signal samples at amplitudes that agree with the original signal. This process is described and implemented in this chapter from a digital signal processing viewpoint.

The chapter starts off with one of the most fundamental concepts of digital signal processing, which is the idea of sampling a continuous signal to provide a set of discrete numbers. It then describes the sample rate conversion process which concludes that digital filtering is the fundamental ingredient to interpolation. The remainder of the chapter focuses on the choice, design and filter structure and simulation of the digital low pass interpolation filter.

## 3.1 Choice of Sampling Rate

### 3.1.1 What is sampling?

A continuous signal firstly needs to be converted into a discrete sequence before any digital signal processing can be performed on it. This conversion process is called sampling and is done by capturing and truncating the continuous input signal amplitude at equidistant intervals.

If an analogue input signal  $x_a(t)$  needs to be “digitized” (where  $t$  is the continuous time variable), it is sampled at an interval known as the sampling period ( $T_s$ ) resulting in a discrete sequence  $x[n]$  given by

$$x[n] = x_a(nT_s) \quad -\infty < n < \infty, \quad (3.1.1)$$

where  $n$  is the discrete index variable.

The sampling frequency ( $F_s = 1/T_s$ ) must be selected so that it is large enough to ensure that the original continuous signal  $x_a(t)$  is recoverable from its sampled counterpart  $x[n]$ . The correct choice of the sampling frequency is known as the Nyquist rate which gives insight on the relation between the continuous  $X_a(F)$  and discrete  $X(f)$  spectra of these signals. Understanding the relation between these spectra leads to the understanding of digital filtering. An important foundation for this understanding is the Nyquist theorem, which will subsequently be derived using [3].

### 3.1.2 Nyquist rate derivation

Assuming  $x_a(t)$  is an aperiodic analogue signal its spectrum is given by the Fourier transform [3],

$$X_a(F) = \int_{-\infty}^{\infty} x_a(t) e^{-j2\pi Ft} dt \quad (3.1.2)$$

whereas its inverse Fourier transform is given by

$$x_a(t) = \int_{-\infty}^{\infty} X_a(F) e^{j2\pi Ft} dF. \quad (3.1.3)$$

The spectrum of the discrete signal  $x[n]$  sampled from  $x_a(t)$  has the Fourier transform

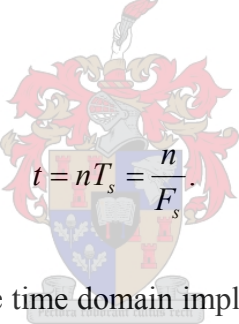
$$X(f) = \sum_{n=-\infty}^{\infty} x[n] e^{-j2\pi fn}. \quad (3.1.4)$$

The discrete signal  $x[n]$  can be recovered from the frequency domain by the inverse Fourier transform given by

$$x[n] = \int_{-1/2}^{1/2} X(f) e^{j2\pi fn} df. \quad (3.1.5)$$

It is known from (3.1.5) that the discrete signal spectrum is finite and repeats periodically with frequency equaling the sampling frequency ( $F_p = F_s$ ).

In order to determine the relationship between the spectra of the discrete signal and the continuous signal, it can be observed that periodic sampling imposes a relationship between the independent variables  $t$  and  $n$  in the signals  $x_a(t)$  and  $x[n]$  respectively. That is,

$$t = nT_s = \frac{n}{F_s}. \quad (3.1.6)$$


The relationship in the time domain implies a corresponding relationship in the frequency domain between the variables  $F$  (infinite frequency variable) and  $f$  in  $X_a(F)$  and  $X(f)$  respectively. Indeed substitution of (3.1.1) into (3.1.3) yields

$$x_a(nT_s) = \int_{-\infty}^{\infty} X_a(F) e^{j2\pi nF/F_s} dF. \quad (3.1.7)$$

When comparing (3.1.7) with (3.1.5) it is concluded that

$$\int_{-1/2}^{1/2} X(f) e^{j2\pi fn} df = \int_{-\infty}^{\infty} X_a(F) e^{j2\pi nF/F_s} dF. \quad (3.1.8)$$

The relationship between the variables  $F$  and  $f$  is formulated by

$$f = \frac{F}{F_s}. \quad (3.1.9)$$



Where it is noted that  $f$  is a normalised frequency variable of  $F$ . This implies that the process of periodic sampling of a continuous-time signal causes a mapping of the infinite frequency range of the variable  $F$  onto a finite frequency range for the variable  $f$ . Where the maximum range of  $f$  is limited to  $F_{\max} = \frac{F_s}{2}$ .

Now with the relation in (3.1.9) a simple change of variable is made in (3.1.8) to obtain the result

$$\frac{1}{F_s} \int_{-F_s/2}^{F_s/2} X\left(\frac{F}{F_s}\right) e^{j2\pi nF/F_s} dF = \int_{-\infty}^{\infty} X_a(F) e^{j2\pi nF/F_s} dF. \quad (3.1.10)$$

The integration range of the right-hand side integral can be divided into an infinite number of intervals of width  $F_s$  because of the periodical spectral property. Therefore the integral over the infinite range can be expressed as a sum of integrals. The right-hand expression of (3.1.10) is now given by

$$\int_{-\infty}^{\infty} X_a(F) e^{j2\pi nF/F_s} dF = \sum_{k=-\infty}^{\infty} \int_{(k-1/2)F_s}^{(k+1/2)F_s} X_a(F) e^{j2\pi nF/F_s} dF \quad (3.1.11)$$

It is observed that  $X_a(F)$  in the frequency interval  $(k-1/2)F_s$  to  $(k+1/2)F_s$  is identical to  $X_a(F - kF_s)$  in the interval  $-F_s/2$  to  $F_s/2$ . Therefore,

$$\begin{aligned} \sum_{n=-\infty}^{\infty} \int_{(k-1/2)F_s}^{(k+1/2)F_s} X_a(F) e^{j2\pi nF/F_s} dF &= \sum_{k=-\infty}^{\infty} \int_{(-1/2)F_s}^{(1/2)F_s} X_a(F - kF_s) e^{j2\pi nF/F_s} dF \\ &= \int_{(-1/2)F_s}^{(1/2)F_s} \left[ \sum_{k=-\infty}^{\infty} X_a(F - kF_s) \right] e^{j2\pi nF/F_s} dF \end{aligned} \quad (3.1.12)$$

where the periodicity of the exponential is used, namely,

$$e^{j2\pi n(F+kF_s)/F_s} = e^{j2\pi nF/F_s} \quad (3.1.13)$$

Comparing (3.1.12), (3.1.11) and (3.1.10), it is concluded that

$$X\left(\frac{F}{F_s}\right) = \sum_{k=-\infty}^{\infty} X_a(F - kF_s), \quad (3.1.14)$$

or, equivalently,

$$X\left(\frac{F}{F_s}\right) = \sum_{k=-\infty}^{\infty} X_a(f - k)F_s. \quad (3.1.15)$$

This is the desired relationship between the spectra  $X(F/F_s)$  or  $X(f)$  of the discrete and the spectrum  $X_a(F)$  of the continuous signal. The right-hand side of (3.1.12) and (3.1.14) consists of a periodic repetition of the scaled spectrum  $F_s X_a(F)$  with period  $F_s$ . This periodicity follows as a consequence of the spectrum  $X(f)$  of the discrete signal having a period of  $f_p = 1$  or  $F_p = F_s$  as mentioned previously.

Because of the periodicity of the discrete-time spectrum  $X_a(F/F_s)$  and its relation to  $X(F)$ , a constraint is placed on the analogue signal spectrum  $X_a(F)$  to be bandlimited.

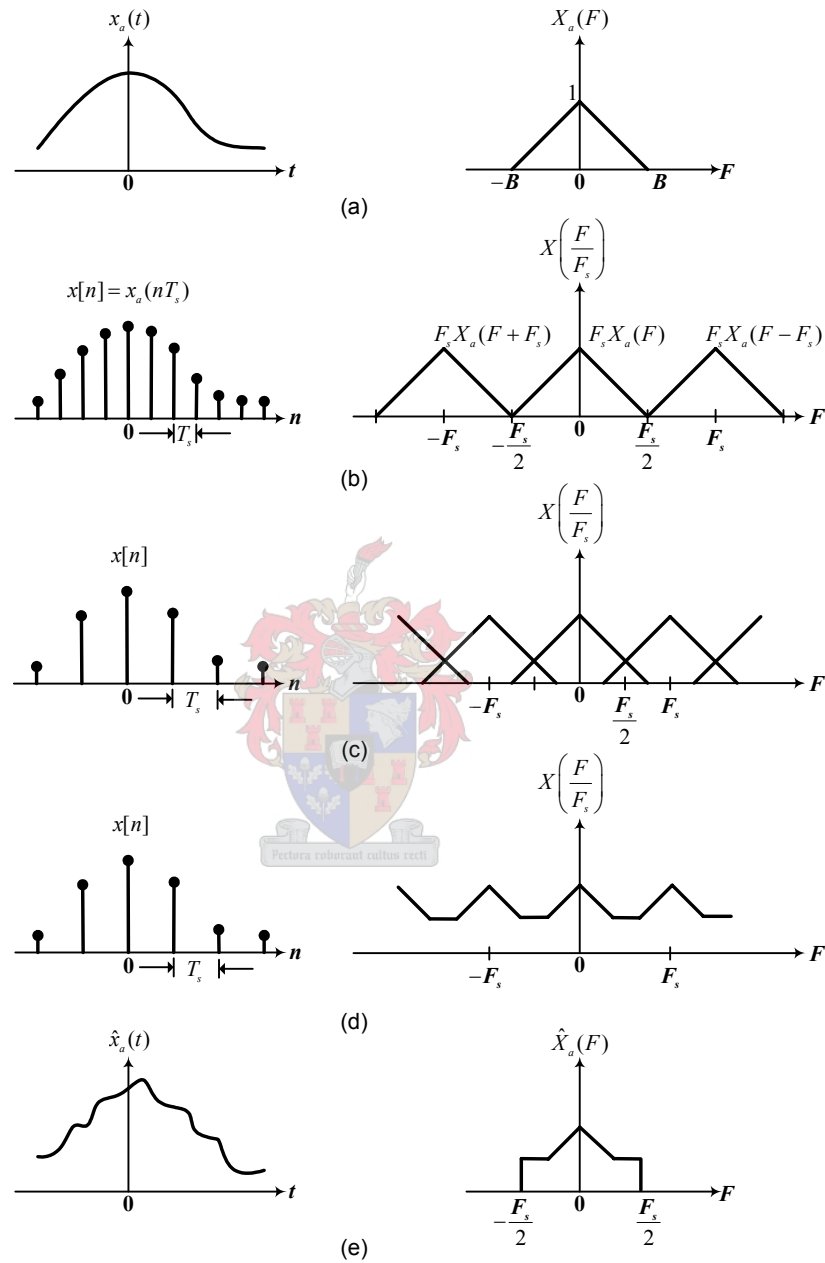
Therefore if all of the spectral content within the analogue signal is to be preserved within the digital signal's spectrum (without distortion), the sampling frequency choice needs to be twice the band-limit of the continuous signal frequency, which is given by

$$F_s = 2B \quad (3.1.16)$$

where  $B$  is the band-limit of  $X_a(F)$ . The relation in (3.1.16) is known as the Nyquist sampling rate.

A bandlimited signal  $x_a(t)$  is shown in Figure 3.1.1(a). If the sampling rate for this continuous signal  $F_s$  is chosen according to the Nyquist rate, the discrete signal  $x[n]$  shown in Figure 3.1.1(b) is the result. No distortion within its spectral content is observed because its baseband frequency content does not overlap with its neighbour's. The baseband frequency content of Figure 3.1.1(b) is given by

$$X\left(\frac{F}{F_s}\right) = F_s X_a(F) \quad |F| \leq \frac{F_s}{2}. \quad (3.1.17)$$



**Figure 3.1.1: Sampling analogue band-limited signal and aliasing of spectral components altered from [3].**

It is therefore observed that the spectrum of the discrete signal is identical (within the scale factor  $F_s$ ) to the spectrum of the analogue signal, within the fundamental frequency range  $|F| \leq F_s/2$  or  $f \leq 1/2$ .

However if  $F_s$  is chosen at a lower rate than the Nyquist rate the discrete signal spectrum  $X(F/F_s)$  includes aliasing as a consequence of the original analogue spectrum  $X_a(F)$  overlapping with its corresponding neighbour. This phenomenon is shown in Figure 3.1.1(c) and (d). The end result is that the aliasing which occurs prevents the recovery of the original signal  $x_a(t)$  from the samples of  $x[n]$  as shown in Figure 3.1.1(e).

### 3.1.3 Reconstructing the continuous input signal

Given the discrete-time signal  $x[n]$  with the spectrum  $X(F/F_s)$ , as illustrated in Figure 3.1.1(b), with no aliasing, it is now possible to reconstruct the original analogue signal from the samples  $x[n]$ . Since in the absence of aliasing it is known that

$$X_a(F) = \begin{cases} \frac{1}{F_s} X\left(\frac{F}{F_s}\right), & |F| \leq \frac{F_s}{2} \\ 0, & |F| > \frac{F_s}{2} \end{cases}, \quad (3.1.18)$$

and by the Fourier transform relationship (3.1.4),

$$X\left(\frac{F}{F_s}\right) = \sum_{n=-\infty}^{\infty} x(n) e^{-j2\pi n F / F_s}. \quad (3.1.19)$$

The inverse Fourier transform of  $X_a(F)$  is

$$x_a(t) = \int_{-F_s/2}^{F_s/2} X_a(F) e^{j2\pi Ft} dF. \quad (3.1.20)$$

If it is assumed that  $F_s = 2B$ , and with the substitution of (3.1.19) into (3.1.18), and (3.1.18) into (3.1.20), the reconstruction function is given as

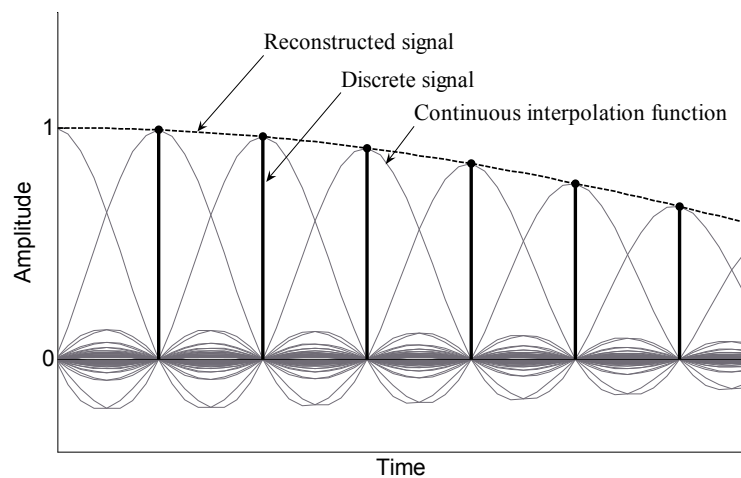
$$\begin{aligned}
 x_a(t) &= \frac{1}{F_s} \int_{-F_s/2}^{F_s/2} \left[ \sum_{n=-\infty}^{\infty} x[n] e^{-j2\pi nF/F_s} \right] e^{j2\pi Ft} dF \\
 &= \frac{1}{F_s} \sum_{n=-\infty}^{\infty} x[n] \int_{-F_s/2}^{F_s/2} e^{j2\pi(t-n/F_s)F} dF \\
 &= \sum_{n=-\infty}^{\infty} x(nT_s) \frac{\sin\left[\frac{\pi}{T_s}(t-nT_s)\right]}{\frac{\pi}{T_s}(t-nT_s)}.
 \end{aligned} \tag{3.1.21}$$

Observing that  $x[n] = x_a(nT_s)$ , and that  $T_s = 1/F_s = 1/2B$  is the sampling interval.

From the reconstruction formula in (3.1.21) it is seen that reconstructing  $x_a(t)$  from  $x[n]$  is a complicated process, involving a weighted sum of the ideal interpolation function,

$$g(t) = \frac{\sin\left[\left(\frac{\pi}{T_s}\right)t\right]}{\left(\frac{\pi}{T_s}\right)t} \tag{3.1.22}$$

and its time-shifted versions  $g(t - nT_s)$  for  $-\infty < n < \infty$ , where the weighting factors are the samples of  $x[n]$ . Figure 3.1.2 illustrates graphically how the continuous-time signal is reconstructed using its sampled counterpart as weights and then convolving with  $g(t)$ .



**Figure 3.1.2: Continuous-time signal generated from discrete-time formula using the reconstruction formula.**

The reconstruction function forms the basis of the sampling theorem which after proof can now be stated for completeness:

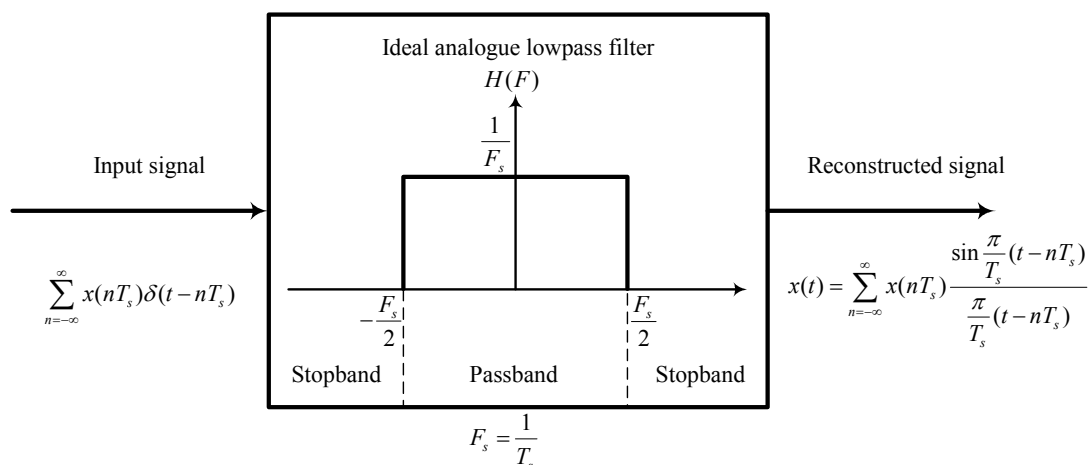
*A bandlimited continuous time signal, with highest frequency (bandwidth)  $B$  Hertz, can be uniquely recovered from its samples provided that the sampling rate is  $F_s \geq 2B$  samples per second.*

### 3.1.4 Reconstruction as an ideal lowpass characteristic

The reconstruction formula given by (3.1.22) in the time-domain could alternatively be seen as a linear filtering process in which a discrete sequence of short pulses (ideally impulses) with amplitudes equal to the signal excites an analogue lowpass filter. The analogue filter corresponding to the ideal interpolation function has the frequency response given by

$$H(F) = \begin{cases} T_s, & |F| \leq \frac{1}{2T_s} = \frac{F_s}{2} \\ 0, & |F| > \frac{1}{2T_s} \end{cases} \quad (3.1.23)$$

The filtering process involving (3.1.23) is illustrated in Figure 2.1.3 where  $H(F)$  is simply the Fourier Transform of the impulse response  $g(t)$  of (3.1.22).



**Figure 3.1.3: Frequency representation of perfect reconstruction.**

$H(F)$  is known as the ideal filter characteristic or an anti-image filter because of its constant gain in the passband and zero gain in the stopband. This brick wall cut-off property prevents any frequency images from passing, and only allows baseband signals to remain completely unchanged.

In all cases, such filters are not physically realisable but serve as a mathematical idealisation of practical filters. For example, from the impulse response given by (3.1.22) it is noted that the filter which it represents (3.1.23) is not causal and is not absolutely summable and therefore also unstable. Consequently, the filter described in (3.1.23) is physically unrealisable but has a use as a benchmark to compare the performance of finite practically realisable low pass filters.

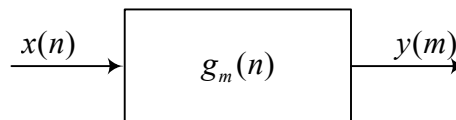
In this subsection the sampling theorem was derived to gain insight on the relation between the spectra of continuous and discrete signal. It was shown that a continuous time signal could be recovered from its discrete counterpart using the reconstruction formula. Unfortunately this formula only serves as a mathematical model to derive the ideal lowpass filter characteristic, and therefore cannot be used practically.

### 3.2 Sampling Rate Increase – by Integer Factors

The process of perfect reconstruction of a discrete signal to its original continuous signal is impossible as was concluded in Section 3.1.4. For increasing the sampling rate of a digital signal from its present rate to another sampling rate a similar reconstruction process is needed. But instead of trying to recover continuous time information as with perfect reconstruction, only fixed distant discrete points between consecutive samples of the input discrete signal is necessary. Since only discrete values are necessary between samples, a relaxation on the low pass filter performing the reconstruction process is implied, which is a discrete filter rather than a continuous one. The process of increasing a discrete signal's sampling rate or interpolating it at fixed intervals with a discrete filter will now be derived theoretically.

### 3.2.1 Sampling rate conversion system

Figure 3.2.1 shows a general description of a sampling rate conversion system. The input  $x(n)$  is a band limited discrete signal sampled at the Nyquist rate  $F_s = 1/T_s \geq 2B$ , and it is desired to attain the output signal  $y(m)$  with a higher sampling rate of  $F' = 1/T'$ . Where  $m$  defines the higher rate discrete index.



**Figure 3.2.1: Sampling rate conversion system.**

If the desired sampling rate increase is a rational factor  $L$  then the new sampling period is given by

$$\frac{T'}{T_s} = \frac{1}{L} \quad (3.2.1)$$

and the new sampling rate  $F'$  is  $F' = LF_s$ . This process of increasing the sampling rate of a signal  $x(n)$  by  $L$  implies that  $L-1$  new samples need to be placed in between  $x(n)$  adjacent samples. Figure 3.2.2 illustrates the process of increasing the sampling rate by a rational of factor  $L=3$ . The input signal is “filled-in” with  $L-1$  zeros between each pair of samples of  $x(n)$  resulting in the signal

$$w(m) = \begin{cases} x(m/L), & m = 0, \pm L, \pm 2L, \dots \\ 0, & \text{otherwise.} \end{cases} \quad (3.2.2)$$

The block diagram in Figure 3.2.2 with an up-arrow symbol corresponds to an increase in sampling rate, resulting in an output signal given by (3.2.2) which is referred to as a sampling rate expander.



### 3.2.2 Ideal digital lowpass filter necessary

It will now be shown how an ideal digital lowpass filter reconstructs the sampling rate expander signal  $w(m)$  to the desired interpolated output signal  $y(m)$  using theory presented in [22].

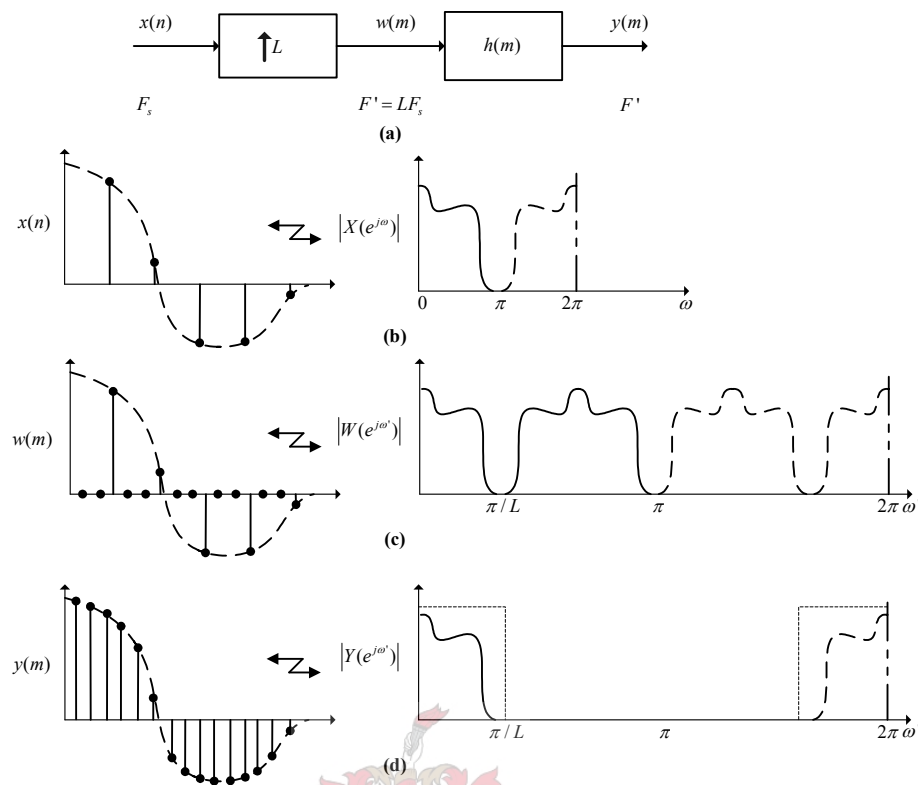
The z-transform of  $w(m)$  given by (3.2.2) yields

$$\begin{aligned} W(z) &= \sum_{m=-\infty}^{\infty} w(m)z^{-m} \\ &= \sum_{m=-\infty}^{\infty} x(m)z^{-mL} \\ &= X(z^L) \end{aligned} \tag{3.2.3}$$

Evaluating  $W(z)$  on the unit circle  $z = e^{j\omega'}$ , gives the result

$$W(e^{j\omega'}) = X(e^{j\omega'L}) \tag{3.2.4}$$

which is the Fourier transform of the signal  $w(m)$  expressed in terms of the spectrum of the input signal  $x(n)$  (where  $\omega' = 2\pi fT'$  and  $\omega = 2\pi fT_s$ ).



**Figure 3.2.2: Block diagram and typical waveforms and spectra for sampling rate increase by a factor of  $L$ , altered from [22].**

The spectrum of  $w(m)$  shown in Figure 3.2.2 contains not only the baseband frequencies of interest (i.e.  $-\pi/L$  to  $\pi/L$ ) but also images of the baseband centered at harmonics of the original sampling frequency  $\pm 2\pi/L, \pm 4\pi/L, \dots$ . To recover the baseband signal of interest and eliminate the unwanted higher frequency components it is necessary to filter the signal  $w(m)$  with a digital low-pass filter which approximates the ideal characteristic

$$\boxed{H}(e^{j\omega'}) = \begin{cases} G, & |\omega'| \leq \frac{2\pi fT'}{2} = \frac{\pi}{L} \\ 0, & \text{otherwise.} \end{cases} \quad (3.2.5)$$

The ideal digital low-pass filter characteristic given in (3.2.5) has a similarity to the ideal low-pass continuous filter characteristic given in (3.1.18), which is, equal cut-off frequencies. They are dissimilar, in that their passband gains are not the same. The ideal continuous characteristic has a gain of  $T_s$  in its passband whereas the digital characteristic has a gain of  $G$  in its passband. It will be shown that in order to ensure

that the amplitude of the upsampled signal  $y(m)$  is correct, the gain of the filter  $G$  must be  $L$  in the passband.

Letting  $H(e^{j\omega'})$  denote the frequency response of an actual filter that approximates the characteristic in (3.2.4) it is seen from [22] that

$$Y(e^{j\omega'}) = H(e^{j\omega'})X(H(e^{j\omega'L})) \quad (3.2.6)$$

and within the approximation of (2.2.4) yielding

$$Y(e^{j\omega'}) = \begin{cases} GX(e^{j\omega'L}), & |\omega'| \leq \pi/L \\ 0, & \text{otherwise.} \end{cases} \quad (3.2.7)$$

With the aid of Figure 3.2.2 and (3.2.5) it is seen that

$$\begin{aligned} y(0) &= \int_{-\pi}^{\pi} Y(e^{j\omega'}) d\omega' \\ &= \int_{-\pi}^{\pi} H(e^{j\omega'}) X(e^{j\omega'L}) d\omega' \\ &= G \int_{-\pi/L}^{\pi/L} X(e^{j\omega'L}) d\omega' \\ &= G \int_{-\pi}^{\pi} X(e^{j\omega}) d\omega / L \\ &= \frac{G}{L} x(0). \end{aligned} \quad (3.2.8)$$

Therefore, a gain  $G = L$  is required to match the amplitude of the envelopes of the signals  $y(m)$  and  $x(n)$ .

### 3.2.3 The impulse response of the lowpass filter

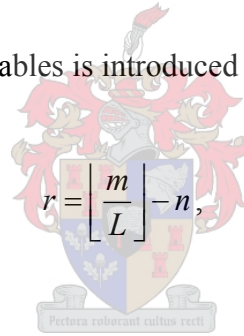
If  $h(m)$  denotes the unit sample response of  $H(e^{j\omega'})$ , then  $y(m)$  can be expressed from [22] as

$$y(m) = \sum_{k=-\infty}^{\infty} h(m-k)w(k). \quad (3.2.9)$$

Combining (3.2.2) and (3.2.9) leads to

$$\begin{aligned} y(m) &= \sum_{k=-\infty}^{\infty} h(m-k)x(k/L) \\ &= \sum_{r=-\infty}^{\infty} h(m-k)x(r). \end{aligned} \quad (3.2.10)$$

Next a change of variables is introduced



$$r = \left\lfloor \frac{m}{L} \right\rfloor - n, \quad (3.2.11)$$

and the identity

$$mM - \left\lfloor \frac{mM}{L} \right\rfloor L = (nM) \oplus L, \quad (3.2.12)$$

where  $\lfloor u \rfloor$  denotes the integer less than or equal to  $u$  and  $(i) \oplus L$  denotes the value of  $i$  modulo  $L$ . Applying (3.2.10) and (3.2.11) (with  $M = 1$ ) to (3.2.9) yields

$$\begin{aligned} y(m) &= \sum_{n=-\infty}^{\infty} h\left(m - \left\lfloor \frac{m}{L} \right\rfloor L + nL\right) x\left(\left\lfloor \frac{m}{L} \right\rfloor - n\right) \\ &= \sum_{n=-\infty}^{\infty} h(nL + m \oplus L) x\left(\left\lfloor \frac{m}{L} \right\rfloor - n\right). \end{aligned} \quad (3.2.13)$$

Equation (3.2.13) expresses the output  $y(m)$  in terms of the input  $x(n)$  and the filter coefficients  $h(m)$  thus interpolation by integer factors of  $L$  giving

$$g_m(n) = h(nL + m \oplus L), \quad \text{for all } m \text{ and all } n \quad (3.2.14)$$

and it is seen that  $g_m(n)$  is periodic in  $m$  with period  $L$ . Therefore  $g_m(n)$  splits up the original low pass filter characteristic up into  $L$  smaller sub-filters, each of these filters are used to compute an interpolated output  $y(m)$  when receiving a new input sample  $x(n)$ . The periodicity property of  $g_m(n)$  will be used in Section 3.5 to implement an efficient structure for digital filtering.

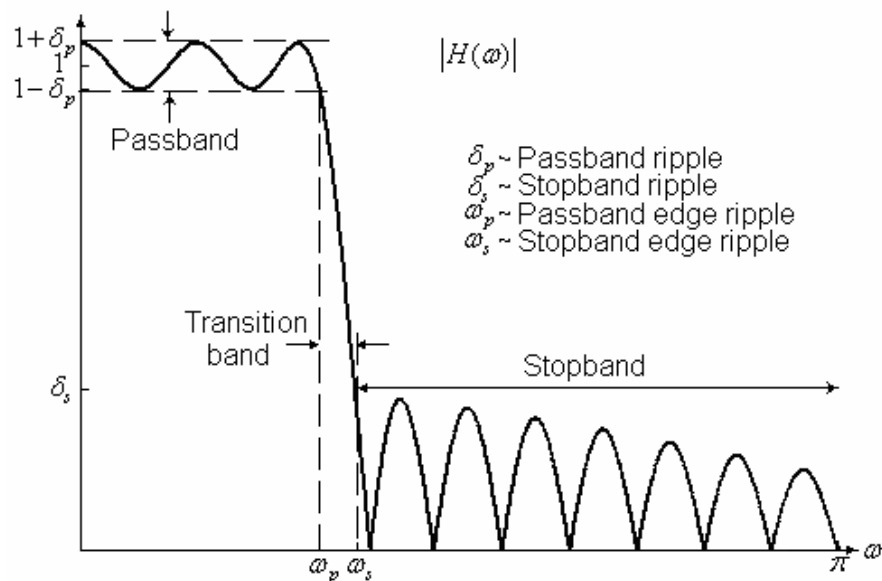
Increasing the sampling rate of a discrete signal involves padding zeros between consecutive samples according to the sampling rate, and then filtering the sampling rate expanded signal with an ideal digital lowpass filter. The impulse response of this filter can be split up into periodical sub filters each used to compute the interpolated output signal.

### 3.3 Digital Interpolation Filters

Section 3.2 concluded that an ideal low pass discrete filter is necessary to interpolate a digital signal. The question now arises if methods exist in which this interpolation process can be implemented practically in the digital domain without adding significant distortion to the interpolated output. Practical implementability implies finiteness, but unfortunately finiteness implies non-ideality.

In this section finite length discrete filters which are used in digital signal processing will be investigated. The goal of this will be to interpolate an input signal at an integer rate, and at equidistant intervals. In Section 3.4 the performance of a finite length discrete interpolation filter method described and chosen here will be used to design and meet constraints according to audio signal specifications. From this point on, a discrete finite filter will be referred to as a digital filter.

### 3.3.1 Low pass filter characteristic



**Figure 3.3.1: Filter magnitude response with specification parameters altered from [3].**

Figure 3.3.1 shows the frequency response of a practical digital filter, when comparing this characteristic with the ideal form given in Figure 3.1.3, various non-idealities or differences can be observed. Firstly it can be observed that the passband isn't completely flat but has a small amount of ripple. Secondly the stopband region also exhibits ripple. Thirdly the transition of the frequency response from passband to stopband is nonzero and is defined as the transition band or transition region of the digital filter. From [3] the following definitions are made.

The band-edge frequency  $\omega_p$  defines the edge of the passband, while the frequency  $\omega_s$  denotes the beginning of the stopband. Thus the width of the transition band is  $\omega_s - \omega_p$ . The width of the passband is usually called the bandwidth of the filter. For example, if the filter is lowpass with a passband edge frequency  $\omega_p$ , its bandwidth is  $\omega_p$ . If there is ripple in the passband of the filter, its value is denoted as  $\delta_p$ , and the magnitude  $|H(\omega)|$  varies between the limits  $1 \pm \delta_p$ . The ripple in the stopband of the filter is denoted as  $\delta_s$ .

Usually the passband and stopband ripple are related to decibels by the following expressions

$$PBR = 10 \log_{10} \frac{(1 + \delta_p)^2}{(1 - \delta_p)^2} \quad (3.2.14)$$

$$\approx 17.36(\delta_p) \quad \text{when } \delta_p \ll 1$$

and

$$SBR = 20 \log_{10}(\delta_2). \quad (3.2.15)$$

Where  $PBR$  and  $SBR$  represent the passband ripple and stopband ripple in decibel.

### 3.3.2 Phase characteristic

A desired characteristic of a digital filter is a linear phase response, and this characteristic will now be derived. It is shown in [3] that if a signal  $x(n)$  with bandwidth  $B$  passes through a digital filter with frequency response

$$H(\omega) = \begin{cases} Ce^{-j\omega n_0}, & \omega_1 < \omega < \omega_2 \\ 0, & \text{otherwise} \end{cases}, \quad (3.3.1)$$

where  $C$  and  $n_0$  are constants, the signal at the output of the filter has a spectrum

$$\begin{aligned} Y(\omega) &= X(\omega)H(\omega) \\ &= CX(\omega)e^{-j\omega n_0} \quad \omega_1 < \omega < \omega_2. \end{aligned} \quad (3.3.2)$$

By applying the scaling and time-shifting properties of the Fourier transform the time-domain output is obtained by

$$y(n) = Cx(n - n_0). \quad (3.3.3)$$

Equation (3.3.3) indicates that the output of the filter is simply a delayed and amplitude scaled version of the input signal. This pure delay is not considered as a distortion of the input signal. Therefore ideal digital filters have a linear phase characteristic within its passband, that is,

$$\Theta(\omega) = -\omega n_0. \quad (3.3.4)$$

The derivative of the phase with respect to frequency has the units of delay. It can therefore be defined that the signal delay as a function of frequency is given as

$$\tau_g(\omega) = -\frac{d\Theta(\omega)}{d\omega}. \quad (3.3.5)$$

$\tau_g(\omega)$  is usually called the envelope delay or the group delay of the filter. The expression  $\tau_g(\omega)$  is interpreted as the time delay that a signal component of frequency  $\omega$  undergoes as it passes from the input to the output of a system. Note that when  $\Theta(\omega)$  is linear,  $\tau(\omega) = n_0 = \text{constant}$ . In this case all frequency components of the input signal undergo the same time delay. The time delay  $n_0$  exists because of the result that half the impulse response of the filter needs to be shifted by half its length to gain the filter causality.

### 3.3.3 Digital filtering methods

Two methods of digital filtering exist which could be used to realise the filter derived in (3.2.13). These two are the finite impulse response (FIR) and the infinite impulse response (IIR) methods. Table 3.3.1 compares these two respective filtering methods characteristics.



<b>Finite Impulse Response (FIR)</b>	<b>Infinite Impulse Response (IIR)</b>
Linear phase possible	Not precise linear phase
Always stable	Can be unstable
Higher order filter needed for sharper cut-off transition band characteristics	Lower order needed for sharp cut-off transition band characteristics
Higher computational complexity	Lower computational complexity

**Table 3.3.1: Comparison between digital filtering methods.**

The two main characteristics that are of importance in the choice of filtering methods are magnitude and phase response. From Section 3.3.1 magnitude response distortion is inevitable in any practical filtering method. But if the magnitude response is designed such that the noise added to the input signal is less than the resolution of the input signal itself, magnitude response transparency is guaranteed. Magnitude transparency of the filter is therefore dependant on the input signal resolution, and to obtain transparency for an increasing signal resolution a larger filter length is required. Secondly phase linearity of a digital filter implies no distortion on the phase of the input and therefore results in phase transparency. According to these characteristics a discussion between the choice of FIR and IIR is now presented.

The FIR structure is inherently phase linear because it is easy to make the impulse response absolutely symmetrical. IIR structures are not capable of delivering exact linear phase within the passband of the filter. Table 3.3.1 shows that FIR filters are computationally more expensive than IIR filters, and that higher order FIR filters are required to obtain the same cut-off characteristics as IIR filters. Although FIR filters have higher computational overhead they still offer higher transparency because of their linear phase properties. It is because of this property that a FIR filter will be used in the interpolation process. In Section 3.5 it will be shown that an efficient filter structure reduces the computational expense when implementing FIR filters.

From this subsection we deduced that finite digital filters do not have ideal magnitude response characteristics, it was shown that finite digital filters could have linear phase which is not seen as a distortion in the filtering process. Two filter methods were presented and compared. Although FIR filters may have a larger filter length as apposed to IIR filters at a specific input signal resolution, FIR filters are on the whole more transparent because of there linear phase response characteristic.

In Section 3.4 the design of a FIR filter will be described to interpolate an audio input signal to a desired output sampling frequency.

### 3.4 Design of a Linear Phase Bandpass FIR Filter

Any processing done on high resolution digital audio requires a high transparency to ensure that non-audible distortion is added to its baseband frequency content. It has been established in Section 3.3 that FIR filters are the best choice when overall transparency is desired at the cost of greater computational complexity. If a specific FIR filter design guarantees phase linearity the only design specification that is left is the magnitude response of the filter. The magnitude response design of the filter should provide enough dynamic range in its passband and attenuation in its stopband to ensure that the resolution of the input signal is retained.

#### 3.4.1 Choice of a FIR filter design method

The most common linear phase FIR design methods to date are the

- Window design,
- Frequency sampling,
- Optimum equiripple linear phase filter and
- Minimum mean-square-error design methods.

From the above list the linear-phase equiripple filters are desirable because they have the smallest maximum deviation from the ideal filter when compared to methods listed above of the same order. Equiripple filters are ideally suited for applications in which a specific tolerance must be met. For example, if it is necessary to design a filter with a given minimum stopband attenuation or a given maximum passband ripple [23].

In the current application of interpolating high resolution digital audio signals, the control of the digital filter parameters (Section 3.3.1) to meet certain tolerances are essential. For this reason the optimum equiripple linear phase filter method is the preferred filter design method.

### 3.4.2 Optimum equiripple linear-phase FIR filters

The optimum equiripple linear-phase FIR filter design method is formulated as a Chebyshev approximation problem [3]. It is viewed as an optimum design criterion in the sense that the weighted approximation error between the ideal frequency response and the actual frequency response is spread evenly across the passband and evenly across the stopband of the filter, minimising the maximum error [3]. The solutions of the Chebyshev approximation problem are based on either a multiple exchange Remez algorithm, or a single exchange linear programming solution. These filter design solutions are readily available in software packages for example: Matlab®.

For the case of the low-pass characteristic of Figure 3.3.1 an empirical formula has been derived that relates the parameters of low pass FIR filters into an optimum equiripple solution.

The formula known as the Hermann-Rabiner-Chan's formula is expressed in terms of the digital filter length  $N$ , which is given by

$$N \cong \frac{D_{\infty}(\delta_p, \delta_s)}{(\omega_s - \omega_p)/(2\pi)} - f(\delta_p, \delta_s) \frac{(\omega_s - \omega_p)}{2\pi} + 1 \quad (3.4.1)$$

where

$$D_{\infty}(\delta_p, \delta_s) = [a_1(\log_{10} \delta_p)^2 + a_2 \log_{10} \delta_p + a_3] \log_{10} \delta_s + [a_4(\log_{10} \delta_p)^2 + a_5 \log_{10} \delta_p + a_6] \quad (3.4.2)$$

and

$$f(\delta_p, \delta_s) = 11.012 + 0.512(\log_{10} \delta_p - \log_{10} \delta_s), \quad (3.4.3)$$

for  $|\delta_s| \leq |\delta_p|$

lastly

$$\begin{aligned} a_1 &= 0.00539 \\ a_2 &= 0.07114 \\ a_3 &= -0.4761 \\ a_4 &= -0.00266 \\ a_5 &= -0.5941 \\ a_6 &= -0.4278 \end{aligned} \quad (3.4.4)$$

From (3.4.1) an estimate of the filter order can be calculated given a set of desired magnitude response specifications.

### 3.4.3 FIR filter specifications

It is needed to upsample a digital audio signal having an input sampling rate of 44.1 kHz. This data rate increase is necessary in the process of converting a PCM signal into a PWM signal. This modulation process will be covered in Chapter 4, it is only of importance now to know that the upsampling rate of  $L = 8$  is needed. From 3.2.7 it was derived that the gain of the filter should be  $G = L = 8$ .

Table 3.4.1 describes the specifications to upsample a digital input audio signal at a resolution of 24-bits to its new upsampling rate of  $F'$ :

Passband ( $\omega_p$ )	19 kHz
Stopband ( $\omega_s$ )	23 kHz
Passband ripple ( $\delta_p$ )	0.001 dB
Stopband ripple ( $\delta_s$ )	150 dB
Final sampling frequency ( $F'$ )	352.8 kHz

**Table 3.4.1: Interpolation filter specifications.**

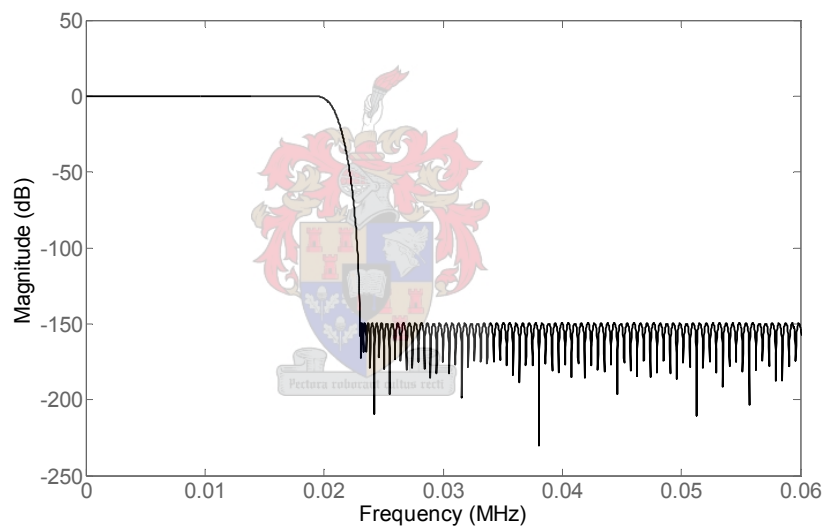
The transition band ( $\omega_s - \omega_p$ ) for this filter is 400 Hz which is relatively wide but necessary to reduce the order of the filter. An estimate of the digital filter coefficient length with the above specifications was calculated using 3.4.1, 3.4.2, 3.4.3 and 3.4.4 which resulted in a filter length of  $N = 631.42$ .

### 3.4.4 Digital filter design characteristics

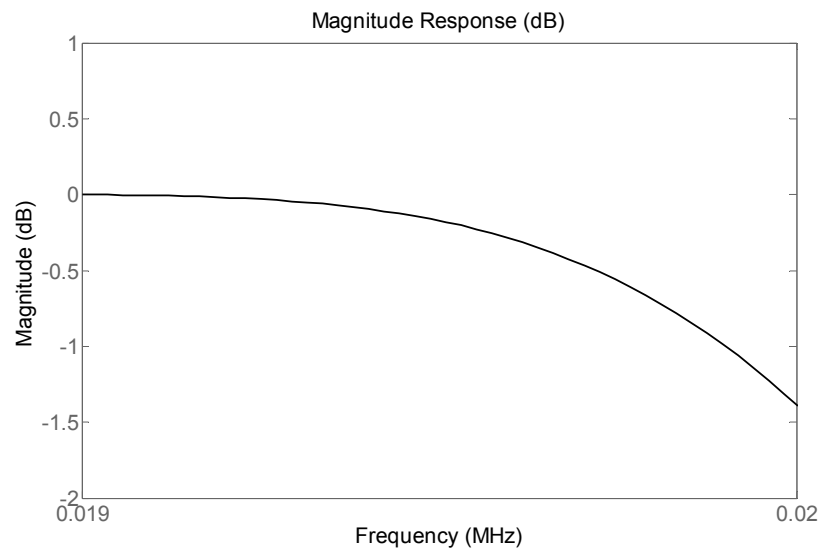
Matlab®'s fdatool filter toolbox was used to design the digital linear phase FIR filter with specifications given in Table 3.4.1. The filter length achieved fulfilling these specifications in Matlab® produced an equiripple FIR filter of coefficient length  $N = 632$ . This value confirms the estimate calculated by (3.4.1). The large filter length also agrees with the discussion that FIR filters do require more coefficients to

realise than IIR filters when given the same desired specifications. An IIR Chebyshev Type II filter was designed using the same specifications given in Table 3.4.1 and attained a filter order of  $N = 35$  but with a non-linear phase response.

Figure 3.4.1 shows the magnitude response of the FIR filter design. The filter's lowpass characteristic keeps the output's baseband undisturbed between 0 kHz and 19 kHz. From 19 kHz the digital audio input signal's frequency content starts to be attenuated and gradually increases attenuation until it reaches a maximum attenuation of -1.5 dB at 20 kHz as shown in Figure 3.4.2. Fortunately most human listeners cannot detect audio above 16 kHz therefore a small attenuation of 1.5 dB above 19 kHz will be unnoticeable. Above 23 kHz any frequency images are attenuated to -150 dB which implies a filter resolution of 24-bits.

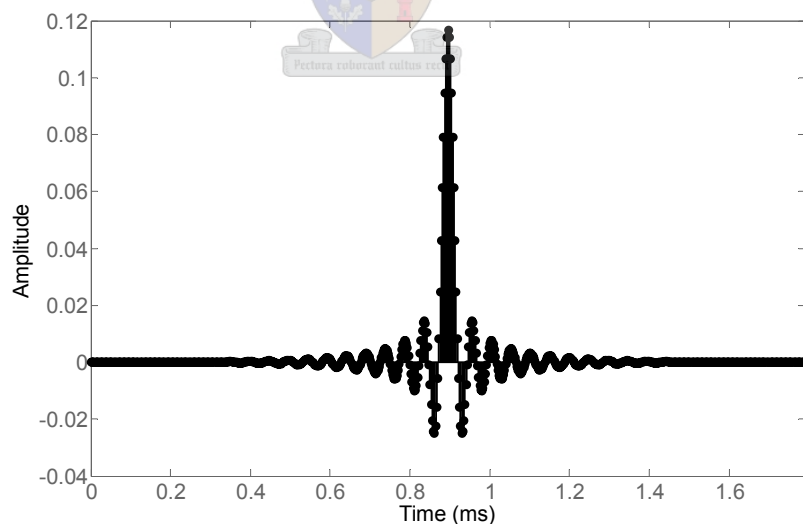


**Figure 3.4.1: Magnitude response of the equiripple FIR filter.**



**Figure 3.4.2: Attenuation of linear FIR filter between 19 kHz and 20 kHz.**

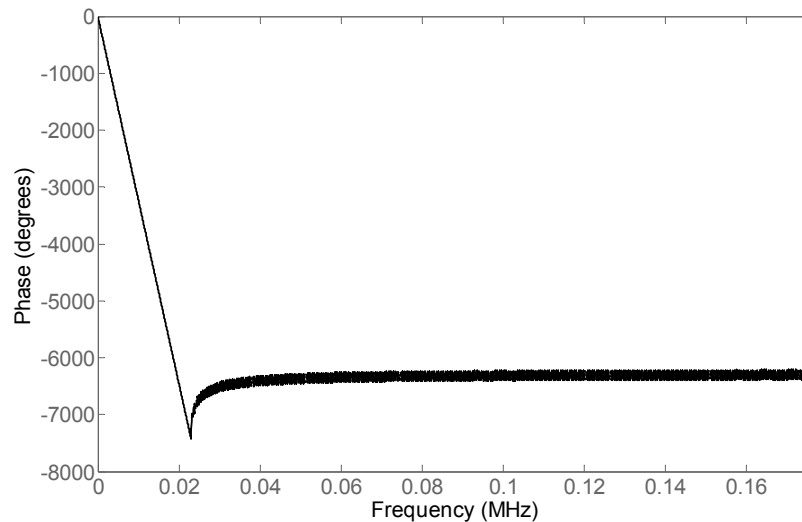
The filter impulse response  $h(n)$  of Figure 3.4.1 is given in Figure 3.4.3. The impulse response resembles the sinc function characteristic encountered in equation (3.1.22) except that  $h(n)$  now has a finite duration. The coefficients of  $h(n)$  are symmetric around its center index with no coefficients present on the negative time axis thus characterising  $h(n)$  as a causal lowpass response.



**Figure 3.4.3: Impulse response.**

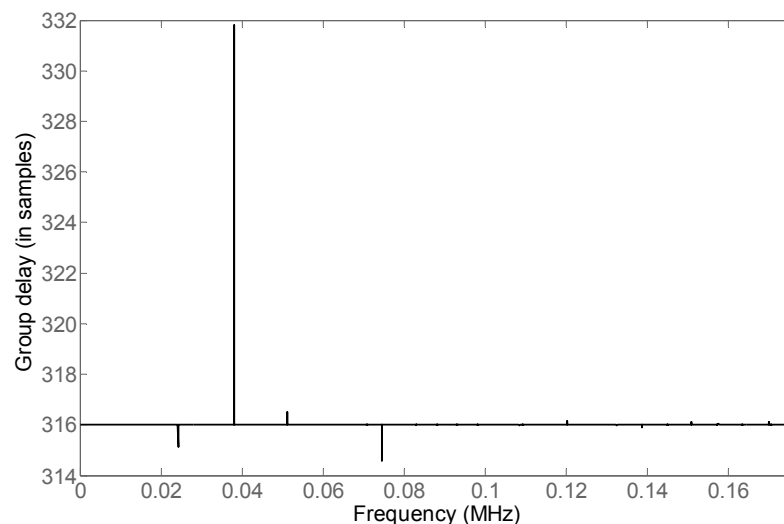
Figure 3.4.4 shows the exact linear phase characteristic within the passband of the filter. The gradient of the linear phase gives the group delay as a function of frequency which is given by  $\tau_g(\omega)$  and expressed in (3.3.5). After 20 kHz the phase

response becomes non-linear, but this is of no consequence, since the filter rejects any frequency content of the digital input signal within this band.



**Figure 3.4.4: Phase response.**

The group delay  $\tau_g(\omega)$  as a function of frequency is given in Figure 3.4.5. It is observed that the group delay remains constant within the audio baseband at  $n_0 = 316$  samples but deviates outside this band. From Section 3.3.2 this constant value implies that all frequencies within this baseband undergo the same delay and therefore no distortion is added to the frequency content of the filtered signal.



**Figure 3.4.5: Group delay.**

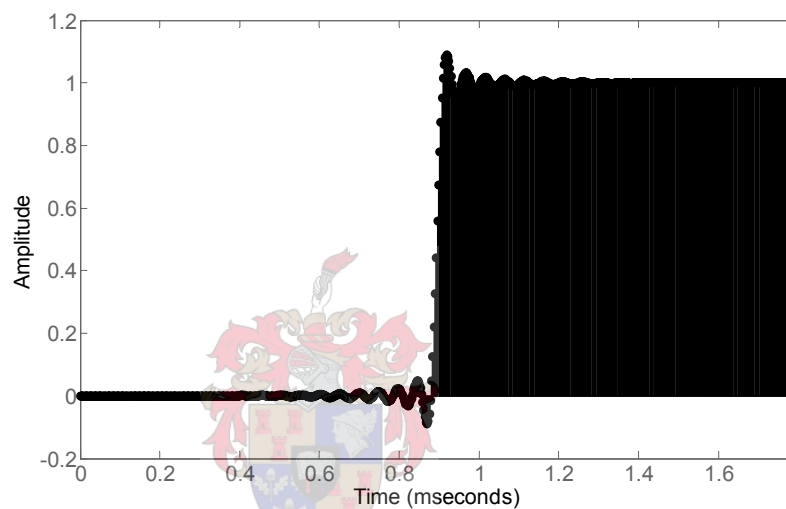
The constant group delay's effect in the time domain can be seen in the step response of the FIR digital filter shown in Figure 3.4.6. The group delay as a time

value is calculated by multiplying the constant sample delay  $n_0$  with the sampling

period  $T_s = \frac{1}{F_s} = \frac{1}{352.8e3} = 2.83447 \mu s$  which gives:

$$t_{delay} = n_0 T_s = 89,569 \text{ ms} . \quad (3.4.5)$$

This time delay value is the same as the centre time value of the impulse response given in Figure 3.4.3. Therefore the time delay of the FIR filter response is dependant on half the coefficient length of the filter.



**Figure 3.4.6: Step response.**

The transfer function of the digital FIR filter in the  $z$ -domain is given by

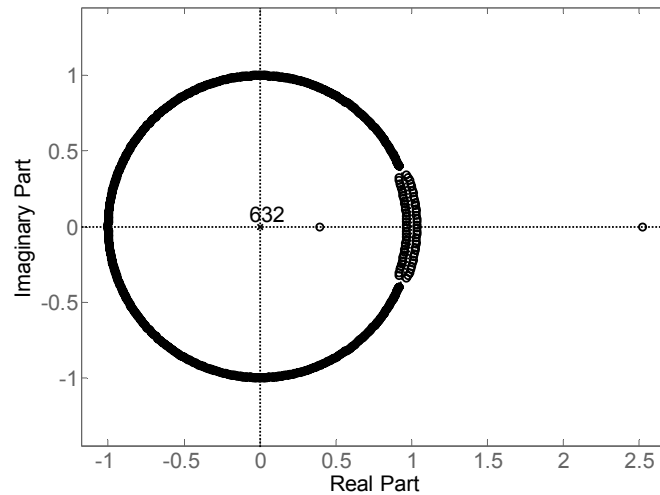
$$H(z) = \sum_{n=0}^{N-1} h(n)z^{-n} . \quad (3.4.6)$$

FIR digital filters are assured to always be stable, this constraint imposes that all poles lie within the unit circle. From the transfer function of (3.4.6) all poles lie at  $z = 0$  and for the current filter design Figure 3.4.7 confirms this, therefore implying stability. The zeros of the transfer function are therefore allowed to have any position without affecting the stability of the filter. The positions of the zeros though have an influence on the phase linearity of the filter. For the FIR filter to exhibit phase linearity, some zeros are constrained to be positioned outside the unit circle.

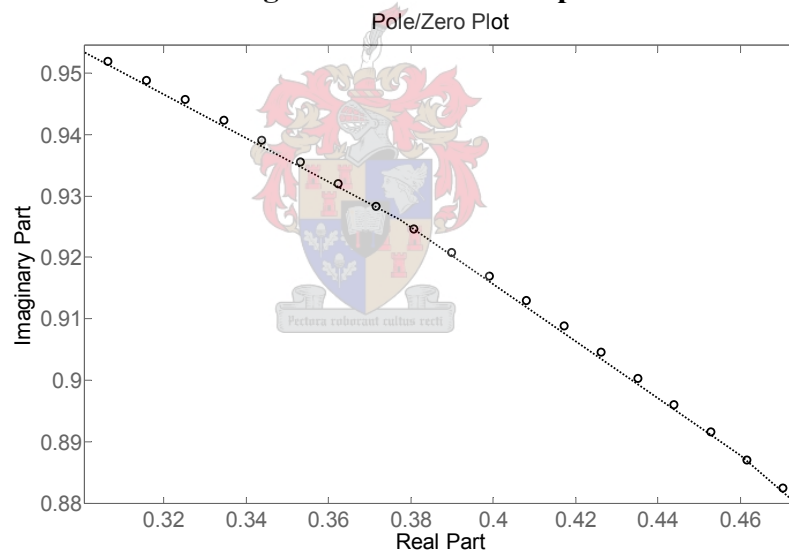
Figure 3.4.8 shows the zoomed version of the pole/zero plot, here it can be seen that some of the zeros which seem to lie on the unit circle in Figure 3.4.7 actually



do lie outside the unit circle confirming phase linearity again. The number of zeros outside the unit circle is equal to the sample group delay given by  $n_0$ .



**Figure 3.4.7: Pole/Zero plot.**



**Figure 3.4.8: Zoomed pole/zero plot.**

### 3.4.5 Summary

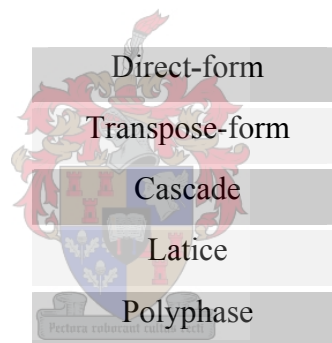
A complete design of an interpolation filter was presented in this section. The interpolation filter is necessary to upsample a 24-bit audio signal sampled at 44.1 kHz to a sampling rate of 352.8 kHz. It was decided to use an optimum equiripple linear phase FIR method to design an interpolating digital filter complying too a list of specifications. These specifications ensure that the resolution of the audio input signal is retained implying no audible distortions on the interpolated output. The Matlab®

filter toolbox was used to design the digital filter with the given specifications and its simulation results were investigated to confirm that it met the specified requirements.

Next an efficient structure will be presented whereby the designed digital FIR filter can be implemented with reduced computational complexity.

### 3.5 Polyphase FIR Structures for Integer Interpolators

The increased filter lengths which FIR methods have over IIR methods make them computationally more expensive. Different FIR filter structures exist which reduce the overall computational overhead when implemented. It is not of relevance here to investigate in detail these different structures, but rather to choose an efficient structure for the present application. The different FIR filtering structures that do exist are listed in Table 3.5.1 for completeness.



Direct-form
Transpose-form
Cascade
Lattice
Polyphase

**Table 3.5.1: FIR filter structures.**

It is needed to implement the low pass filter designed in Section 3.4 efficiently within a FPGA. For this implementation it was decided to choose the polyphase filtering structure and in the following sub-sections it will be described how this structure works and why it was chosen for this application.

Firstly the theory of the polyphase filter will be reviewed as described in [22] to show how this filtering technique is efficient; thereafter some important properties of the structure will be given which defines polyphase filters. Thirdly the filter design in Section 3.4 will be converted to the polyphase structure using Matlab®.

### 3.5.1 Polyphase FIR structure

In Section 3.2 it was derived that the general form for the input-to-output time-domain relationship for a 1 to  $L$  interpolator from [22] was

$$y(m) = \sum_{n=-\infty}^{\infty} g_m(n) x \left( \left\lfloor \frac{m}{L} \right\rfloor - n \right), \quad (3.5.1)$$

where

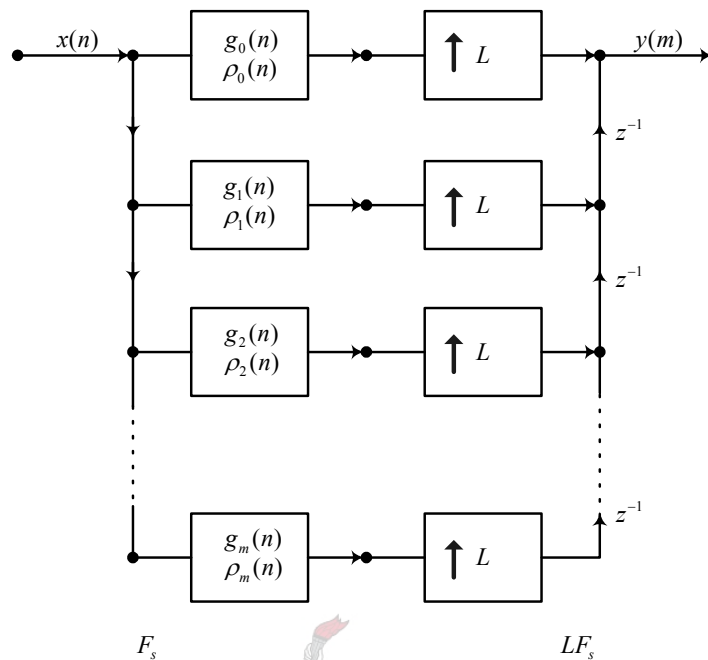
$$g_m(n) = h(nL + m \oplus L), \quad \text{for all } m \text{ and } n \quad (3.5.2)$$

is a periodically time-varying filter with period  $L$ . Thus to generate each output sample  $y(m)$ ,  $m = 0, 1, 2, \dots, L-1$ , a different set of coefficients  $g_m(n)$  are used. After  $L$  outputs are generated, the coefficient pattern repeats; thus  $y(L)$  is generated using the same set of coefficients  $g_0(n)$  as  $y(0)$ ,  $y(L+1)$  uses the same set of coefficients  $g_1(n)$  as  $y(1)$ , etc.

Similarly the term  $\lfloor m/L \rfloor$  in (3.5.1) increases by for every  $L$  samples of  $y(m)$ . Thus for output samples  $y(L), y(L+1), \dots, y(2L-1)$  the coefficients  $g_m(n)$  are multiplied by samples  $x(1-n)$ . In general, for output samples  $y(rL), y(rL+1), \dots, y(rL+L-1)$  the coefficients  $g_m(n)$  are multiplied by samples  $x(r-n)$ . Thus it is observed that  $x(n)$  in (3.5.1) is updated at the low sampling rate  $F_s$ , whereas  $y(m)$  is evaluated at the high sampling rate  $LF$ .

An implementation of the 1 to  $L$  interpolator based on the computation of (3.5.1) is shown in Figure 3.5.1. The way in which this structure works is as follows. The partitioned subsets,  $g_0(n), g_1(n), \dots, g_{L-1}(n)$ , of  $h(m)$  can be identified with  $L$  separate linear, time invariant filters which operate at a low sampling rate  $F_s$ . To make this subtle notational distinction between the time-varying coefficients and the time-invariant filters, the time-invariant filters will be referred to as  $\rho_0(n), \rho_1(n), \dots, \rho_{L-1}(n)$ . Thus

$$\rho_p(n) = g_p(n), \quad \text{for } p = 0, 1, 2, \dots, L-1 \text{ and all } n \quad (3.5.3)$$



**Figure 3.5.1: Polyphase structures for a 1 to  $L$  interpolator.**

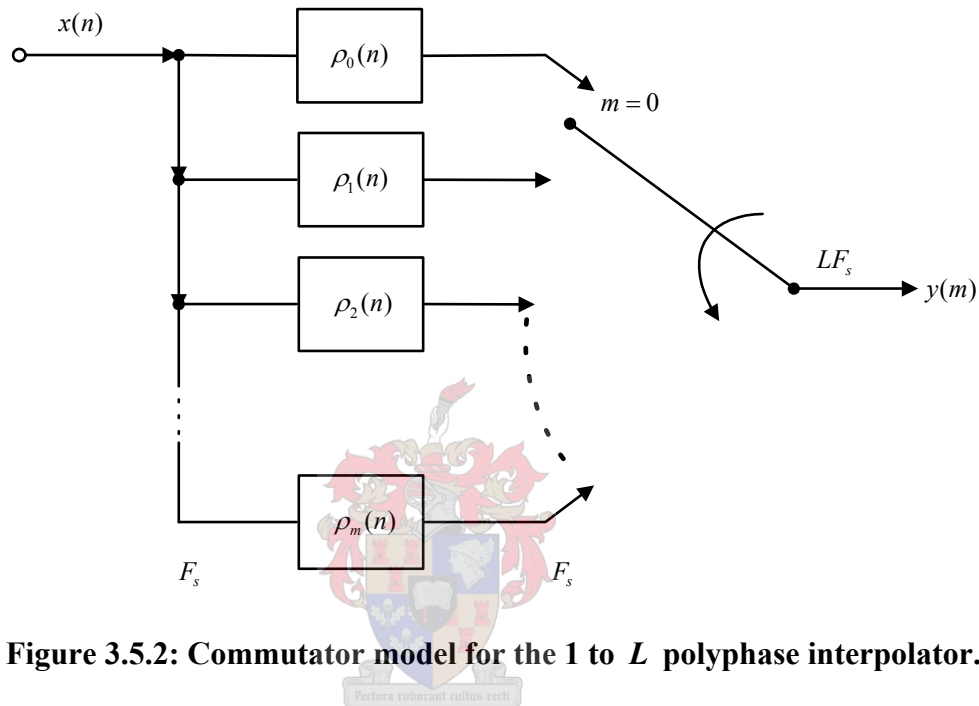
These filters  $\rho_p(n)$  are referred to as the polyphase filters. By combining (3.5.3) with (3.5.2) it is apparent that

$$\rho_p(n) = h(nL + p), \quad \text{for } p = 0, 1, 2, \dots, L-1 \text{ and all } n \quad (3.5.4)$$

For each new input sample  $x(n)$  there are  $L$  output samples (see Figure 14). The output from the upper path  $y_0(m)$  has non-zero values for  $m = nL, n = 0, \pm 1, \pm 2, \dots$ , which correspond to system outputs  $y(nL), n = 0, \pm 1, \dots$ . The output from the next path  $y_1(m)$  is nonzero for  $m = nL + 1, n = 0, \pm 1, \pm 2, \dots$  because of the delay of one sample at the high sampling rate. Thus  $y_1(m)$  corresponds to the interpolation output samples  $y(nL + 1), n = 0, \pm 1, \dots$ . In general the output of the  $p$ th path,  $y_p(m)$  corresponds to the interpolation output samples  $y(nL + p), n = 0, \pm 1, \dots$ . Thus each input sample  $x(n)$  of the  $L$  branches of the polyphase network (Figure 3.5.1) contributes one nonzero output which corresponds to one of the  $L$  outputs of the network.

From a practical point of view it is often convenient to implement the polyphase structures in terms of a commutator model. By careful examination of

Figure 3.5.2 it can be seen that the outputs of each of the polyphase branches contributes samples of  $y(m)$  for different time slots. Thus the 1 to  $L$  sampling rate expander and delays can be replaced by a commutator as shown in Figure 3.5.2. The commutator rotates in a counter clockwise direction starting with the zeroth-polyphase branch at time  $m = 0$ .



**Figure 3.5.2: Commutator model for the 1 to  $L$  polyphase interpolator.**

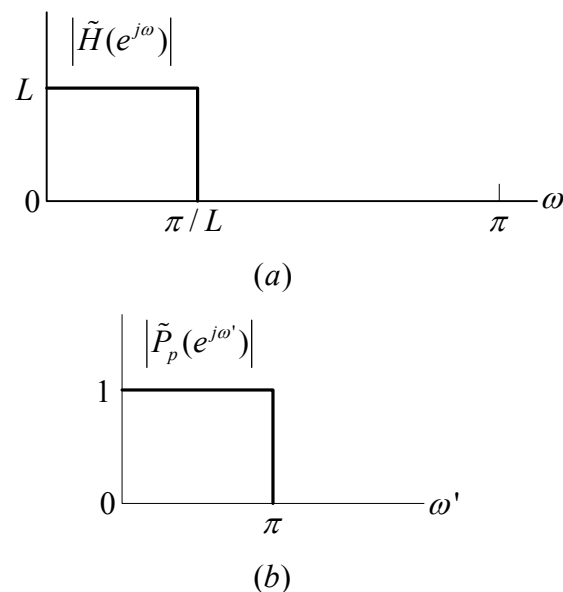
From the interpolation equation in (3.5.1) it has been shown theoretically and diagrammatically that the polyphase FIR filtering structure is efficient because the filtering process is done at the low sampling rate. There is therefore no need to first append  $L$  zeros between consecutive samples as was the case when the interpolation process was described in Section 3.2.1. The polyphase filter structure rather divides the interpolation low pass filter into sub filters (polyphase filters) which directly filters the input signal  $x(n)$  to the interpolated, upsampled output signal  $y(m)$ . It is now necessary to investigate the properties of these individual polyphase filters to gain further insight into the operation of the polyphase structure.

### 3.5.2 Properties of polyphase filters

The individual polyphase filters have two interesting properties as a consequence of the fact that the impulse responses  $\rho_p(n)$  correspond to decimated versions of the impulse response of the prototype filter  $h(m)$  given by (3.5.4).

First, different phase shifts are associated with the different filters  $\rho_p(n)$ . These delays though are compensated for by the delays ( $z^{-1}$ ) which occur at the high sampling rate  $LF_s$  in the network (Figure 3.5.1). The fact that different phases are associated with different paths of the network is, of course, the reason for the term polyphase network.

A second property of the polyphase filters is shown in Figure 3.5.3. The frequency response of the prototype filter  $h(m)$  approximates the ideal low-pass characteristic  $\tilde{H}(e^{j\omega})$  shown in Figure 3.5.3(a). Since the polyphase filters  $\rho_p(n)$  are decimated versions of  $h(m)$  (decimated by  $L$ ) the frequency response  $0 \leq \omega \leq \pi/L$  of  $\tilde{H}(e^{j\omega})$  scales to the range  $0 \leq \omega' \leq \pi$  for  $\tilde{P}_p(e^{j\omega'})$  as seen in Figure 3.5.3 where  $\tilde{P}_p(e^{j\omega'})$  is the ideal characteristic that the polyphase filter  $\rho_p(n)$  approximates. Thus the polyphase filters approximate all-pass functions and each value of  $\rho, \rho = 0, 1, 2, \dots, L-1$ , corresponds to a different phase shift.

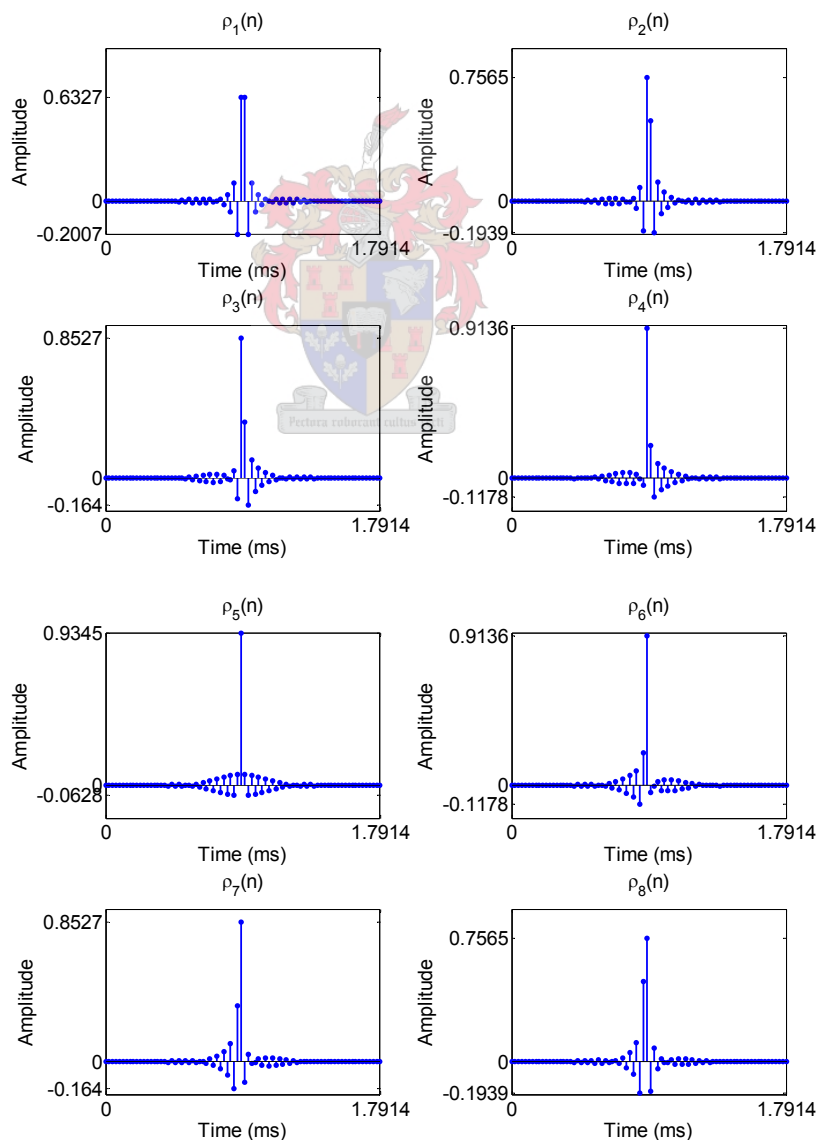


**Figure 3.5.3: Ideal frequency response of the polyphase networks.**

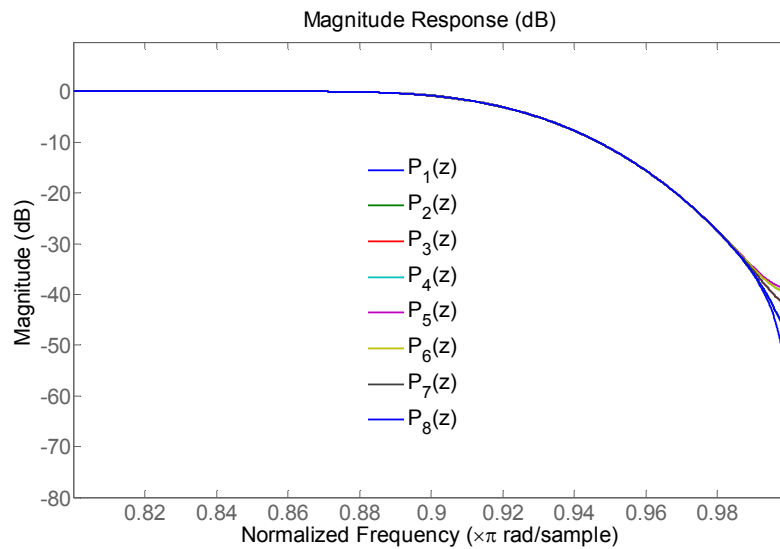
### 3.5.3 Conversion to the polyphase structure

The low pass equiripple FIR filter designed in Section 3.4 will now be converted to a polyphase structure using the multirate filter function in Matlab®. The properties of the individual polyphase filters attained from the conversion should therefore confirm Section 3.5.2.

The designed filter  $h(m)$  in Section 3.4 has a length of  $N = 632$ , the individual polyphase filters will each therefore have lengths of  $N/L = 79$  where  $L = 8$  is the upsampling rate. Figure 3.5.4 and Figure 3.5.5 shows the impulse responses and the magnitude responses of the  $L$  polyphase filters  $\rho_p(n)$  respectively.



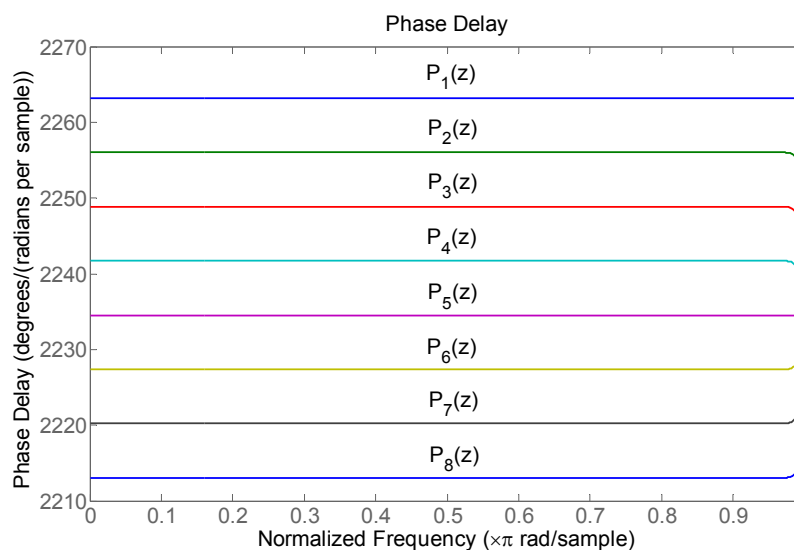
**Figure 3.5.4: Impulse responses of the eight polyphase filters.**



**Figure 3.5.5: Magnitude response of polyphase filters.**

Figure 3.5.5 shows the magnitude response of the different polyphase filters, they only allow frequency content below  $f = 1$  or  $\omega = \pi$  to pass. Since the polyphase filtering is done at the input sampling rate, the normalized frequency of  $f = 1$  represents  $F = 44.1$  kHz.

Figure 3.5.6 shows the phase shifts of each polyphase filter. The different phase delays of the corresponding polyphase filters are a result of the different sample delays each polyphase filter has from the centre of the prototype filter's symmetry.



**Figure 3.5.6: Phase delay of respective polyphase filters.**

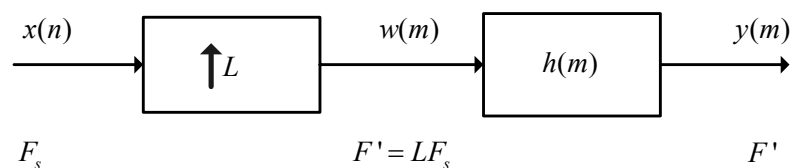


### 3.5.4 Summary

It has been shown that the polyphase filtering structure is an efficient method to interpolate an input signal at a specific sampling rate to a higher sampling rate. This is achieved by subdividing the low pass interpolation filter into polyphase filters according to the upsampling rate  $L$ . For each new sample input the  $L$  polyphase filters produce an output which constitutes the interpolated output. The lowpass filter designed in Section 3.4 was converted to the polyphase filtering structure to efficiently interpolate audio input signals of 24-bit resolution.

## 3.6 Example of the Interpolation Process

In this the final section of the chapter a complete simulation of the interpolation process will be done in Matlab®. The polyphase structure described in Section 3.5 has no need to zero-pad between adjacent samples. But here the zero padding phase of the theoretical interpolation process will be included in the simulation to see the effect it has on the frequency content of the input signal as it passes through the interpolator given in Figure 3.6.1. Also the goal of this section is to confirm that the interpolation filter designed and converted to the polyphase structure meets the specification of interpolating 24-bit audio data.



**Figure 3.6.1: Total interpolation process.**

### 3.6.1 Cosine input signal

The input  $x(n)$  which will be used in the simulation of the interpolation system in Figure 3.6.1 is a real cosine signal given by the expression

$$x(n) = A \cos(2\pi fn), \quad (3.6.1)$$

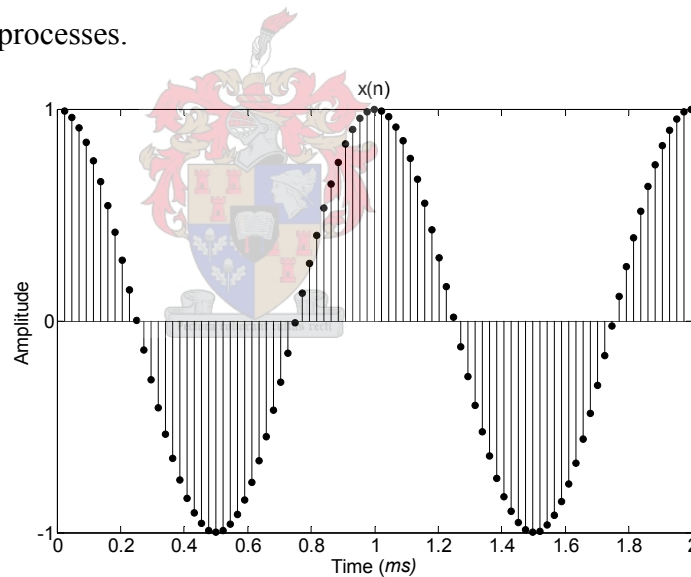
where

$$f = \frac{F}{F_s}, \quad (3.6.2)$$

with

$$\begin{aligned} F &= 1000 \text{ Hz} \\ F_s &= 44.1 \text{ kHz}. \end{aligned} \quad (3.6.3)$$

The time characteristic of the cosine input is shown in Figure 3.6.2. The frequency content of this cosine input is estimated using the MTM PSD estimation method which is shown in Figure 3.6.3 (the Matlab® code used to implement the simulations presented next are given in Appendix A). This spectral estimation method is used to attain a high frequency resolution of the signal spectrum and is normally used with random processes.



**Figure 3.6.2: 1 kHz sinusoidal input signal  $x(n)$ .**

The 1 kHz frequency components of the cosine have been clearly detected by the MTM estimate. From the normalized frequency axis these frequency components lie at  $f = 0.0227$  and  $f = 0.9773$  respectively. The former being the positive frequency component of 1000 Hz calculated using the relation between the variables  $F$  and  $f$  in (3.1.9). The latter frequency component is the frequency image of the cosine function as a result of the periodicity characteristic of digital signals.

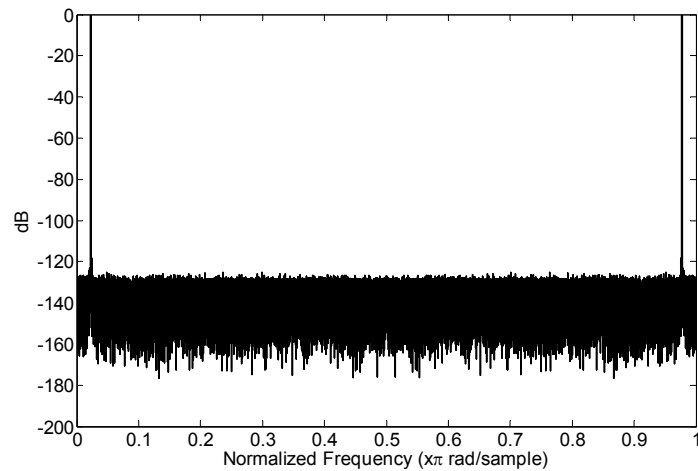


Figure 3.6.3: MTM PSD estimate of  $x(n)$ .

### 3.6.2 Sample rate expanded signal

With the characteristics of the input signal now defined it can be applied to the interpolation process. The signal is upsampled by zero padding  $L-1=7$  zeros between adjacent  $x(n)$  samples. The sample rate expander therefore upsample's the input to an increased rate of  $L=8$ . Figure 3.6.4 shows this output  $w(m)$  which was described mathematically in (3.2.2). Figure 3.6.5 shows a zoomed view of the padded zeros.

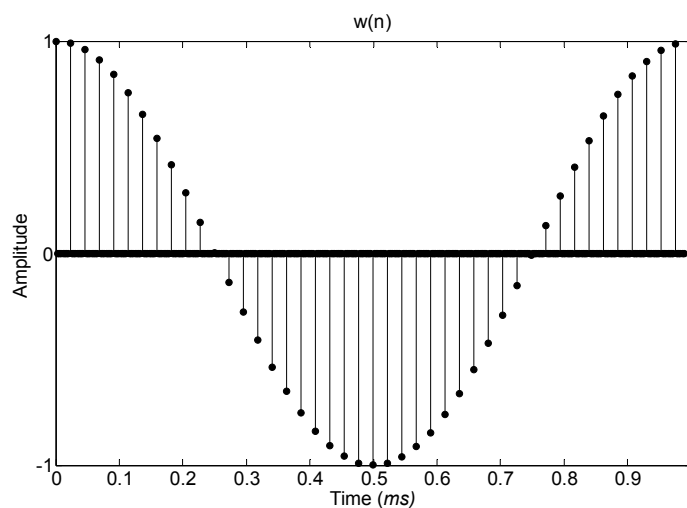


Figure 3.6.4: Sample rate expanded signal  $w(n)$ .

The spectral estimate of  $w(m)$  using the MTM PSD estimate is shown in Figure 3.6.6. From this figure the images of the baseband frequency component

appears at harmonic intervals of the original sampling frequency (44.1 kHz). Table 3.6.1 shows these harmonic intervals in different frequency formats according to the relationship of (3.1.9) at the new sampling rate of  $F' = 352.8$  kHz. These intervals concur with normalized frequency axis of Figure 3.6.6, therefore validating the sample rate expander simulation.

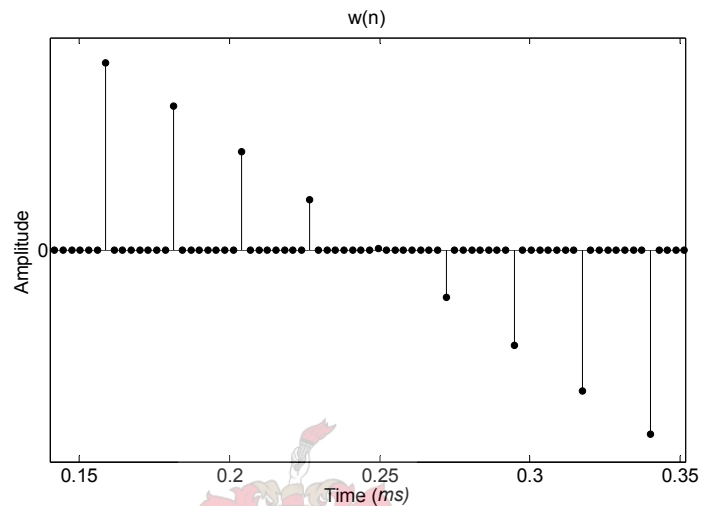


Figure 3.6.5: Zoomed view of  $w(n)$ .

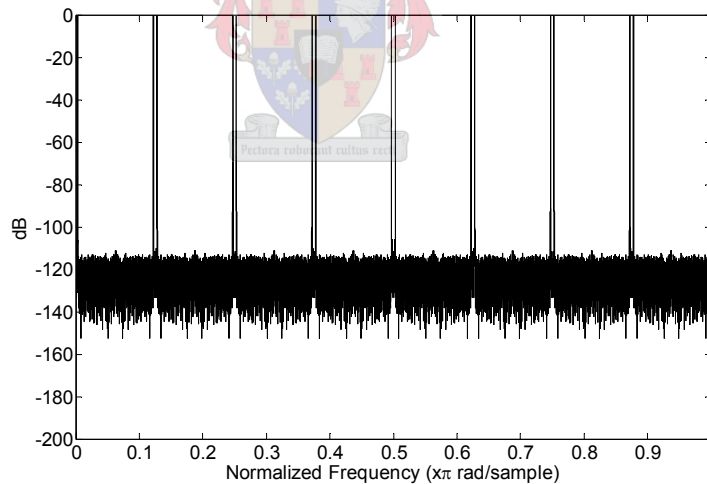


Figure 3.6.6: MTM PSD estimate of  $w(n)$ .

	Frequency (Hz)	Radians per Sample (rad/s)	Normalized frequency ( $f$ )
1.	44100	0.3927	0.125
2.	88200	0.7854	0.25
3.	132300	1.1781	0.375
4.	176400	1.5708	0.5
5.	220500	1.9635	0.625
6.	264600	2.3562	0.75
7.	308700	2.7489	0.875
8.	352800	3.1416	1

**Table 3.6.1: Harmonic intervals at which baseband frequencies are centered.**

### 3.6.3 Polyphase filtering

Physical zero-padding doesn't occur when the polyphase filtering structure is used for interpolation. The input signal is directly filtered. This doesn't mean that no baseband images are present in the frequency content, since filtering is still necessary in the polyphase filtering structure. It will be shown through simulation that images of the baseband do exist but are attenuated after polyphase filtering.

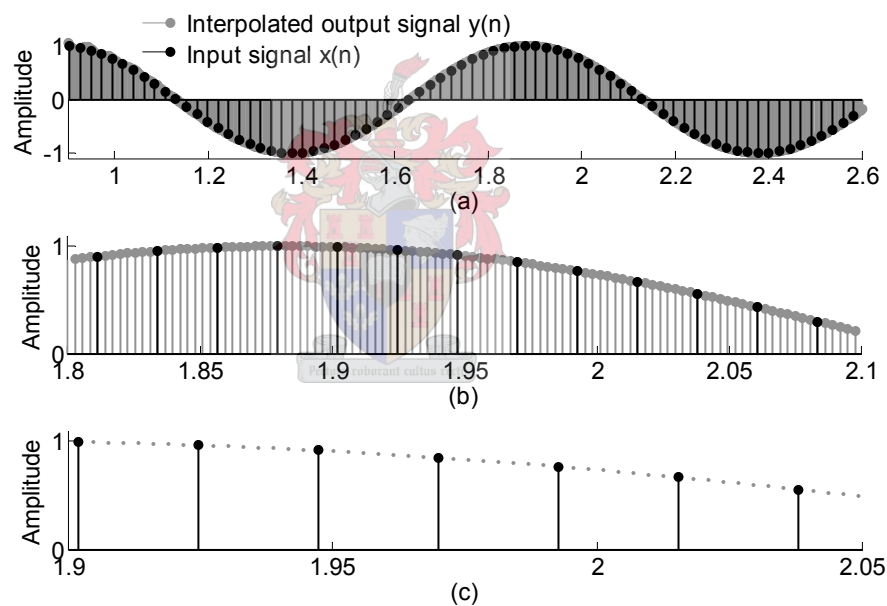
Table 3.6.2 shows the specifications of the interpolation filter designed in Section 3.4 to filter away the baseband images.

Passband ( $\omega_p$ )	19 kHz
Stopband ( $\omega_s$ )	23 kHz
Passband ripple ( $\delta_p$ )	0.001 dB
Stopband ripple ( $\delta_s$ )	150 dB
Final sampling frequency ( $F'$ )	352.8 kHz

**Table 3.6.2: Specifications of low-pass digital filter.**

The passband ( $\omega_p$ ) of the filter preserves all the audio content beneath the 19 kHz band and prevents any frequency content above 23 kHz to be passed to the output.

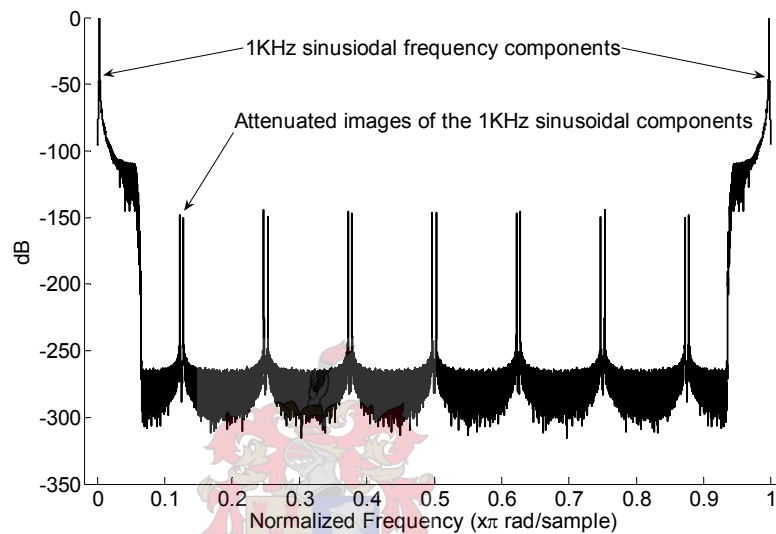
The impulse response  $h(n)$  of the above filter was converted to the polyphase structure in Section 3.5. Figure 3.6.7 shows the interpolated output signal from this filtering process and also zoomed views to see how the input signal has been interpolated. It is observed that the interpolated signal follows the original signal exactly except for the delay and transient response at the start of  $y(n)$ . The transient response is a result of the finiteness of the filter and placing of the poles and zeros in the complex plane. The delay is a result of the linear phase of the FIR filter which was calculated in (3.4.9) to be  $\pm 90$  ms.



**Figure 3.6.7: Input  $x(n)$  and interpolated output  $y(m)$ .**

Figure 3.6.8 shows the spectral estimate of  $y(n)$ . Here it can be seen how the filter has preserved the cosine frequency components and have attenuated the frequency images below -150 dB outside the passband of the filter. As mentioned previously this output was generated by the polyphase filtering structure, but the output spectra still shows attenuated images even though no physical zero-padding was applied.

With the original baseband in tact and images attenuated sufficiently, the original signal is restored at a new increased sampling rate. Sufficient attenuation here implies that unwanted frequency components within the spectra fall below the resolution of the input signal which is 24-bits. This implies that unwanted frequency content should fall below -144 dB, which is the case when looking again at Figure 3.6.8.



**Figure 3.6.8: MTM PSD of output  $y(n)$ .**

### 3.6.4 Summary

It has been confirmed through simulation and spectral estimates that the polyphase filtering structure using the low pass equiripple FIR method successfully interpolates a discrete input signal. A cosine input signal at a sampling rate of  $F_s = 44.1$  kHz and resolution of 24-bits was interpolated to a higher sampling rate of  $F' = 352.8$  kHz. The interpolated output retains a resolution of 24-bits as a result of the high dynamic range the interpolation filter exhibits, which is -150 dB.

The next chapter will now use this upsampled or interpolated PCM signal to produce digital PWM.

# Chapter 4 - Pulse Width Modulation

## 4.0 Introduction

Chapter 3 thoroughly described the process of increasing the sampling rate ( $F_s$ ) of a discrete signal to an upsampled rate ( $F'$ ). This rate corresponds to the switching frequency ( $F_c$ ) of the PWM process which is illustrated in Figure 4.1. The goal of this chapter is to show how the modulation process transforms the now upsampled PCM (Pulse Code Modulation) signal having 24-bit resolution to a pulse width signal of equal resolution within the digital domain.

Firstly, PWM and its variant schemes are defined and then compared according to their spectral content. Secondly a digital PWM technique will be introduced using an appropriate modulation scheme for implementation. The mathematical calculations necessary to implement this digital PWM scheme will be derived. Two methods, known as the Newton's method and the Bisection method, will be compared. From this comparison, one method will be chosen to generate 24-bit digital PWM within VHDL firmware.

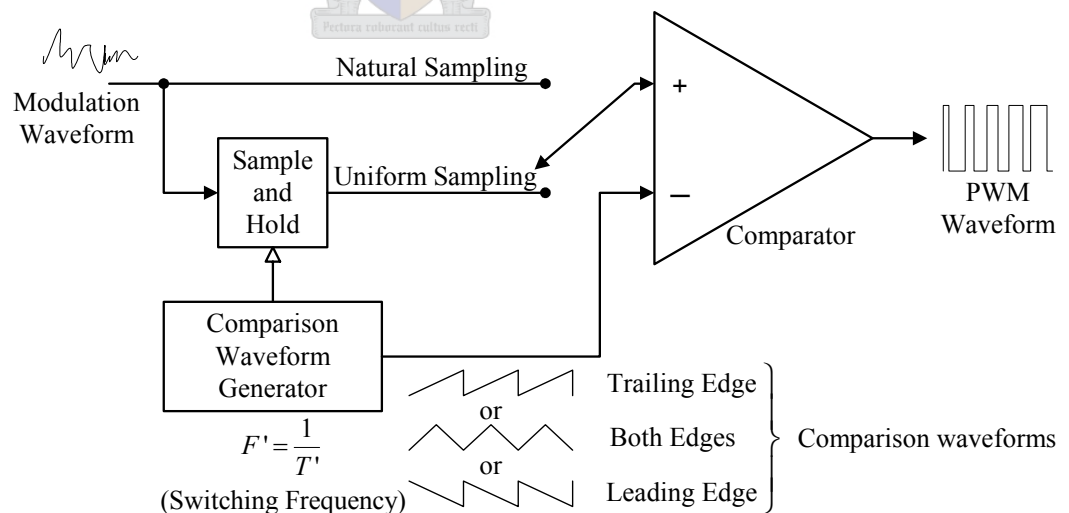


Figure 4.1: Two-level pulse-width modulator adapted from [17].



## 4.1 Pulse Width Modulation Schemes

*“PWM is the modulation of a pulse carrier in which the value of each instantaneous sample of a modulating wave is caused to vary the duration of a particular pulse. The modulating wave may vary the time of occurrence of the leading edge, trailing edge, or both edges of the carrier pulse” [5].*

PWM is therefore a process involving two types of waveforms. The first waveform is the input signal (modulation waveform) which contains the desired information. The second waveform is known as the carrier waveform. The modulation process is performed by comparing these two waveforms to produce a pulse varying output signal (see Figure 4.1). This modulated signal will be called the PWM output from this point forward. When the modulating waveform is greater than the comparison waveform the PWM output is high, but if the waveform is less than the comparison waveform, the PWM output is low. Since only two values (high or low) occur at the output the process is called a two-level pulse width modulator.

The characteristics of the PWM output signal are therefore dependant on both the modulation waveform and the carrier waveform (comparison waveform). Within the next sections it will be determined which of the two waveforms affect which characteristics of PWM output. PWM output waveforms generated from the different input waveforms will afterwards be compared according to their spectral content.

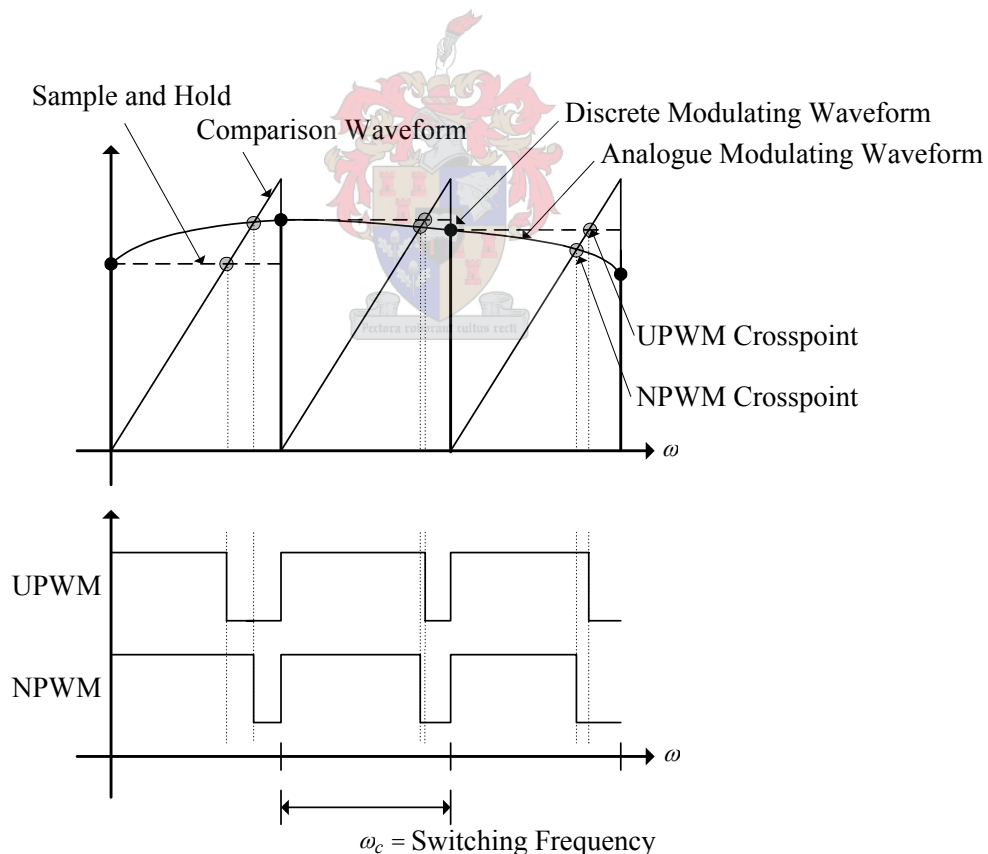
### 4.1.1 Natural and uniform PWM

As noted in the previous section the characteristics of the PWM output signal are dependant on both the comparison waveform and input information bearing signal characteristics. Attention will firstly be focussed on how the PWM characteristics are influenced by the input modulation waveform.

Whenever an analogue input signal is used for modulation, natural pulse width modulation (NPWM) results. This is illustrated in Figure 4.1 when the comparator is connected to the top node. The sample instants (at the crosspoints) which determine the pulse variation of the PWM waveform in this type of modulation is signal dependant and thus non-uniformly spaced.

When the modulating waveform is discrete and sampled at fixed intervals these fixed time instant amplitudes are used for comparison with the carrier wave. This is called uniform pulse width modulation (UPWM). Figure 4.1 illustrates UPWM when the comparator is connected to the bottom node, and Figure 4.2 shows the difference between the NPWM and UPWM process.

UPWM can be generated from an analogue input waveform when a sample and hold function is applied to it as illustrated in Figure 4.1. The sampling then occurs at linearly spaced intervals ( $T_c$ ). This sample and hold rate is the same rate at which the carrier waveform is modulated and is known as the switching frequency ( $F_c$ ) of the modulator. It is important to note that when a discrete signal is applied to the modulator input, its sampling frequency equal the same rate as the switching frequency of the modulator. This ensures that a correct PWM output is generated as illustrated in Figure 4.2.



**Figure 4.2: Difference between UPWM and NPWM.**

Further characteristic changes occur within the PWM output when different comparison waveforms are used in the modulation process. These various waveforms have an effect on how the edge or edges of the output pulse are modulated. Both UPWM and NPWM will be considered when evaluating the different comparison waveforms. For each comparison waveform used a different PWM scheme results.

These PWM schemes are compared to each other according to their spectral content. Before these different schemes can be presented, however, the method of computing the spectral content of a PWM process must be stated.

### 4.1.2 Harmonic components of PWM

The spectra of PWM outputs consist of different harmonic components. These different harmonic components can be computed mathematically when using a sinusoidal modulating input waveform, also known as a tonal input. The harmonic components generated by the PWM schemes can then be determined using the well known analytical method developed by Bennet and Black. This method is based on a double Fourier series expansion in two variables.

The detailed derivation of the Fourier series expansion falls outside the scope of this thesis and will not be done here. From [4], the result can be stated:

$$f(t) = T_0 + T_1 + T_2 + T_3 \quad (4.1)$$

where

$$\begin{aligned} T_0 &= \frac{A_{00}}{2} \\ T_1 &= \sum_{n=1}^{\infty} [A_{0n} \cos(n[\omega_0 t + \theta_0]) + B_{0n} \sin(n[\omega_0 t + \theta_0])] \\ T_2 &= \sum_{m=1}^{\infty} [A_{m0} \cos(m[\omega_c t + \theta_c]) + B_{m0} \sin(m[\omega_c t + \theta_c])] \\ T_3 &= \sum_{m=1}^{\infty} \sum_{\substack{n=-\infty \\ (n \neq 0)}}^{\infty} \left[ \begin{aligned} &A_{mn} \cos(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]) \\ &+ B_{mn} \sin(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]) \end{aligned} \right] \end{aligned} \quad (4.2)$$

and where

$$A_{mn} = \frac{1}{2\pi^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} f(x, y) \cos(mx + ny) dx dy \quad (4.3)$$

$$B_{mn} = \frac{1}{2\pi^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} f(x, y) \sin(mx + ny) dx dy, \quad (4.4)$$

Alternatively, in complex form

$$C_{mn} = A_{mn} + jB_{mn} = \frac{1}{2\pi^2} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} f(x, y) e^{j(mx+ny)} dx dy, \quad (4.5)$$

also

$$\begin{aligned} x(t) &= \omega_c t + \theta_c \text{ and } y(t) = \omega_0 t + \theta_0, \text{ where} \\ \omega_c &= \frac{2\pi}{T_c} = \text{carrier angular frequency} \end{aligned} \quad (4.6)$$

with

$T_c$  = carrier interval

$\theta_c$  = arbitrary phase offset angle for carrier waveform (4.7)

$\omega_0 = 2\pi / T_0$  = fundamental (sinusoid) angular frequency,  $\omega_0 < \omega_c$ ,

and lastly

$$\begin{aligned} T_0 &= \text{period of fundamental waveform} \\ \theta_0 &= \text{arbitrary phase offset angle for fundamental waveform.} \end{aligned} \quad (4.8)$$

The variables  $m$  and  $n$  in (4.2) represent the carrier index and baseband index respectively. Together  $m$  and  $n$  define the frequency of each harmonic component of the PWM output spectra as  $(m\omega_c + n\omega_0)$  [4]. The magnitudes of the harmonic components defined in equation (4.2) are the  $A_{mn}$  and  $B_{mn}$  coefficients, which must be evaluated for particular values of  $m$  and  $n$  for each PWM scheme to be considered [4].

The expression  $f(x, y)$  represents the pulse width amplitude value at a specific time  $t$ . The variable  $y(t)$  represents the time information of the sinusoid (modulating wave), whereas the variable  $x(t)$  represents the time information of the switching frequency. Combining these parameters to form  $f(x, y)$  implies the forming of a surface consisting of periodical cells. This enables the use of two-dimensional Fourier analysis to calculate the PWM signal output. For a clearer description of  $f(x, y)$  the reader is referred to [4] or [5].

The first term of (4.2),  $T_0$ , corresponds to the DC offset of the PWM waveform. The second term in (4.2),  $T_1$ , defines the output fundamental low-frequency component (sinusoidal component) and its baseband harmonics. These baseband harmonics should preferably be eliminated by the modulation process,

except for the fundamental frequency component. The third summation term in (4.2),  $T_2$ , represents the carrier wave harmonics.

These are relatively high-frequency components, since the lowest frequency term represented is the modulating carrier frequency component. The last double summation term,  $T_3$ , is the ensemble of all possible frequencies formed by taking the sum and difference between the carrier or comparison waveform harmonics and the reference waveform and its associated baseband harmonics. These combinations are generally referred to as sideband harmonics, and exist as groups around the carrier harmonic frequencies [4].

Expressing the harmonic content of a PWM waveform mathematically, enables the comparison of various PWM schemes. The different PWM schemes are a result of the usage of different comparison waveforms. These waveforms are illustrated in Figure 4.3. Trailing edge modulation is the result of a positive gradient saw-tooth comparison waveform; leading edge modulation is the result of a negative gradient saw-tooth comparison waveform. Double edge modulation is the result of a triangular comparison waveform.

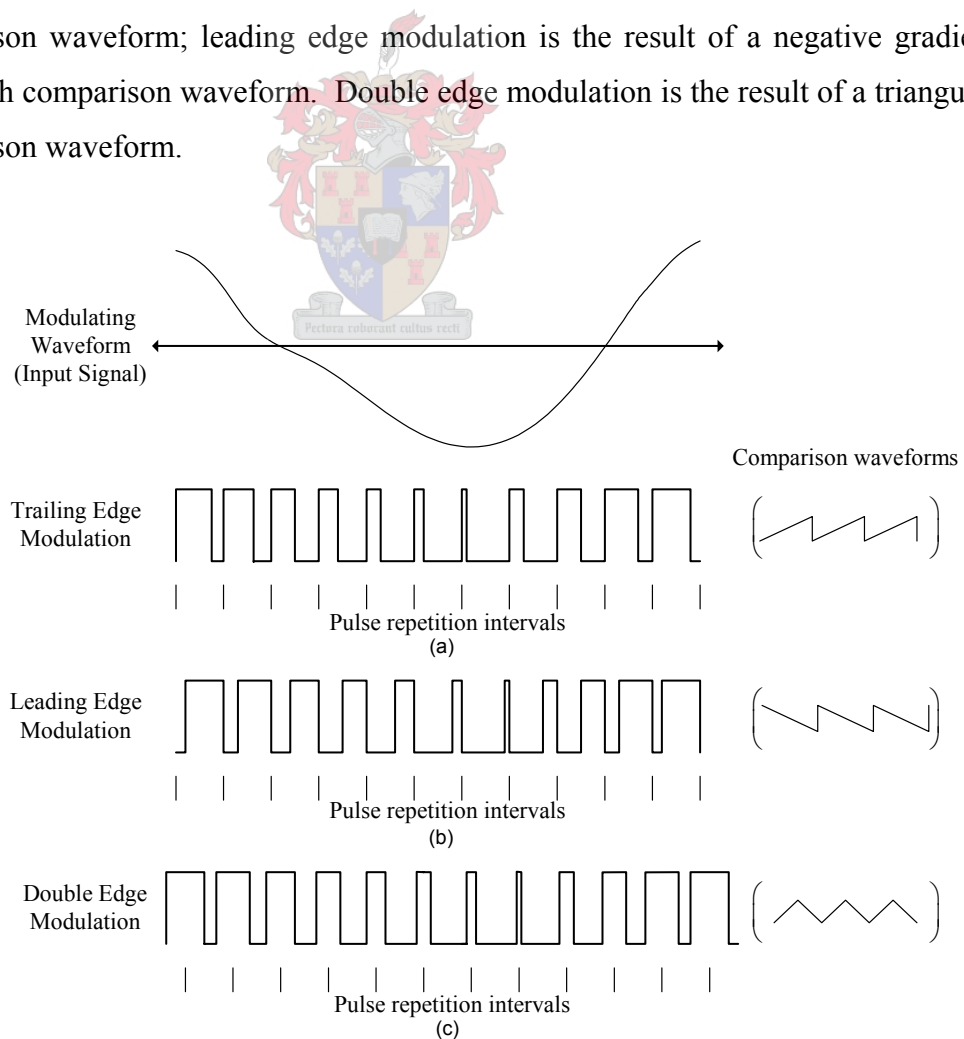


Figure 4.3: PWM Schemes altered from [17]

### 4.1.3 Trailing edge naturally sampled modulation

An analogue modulating waveform is used here to produce the PWM output signal. Thus this is an example of naturally sampled modulation. The carrier waveform is a positive gradient saw-tooth signal. Only the trailing edge of the output pulse of this modulation scheme is varied to form the PWM output as illustrated in Figure 4.3(a). Equation (4.9) gives the Fourier series representation of the PWM output. Its different terms represents the different harmonic components present within its spectrum.

$$\begin{aligned}
 v_1(t) = & V_{dc} + V_{dc}M \cos(\omega_0 t + \theta_0) \\
 & + \frac{2V_{dc}}{\pi} \sum_{m=1}^{\infty} \frac{1}{m} [\cos(m\pi) - J_0(m\pi M)] \sin(m[\omega_c t + \theta_c]) \\
 & + \frac{2V_{dc}}{\pi} \sum_{m=1}^{\infty} \sum_{\substack{n=-\infty \\ (n \neq 0)}}^{n=\infty} \frac{1}{m} J_n(m\pi M) \left[ \begin{array}{l} \sin\left(n \frac{\pi}{2}\right) \cos(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]) \\ -\cos\left(n \frac{\pi}{2}\right) \sin(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]) \end{array} \right] \quad (4.9)
 \end{aligned}$$

Where  $J_n$  denotes a Bessel function of the first kind in (4.9). The first term of (4.9),  $V_{dc}$  represents the DC-voltage component of the PWM output. In the second term of (4.9) the expression  $M \cos(\omega_0 t + \theta_0)$  represents the modulating input waveform at a frequency  $\omega_0$ . The parameter  $M$  is known as the modulation index and falls within the normalized range of  $0 < M < 1$ . If  $M \geq 1$ , over-modulation occurs this causes the PWM output to saturate. Only one frequency component is therefore represented by the second term which is known as the fundamental component exhibiting an amplitude of  $V_{dc}M$  and frequency  $\omega_0$ .

No baseband harmonics of this fundamental component are present within the second term which is desirable. The third term represents the presence of carrier wave harmonics at  $m$  integer multiples. The fourth term represents the presence of sideband harmonics situated around the multiples of the carrier harmonics.

Figure 4.4 illustrates a Matlab® simulation (with code given in Appendix C1) of the various harmonic components present within a trailing edge NPWM output using a sinusoidal input waveform of 1 kHz, and a modulation index of  $M = 0.95$ .

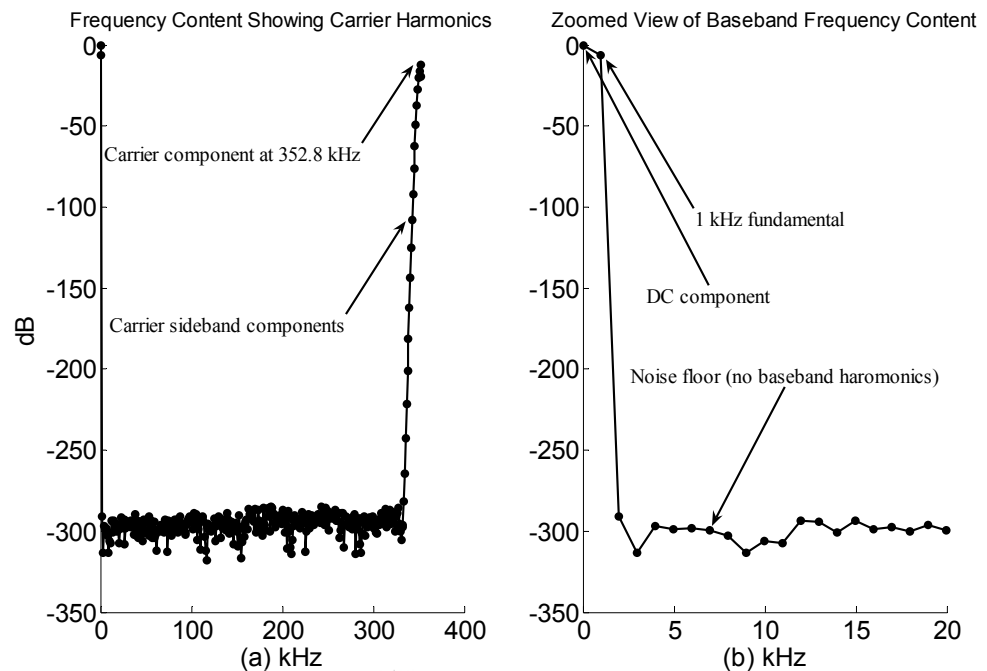


Figure 4.4: Trailing edge NPWM spectrum.

#### 4.1.4 Trailing edge uniformly sampled modulation

A discrete modulating waveform input, sampled at fixed intervals is used here to produce trailing edge UPWM. Equation 4.10 shows the spectral content of the PWM output. From the second term in (4.10) it is evident that baseband harmonics are present. These harmonics are situated at multiples of the fundamental frequency around the fundamental component. Baseband harmonics are a consequence of the regular sampling process and occur for any fixed sampled PWM strategy [4]. The roll-off of these harmonic components is dependant on the carrier ratio (ratio between modulation waveform frequency and switching frequency).

The rest of the terms within (4.10) are similar to (4.9) except for a slight shift in sideband energy between the lower and higher sideband harmonics [4]. Figure 4.5 shows the results of a Matlab® simulation using trailing edge UPWM. The same modulating input waveform was used as in the trailing edge NPWM simulation, except that this input was sampled at a fixed frequency equalling the switching frequency  $F_c$  the Matlab® code is given in Appendix C2.

$$\begin{aligned}
 v_2(t) = & V_{dc} + \frac{2V_{dc}}{\pi} \sum_{m=1}^{\infty} \frac{J_n\left(n \frac{\omega_0}{\omega_c} \pi M\right)}{\left(n \frac{\omega_0}{\omega_c}\right)} \left[ \begin{array}{l} \sin\left(n \frac{\pi}{2}\right) \cos(n[\omega_0 t + \theta_0]) \\ -\cos\left(n \frac{\pi}{2}\right) \sin(n[\omega_0 t + \theta_0]) \end{array} \right] \\
 & + \frac{2V_{dc}}{\pi} \sum_{m=1}^{\infty} \frac{1}{m} [\cos(m\pi) - J_0(m\pi M)] \sin(m[\omega_0 t + \theta_0]) \\
 & + \frac{2V_{dc}}{\pi} \sum_{m=1}^{\infty} \sum_{\substack{n=-\infty \\ (n \neq 0)}}^{\infty} \frac{J_n\left(\left[m+n \frac{\omega_0}{\omega_c}\right] \pi M\right)}{\left[m+n \frac{\omega_0}{\omega_c}\right]} \left[ \begin{array}{l} \sin\left(n \frac{\pi}{2}\right) \cos(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]) \\ -\cos\left(n \frac{\pi}{2}\right) \sin(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]) \end{array} \right]
 \end{aligned} \tag{4.10}$$

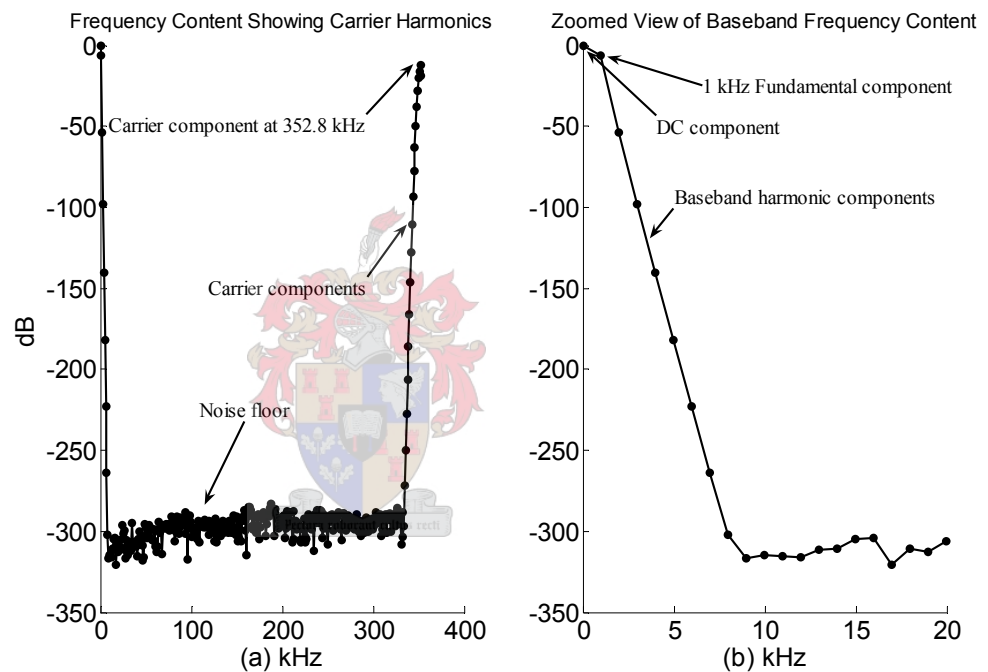


Figure 4.5: Trailing Edge UPWM spectrum.

### 4.1.5 Leading edge naturally and uniformly sampled modulation

PWM output signals with trailing edges modulated become pulses with leading edges modulated when the time scale is reversed as illustrated in Figure 4.3(b). This is because the leading edge saw-tooth comparison waveform has a negative gradient rather than the positive gradient that was the case with trailing edge modulation. Therefore, to obtain the series expression for the leading edge pulses, it is only necessary to put a negative sign in front of  $t$  in the expressions of the preceding two sections.



For leading edge naturally sampled modulation the series expansion is given by

$$v_3(t) = v_1(-t). \quad (4.11)$$

Similarly, for leading edge uniformly sampled modulation the series is given by

$$v_4(t) = v_2(-t). \quad (4.12)$$

The sign change of the time variable  $t$  has no effect on the magnitude of the frequency content and only influences its phase. Thus arguments made for trailing edge modulation schemes also hold for leading edge modulation schemes.

#### 4.1.6 Double edge naturally sampled modulation

PWM outputs with both edges modulated may be considered as a combination of two pulse trains: One with leading edges modulated, and the other with trailing edges modulated. When leading and trailing edge modulation occur in the proper time relationship toward one another, and are added, double edge modulation results. That is,

$$v_5(t) = v_1(t) + v_3(t) = v_1(t) + v_1(-t). \quad (4.13)$$

Through the combination of trailing edge and leading edge modulation of equation (4.13) the following series is obtained in terms of its harmonic components,

$$\begin{aligned} v_5(t) &= V_{dc} + V_{dc} M \cos(\omega_0 t + \theta_0) \\ &+ \frac{4V_{dc}}{\pi} \sum_{m=1}^{\infty} \frac{1}{m} J_o \left( m \frac{\pi}{2} M \right) \sin \left( m \frac{\pi}{2} \right) \cos(m[\omega_c t + \theta_c]) \\ &+ \frac{4V_{dc}}{\pi} \sum_{m=1}^{\infty} \sum_{\substack{n=-\infty \\ (n \neq 0)}}^{\infty} \frac{1}{m} J_n \left( m \frac{\pi}{2} M \right) \sin \left( [m+n] \frac{\pi}{2} \right) \cos(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]). \end{aligned} \quad (4.14)$$

The second term in equation (4.14) is identical to that of equation (4.9) showing that no harmonics are present within the baseband. The significant feature of double edge naturally sampled PWM is that the odd harmonic sideband components around odd multiples of the carrier fundamental, and even harmonic sideband

components around even multiples of the carrier fundamental, are completely eliminated by the  $\sin[(m+n)\pi/2]$  expression in equation (4.14), [4].

#### 4.1.7 Double edge uniformly sampled modulation

Similarly as with double edge naturally sampled modulation, the double edge uniform sampled modulation Fourier series is made by the addition of trailing and leading edge modulation in the correct relation.

Therefore,

$$v_6(t) = v_2(t) + v_4(t) = v_2(t) + v_2(-t). \quad (4.15)$$

Two variants of double edge uniformly sampled modulation exist. These are known as symmetrical and asymmetrical uniformly sampled modulation. The former modulation type's Fourier series representation is given by

$$\begin{aligned} v_{6(sym)}(t) = & V_{dc} + \frac{4V_{dc}}{\pi} \sum_{m=1}^{\infty} \frac{J_n \left( n \frac{\omega_0}{\omega_c} \frac{\pi}{2} M \right)}{\left( n \frac{\omega_0}{\omega_c} \right)} \sin \left( n \left[ 1 + \frac{\omega_0}{\omega_c} \right] \frac{\pi}{2} \right) \cos(n[\omega_0 t + \theta_0]) \\ & + \frac{4V_{dc}}{\pi} \sum_{m=1}^{\infty} \frac{1}{m} J_0 \left( m \frac{\pi}{2} M \right) \sin \left( m \frac{\pi}{2} \right) \cos(m[\omega_c t + \theta_c]) \\ & + \frac{4V_{dc}}{\pi} \sum_{m=1}^{\infty} \sum_{\substack{n=-\infty \\ (n \neq 0)}}^{n=\infty} \frac{J_n \left( \left[ m + n \frac{\omega_0}{\omega_c} \right] \frac{\pi}{2} M \right)}{\left[ m + n \frac{\omega_0}{\omega_c} \right]} \left[ \sin \left( \left[ m + n \frac{\omega_0}{\omega_c} + n \right] \frac{\pi}{2} \right) \right. \\ & \left. \times \cos(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]) \right], \end{aligned} \quad (4.16)$$

whereas the latter is given by

$$\begin{aligned}
 v_{6(asy)}(t) = & V_{dc} + \frac{4V_{dc}}{\pi} \sum_{m=1}^{\infty} \frac{J_n \left( n \frac{\omega_0}{\omega_c} \frac{\pi}{2} M \right)}{\left( n \frac{\omega_0}{\omega_c} \right)} \sin \left( n \frac{\pi}{2} \right) \cos(n[\omega_0 t + \theta_0]) \\
 & + \frac{4V_{dc}}{\pi} \sum_{m=1}^{\infty} \frac{1}{m} J_0 \left( m \frac{\pi}{2} M \right) \sin \left( m \frac{\pi}{2} \right) \cos(m[\omega_c t + \theta_c]) \\
 & + \frac{4V_{dc}}{\pi} \sum_{m=1}^{\infty} \sum_{\substack{n=-\infty \\ (n \neq 0)}}^{\infty} \frac{J_n \left( \left[ m + n \frac{\omega_0}{\omega_c} \right] \frac{\pi}{2} M \right)}{\left[ m + n \frac{\omega_0}{\omega_c} \right]} \left[ \sin \left( \left[ m + n \right] \frac{\pi}{2} \right) \right. \\
 & \left. \times \cos(m[\omega_c t + \theta_c] + n[\omega_0 t + \theta_0]) \right].
 \end{aligned} \tag{4.17}$$

The symmetrical uniform sampled modulation spectra of equation (4.16) gives a considerable reduction in the magnitude of the odd sideband harmonics around the odd carrier multiples and the even sideband harmonics around the even carrier multiples. The cancellation is not complete, however. The unwanted baseband harmonics in equation (4.16) still exist in this modulation scheme. They do have a much quicker roll-off when compared to the previous trailing or leading edge uniform modulation schemes.

The asymmetrical modulation variant has the odd harmonic sideband components around odd multiples of the carrier fundamental and even harmonic sideband components around even multiples of the carrier fundamental eliminated by the  $\sin[(m+n)\pi/2]$  in equation (4.16). Some but not all baseband harmonics are also cancelled due to the  $\sin(n\pi/2)$  expression in the second term of (4.16).

#### 4.1.8 Conclusion of PWM schemes studied

A brief overview of the most common PWM schemes was given, Fourier series expressions was used to gain insight into the differences in the harmonic content of the schemes studied. These schemes are divided into two main categories namely NPWM and UPWM schemes. NPWM schemes use analogue modulation waveforms, whereas UPWM schemes use discrete modulation waveforms. Both these categories each vary according to the type of carrier waveform used for modulation.

It was found that trailing and leading edge NPWM exhibited no baseband harmonics except for the fundamental frequency component of the modulation wave.

In contrast trailing and leading edge UPWM exhibited baseband harmonics which was considered as a form of baseband distortion.

Double edge NPWM sees the cancellation of certain sideband harmonic components centered on carrier frequency components. Also no harmonic components are present within the baseband. Double edge UPWM is divided into symmetrical and asymmetrical modulation. The former sees a partial attenuation of certain side-band harmonic components, and the latter sees a total cancellation of the same sideband harmonic components. Baseband harmonics are still present in the symmetrical modulation scheme but have a much faster roll off when compared to single edge UPWM. Asymmetric double edge UPWM has some baseband harmonics cancelled while those that remain exhibit the same roll-off as symmetrical double edge UPWM. It is concluded that the NPWM schemes generally perform better than UPWM schemes when considering harmonic distortion within the baseband. The double sided NPWM scheme has the least amount of harmonic distortion when considering the whole spectral space.

In the next section, a suitable PWM scheme is chosen to modulate a 24-bit discrete audio signal. If any baseband distortion is added to the audio data due to the modulation process it may become audible to the human ear. This is inevitable if a UPWM scheme is used as described above. NPWM schemes cannot be considered since they use analogue modulation waveforms whereas the input data and modulation process need to be discrete. The next section therefore investigates a discrete PWM process in which NPWM spectra properties are attained.

## 4.2 Pseudo-Natural Pulse Width Modulation

The previous section briefly evaluated the most common schemes of PWM in terms of their harmonic content. It was concluded that NPWM schemes do not generate baseband harmonic components. UPWM methods however do generate baseband harmonics. This is an undesirable characteristic, especially if it is necessary to modulate digital audio data for audio amplification purposes.

Digital audio amplifiers convert digital audio input data to digital PWM signals which act as gating signals for the power switching stage. UPWM is therefore imposed on Class-D amplifiers since digital data is used as the modulating input signal to the pulse width modulator.

The baseband harmonic components which are generated by UPWM are then audible to the human listener since they are present below 20 kHz. UPWM in Class-D audio amplifiers cannot be considered if high fidelity audio is desired.

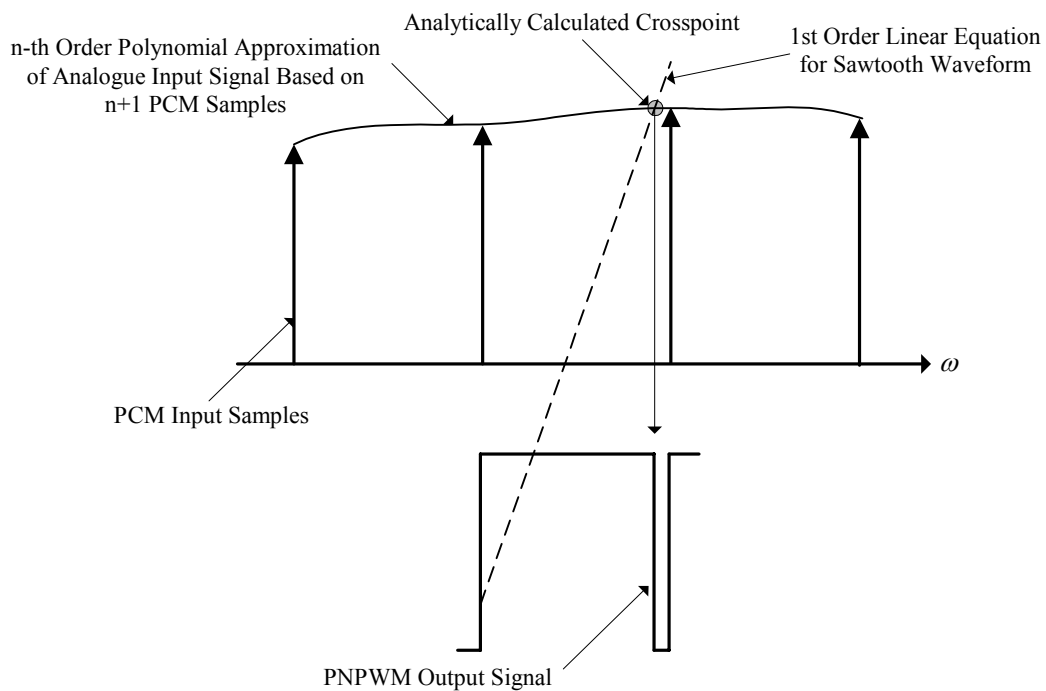
However, a technique known as pseudo-natural pulse width modulation (PNPWM) was introduced in [17] which mimics NPWM even though a PCM (discrete) modulating signal is applied at the input of a pulse width modulator.

The rest of this section will concern itself with this modulation scheme. It will describe exactly what is meant by PNPWM, and will evaluate how implementations of PNPWM can be used to generate high fidelity PWM signals from PCM input signals.

#### 4.2.1 What is PNPWM?

PNPWM is a digital PWM scheme where uniform spaced samples are used to estimate the crosspoint which the analogue waveform would have made with the comparison waveform in NPWM. The modulation process and pulse width is digital, which implies that the resolution of the pulse width is dependant on the clock speed performance of the hardware used for implementation.

The cross point is estimated from the uniformly spaced samples by fitting an  $n$ th-degree polynomial through these points. An expression is then derived for the particular comparison waveform within the pulse repetition interval. Both functions are then used to calculate the root of the cross point which corresponds to the pulse width within that particular switching interval. This is illustrated in Figure 4.6.



**Figure 4.6: Calculation of PNPWM output signal adapted from [15].**

#### 4.2.2 What PNPWM scheme should be used?

With the new PNPWM technique introduced from the previous section, a comparison waveform needs to be chosen to modulate the information bearing audio signal. In Section 4.1 it was concluded that when a triangular comparison waveform was used for modulation it produced the least amount of harmonic distortion within the entire frequency space. This was because double edge sampled modulation exhibited reduced sideband harmonic distortion around the carrier frequency components. But do these harmonic components influence the baseband where audio frequency components exist?

If a high enough switching frequency is chosen for the modulation processes the sideband harmonics do not influence the baseband or audio band frequencies. This is a result of the large roll off that the sideband components exhibit. The effect is clearly illustrated in Figure 4.4, where a trailing edge NPWM scheme was used. Here a switching frequency of 352.8 kHz was used which was more than 16 times larger than the audio bandwidth of  $B = 20$  kHz. None of the sideband components had an influence within the audio baseband, since they could not be discerned from the noise floor which was far below the 24-bit noise floor (-145 dB). Thus if a large enough

switching frequency is chosen, carrier sideband harmonics do not influence the baseband frequency content.

Double edge modulation is computationally very expensive since it requires the modulation of two edges within the switching frequency. This modulation process using the PNPWM technique estimates the cross point or point using numerical algorithms. If two cross points need to be estimated within one switching interval very fast clock speeds are required.

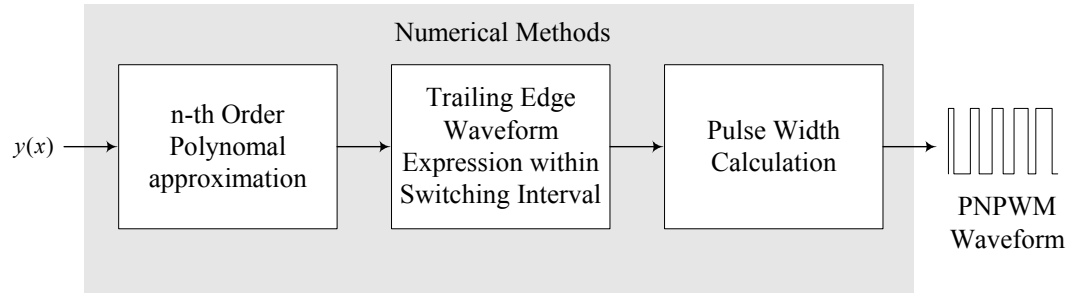
Trailing edge sampled modulation is therefore chosen as the best method for implementing PNPWM. It requires only one cross point calculation within a switching interval, and it is assured that when the switching frequency is chosen sufficiently large, carrier sideband harmonics have a negligible effect on the baseband frequency content. Trailing edge sampled modulation is chosen above leading edge sampled modulation because of its popularity.

The next section will explain how the cross-point within a switching interval is calculated by using numerical methods to accomplish PNPWM.

### 4.2.3 PNPWM building blocks

The previous section described a PWM technique which could be used to generate digital PWM without baseband harmonic distortion. It is similar to the NPWM but is a digital technique. It was decided that trailing edge modulation would be used to implement PNPWM, because of the reduction in computational overhead and the negligible effect the sideband harmonic components have in the baseband. That is if a high switching frequency is chosen relative to the input signal bandwidth. The building blocks of the PNPWM technique will be described here.

Polynomial interpolation for the input discrete modulation signal, finding an expression for the comparison waveform (linear interpolation), and calculating the crosspoint with these to generate a pulse varying output signal. These blocks are illustrated in Figure 4.7.



**Figure 4.7: Building blocks of the PNPWM modulation technique**

#### 4.2.3.1 Polynomial interpolation

An analogue waveform needs to be approximated using the discrete PCM samples at the input of the PWM modulator. It is done by fitting an  $n$ th order polynomial through the  $n+1$  PCM samples from both sides of the interval to be reconstructed.

These  $n+1$  samples are distinct and satisfy

$$x_0 < x_1 < x_2 < \dots < x_n, \quad (4.18)$$

where  $(y_0, y_1, y_2, \dots, y_n)$  represent these PCM sample amplitudes.

The objective is to find a polynomial curve that passes through the given points  $(x_i, y_i)$ ,  $i = 0, 1, \dots, n$ . Hence, as is described in [1], a polynomial  $p(x)$  needs to be found such that

$$p(x_i) = y_i \quad \text{for } i = 0, 1, \dots, n. \quad (4.19)$$

It is said that the polynomial  $p(x)$  interpolates the amplitudes  $y_i$  at the points  $x_i$  which are known as nodes. The polynomial  $p(x)$  is given by the form

$$p(x_i) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n, \quad (4.20)$$

which interpolates the PCM samples  $y_i$ .



Applying (4.19) to (4.20) leads to the system

$$\begin{aligned} a_0 + a_1x_0 + a_2x_0^2 + \dots + a_nx_0^n &= y_0 \\ a_0 + a_1x_1 + a_2x_1^2 + \dots + a_nx_1^n &= y_1 \\ &\vdots \\ a_0 + a_1x_n + a_2x_n^2 + \dots + a_nx_n^n &= y_n. \end{aligned} \quad (4.21)$$

This is a system of  $(n+1)$  linear equations in  $(n+1)$  unknowns:  $\{a_0, a_1, \dots, a_n\}$ .

In matrix form, the system is described by

$$\bar{X}a = \underline{y}, \quad (4.22)$$

where

$$\begin{aligned} \bar{X} &= [x_i^j] \quad i, j = 0, 1, \dots, n \\ a &= [a_0, \dots, a_n]^T, \quad \underline{y} = [y_0, \dots, y_n]^T. \end{aligned} \quad (4.23)$$

The matrix  $\bar{X}$  is known as the Vandermonde matrix [1]. Thus, solving for the system in equation (4.21) is equivalent to solving the polynomial interpolation problem.

The polynomial interpolation theorem can hence be stated as follows: *Given  $n+1$  distinct points  $x_0, x_1, \dots, x_n$  and  $n+1$  arbitrary real values  $y_0, y_1, y_2, \dots, y_n$ , there is a unique polynomial  $p(x)$  of degree  $\leq n$  that interpolates the points  $(x_i, y_i), [1]$ .*

From the  $n+1$  PCM samples a polynomial  $p(x)$  can thus be fitted and any value between these samples can be computed if the polynomial coefficients  $\{a_0, a_1, \dots, a_n\}$  are known. Using the PCM sample amplitudes and their position coordinates the polynomial coefficients can be directly calculated by using matrix algebra to manipulate (4.22) to give,

$$a = \bar{X}^{-1}\underline{y}. \quad (4.24)$$

With the polynomial coefficients calculated the expression for  $p(x)$  is solved.

Other methods do also exist in the calculation of the polynomial coefficients and are listed in Table 4.1.

Bezier Techniques
Cubic Splines
Neville's Algorithm
Newtons Interpolation Formula
Lagrange

**Table 4.1: Polynomial interpolation methods.**

The direct form calculation derived here is sufficient for implementation within an FPGA, since it only involves matrix multiplication and uses no other formulae.

The next step in calculating the cross point of  $p(x)$  within a particular switching interval involves finding an expression for the trailing edge saw-tooth wave.

#### 4.2.3.2 Trailing Edge Sawtooth Wave

Having fitted a polynomial  $p(x)$  of order  $n$  through  $n+1$  PCM samples it is now required to find an expression for the trailing edge saw-tooth wave (comparison waveform) within a switching interval to completely define the PNPWM process.

Suppose the maximum and minimum amplitude of the saw-tooth wave is 1 and -1 respectively, and the polynomial function  $|p(x)| < 1$  to insure that over modulation does not occur. Then an expression for the saw-tooth wave within any given switching frequency from Figure 4.5 is given by

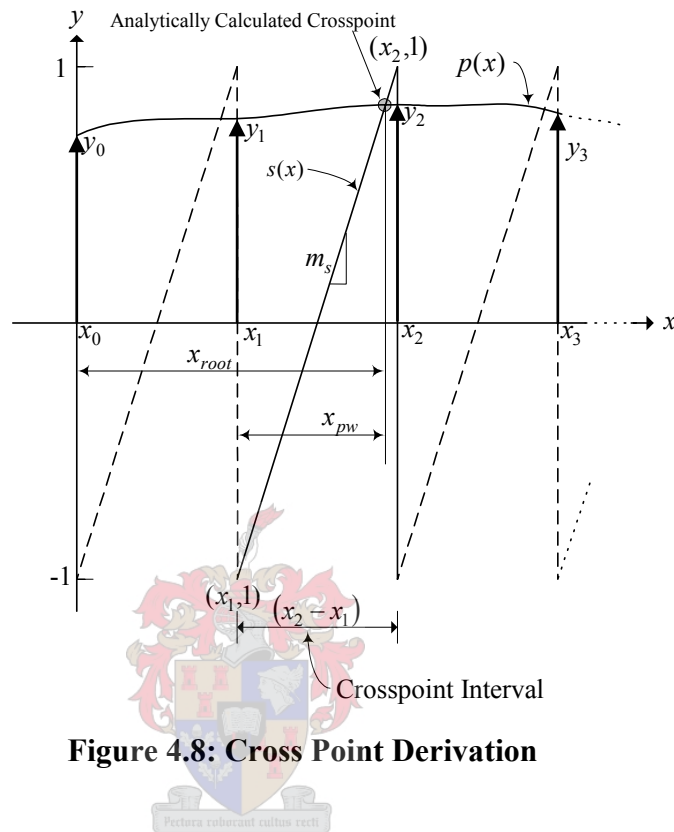
$$s(x) = m_s x - (1 + 2n), \quad (4.25)$$

where  $m_s$  is the gradient expressed as

$$m_s = \frac{2}{(x_2 - x_1)}, \quad (4.26)$$

and  $n$  is the current switching interval. This 1<sup>st</sup> order linear equation  $s(x)$  in (4.25) and the  $n^{\text{th}}$  order polynomial equation  $p(x)$  in (4.20) can now be used to determine the cross point within the switching interval. This is illustrated in Figure 4.8.

In Section 4.3 different methods of calculating the root or pulse width from the cross point will be evaluated for practical implementation within an FPGA. In the next section it will be shown how  $p(x)$  and  $s(x)$  can be used to calculate the PWM output.



**Figure 4.8: Cross Point Derivation**

#### 4.2.3.3 Root Finding and Pulse Width Calculation

To calculate the original analogue modulating wave cross point with the comparison waveform, expressions for both the approximated analogue waveform and comparison waveform are necessary. These expressions have been derived in the previous two sections and are given by  $p(x)$  and  $s(x)$  respectively.

The cross point of these two functions within a particular switching interval is given by

$$p(x) = s(x). \quad (4.27)$$

Equation (4.27) is transformed into a root finding problem when this expression is written as:

$$0 = p(x) - s(x). \quad (4.28)$$

Substituting (4.20) and (4.25) into (4.28) yields

$$\begin{aligned} 0 &= a_0 + a_1x + a_2x^2 + \dots + a_nx^n - [m_sx - (1 + 2n)] \\ &= [a_0 + (1 + 2n)] + (a_1 - m_s)x + a_2x^2 + \dots + a_nx^n. \end{aligned} \quad (4.29)$$

Equation (4.29) gives the final rational polynomial expression, the root or zero of this expression gives the x-coordinate ( $x_{root}$ ) of the cross point between  $p(x)$  and  $s(x)$ . The pulse width of this particular interval from Figure 4.8 is given by

$$x_{pw} = x_{root} - n(x_{n+1} - x_n). \quad (4.30)$$

Unfortunately solving for  $x_{root}$  in (4.29) is analytically impossible, necessitating the use of numerical methods for an approximate solution. For this reason two popular techniques for numerical root finding will be presented in the next section.

#### 4.2.4 Numerical root finding algorithms for PNPWM

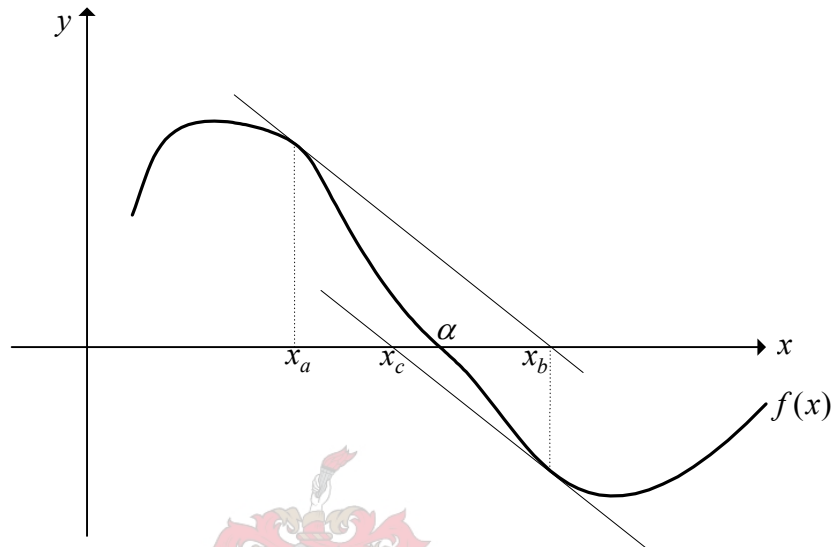
All of the PNPWM building blocks have been described in the previous section. The last stage of calculating the pulse width within a particular switching interval cannot be solved exactly. This is because no mathematical equation exist for finding the root of a polynomial function exhibiting an order larger than three.

Iterative methods however do exist which approximate these roots within a given interval, in this case, the switching interval. A couple of root finding techniques exist which can be used to iteratively solve polynomial roots. Within this chapter only two methods will be described and compared for the use of computing a pulse varying output signal.

These two methods are known as the Newton's and bisection methods. A variant of the bisection method will also be described which works on the same principle but is implemented more practically.

#### 4.2.4.1 Newton's Method

Newton's method is one of the most widely used iterative techniques for solving roots of equations [1]. Figure 4.9 gives a graphical interpretation of the method.



**Figure 4.9: Newton's method and first two approximations to its zero  $\alpha$ .**

To use the method an initial guess ( $x_a$ ) sufficiently close to the root  $\alpha$  is needed. The next approximation  $x_b$  is given by the point at which the tangent line to  $f(x)$  at  $f(x_a, f(x_a))$  crosses the  $x$ -axis. It is clear that the value  $x_b$  is much closer to  $\alpha$  than the original guess  $x_a$ . If  $x_{n+1}$  denotes the value obtained by the succeeding iterations, that is the  $x$ -intercept of the tangent line to  $f(x)$  at  $f(x_n, f(x_n))$ , then a formula relating  $x_n$  and  $x_{n+1}$ , known as Newton's method, is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n \geq 0 \quad (4.31)$$

provided  $f'(x_n)$  is not zero [1]. With the theory of Newton's method described, an example will be given to clarify how this method is used to generate a PNPWM output.

#### 4.2.4.2 PNPWM example using Newton's Method

This example continues from Section 3.6 where a 1 kHz sinusoidal discrete signal was interpolated to a sampling frequency of  $F_c = 352.8$  kHz. The increased sampling frequency ratio was chosen to be the same value as the switching frequency of the pulse width modulator; as is required from Section 4.1.1. A PNPWM output will be generated from this upsampled sinusoidal signal using Newton's method to estimate the crosspoint within a particular switching interval. This example therefore starts from the input signal and then progresses through the different building blocks of PNPWM process as illustrated in Figure 4.7.

The discrete upsampled sinusoid given by

$$y(m) = A \cos[2\pi(F / F')m], \quad (4.32)$$

here

$$\begin{aligned} A &= 0.95 \\ F &= 1 \text{ kHz} \\ F' &= 352.8 \text{ kHz.} \end{aligned} \quad (4.33)$$

For each new input PCM sample, an 8<sup>th</sup> order polynomial  $p(x)$  is fitted through the new sample and the previous eight samples as illustrated in Figure 4.10. Within the 5<sup>th</sup> interval the crosspoint between the polynomial and trailing edge saw-tooth wave is calculated using Newton's method. The saw-tooth comparison waveform within this interval is given by

$$s(x) = m_s x - 9, \quad (4.34)$$

where

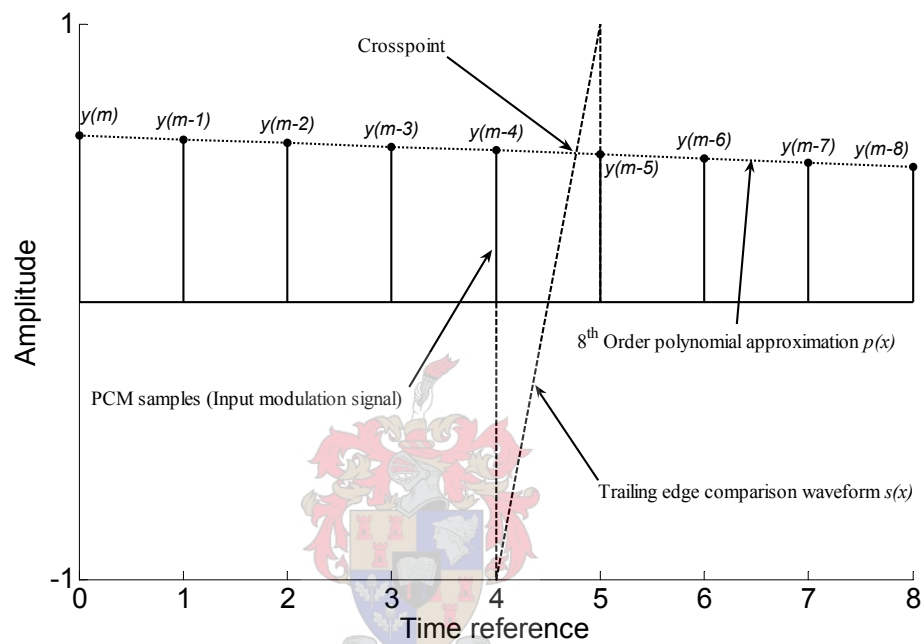
$$m_s = \frac{2}{T'}, \quad (4.35)$$

and

$$T' = \frac{1}{F'}. \quad (4.36)$$

The polynomial expression which is generated by the comparison between the polynomial and trailing edge saw-tooth from (4.29) yields,

$$\begin{aligned}
 f(x) &= a_0 + a_1x + a_2x^2 + \dots + a_8x^8 - [m_sx - (1 + 2n)] \\
 &= (a_0 + 9) + \left(a_1 - \frac{2}{T'}\right)x + a_2x^2 + \dots + a_8x^8.
 \end{aligned}
 \tag{4.37}$$



**Figure 4.10: PNPWM crosspoint derivation.**

The polynomial coefficients in (4.37) are calculated using (4.24) which is stated here for convenience

$$a = \bar{X}^{-1}y.
 \tag{4.38}$$

The system matrix  $\bar{X}$  contains the position or time parameter and is given by

$$\bar{X} = \begin{bmatrix} 1 & x & x^2 & \dots & x^8 \\ 1 & x & x^2 & \dots & x^8 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x & x^2 & \dots & x^8 \end{bmatrix}.
 \tag{4.39}$$

When each sample's integer position (see Figure 4.9) is substituted in (4.25) the system matrix becomes,

$$\bar{X} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 4 & \dots & 256 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 8 & 64 & \dots & 16777216 \end{bmatrix}. \quad (4.40)$$

The result obtained in (4.26) will be the same for every set of polynomial coefficients calculated, because the time window stays the same as the sample values are shifted down according to their time reference. The inverse of (4.40) is then calculated and multiplied with the current nine input PCM samples according to (4.38).

With the polynomial coefficients calculated, Newton's method is used to determine the crosspoint within the 5<sup>th</sup> interval. Firstly, it is necessary to determine the derivative of (4.37) which is

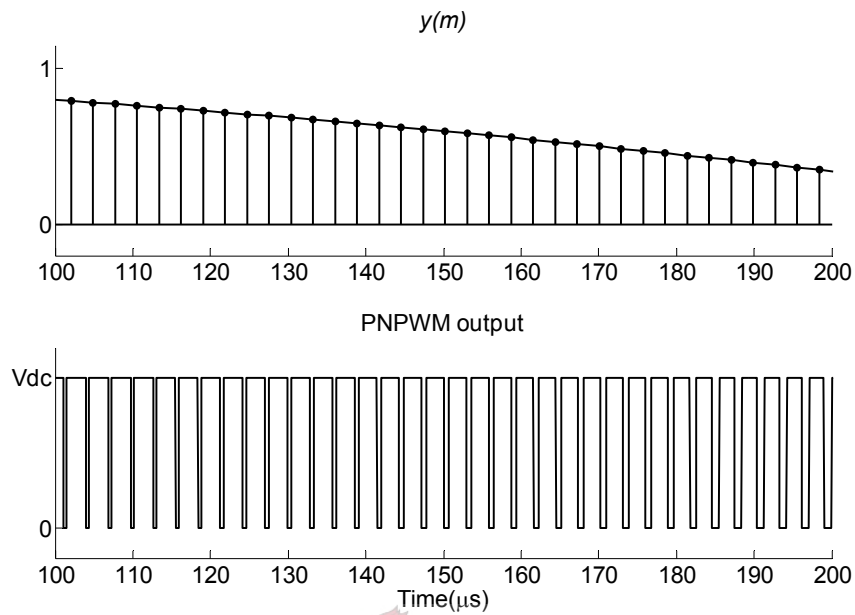
$$f'(x) = \left( a_1 - \frac{2}{T'} \right) + 2a_2x + 3a_3x^2 + \dots + 8a_8x^7. \quad (4.41)$$

Secondly, an initial guess of the crosspoint time coordinate in the center of the switching interval ( $x_0 = 4.5$ ) is inserted into (4.31) after which each iteration result is substituted back into (4.31) and re-iterated until a satisfactory crosspoint value is achieved. In this example three iterations were used to attain an appropriate root of (4.41), equivalent to the crosspoint coordinate between the polynomial and comparison waveforms.

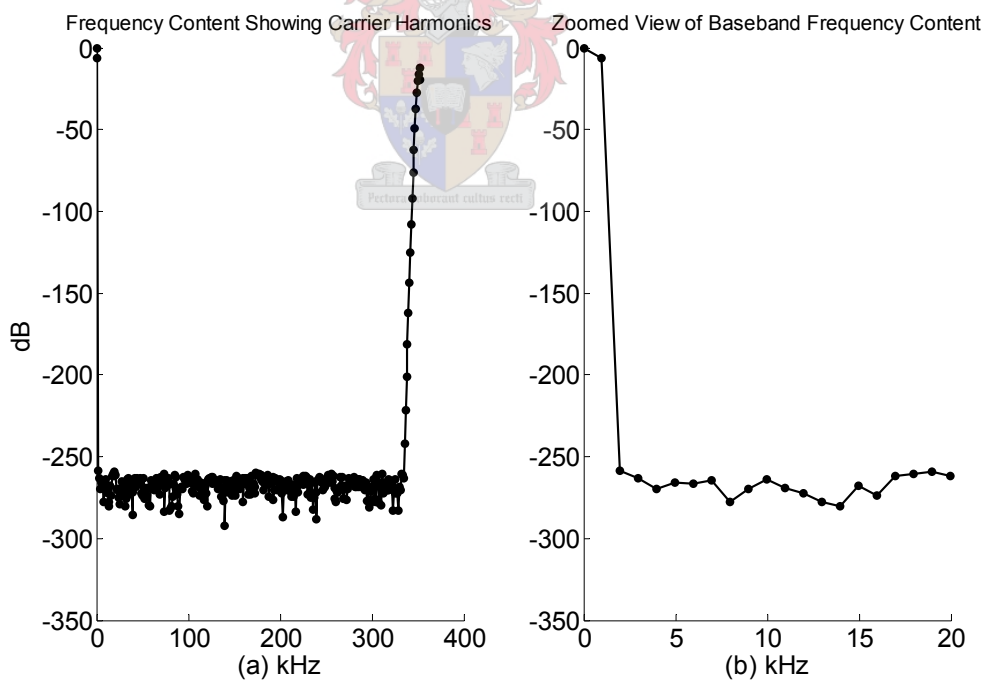
This root or crosspoint coordinate within the switching interval is then scaled as a ratio of the switching interval, thus yielding the current pulse width which is output by the pulse width modulator. The pulse train generated from one period of  $y(m)$  is evaluated by looking at its spectral content, calculated by a spectral estimate



method described in Appendix C. Figure 4.11 shows a section of the input modulating signal  $y(m)$  and its corresponding PNPWM output below it.

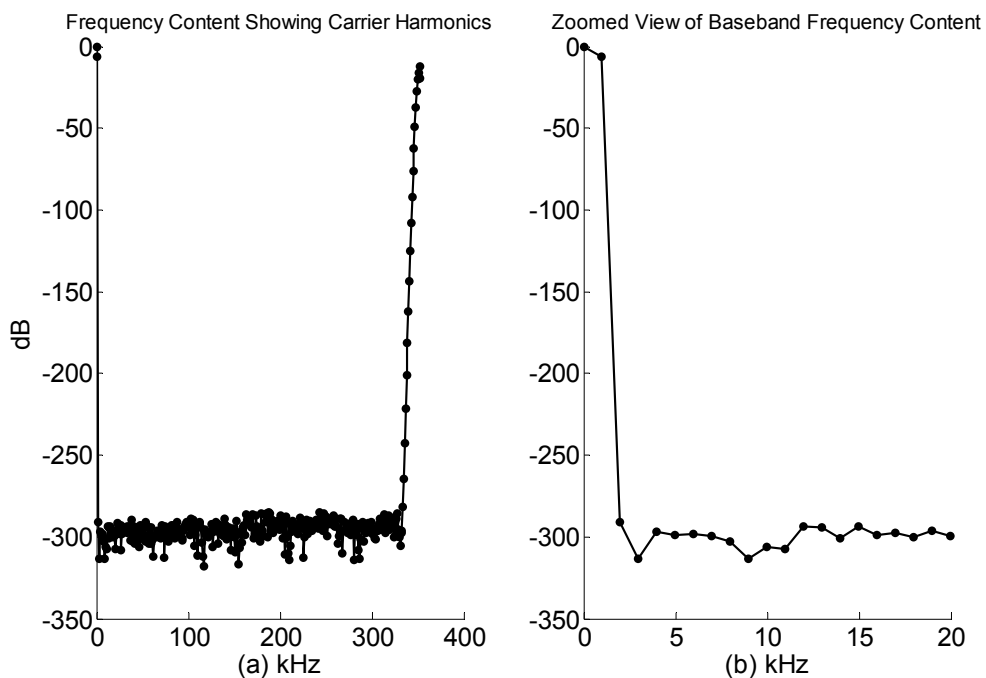


**Figure 4.11: PNPWM output of Newton's method**



**Figure 4.12: Spectrum of PNPWM using Newton's method.**

The PNPWM estimated spectra using Newton's method is shown in Figure 4.12. The Matlab ® code for this estimate is given in Appendix C2.



**Figure 4.13: Spectrum of NPWM using Newton's method.**

Figure 4.13 shows the NPWM spectra estimate using Newton's method as given previously in Figure 4.4. The carrier harmonic and its sideband components are the same in both Figure 4.12(a) and Figure 4.13(a). The only difference between the former and the latter figures is the increased noise floor which Figure 4.12(a) exhibits, causing some of the sideband carrier harmonics to be masked by the noise. The increased noise floor is of no concern since it still lies below -145 dB which is the desired dynamic range for 24-bit resolution.

Figure 4.12 (b) and Figure 4.13 (b) illustrate the PWM baseband spectral content of human hearing for the two schemes. It is important to note that no baseband harmonics occur above the noise floor of the PNPWM spectral content when using Newton's method. The fundamental 1 kHz tone and the DC component of both NPWM and PNPWM schemes are the same. Digital PWM signal's can therefore be generated with baseband harmonic components sufficiently attenuated when an efficient crosspoint derivation technique is used in the PWM modulator.

Now that Newton's method has been used to generate digital PWM by the use of the PNPWM, another crosspoint derivation scheme will be evaluated known as the bisection method. The binary search method will then be derived from the principle of the bisection method to iteratively find the crosspoint. After the binary method is described, the same example as was used with Newton's method will be evaluated for

the binary search method. These two methods will then be compared according to their practicality of implementation and speed of convergence to the root or crosspoint.

#### 4.2.4.3 Introduction to bisection and binary search methods

With Newton's method it was seen that the crosspoint between two waveforms could be determined by forming a new rational function: in (4.28) the crosspoint derivation problem became a root finding problem.

The bisection method evaluated here, also solves the root of a rational function based on the principle of halving intervals. Instead of creating a new rational function and finding its root to determine the crosspoint, a different approach will be used based on the binary search method.

The binary search method works on the same principle as the bisection method, but differs in the way it determines the crosspoint. It compares the two waveforms of interest  $p(x)$  and  $s(x)$  rather than creating a new rational function as with the bisection method. It should be clear that  $p(x)$  and  $s(x)$  are the input modulating and comparison waveforms respectively.

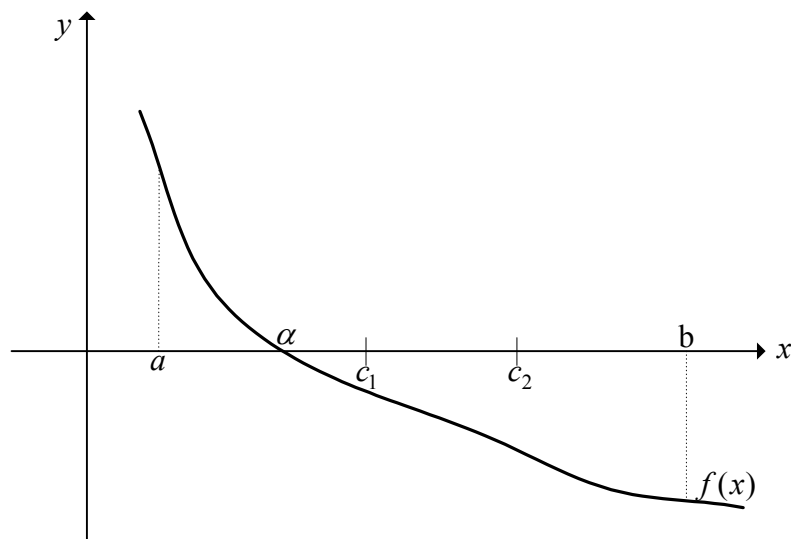
Firstly the theory of the bisection method will be presented, after which the binary search algorithm will be given.

#### Bisection method

Using [1], let  $f(x)$  be the function in (4.37), which is continuous on an interval  $[a, b]$ , such that

$$f(a)f(b) < 0. \quad (4.42)$$

It follows from (4.42) that there exists at least one zero of  $f(x)$  in  $(a, b)$ . It is assumed that  $f(x)$  has exactly one root  $\alpha$ , which corresponds to the desired crosspoint. Such a function is shown in Figure 4.14.



**Figure 4.14: Bisection method and the first two approximations to its zero  $\alpha$**

The bisection method is based on halving the interval  $[a, b]$  to determine a smaller and smaller interval within which  $\alpha$  must lie. The procedure is carried out by first defining the midpoint of  $[a, b]$ ,  $c = (a + b)/2$  and then computing the product  $f(c)f(b)$ . If the product is negative, the root is in the interval  $[c, b]$ . If the product is positive, the root is in the interval  $[a, c]$ . Thus, a new interval containing  $\alpha$  is obtained. The process of halving the new interval continues until the root is located as accurately as desired, that is

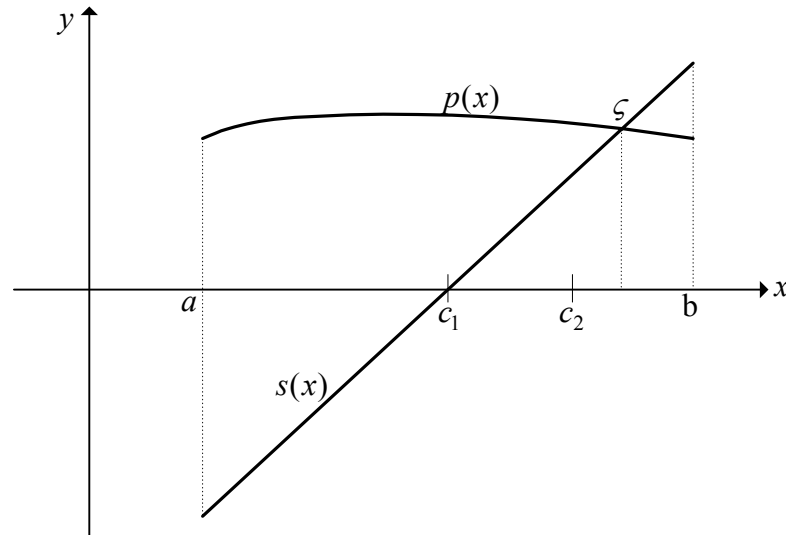
$$|a_n - b_n| < \varepsilon \quad (4.43)$$

Where  $a_n$  and  $b_n$  are the endpoints of the  $n$ -th interval  $[a_n, b_n]$  and  $\varepsilon$  is a specified tolerance value [1].

### Binary Search method

This approach uses the search principle that the bisection method uses to locate the root of the polynomial function. Instead of using the rational function in (4.28) to find the root, functions  $p(x)$  and  $s(x)$  are used to find the crosspoint within a certain interval as described previously. Therefore within the interval  $[a, b]$  a crosspoint between  $p(x)$  and  $s(x)$  exists. The midpoint of  $[a, b]$  is calculated by

$c = (a + b)/2$ , then if  $p(c) > s(c)$  the crosspoint  $\zeta$  occurs within the interval  $[c, b]$ , but if  $p(c) < s(c)$  the crosspoint  $\zeta$  occurs within the interval  $[a, c]$ . Figure 4.15 shows such a procedure.



**Figure 4.15: Binary search method and the first two approximations to its crosspoint  $\zeta$**

The process of halving the interval is repeated until a satisfactory crosspoint value is determined with an allowable tolerance given by

$$|a_n - b_n| = 2^{-n} < \varepsilon. \quad (4.44)$$

Where the exponent  $n$  of base 2 in (4.44) gives the amount of intervals formed before reaching the desired tolerance value. Because the allowable tolerance is calculated using a base of two, the name binary search was given to the method.

This method of searching is more efficient than the bisection method because it does not need to do a multiplication when determining the next halved search interval. Only a comparison is needed.

Next, this method will be evaluated using the same example used for Newton's method to generate PNPWM.

#### 4.2.4.4 PNPWM example using the binary search method

The same example used in Section 4.2.3.5 will now be done here using the binary search algorithm described in the previous section. The derivation of the 8<sup>th</sup> order polynomial expression  $p(x)$  and the trailing edge saw tooth wave expression  $s(x)$  stays the same here as derived in Section 4.2.3.5. Again, the crosspoint between these functions within the 5<sup>th</sup> interval needs to be determined as illustrated in Figure 4.10. The strategy for using the binary search method in calculating the PNPWM will now be explained.

After  $p(x)$  is approximated from  $y(m)$  the binary search algorithm is used to find the crosspoint within a tolerance of  $\varepsilon_1 = 2^{-9}$ . After this first tolerance has been achieved,  $p(x)$  is approximated by a 1<sup>st</sup> order linear equation  $p_{lin}(x)$  which is given by

$$p_{lin}(x) = \begin{cases} m_{lin}x + p(c_8) & \text{if } c_9 > c_8 \\ m_{lin}x + p(c_9) & \text{if } c_9 < c_8 \end{cases}, \quad (4.45)$$

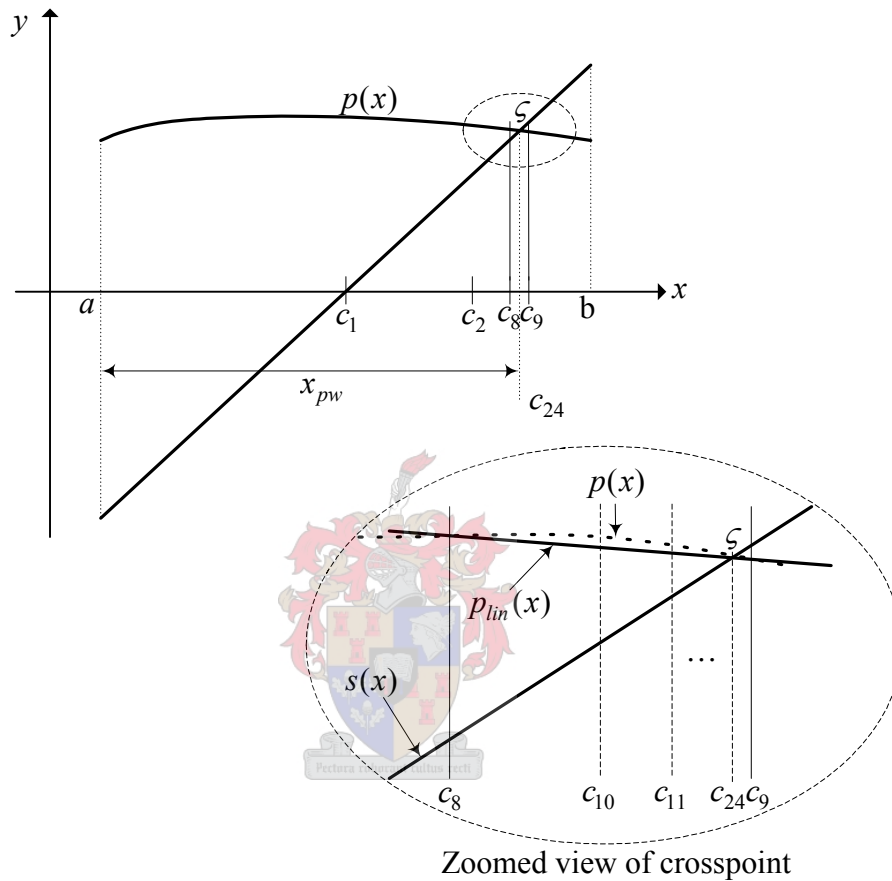
where

$$m_{lin} = \left( \frac{p(c_9) - p(c_8)}{c_9 - c_8} \right). \quad (4.46)$$

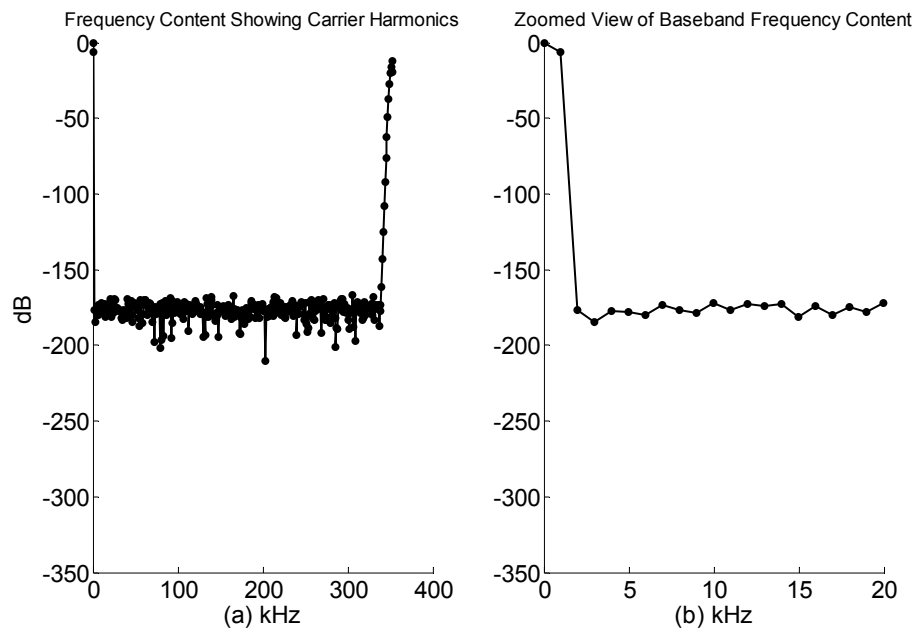
A further binary search is then performed between  $p_{lin}(x)$  and  $s(x)$  within the first tolerance interval until a second tolerance  $\varepsilon_2 = 2^{-24}$  is achieved. The 1<sup>st</sup> order approximation very closely follows the 8<sup>th</sup> order polynomial within the first tolerance interval and therefore reduces the amount of calculation needed to achieve the second tolerance value. The midpoint value attained from the second tolerance interval represents the 24-bit PWM signal.

Figure 4.16 illustrates graphically how the binary search method is applied to generate PNPWM.  $x_{pw}$  represents the pulse width distance achieved using the binary search method. Figure 4.17 shows the spectrum of the PNPWM generated using the binary search strategy. It can be seen that a SNR or dynamic range of more than 150 dB was achieved, which is more than sufficient for 24-bit pulse width modulation.

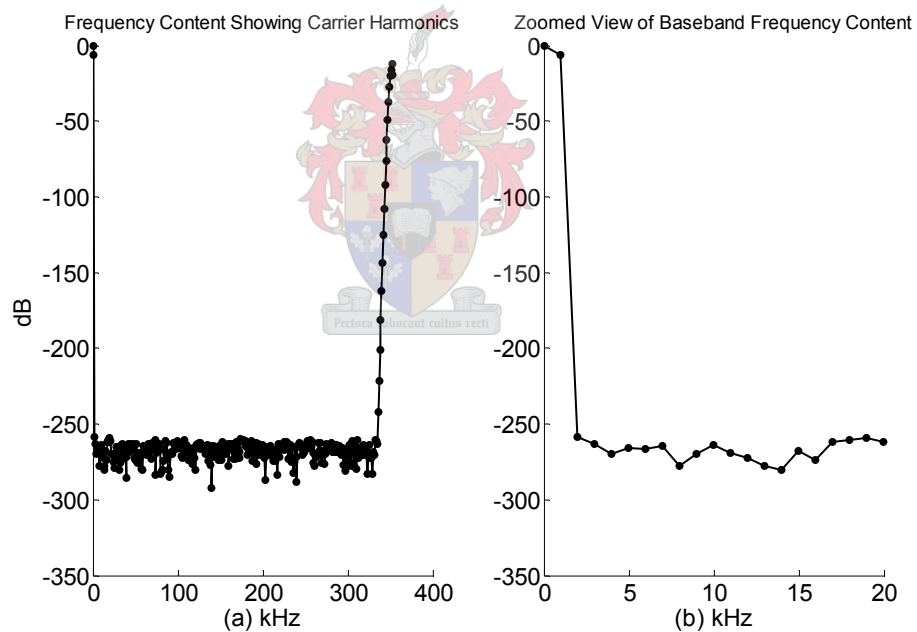
It is important to note that many other numerical methods exist in which 24-bit PNPWM can be generated. But to investigate and compare all of these methods, would be impossible within the scope of this theses. A proper comparison between the two previously discussed methods will be done instead. Important properties of both methods will be considered for practical implementation within an FPGA.



**Figure 4.16: Binary search strategy for generating PNPWM.**



**Figure 4.17: Spectrum of PNPWM using Binary search strategy.**



**Figure 4.18: Spectrum of PNPWM using Newton's method.**

4.2.4.5 Comparison between Newton's and the Binary method for PNPWM

When comparing the spectra of Figure 4.17 (Binary search strategy with Matlab code given in Appendix C3) to Figure 4.18 (Newton's method) it is clearly seen that the binary searched PNPWM has a reduced dynamic range. Both exhibit an SNR well above 145 dB, with no baseband harmonic distortion above the noise floor.



This qualifies both methods for 24-bit PWM implementation. Only one of these methods can however be implemented.

The criteria for comparison and selection are based on the convergence and complexity of each method. Convergence is defined as the number of iterations needed to find the crosspoint or root within a certain error constraint resulting in 24-bit PWM output. Complexity is defined as the number of multiplications, divisions, additions and subtractions used in the method.

Newton's method converges quadratically to the desired root whilst the binary search method converges according to  $\log_2 N$  where  $N$  represents the maximum integer value of the crosspoint  $x$ -coordinate within a switching interval. It was seen in Section 4.2.3.5 that Newton's method only required three iterations to assure a SNR of more than 145 dB. In contrast the binary search method required nine iterations within the first tolerance interval and a further fifteen iterations within the second interval to reach a required SNR of more than 145 dB. Newton's method therefore outperforms the binary search method in terms of the convergence criteria.

Comparison of complexity involves counting all multiplications, divisions, additions and subtractions of one iteration of the specific method being evaluated. These arithmetic operation counts are not considered in the calculation of the polynomial coefficients or in the comparison waveform calculations. This is since they remain the same for both methods in the calculation of the PWM width outputs.

For Newton's method, arithmetic calculations are considered for expressions (4.37), (4.41) and (4.31). Table 4.2 shows the count of the arithmetic calculations for each of these expressions.

Expressions	Multiplications & Divisions	Additions & Subtractions
$f(x)$	16	10
$f'(x)$	14	7
$x_{n+1}$	1	1
<b>Total</b>	<b>31</b>	<b>18</b>

**Table 4.2: Arithmetic counts of one iteration using Newton's method.**

The binary method arithmetic calculations are considered for expressions (4.20) and (4.25) within the first tolerance interval  $\varepsilon_1$ , and expressions (4.20) and (4.45) within the second tolerance interval  $\varepsilon_2$ . Table 4.3 shows the arithmetic count for each expression within each tolerance interval.

Tolerance Interval	Expressions	Multiplications & Divisions	Additions & Subtractions
$\varepsilon_1$	$p(x)$	16	8
	$s(x)$	1	1
	<b>Total</b>	<b>17</b>	<b>9</b>
$\varepsilon_2$	$p_{lin}(x)$	1	1
	$s(x)$	1	1
	<b>Total</b>	<b>2</b>	<b>2</b>

**Table 4.3: Arithmetic counts of one iteration using the binary search method.**

From Table 4.2 and Table 4.3 it is clear that the binary search method has a reduced computational complexity when considering a single iteration. But when considering that Newton's method only required 3 iterations, whereas the binary search method required 24 iterations to achieve 145 dB SNR, it seems that Newton's method should be the preferred choice.

However, the binary search arithmetic can further be reduced since every possible midpoint within a switching frequency can be calculated beforehand (as is the case with the polynomial coefficients). When these midpoints  $\{c_1, c_2, c_3, \dots, c_N\}$  or  $x$ -coordinates as illustrated in Figure 4.15 are substituted in equation (4.20) a system matrix results, given by

$$\bar{X}_{mid} = \begin{bmatrix} 1 & c_0 & c_0^2 & c_0^3 & \cdots & c_0^8 \\ 1 & c_1 & c_1^2 & c_1^3 & \cdots & c_1^8 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & c_N & c_N^2 & c_N^3 & \cdots & c_N^8 \end{bmatrix} \quad (4.47)$$

where  $N$  is the maximum amount of possible midpoint values within a switching frequency. The amplitude of  $p(x)$  at a midpoint  $c_n$  is calculated by multiplying the transpose of the  $n^{\text{th}}$  row of (4.47) with the polynomial coefficients ( $a$ ) which yields,

$$p(c_n) = [a_0 \quad a_1 \quad \cdots \quad a_8] \begin{bmatrix} c_n \\ c_n^2 \\ \vdots \\ c_n^8 \end{bmatrix}. \quad (4.48)$$

If the system matrix in (4.47) is stored in memory as a lookup table, the polynomial amplitude at that specific midpoint can be calculated using the appropriate memory address. This approach halves the number of multiplications necessary to compute  $p(x)$ . Table 4.4 shows the updated arithmetic using (4.47) as a lookup table.

Tolerance Interval	Expressions	Multiplications & Divisions	Additions & Subtractions
$\varepsilon_1$	$p(x)$	8	8
	$s(x)$	1	1
	<b>Total</b>	<b>9</b>	<b>9</b>
$\varepsilon_2$	$p_{lin}(x)$	1	1
	$s(x)$	1	1
	<b>Total</b>	<b>2</b>	<b>2</b>

**Table 4.4: Arithmetic counts of one iteration using the binary search method and lookup table.**

With the binary search method optimized, a complete comparison between the two methods can be done taking into account their number of iterations needed to converge and the complexity in converging. The number of iterations used in the Newton's method example (4.2.3.5), and the binary search strategy example (4.2.3.8) to generate 24-bit PNPWM will be used here for comparison. Table 4.4 shows the arithmetic complexity when these iteration values are considered.

	Tolerance Interval	Mul & Div	Add & Sub	Iterations	Total Mul & Div	Total Add & Sub
Newton's Method	-	31	18	3	<b>93</b>	<b>54</b>
Binary Search Method	$\varepsilon_1$	9	9	9	81	81
	$\varepsilon_2$	2	2	15	30	30
	<b>Total</b>				<b>111</b>	<b>111</b>

**Table 4.5: Total arithmetic complexity of the two PNPWM methods.**

Results from Table 4.5 indicate that Newton's method theoretically outperforms the optimized binary search method in both the categories of convergence and complexity. It would seem that the Newton's method would be the preferred choice for PNPWM implementation again.

Unfortunately it has a drawback in its practical computation: It has a division arithmetic operation when calculating iterations. Generally, division arithmetic takes more time and resources within practical implementation than multiplication arithmetic.

Because of this reason, Newton's method and the optimized binary search method compares head to head and both are suitable methods for PNPWM implementation. Only one of these methods will however be implemented practically. It was decided to implement the binary search method to avoid using any division arithmetic in the practical implementation of PNPWN within an FPGA.

### 4.3 Summary

This chapter started off defining the PWM process. It then went on to define different PWM schemes. These schemes were compared and evaluated according to their spectral content using a two-dimensional Fourier series expansion. From these comparisons it became clear that UPWM schemes, which are digital, exhibited unwanted baseband harmonic distortion whereas NPWM schemes, which are analogue, did not exhibit baseband harmonic distortion. Another digital PWM scheme was therefore introduced known as PNPWM which could estimate the NPWM crosspoint from digital data through numerical methods.

The numerical methods used in PNPWM are polynomial interpolation and linear interpolation. These methods were implemented to estimate the crosspoint within a particular switching interval ( $F_c$ ). The crosspoint of these numerical expressions was estimated to find the pulse width using two different root finding methods.

The methods used are known as the Newton's method and the bisection method. The bisection method was adapted into an optimal binary search method. Newton's method and the binary search method were compared according to their characteristics of convergence and computational complexity.

Newton's method has a better theoretical performance than the binary search method when generating PNPWM. Unfortunately this method contains division arithmetic which takes more time and resources to compute. Therefore, the binary search method will perform similarly to Newton's method when implemented practically, since it exhibits no division arithmetic in its computation. Because of this reason it was decided to implement the binary search strategy described to implement a 24-bit PNPWM process.

# Chapter 5 - Noise Shaping

## 5.0 Introduction

The pulse width is calculated digitally at a resolution of 24-bits through the PNPWM technique. The question now arises: Are existing hardware clock speeds capable of pulsing out these high resolution gating signals to the power electronic converter? The answer to the question is “no”, and this chapter describes why it is impossible. It then presents a solution known as “noise shaping” which reduces the bit resolution of the PWM output without sacrificing the SNR within the audio baseband.

Before the noise shaping process is presented, it is necessary to clarify the decision for choosing the switching frequency at 352.8 kHz. After this discussion the noise shaper will be described in its entirety. Continuing from the previous chapters, simulations will follow to confirm the working and design of the noise shaper both theoretically and practically.

## 5.1 Choice of Switching Frequency

Chapter 3 provided an in-depth description of an interpolation process whereby a 24-bit digital audio signal (sampled at 44.1 kHz) was upsampled to a frequency of 352.8 kHz. This upsampled frequency is the switching frequency ( $F_c$ ) of the modulator described in Chapter 4.

But why is this specific frequency value used as the switching frequency of the modulator? There are various reasons for its choice, but the most important of these enables the use of the noise shaping process which will be described within this chapter in later sections. The different factors influencing the choice of the switching frequency are now presented:

First, from Chapter 4, it was shown that the switching frequency represents the first carrier harmonic, which produces multiples of itself within the PWM output spectrum. Around these carrier components, sideband harmonics exist. When the switching frequency is chosen at a higher rate, the carrier harmonic components move further away from the baseband frequency content. This implies that the sideband harmonic components situated around the carrier components also move with the

carrier harmonics, causing less sideband harmonic distortion within the baseband. In [12] it is given that when the switching frequency is chosen approximately ten times the bandwidth of the baseband, the sideband harmonic components have a negligible effect within the baseband. The choice of the current switching frequency ( $F_c = 352.8$  kHz) is seventeen times that of the audio baseband, implying that sideband harmonic distortion for this application is negligible.

A second consideration in the choice of the switching frequency concerns itself with the switching ability of the amplifier or converter stage. MOSFET's are used to amplify the small PWM gating signals output from the two-level modulator. The choice of the switching frequency therefore has an influence on the switching ability of a specific MOSFET. A switching frequency therefore needs to be chosen according to the MOSFET's switching characteristics to obtain the least amount of distortion. Not only this, but many other factors and components within the converter stage influence the switching frequency. To consider all of these factors falls outside the scope of this thesis. A switching frequency of 352.8 kHz does seem to be a practicable choice for the power electronic output stage since it was successfully implemented in [15].

Third, according to the Nyquist theorem presented in Section 3.1.3 an analogue signal can be recovered from its digital counterpart when it is sampled at twice its bandwidth. The sampling frequency therefore represents the range of frequencies (negative and positive) that will be kept in the sampling process. When the sampling frequency is however chosen to be higher than twice the bandwidth of the signal baseband, oversampling occurs. This means that a larger bandwidth is generated than actually needed for the signal's frequency content. In other words, some part of the bandwidth is unused. Noise shaping coders uses this unused part of the bandwidth to increase the SNR within the signals baseband. Therefore a sampling frequency chosen at a high enough rate (when compared to the Nyquist frequency) implies that noise shaping coders can be considered. The current switching frequency ( $F_c$ ) is a multiple of eight larger than the Nyquist frequency of 44.1 kHz, implying that a large enough unused bandwidth will be generated for the implementation of a noise shaping coder.

Taking all the above factors into consideration, it seems that the switching frequency of  $F_c = 352.8$  kHz suffices. This is since the choice agrees with negligible sideband harmonic distortion, practical implementation of the converter amplifier stage and ensures enough increased bandwidth for the noise shaping process.

It is now clear why this frequency number was used as the upsampling rate in Chapter 3, and then related to the switching frequency in Chapter 4. Next, the physical noise shaping process will be extrapolated using this specific switching frequency.

## 5.2 Noise-Shaping Coders

The third factor in the choice of the oversampling frequency given above, vaguely described the noise shaping process and why it is necessary. It is the goal of this section to give a clearer understanding of the working of the noise shaping process and why its use is necessary to implement digital PWM.

### 5.2.1 Clock speed constraints

Hardware constraints do not make it possible for a modulator to directly convert a 24-bit crosspoint value to a 24-bit PWM output. This is because  $2^{24} - 1$  integer values are necessary to represent the 24-bit pulse width signal on the discrete time axis within one switching interval. Therefore,

$$T_{clock} = \left(\frac{1}{F_c}\right) \times \left(\frac{1}{2^b - 1}\right) = \left(\frac{1}{352.8 \times 10^3}\right) \times \left(\frac{1}{2^{24} - 1}\right) \approx 1.689 \times 10^{-13} \text{ s}. \quad (5.1)$$

This implies a clock frequency of at least:  $F_{clock} = 1/T_{clock} \approx 5919$  GHz. Table 5.1 shows the range of approximate clock frequencies needed for different desired PWM output resolutions using (5.1) and the chosen switching frequency of  $F_c = 352.8$  kHz.



PWM bit resolution	Clock Rate (GHz)
24	5919.00
16	23.12
10	0.361
8	0.09

**Table 5.1: Needed clock rates for certain PWM bit resolutions.**

The maximum clock frequency available within the specific FPGA used in this thesis is 400 MHz, a far cry from the clock frequencies needed in the first and second rows of Table 5.1. to attain 24-bit or 16-bit PWM output resolution. This necessitates the use of noise shaping coders which have the ability to reduce the bit size of the PWM output (therefore reducing the clock speed needed to generate it), but still maintaining baseband signal quality. From Table 5.1 only 10-bit and 8-bit PWM resolutions are therefore feasible for implementation with the current clock speed of 400 MHz. Next it will be described how bit-size reduction is accomplished through noise shaping.

### 5.2.2 Bit-size reduction through noise shaping

Information theory [12] predicts that when a bandlimited signal is oversampled, the output data can tolerate a reduction in amplitude resolution, yet maintain a similar baseband SNR [21].

The amplitude resolution referred to here is similar to the 24-bit pulse width calculated by the PNPWM process in Chapter 4, and the in-band frequency range is similar to the bandwidth of the input modulating signal (digital audio signal).

Therefore, the PWM output value which is finely quantized at a resolution of 24-bits can be requantized to a lower resolution. This implies an increased noise floor of the signal bandwidth. The role of the noise shaper is to take the requantized noise added to the baseband and shift it into the unused bandwidth area which was generated by the interpolation process in Chapter 3. This causes the noise floor within the baseband to be reduced and the noise floor in the unused band to be increased, thus resulting in an increased SNR within the baseband. Although the bit resolution of the PWM output has now been reduced, the increased SNR within the baseband

ensures that the quality of the PWM output is similar before requantization was introduced.

The next section will present the different components of the noise shaper which reduces the bit resolution of the PWM output.

### 5.2.3 Recursive noise shaper

Figure 5.1 shows an efficient recursive noise shaping architecture. It consists of a coarse quantizer embedded in an error feedback loop to alter the frequency distribution of the quantization noise associated with an input digital signal [18].

The noise shaper accepts a finely quantized  $b$ -bit input and produces a more coarsely quantized  $b'$ -bit output ( $b > b'$ ). The quantizer generates a requantization error signal by truncating the low-order bits of the signal presented to its input by the summing node. The spectrum of this requantization error signal is then frequency shaped by a filter  $H(z)$  and then fed back to the input as illustrated in Figure 5.1.

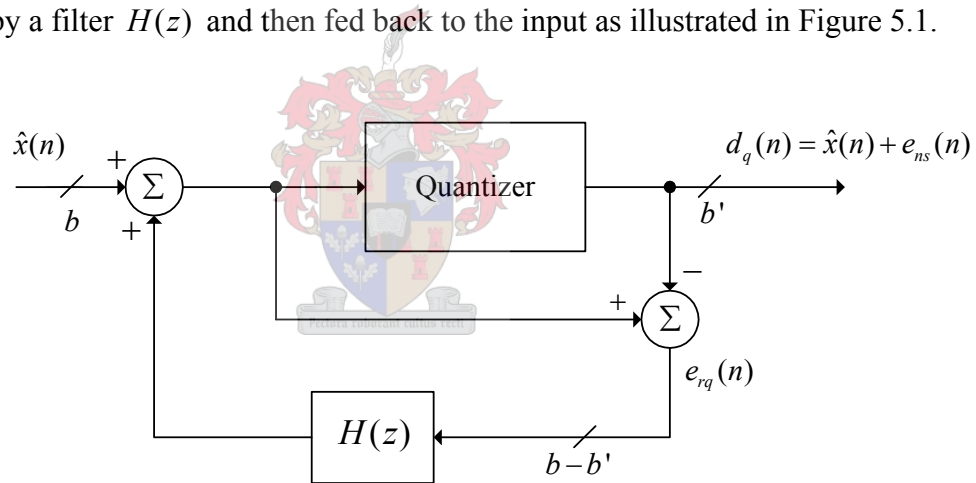


Figure 5.1: Noise-shaper architecture altered from [18].

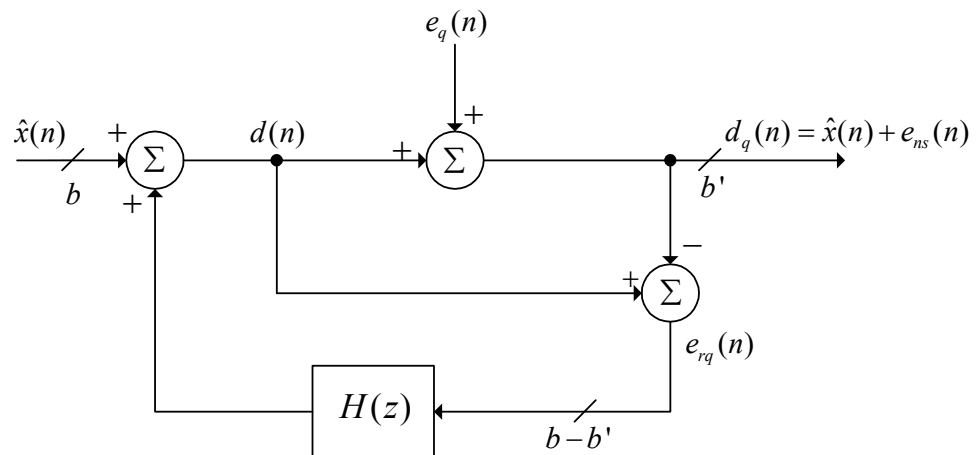


Figure 5.2: Quantizer modeled as added noise source.

### 5.2.4 Noise shaping quantizer

The midtread quantizer within the noise shaper can be modeled as a linear network with an additive noise source  $e_q(n)$ , this is shown in Figure 5.2. The following assumptions are made concerning the statistical properties of  $e_q(n)$  [3]:

- The error  $e_q(n)$  is uniformly distributed over the range  $-\Delta/2 < e_q(n) < \Delta/2$ , where  $\Delta = R/2^{b+1}$  is the step size of the quantizer,  $R$  represents the amplitude range of the quantizer and  $b+1$  represents the word length (Figure 5.3).
- The error sequence  $e_q(n)$  is a stationary white noise sequence.
- The error sequence  $e_q(n)$  is uncorrelated with the signal sequence  $d(n)$  in Figure 5.2.

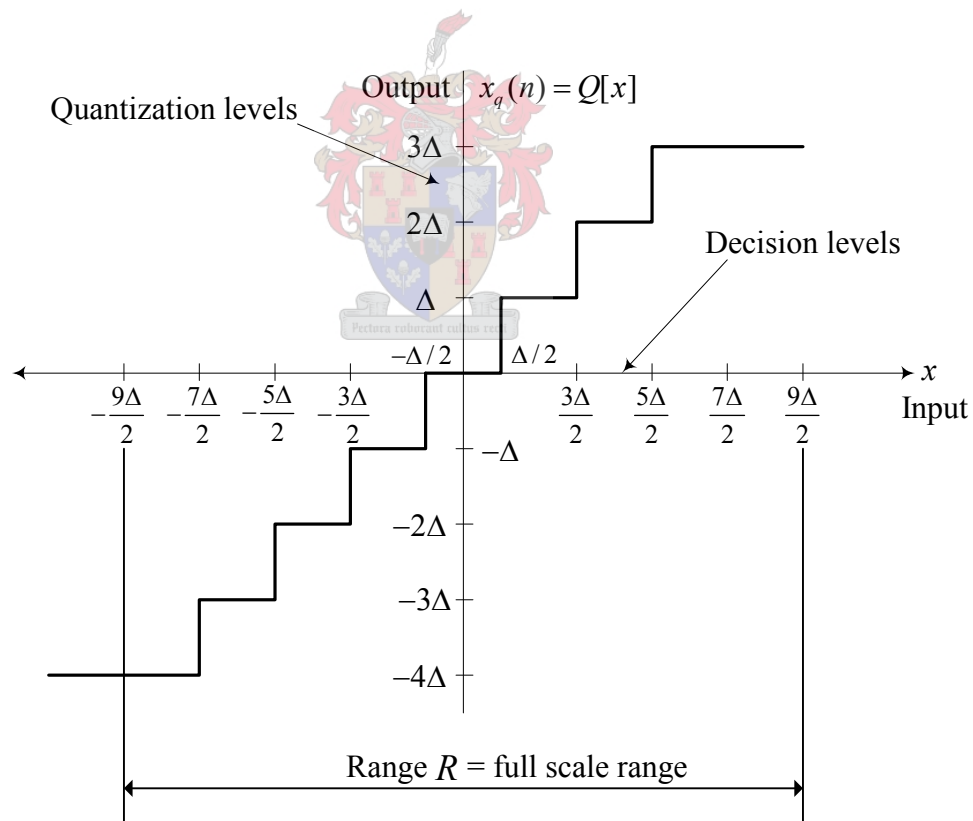


Figure 5.3: Example of a midtread quantizer [Digital signal processing textbook].

### 5.2.5 Derivation of the noise transfer function

An analysis of the noise shaper in Figure 5.2 is now given. Let  $\hat{X}(z)$ ,  $D_q(z)$ ,  $E_{rq}(z)$ ,  $E_{ns}(z)$  be the respective  $z$ -transforms of  $\hat{x}(n)$ ,  $d_q(n)$ ,  $e_{rq}(n)$ ,  $e_{ns}(n)$ . With the assumptions made for  $e_q(n)$  it is clear that  $d_q(n)$  is simply the sum of the input and the noise-shaped error,

$$d_q(n) = \hat{x}(n) + e_{ns}(n). \quad (5.2)$$

From Figure 5.2 it is apparent that in the  $z$ -domain the following expression can be deduced,

$$E_{rq}(z) = \hat{X}(z) + E_{rq}(z)H(z) - [\hat{X}(z) + E_{ns}(z)], \quad (5.3)$$

this is manipulated into

$$NTF(z) = \frac{E_{ns}(z)}{E_{rq}(z)} = H(z) - 1, \quad (5.4)$$

where  $NTF(z)$  is a noise transfer function [18]. If  $NTF(z)$  is evaluated on the unit circle,  $z = e^{j\omega T_c}$ , it is seen that the output error spectrum  $E_{ns}(\omega)$  is a frequency-weighted version of the spectrum of the requantization error signal produced by the quantizer  $E_{rq}(\omega)$ .

Any information available about the statistical or spectral properties exhibited by the error signal (generated by the quantizer  $e_{rq}(n)$ ) can be used in order to select an appropriate feedback filter ( $H(z)$ ) to approximate some desired output error spectrum ( $E_{ns}(\omega)$ ) as described in [17].

It is known from Section 5.2.4 that  $e_q(n)$  closely approximates a wide-sense-stationary white-noise discrete-time random process. As such,  $S_{ns}(\omega)$ , the power spectral density (PSD) of the output error, is proportional to the squared magnitude frequency response of  $NTF(z)$ :

$$S_{ns}(\omega) = |NTF(e^{j\omega})|^2 S_{rq}(\omega) \propto |NTF(e^{j\omega})|^2 \quad (5.5)$$

The noise shaper cannot reduce the total output requantization error power associated with a regular  $b'$ -bit linear digital signal. The noise shaped requantized error power of the  $b'$ -bit quantization process (Figure 5.2) actually exceeds the conventional  $b'$ -bit non-noise shaped error power by a factor:

$$K = \sum_{n=0}^{N-1} ntf[n]^2 = 1.0 + \sum_{n=1}^{N-1} h[n]^2 \geq 1.0, \quad (5.6)$$

with  $ntf(n)$  and  $h(n)$  the impulse responses of the noise transfer function and the feedback filter, respectively[17].

Next a few of the noise shaper transfer function characteristics will be highlighted in order to gain an understanding of the design of the feedback filter  $H(z)$ .

### 5.2.6 Characteristics of the noise shaper

Tewksbury and Hallock [25] have shown that an optimal function for the noise shaper characteristic  $NTF(z)$  is

$$H(z) - 1 = \left( \frac{z-1}{z} \right)^N, \quad (5.7)$$

this is effectively  $N$  cascaded digital differentiators. Thus for a given filter order  $N$ , the slope of the shaping function against frequency is maximum and gives the best suppression of low-frequency distortion.

The frequency domain representation of  $NTF(z)$  is determined by substituting

$$z = e^{j2\pi f / F_c}, \quad (5.8)$$

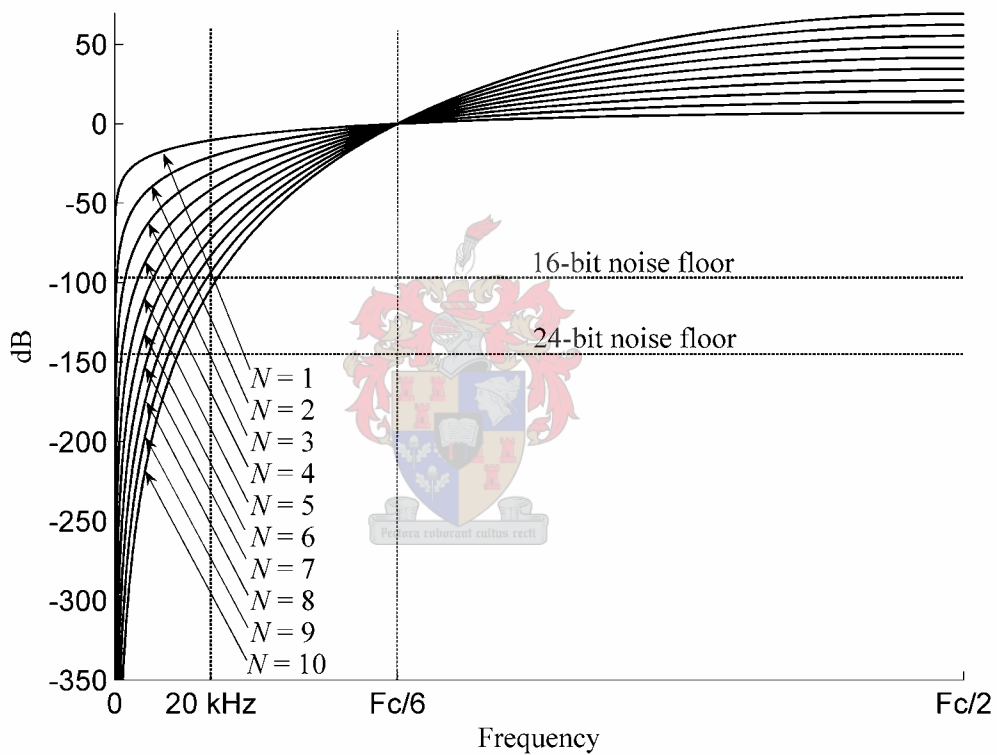
resulting in

$$|NTF(f)| = \left[ 2 \sin \left( \frac{\pi f}{F_c} \right) \right]^N. \quad (5.9)$$

Figure 5.4 illustrates the frequency response of (5.7) for different orders of  $N$ . These graphs reveal two frequencies of interest,

$$\left|NTF\left(\frac{F_c}{6}\right)\right|=1, \tag{5.10}$$

$$\left|NTF\left(\frac{F_c}{2}\right)\right|=2^N. \tag{5.11}$$



**Figure 5.4: Noise Transfer Function at various orders of  $N$ .**

Equation (5.9) reveals that all graphs take unit value at  $F_c/6$  and that reduction is achieved only for  $f < F_c/6$  Hz, while for  $F_c/6 < f < F_c/2$  the noise spectrum is actually amplified, reaching a maximum at  $F_c/2$  Hz [21].

All parts of the noise shaping coder have now been described; an optimal design expression has also been presented for the feedback filter  $H(z)$  in (5.7). This filter characteristic could be generated based on the assumptions made for the

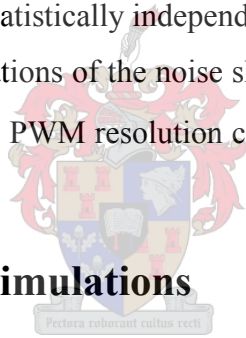
quantizer process described in 5.2.4. A few further remarks are in order before continuing to the simulation of the noise shaping process:

Oversampling noise shapers (ONS), as described here, first found its use in amplitude quantization of an analogue signal input. This assumes that no carrier or sideband harmonic frequency components exist within the frequency content of the input to the noise shaper. Within this chapter ONS is used for requantization of digital PWM signals. Carrier and sideband frequency components therefore do exist within the signal input of the noise shaper. It is not totally clear what effect the noise shaper has on these frequency components. What is known is that the noise shaping process causes a form of PWM foldback which results in baseband distortion [17].

It should also be noted that the noise shaper does not have the ability to attenuate correlated noise added to the digital input, since the filter  $H(z)$  is derived from the assumption that a statistically independent quantizer is used.

With this said, simulations of the noise shaper can now be done using  $H(z)$  to show the extent to which the PWM resolution can be reduced for feasible clock speed implementation.

### 5.3 Noise Shaping Simulations



The simulations that follow continue from Chapter 4 where 24-bit PNPWM was generated using a 1 kHz modulating wave and a trailing edge modulation scheme. The goal of the simulations used here is to show how the noise shaper has the ability to increase the SNR within the audio baseband when the high digital pulse width has been quantized to a lower resolution or bit size.

This section starts off by firstly describing the characteristics of a fifth order noise shaping filter derived from expressions presented in Section 5.2.6. This filter is then used to generate the lower resolution PWM output using different quantizer resolutions.

### 5.3.1 Noise shaping filter

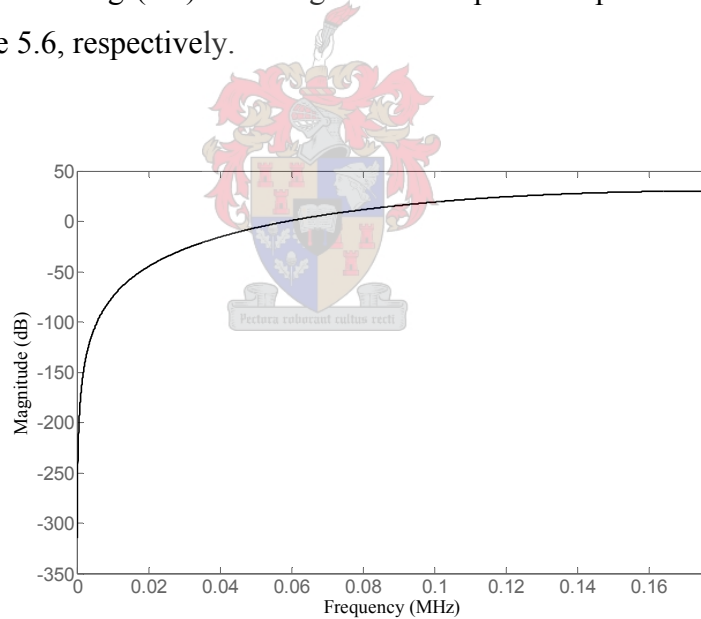
A fifth order noise shaping transfer characteristic is given by

$$NTF(z) = a_0 + a_1z^{-1} + a_2z^{-2} + a_3z^{-3} + a_4z^{-4} + a_5z^{-5} \quad (5.12)$$

with

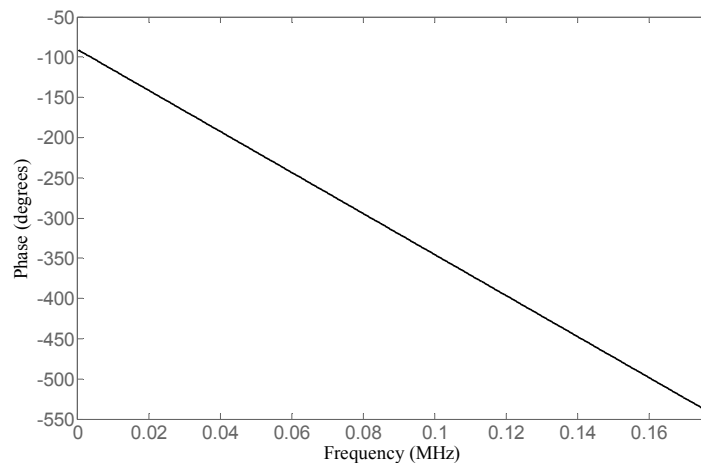
$$\begin{aligned} a_0 &= -1 \\ a_1 &= 5 \\ a_2 &= -10 \\ a_3 &= 10 \\ a_4 &= -5 \\ a_5 &= 1 \end{aligned} \quad (5.13)$$

calculated using (5.7). Its magnitude and phase response are given in Figure 5.5 and Figure 5.6, respectively.



**Figure 5.5: Magnitude response of fifth order noise transfer function.**





**Figure 5.6: Phase response of fifth order noise transfer function.**

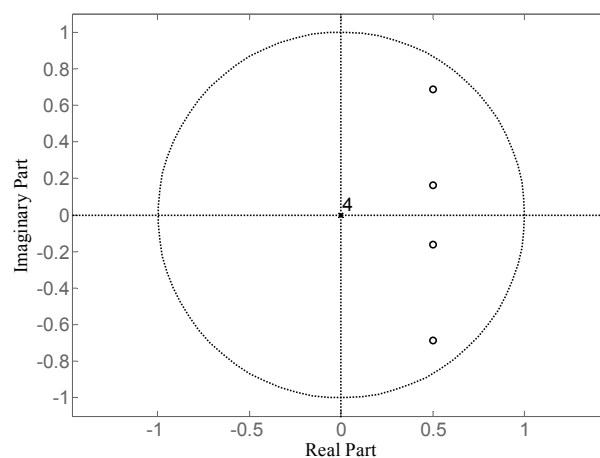
The magnitude response in Figure 5.5 agrees with the theoretical results given in Figure 5.4 derived using (5.9). The linear phase characteristic exhibited in Figure 5.6 ensures that no additional distortion is added to the filtered noise output.

Obtaining  $H(z)$  from  $NTF(z)$  yields

$$H(z) = a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4} + a_5 z^{-5}, \quad (5.14)$$

assuming that  $a_0$  is unity in (5.12). Figure 5.7 gives the pole zero plot of  $H(z)$ .

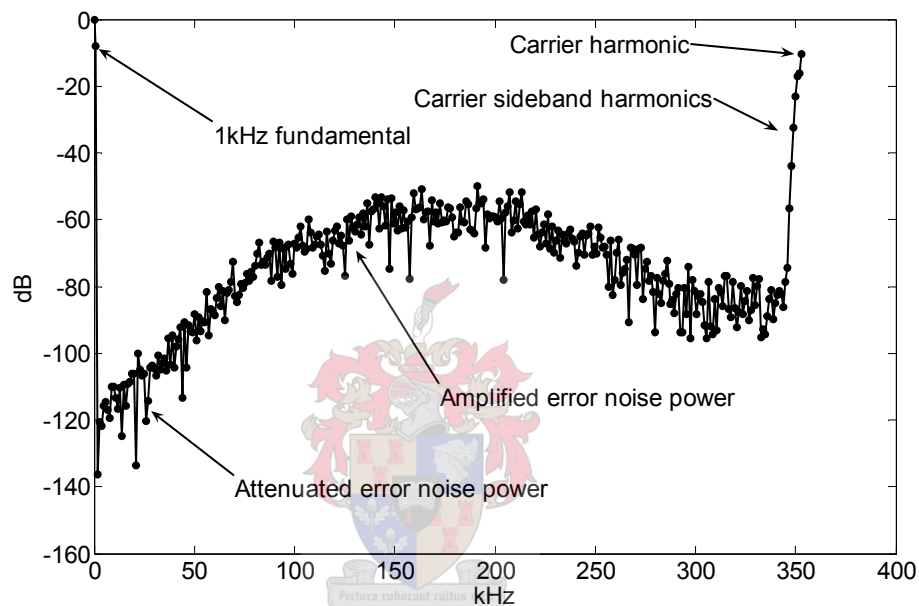
From this figure it is seen that all poles lie within the unit circle, therefore ensuring that the feedback loop of the noise shaping coder is stable.



**Figure 5.7: Pole/Zero plot of  $H(z)$ .**

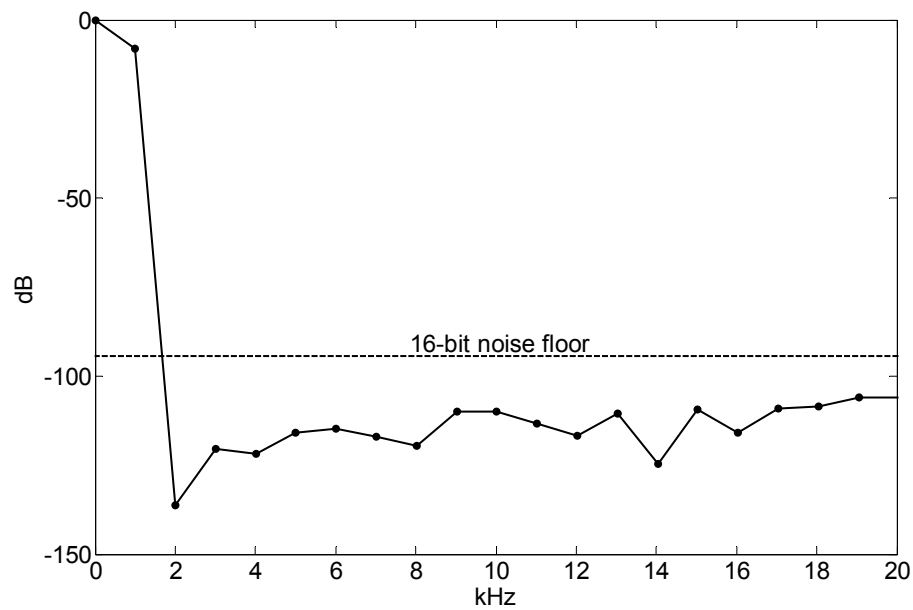
### 5.3.2 Noise shaping of the PNPWM output

The noise shaping process using the fifth order filter  $H(z)$  described above is now applied to the 24-bit PNPWM output calculated in Chapter 4. Figure 5.8 illustrates the noise shaped PWM output of the PNPWM widths using an 8-bit quantizer (Matlab ® code given in Appendix C2). It can be seen from this figure how the noise transfer function has displaced the baseband noise error power to the higher frequency band.



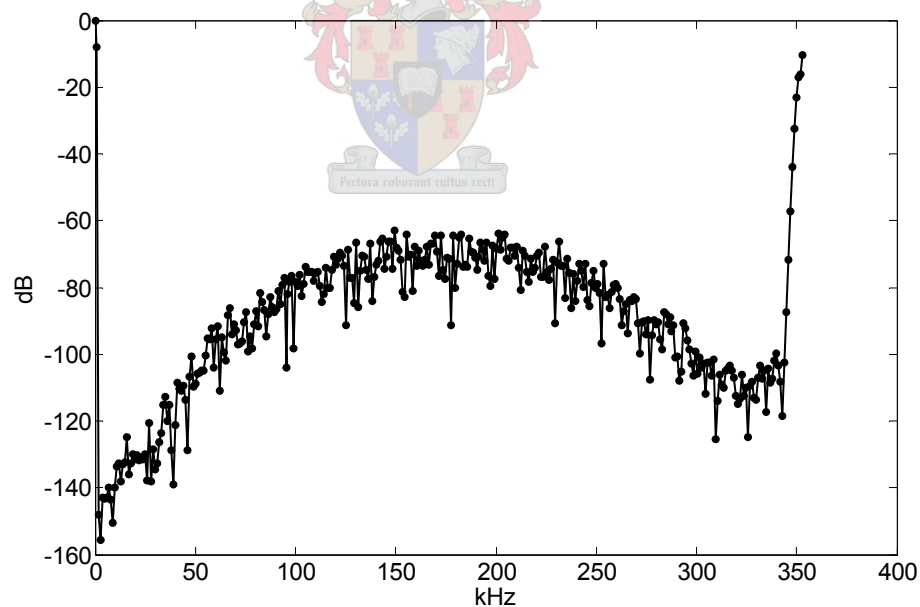
**Figure 5.8: Noise shaped 8-bit PWM output.**

Figure 5.9 shows a zoomed view of the audio band frequency content. It is seen from this figure that the noise floor falls below the 16-bit noise floor which is -96dB. The fifth order noise shaping filter is therefore sufficient to obtain 16-bit quality audio resolution from an 8-bit PWM output.



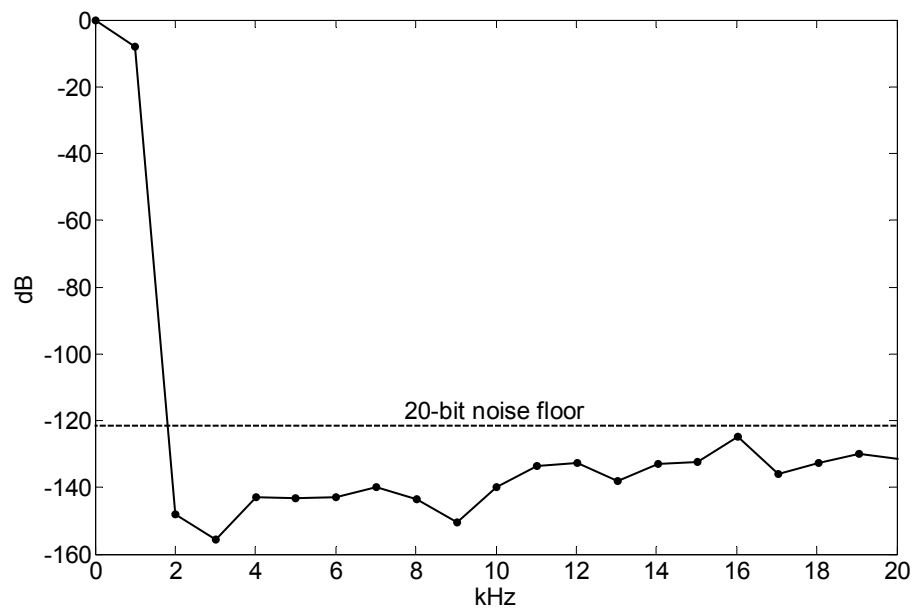
**Figure 5.9: Zoomed view of 8-bit PWM output.**

Figure 5.10 shows the PWM spectrum with the same fifth order noise shaping filter but with a 10-bit quantizer (Matlab<sup>®</sup> code given in Appendix C2).



**Figure 5.10: Noise shaped 10-bit PWM output.**

The zoomed view of the 10-bit PWM spectrum shows that an SNR of 120 dB has been attained within the audio baseband. This gives the PWM output a resolution of 20-bits.



**Figure 5.11: Zoomed view of 10-bit PWM output.**

From Table 5.1 it was concluded that only 8-bit and 10-bit PWM outputs were feasible with the current FPGA clock speed of 400 MHz. Here it has been shown that with these PWM output resolutions of 8-bit and 10-bit, audio resolutions of 16-bit and 20-bit can be attained respectively using a fifth order noise shaper.

Unfortunately 24-bit baseband resolution could not be attained using higher order noise shapers with 10-bit quantisers.

## 5.4 Summary

This chapter started by discussing different factors which influence the choice of the switching frequency for a digital pulse width modulator. It was concluded that a switching frequency of  $F_c = 352.8$  kHz suffices in the consideration of all these mentioned factors.

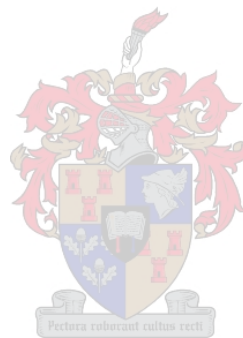
Next, it was shown that numerically calculated 24-bit PNPWM outputs could not be physically generated within current hardware because of clock speed constraints.

Noise shaping coders were then introduced, which provide a means to reduce the PWM bit output through a requantising process and a feedback filter. This structure increases the SNR within the baseband of the audio whilst moving the quantization noise power to an unused part of the bandwidth created by oversampling.

Simulations were then done using a noise shaping coder having a feedback filter order of five. This noise shaping coder was then used to generate PWM outputs of 8-bits and 10-bits respectively. The 8-bit PWM output attained a baseband resolution of 16-bits whereas the 10-bit PWM output achieved a baseband resolution of 20-bits.

Unfortunately 10-bit PWM outputs with a baseband resolution of 24-bits could not be achieved, therefore causing a bottle neck at this last output stage of the PCM to PWM modulator. The simulation results attained shows that high resolution PWM can be practically implemented within an FPGA exhibiting a maximum clock speed of 400 MHz.

Within the next chapter, the VHDL firmware implementation of the different block-sets of the digital PWM calculation will be given. These blocks include interpolation from Chapter 3, PNPWM from Chapter 4 and noise shaping coders from this chapter.



# Chapter 6 - Firmware Implementation

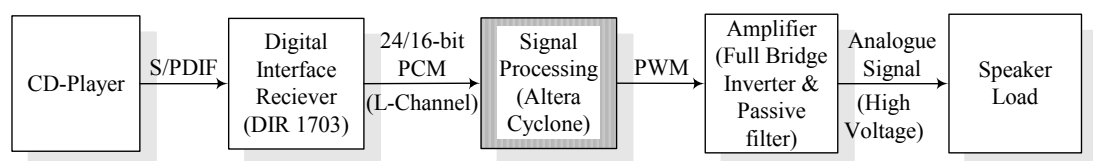
## 6.0 Introduction

Most of the premodulation building blocks needed to convert a PCM input signal to a digital PWM output signal have been completely described theoretically within the previous three chapters. Here these building blocks, consisting of interpolation, PNPWM and noise shaping will be translated into VHDL firmware, for implementation within an FPGA.

The chapter starts off by first describing the hardware system which comprises the digital audio amplifier. It then describes the VHDL firmware in two categories, namely: configuration firmware and algorithm firmware. After these firmware blocks have been described, a timing diagram of the implementation will be given, where after attention is then given to the synthesis involved in the firmware development. Next, fixed point arithmetic computation within the FPGA is investigated and some difficulties encountered with the firmware development are mentioned. Lastly the chapter shows how much resources are necessary to implement the described firmware for one digital audio channel.

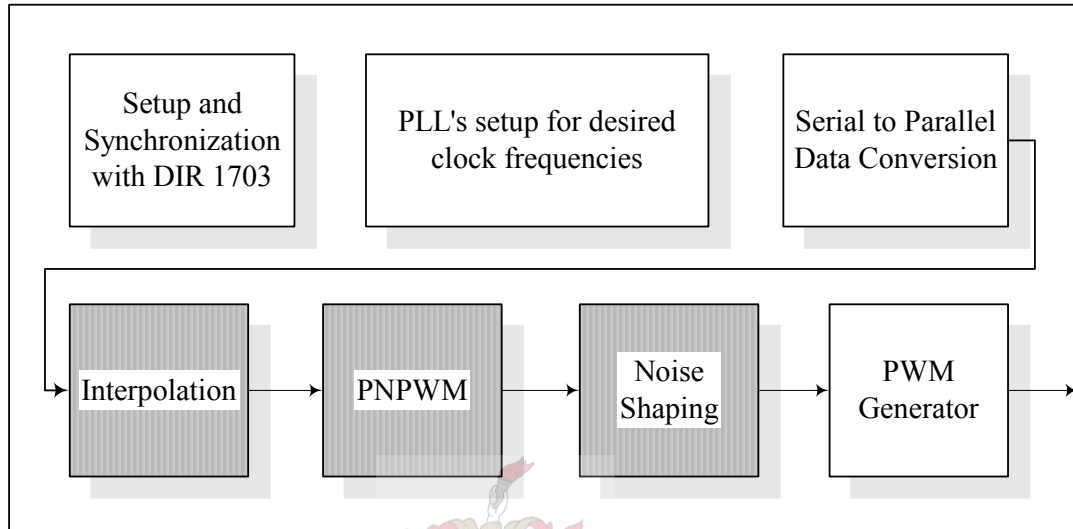
## 6.1 Hardware Description

The digital audio amplifier configuration is shown in Figure 6.1. The first block on the left represents the audio source which could either be a 24-bit or 16-bit resolution S/PDIF (Sony/Philips Digital Interface) output sampled at a rate of 44.1 kHz. The DIR (Digital Interface Receiver) converts the S/PDIF file transfer format into a PCM signal which is then converted to the digital PWM output using an ALTERA® CYCLONE FPGA. The PWM output gating signals are fed to a full bridge inverter which amplifies these signals into a speaker load.



**Figure 6.1: Signal processing building block for PCM to PWM conversion.**

The firmware developed in VHDL to implement the signal processing block (coloured grey) in Figure 6.1 comprises all of the practical work done within this thesis. Firmware blocks developed within the ALTERA® CYCLONE FPGA are shown in Figure 6.2. These various blocks will be extrapolated within the next section.



**Figure 6.2: Firmware blocks developed within the FPGA.**

## 6.2 Firmware Development

The firmware blocks given in Figure 6.2 are divided into two categories which have been developed during this study. The top row represents firmware blocks necessary for the configuration of resources, and the transformation of data, which are present within or at the disposal of the FPGA.

The bottom row represents the utilization of these resources and data to implement the digital pulse width modulation.

A third group of firmware is not shown in Figure 6.2 since it has not been developed. This firmware is known as megafunctions which are already available for use within the Quartus II software [26]. These Megafunctions form part of the developed blocks and will be described when used.

### 6.2.1 Configuration firmware

Three configuration firmware blocks are described here. The first block configures the digital interface receiver shown in Figure 6.2. The second sets up the necessary clock rates needed by the different PWM firmware blocks to execute. The

last block converts the serial bit data received from the DIR in the FPGA to 16-bit words.

#### 6.2.1.1 DIR 1703

The top left firmware block in Figure 6.2 sets up the TI (Texas Instruments) digital interface receiver (DIR 1703, Appendix D) to do the following:

- Reset it
- Set its sampling rate to  $F_s = 44.1$  kHz
- Set its audio bit clock to 22.5792 MHz which is output to one of the phase lock loops (PLL) of the FPGA
- Sets the data formatting to 16-bit, MSB first, right justified

The DIR 1703 therefore converts the S/PDIF audio input to a 16-bit PCM output serial stream sampled at 44.1 kHz. Unfortunately a 24-bit S/PDIF source was not available when implementing the firmware, but can be easily introduced by changing the configuration of the DIR 1703.

It is desired to use a 24-bit audio source in future work since all processes within the PWM firmware have 24-bit resolution, and to attain a high as possible audio-band bit quality at the noise shaping coder output.

#### 6.2.1.2 PLL setup

The various firmware blocks used to calculate the digital PWM need different clock frequencies. These clock frequencies are setup using the two PLLs of the ALTERA® CYCLONE FPGA. External clocks are coupled to each PLL respectively. The first is coupled to a 40 MHz external crystal oscillator, the second is coupled to the DIR 1703 BCKO clock output which has a clock frequency of 22.5792 MHz.

From these two PLLs four different clock frequencies are generated within the FPGA which service the different firmware or signal processing blocks within the second group in Figure 6.2.

The clock frequencies of the PLLs are setup using the GUI, PLL megafunction within the Quartus II software. The specific clock frequencies needed for specific processing blocks will be given in Section 6.2.3.2.



### 6.2.1.3 Serial to parallel data conversion

This firmware block receives the left audio channel serial bits from the DIR 1703 IC within a sampling period and converts them to a 16-bit word for latching within the FPGA. After the 16-bit word has been latched or stored it is used by the interpolation process.

Within the next section it will be shown how the latched 16-bit PCM word is used to implement a digital PWM output.

## 6.2.2 Digital PWM firmware

The second row of firmware blocks showed in Figure 6.2 implements the PCM to PWM conversion process within the FPGA. All of these blocks together form the PWM modulator. The grey coloured blocks in Figure 6.2, each refers to a chapter specifically devoted to the theory and functionality of it within the PWM conversion process.

Here the implementation of each of these blocks will be discussed separately in order of their chapter layout within this thesis. At the end of this section a timing diagram will be given of the firmware implementation.

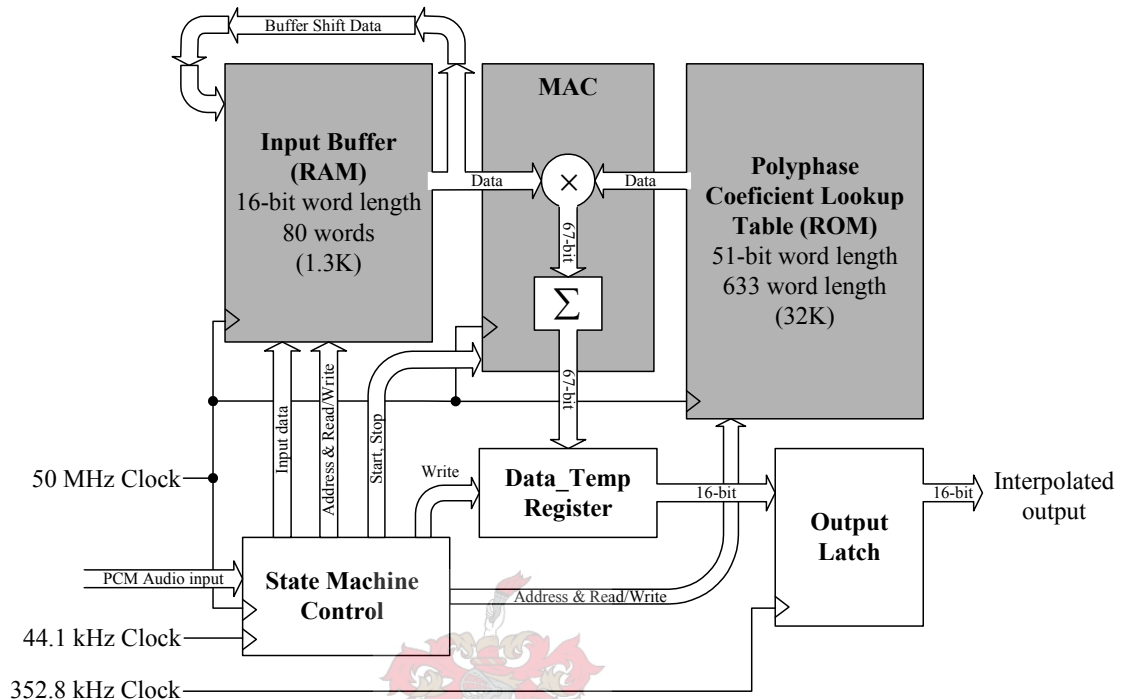
### 6.2.2.1 Interpolation

The interpolation process implements a FIR polyphase filter structure to up sample the input PCM audio signal. It has a total filter tap count of 632 and an upsampling rate of  $L = 8$ , implying eight polyphase filters each exhibiting a length of 79 taps.

Figure 6.3 illustrates the different components that make up the interpolation process, where the grey coloured blocks represent the Quartus II megafunction firmware. Please note that all figures that follow in this chapter that contain grey-shaded blocks continue to represent Quartus II megafunction firmware.

On the rising edge of the 44.1 kHz clock a 16-bit PCM value is stored at the second address of the input buffer, 78 previous samples are stored at the next consecutive addresses of the input buffer. The first address of the input buffer has a value of zero and this address is assigned by the state machine whenever the MAC

output needs to be retained (since a zero multiplication result will be added to the current output value).



**Figure 6.3: Blockdiagram of the interpolation process.**

The input buffer (1-PORT RAM megafunction) samples are then used to calculate eight output samples using the eight polyphase filters stored consecutively in the coefficient lookup table (1-PORT ROM megafunction). Each polyphase coefficient set is filtered with the input buffer sample values using the MAC (Multiply Accumulate) megafunction firmware. Before a 352.8 kHz rising edge occurs the specific polyphase filter completes and stores the output sample in a temporary register. When the 352.8 kHz rising edge occurs, the 16-bit interpolated value is latched for use by the next PNPWM process.

During the calculation of the last polyphase filter, all the sample values of the input buffer are shifted down to make space for the new PCM input occurring at the next 44.1 kHz rising edge. Before this 44.1 kHz edge occurs, the 8<sup>th</sup> polyphase filter output needs to be computed and stored within the temporary register.

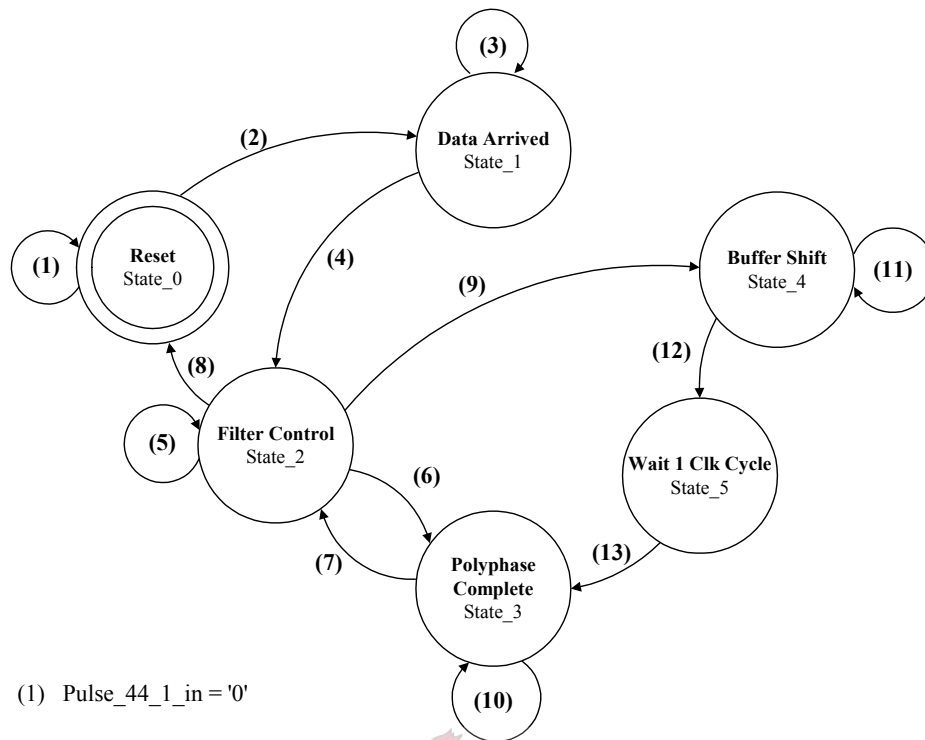
Figure 6.4 shows a state diagram of how the polyphase filtering process is controlled by the state machine. The VHDL code used to implement the interpolation

process is given in Appendix F1. Below the state diagram in Figure 6.4 the requirements for changing state are given for easy reference within the VHDL code.

The state machine starts at reset (`state_0`) and waits for the 44.1 kHz pulse. If it occurs, `state_1` is entered which stores the PCM input in the input buffer. When storage has completed, `state_2` is entered. This state implements the polyphase filtering and keeps track of which polyphase phase filter is currently busy executing. After a polyphase filter has completed `state_3` is entered where some time is used to wait until the MAC function has completed all its calculations where after the final value is stored within the temporary register.

`State_2` is entered again and after each polyphase filter `state_3` is entered to wait and store the interpolated output. The last polyphase filter executes within `state_4` since the buffer shift function needs to be performed while computing the polyphase filter. When the next 16-bit PCM sample is used for filtering it is overwritten by the previous, the last sample in the input buffer is therefore lost to make space for the new input sample.

After the input buffer data has been shifted, a one clock cycle delay is needed to finish the last polyphase filter where after `state_3` is entered again to store the final output in the temporary register. A second process reacting on the 352.8 kHz rising edge and synchronized with the 50 MHz clock, reads the temporary register and latches its value for use by the following process which will be described in the next section.



- (1) Pulse\_44\_1\_in = '0'
- (2) Pulse\_44\_1\_in = '1'
- (3) State1\_f\_sig = '1'
- (4) State1\_f\_sig = '0'
- (5) \*Buf\_adr\_sig < Buf\_cnt\_max
- (6) \*Buf\_adr\_sig = Buf\_cnt\_max
- (7) State3\_cnt\_sig = pf\_cnt\_max
- (8) Coef\_adr\_sig = Coef\_cnt\_max
- (9) Coef\_adr\_sig = (Coef\_cnt\_max - Buf\_cnt\_max - 1)
- (10) State3\_cnt\_sig < pf\_cnt\_max
- (11) State4\_f\_sig = '1'
- (12) Buf\_adr\_sig = Buf\_cnt\_max + 2
- \* When Coef\_adr\_sig < Coef\_cnt\_max
- Coef\_cnt\_max = 632
- Buf\_cnt\_max = 79

**Figure 6.4: State diagram of the interpolation process.**

6.2.2.2 PNPWM

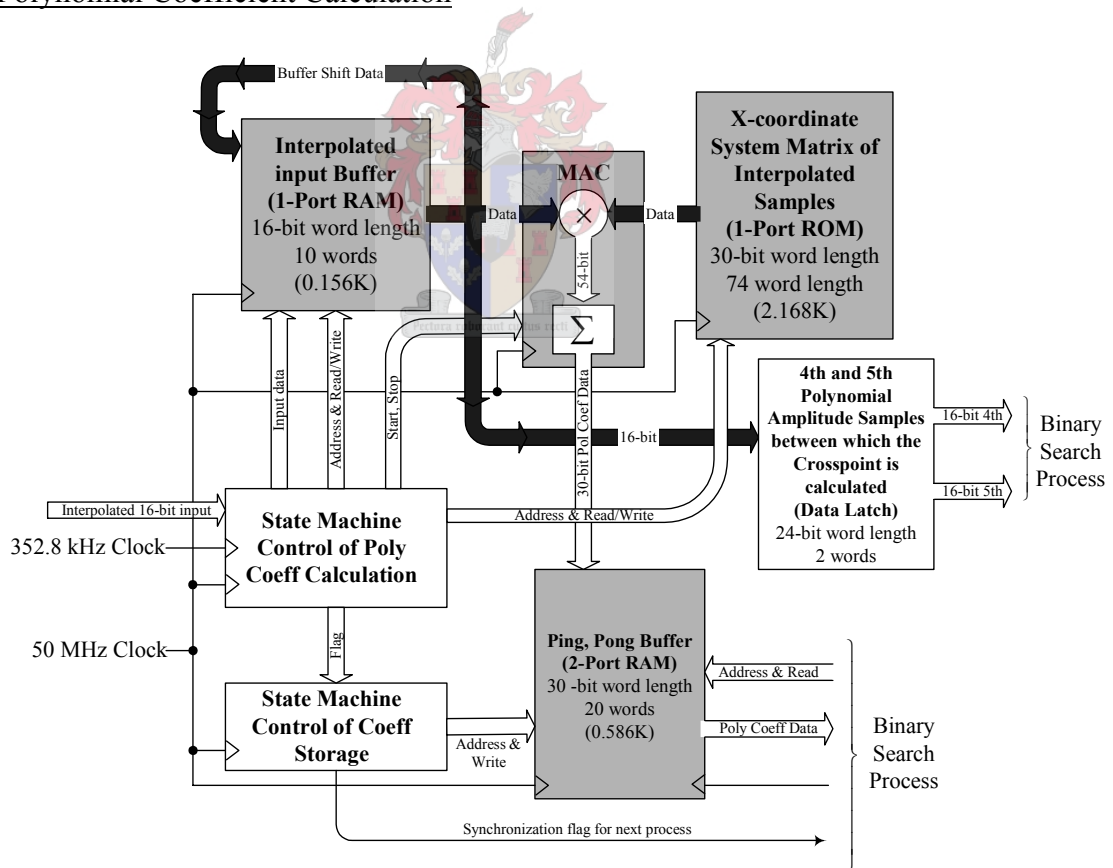
Here the process of calculating a 24-bit PWM signal as described in Chapter 4 will be implemented within VHDL firmware. The description of this implementation will be done within three subsections.

The first section will show how the polynomial coefficients are calculated using the interpolated input data.

Secondly, the binary search algorithm implementation will be extrapolated which finds the crosspoint within a 9-bit resolution or a first tolerance interval.

Lastly the crosspoint calculation implementation will be done by linear interpolation within the first tolerance interval and then a binary search algorithm is used again to obtain a 24 bit PWM output.

Polynomial Coefficient Calculation



**Figure 6.5: Block diagram of polynomial coefficient calculation.**

Figure 6.5 gives a graphical view of how the VHDL firmware which is used to compute the 8<sup>th</sup> order polynomial coefficients. A new interpolated sample is stored

within the input buffer at a frequency of 352.8 kHz and the previous samples are shifted down to make space for the new sample to be stored. The input buffer represents nine polynomial amplitude values, and their respective polynomial coefficient values are calculated by the multiplication given in (4.40).

The system matrix of (4.40) is stored in memory as a lookup table and illustrated in Figure 6.5. The multiplication between the amplitude samples and the system matrix is done using a MAC megafunction. After the coefficients have been calculated they are stored within a ping-pong buffer.

While one set of coefficients are stored within one part of the buffer, the previous set is stored in the other half of the buffer and are read by the binary search process, hence the expression ping-pong buffer. Therefore within one 352.8 kHz clock period one set of coefficients are calculated.

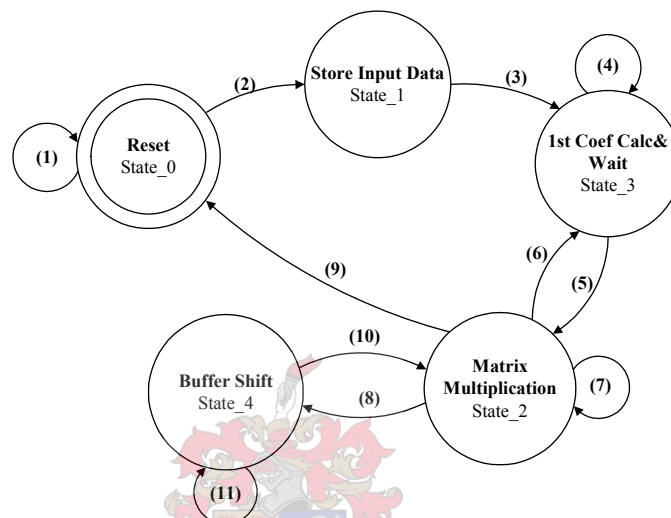
The fourth and fifth polynomial amplitudes are also latched out at this rate since the crosspoint is calculated within this switching interval as described in Chapter 4, and is also needed by the binary search process to find the first tolerance interval.

The state diagram that controls the polynomial coefficient calculation is shown in Figure 6.6. The state transition conditions are also given below the state diagram for easy referral to the VHDL code which is given in Appendix F2. The polynomial coefficient calculation is similar to the filtering process and will now be explained using the state diagram.

The reset state (state\_0) in Figure 6.6 waits for a 352.8 kHz rising edge, when it occurs, states are changed and the new interpolated data sample is stored within the input buffer in state\_1. After one 60 MHz clock cycle state\_3 is entered. This state computes the first polynomial coefficient. Only one multiplication arithmetic is necessary to compute the first coefficient since the first column of the first row in (4.40) exhibits the only non-zero value.

Because of this, the first coefficient is calculated in state\_3. After the first calculation is completed, state\_2 is entered where the rest of the polynomial coefficients are calculated. When the last coefficient needs to be calculated the buffer shift state (state\_4) is entered. Here both the last coefficient calculation and the input buffer shift functions are accomplished. Whenever a coefficient calculation has been completed state\_2 sets a flag which is discerned by a second state machine given in Figure 6.7.

This state machine stores the coefficient value in the appropriate address within the ping-pong buffer. It keeps record of which half is currently used for writing and which half is currently being read. When the first complete set of coefficients has been stored, a flag is set to indicate to the binary search process that it can start on the next 352.8 kHz rising edge clock. Whenever a coefficient has been stored the state machine returns to state\_0, waiting until the next coefficient value has been calculated.



- (1) pulse\_352\_in = '0'
- (2) pulse\_352\_in = '1'
- (3) No condition
- (4) (\*state3\_cnt\_sig < 7) or (\*\*state3\_cnt\_sig < wait\_cnt\_max)
- (5) (\*state3\_cnt\_sig = 8) or (\*\*state3\_cnt\_sig = wait\_cnt\_max)
- (6) \*\*\*Buf\_adr\_sig = Buf\_cnt\_max
- (7) \*\*\*Buf\_adr\_sig < Buf\_cnt\_max
- (8) \*\*\*Mat\_adr\_sig = (Mat\_cnt\_max - Buf\_cnt\_max)
- (9) Mat\_adr\_sig = Mat\_cnt\_max
- (10) Buf\_adr\_sig = Buf\_cnt\_max + 2
- (11) Buf\_adr\_sig < Buf\_cnt\_max + 2
- \* Only when frst\_coef\_f\_sig = '0'
- \*\* Only when frst\_coef\_f\_sig = '1'
- \*\*\* Only when Mat\_adr\_sig < Mat\_cnt\_max
- wait\_cnt\_max = 6
  - Mat\_cnt\_max = 73
  - Buf\_cnt\_max = 9

**Figure 6.6: State diagram of the polynomial coefficient calculation.**

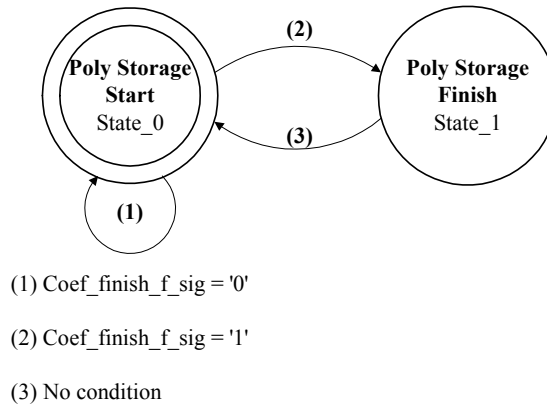


Figure 6.7 State diagram of the polynomial coefficient storage.

Binary search

The previous section explained how the polynomial coefficients are calculated using VHDL firmware. With the polynomial coefficients calculated it is now necessary to find the first interval wherein the crosspoint between the polynomial and comparison waveform is located. This is done using the binary search strategy explained in Chapter 4. Figure 6.8 shows a map of how this is accomplished within VHDL firmware. This process will now be explained.

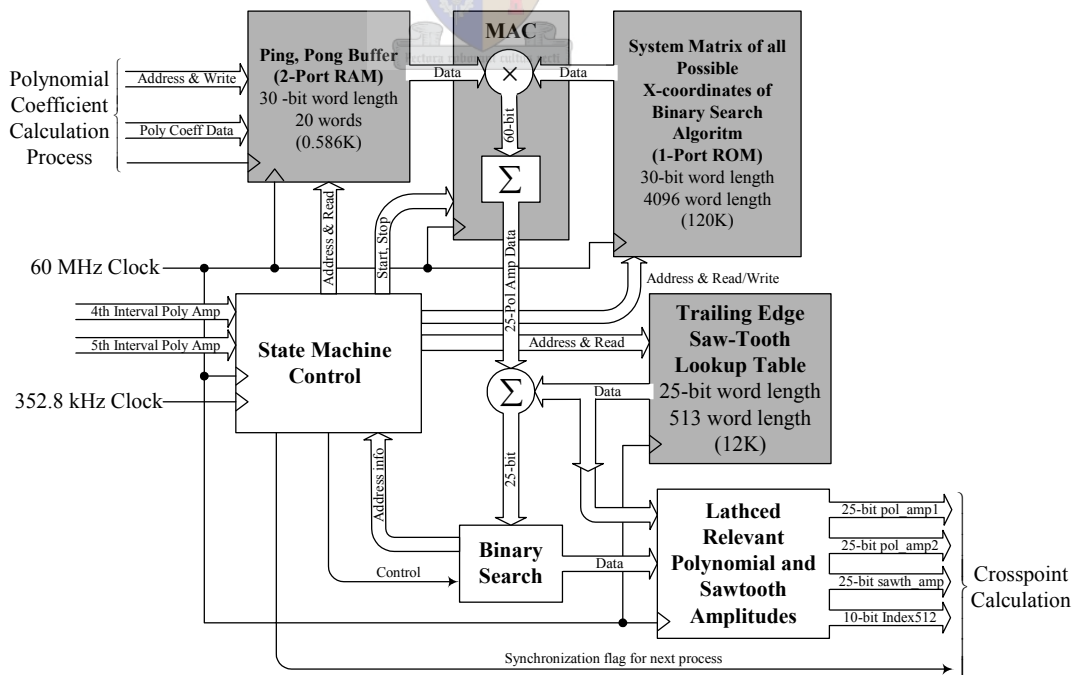


Figure 6.8: Block diagram of binary search within first tolerance interval.



An initial guess of the crosspoint is made at the position halfway between the fourth and fifth polynomial amplitude positions which represent the switching interval. There are 512 time positions, since the first tolerance interval searches the crosspoint time to an accuracy of 9-bits ( $2^9$ ).

The initial guess is thus chosen to be 256. The amplitude of the polynomial is calculated at this initial guess and then compared to the trailing edge sawtooth waveform. The result of this comparison determines a new position which falls within the 512 value set. This new position value halves the previous interval therefore moving closer to the crosspoint interval. The polynomial amplitude of this position is calculated again and the process repeats until nine comparisons have been made, after which the polynomial amplitudes left and right of the searched interval is output. Also, the sawtooth amplitude located at the left polynomial amplitude is output for crosspoint calculation within the next process.

From Figure 6.8 the polynomial amplitude is calculated by multiplying the stored polynomial coefficients in the ping-pong buffer with the x-coordinate system matrix lookup table. This multiplication process is represented by (4.48). The polynomial amplitude calculated is then compared by the appropriate sawtooth amplitude read from a lookup table. From the comparison the binary search firmware determines if the search is complete. If not, it determines the new address information which is then used by the state machine for the next polynomial amplitude calculation. If the search has been completed the relevant amplitude data is latched for use by the next process.

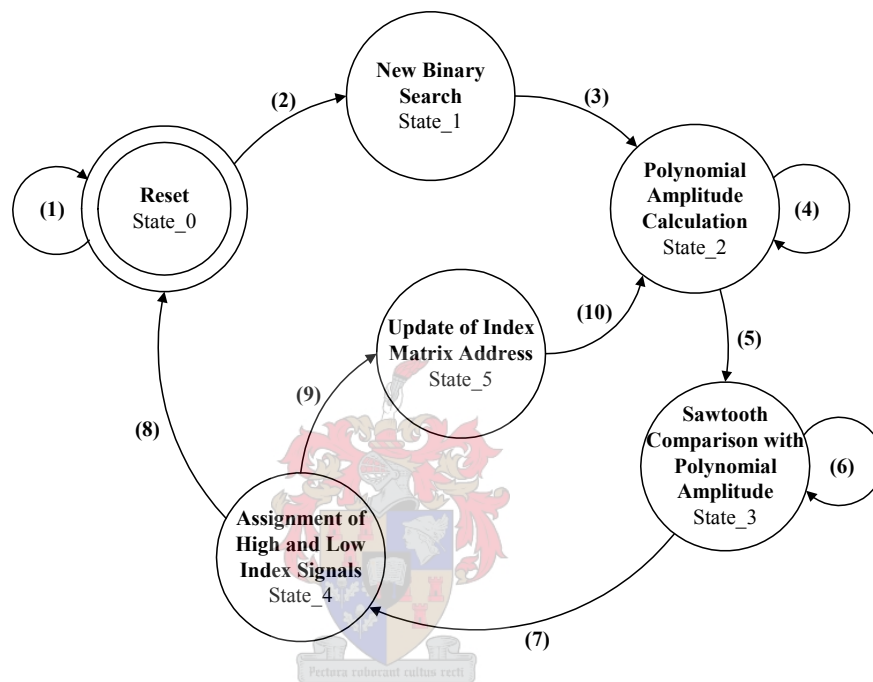
Figure 6.9 shows the state diagram which implements the binary search process (its VHDL code is given in Appendix F3).

In the reset state a rising edge of the 352.8 kHz clock and a high value from the synchronization flag from the previous process causes state\_1 to be entered. Within this state the initial crosspoint position guess is assigned where after state\_2 is entered which calculates the polynomial amplitude. State\_3 is then entered which waits until the MAC completes its calculation. The result of this calculation is then compared to the sawtooth amplitude which is located at the same position as the calculated polynomial amplitude.

When the comparison has completed, state\_4 is entered which decides whether the binary search has completed. If it has, the reset state is entered. If it has not,

state\_5 is entered where relevant memory addresses are updated for the next polynomial amplitude calculation and sawtooth wave comparison.

When the 9-bit interval has been found, the relevant polynomial amplitudes and sawtooth amplitudes are latched to the output in state\_4. The final 9-bit index value is also latched to the output for the final 24-bit PWM calculation. A synchronization flag is set for the next process to know when this relevant data has been latched for use.



- (1)  $[(\text{pulse\_352\_in} = '0') \ \& \ (\text{Pol\_coef\_sync\_in} = '0')]$ , or  $[(\text{pulse\_352\_in} = '1') \ \& \ (\text{Pol\_coef\_sync\_in} = '0')]$ , or  $[(\text{pulse\_352\_in} = '0') \ \& \ (\text{Pol\_coef\_sync\_in} = '1')]$
- (2)  $(\text{pulse\_352\_in} = '1') \ \& \ (\text{Pol\_coef\_sync\_in} = '1')$
- (3) No condition
- (4)  $[\text{state2\_f\_sig} = '0']$  or  $[(\text{state2\_f\_sig} = '1') \ \& \ (\text{Mult\_adr\_sig} < 7)]$
- (5)  $(\text{state2\_f\_sig} = '1') \ \& \ (\text{Mult\_adr\_sig} = 8)$
- (6)  $\text{state3\_cnt\_sig} < 6$
- (7)  $\text{state3\_cnt\_sig} = 6$
- (8)  $(\text{High\_var} - \text{Low\_var}) = 1$
- (9)  $(\text{High\_var} - \text{Low\_var}) > 1$
- (10) No condition

**Figure 6.9: State diagram of the binary search process.**

### Crosspoint calculation

This section concludes the final stage of the 24-bit PWM calculation. From the binary search process four data values have been latched to calculate the PWM output. Firstly, two polynomial amplitudes are available which are used to calculate a linear equation approximating the polynomial (audio wave) within the second tolerance interval. These two amplitudes are necessary to calculate the linear equation gradient using (4.46). The trailing edge sawtooth waveform is also calculated using the latched sawtooth amplitude. Its gradient is known within the interval, and therefore its calculation is simple. The 9-bit index value which has also been latched is used to calculate the 24-bit output.

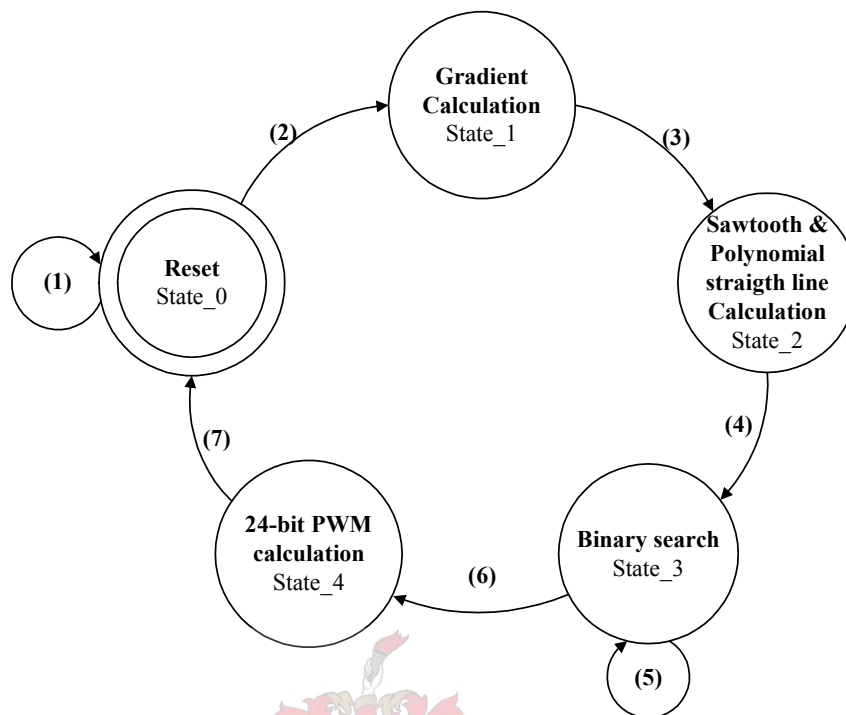
Figure 6.10 illustrates the state diagram which calculates the 24-bit PWM output within VHDL firmware. Within this process, no megafunctions are used. The VHDL code implementing this process is given in Appendix F4.

State\_1 is entered when the synchronization flag from the previous state is high and a 352.8 kHz rising edge occurs. Within state\_1 the gradient of the straight line polynomial approximation is calculated and the initial binary search index is assigned for the second tolerance interval. This index variable has a range of  $2^{15} = 32768$ . Therefore when the second tolerance interval has completed its binary search, a 24-bit PWM value can be attained. This is due to the first binary search attaining a 9-bit resolution where after the second tolerance interval attains a 15-bit search resolution.

After one 60 MHz clock cycle state\_1 is finished and state\_2 is entered which calculates both the sawtooth and straight line polynomial equation values at the current index value.

Then state\_3 is entered and a new index value is calculated according to the binary search algorithm. With the new index value attained the algorithm tests to see if the search has completed. If it has, state\_4 is entered, and if it has not, state\_2 is entered again. Here the new index value is substituted in the sawtooth and polynomial linear equation to gain new amplitude values which will be compared at the next binary search in state\_3. This iteration repeats itself until the interval to be searched reaches a width of one. When this occurs as mentioned, state\_4 is entered. The 15-bit second tolerance interval index and the 9-bit first tolerance interval index value is then used together to calculate a 24-bit PWM output. This PWM output value is then

latched for the use of the next process which reduces its resolution through a noise shaping coder.



(1) [(pulse\_352\_in = '0') & (sync\_in = '1')], or  
[(pulse\_352\_in = '0') & (sync\_in = '1')], or  
[(pulse\_352\_in = '1') & (sync\_in = '0')]

(2) (pulse\_352\_in = '1') & (sync\_in = '1')

(3) No condition

(4) No condition

(5) (High\_var - Low\_var) > 1

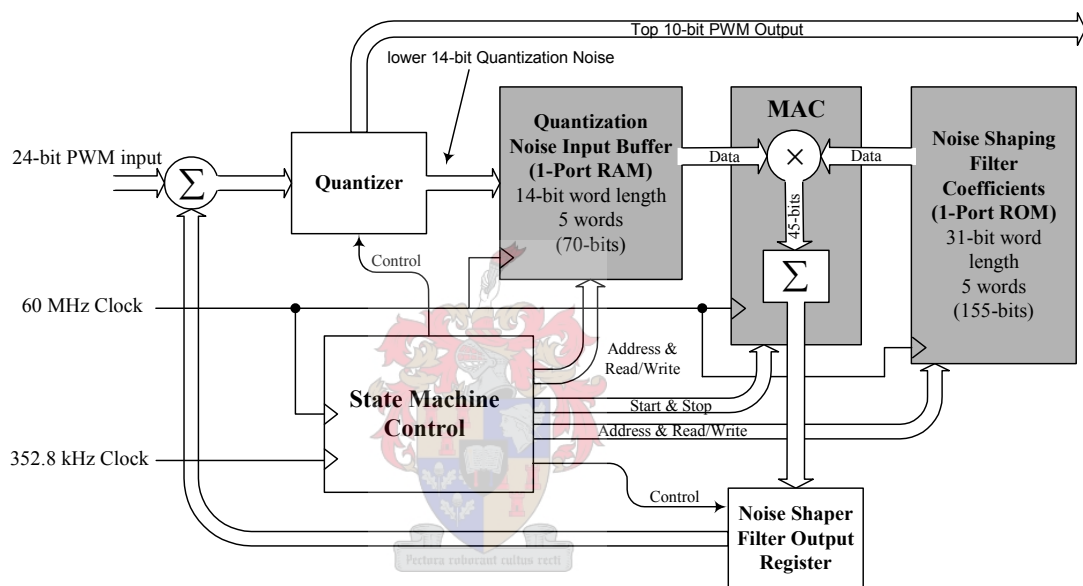
(6) (High\_var - Low\_var) = 1

(7) No condition

**Figure 6.10: State diagram of final crosspoint calculation.**

### 6.2.2.3 Noise Shaping

The last signal processing block necessary to convert a PCM signal to a practical PWM output signal is described here within VHDL firmware. The physical VHDL code of this firmware is given in Appendix F5. The 24-bit PWM output calculation accomplished within the previous PNPWM firmware is now reduced in bit size through the use of a noise shaping coder. Figure 6.11 gives a block diagram layout of the complete noise shaping process.

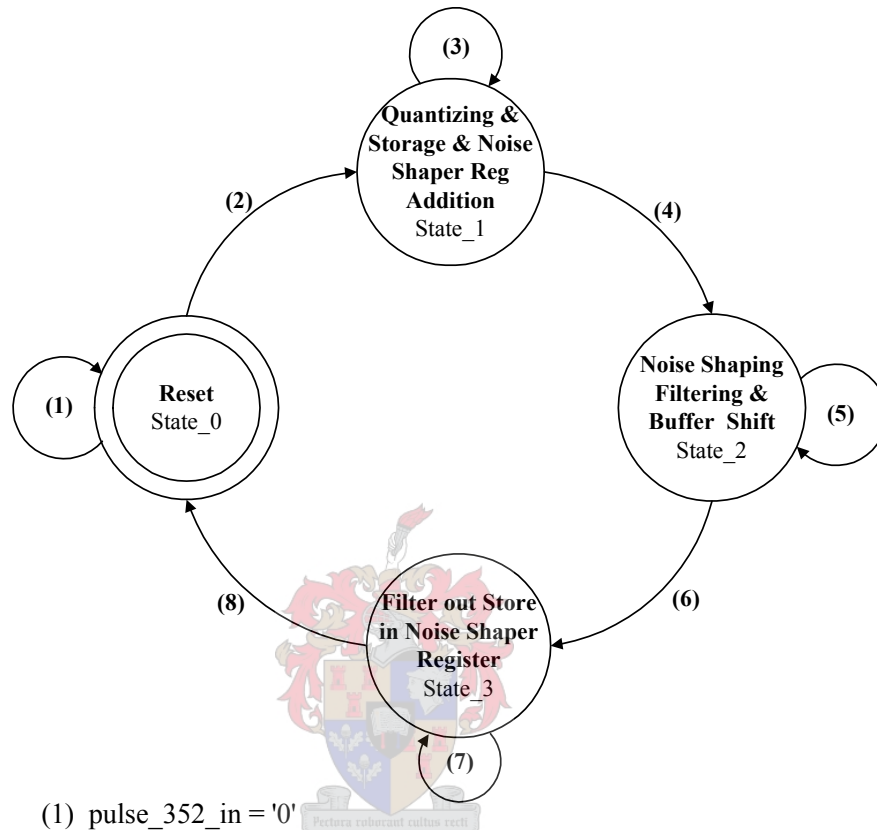


**Figure 6.11: Block diagram of noise shaping process.**

From the above figure it can be seen that the 24-bit PWM input signal is quantized after the addition node through truncating the 14 least significant bits (LSB). The top 10 most significant bits (MSB) are output to the full bridge converter for amplification.

It was shown in Chapter 5 that 20-bit audio quality is achievable when using a 10-bit PWM resolution. The 14 LSB are stored within a buffer. This buffer is then used for filtering, much the same as described previously with the interpolation firmware implementation. A MAC megafunction is used to perform the multiplication and addition arithmetic between the quantization noise buffer and the noise shaper filter coefficients.

Chapter 5 described a fifth order noise shaping filter whose coefficients are stored within the coefficient lookup table configured using a 1-Port ROM megafunction. The noise shaped filtered result is stored within a register which is then fed back and added to the 24-bit PNPWM input.



- (1) pulse\_352\_in = '0'
- (2) pulse\_352\_in = '1'
- (3) State1\_f\_sig = '0'
- (4) State1\_f\_sig = '0'
- (5) Buf\_adr\_sig < Buf\_adr\_max + 2
- (6) Buf\_adr\_sig = Buf\_adr\_max + 2
- (7) State3\_cnt\_sig < 5
- (8) State3\_cnt\_sig = 5

**Figure 6.12: State diagram of noise shaping process.**

Figure 6.12 shows the state diagram of the state machine which is implemented to control the noise shaping process. Again a reset state waits until a 352.8 kHz rising clock edge is detected. When it has, a 24-bit PNPWM result is available for noise shaping, resulting in the entrance of state\_1. Within state\_1 the previous noise shaping filter ( $H(z)$ ) output is added to the current 24-bit PNPWM input and then quantized. As described above, quantization is performed through capturing the top 10-bits of the added result. The lower 14-bits are also stored in a buffer within state\_1. When quantization and storage has completed, state\_2 is entered where the feedback noise shaping filter is executed. While the filter is executing the input buffer data is shifted down to make space at the first buffer address for the next quantization noise value to be stored.

Upon filter completion, state\_3 is entered which scales and stores the filter output within a register for addition with the next 24-bit PNPWM input at the rising edge of the next 352.8 kHz clock.

#### 6.2.2.4 PWM Generator

The 10-bit PNPWM data value obtained from the noise shaping process needs to be converted to a physical pulse width. From Table 5.1 it is seen that a clock frequency of 361 MHz is needed to generate a physical 10-bit PNPWM signal. Although the ALTERA® CYCLONE FPGA can attain a maximum clock frequency of 400 MHz, a 10-bit PWM generator which was implemented within VHDL could not run at needed 361 MHz. This was because the physical logic generated from the VHDL code on the FPGA could not execute at this fast clock frequency. This was evaluated by the Quartus II timing analyser [28]. An 8-bit PNPWM was rather implemented in VHDL firmware executing at a clock frequency of 90.317 MHz. This implies that only 16-bit audio output quality is attainable from the simulations presented in Chapter 5. A description of this generator will now be given

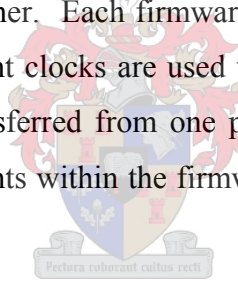
The PWM generator consists of a counter whose maximum value is assigned the 8-bit PWM input value for a specific switching interval. The counter therefore starts at zero and increments at a rate of 90.317 MHz. As soon as the counter reaches its maximum assigned value, the logic output goes low, therefore ending the current high output pulse duration. The output then stays low until the switching interval has

ended. At the start of the next switching interval the logic output becomes high again until the counter has reached its new maximum value which represents the pulse width. This form of PWM generation is known as trailing edge modulation as was thoroughly described in Chapter 4. The VHDL firmware implementing this PNPWM generation process is given in Appendix F6.

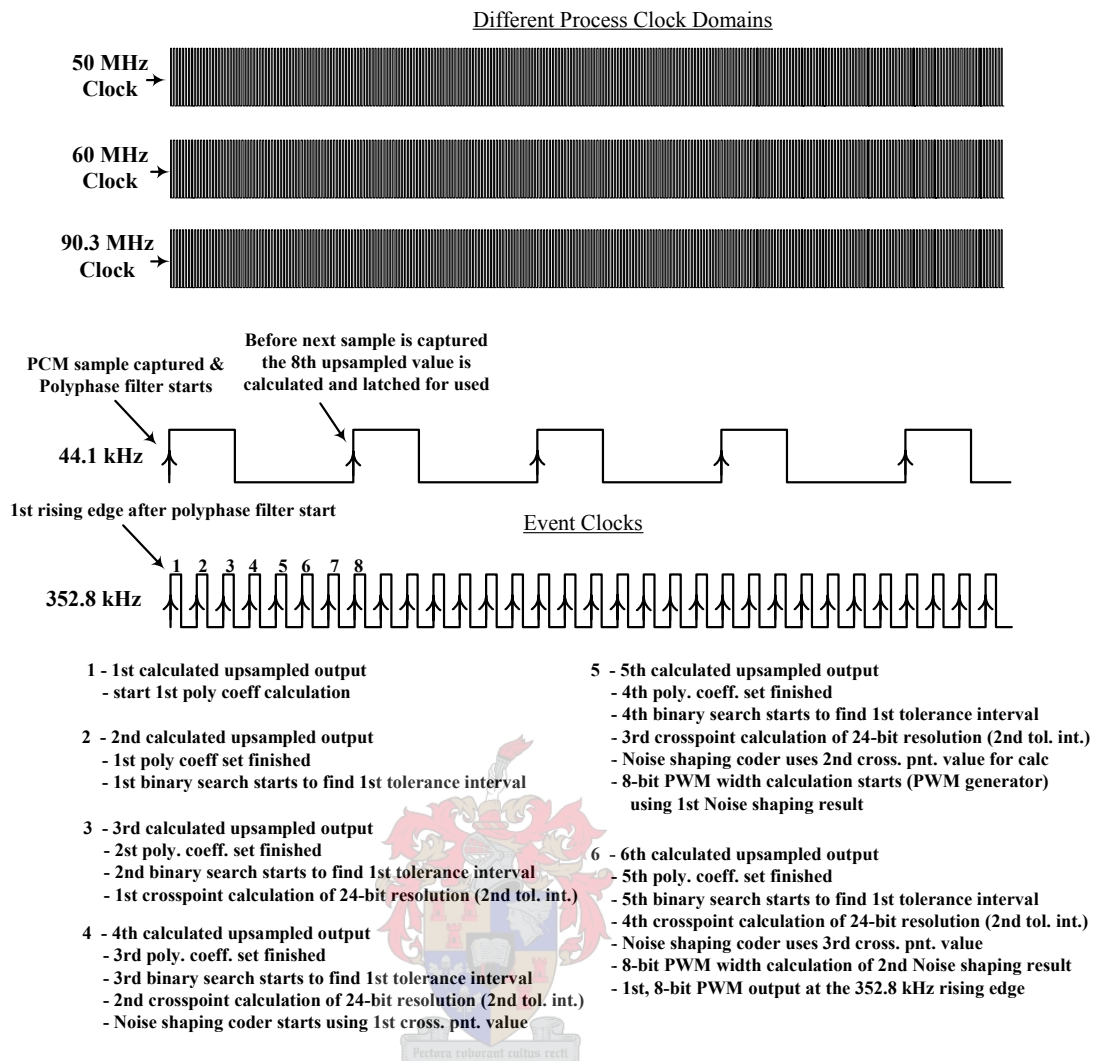
Most important processes have now been described which work together to produce the desired PWM output. These processes use state machines within the VHDL firmware to control them. Different clock domains also exist between these different processes and it is thus necessary to synchronize control signals and data between them. In the next section a timing diagram description of how these firmware blocks work together will be given.

#### 6.2.2.5 Timing diagram

Figure 6.13 shows a timing diagram representation of how the different firmware blocks work together. Each firmware block executes at its clock rate or in its clock domain. Two event clocks are used which determine when processes start, when data needs to be transferred from one process to the other, or when the data needs to be output. All events within the firmware processes occur at the rising edge of a clock.







**Figure 6.13 Timing diagram description of firmware.**

Therefore when a PCM sample is available for processing it is captured by the polyphase filtering process as described previously. After six 352.8 kHz clock cycle delays ( $17\mu s$ ) the first 8-bit PWM calculated value is output. This is the total execution time needed for the digital processing to calculate a PWM output. With the negligible delay, pulse width value outputs are therefore generated real-time.

Figure 6.13 shows and describes which firmware block is active during each 352.8 kHz clock count until the PWM signal is output.

The next section describes how state machines are implemented within the VHDL firmware to control the processes which comprise the PWM modulator, and how these different clock domains can be synchronized among them.

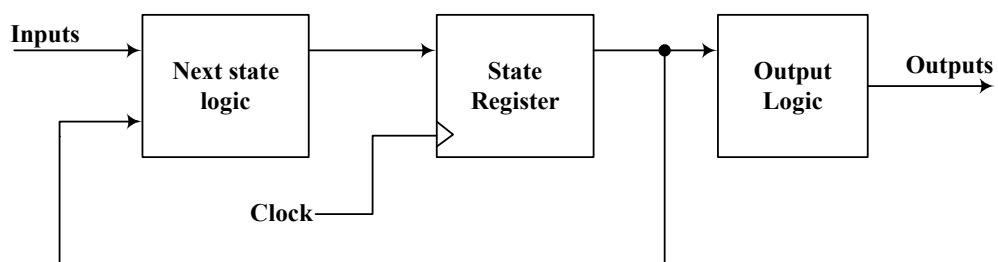
### 6.2.3 VHDL synthesis

Synthesis is the process of building up separate elements; these elements within the context of this thesis are the different firmware blocks described above. The firmware blocks are controlled by state machines and their structure will be described here. The firmware blocks also use different clock frequencies; the communication accomplished between these asynchronous clock domains will also be described.

#### 6.2.3.1 State Machines

All the state machines implemented within this thesis are called Moore state machines [6]. Figure 6.14 illustrates the architecture of such a state machine which is an example of a synchronous sequential system. Sequential systems are those who change state because of past input values. The present state of the system can either be updated as soon as the next state changes, in which case the system is said to be asynchronous, or the present state can be updated only when a clock signal changes, which is synchronous behaviour [6].

The Moore machine is triggered by a single clock for synchronization. Its next state is determined by some (combinational) function of the inputs and the present state.



**Figure 6.14: Mealy Machine [6].**

The Moore machines implemented within the current VHDL firmware described in the previous sections use two processes. The first process represents the middle block within Figure 6.14. It is therefore sensitive to the clock input and only assigns the next state or output signals when a rising edge on the clock occurs or when the process is in reset. The second process represents the first block within Figure 6.14. It contains the combinational logic which decides what the value or state will be

of the registers at the following clock edge. Figure 6.15 shows a VHDL example of a two-process state machine [27].

```

1 ns: PROCESS (car, timed, present_state) IS
2 Begin
3   Case present_state IS
4     When state_0 =>
5       IF (car = '1') THEN
6         next_state <= state_0;
7       ELSE
8         next_state <= state_1;
9       END IF;
10    When state_1 =>
11      IF (timed='1') THEN
12        next_state <= state_1;
13      ELSE
14        next_state <= state_0;
15      END IF;
16    END CASE;
17 END PROCESS ns;
18
19 PROCESS (clk, reset)
20 BEGIN
21   IF reset = '1' THEN
22     state <= state_0;
23   ELSIF rising_edge (clk) THEN
24     present_state <= next_state;
25   END IF;
26 END PROCESS;

```

**Figure 6.15: Example of a two-process Moore state machine.**

The top process in the above example is the combination logic which determines the next state. It can be seen that the two input signals (car and timed) determine which state will be entered next within a specific state. The second process in the example assigns the result of the combination logic state calculation at the rising edge of the clock signal to the present state register. The asynchronous reset signal is the only other signal to which this process is sensitive and is only used to assure the correct state is entered at startup of the firmware. Two-process state machines assures that the Quartus II development software interprets the VHDL code correctly and assures that hazards do not occur which can cause the wrong state to be entered [27].

### 6.2.3.2 Asynchronous Clock Domains

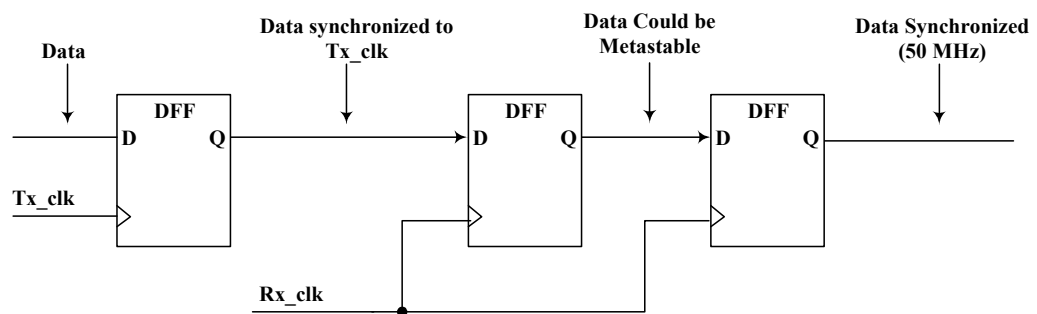
Different clock domains exist among the various firmware processes described in Section 6.2.2.5 which are now listed in Table 6.1. A design subsection, driven by a single clock source is called a clock domain [16]. The frequency and phase of each

clock source can be different from the rest. If two clock signals do not have a synchronous, or fixed, relationship, they are asynchronous to each other. For instance, the interpolation process represents one clock domain, while the PNPWM and noise shaping processes represent another. A third clock domain is represented by the PWM generation process. These various clock domains are asynchronous toward each other.

<b>Firmware Process</b>	<b>Clock Speed</b>
Interpolation	50 MHz
PNPWM	60 MHz
Noise Shaping	60 MHz
PWM Generation	90.317 MHz

**Table 6.1: Clock speeds of different processes.**

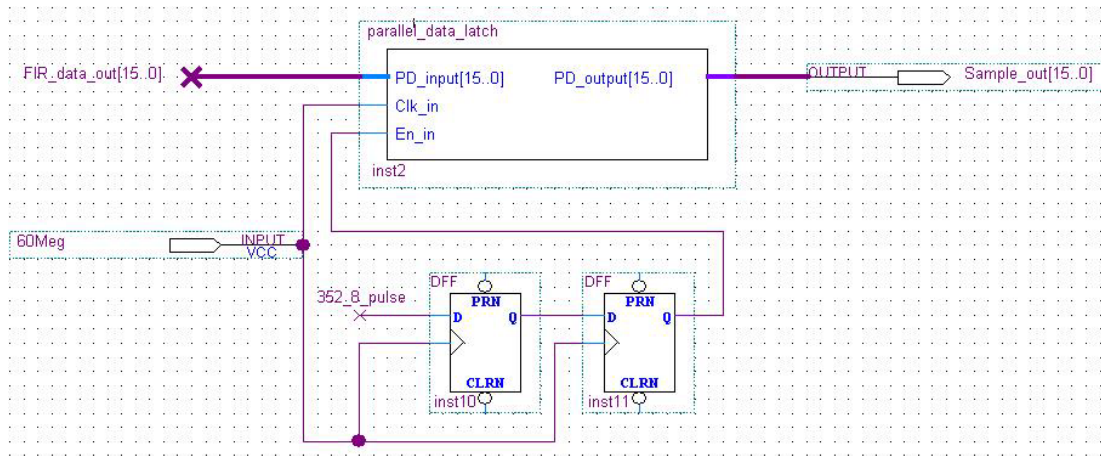
From the description of the above digital PWM modulator design it is obvious that data and signals are read and written between different clock domains. These signals and data cannot be directly read when sent from a process representing a different clock domain. If two asynchronous clock domains need to communicate with each other, some consideration needs to be given to how this operation can be performed reliably. Figure 6.16 shows a double synchroniser for simple bit data transfer consisting of a 2-bit shift register structure clocked by the receiving clock [27].



**Figure 6.16: A double synchroniser circuit.**

The second stage of the shift register reduces the probability of metastability (unknown state) on the data output from the first register propagating through to the output of the second register. The data from the transmitting clock domain should come directly from a register. All 1-bit signals transmitted from one clock domain to another within the VHDL firmware implemented within this thesis are synchronised

using the double synchroniser circuit. Figure 6.17 shows an example of how a data bus is synchronised between asynchronous clock domains within the implemented VHDL firmware described previously.



**Figure 6.17: Data bus synchronisation between asynchronous clock domains.**

The FIR\_data\_out[15..0] data bus (Figure 6.17) result is a latched output from a 50 MHz process which is kept unchanged for a couple of 50 MHz clock cycles. At the rising edge of the 352.8 kHz pulse which is synchronised to the 60 MHz data latch, the data bus result is stored within the 16-bit latch. This stored data value is then available at the output of the latch at the next rising edge of the 60 MHz clock. The next process which reads the latch output is synchronised at 60 MHz, therefore assuring data integrity.

### 6.3 Fixed Point Arithmetic

An important aspect to consider in the implementation of the firmware described is the arithmetic computations executed within the FPGA. A factor which needs to be considered are the accuracy with which arithmetic is executed to attain the desired resolution output. All arithmetic performed within the FPGA is fixed point implying that only integer valued arithmetic is done. To assure that the correct resolution is achieved the following provisions are made within the firmware:

- All filter coefficients and system matrix values are normalised and scaled sufficiently before stored within the ROM megafunctions.
- The MAC megafunctions used to perform arithmetic have sufficient bit widths ensuring that multiplication and addition results do not overflow or truncate.

Therefore all arithmetic computed within the FPGA has a resolution higher than 24-bits, but since the maximum desired PWM output resolution is 24-bits, the MAC output results are truncated when used by other processes. Making sure that all computational results and data have 24-bit resolution implies that minor alterations need to be made to the VHDL firmware when 24-bit audio quality can be achieved through the noise shaping process described previously.

## 6.4 Cyclone FPGA Resources

The last factor considered with the implementation of the VHDL firmware is to evaluate how much of the Cyclone FPGA resources are used. It is important to note that only one channel of the audio data has been implemented, therefore the resources used for one channel need to be doubled when both channels are implemented in a practical system. Appendix E gives a complete list of features of the Cyclone FPGA (EP1C12Q240C6) used to implement the firmware described within this chapter.

The Quartus II software report shown in Figure 6.18 gives the percentage of these resources used. The two resources which are of main concern are the total logic elements and total memory bits. When doubling the percentages of these resources it is concluded that not enough logic elements would be available for a second channel, since 80% of memory bits are used for the one channel implementation. It will therefore be necessary to use two Cyclone FPGA devices if a reduction in memory usage of the digital PWM modulator cannot be achieved.

Flow Status	Successful - Mon Nov 28 19:42:18 2005
Quartus II Version	4.1 Build 181 06/29/2004 SJ Web Edition
Revision Name	PWM_352
Top-level Entity Name	PWM_352
Family	Cyclone
Device	EP1C12Q240C6
Timing Models	Production
Total logic elements	5,785 / 12,060 ( 47 % )
Total pins	48 / 173 ( 27 % )
Total memory bits	192,957 / 239,616 ( 80 % )
Total PLLs	2 / 2 ( 100 % )

**Figure 6.18: Quartus II flow summary.**

## 6.5 Implementation Difficulties

The following difficulties were encountered with the implementation of the VHDL firmware described.

- Meeting the timing constraints of the Quartus II megafunctions
- Proper testing of the complete firmware implementation using the Quartus II waveform simulator.

These mentioned difficulties caused the implementation of the firmware to take an excessively long time. This is because a long simulation time window is needed to properly simulate timing characteristics of the megafunctions and the other developed firmware.

When a long time window is used (typically 2 ms) the simulator runs for hours before a result is obtained. If a small adjustment is made to the VHDL code, hours are then past until the next result is available. Table 6.2 gives the estimated time duration used to implement the various firmware blocks, excluding the time still needed to test all the firmware blocks together. Based on these tabulated projections, the complete implementation involving the different signal processing blocks could regrettably not be tested to its full extent through Quartus II simulation. This was only due to a limited time constraint.



Firmware	Duration in weeks
Configuration	5
Polyphase filtering	21
PNPWM	6
Noise Shaper	2
PWM Generator	2
Total	36 $\approx$ (9 Months)

**Table 6.2: Firmware development time.**

## 6.6 Summary

The chapter started off by describing the complete hardware system for a mono channel class-D audio amplifier. It then described the VHDL firmware blocks implemented within an ALTERA® CYCLONE FPGA. These blocks represent the signal processing necessary for a digital PWM converter. Although a 16-bit audio source was used to implement the converter all arithmetic processes used within the firmware assured a computational resolution of 24-bits or higher. The high resolution arithmetic processes simplify the alteration of the firmware when a 24-bit audio source is used.

The highest digital PWM output resolution attained was 8-bit since logic switching at 361 MHz could not be implemented within the FPGA. The highest audio resolution expected at the output would therefore be 16-bits according to simulation results in Chapter 5. Difficulties with the timing constraints of the Quartus II megafunctions and long simulation run time hindered the complete testing of the firmware developed. It was also concluded that two ALTERA® CYCLONE FPGAs would be necessary to implement two audio channels since 80% of the ICs memory is used to implement one channel using the current developed firmware.

The next chapter will proceed to give a practical measurement of the digital PWM modulator firmware explained and developed within this chapter.



# Chapter 7 - Measurements and Results

## 7.0 Introduction

Complete simulations have been done in Matlab® of the digital signal processing blocks given in chapters three to five which comprise the premodulation algorithms. Chapter 6 then implemented these algorithms within VHDL firmware for programming of an ALTERA® CYCLONE FPGA. Within this chapter the performance of this firmware implementation is determined through measurement. The measurement setup is firstly described and various measurements obtained are given. A discussion of these measurements will then follow ending off the chapter.

## 7.1 Measurement Setup

Figure 7.1 shows the measurement setup. The Audio precision test system or ATS1-system generates a 16-bit S/PDIF digital input signal connected via an optical fibre to the development board which implements the digital modulation and generates four PWM gating signals.

These gating signals are then fed to a full bridge inverter which passes through a passive low pass filter for audio amplification. The amplified left channel output is then filtered again by an external filter which assures that all carrier and sideband harmonics outside the audio band is sufficiently attenuated.

This output is then sent to the ATS1-system for measurement. The actual system and measurement setup is illustrated in Figure 7.2.

The hardware design of the digital modulation board and the full bridge inverter with passive low pass output filter was developed by Prof H dt Mouton at the faculty of Electric and Electronic Engineering, University of Stellenbosch.

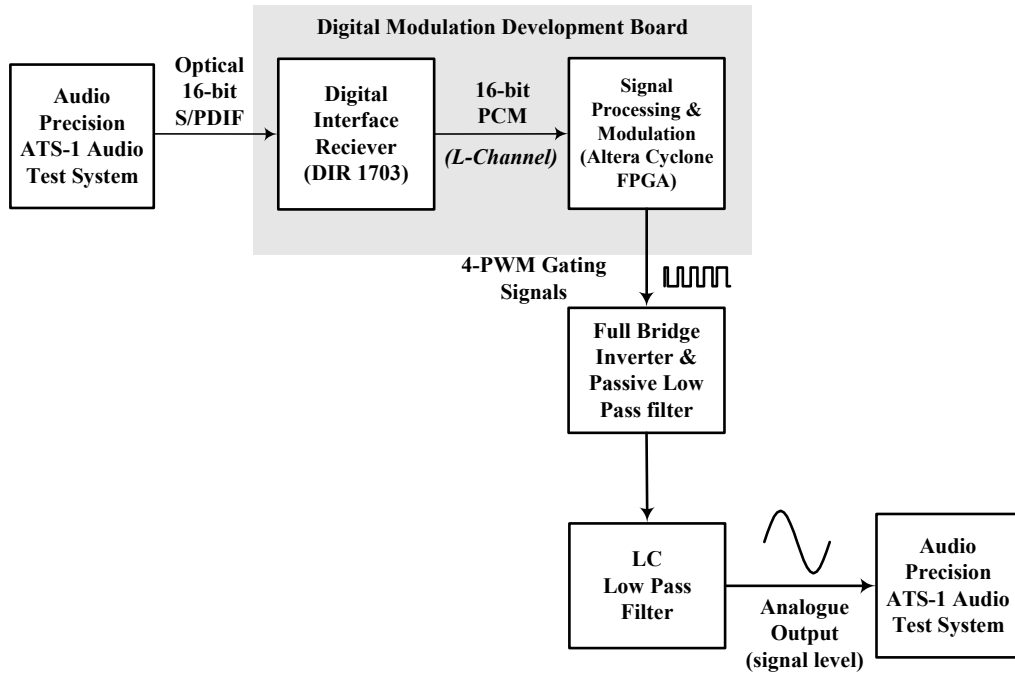


Figure 7.1: Digital modulation measurement setup.

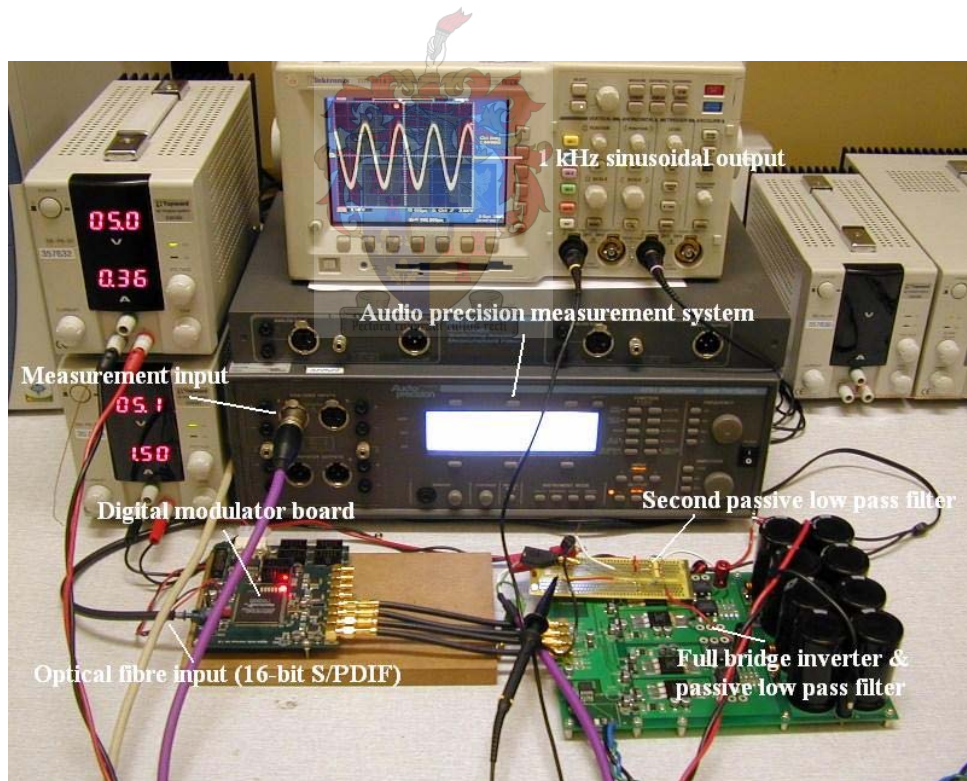


Figure 7.2: Complete measurement setup.

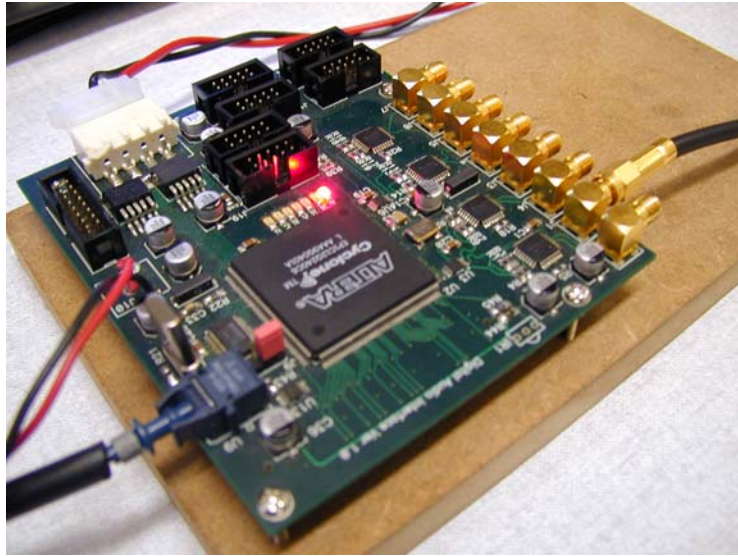


Figure 7.3: Zoomed view of digital modulation development board.

## 7.2 Measurements

A 1 kHz 16-bit sinusoidal tone exhibiting an amplitude of 0 dBFS was generated by the ATS-1 audio test system and input to the digital modulator as described above. All the simulations done previously within this thesis used a 1 kHz sinusoidal tone which is a common choice for measuring audio systems. The measurements which were done here not only used the 1 kHz tone but used various other frequencies as well to gain a good understanding of the amplifier performance. The measurements which were taken will now be described.

### 7.2.1 PWM gating signals

Figure 7.4 shows one of the FPGAs PWM output gating signals when a 16-bit, 1 kHz tone is applied by the ATS1-system. The different pulse durations are superimposed on each other in this measurement. From this figure it is seen that the duty cycle of the PWM signal doesn't vary more than 50%. Figures 7.5 and 7.6 show single cycle measurements of the PWM gating signal, from these figures it can be seen how the pulse width signal is varying. An overshoot of 38% is also observed on these PWM gating signals.

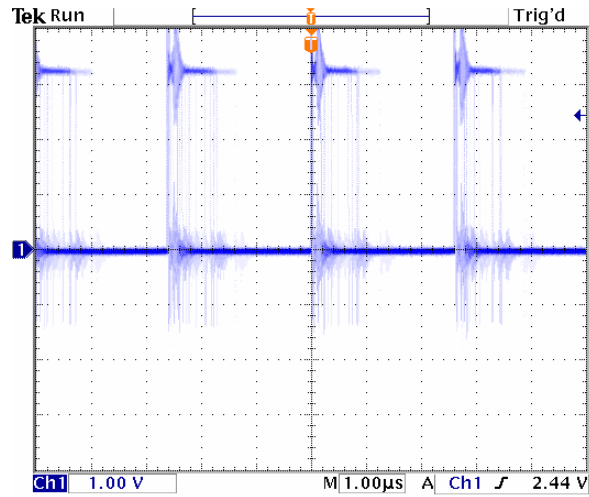


Figure 7.4: PWM gating output signal from FPGA.

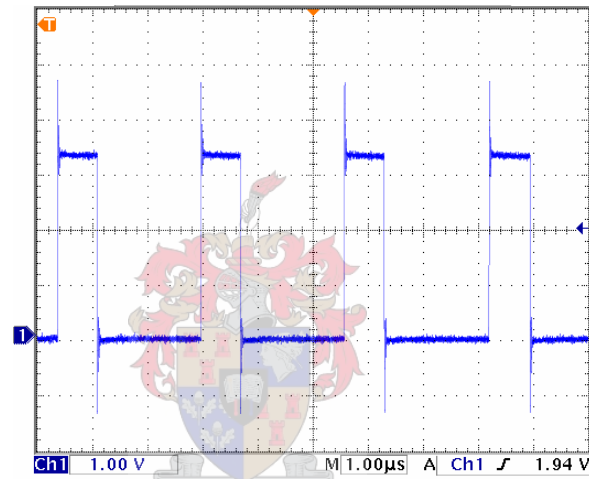


Figure 7.5: Single cycle of PWM gating signal.

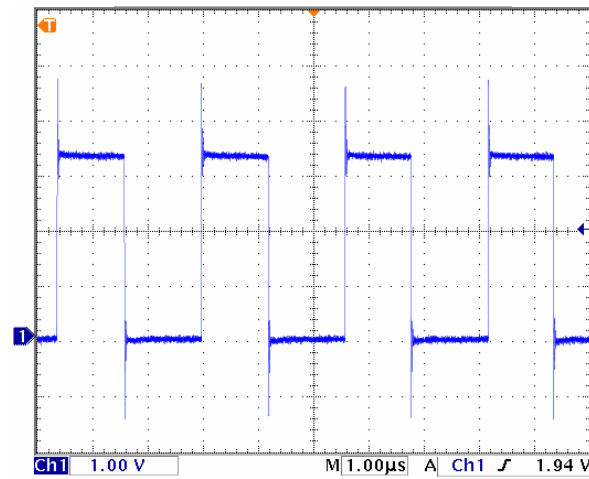


Figure 7.6: Single cycle of PWM gating signal.

## 7.2.2 Amplified output measurement

The amplified output of the filtered inverter was then measured using an oscilloscope at frequencies of 1, 10, 12 and 16 kHz respectively. These measurements are illustrated in Figure 7.7, 7.8, 7.9 and 7.10 respectively

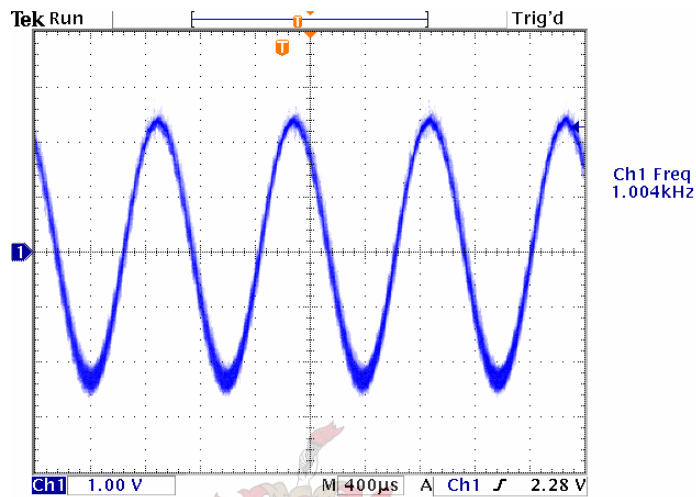


Figure 7.7: Amplified 1 kHz sinusoidal output.

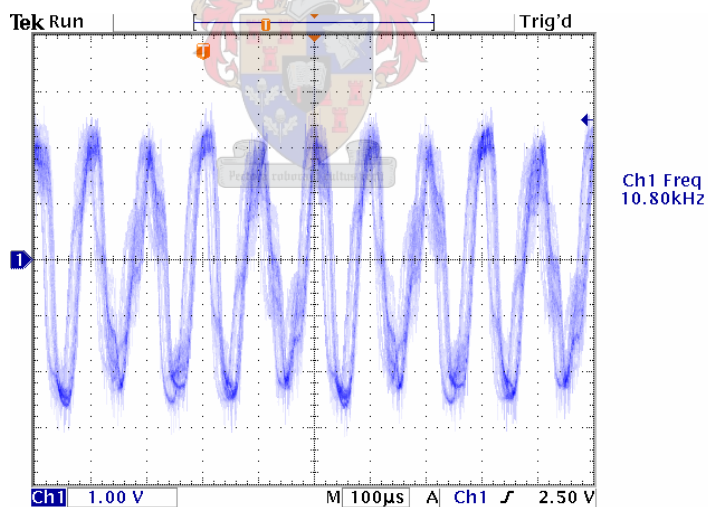
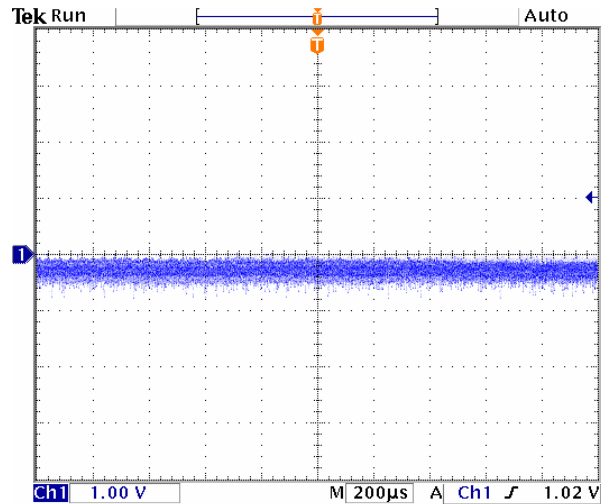
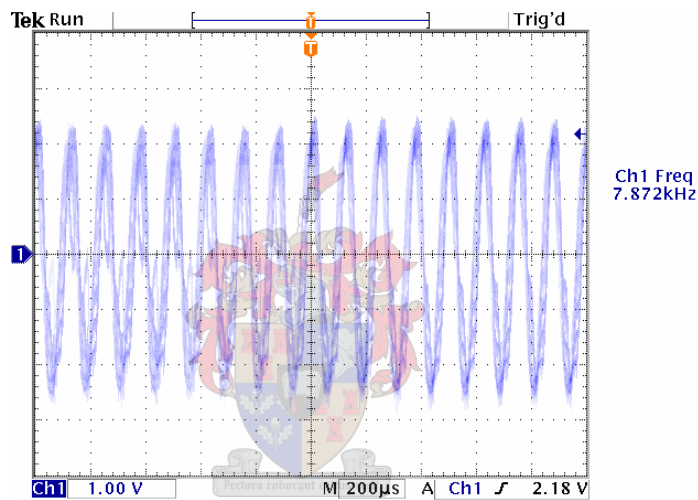


Figure 7.8: Amplified 10 kHz sinusoidal output.



**Figure 7.9: Amplified 12 kHz sinusoidal output**



**Figure 7.10: Amplified 16 kHz sinusoidal output**

From Figure 7.7 it is seen that the 1 kHz amplified output has a clear sinusoidal form at  $5 V_{pp}$  with some added noise component. The 10 kHz output does not have a clear sinusoidal form but is distorted by some non-linearity. The 12 kHz output shows a peculiar result of about -1 V with noise superimposed on it. An even more peculiar result is given in Figure 7.10 here a 16 kHz input was applied and a distorted 7.9 kHz output was attained.

### 7.2.3 Frequency response measurement

With the strange results obtained above the frequency response of the Class-D amplifier was measured in dBV using the ATSI-system and is given in Figure 7.11. In this figure a clear attenuation is observed in the region between 10 and 14 kHz.



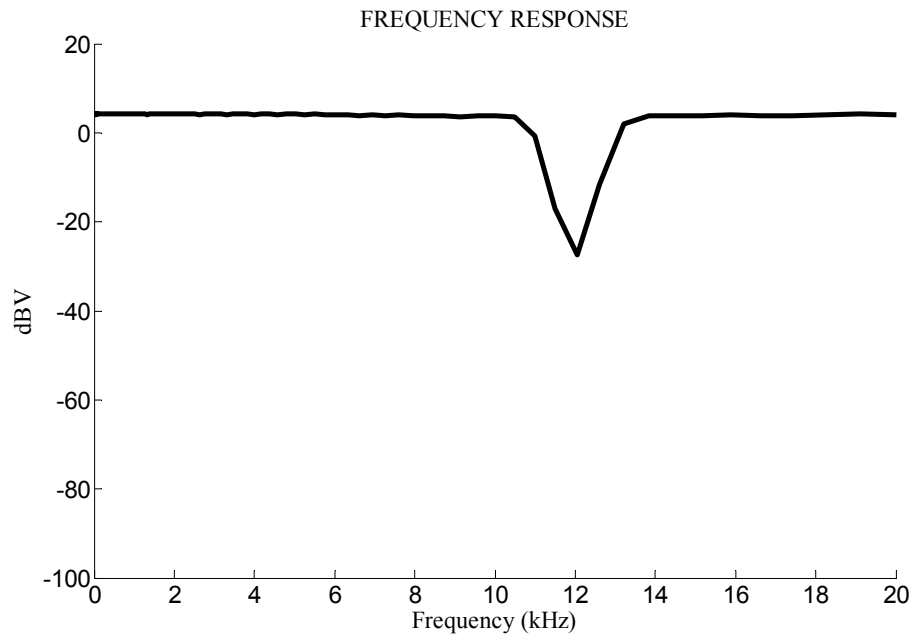


Figure 7.11: Frequency response of Class-D amplifier system.

#### 7.2.4 THD+N measurement

A Common measurement for a DAC system as is investigated here is the THD+N (Total Harmonic Distortion plus Noise) measurement [2]. The ATS-1-system was used to do this measurement and is shown in Figure 7.11.

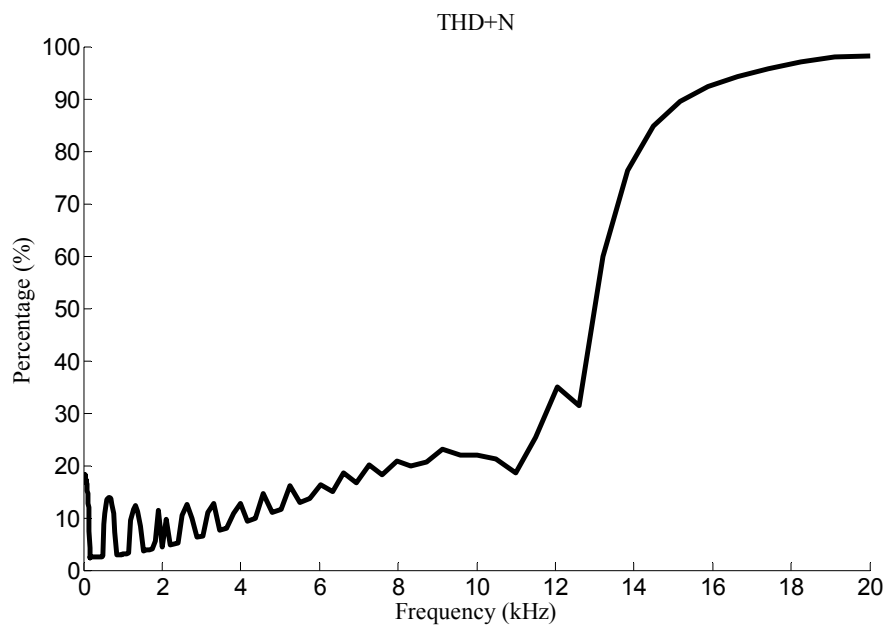


Figure 7.11: THD+N across the audio band.

High fidelity systems of 16-bit resolution should exhibit a THD+N of 0.02 within the entire audio band. When examining Figure 7.11 it is seen that the lowest THD+N is around 2% at an audible frequency of about 1 kHz. The THD+N level rises gradually until about 12 kHz where it then increases exponentially and then flattens out at 20 kHz with a THD+N of 98%. This measurement of THD+N obtained here is extraordinarily high.

### 7.3 Discussion of Measurement Results

It is clearly deduced from the above measurements that the digital modulator implemented using the developed firmware described in Chapter 6 doesn't work correctly. The first concern with these measurements starts with the PWM gating signal which has a maximum duty cycle of 50%, the duty cycle should be able to vary to 90%. The second concern is the attenuation present in the frequency response of the amplifier between 10 and 14 kHz. This attenuation was confirmed by both the oscilloscope amplitude measurement and the frequency response measurement of the ATS1-system. The third concern is the frequency division which occurs after 12 kHz frequency, this division phenomenon is confirmed by the THD+N measurements which increases drastically from this frequency value. Fourthly the overall THD+N measurement results are too high, and therefore this system cannot be considered as high fidelity.

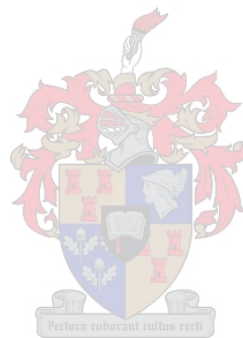
It is not totally clear what the source/s of these system inaccuracies are, what is clear is that the design methodology followed to implement the firmware was proven by theory and simulation in Matlab® within the previous chapters. The source of these inaccuracies therefore probably lies within the synthesis of the firmware implementation. More testing and simulation of the firmware is therefore necessary to identify and rectify the faults within the digital modulator.

### 7.4 Conclusions

Within this chapter a complete Class-D amplifier was setup for measurement. The goal of the setup was to test the firmware which was developed to implement a digital modulator. It was concluded through relevant measurements that the current developed firmware does not implement a high fidelity digital PWM modulator. It



was suggested that more simulation and testing of the firmware be done since the fault most probably lies within the synthesis of the firmware implementation.



# Chapter 8 - Conclusions

## 8.0 Overview

At the outset of this thesis the following objectives were given to address the difficulties associated with the digital PWM process:

- **Identifying** how these algorithms address these difficulties.
- **Dividing** these algorithms into appropriate blocks.
- Sufficiently **describing** each block in theory.
- Presenting a **design** solution for each block.
- **Simulating** these designs in Matlab®.
- **Developing** VHDL firmware of the simulated designs.
- **Attempting** to realize a practical Class-D amplifier using the developed firmware.

## 8.1 Fulfilment of Objectives

It was identified that linearization and clock rate reduction addresses the digital PWM difficulties of performance and practicality respectively. The pre-compensation, pre-modulation algorithms which implement linearization are divided into interpolation and PNPWM. A noise shaping coder was used to implement clock rate reduction of PWM output signals.

These defined signal processing blocks were each described separately within their own chapter. Design of these algorithm blocks were implemented successfully within Matlab ® attaining satisfactory results.

If 24-bit audio data is available, the digital modulator designed (comprising of the modular pre-compensation algorithms) and simulated in Matlab® is able to transparently process the data up to the noise shaping coder. The highest audio bit resolution attained at the output of the noise shaping coder was 20-bits. These results where confirmed through relevant graphs.

The firmware implementation of the digital PWM modulator exhibits 24-bit resolution within the arithmetic computation. Only an 8-bit noise shaping coder could

be implemented within the firmware resulting in a maximum audio resolution output of 16-bits. This is still regarded as high fidelity. Difficulties were encountered with the implementation of the firmware which implied more time for development than expected, therefore hindering the complete testing of the firmware in simulation.

Relevant measurements of the developed firmware used in a practical Class-D audio amplifier setup were then taken. The measurements concluded that the firmware developed does not exhibit a high fidelity digital PWM modulator. The fault(s) present within the PWM modulator is(are) most probably caused by VHDL synthesis problems.

Apart from the fault(s) which is(are) present within the VHDL firmware, all the objectives given have therefore been attained.

## 8.2 Recommendations and Future Research

It is firstly recommended that more time should be spent simulating and debugging the VHDL firmware. When the synthesis fault(s) has(have) been found, the current VHDL firmware developed should attain a 16-bit PWM audio resolution output.

A second recommendation is to implement a 10-bit noise shaping coder which has the potential to increase the audio resolution output to 20-bits.

Thirdly a reduction in VHDL memory resources could enable the processing of both audio channels on one Cyclone FPGA. This would cut development costs quite dramatically, since the Cyclone FPGA used in this thesis to implement one audio channel costs R405.

Further research needs to be done on the noise shaping coder since it is the only block within the digital PWM modulator that does not attain a 24-bit output. Noise shaping coders therefore need to be developed which are able to produce 24-bit audio resolution from a 10-bit output, or FPGA clock speeds of 723 MHz or faster are necessary to implement noise shapers of 11-bits or more. With current clock speed constraints the second option is not viable within an FPGA implementation.

# References and Bibliography

## Textbooks

- [1] Abdelwahab Kharab, Ronald B. Guenther, *An Introduction to Numerical Methods a Matlab Approach*, Chapman & Hall/CRC, 2002.
- [2] Bob Metzler, *Audio Measurement Handbook*, Audio Precision, Inc 1993
- [3] John G. Proakis, Dimitris G. Manolakis, *Digital Signal Processing*, Third Edition, Prentice Hall, Inc 1996, p269-273, p331-332, p619-p620, p637, p638.
- [4] Grahame D. Holmes, *Pulse Width Modulation Theory for Power Converters*, John Wiley, 2003.
- [5] Harold S. Black, *Modulation Theory*, D. Van Nostrand Company, Inc 1953.
- [6] Mark Zwolinski, *Digital System Design with VHDL*, Prentice Hall, 2000.
- [7] Peter J. Ashenden, *The Designer's Guide to VHDL*, Academic Press, 2002.
- [8] Peter Deufhard, Andreas Hohmann, *Numerical Analysis in Modern Scientific Computing*, Second Edition, Springer-Verlag New York, Inc 2003.
- [9] R. W. Hamming, *Numerical Methods for Scientists and Engineers*, McGraw-Hill Book Company, Inc 1962.
- [10] Simon Haykin, *An Introduction to Analogue and Digital Communications*, John Wiley & Sons, Inc 1989.
- [11] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flanery, *Numerical Recipes in C++, The Art of Scientific Computing*, Second Addition, Cambridge University Press, 2002.

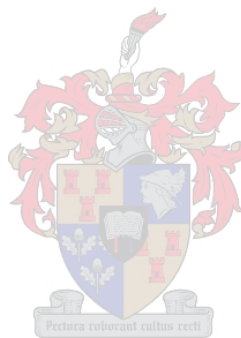
## Papers

- [12] C. E. Shannon, "A Mathematical Theory of Communication", Reprinted with corrections from The Bell System Technical Journal, Vol. 27, pp. 379-423, 623-656, July, October, 1948.
- [13] C. Pascual, Z. Song, P.T. Krein, D.V. Sarwate, P. Midya, W.J. Roeckner, "High Fidelity PWM Inverter for Digital Audio Amplification: Spectral Analysis, Real-Time DSP Implementation, and Results" IEEE Transactions on Power Electronics, Vol. 18, No.1, January 2003.
- [14] D. De Koning, W. Verhelst, "On Psychoacoustic Noise Shaping for Audio Requantization", Vrije Universiteit Brussels, dept. ETRO-DSSP, Pleinlaan 2, B-1050 Brussels, Belgium.

- [15] E. Bresch, W. T. Padgett, "TMS320C67-Based Design of a Digital Audio Power Amplifier Introducing Novel Feedback Strategy", Rose-Hulman Institute of Technology, Electrical and Computer Engineering Department, 5500 Wabash Ave., Terre Haute, IN 47803.
- [16] J. Stephenson, "Design Guidelines for Optimal results in FPGAs", Altera Corporation, 2005.
- [17] J. M. Goldberg, M. B. Sandler, "Digital-to-Analogue convertor/Power Amplifier", IEE Proc.-Circuits Devices Syst., Vol. 141, No. 4, August 1994
- [18] J.M. Goldberg, M. B. Sandler, "Noise Shaping and Pulse-Width Modulation for an All-Digital Audio Power Amplifier", Department of Electronic and Electrical Engineering, King's College, University of London, London WC2R 2LS, UK.
- [19] J. M. Goldberg, M. B. Sandler, "Pseudo-Natural Pulse Width Modulation for High Accuracy Digital-to-Analogue Conversion", Electronic Letters, Vol. 27, No. 16, 1 August 1991.
- [20] M. Streitenberger, H. Bresch, W. Mathis, "Theory and Implementation of a New Type of Digital Power Amplifiers for Audio Applications", IEEE International Symposium on Circuits and Systems, May 2000.
- [21] M. O. J. Hawksford, Chaos, Oversampling, and Noise Shaping in Digital-to-Analog Conversion, Department of Electronic Systems Engineering, University of Essex, Colchester, Essex, C04 3SQ, UK.
- [22] R. E. Crochiere, L. R. Rabiner, "Interpolation and Decimation of Digital Signals- A Tutorial Review", Proceedings of the IEEE, Vol. 69, No. 3, March 1981.
- [23] R. A. Losada, "Practical FIR Filter Design in Matlab®", Revision 1.0, The Math Works, Inc., 3 Apple Hill, Dr Natick, MA 01760, USA, March 31, 2003
- [24] R. W. Schafer, L. R. Rabiner, "A Digital Signal Processing Approach to Interpolation", Proceedings of the IEEE, Vol 61, No. 6, June 1973.
- [25] S. K. Tewksbury, R. W. Hallock, "Oversampled Linear Predictive and Noise-Shaping Coders of Order  $N > 1$ ", IEEE Trans. Circuits Sys., Vol. CAS-25, pp 437-447, July 1978.

**Websites**

- [26] ALTERA.COM, “Altera Megafunctions”,  
<http://www.altera.com/products/ip/altera/mega.html>, 2005.
- [27] ALTERA.COM, “Hardware Design Considerations”,  
[http://www.altera.com/literature/hb/hrd/hc\\_h5v1\\_03.pdf](http://www.altera.com/literature/hb/hrd/hc_h5v1_03.pdf), 2005
- [28] ALTERA.COM, “Quartus II Timing Analysis”,  
[http://www.altera.com/literature/hb/qts/qts\\_qii53004.pdf](http://www.altera.com/literature/hb/qts/qts_qii53004.pdf), 2005



# Appendix A

## A1. Interpolation Matlab Code

```

%Programmer   : Deon Jacobs
%Date        : 15 July 2005
%Goal        : (a) Find PSD of a signal sampled at 44.1 KHz
%            : (b) Using mfilter construct a polyphase filter structure
%            :         from the equiripple FIR filter (ER_632.fda) designed
%            :         with the fdatool. The goal of the structure is to interpolate
%            : (c) Filter the input signal with the polyphase structure
% Filename    : signal_psd.m
% Revision    : 1.1

clc
data_length = 2^16;
n           = [0:1:data_length];           %signal data length vector
n_up       = [0:1:data_length*8];         %Upsampled filtered output data length vector
F1         = 1000;                         %sinusoidal frequency
Fs         = 44100;                        %sampling frequency
F_up       = 352.8e3;                     %upsampling frequency
Na         = 1/(2^24);                    %noise component amplitude
A          = 1;                           %sinusoidal component amplitude
L1         = 1;                           %amounts of experiments of signal
L          = 8;                           %upsampling ratio
nfft       = 2^16;                        %length of the PSD estimate (nfft-data_length=zero_padded)
Z          = 1024;                        %Amount of zeros to be padded after input signal
fs         = 18;                          %Font size of graph labels
lw         = 2;

%-----
% Input signal array
%-----

[xn_arr] = data_array(n,F1,Fs,Na,L1,A,Z);   % Sinusoidal input signal

%-----
% Setup Polyphase structure from the filter coefficients contained in Num,
% for interpolation. Using Matlab function mfilter
%-----

Pn = mfilter.firinterp(L,Num);

%-----
%Use own polyphase structure code
%-----

[yn_own_arr,Poly_matrix] = polyphase_filter_arr(xn_arr,L1,Num,L);

%-----
% Zero pad between consecutive samples (sample rate expander)
%-----
% Oversample by 8 through pending zeros between consecutive samples if L1=1
if (L1 == 1),
    z = zeros(1,7); % Zero vector for oversampling
    for i=[1:1:data_length],
        if i==1
            xn_zero = [xn_arr(i) z];
        else
            xn_zero = [xn_zero xn_arr(i) z];
        end;
    end;
end;

```

```

    end;
end;
%-----
% PSD estimates of the input signal
%-----

for i=[1:1:L1], % Make L1 realizations of each estimate

    [MTM_xx,fx_MTM] = pmtm(xn_arr,[],length(xn_arr),'onesided'); % MTM estimate of input
    [MTM_ww,fw_MTM] = pmtm(xn_zero,[],length(xn_zero),'twosided'); % MTM estimate of sample rate
                                expanded signal

end;

%-----
%PSD estimates of the output signal
%-----

for i=[1:1:L1], % Make L1 realizations of each estimate

    [MTM_yy,fy_MTM] = pmtm(yn_own_arr,[],length(yn_own_arr),'twosided');
end;

%-----
%Averages of PSD estimates
%-----
    MTM_xx_avg = MTM_xx;
    MTM_ww_avg = MTM_ww;
    MTM_yy_avg = MTM_yy;

%-----
% Logarithms of PSD estimates
%-----

    MTM_xx_dB = 10*log10(MTM_xx_avg);
    MTM_ww_dB = 10*log10(MTM_ww_avg);
    MTM_yy_dB = 10*log10(MTM_yy_avg);

%-----
%%-----
%% Graph code -
%%-----
set(0,'DefaultAxesColorOrder',[0 0 0],'DefaultLineLineWidth',lw);

%-----
%Plots of PSD estimates
%-----

%Plot of the input signal PSD
% Figure(1)
% plot(fx_MTM,MTM_xx_dB);
% set(gca,'FontSize',fs);
% set(get(gca,'XLabel'),'FontSize',fs);
% set(get(gca,'YLabel'),'FontSize',fs);
% set(get(gca,'Title'),'FontSize',fs);
% xlabel('Normalized Frequency (x\pi rad/sample)');
% ylabel('dB');
% axis([0 1 -200 0]);

% Plot of the sample rate expanded PSD
% Figure(2)
% plot(fw_MTM,MTM_ww_dB);
% set(gca,'FontSize',fs);
% set(get(gca,'XLabel'),'FontSize',fs);
% set(get(gca,'YLabel'),'FontSize',fs);
% set(get(gca,'Title'),'FontSize',fs);
% xlabel('Normalized Frequency (x\pi rad/sample)');

```



```

% ylabel('dB');
% axis([0 1 -200 0]);

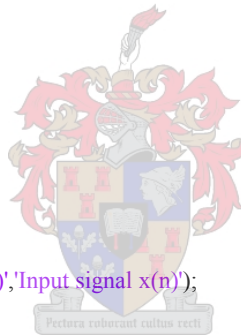
% Plot of upsampled and original signal
Figure(3)
subplot(3,1,1)
begin1 = 312;
index = 800;
extra = 5;
end1 = begin1+index;
end2 = index/L+1;
%Upsampled signal
if (L1 ~= 1),
    stem((n_up(begin1:end1)/F_up)*1e3,yn_own_arr(1,begin1+extra:end1+extra),'b','filled');
else
    stem((n_up(begin1:end1)/F_up)*1e3,yn_own_arr(begin1+extra:end1+extra),'filled','color',[0.569 0.569 0.569]);
end;
hold on
% stem(n_up(1115:1483)/F_up,yn_matlab_arr(1,1117:1485),'b');

%Original signal
if (L1 ~= 1),
    stem((n_up(begin1:L:end1)/F_up)*1e3,xn_arr(1,1:end2),'k','filled');
else
    stem((n_up(begin1:L:end1)/F_up)*1e3,xn_arr(1:end2),'k','filled');
end;
hold off
xlabel('(a)');
ylabel('Amplitude');
%title('y(n)');
set(gca,'FontSize',fs);
set(get(gca,'XLabel'),'FontSize',fs);
set(get(gca,'YLabel'),'FontSize',fs);
set(get(gca,'Title'),'FontSize',fs);
set(gca,'YTick',[-1 0 1]);
axis([0.9 2.6 -1.1 1.1]);
legend('Interpolated output signal y(n)','Input signal x(n)');
legend('boxoff');
box off
subplot(3,1,2)

%Upsampled signal
if (L1 ~= 1),
    stem((n_up(begin1:end1)/F_up)*1e3,yn_own_arr(1,begin1+extra:end1+extra),'b','filled');
else
    stem((n_up(begin1:end1)/F_up)*1e3,yn_own_arr(begin1+extra:end1+extra),'filled','color',[0.569 0.569 0.569]);
end;
hold on
% stem(n_up(1115:1483)/F_up,yn_matlab_arr(1,1117:1485),'b');

%Original signal
if (L1 ~= 1),
    stem((n_up(begin1:L:end1)/F_up)*1e3,xn_arr(1,1:end2),'k','filled');
else
    stem((n_up(begin1:L:end1)/F_up)*1e3,xn_arr(1:end2),'k','filled');
end;
hold off
xlabel('(b)');
ylabel('Amplitude');
%title('y(n)');
set(gca,'FontSize',fs);
set(get(gca,'XLabel'),'FontSize',fs);
set(get(gca,'YLabel'),'FontSize',fs);
set(get(gca,'Title'),'FontSize',fs);
set(gca,'YTick',[-1 0 1]);
axis([1.8 2.1 0 1.1]);

```



```

box off
subplot(3,1,3)

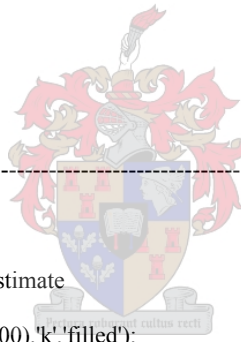
%Upsampled signal
if (L1 ~= 1),
    stem((n_up(begin1:end1)/F_up)*1e3,yn_own_arr(1,begin1+extra:end1+extra),'b','filled');
else
    stem((n_up(begin1:end1)/F_up)*1e3,yn_own_arr(begin1+extra:end1+extra),'!',color',[0.569 0.569
0.569],LineStyle,'none');
end;
hold on
% stem(n_up(1115:1483)/F_up,yn_matlab_arr(1,1117:1485),'b');

%Original signal
if (L1 ~= 1),
    stem((n_up(begin1:L:end1)/F_up)*1e3,xn_arr(1,1:end2),'k','filled');
else
    stem((n_up(begin1:L:end1)/F_up)*1e3,xn_arr(1:end2),'k','filled');
end;
hold off
xlabel('Time (itms)');
xlabel('(c)');
ylabel('Amplitude');
text(0.5,0.5,('\itms'));
%title('y(n)');
set(gca,'FontSize',fs);
set(get(gca,'XLabel'),'FontSize',fs);
set(get(gca,'YLabel'),'FontSize',fs);
set(get(gca,'Title'),'FontSize',fs);
set(gca,'YTick',[-1 0 1]);
axis([1.9 2.05 0 1.1]);
box off
%-----
%
% set(0,'DefaultAxesFontSize',fs);

%Sinusoidal input signal & Spectral estimate
% Figure(5)
% stem((n(1:100)/Fs)*1e3,xn_avg(1:100),'k','filled');
% xlabel('Time (itms)');
% ylabel('Amplitude');
% title('x(n)');
% set(gca,'FontSize',fs);
% set(get(gca,'XLabel'),'FontSize',fs);
% set(get(gca,'YLabel'),'FontSize',fs);
% set(get(gca,'Title'),'FontSize',fs);
% set(gca,'YTick',[-1 0 1]);
% axis([0 2 -1 1]);
% grid off

%Sample rate expanded signal
% Figure(6)
% stem((n(1:350)/F_up)*1e3,xn_zero(1:350),'k','filled');
% xlabel('Time (itms)');
% ylabel('Amplitude');
% title('w(n)');
% set(gca,'FontSize',fs);
% set(get(gca,'XLabel'),'FontSize',fs);
% set(get(gca,'YLabel'),'FontSize',fs);
% set(get(gca,'Title'),'FontSize',fs);
% set(gca,'YTick',[-1 0 1]);
% grid off

```

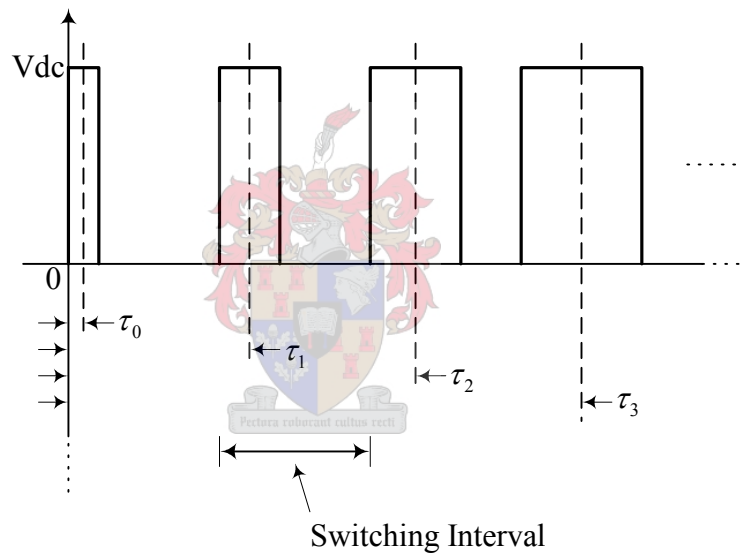


# Appendix B

## PWM Spectral Calculation

### B1. Estimation method

All spectrum plots of PWM signals are generated using the following spectral estimate method (assuming a periodical modulating waveform): A typical PWM signal has a pulse varying characteristic shown in Figure B1.



**Figure B1: PWM signal.**

The spectrum is estimated by storing a number of pulses of the PWM output signal which represents one period of the modulating waveform. Each one of these pulses within the period represents a rectangular function in the time domain which is offset by a certain time value from the zero time coordinate. These time intervals are illustrated in Figure B1. Each of these rectangular functions has a frequency characteristic related to it and given by the following mathematical expression

$$A\text{Rect}\left(\frac{t-\tau_n}{T_n}\right) \Leftrightarrow AT\text{Sinc}(T_n f) \left[ e^{-j2\pi f \tau_n} \right]. \quad (\text{C.1})$$

Where  $\tau_n$  represents the time position of a particular pulse from the time origin, and  $T_n$  represents the particular pulse width. The exponential term is present because of the time displacement each pulse width exhibits from the origin.

Figure B2 shows how each rectangle is converted to its frequency domain counterpart using (B.1). A frequency index is generated to include the relevant frequency values according to the signal characteristics.

For example if it is known that the switching frequency component is 352.8 kHz, this frequency number needs to be included in the frequency index otherwise it will not be represented within the estimated spectrum. The frequency index therefore needs to be chosen as fine as possible to insure that the estimate is a true representation of the PWM signal spectrum.

After the relevant frequency sinc values have been calculated from the corresponding pulse width, these sequences are then multiplied with a Hanning window function to minimize spectral leakage occurring (Figure B.2). The windowed sinc sequences resulting from each pulse width forms a matrix which is then added column wise and then averaged according to the number of pulse widths within used for the spectral estimate. This averaged sequence represents a statistical average of the PWM spectra. The more modulating periods used to estimate the spectra the more accurate the result becomes.

The next section in this Appendix gives the Matlab ® code which is used to estimate the PWM spectra within this thesis.

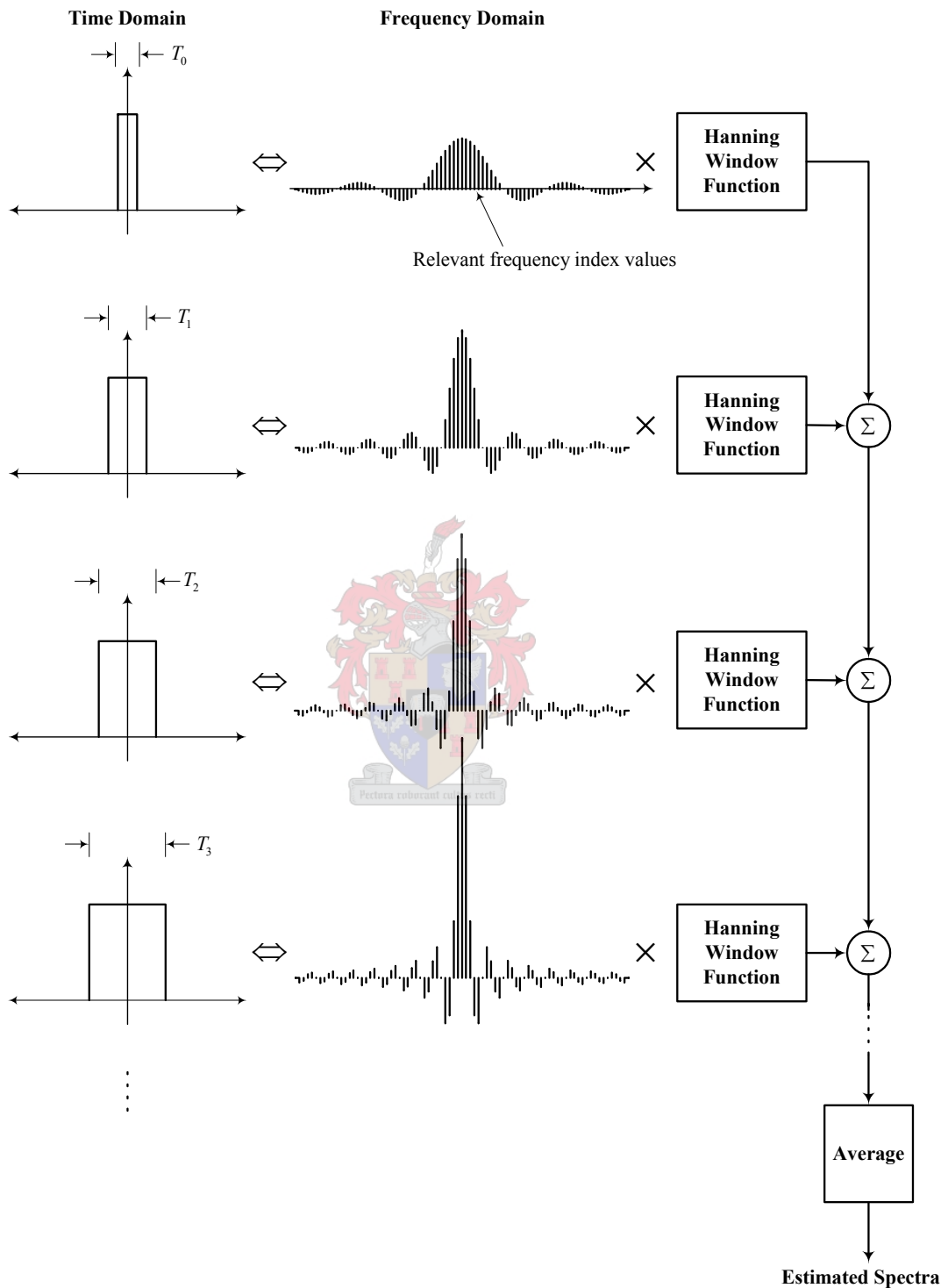


Figure B.2: Spectrum estimate calculation.

## B2. Matlab Code for PWM Spectral Estimate

```

%Programmer   : Deon Jacobs
%Date        : 4 August 2005
%Goal        : Determine the spectral content of a PWM signal
%            : Instead of using spectral estimate methods, the
%Filename     : PWM_spectra.m
%Revision    : 1.0

function [pwm_spec,f] = PWM_spectra(pwm_widths,Fs,Fstep,F_end)

% pwm_widths : vector containing calculated pwm_widths
% Fs         : sampling frequency
% F1         : resolution of the frequency axis

Ts = 1/Fs; % Sampling frequency
f = [0:Fstep:F_end]; % Frequency index
Window = hanning(length(pwm_widths)); % Window function

% Make sure window has unity variance

window_square_sum = sum(Window.^2);
ratio = sqrt(window_square_sum);
Hann_window = Window/ratio*sqrt(length(Window));

% Compute different sinc contributions at specified frequency values

for i = [1:length(pwm_widths)],
    %half the length of the pwm_width
    half_width = pwm_widths(i)/2;

    %Middle point time of square window with respect to the origin
    middle_point(i) = Ts*(i-1)+half_width;

    %Sinc calculation in the frequency domain
    f_sinc(i,:) = ((pwm_widths(i)*sinc(f*pwm_widths(i))).*exp(-j*2*pi*f*middle_point(i)))*Hann_window(i);
end;
pwm_spec = sum(f_sinc)/length(pwm_widths);

```

# Appendix C

## Spectral Estimate Matlab Code for PWM Schemes

### C1. Trailing edge NPWM

```

% Programmer   : Deon Jacobs
% Date        : 5 Augustus 2005
% Goal        : NPWM spectral estimation using Newton Raphson
%             : iteration
% Filename     : Newton_Raphson.m
% Revision    : 1.0
clear all
close all
clc

% Graph properties
%-----
fs = 18;
lw = 2;
set(0,'DefaultAxesColorOrder',[0 0 0],'DefaultLineLineWidth',lw);

Fs          = 352.8e3; %Saw-tooth frequency
ws          = 2*pi*Fs; %Corner frequency of sampling frequency
Ts          = 1/Fs;    %Period of the saw-tooth wave
f_divider   = 353;    %Divider to get the tonal frequency
F_end       = Fs;     %Maximum spectral frequency value
k           = [0:1:f_divider]; %Cross point index
F1          = Fs/(f_divider); %Signal frequency content
w1          = 2*pi*F1; %Corner frequency of input signal
T1          = 1/F1;   %Signal period
t_cross_pnt = k*Ts;  %Crosspoint interval boundaries
t_app_zero  = (k+0.5)*Ts; %Guesses to crosspoints within PWM intervals
A           = 0.95;  %Sinusoidal amplitude
omega       = 2*pi*F1; %Corner frequency of sinusoidal signal

for i = [1:1:f_divider],

    for iter = [1:1:5],
        % Calculate the sawtooth waveform for the present interval
        st(i) = (2/Ts)*t_app_zero(i) + (1-2*i);

        % Input signal
        xt(i) = A*sin(omega*t_app_zero(i));

        % Subtract above two equations to gain the new function which roots
        % need to be determined
        ft(i) = st(i) - xt(i);

        % Calculate the denominator derivative
        ft_der(i) = (2/Ts) - A*omega*cos(omega*t_app_zero(i));

        % Newton Raphson iteration
        t_app_zero(i) = t_app_zero(i) - ft(i)/ft_der(i);
    end;
    t_pwm_width(i) = t_app_zero(i) - t_cross_pnt(i);
end;

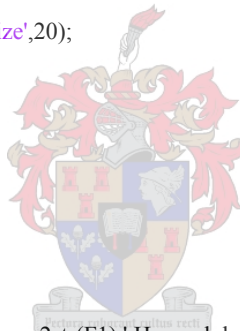
```

```
[pwm_spec,f] = PWM_spectra(t_pwm_width,Fs,F1,F_end);    % Compute PWM spectra
pwm_spec = pwm_spec/max(pwm_spec);
pwm_spec_dB = 10*log10((abs(pwm_spec)).^2);           % Decibel value of spectra
```

```
Figure(1)
stem(t_pwm_width);
```

```
Figure(2)
subplot(1,2,1)
plot((f/1e3),pwm_spec_dB,'.', 'markersize',20);
hold on
plot((f/1e3),pwm_spec_dB);
hold off
%hold on
%plot(f,pwm_spec_wind_dB,'.r');
%hold off
set(gca,'FontSize',fs);
set(get(gca,'XLabel'),'FontSize',fs);
set(get(gca,'YLabel'),'FontSize',fs);
set(get(gca,'Title'),'FontSize',fs);
%title(['NPWM Spectral Estimate of ',num2str(F1),' Hz modulation tone']);
title('Frequency Content Showing Carrier Harmonics','FontSize',14);
xlabel('(a) kHz');
ylabel('dB');
box off
```

```
subplot(1,2,2)
plot((f/1e3),pwm_spec_dB,'.', 'markersize',20);
hold on
plot((f/1e3),pwm_spec_dB);
hold off
%hold on
%plot(f,pwm_spec_wind_dB,'.r');
%hold off
set(gca,'FontSize',fs);
set(get(gca,'XLabel'),'FontSize',fs);
set(get(gca,'YLabel'),'FontSize',fs);
set(get(gca,'Title'),'FontSize',fs);
%title(['NPWM Spectral Estimate of ',num2str(F1),' Hz modulation tone']);
title('Zoomed View of Baseband Frequency Content','FontSize',14);
xlabel('(b) kHz');
%ylabel('dB');
axis([0 20 -350 0]);
box off
```





## C2. Trailing edge UPWM, and PNPWM using Newton's method

```

% Programmer   : Deon Jacobs
% Date        : 26 October 2005
% Goal        : Pseudo pulse width modulation simulation, using an 8th order
%             : polynomial p(t) fitted through PCM samples x(n) and trailing edge
%             : modulation s(t).
% Filename    : PNPWM.m
% Revision    : 3.1

clc
clear
fsize = 18;
lw = 2;
set(0,'DefaultAxesColorOrder',[0 0 0],'DefaultLineLineWidth',lw);
set(0,'DefaultFontSize',18,'DefaultAxesFontSize',18);

A      = 0.95;           % Amplitude
bit    = 24;            % Bit resolution
fs     = 352.8e3;       % Switching frequency
Ts     = 1/fs;         % Switching period
fl     = fs/352;       % Tonal frequency
f_end  = fs;           % Last frequency value represented in spectra
cycle_num = 10;       % Number of sinusoidal cycles, must be larger than 3;
ratio  = fs/fl;
n      = [-3:1:cycle_num*ratio+4]; % Index
t      = n/fs;         % Time index representation
O      = 8;            % Order of approximated polynomial
Qbit   = 8;            % Bit resolution after quantization
bit_res = 24;         % Resolution of PWM input
ms     = 2;           % Sawtooth gradient
c      = 9;           % y-axis crosspoint
I      = 1;           % Number of experimnets for statistical average

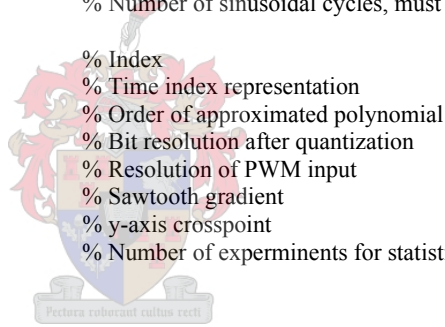
%-----
% Sinusoidal modulation wave
%-----
xn = A*cos(2*pi*(fl/fs)*n); %+(1/2^bit)*randn(1,length(n));

% Determine polynomial 8th order coefficients
%-----
% Xa=b
% a=polynomial coefficients
% X=system matrix
% b=input samples
x = [0:1:O]; %O sample index vector
for i=[1:1:O+1], %Row array loop
    matrix_temp = 1;
    for j=[2:1:O+1]; %Column fill in
        matrix_temp = [matrix_temp x(i)^(j-1)];
    end;
    X(i,:) = matrix_temp;
end;
% Inversion of the square system matrix
X_inv = inv(X);

% Step through input modulation wave:
%-----

%ns_filter_num = [5 -10 10 -5 1]; % Fifth order feedback filter
filt_order = length(ns_filter_num); % Order of noise shaper feedback filter

```



```

int_cnter = 1; % number of switching intervals
x_iter = 4.5; %Crosspoint guess within switching interval
ns_output = [];
ns_output(1)= 0; %first output of feedback filter, delay of one sample
erq_vec = zeros(1,fil_order); %feedback filtering vector

for i=[1:1:(length(xn)-O)],

    xn_capt(int_cnter,:) = xn(i:i+O); % Captured 9 samples

    % Calculate polynomial coefficients
    a(int_cnter,:) = X_inv*xn_capt(int_cnter,:); % a = X_inv*y

    %(1) UPWM modulation
    UPWM(int_cnter) = (((xn_capt(int_cnter,4)+c)/2)-4)/fs;

    %(2) Newton's Method for PNPWM
    [Newton_width(int_cnter)] = Newton_PNPWM(ms,c,a,int_cnter,x_iter,fs);

    %(3) Noise shaper
    %Generate 24-bit pwm
    Newton_width_24bit = round(Newton_width(int_cnter)*2^24);

    %1st adder
    dn = Newton_width_24bit + ns_output;

    %Quantizer
    pwm_8bit = bitshift(dn,-16);

    %Scale 8-bit resolution to 24-bit
    pwm_8bit_scale = bitshift(pwm_8bit,16);

    %Noise from quantization process (2nd adder/subtractor)
    e_rq = (dn-pwm_8bit_scale);

    %Update filter vector
    erq_vec = [e_rq erq_vec(1:end-1)];

    %Filter error e_rq
    filt_out = erq_vec(1:end)*(bitshift(ns_filter_num,32)); %round(ns_filter_num*2^32);

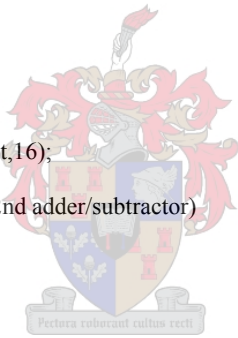
    %New feedback filter H(z) output
    ns_output = bitshift(filt_out,-32);

    %8-bit PWM vector
    pwm_ns_t(int_cnter) = pwm_8bit/((2^Qbit)*fs);

    %Increment counter
    int_cnter = int_cnter+1;
end;

[pwm_spec_ns,f_ns] = PWM_spectra(pwm_ns_t(356:end),fs,f1,f_end);
pwm_spec_ns_avg = pwm_spec_ns;
pwm_spec_norm = pwm_spec_ns_avg/max(pwm_spec_ns_avg);
pwm_spec_ns_dB = 10*log10(abs(pwm_spec_norm).^2);

```



```

%% => UPWM
%% -----
%% Spectral calculation
% [UPWM_spec,f] = PWM_spectra(UPWM,fs,f1,f_end);
% pwm_spec_norm = UPWM_spec/max(UPWM_spec);
% UPWM_spec_dB = 10*log10(abs(pwm_spec_norm).^2);
%% Spectral plot
%% Figure(1)
%% plot(f/1e3,UPWM_spec_dB,');
%% set(gca,'FontSize',fs);
%% set(get(gca,'XLabel'),'FontSize',fs);
%% set(get(gca,'YLabel'),'FontSize',fs);
%% set(get(gca,'Title'),'FontSize',fs);
%% title(['Uniform Pulse Width Modulated Spectral Estimate: ',num2str(f1),' Hz']);
%% xlabel('kHz');
%% ylabel('dB');
%% axis([0 400 -400 0]);
% Figure(1)
% subplot(1,2,1)
% plot(f/1e3,UPWM_spec_dB,',' markersize',20);
% hold on
% plot(f/1e3,UPWM_spec_dB);
% hold off
% box off
% title('Frequency Content Showing Carrier Harmonics','FontSize',14);
% xlabel('(a) kHz');
% ylabel('dB');
% axis([0 400 -350 0]);
% subplot(1,2,2)
% plot(f/1e3,UPWM_spec_dB,',' markersize',20);
% hold on
% plot(f/1e3,UPWM_spec_dB);
% hold off
% box off
% title('Zoomed View of Baseband Frequency Content','FontSize',14);
% xlabel('(b) kHz');
% ylabel('dB');
% axis([0 20 -350 0]);

% => Newton's Method for PNPWN
% -----
% Spectral calculation
[pwm_spec_N,f] = PWM_spectra(Newton_width(355:end),fs,f1,f_end);
pwm_spec_norm = pwm_spec_N/max(pwm_spec_N);
pwm_spec_N_dB = 10*log10(abs(pwm_spec_norm).^2);

%% PWM_output generate
%% [pwm_N_output] = plot_pwm(Newton_width,fs);
%%
%% PWM output plot
%% Figure(1)
%% x-axis scaling
%% m = [0:length(pwm_N_output)/length(xn):length(pwm_N_output)-1]*2.8987e-2;
%% m_pwm = [0:2.898704e-2:1.023242e3];
%% plot details
%% subplot(2,1,1)
%% Modulating input
%% plot(m,xn);
%% hold on
%% stem(m,xn,',' markersize',20);
%% hold off
%% axis([1.00e2 2.00e2 -0.2 max(xn)+0.2]);
%% set(gca,'YTick',[0 1]);

```



```

%% xlabel('Time({\mu}s');
%% ylabel('Amplitude');
%% title('ivity(m)');
%% box off
%% % PWM output
%% subplot(2,1,2)
%% plot(m_pwm,pwm_N_output);
%% axis([1.00e2 2.00e2 min(pwm_N_output)-0.2 max(pwm_N_output)+0.2]);
%% set(gca,'YTick',[0 1]);
%% set(gca,'YTickLabel',{'0','Vdc'})
%% xlabel('Time({\mu}s');
%% ylabel('Amplitude');
%% title('PNPWM output');
%% box off
%%
%% PNPWM output spectra
Figure(2)
subplot(1,2,1)
plot(f/1e3,pwm_spec_N_dB,'!','markersize',20);
% hold on
% plot(f/1e3,pwm_spec_N_dB);
% hold off
box off
title('Frequency Content Showing Carrier Harmonics','FontSize',14);
xlabel('(a) kHz');
ylabel('dB');
axis([0 400 -350 0]);
subplot(1,2,2)
plot(f/1e3,pwm_spec_N_dB,'!','markersize',20);
% hold on
% plot(f/1e3,pwm_spec_N_dB);
% hold off
box off
title('Zoomed View of Baseband Frequency Content','FontSize',14);
xlabel('(b) kHz');
%ylabel('dB');
axis([0 20 -350 0]);

% Noise shaped PWM spectra
Figure(3)
plot(f_ns/1e3,pwm_spec_ns_dB,'!','markersize',20);
hold on
plot(f_ns/1e3,pwm_spec_ns_dB);
hold off
xlabel('kHz');
ylabel('dB');
axis([0 400 -160 0]);
Figure(4)
plot(f_ns/1e3,pwm_spec_ns_dB,'!','markersize',20);
hold on
plot(f_ns/1e3,pwm_spec_ns_dB);
hold off
xlabel('kHz');
ylabel('dB');
axis([0 20 -160 0]);

```



### C3. Trailing edge PNPWM using binary search strategy

```

%Programmer   : Deon Jacobs
%Date        : 29 October 2005
%Goal        : Polynomial interpolation using the direct method
%Filename    : direct_pol_int.m

clc;
clear;
fsize = 18;
lw = 2;
set(0,'DefaultAxesColorOrder',[0 0 0],'DefaultLineLineWidth',lw);
set(0,'DefaultFontSize',18,'DefaultAxesFontSize',18);

O   = 8;                % Order of interpolation polynomial
O_slc = ((O+1)/2)-1;    % Samples on the left of the centre of O+1
L   = 8;                % Interpolation ratio
divider = 10584;        % Divider of sampling frequency to get tonal frequency
N   = divider+8;        % Sequence length
n   = [0:1:N];          % Sequence vector
n_up = [0:1:(N*L)-1];  % Interpolated sequence vector
L1  = 1;                % Amount of experiments
A   = 0.95;             % Amplitude cosine input signal
Na  = 0;                % Amplitude of noise superiposed on cosinal signal
Z   = 0;                % Zeros pended after input signal has been generated
Fs  = 352.8e3;          % Sampling frequency of the input
Ts  = 1/Fs;             % Sampling period of input signal
F1  = 1e3;              %Fs/divider; % Frequency of the cosinal input
F2  = 1e3;
Fs_up = L*Fs;           % The interpolated sampling frequency
F_end = Fs;
intervals = 15;        % Number of intervals until 24bit crosppoint resolution is found

%-----
%Generated signals
%-----

%Cosinal input sequence
%[xn] = data_array(n,F1,Fs,Na,L1,A,Z);
xn = A*cos(2*pi*(F2/Fs)*n);
%Interpolated input sequency using matlab
[xn_up_mat] = data_array(n_up,F1,Fs_up,Na,L1,A,Z);

%-----
% Determine polynomial coefficients
%-----

% Xa=b
% a=polynomial coefficients
% X=design matrix
% b=input samples
x = [0:1:O];                %O sample index vector
for i=[1:1:O+1],           %Row array loop
    matrix_temp = 1;
    for j=[2:1:O+1];        %Column fill in
        matrix_temp = [matrix_temp x(i)^(j-1)];
    end;
    X(i,:) = matrix_temp;
end;
% Inversion of the square design matrix
X_inv = inv(X);
% Scale X_inv until no decimals are present
X_inv_scale = X_inv*2^29;

```

```

% Restructure the scaled inverse matrix to be in vector format
% for use in a lookup table
for y = [1:1:O+1],
    if (y==1)
        X_inv_sc_vec = X_inv_scale(y,1);
    else
        X_inv_sc_vec = [X_inv_sc_vec X_inv_scale(y,:)];
    end;
end;
X_inv_sc_vec = X_inv_sc_vec';

%-----
% Calculation of system matrix for interpolated points
%-----

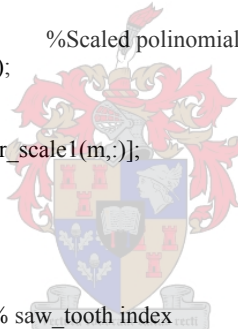
x_int = [4:(1/L):5]; % interval of interpolation from vector x
pol_arr = [];
% Calculate the characteristic polynomial design matrix with the use of x_int
% x_int is the interval where the polynomial needs to be calculated
for i=[1:1:L+1], %Row array loop
    pol_temp = 1;
    for j=[2:1:O+1]; %Column fill in
        pol_temp = [pol_temp x_int(i)^(j-1)]; %System matrix row [1 x x^2...x^(O-1)]
    end;
    pol_arr(i,:) = pol_temp;
end;
pol_arr_scale = (pol_arr*2^10); %Scaled polinomial matrix
pol_arr_scale1 = pol_arr_scale(:,2:end);
pol_mat_vec = [];
for m=[2:1:L+1],
    pol_mat_vec = [pol_mat_vec pol_arr_scale1(m,:)];
end;
pol_mat_vec = pol_mat_vec';

% Matrix multiplication (a = X_inv*b)
% and interpolated point calculation
x_saw = [0:1:L]; % saw_tooth index
m = 2/(L); % gradient of the saw_tooth
c = -1; % crosspoint of saw-tooth waveform with vertical axis
saw_int = (m*x_saw +c); % Saw-tooth waveform values to find the first tolerance interval

saw_int_scale = (saw_int*2^23);
yn_up = [];
x1_vec = [];
x2_vec = [];
pwm_width = [];
pwm_width_24bit = [];
y_int_cubic = [];
for j=[((O+1)/2):1:(n(end)-((O+1)/2))],

    %Extract O+1 samples from the data record with only one new sample each
    %time
    xn_sampl = (xn(j-O_slc:(j+((O+1)/2))))';
    %Matrix multiplication to get O+1 coefficients
    b = X_inv*xn_sampl;
    %Matrix multiplication for L+1 interpolated points between vector x = 4
    %and x = 5 (interpolated point at x=5 is included)
    int_points = pol_arr*b; %Calculate interpolated output
    yn_up = [yn_up int_points(1:end-1)]; %Interpolated output
    [x1,x2] = binary_search(int_points,saw_int); %Search interval of crosspoint with saw_tooth
    x1_vec = [x1_vec x1]; %Build x1 coordinate vector
    x2_vec = [x2_vec x2]; %Build x2 coordinate vector
    %Determine the crosspoint between saw_tooth and consecutive samples of xn
    [pwm_temp] = crosspoint(x1,x2,int_points(x1),int_points(x2),m); %Linear interpolation
                                                    crosspoint derivation

```



```

%Second interval crosspoint derivation
[x1_int,pol_int,saw_tooth] = Crosspoint_24bit(x1,int_points(x1),int_points(x2),m,c,intervals)
pwm_width = [pwm_width pwm_temp]; %Build PWM width vector
pwm_width_24bit = [pwm_width_24bit x1_int];

end;
pwm_width = (pwm_width/L)*Ts;
pwm_width_24bit = (pwm_width_24bit/L)*Ts;

[pwm_spec,f] = PWM_spectra(pwm_width,Fs,F1,F_end); % Compute PWM spectra
[pwm_spec_24bit,f] = PWM_spectra(pwm_width_24bit,Fs,F1,F_end); % Compute PWM spectra
pwm_spec_24bit_norm = pwm_spec_24bit/max(pwm_spec_24bit);

pwm_spec_dB = 10*log10(abs(pwm_spec).^2);
pwm_spec_24bit_dB = 10*log10(abs(pwm_spec_24bit_norm).^2);
pwm_spec_dB = pwm_spec_dB - max(pwm_spec_dB);

xn_up_mat_par = xn_up_mat(33:end-40);
% Figure(1)
% stem(yn_up);
% hold on
% stem(xn_up_mat_par,'r');
% hold off

% Figure(2)
% plot(x1_vec);
% hold on
% plot(x2_vec,'r');
% hold off
%
% Figure(3)
% stem(pwm_width,'markersize',1);

% Figure(4)
% plot(f,pwm_spec_dB,'.');
% title('19.6 KHz sinusoidal input, 512 interval search');

% Figure(6)
% plot(f,pwm_spec,'.');
```



```

% Figure(5)
% plot(y_int_cubic);

%PNPWM output spectra
Figure(6)
subplot(1,2,1)
plot(f/1e3,pwm_spec_24bit_dB,'.', 'markersize',20);
hold on
plot(f/1e3,pwm_spec_24bit_dB);
hold off
box off
title('Frequency Content Showing Carrier Harmonics','FontSize',14);
xlabel('(a) kHz');
ylabel('dB');
axis([0 400 -350 0]);
subplot(1,2,2)
plot(f/1e3,pwm_spec_24bit_dB,'.', 'markersize',20);
hold on
plot(f/1e3,pwm_spec_24bit_dB);
hold off
box off
title('Zoomed View of Baseband Frequency Content','FontSize',14);
xlabel('(b) kHz');
%ylabel('dB');
axis([0 20 -350 0]);

```

# Appendix D

## DIR 1703

SLES007– JULY 2001

### DIGITAL AUDIO INTERFACE RECEIVER

#### FEATURES

- Standard Digital Audio Interface Receiver (EIAJ1201)
- Sampling Rate: 32 / 44.1 / 48 / 88.2 / 96 kHz
- Recover 128 / 256 / 384 / 512  $f_s$  System Clock
- Very Low Jitter System Clock Output (75 ps Typically)
- On-Chip Master Clock Oscillator, Only an External Crystal Is Required:  
24.576 / 22.5792 / 18.432 / 16.9344 / 16.384 / 12.288 / 11.2896 / 8.192 / 6.144 / 5.6448 / 4.096 MHz Crystals Are Available
- Selectable Output PCM Audio Data Format
- Selectable Crystal Clock and PPL Clock Operation Mode
- Output User Bit Data, Flag Signals, and Channel Status Data With Block Start Signal
- Single 3.3-V Power Supply
- Package: 28 SSOP

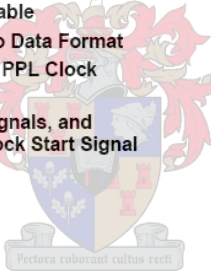
#### APPLICATIONS

- AV Receiver
- MD Player
- DAC Unit

#### DESCRIPTION

The DIR1703 is a digital audio interface receiver (DIR) which receives and decodes audio data up to 96 kHz according to the AES/EBU, IEC958, S/PDIF, and EIAJCP340/1201 consumer and professional format interface standards. The DIR1703 demultiplexes the channel status bit and user bit directly to serial output pins, and has dedicated output pins for the most important channel status bits. It also includes extensive errors reporting.

The significant advantages of the DIR1703 are *96-kHz sampling rate capability* and *Low-jitter clock recovery by the Sampling Period Adaptive Controlled Tracking (SpAct™) system*. The input signal is reclocked with the patented *Sampling period Adaptive controlled tracking system* for maximum quality. These features are required for recent consumer and professional audio instruments, in which the DIR has an interface to any kind of delta-sigma type ADC/DAC with a 96-kHz sampling rate.



This integrated circuit can be damaged by ESD. Burr-Brown recommends that all integrated circuits be handled with appropriate precautions. Failure to observe proper handling and installation procedures can cause damage.

ESD damage can range from subtle performance degradation to complete device failure. Precision integrated circuits may be more susceptible to damage because very small parametric changes could cause the device not to meet its published specifications.



Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

SpAct and Burr-Brown are trademarks of Texas Instruments.

PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

**TEXAS  
INSTRUMENTS**  
www.ti.com

Copyright © 2001, Texas Instruments Incorporated



## DIR1703

SLES007–JULY 2001

### basic operation theory

The DIR1703 is operated as either a PLL clock operation mode or a crystal clock operation mode. These basic operation modes are user selectable.

Sampling period adaptive controlled tracking system (SpAct) is a newly developed clock recover architecture, giving very low jitter clock from S/PDIF data input.

The DIR1703 has two PLLs, PLL1 and PLL2. SpAct is supplied with a 100 MHz executing clock from PLL1.

The DIR1703 requires system clock input for operation of SpAct at both the PLL clock operation mode and the crystal clock operation mode. This system clock can be obtained by connecting a crystal resonator at the XTI/XTO pins or applying an external clock input at the XTI pin as shown in Figure 1.

PLL2 generates the system clock SCKO by using the output signal of the SpAct. The source of SCKO, either OSC (crystal) or PLL2, is selected by the CKSEL pin (called PLL clock operation mode and crystal clock operation mode).

In the PLL clock operation mode, when the S/PDIF signal goes to noninput, SCKO may hold the latest tracked frequency.

Also, the DIR1703 indicates the unlocked state by a high level output at the UNLOCK pin. When the S/PDIF signal restarts, the analog PLL will lock to the incoming S/PDIF signal with very low jitter. The PLL lock-in time is around 1 ms using the SpAct.

Then, the DIR1703 indicates the locked status by a low output at the UNLOCK pin. In this status, the BRATE pins simultaneously indicate the bit rate of the incoming S/PDIF signal.

After  $\overline{\text{RST}}$  (pin 21) is removed, SCKO is set to the default frequency, which can be selected by the BRSEL and SCF pins. The sampling rate ( $f_S$ ), 32 k, 44.1 k, 48 k, 88.2 k, or 96 k is selected by the BRSEL pin. The system clock frequency, 128, 256, 384, or 512  $f_S$  is also selected by the SCF pins.

In the crystal clock operation mode, the crystal oscillator generates three audio clocks SCKO, BCKO, and LRCKO. In this mode, DOUT is always set to mute (zero). BRATE and UNLOCK can be indicated according to the incoming S/PDIF signal.

If CKSEL (pin 28) is connected to UNLOCK (pin 27), which indicates the S/PDIF decoding status and the PLL2 lock-state, the system clock source can be selected automatically when the S/PDIF signal is active and the bit rate is detected.

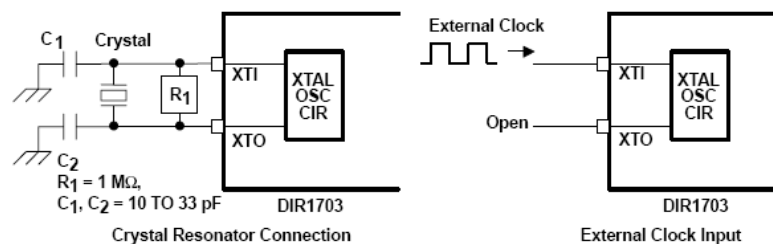


Figure 1. System Clock Connections

## DIR1703

SLES007–JULY 2001

**system clock output**

The primary function of the DIR1703 is to recover audio data and a low jitter clock from a digital audio transmission line. The system clock (SCKO) can be selected in two clocks that are generated by the crystal oscillator clock (crystal mode) or the PLL clock (PLL mode) by the SpAct.

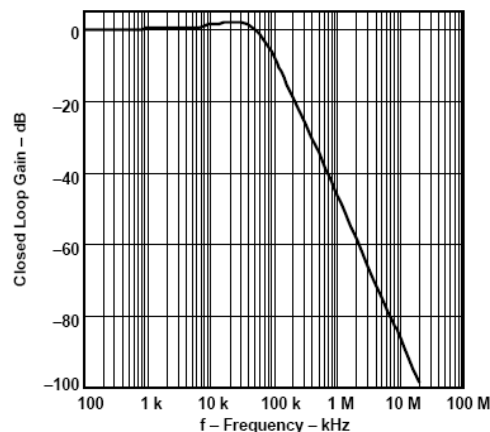
The two operation modes are selected by the CKSEL pin. In the PLL clock operation mode, the clock that can be generated is SCKO ( $128 / 256 / 384 / 512 f_S$ , shown in Table 1), BCKO ( $64 f_S$ ), and LRCKO ( $1 f_S$ ). SCKO is the output of the voltage controlled oscillator (VCO) in an analog PLL. The PLL function consists of a VCO, phase and frequency detector, and an external second-order loop filter. The closed-loop transfer function, which specifies the PLL jitter attenuation characteristics, is shown in Figure 2. In the crystal clock operation mode, SCKO can be generated from several crystal oscillators shown in Table 2.

The crystal frequency should be defined for internal PLL by connecting the BRSEL pin to one of the output pins BFRAME, EMFLG, URBIT, or CSBIT as shown in Table 3. A 12.288 MHz crystal resonator can be used for  $96\text{-kHz} - 128 f_S$  (CSBIT),  $48\text{-kHz} - 256 f_S$  (OPEN) and  $32\text{-kHz} - 384 f_S$  (BFRAME). If BRSEL is not connected to any pins, the 48-kHz sampling rate is selected. The system clock frequency of both modes can be selected by control data at SCF0 and SCF1 pins shown in Table 4.

Table 5 shows the state of the system and the condition of audio clocks and flags in both the PLL and crystal operation modes. In the crystal clock operation mode, SpAct also detects the bit rate of the incoming S/PDIF signal and indicates the state at the UNLOCK pin. Therefore, by connecting CKSEL pin 28) to UNLOCK (pin 27), the system clock source can be selected automatically when the S/PDIF signal arrives and the bit rate is detected. The required accuracy for clock frequency of the crystal resonator or external clock input is  $\pm 500$  ppm.

**Table 1. Generated System Clock (SCKO) PLL Clock Operation Mode**

SAMPLING RATE	128 $f_S$	256 $f_S$	384 $f_S$	512 $f_S$
32 kHz	yes	yes	yes	yes
44.1 kHz	yes	yes	yes	yes
48 kHz	yes	yes	yes	yes
88.2 kHz	yes	yes	yes	yes
96 kHz	yes	yes	yes	yes

**Figure 2. Jitter Attenuator Characteristics With Specified Loop Filter**

# Appendix E

## ALTERA CYCLONE (EP1C12Q240C6) Features

Cyclone Device Handbook, Volume 1

### Features

The Cyclone device family offers the following features:

- 2,910 to 20,060 LEs, see [Table 1-1](#)
- Up to 294,912 RAM bits (36,864 bytes)
- Supports configuration through low-cost serial configuration device
- Support for LVTTTL, LVCMOS, SSTL-2, and SSTL-3 I/O standards
- Support for 66- and 33-MHz, 64- and 32-bit PCI standard
- High-speed (640 Mbps) LVDS I/O support
- Low-speed (311 Mbps) LVDS I/O support
- 311-Mbps RSDS I/O support
- Up to two PLLs per device provide clock multiplication and phase shifting
- Up to eight global clock lines with six clock resources available per logic array block (LAB) row
- Support for external memory, including DDR SDRAM (133 MHz), FCRAM, and single data rate (SDR) SDRAM
- Support for multiple intellectual property (IP) cores, including Altera® MegaCore® functions and Altera Megafunctions Partners Program (AMPP<sup>SM</sup>) megafunctions.

**Table 1-1. Cyclone Device Features**

Feature	EP1C3	EP1C4	EP1C6	EP1C12	EP1C20
LEs	2,910	4,000	5,980	12,060	20,060
M4K RAM blocks (128 × 36 bits)	13	17	20	52	64
Total RAM bits	59,904	78,336	92,160	239,616	294,912
PLLs	1	2	2	2	2
Maximum user I/O pins (1)	104	301	185	249	301

*Note to Table 1-1:*

(1) This parameter includes global clock pins.

# Appendix F

## VHDL Code

### F1. Polyphase filtering

```

-----
-- Design unit      : FIR polyphase filter
-- File name       : FIR_poly.vhd
-- Description      : Implements a FIR filter using the polyphase filtering technique:
--                  : => 8 polyphase filters are implemented within the 44.1Khz sampling frequency,
--                  : => Therefore the input is upsampled 8 times, increasing the
--                  :                  sampling frequency to 352.8 Khz.
--                  : => The main state machine executes at a rate of 50MHz, allowing a filter
--                  :                  length of 632 coefficients(L), thus 8 polyphase filters having 79 coefficients each(P).
--                  : => The state machine is implemented using a two process structure ensuring that
--                  :                  Quartus II synthesizes it correctly.
--                  : => The filter implementation makes use of 3 Megafunctions: 1-port ROM, 1-port RAM, MAC
--                  :                  function.
--                  :                  * The ROM is used for the Input buffer
--                  :                  * The RAM is used for the Coefficient lookup table
--                  :                  * The MAC function is used to implement the multiply and accumulate arithmetic
--                  : => The input buffer (where input samples are stored) has a length equal to P+1,
--                  :                  where the first buffer address value is zero to enforce a zero result from
--                  :                  the MAC function where necessary.
--                  : => The coefficient lookup table's length equals L= 632 taps/coefficients, where the
--                  :                  last address value memory is zero to force a zero result from the MAC if the input buffer
--                  :                  isn't able to do it.
--                  : => The third process (Latch_FIR_output) latches the Data_temp register as output data at
--                  :                  352.8KHz.
--                  : Data_temp captures the top 16-bits of the Polyphase FIR filter output, within the filtering
--                  :                  process.
-- System          : VHDL'93
-- Author          : Deon Jacobs
--                  : Department of Electrical Engineering
--                  : University of Stellenbosch
--                  : Deonj@sun.ac.za
-- Revision       : Version 4.1 12/05/2005
-----

```

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

```

ENTITY FIR_poly IS
PORT ( clk50Meg_in      : IN STD_LOGIC;          -- 50 MHz Clock input
       clk352KHz_in   : IN STD_LOGIC;          -- 352.8 KHz clock input
       reset_in        : IN STD_LOGIC;          -- Signal to reset
       pulse_44_1_in  : IN STD_LOGIC;          -- Data pulse (44.1Khz)=> input data ready
       data_in         : IN STD_LOGIC_VECTOR(15 downto 0); -- 16 bit data input
       filt_stat_out   : OUT STD_LOGIC;          -- Filter status output
       LED_out         : OUT STD_LOGIC;          -- State indicators
       Poly_out        : OUT STD_LOGIC;          -- Polyphase output pulse indicator
       Data_out        : OUT STD_LOGIC;
       latch_out       : OUT STD_LOGIC_VECTOR(15 downto 0)
    );
END ENTITY FIR_poly;
```

```
ARCHITECTURE Poly_do OF FIR_poly IS
```

```

COMPONENT MAC
PORT
(
    dataa      : IN STD_LOGIC_VECTOR (50 DOWNT0 0);
    datab     : IN STD_LOGIC_VECTOR (15 DOWNT0 0);
    clock0    : IN STD_LOGIC := '1';
    aclr0     : IN STD_LOGIC := '0';
    result    : OUT STD_LOGIC_VECTOR (66 DOWNT0 0)
);
END COMPONENT;

COMPONENT Buf
PORT
(
    address   : IN STD_LOGIC_VECTOR (6 DOWNT0 0);
    clock    : IN STD_LOGIC ;
    data     : IN STD_LOGIC_VECTOR (15 DOWNT0 0);
    wren    : IN STD_LOGIC ;
    q       : OUT STD_LOGIC_VECTOR (15 DOWNT0 0)
);
END COMPONENT;

COMPONENT Coef
PORT
(
    address   : IN STD_LOGIC_VECTOR (9 DOWNT0 0);
    clock    : IN STD_LOGIC ;
    q       : OUT STD_LOGIC_VECTOR (50 DOWNT0 0)
);
END COMPONENT;

-- State machine decleration
TYPE State_machine IS (state_0,state_1,state_2,state_3,state_4,state_5);
SIGNAL current_state,next_state : State_machine;
-----
-- Combinational logic signals
-----
-- Misc
SIGNAL cstate1_f_sig : STD_LOGIC;           -- One clock cycle wait flag in state1
SIGNAL cstate3_cnt_sig : unsigned(7 downto 0); -- State_3 wait cnt signal
SIGNAL cstate4_f_sig : STD_LOGIC;           -- One clock cycle wait flag in state4
SIGNAL cFil_stat_f_sig : STD_LOGIC;         -- Polyphase filtering status flag
SIGNAL cLED_sig : STD_LOGIC;               -- LED State status indicator
SIGNAL cPoly_finish_f_sig: STD_LOGIC;       -- Polyphase filter has finished indicator flag
SIGNAL cpf_cnt_max : unsigned(7 downto 0); -- Polyphase finished max wait clk cycles
SIGNAL cData_temp : STD_LOGIC_VECTOR(15 downto 0); -- Temp signal to store poly
                                                    filter output

-- MAC
SIGNAL cMAC_clr_sig : STD_LOGIC;           -- clear
SIGNAL cMAC_datab_sig : STD_LOGIC_VECTOR(15 downto 0);-- Data input from the input buffer

-- Input Buffer
SIGNAL cBuf_adr_sig : unsigned(6 downto 0); -- address
SIGNAL cBuf_dat_sig : STD_LOGIC_VECTOR(15 downto 0); -- data input
SIGNAL cBuf_wren_sig : STD_LOGIC;         -- Write/read pin

-- Coeficient Lookup Table
SIGNAL cCoef_adr_sig : unsigned(9 downto 0); -- Coeficient lookup table address

```

```

-----
-- Latched output (memory) signals
-----
-- Misc
SIGNAL state1_f_sig      : STD_LOGIC;           -- One clock cycle wait flag in state1
SIGNAL state3_cnt_sig    : unsigned(7 downto 0); -- State_3 wait cnt signal
SIGNAL state4_f_sig      : STD_LOGIC;           -- One clock cycle wait flag in state4
SIGNAL Poly_finish_f_sig : STD_LOGIC;           -- Polyphase filter indicator flag
SIGNAL pf_cnt_max        : unsigned(7 downto 0); -- Polyphase finished max wait clk cycles
SIGNAL Data_temp         : STD_LOGIC_VECTOR(15 downto 0); -- Temp signal to store poly
                                                                filter output

-- MAC
SIGNAL MAC_clr_sig       : STD_LOGIC;           -- MAC clear signal
SIGNAL MAC_dataa_sig     : STD_LOGIC_VECTOR(50 downto 0); -- MAC dataA input signal
SIGNAL MAC_datab_sig     : STD_LOGIC_VECTOR(15 downto 0); -- MAC dataB input signal
SIGNAL MAC_result_sig    : STD_LOGIC_VECTOR(66 downto 0);

-- Input Buffer
SIGNAL Buf_adr_sig       : unsigned(6 downto 0); -- address
SIGNAL Buf_out_sig       : STD_LOGIC_VECTOR(15 downto 0); -- data output
SIGNAL Buf_dat_sig       : STD_LOGIC_VECTOR(15 downto 0); -- data input
SIGNAL Buf_wren_sig      : STD_LOGIC;           -- Write/read pin

-- Coefficient Lookup Table
SIGNAL Coef_adr_sig      : unsigned(9 downto 0); -- Coefficient lookup table address

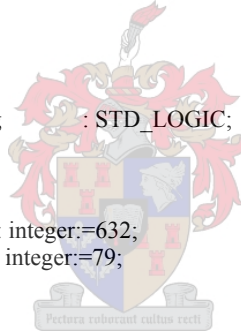
-----
--Latch_FIR_output Signals
-----
SIGNAL Data_capture_f_sig : STD_LOGIC; -- Flag indicating that output data
                                                                -- from polyphase filter has been latched

-- Constant values
CONSTANT coef_cnt_max : integer:=632; -- Maximum coefficient address range
CONSTANT buf_cnt_max  : integer:=79;  -- Maximum input buffer address range

BEGIN

MAC_inst : MAC PORT MAP (
    dataa    => MAC_dataa_sig,
    datab   => MAC_datab_sig,
    clock0   => clk50Meg_in,
    aclr0    => MAC_clr_sig,
    result   => MAC_result_sig
);
Buf_inst : Buf PORT MAP (
    address => std_logic_vector(unsigned(Buf_adr_sig(6 downto 0))),
    clock   => clk50Meg_in,
    data    => Buf_dat_sig,
    wren    => Buf_wren_sig,
    q       => Buf_out_sig
);
Coef_inst : Coef PORT MAP (
    address => std_logic_vector(unsigned(Coef_adr_sig(9 downto 0))),
    clock   => clk50Meg_in,
    q       => MAC_dataa_sig
);

```



```
-----
Register_process:
-----
```

```
PROCESS (clk50Meg_in,reset_in,cCoef_adr_sig,cBuf_adr_sig,cstate3_cnt_sig,cBuf_dat_sig) IS
```

```
BEGIN
```

```
IF (reset_in = '1') THEN
```

```
-- Misc
filt_stat_out    <='0';           -- Filtering is not in progress
Coef_adr_sig     <= "0000000000"; -- Reset counter
LED_out         <='1';           -- " LEDS state indicator
state1_f_sig     <='0';           -- " State1 flag
state3_cnt_sig   <= "00000000";  -- " State3 wait counter
state4_f_sig     <='0';           -- " State4 flag
Poly_finish_f_sig <='0';         -- " flag
pf_cnt_max      <="00110010";    -- Set to maximum_wait of 10 clock cycles
Data_temp       <="0000000000000000"; -- Inisialize temp data signal

-- MAC
MAC_clr_sig     <= '1';           -- Clear MAC output
-- Input Buffer
Buf_adr_sig     <= "0000000";     -- Reset Buffer address to output zero (0h)
Buf_dat_sig     <= "0000000000000000"; -- " data input
Buf_wren_sig    <= '0';           -- Buffer in read mode
current_state   <= state_0;      -- Force state machine to reset state
```

```
ELSIF rising_edge(clk50Meg_in) THEN
```

```
-- Update latches (memory units)
--Misc registers
Coef_adr_sig    <= cCoef_adr_sig; -- Coeficient Address
LED_out        <= cLED_sig;       -- State indicate output
filt_stat_out   <= cFil_stat_f_sig; -- Filter completion indicate
state1_f_sig    <= cstate1_f_sig;  -- State_1 one clk cycle wait flag
state3_cnt_sig  <= cstate3_cnt_sig; -- State_3 wait flag
state4_f_sig    <= cstate4_f_sig;  -- State_4 one clk cycle wait flag
Poly_finish_f_sig <= cPoly_finish_f_sig;
pf_cnt_max     <= cpf_cnt_max;    -- Updata maximum clock cylces wait (State3)
Data_temp      <= cData_temp;     -- Update data temp
--MAC registers
MAC_datab_sig  <= cMAC_datab_sig; -- Update MAC datab input from input buffer
Buf_adr_sig    <= cBuf_adr_sig;
-- Address input
Buf_dat_sig    <= cBuf_dat_sig;
-- Data input
Buf_wren_sig   <= cBuf_wren_sig;
Poly_out       <= Poly_finish_f_sig;
Data_out       <= Data_capture_f_sig;

-- State register
current_state  <= next_state;    -- Assign next state as current state
```

```
END IF;
```

```
END PROCESS Register_process;
```

```
-----
Combinational_process:
-----
```

```
PROCESS (current_state,data_in,pulse_44_1_in,Coef_adr_sig,Buf_adr_sig,state1_f_sig,state3_cnt_sig,
MAC_datab_sig,Buf_dat_sig,Buf_out_sig,state4_f_sig,Poly_finish_f_sig,pf_cnt_max,
Data_temp, Data_capture_f_sig, MAC_result_sig) IS
```

```
BEGIN
```

```
CASE (current_state) IS
```



```

-----
-- Reset state
-----
When state_0 =>
-- Mealy machine
  cFil_stat_f_sig <='0';           -- Filtering hasn't started
  cLED_sig <='0';                 -- Reset state
  cCoef_adr_sig <="0000000000";   -- Reset counter
  cMAC_clr_sig <='1';             -- Clear MAC output: no data yet
  cMAC_datab_sig <="0000000000000000";
  cBuf_adr_sig <="00000000";      -- Reset Buffer address to output zero
  cBuf_dat_sig <="0000000000000000"; -- " data input
  cBuf_wren_sig <='0';           -- Write zero at the 1st buffer address
  cstate1_f_sig <='0';           -- State1 wait flag reset
  cstate3_cnt_sig <="00000000";  -- Reset State3 wait counter
  cstate4_f_sig <='0';           -- Reset state4 flag
  cPoly_finish_f_sig<= Poly_finish_f_sig; -- Reset Polyphase filter finished flag
  cpf_cnt_max <="00110010";      -- Set to maximum_wait of 10 clock cycles
  cData_temp <= Data_temp;       -- Keep Data Temp at its current value
  IF (pulse_44_1_in = '1') THEN  -- Have new data arrived
    next_state <= state_1;      -- Go store data
  ELSE
    next_state <= state_0;      -- Stay in reset and wait for data
  END IF;
-----
-- Data arrived: store it in input buffer at address 0h
-----
When state_1 =>
  cFil_stat_f_sig <='1';         -- Filtering process busy
  cLED_sig <='0';               -- Storage LED indicate
  cMAC_clr_sig <='0';           -- Set MAC to start arithmetic in next state
  cstate3_cnt_sig <="00000000"; -- Reset State3 wait counter
  cstate4_f_sig <='0';         -- Reset state4 flag
  cMAC_datab_sig <= Buf_out_sig;
  cPoly_finish_f_sig<= Poly_finish_f_sig; -- Reset Polyphase filter finished flag
  cpf_cnt_max <="00110010";     -- Set to maximum_wait of 10 clock cycles
  cData_temp <= Data_temp;     -- Keep Data Temp at its current value

  IF (state1_f_sig='0') THEN
    -- Inputs of MAC are used for calculation
    cBuf_dat_sig <= data_in;     -- New sample applied to input buffer
    cBuf_wren_sig <='1';        -- Write new sample to buffer
    cstate1_f_sig <='1';        -- Wait cycle entered: indicate
    cCoef_adr_sig <="0000000000"; -- Zero Coefficient address
    cBuf_adr_sig <="00000001";  -- Reset buffer address (1h) to 1st input
                                -- sample
    next_state <= state_1;      -- Wait to ensure data has been stored
  ELSE
    -- Second Arithmetic calculation of MAC inputs
    cBuf_dat_sig <="0000000000000000";
    cBuf_wren_sig <='0';
    cstate1_f_sig <='0';        -- Reset flag for next new sample
                                -- occurrence

    cCoef_adr_sig <="0000000000";
    cBuf_adr_sig <="00000010";  -- 2nd buffer address for arithmetic
                                -- calculation
    next_state <= state_2;      -- Start Controlling filter operation
  END IF;
-----
-- Filter control
-----
When state_2 =>
  cFil_stat_f_sig <='1';         -- Filtering process busy
  cLED_sig <='0';               -- Filter control indicate

```



```

cstate1_f_sig    <= '0';          -- Reset state_1 wait cycle flag
cstate3_cnt_sig  <= "00000000";  -- Reset State3 wait counter
cstate4_f_sig    <= '0';          -- Keep State4 first entrance flag value
                                   equal 2 zero
cPoly_finish_f_sig<= Poly_finish_f_sig; -- Reset Polyphase filter finished flag
cpf_cnt_max     <= "00110010";   -- Keep current value
cData_temp      <= Data_temp;    -- Keep Data Temp at its current value

-- Test for filter completion
IF (to_integer(Coef_adr_sig) < coef_cnt_max) THEN
  -- Whole filter has not been completed
  cMAC_clr_sig   <= '0';          -- Start/Continue arithmetic

  -- Test for buffer content shift when last poly-filt is executing
  IF (to_integer(Coef_adr_sig) >= (coef_cnt_max - buf_cnt_max-1)) THEN

    -- Last polyphase filter execution => Buffer shift operation
    cCoef_adr_sig <= Coef_adr_sig+1; -- Keep Coef address at
                                   current value
    cBuf_adr_sig  <= "0000001";    -- Buffer address equal to
                                   buffer temp value

    cMAC_datab_sig <= Buf_out_sig;
    cBuf_wren_sig  <= '0';
    cBuf_dat_sig   <= "0000000000000000"; -- Reset input data to
                                   Input Buffer

    next_state    <= state_4;
  ELSE
    -- Not last polyphase filter execution
    cMAC_datab_sig <= Buf_out_sig;
    cBuf_dat_sig   <= "0000000000000000";
    cBuf_wren_sig  <= '0';
    cCoef_adr_sig <= Coef_adr_sig + 1; -- Set address to next
                                   coefficient value

    -- Test if current polyphase filter operation is finished
    IF (to_integer(Buf_adr_sig) < buf_cnt_max) THEN
      -- polyphase filter not finished yet
      cBuf_adr_sig <= Buf_adr_sig + 1;
      next_state   <= state_2;
    ELSE
      -- polyphase filter is finished, start next one
      cBuf_adr_sig <= "0000000";
      next_state   <= state_3;
    END IF;
  END IF;
END IF;
ELSE
  -- Whole filter has been completed & state 4 did not terminate the filtering,
  -- this ELSE is therefore pre-cautionary for state 2 not to hang/loop

  cMAC_datab_sig <= "0000000000000000";
  cMAC_clr_sig   <= '1'; -- Reset MAC function (stop arithmetic)
  cCoef_adr_sig  <= "0000000000";
  cBuf_adr_sig   <= "0000000";
  cBuf_dat_sig   <= "0000000000000000";
  cBuf_wren_sig  <= '0';
  next_state     <= state_0; -- Enter reset state to wait for new sample
END IF;

-----
-- Polyphase complete
-----
When state_3 =>

cMAC_datab_sig <= Buf_out_sig;
cFil_stat_f_sig <= '1'; -- Filtering process busy
cLED_sig        <= '0'; -- Filter control indicate

```

```

cstate1_f_sig <= '0';          -- Reset state_1 wait cycle flag
cstate4_f_sig <= state4_f_sig; -- Keep value to see if state4 assigned this state
cCoef_adr_sig <= Coef_adr_sig; -- Keep Coef lookup table at current address
cBuf_dat_sig <= "0000000000000000"; -- Reset input data to Input Buffer
cBuf_wren_sig <= '0';          -- Buffer in read mode
cpf_cnt_max <= pf_cnt_max;    -- Keep current value

-- Did the delay complete between polyphase filters
IF (state3_cnt_sig < pf_cnt_max) THEN
  IF ((to_integer(state3_cnt_sig) = 15) or (state4_f_sig = '1')) THEN
    -- Decision when between consecutive poly filters
    IF ((Poly_finish_f_sig = '0') and (Data_capture_f_sig = '0')) THEN
      cData_temp <= MAC_result_sig(66) &
        MAC_result_sig(63 DOWNTO 49);
        -- Copy top MS-16-bits to
        -- temporary register
      cPoly_finish_f_sig <= '1'; -- Toggle Poly flag according
        -- to current instance
      cstate4_f_sig <= '0';    -- Make sure state4 flag is
        -- reset
    ELSE
      cData_temp <= MAC_result_sig(66) &
        MAC_result_sig(63 DOWNTO 49);
        -- Polyphase filter also
        -- finished but at different
        -- instance
      cPoly_finish_f_sig <= '0'; -- Toggle Poly flag according
        -- to current instance
      cstate4_f_sig <= '0';
    END IF;
  ELSE
    cData_temp <= Data_temp;
    cPoly_finish_f_sig <= Poly_finish_f_sig;
  END IF;
  cstate3_cnt_sig <= state3_cnt_sig + 1; -- State_3 wait cycle entered indicate
  cMAC_clr_sig <= '0';                    -- Allow MAC to finish its calculations
  cBuf_adr_sig <= "0000000";             -- Reset Buffer address to output zero
  next_state <= state_3;
ELSE
  cstate3_cnt_sig <= "00000000"; -- Wait cycle period hasn't ended yet
  cMAC_clr_sig <= '1';           -- Clear MAC for next polyphase
    -- filtering procedure
  cBuf_adr_sig <= "0000001";    -- Reset buffer address (1h) to 1st input
    -- sample
  cData_temp <= Data_temp;
  cPoly_finish_f_sig <= '0';    -- Reset Polyphase filter finished flag
  cData_temp <= Data_temp; -- Keep Data Temp at its current value
  cPoly_finish_f_sig <= Poly_finish_f_sig;
  next_state <= state_2; -- Start next polyphase filter
END IF;
-----
-- Buffer Shift
-----
When state_4 =>

cFil_stat_f_sig <= '1'; -- Filtering process busy
cMAC_clr_sig <= '0'; -- Continue MAC arithmetic function
cLED_sig <= '0'; -- Filter control indicate
cstate1_f_sig <= '0'; -- Reset state_1 wait cycle flag
cstate3_cnt_sig <= "00000000"; -- Reset State3 wait counter
cPoly_finish_f_sig <= Poly_finish_f_sig; -- Reset Polyphase filter finished flag
cData_temp <= Data_temp; -- Keep Data Temp at its current value
-- Has buffer address reached its end
IF (Buf_adr_sig < (Buf_cnt_max+2)) THEN

```

```

    cpf_cnt_max    <= pf_cnt_max;  -- Keep current value
    -- No it hasn't...
    -- Has state4 flag been set
    IF (state4_f_sig = '0') THEN
        --No it hasn't...
        cstate4_f_sig    <='1';      -- For Buffer write mode
        cCoef_adr_sig    <= Coef_adr_sig; -- Keep address the same
        cBuf_adr_sig     <= Buf_adr_sig; -- Keep address the same
        cBuf_wren_sig    <= '1';      -- Buffer in read mode
        cBuf_dat_sig     <= Buf_out_sig;
        cMAC_datab_sig  <= Buf_out_sig; -- Output to MAC for
            multiplication
        next_state      <= state_4; -- Enter Buffer write mode of state
    ELSE
        -- Yes, data will now be written to new shifted address
        cBuf_adr_sig    <= Buf_adr_sig +1; -- Inc Buffer address for next
            multiplication
        cstate4_f_sig    <= '0';      -- For Buffer read mode
        cCoef_adr_sig    <= Coef_adr_sig +1; -- Inc Coeficient address for
            next calculation
        cBuf_wren_sig    <= '0';      -- Write data to Buffer
        cBuf_dat_sig    <= "0000000000000000"; -- Reset input data
            to Input Buffer
        cMAC_datab_sig  <= "0000000000000000"; -- Reset MAC input
            B, for zero
            multiplication result
        next_state<= state_4;      -- Enter Buffer read mode of state
    END IF;
ELSE
    --Buffer shift completed
    cBuf_adr_sig    <= "00000000";
    cMAC_datab_sig  <= "0000000000000000";
    cCoef_adr_sig    <= "1001111000"; -- Address that stores zero value, for
        zero value MAC result
    cBuf_dat_sig    <= "0000000000000000"; -- Reset input data to Input
        Buffer
    cBuf_wren_sig    <= '0';      -- Buffer in read mode
    cstate4_f_sig    <= '1';      -- Set to inform state3 filter is finished
    cpf_cnt_max     <= "00000001"; -- Change max value to 1
    next_state      <= state_5;
END IF;

```

When state\_5 => -- Extra state to wait one clock cycle

```

    cFil_stat_f_sig    <= '1';  -- Filtering process busy
    cMAC_clr_sig       <= '0';  -- Continue MAC arithmetic function
    cLED_sig           <= '0';  -- Filter control indicate
    cstate1_f_sig      <= '0';  -- Reset state_1 wait cycle flag
    cstate3_cnt_sig    <= "00000000"; -- Reset State3 wait counter
    cPoly_finish_f_sig <= Poly_finish_f_sig; -- Reset Polyphase filter finished flag
    cData_temp         <= Data_temp; -- Keep Data Temp at its current value
    cBuf_adr_sig       <= "00000000"; -- Reset Buffer address to output zero (0h)
    cMAC_datab_sig     <= "0000000000000000"; -- filter complete no input data
        required
    cCoef_adr_sig      <= "1001111000"; -- Address that stores zero value, for
        zero value MAC result
    cBuf_dat_sig       <= "0000000000000000"; -- Reset input data to Input Buffer
    cBuf_wren_sig      <= '0';  -- Buffer in read mode
    cstate4_f_sig      <= '1';  -- Set to inform state3 filter is finished
    cpf_cnt_max        <= "00000001"; -- Change max value to 1
    next_state         <= state_3;

```

```

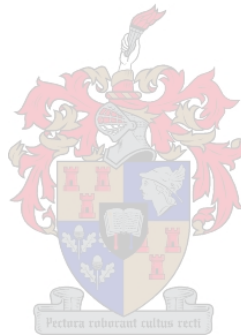
    END CASE current_state;
END PROCESS Combinational_process;

```

```

-----
Latch_FIR_output:
-----
-- Latches top 16-bits of FIR output to register
PROCESS (clk50Meg_in) IS
BEGIN
    IF rising_edge(clk50Meg_in) THEN
        IF (clk352KHz_in = '1') THEN          -- Latch at 352_8KHz
            IF((Poly_finish_f_sig = '1')and(Data_capture_f_sig = '0')) THEN
                Data_capture_f_sig <='1';    -- Set flag according to latch instance
                Latch_out          <= cData_temp; -- Latch Data_temp reg to
                                                output reg
            ELSIF((Poly_finish_f_sig = '0')and(Data_capture_f_sig = '1')) THEN
                Data_capture_f_sig <='0';    -- Set flag according to latch
                                                instance
                Latch_out          <= cData_temp; -- Latch Data_temp reg to
                                                output reg
            END IF;
        END IF;
    END IF;
END PROCESS Latch_FIR_output;
END ARCHITECTURE Poly_do;

```



## F2. Polynomial coefficient calculation

```
-----  
-- Design unit : Polynomial coefficient calculation  
-- File name   : Poly_coef.vhd  
-- Description : Calculates the coefficients of an 8th order polynomial which will be used  
--              : to calculate the crosspoint between the polynomial and a sawtooth waveform  
--              : A new set of coefficients are calculated each time an interpolated data sample  
--              : at a frequency of 352.8 kHz is input. After each of the 9 coefficients  
--              : are calculated they are stored in a ping pong buffer.  
-- System      : VHDL'93  
-- Author      : Deon Jacobs  
--              : Department of Electrical Engineering  
--              : University of Stellenbosch  
--              : Deonj@sun.ac.za  
-- Revision    : Version 2.0 18/08/2005  
-----
```

```
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
USE ieee.numeric_std.all;
```

```
ENTITY poly_coef IS  
PORT (
```

```
    clk_60Meg_in  : IN STD_LOGIC;  
    pulse_352_in  : IN STD_LOGIC;  
    reset_in      : IN STD_LOGIC;  
    sample_in     : IN STD_LOGIC_VECTOR(15 DOWNTO 0);  
    Polcoef_wren_in : IN STD_LOGIC;  
    Polcoef_adr_in : IN STD_LOGIC_VECTOR(4 DOWNTO 0);  
    Coef_sync_out : OUT STD_LOGIC;  
    Input_buf4_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);  
    Input_buf5_out : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);  
    Polcoef_out    : OUT STD_LOGIC_VECTOR(29 DOWNTO 0)
```

```
);  
END ENTITY poly_coef;
```

```
ARCHITECTURE coef_calc OF poly_coef IS
```

```
    COMPONENT input_buf  
    PORT
```

```
    (  
        address : IN STD_LOGIC_VECTOR (3 DOWNTO 0);  
        clock   : IN STD_LOGIC ;  
        data    : IN STD_LOGIC_VECTOR (15 DOWNTO 0);  
        wren    : IN STD_LOGIC ;  
        q       : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
```

```
    );  
END COMPONENT;
```

```
    COMPONENT interpol_mat  
    PORT
```

```
    (  
        address : IN STD_LOGIC_VECTOR (6 DOWNTO 0);  
        clock   : IN STD_LOGIC ;  
        q       : OUT STD_LOGIC_VECTOR (29 DOWNTO 0)
```

```
    );  
END COMPONENT;
```

```

COMPONENT MAC_coef
  PORT
  (
    dataa      : IN STD_LOGIC_VECTOR (29 DOWNT0 0);
    datab     : IN STD_LOGIC_VECTOR (15 DOWNT0 0);
    clock0    : IN STD_LOGIC := '1';
    aclr0     : IN STD_LOGIC := '0'; -- Result high enough resolution for 24-bit input
    result    : OUT STD_LOGIC_VECTOR (45 DOWNT0 0)
  );
END COMPONENT;

COMPONENT pol_coef
  PORT
  (
    data_a      : IN STD_LOGIC_VECTOR (29 DOWNT0 0);
    wren_a      : IN STD_LOGIC := '1';
    address_a   : IN STD_LOGIC_VECTOR (4 DOWNT0 0);
    data_b      : IN STD_LOGIC_VECTOR (29 DOWNT0 0);
    address_b   : IN STD_LOGIC_VECTOR (4 DOWNT0 0);
    wren_b      : IN STD_LOGIC := '1';
    clock_a     : IN STD_LOGIC ;
    clock_b     : IN STD_LOGIC ;
    q_a         : OUT STD_LOGIC_VECTOR (29 DOWNT0 0);
    q_b         : OUT STD_LOGIC_VECTOR (29 DOWNT0 0)
  );
END COMPONENT;

-----
-- State machine declerations
-----
--Matrix multiplication
TYPE State_machine1 IS (state_0,state_1,state_2,state_3,state_4);
SIGNAL current_state1, next_state1 : State_machine1;

--Polynomial coefficient latching
TYPE State_machine2 IS (state_0,state_1);
SIGNAL current_state2, next_state2 : State_machine2;

-----
-- Combinational logic
-----

-- Input Buffer signals

SIGNAL cBuf_dat_sig      : STD_LOGIC_VECTOR(15 DOWNT0 0);
SIGNAL cBuf_wren_sig     : STD_LOGIC;
SIGNAL cBuf_adr_sig      : unsigned(3 DOWNT0 0);

-- Interpolation matrix signals

SIGNAL cMat_adr_sig      : unsigned(6 DOWNT0 0);

-- Coefficient MAC signals

SIGNAL cMAC_coef_datab_sig: STD_LOGIC_VECTOR(15 DOWNT0 0);
SIGNAL cMAC_coef_clr_sig  : STD_LOGIC;

-- Polynomial Coefficient storage buffer

SIGNAL cPolcoef_adr_sig   : unsigned(4 DOWNT0 0);
SIGNAL cPolcoef_wren_sig  : STD_LOGIC;
SIGNAL cPolcoef_dat_sig   : STD_LOGIC_VECTOR(29 DOWNT0 0);
SIGNAL cPolcoef_unused_sig : STD_LOGIC_VECTOR(29 DOWNT0 0);

```

-- Misc signals

```

    SIGNAL ccoef_data_ready_sig : STD_LOGIC;
    SIGNAL cstate3_cnt_sig      : unsigned(4 DOWNTO 0);
    SIGNAL cstate4_f_sig       : STD_LOGIC;
    SIGNAL cfirst_coef_f_sig   : STD_LOGIC;
    SIGNAL cData_temp          : STD_LOGIC_VECTOR(45 DOWNTO 0);
    SIGNAL cCoef_finish_f_sig  : STD_LOGIC;
    SIGNAL cWait_cnt_max       : unsigned(2 DOWNTO 0);
    SIGNAL ccoef_sync_out_sig  : STD_LOGIC;
    SIGNAL cInput_buf4_sig     : STD_LOGIC_VECTOR(15 DOWNTO 0);--1st interval
                                                                polynomial amplitude
    SIGNAL cInput_buf5_sig     : STD_LOGIC_VECTOR(15 DOWNTO 0);--last interval polynomial
                                                                amplitude

```

-----  
-- Latched output (memory) signals  
-----

-- Input Buffer signals

```

    SIGNAL Buf_dat_sig         : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL Buf_wren_sig       : STD_LOGIC;
    SIGNAL Buf_adr_sig        : unsigned(3 DOWNTO 0);
    SIGNAL Buf_out_sig        : STD_LOGIC_VECTOR(15 DOWNTO 0);

```

-- Interpolation matrix signals

```

    SIGNAL Mat_adr_sig        : unsigned(6 DOWNTO 0);

```

-- Coefficient MAC signals

```

    SIGNAL MAC_coef_dataa_sig : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL MAC_coef_clr_sig   : STD_LOGIC;
    SIGNAL MAC_coef_dataaa_sig : STD_LOGIC_VECTOR(29 DOWNTO 0);
    SIGNAL MAC_coef_result_sig : STD_LOGIC_VECTOR(45 DOWNTO 0);

```

-- Polynomial Coefficient storage buffer

```

    SIGNAL Polcoef_adr_sig    : unsigned(4 DOWNTO 0);
    SIGNAL Polcoef_wren_sig  : STD_LOGIC;
    SIGNAL Polcoef_out_sig   : STD_LOGIC_VECTOR(29 DOWNTO 0);
    SIGNAL Polcoef_dat_sig   : STD_LOGIC_VECTOR(29 DOWNTO 0);
    SIGNAL Polcoef_unused_sig : STD_LOGIC_VECTOR(29 DOWNTO 0);

```

-- Misc signals

```

    SIGNAL coef_data_ready_sig : STD_LOGIC;
    SIGNAL state3_cnt_sig      : unsigned(4 DOWNTO 0);
    SIGNAL state4_f_sig       : STD_LOGIC;
    SIGNAL first_coef_f_sig   : STD_LOGIC;
    SIGNAL Data_temp          : STD_LOGIC_VECTOR(45 DOWNTO 0);
    SIGNAL Coef_finish_f_sig  : STD_LOGIC;
    SIGNAL wait_cnt_max       : unsigned(2 DOWNTO 0);
    SIGNAL coef_sync_out_sig  : STD_LOGIC;
    SIGNAL Input_buf4_sig     : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL Input_buf5_sig     : STD_LOGIC_VECTOR(15 DOWNTO 0);

```

-- Constants

```

    CONSTANT Mat_cnt_max    : integer:=73;
    CONSTANT Buf_cnt_max    : integer:=9;
    CONSTANT Zero           : integer:=0;

```





```

        current_state1      <= state_0;-- Force state machine to reset state

ELSIF rising_edge(clk_60Meg_in) THEN
    -- Update latches (memory units)
    -- Interpolation matrix register update
    Mat_adr_sig            <= cMat_adr_sig;

    -- Coeficient MAC registers update
    MAC_coef_datab_sig    <= cMAC_coef_datab_sig;
    MAC_coef_clr_sig      <= cMAC_coef_clr_sig;

    -- Input Buffer registers update
    Buf_adr_sig           <= cBuf_adr_sig;
    Buf_dat_sig           <= cBuf_dat_sig;
    Buf_wren_sig          <= cBuf_wren_sig;
    -- Misc registers
    state3_cnt_sig        <= cstate3_cnt_sig;
    state4_f_sig          <= cstate4_f_sig;
    frst_coef_f_sig       <= cfrst_coef_f_sig;
    Data_temp             <= cData_temp;
    Coef_finish_f_sig     <= cCoef_finish_f_sig;
    coef_data_ready_sig   <= ccoef_data_ready_sig;
    wait_cnt_max          <= cwait_cnt_max;
    Input_buf4_sig        <= cInput_buf4_sig;
    Input_buf4_out        <= cInput_buf4_sig;
    Input_buf5_sig        <= cInput_buf5_sig;
    Input_buf5_out        <= cInput_buf5_sig;
    -- State register
    current_state1        <= next_state1;
END IF;
END PROCESS Register_process1;
-----
Combinational_process1:
-----
PROCESS(current_state1,Mat_adr_sig,MAC_coef_clr_sig,Input_buf4_sig,Input_buf5_sig,Buf_adr_sig,
Buf_dat_sig,Buf_wren_sig,pulse_352_in,Buf_out_sig,sample_in,Polcoef_adr_sig,Polcoef_wren_sig,
MAC_coef_result_sig,MAC_coef_datab_sig,state3_cnt_sig,frst_coef_f_sig,state4_f_sig,
Polcoef_unused_sig,Data_temp,Coef_finish_f_sig,wait_cnt_max,coef_data_ready_sig) IS
BEGIN
    CASE (current_state1) IS
        -----
        -- Reset state
        -----
        When state_0 =>
            cMat_adr_sig            <= "1001001"; -- Default interpolation matrix address
            cMAC_coef_clr_sig       <= '1';      -- MAC arithmetic disabled
            cMAC_coef_datab_sig     <= "0000000000000000"; -- Reset
            cBuf_adr_sig            <= "0000";
            -- Reset Input Buffer address to output zero (0h)
            cBuf_dat_sig            <= "0000000000000000";
            -- Reset Input Buffer data input
            cBuf_wren_sig           <= '0';      -- Input Buffer in read mode
            cstate3_cnt_sig         <= "00000";
            cstate4_f_sig           <= '0';
            cfrst_coef_f_sig        <= '0';
            cData_temp              <= Data_temp;
            cCoef_finish_f_sig      <= '0';
            ccoef_data_ready_sig    <= '0';
            cwait_cnt_max           <= "101";
            cInput_buf4_sig         <= Input_buf4_sig;
            cInput_buf5_sig         <= Input_buf5_sig;
    
```

```

IF (pulse_352_in = '1') THEN          -- Has new data arrived
    next_state1 <= state_1;          -- Go store data
ELSE
    next_state1 <= state_0;          -- Stay in reset and wait for data
END IF;
-----
-- Store Input data
-----
When state_1 =>

    cMAC_coef_clr_sig    <= '0';      -- Set MAC to start arithmetic
    cMAC_coef_datab_sig  <= Buf_out_sig; -- Connect input buffer to MAC
    cstate3_cnt_sig      <= state3_cnt_sig;
    cstate4_f_sig        <= state4_f_sig;
    cfrst_coef_f_sig     <= '0';
    cData_temp           <= Data_temp;
    cCoef_finish_f_sig   <= Coef_finish_f_sig;
    cwait_cnt_max        <= wait_cnt_max;
    ccoef_data_ready_sig <= coef_data_ready_sig;
    -- Store input in 1H address space of the input buffer
    cBuf_dat_sig         <= sample_in;  -- New sample applied to input buffer
    cBuf_wren_sig        <= '1';       -- Write new sample to buffer
    cMat_adr_sig         <= "00000000"; -- Zero Coefficient address
    cBuf_adr_sig         <= "0001";    -- Reset buffer address (1h) to 1st input sample
    cInput_buf4_sig      <= Input_buf4_sig;
    cInput_buf5_sig      <= Input_buf5_sig;
    next_state1          <= state_3;    -- Wait to ensure data has been stored
-----
-- Matrix multiplication
-----
When state_2 =>

    cstate3_cnt_sig <= state3_cnt_sig;
    cfrst_coef_f_sig <= frst_coef_f_sig;
    cstate4_f_sig <= state4_f_sig;
    cwait_cnt_max <= wait_cnt_max;
    -- Complete storing coefficient value in two-port RAM
    IF (coef_data_ready_sig = '1') THEN -- coef_data_ready_sig set in state 3 to
        store here
        cMAC_coef_clr_sig <= '1'; -- stop arithmetic
        cCoef_finish_f_sig <= '1';
        cData_temp <= MAC_coef_result_sig; -- Coefficient in
            temporary register
    ELSE
        ccoef_data_ready_sig <= '0'; -- Reset signal
    END IF;

    cMAC_coef_clr_sig <= '0';
    cCoef_finish_f_sig <= '0';
    cData_temp <= Data_temp;
    ccoef_data_ready_sig <= coef_data_ready_sig;

END IF;

IF (to_integer(Mat_adr_sig) < Mat_cnt_max) THEN

    IF (to_integer(Mat_adr_sig) >= (Mat_cnt_max - Buf_cnt_max)) THEN
        cMat_adr_sig <= Mat_adr_sig + 1; -- Next row of
            matrix coefficients
        cBuf_adr_sig <= "0010"; -- Set input buffer to first
            address
        cBuf_dat_sig <= "0000000000000000";
        cBuf_wren_sig <= '0';
        cMAC_coef_datab_sig <= Buf_out_sig;

```

```

cInput_buf4_sig      <= Input_buf4_sig;
cInput_buf5_sig      <= Input_buf5_sig;
next_state1          <= state_4;
ELSE
  cMat_adr_sig        <= Mat_adr_sig +1; -- Next row of
                                     matrix coefficients
  cMAC_coef_datab_sig <= Buf_out_sig;
  cBuf_dat_sig        <= "0000000000000000";
  cBuf_wren_sig       <= '0';
  IF (to_integer(Buf_adr_sig) < buf_cnt_max) THEN
    -- Write first interval polynomial amplitude to register
    IF (to_integer(Buf_adr_sig) = 6) THEN
      cInput_buf4_sig <= Buf_out_sig;
    ELSE
      cInput_buf4_sig <= Input_buf4_sig;
    END IF;

    -- Write last interval polynomial amplitude to register
    IF (to_integer(Buf_adr_sig) = 7) THEN
      cInput_buf5_sig <= Buf_out_sig;
    ELSE
      cInput_buf5_sig <= Input_buf5_sig;
    END IF;
    -- Increment input buffer address
    cBuf_adr_sig <= Buf_adr_sig + 1;
    next_state1 <= state_2;
  ELSE
    cInput_buf4_sig <= Input_buf4_sig;
    cInput_buf5_sig <= Input_buf5_sig;
    cBuf_adr_sig <= "0000";
    next_state1 <= state_3;
  END IF;
END IF;
ELSE
  cMAC_coef_datab_sig <= "0000000000000000";
  cMAC_coef_clr_sig   <= '1';
  cMat_adr_sig        <= "1001001";
  cBuf_adr_sig        <= "0000";
  cBuf_dat_sig        <= "0000000000000000";
  cBuf_wren_sig       <= '0';
  cfirst_coef_f_sig   <= '0';
  cInput_buf4_sig     <= Input_buf4_sig;
  cInput_buf5_sig     <= Input_buf5_sig;
  next_state1         <= state_0;
END IF;

```

-----  
 -- Latch polynomial coefficient to output buffer  
 -----

When state\_3 =>

```

cstate4_f_sig        <= state4_f_sig;
cwait_cnt_max        <= wait_cnt_max;
cInput_buf4_sig      <= Input_buf4_sig;
cInput_buf5_sig      <= Input_buf5_sig;
IF (first_coef_f_sig = '0') THEN
  -- First polynomial coefficient value calculation and storage
  cMAC_coef_datab_sig <= Buf_out_sig;
  cBuf_dat_sig        <= "0000000000000000";
  -- Zero input buffer
  cBuf_wren_sig       <= '0';
  -- Buffer in read mode
  ccoef_data_ready_sig <= coef_data_ready_sig;

```

```

IF (to_integer(state3_cnt_sig) < 7) THEN
  cData_temp      <= Data_temp;
  cMAC_coef_clr_sig <= '0';
  cCoef_finish_f_sig <= Coef_finish_f_sig;
  IF (to_integer(state3_cnt_sig) = 0) THEN
    -- Calculate first coefficient
    cBuf_adr_sig <= "0001";
    cMat_adr_sig <= Mat_adr_sig;
  ELSE
    -- Time delay for MAC to finish coefficient calculation
    cBuf_adr_sig <= "0000";
    cMat_adr_sig <= "1001001";
  END IF;
  cstate3_cnt_sig <= state3_cnt_sig+1;
  cfrst_coef_f_sig <= '0';
  next_state1 <= state_3;
ELSE-- Setup variable for next arithmetic calculations
  cMAC_coef_clr_sig <= '1';
  cBuf_adr_sig <= "0001"; -- Set to 2nd input buffer data
                          address
  cMat_adr_sig <= "0000000";-- Set to start address of
                          next row multiplication
  cstate3_cnt_sig <= "00000"; -- Reset state3 counter since
                          leaving the state
  cfrst_coef_f_sig <= '1'; -- Next entrance to state3 uses
                          different path
  cCoef_finish_f_sig <= '1';
  cData_temp <= MAC_coef_result_sig; -- Write
                          first coefficient result
                          to the output
  next_state1 <= state_2;
END IF;
ELSE
  cMAC_coef_datab_sig <= Buf_out_sig;
  cBuf_dat_sig <= "0000000000000000";
  cBuf_wren_sig <= '0';
  cfrst_coef_f_sig <= frst_coef_f_sig;
  cCoef_finish_f_sig <= Coef_finish_f_sig;
  cMat_adr_sig <= Mat_adr_sig;
  cMAC_coef_clr_sig <= '0';
  cData_temp <= Data_temp;
  IF (state3_cnt_sig < wait_cnt_max) THEN
    cstate3_cnt_sig <= state3_cnt_sig + 1;
    ccoef_data_ready_sig <= coef_data_ready_sig;
    IF (to_integer(state3_cnt_sig) = (wait_cnt_max-1)) THEN
      cBuf_adr_sig <= "0000";
    ELSE
      cBuf_adr_sig <= "0000";
    END IF;
    next_state1 <= state_3;
  ELSE
    cstate3_cnt_sig <= "00000";
    cBuf_adr_sig <= "0001";
    ccoef_data_ready_sig <= '1';
    next_state1 <= state_2;
  END IF;
END IF;
-----
-- Buffer Shift
-----
When state_4 =>

```

```

cMAC_coef_clr_sig <= '0';
cstate3_cnt_sig   <= state3_cnt_sig;
cfrst_coef_f_sig <= frst_coef_f_sig;
cData_temp        <= Data_temp;
ccoef_data_ready_sig <= coef_data_ready_sig;
cInput_buf4_sig   <= Input_buf4_sig;
cInput_buf5_sig   <= Input_buf5_sig;
IF (to_integer(Buf_adr_sig) < (Buf_cnt_max+2)) THEN
    cwait_cnt_max <= wait_cnt_max;

    IF (state4_f_sig='0') THEN
        cstate4_f_sig <= '1';
        cMat_adr_sig <= Mat_adr_sig;
        cBuf_adr_sig <= Buf_adr_sig;
        cBuf_wren_sig <= '1';
        cBuf_dat_sig <= Buf_out_sig;
        cMAC_coef_datab_sig <= Buf_out_sig;
        cCoef_finish_f_sig <= '0';
        next_state1 <= state_4;
    ELSE
        cstate4_f_sig <= '0';
        cMat_adr_sig <= Mat_adr_sig + 1;
        cBuf_adr_sig <= Buf_adr_sig + 1;
        cBuf_wren_sig <= '0';
        cBuf_dat_sig <= "0000000000000000";
        cMAC_coef_datab_sig <= "0000000000000000";
        cCoef_finish_f_sig <= Coef_finish_f_sig;
        next_state1 <= state_4;
    END IF;
ELSE
    cstate4_f_sig <= '1';
    cMat_adr_sig <= "1001001";
    cBuf_adr_sig <= "0000";
    cBuf_wren_sig <= '0';
    cBuf_dat_sig <= "0000000000000000";
    cMAC_coef_datab_sig <= "0000000000000000";
    cwait_cnt_max <= "001";
    cCoef_finish_f_sig <= Coef_finish_f_sig;
    next_state1 <= state_3;
END IF;
END CASE;
END PROCESS Combinational_process1;

```

-----  
Register\_process2:  
-----

```

PROCESS (clk_60Meg_in,reset_in) IS
BEGIN
    IF (reset_in = '1') THEN
        -- Polynomial coefficient initial assignments
        Polcoef_adr_sig <= "00000";- Reset polcoef output buf address to output zero
        Polcoef_wren_sig <= '0'; -- Set to read mode
        Polcoef_unused_sig <= "00000000000000000000000000000000";
        Polcoef_dat_sig <= "00000000000000000000000000000000";
        coef_sync_out_sig <= '0'; -- State initial assignment
        current_state2 <= state_0; -- Force state machine to reset state

    ELSIF rising_edge(clk_60Meg_in) THEN
        -- Update latches (memory units)
        -- Polynomial coefficient registers update
        Polcoef_adr_sig <= cPolcoef_adr_sig;
        Polcoef_wren_sig <= cPolcoef_wren_sig;
    END IF;
END PROCESS;

```

```

        Polcoef_unused_sig <= cPolcoef_unused_sig;
        Polcoef_dat_sig <= cPolcoef_dat_sig;
        coef_sync_out_sig <= ccoef_sync_out_sig;
        Coef_sync_out <= ccoef_sync_out_sig;
        -- State register
        current_state2 <= next_state2;
    END IF;
END PROCESS Register_process2;
-----
Combinational_process2:
-----
PROCESS(current_state2,Polcoef_adr_sig,Polcoef_wren_sig,Polcoef_unused_sig,Polcoef_dat_sig,
Coef_finish_f_sig,coef_sync_out_sig,Data_temp) IS
BEGIN
    CASE (current_state2) IS
        -----
        -- Start polynomial coefficient store in buffer
        -----
        When state_0 =>

            IF (Coef_finish_f_sig='1') THEN
                cPolcoef_adr_sig <= Polcoef_adr_sig;
                cPolcoef_unused_sig <= Polcoef_unused_sig;
                cPolcoef_wren_sig <= '1';
                cPolcoef_dat_sig <= Data_temp(45 DOWNTO 16);-- top 25 bits of
                                                                result in
                                                                output buffer
                ccoef_sync_out_sig <= coef_sync_out_sig;
                next_state2 <= state_1;
            ELSE
                cPolcoef_adr_sig <= Polcoef_adr_sig;
                cPolcoef_unused_sig <= Polcoef_unused_sig;
                cPolcoef_wren_sig <= Polcoef_wren_sig;
                cPolcoef_dat_sig <= Polcoef_dat_sig;
                ccoef_sync_out_sig <= coef_sync_out_sig;
                next_state2 <= state_0;
            END IF;
        -----
        -- Finish polynomial coefficient store in buffer
        -----
        When state_1 =>

            cPolcoef_wren_sig <= '0';
            cPolcoef_unused_sig <= Polcoef_unused_sig;
            cPolcoef_dat_sig <= Polcoef_dat_sig;
            IF (to_integer(Polcoef_adr_sig) > 8) THEN
                ccoef_sync_out_sig <= '1';
                IF (to_integer(Polcoef_adr_sig) = 17) THEN
                    cPolcoef_adr_sig <= "00000";
                ELSE
                    cPolcoef_adr_sig <= Polcoef_adr_sig + 1;
                END IF;
            ELSE
                ccoef_sync_out_sig <= coef_sync_out_sig;
                cPolcoef_adr_sig <= Polcoef_adr_sig + 1;
            END IF;
            next_state2 <= state_0;
    END CASE;
END PROCESS Combinational_process2;
END ARCHITECTURE coef_calc;

```

### F3. Binary search

```

-----
-- Design unit   : Binary search
-- File name    : binary_search.vhd
-- Description   : Determines the interval where a crosspoint between a polynomial and saw-tooth
--               : wave exists to a resolution of 9 bits. The polynomial amplitude is calculated
--               : at a specific index determined by the binary search algorithm, when the calculation
--               : is complete the amplitude is compared to the trailing edge sawtooth amplitude.
--               : From this comparison the index value for the next polynomial amplitude calculation is
--               : is determined until a 9-bit accurate crosspoint has been calculated. When this occurs
--               : the two interval amplitudes wherein the crosspoint exists is latched for the next crosspoint
--               : calculation process
-- System       : VHDL'93
-- Author      : Deon Jacobs
--             : Department of Electrical Engineering
--             : University of Stellenbosch
--             : Deonj@sun.ac.za
-- Revision    : Version 1.2 25/08/2005
-----

```

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

```
ENTITY binary_search IS
```

```
PORT (
    clk_60Meg_in      : IN STD_LOGIC;
    pulse_352_in     : IN STD_LOGIC;
    reset_in         : IN STD_LOGIC;
    Pol_coef_sync_in : IN STD_LOGIC;
    Pol_coef_in      : IN STD_LOGIC_VECTOR(29 DOWNTO 0);
    Start_int_in     : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    End_int_in       : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    Pol_coef_wren_out : OUT STD_LOGIC;
    Pol_coef_adr_out  : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
    Index_out        : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
    Saw_out          : OUT STD_LOGIC_VECTOR(24 DOWNTO 0);
    Cross_pnt_sync_out : OUT STD_LOGIC;
    Latch_pol2_out   : OUT STD_LOGIC_VECTOR(24 DOWNTO 0);
    Latch_pol1_out   : OUT STD_LOGIC_VECTOR(24 DOWNTO 0)
);
```

```
END ENTITY binary_search;
```

```
ARCHITECTURE interval_calc OF binary_search IS
```

```
    COMPONENT Int_matrix
    PORT
    (
        address : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
        clock   : IN STD_LOGIC;
        q       : OUT STD_LOGIC_VECTOR (29 DOWNTO 0)
    );
```

```
END COMPONENT;
```

```
    COMPONENT saw_tooth
    PORT
    (
        address : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
        clock   : IN STD_LOGIC;
        q       : OUT STD_LOGIC_VECTOR (24 DOWNTO 0)
    );
```

```
END COMPONENT;
```

```

COMPONENT Pol_MAC
  PORT
  (
    dataa      : IN STD_LOGIC_VECTOR (29 DOWNT0 0);
    datab     : IN STD_LOGIC_VECTOR (29 DOWNT0 0);
    clock0     : IN STD_LOGIC := '1';
    aclr0      : IN STD_LOGIC := '0';
    result     : OUT STD_LOGIC_VECTOR (59 DOWNT0 0)
  );
END COMPONENT;

```

```

-----
-- State machine declarations
-----

```

```

--Matrix multiplication
  TYPE State_machine IS (state_0,state_1,state_2,state_3,state_4,state_5);
  SIGNAL current_state, next_state : State_machine;

```

```

-----
-- Combinational logic signals
-----

```

```

-- Interpolation System Matrix
  SIGNAL cInt_mat_adr_sig      : unsigned(13 DOWNT0 0);

```

```

-- Polynomial Multiply Accumulate
  SIGNAL cPol_MAC_datab_sig    : STD_LOGIC_VECTOR(29 DOWNT0 0);
  SIGNAL cPol_MAC_dataa_sig    : STD_LOGIC_VECTOR(29 DOWNT0 0);
  SIGNAL cPol_MAC_clr_sig      : STD_LOGIC;

```

```

-- Polinomial coefficient inputs
  SIGNAL cPol_coef_adr_sig     : unsigned(4 DOWNT0 0);

```

```

-- Saw-Tooth waveform
  SIGNAL cSaw_adr_sig          : unsigned(9 DOWNT0 0);
  SIGNAL cSaw_tooth_sig        : STD_LOGIC_VECTOR(24 DOWNT0 0);

```

```

-- Misc signals
  SIGNAL cIndex_sig            : unsigned(9 DOWNT0 0);
  SIGNAL cIndex_out_sig        : unsigned(9 DOWNT0 0);
  SIGNAL cLow_sig               : unsigned(9 DOWNT0 0);
  SIGNAL cHigh_sig              : unsigned(9 DOWNT0 0);
  SIGNAL cMult_adr_sig          : unsigned(3 DOWNT0 0);
  SIGNAL cState0_f_sig          : STD_LOGIC;-- Flag determining polynomial coefficient address
  SIGNAL cState2_f_sig          : STD_LOGIC;
  SIGNAL cState3_cnt_sig        : unsigned(3 DOWNT0 0);
  SIGNAL cState6_f_sig          : STD_LOGIC;
  SIGNAL cPol_amp_temp_sig      : STD_LOGIC_VECTOR(59 DOWNT0 0);
  SIGNAL cHigh_data_sig         : STD_LOGIC_VECTOR(24 DOWNT0 0);
  SIGNAL cLow_data_sig          : STD_LOGIC_VECTOR(24 DOWNT0 0);
  SIGNAL cComp_f_sig            : STD_LOGIC;
  SIGNAL cLatch_pol1_sig        : STD_LOGIC_VECTOR(24 DOWNT0 0);
  SIGNAL cLatch_pol2_sig        : STD_LOGIC_VECTOR(24 DOWNT0 0);
  SIGNAL cStart_int_sig         : STD_LOGIC_VECTOR(15 DOWNT0 0);
  SIGNAL cEnd_int_sig           : STD_LOGIC_VECTOR(15 DOWNT0 0);
  SIGNAL cCrosspnt_start_f_sig  : STD_LOGIC;
  SIGNAL cCross_pnt_sync_sig    : STD_LOGIC;

```

```

-----
-- Latched output (memory) signals
-----

```

```

-- Interpolation System Matrix
  SIGNAL Int_mat_adr_sig       : unsigned(13 DOWNT0 0);

```



```

        SIGNAL Int_mat_out_sig          : STD_LOGIC_VECTOR(29 DOWNT0 0);
-- Polynomial Multiply Accumulate
        SIGNAL Pol_MAC_dataaa_sig      : STD_LOGIC_VECTOR(29 DOWNT0 0);
        SIGNAL Pol_MAC_datab_sig      : STD_LOGIC_VECTOR(29 DOWNT0 0);
        SIGNAL Pol_MAC_clr_sig         : STD_LOGIC;
        SIGNAL Pol_MAC_result_sig      : STD_LOGIC_VECTOR(59 DOWNT0 0);

-- Saw-Tooth waveform
        SIGNAL Saw_adr_sig             : unsigned(9 DOWNT0 0);
        SIGNAL Saw_out_sig            : STD_LOGIC_VECTOR(24 DOWNT0 0);
        SIGNAL Saw_tooth_sig          : STD_LOGIC_VECTOR(24 DOWNT0 0);

-- Polinomial coefficient inputs
        SIGNAL Pol_coef_adr_sig        : unsigned(4 DOWNT0 0);

-- Misc signals
        SIGNAL Index_sig               : unsigned(9 DOWNT0 0);
        SIGNAL Index_out_sig           : unsigned(9 DOWNT0 0);
        SIGNAL Low_sig                 : unsigned(9 DOWNT0 0);
        SIGNAL High_sig                : unsigned(9 DOWNT0 0);
        SIGNAL Mult_adr_sig            : unsigned(3 DOWNT0 0);
        SIGNAL State0_f_sig            : STD_LOGIC; -- Flag determining polynomial coefficient address
        SIGNAL State2_f_sig            : STD_LOGIC;
        SIGNAL State3_cnt_sig          : unsigned(3 DOWNT0 0);
        SIGNAL State6_f_sig            : STD_LOGIC;
        SIGNAL Pol_amp_temp_sig        : STD_LOGIC_VECTOR(59 DOWNT0 0);
        SIGNAL High_data_sig           : STD_LOGIC_VECTOR(24 DOWNT0 0);
        SIGNAL Low_data_sig            : STD_LOGIC_VECTOR(24 DOWNT0 0);
        SIGNAL Comp_f_sig              : STD_LOGIC;
        SIGNAL Latch_pol1_sig          : STD_LOGIC_VECTOR(24 DOWNT0 0);
        SIGNAL Latch_pol2_sig          : STD_LOGIC_VECTOR(24 DOWNT0 0);
        SIGNAL Start_int_sig           : STD_LOGIC_VECTOR(15 DOWNT0 0);
        SIGNAL End_int_sig             : STD_LOGIC_VECTOR(15 DOWNT0 0);
        SIGNAL Crosspnt_start_f_sig    : STD_LOGIC;
        SIGNAL Cross_pnt_sync_sig      : STD_LOGIC;

-- Constant
        CONSTANT eight_cnst           : unsigned(3 DOWNT0 0) := "1000"; -- Constant value 8
        CONSTANT first_mat_coef       : STD_LOGIC_VECTOR(29 DOWNT0 0)
                                        := "0000000000000000000000001010101111"; -- Constant value 687

BEGIN

Int_matrix_inst : Int_matrix PORT MAP (
    address      => std_logic_vector(unsigned(Int_mat_adr_sig(11 DOWNT0 0))),
    clock        => clk_60Meg_in,
    q            => Int_mat_out_sig
);

saw_tooth_inst : saw_tooth PORT MAP (
    address      => std_logic_vector(unsigned(Saw_adr_sig(9 DOWNT0 0))),
    clock        => clk_60Meg_in,
    q            => Saw_out_sig
);

Pol_MAC_inst : Pol_MAC PORT MAP (
    dataaa       => Pol_MAC_dataaa_sig,
    datab       => Pol_MAC_datab_sig,
    clock0       => clk_60Meg_in,
    aclr0        => Pol_MAC_clr_sig,
    result       => Pol_MAC_result_sig
);

```

-----  
 Register\_process:  
 -----

```

PROCESS (clk_60Meg_in,reset_in) IS
BEGIN
  IF (reset_in = '1') THEN
    --Initialize relevant signals

    -- Interpolation System Matrix
    Int_mat_adr_sig      <= "0000000000000000";

    --Polynomial Multiply Accumulate
    Pol_MAC_dataa_sig   <= "00000000000000000000000000000000";
    Pol_MAC_datab_sig   <= "00000000000000000000000000000000";
    Pol_MAC_clr_sig     <= '1';

    --Polynomial Coefficient input
    Pol_coef_wren_out   <= '0';
    Pol_coef_adr_sig    <= "00000";

    -- Saw-Tooth waveform
    Saw_adr_sig         <= "0000000000";
    Saw_tooth_sig       <= "00000000000000000000000000000000";

    -- Misc signals
    Index_sig           <= "0100000000";
    Index_out_sig       <= "0000000000";
    High_sig            <= "1000000000";
    Low_sig             <= "0000000000";
    Mult_adr_sig        <= "0000";
    State0_f_sig        <= '0';
    State2_f_sig        <= '0';
    State3_cnt_sig      <= "0000";
    State6_f_sig        <= '0';
    Pol_amp_temp_sig    <=
"0000000000000000000000000000000000000000000000000000000000000000";
    High_data_sig      <= "00000000000000000000000000000000";
    Low_data_sig       <= "00000000000000000000000000000000";
    Comp_f_sig         <= '0';
    Latch_pol1_sig     <= "00000000000000000000000000000000";
    Latch_pol2_sig     <= "00000000000000000000000000000000";
    Start_int_sig      <= "0000000000000000";
    End_int_sig        <= "0000000000000000";
    Crosspnt_start_f_sig <= '0';
    Cross_pnt_sync_sig <= '0';

    -- Initial state assignment
    current_state      <= state_0;

  ELSIF rising_edge(clk_60Meg_in) THEN

    -- Interpolation System Matrix
    Int_mat_adr_sig    <= cInt_mat_adr_sig;

    -- Polynomial Multiply Accumulate
    Pol_MAC_dataa_sig <= cPol_MAC_dataa_sig;
    Pol_MAC_datab_sig <= cPol_MAC_datab_sig;
    Pol_MAC_clr_sig   <= cPol_MAC_clr_sig;
    Pol_coef_wren_out <= '0';           -- Always in read mode
    Pol_coef_adr_sig  <= cPol_coef_adr_sig;
    Pol_coef_adr_out<= std_logic_vector(unsigned(cPol_coef_adr_sig(4 DOWNT0 0)));
  
```

```

-- Saw-Tooth waveform
Saw_adr_sig      <= cSaw_adr_sig      ;-- Address signal for saw_tooth lookup
                                     table
Saw_tooth_sig    <= cSaw_tooth_sig;  -- Signal holding Saw_tooth current ouput
                                     value
Saw_out          <= cSaw_tooth_sig;

-- Misc signals
Index_sig        <= cIndex_sig;
Index_out_sig    <= cIndex_out_sig;
Index_out        <= std_logic_vector(unsigned(cIndex_out_sig(9 DOWNT0 0)));
High_sig         <= cHigh_sig;
Low_sig         <= cLow_sig;
Mult_adr_sig     <= cMult_adr_sig;
State0_f_sig     <= cState0_f_sig;
State2_f_sig     <= cState2_f_sig;
State3_cnt_sig  <= cState3_cnt_sig;
State6_f_sig     <= cState6_f_sig;
Pol_amp_temp_sig <= cPol_amp_temp_sig;
High_data_sig   <= cHigh_data_sig;
Low_data_sig    <= cLow_data_sig;
Comp_f_sig      <= cComp_f_sig;
Latch_pol1_sig  <= cLatch_pol1_sig;
Latch_pol2_sig  <= cLatch_pol2_sig;
Latch_pol1_out  <= cLatch_pol1_sig;
Latch_pol2_out  <= cLatch_pol2_sig;
Start_int_sig   <= cStart_int_sig;
End_int_sig     <= cEnd_int_sig;
Crosspnt_start_f_sig <= cCrosspnt_start_f_sig;
Cross_pnt_sync_sig <= cCross_pnt_sync_sig;
Cross_pnt_sync_out <= cCross_pnt_sync_sig;

-- Next state assignment
current_state    <= next_state;

END IF;
END PROCESS Register_process;

```

-----  
Combinational\_process:  
-----

```

PROCESS(current_state,Crosspnt_start_f_sig,Cross_pnt_sync_sig,pulse_352_in,Start_int_sig,End_int_sig,
End_int_in,Start_int_in,High_data_sig,Low_data_sig,Latch_pol1_sig,Latch_pol2_sig,Pol_coef_sync_in,Mult_adr_sig,
Pol_coef_in,Index_sig,Index_out_sig,High_sig,Low_sig,Int_mat_adr_sig,Pol_MAC_dataa_sig,Pol_MAC_datab_sig,
Pol_MAC_result_sig,Int_mat_out_sig,Saw_adr_sig,Pol_coef_adr_sig,State2_f_sig,State3_cnt_sig,Pol_amp_temp_sig,
Comp_f_sig,State6_f_sig,State0_f_sig,Saw_out_sig,Saw_tooth_sig) IS

```

```

-- Procedure updating the necessary signals for next saw_tooth wave comparison
PROCEDURE Binary_search_signal_update IS

```

```

-- Internal process variables

```

```

VARIABLE Index_var      : unsigned(9 DOWNT0 0);
VARIABLE High_var       : unsigned(9 DOWNT0 0);
VARIABLE Low_var        : unsigned(9 DOWNT0 0);
VARIABLE High_data_var  : std_logic_vector(24 DOWNT0 0);
VARIABLE Low_data_var   : std_logic_vector(24 DOWNT0 0);

```

```

BEGIN

```

```

-- Update signals and variables according to Saw-tooth comparison

```

```

IF (Comp_f_sig = '1') THEN -- Polynomial larger than saw-tooth
  cLow_sig <= Index_sig; -- Low value = index value
  Low_var := Index_sig; -- Test if the low data value is equal to zero
  IF (to_integer(Index_sig) = 0) THEN
    cLow_data_sig <= Start_int_sig(15)&"0"&
      Start_int_in(14 DOWNT0 0)&"00000000";
    -- Start interval amplitude used for low_data

```

```

        Low_data_var := Start_int_sig(15)&"0"&
                        Start_int_in(14 DOWNT0 0)&"00000000";

ELSE-- Normal data assignment
    cLow_data_sig <= cPol_amp_temp_sig(59)&
                    cPol_amp_temp_sig(32 DOWNT0 9);
                    -- Low data = polynomial amplitude
    Low_data_var := cPol_amp_temp_sig(59)&
                    cPol_amp_temp_sig(32 DOWNT0 9);

END IF;
cHigh_sig <= High_sig;
High_var := High_sig;
cHigh_data_sig <= High_data_sig;

-- High data stays the same
High_data_var := High_data_sig;

-- Current saw-tooth value output
cSaw_tooth_sig <= Saw_out_sig; -- Output saw-tooth is current saw-tooth
                                amplitude

-- Current index output value
cIndex_out_sig <= Index_sig;
ELSE
                                -- Polynomial smaller than saw-tooth
                                -- High value = index value
                                -- Test if the high data value is equal to 512
    cHigh_sig <= Index_sig;
    High_var := Index_sig;
    IF (to_integer(Index_sig) = 512) THEN
        cHigh_data_sig <= End_int_sig(15)&"0"&
                        End_int_sig(14 DOWNT0 0)&"00000000";
        High_data_var := End_int_sig(15)&"0"&
                        End_int_sig(14 DOWNT0 0)&"00000000";
    ELSE-- Normal data assignment
        cHigh_data_sig <= cPol_amp_temp_sig(59)&
                        cPol_amp_temp_sig(32 DOWNT0 9);
                        -- High data = polynomial amplitude
        High_data_var := cPol_amp_temp_sig(59)&
                        cPol_amp_temp_sig(32 DOWNT0 9);
    END IF;
    cLow_sig <= Low_sig;
    Low_var := Low_sig;
    cLow_data_sig <= Low_data_sig;

    -- Low data stays the same
    Low_data_var := Low_data_sig;
    -- Current saw-tooth value output
    cSaw_tooth_sig <= Saw_tooth_sig; -- Output saw-tooth amplitude stays the same
    -- Current index output value
    cIndex_out_sig <= Index_out_sig;
END IF;
-- Test to see if the pol coef address needs resetting

IF (to_integer(Pol_coef_adr_sig) = 9) THEN
    cPol_coef_adr_sig <= "00000";
ELSIF(to_integer(Pol_coef_adr_sig) = 18) THEN
    cPol_coef_adr_sig <= "01001";
ELSE
    cPol_coef_adr_sig <= Pol_coef_adr_sig;
END IF;

-- Test for binary search completion
IF (to_integer(High_var - Low_var) = 1) THEN

    cLatch_pol1_sig <= Low_data_var;
    cLatch_pol2_sig <= High_data_var;
    cIndex_sig <= Index_sig;

```

```

        next_state          <= state_0;
    ELSE
        cLatch_pol1_sig <= Latch_pol1_sig;
        cLatch_pol2_sig <= Latch_pol2_sig;
        cIndex_sig      <= Low_var + shift_right((High_var - Low_var),1); -- Update
                                                                    index
        next_state      <= state_5;
    END IF;

END PROCEDURE Binary_search_signal_update;

BEGIN
CASE (current_state) IS
-----
-- Reset state
-----
When state_0 =>

        cInt_mat_adr_sig      <= "00000000000000";

        cPol_MAC_dataa_sig    <= "00000000000000000000000000000000";
        cPol_MAC_datab_sig    <= "00000000000000000000000000000000";
        cPol_MAC_clr_sig      <= '1'; -- MAC disabled
        cSaw_adr_sig          <= "0000000000"; -- Reset value of Saw-tooth
                                                                    lookup table address
        cIndex_sig            <= "0100000000"; -- Initial index value of 256
        cIndex_out_sig        <= Index_out_sig;
        cHigh_sig             <= "1000000000"; -- Initial high value of 512
        cLow_sig              <= "0000000000"; -- Initial low value of 0
        cMult_adr_sig         <= "0000"; -- address signal for
                                                                    polynomial amplitude calculation
        cState2_f_sig         <= '0';
        cState3_cnt_sig       <= "0000";
        cState6_f_sig         <= State6_f_sig;
        cPol_amp_temp_sig     <= Pol_amp_temp_sig;
        cHigh_data_sig        <= High_data_sig;
        cLow_data_sig         <= Low_data_sig;
        cComp_f_sig           <= '0';
        cSaw_tooth_sig        <= Saw_tooth_sig;
        cLatch_pol1_sig       <= Latch_pol1_sig;
        cLatch_pol2_sig       <= Latch_pol2_sig;

    IF ((pulse_352_in = '1')and(Pol_coef_sync_in='1')) THEN

        IF (State0_f_sig = '0') THEN
            -- Use first half of polynomial coefficients in ping-pong buffer
            cState0_f_sig      <= '1';
            cPol_coef_adr_sig   <= "00000";
        ELSE
            -- Use second half of pol coef in ping-pong buffer
            cState0_f_sig      <= '0';
            cPol_coef_adr_sig   <= "01001";
        END IF;
        -- 2nd 352_8kHz pulse set Crosspoint sync signal to start Crosspoint
        derivation
        IF (Crosspnt_start_f_sig = '0') THEN
            cCrosspnt_start_f_sig <= '1';
            cCross_pnt_sync_sig <= '0';
        ELSE
            cCrosspnt_start_f_sig <= Crosspnt_start_f_sig;
            cCross_pnt_sync_sig <= '1';
        END IF;
        -- Assign input start and end interval amplitudes to internal signals
        cEnd_int_sig           <= End_int_in;
        cStart_int_sig        <= Start_int_in;
    
```

```

next_state      <= state_1;

ELSE
-- Stay in reset and wait for next 352kHz pulse
  cEnd_int_sig      <= End_int_sig;
  cStart_int_sig    <= Start_int_sig;
  cCrosspnt_start_f_sig <= Crosspnt_start_f_sig;
  cCross_pnt_sync_sig <= Cross_pnt_sync_sig;
  cPol_coef_adr_sig <= "00000";
  cState0_f_sig     <= State0_f_sig;
  next_state       <= state_0;
END IF;

-----
-- First entrance - New binary search
-----
When state_1 =>

  cIndex_sig      <= Index_sig;
  cIndex_out_sig  <= Index_out_sig;
  cHigh_sig       <= High_sig;
  cLow_sig        <= Low_sig;
  cSaw_adr_sig    <= Saw_adr_sig;
  cMult_adr_sig   <= Mult_adr_sig;
  cState0_f_sig   <= State0_f_sig;
  cState2_f_sig   <= State2_f_sig;
  cState3_cnt_sig <= State3_cnt_sig;
  cState6_f_sig   <= State6_f_sig;
  cInt_mat_adr_sig <= "0001111111000"; -- Middle index value
  cPol_MAC_dataa_sig <= Pol_MAC_dataa_sig;
  cPol_MAC_datab_sig <= Pol_MAC_datab_sig;
  cPol_MAC_clr_sig <= '1';
  -- MAC disabled
  cPol_coef_adr_sig <= Pol_coef_adr_sig + 1; -- Polynomial coefficient
                                                    address increase

  cPol_amp_temp_sig <= Pol_amp_temp_sig;
  cHigh_data_sig    <= High_data_sig;
  cLow_data_sig     <= Low_data_sig;
  cComp_f_sig       <= Comp_f_sig;
  cSaw_tooth_sig    <= Saw_tooth_sig;
  cLatch_pol1_sig   <= Latch_pol1_sig;
  cLatch_pol2_sig   <= Latch_pol2_sig;
  cEnd_int_sig      <= End_int_sig;
  cStart_int_sig    <= Start_int_sig;
  cCrosspnt_start_f_sig <= Crosspnt_start_f_sig;
  cCross_pnt_sync_sig <= Cross_pnt_sync_sig;
  next_state       <= state_2;

-----
-- Calculation of polynomial amplitude
-----
When state_2 =>

  cIndex_sig      <= Index_sig;
  cIndex_out_sig  <= Index_out_sig;
  cHigh_sig       <= High_sig;
  cLow_sig        <= Low_sig;
  cSaw_adr_sig    <= Index_sig; -- Get Saw-tooth amplitude at Index address
  cState0_f_sig   <= State0_f_sig;
  cState3_cnt_sig <= State3_cnt_sig;
  cState6_f_sig   <= State6_f_sig;
  cPol_amp_temp_sig <= Pol_amp_temp_sig;
  cHigh_data_sig  <= High_data_sig;
  cLow_data_sig   <= Low_data_sig;
  cComp_f_sig     <= Comp_f_sig;

```

```

cSaw_tooth_sig <= Saw_tooth_sig;
cLatch_pol1_sig <= Latch_pol1_sig;
cLatch_pol2_sig <= Latch_pol2_sig;
cEnd_int_sig    <= End_int_sig;
cStart_int_sig  <= Start_int_sig;
cCrosspnt_start_f_sig <= Crosspnt_start_f_sig;
cCross_pnt_sync_sig <= Cross_pnt_sync_sig;

IF (State2_f_sig = '0') THEN -- First entrance to state_2 from prior state
  cMult_adr_sig    <= Mult_adr_sig;
  cPol_MAC_dataa_sig <= "00000000000000000000000000000000";
  cPol_MAC_datab_sig <= "00000000000000000000000000000000";
  cPol_MAC_clr_sig <= '0';
  --MAC enabled
  cPol_coef_adr_sig <= Pol_coef_adr_sig+1;
  cInt_mat_adr_sig <= Int_mat_adr_sig+1;
  cState2_f_sig    <= '1';
  next_state       <= state_2;
ELSE
  IF (to_integer(Mult_adr_sig) < 7) THEN
    IF (to_integer(Mult_adr_sig) = 0) THEN -- First coefficient
      calculation
      cPol_MAC_dataa_sig <= first_mat_coef;
      cInt_mat_adr_sig   <= Int_mat_adr_sig + 1;
      cPol_coef_adr_sig <= Pol_coef_adr_sig + 1;

      ELSIF (to_integer(Mult_adr_sig) = 6) THEN
        cPol_MAC_dataa_sig <= Int_mat_out_sig;
        cInt_mat_adr_sig   <= Int_mat_adr_sig;
        cPol_coef_adr_sig <= Pol_coef_adr_sig;
      ELSE
        cPol_MAC_dataa_sig <= Int_mat_out_sig;
        cInt_mat_adr_sig   <= Int_mat_adr_sig + 1;
        cPol_coef_adr_sig <= Pol_coef_adr_sig + 1;
      END IF;
      cPol_MAC_datab_sig <= Pol_coef_in;
      cPol_MAC_clr_sig   <= '0';
      cMult_adr_sig     <= Mult_adr_sig + 1;
      cState2_f_sig     <= State2_f_sig;
      next_state        <= state_2;

    ELSE
      cInt_mat_adr_sig <= Int_mat_adr_sig;
      cPol_MAC_dataa_sig <= Int_mat_out_sig;
      cPol_MAC_datab_sig <= Pol_coef_in;
      cPol_MAC_clr_sig <= '0'; -- MAC still enabled
      cMult_adr_sig <= "0000"; -- Restore Multiply address
      counter
      cPol_coef_adr_sig <= Pol_coef_adr_sig;
      cState2_f_sig <= '0'; -- Reset flag: next entrance to this
      state occurs from call from another state
      next_state <= state_3;

    END IF;
  END IF;
END IF;

-----
-- Wait until polinomial amplitude has been calculated & Saw-Tooth comparison with
--Polynomial amplitude
-----
When state_3 =>

  cIndex_sig <= Index_sig;
  cIndex_out_sig <= Index_out_sig;
  cHigh_sig <= High_sig;
  cLow_sig <= Low_sig;

```

```

cSaw_adr_sig    <= Saw_adr_sig;
cMult_adr_sig   <= Mult_adr_sig;
cState0_f_sig   <= State0_f_sig;
cState2_f_sig   <= State2_f_sig;
cState6_f_sig   <= State6_f_sig;
cHigh_data_sig  <= High_data_sig;
cLow_data_sig   <= Low_data_sig;
cSaw_tooth_sig  <= Saw_tooth_sig;
cLatch_pol1_sig <= Latch_pol1_sig;
cLatch_pol2_sig <= Latch_pol2_sig;
cEnd_int_sig    <= End_int_sig;
cStart_int_sig  <= Start_int_sig;
cCrosspnt_start_f_sig <= Crosspnt_start_f_sig;
cCross_pnt_sync_sig <= Cross_pnt_sync_sig;

IF (to_integer(State3_cnt_sig) = 5) THEN -- Latch MAC result to
                                         Pol_amp_temp_sig
    cState3_cnt_sig <= State3_cnt_sig + 1;
    cPol_MAC_dataa_sig <= "00000000000000000000000000000000";
    cPol_MAC_datab_sig <= "00000000000000000000000000000000";
    cPol_MAC_clr_sig <= '0'; -- MAC enabled changed to disabled
    cInt_mat_adr_sig <= Int_mat_adr_sig;
    cPol_coef_adr_sig <= Pol_coef_adr_sig;
    cPol_amp_temp_sig <= Pol_MAC_result_sig; -- Pol temp assignment
    cComp_f_sig <= Comp_f_sig;
    next_state <= state_3;

ELSIF (to_integer(State3_cnt_sig) = 6) THEN-- Compare Calculated polynomial
                                         output to saw-tooth
    cState3_cnt_sig <= "0000";
    cPol_MAC_dataa_sig <= "00000000000000000000000000000000";
    cPol_MAC_datab_sig <= "00000000000000000000000000000000";
    cPol_MAC_clr_sig <= '1';
    cInt_mat_adr_sig <= Int_mat_adr_sig;
    cPol_coef_adr_sig <= Pol_coef_adr_sig+1;
    cPol_amp_temp_sig <= Pol_amp_temp_sig;

    IF (to_integer(signed(Pol_amp_temp_sig(59)&
        Pol_amp_temp_sig(32 DOWNT0 9))) >
        to_integer(signed(Saw_out_sig))) THEN

        cComp_f_sig <= '1';
    ELSE
        cComp_f_sig <= '0';
    END IF;
    next_state <= state_4; -- Go back to reset, later to
                            interval update

ELSIF (to_integer(State3_cnt_sig) = 0) THEN -- Last multiplication for
                                         polynomial amplitude result
    cState3_cnt_sig <= State3_cnt_sig + 1;
    cPol_MAC_dataa_sig <= Int_mat_out_sig;
    cPol_MAC_datab_sig <= Pol_coef_in;
    cPol_MAC_clr_sig <= '0';
    cPol_coef_adr_sig <= Pol_coef_adr_sig;
    cInt_mat_adr_sig <= Int_mat_adr_sig;
    cPol_amp_temp_sig <= Pol_amp_temp_sig;
    cComp_f_sig <= Comp_f_sig;
    next_state <= state_3;

ELSE -- Wait until polynomial amplitude calculation has completed
    cState3_cnt_sig <= State3_cnt_sig + 1;
    cPol_MAC_dataa_sig <= "00000000000000000000000000000000";
    cPol_MAC_datab_sig <= "00000000000000000000000000000000";
    cPol_MAC_clr_sig <= '0';
    cPol_coef_adr_sig <= Pol_coef_adr_sig;
    cInt_mat_adr_sig <= Int_mat_adr_sig;

```



```

        cPol_amp_temp_sig      <= Pol_amp_temp_sig;
        cComp_f_sig           <= Comp_f_sig;
        next_state            <= state_3;

    END IF;

-----
-- Assign new low & high signals according to comparison
-----
When state_4 =>

    cSaw_adr_sig             <= Saw_adr_sig;
    cMult_adr_sig            <= Mult_adr_sig;
    cState0_f_sig            <= State0_f_sig;
    cState2_f_sig            <= State2_f_sig;
    cState3_cnt_sig          <= State3_cnt_sig;
    cState6_f_sig            <= State6_f_sig;
    cInt_mat_adr_sig         <= Int_mat_adr_sig;
    cPol_MAC_dataa_sig       <= Pol_MAC_dataa_sig;

    cPol_MAC_datab_sig       <= Pol_MAC_datab_sig;
    cPol_MAC_clr_sig         <= '1';           -- MAC disabled
    cPol_amp_temp_sig        <= Pol_amp_temp_sig;
    cComp_f_sig              <= Comp_f_sig;
    cEnd_int_sig             <= End_int_sig;
    cStart_int_sig           <= Start_int_sig;
    cCrosspnt_start_f_sig    <= Crosspnt_start_f_sig;
    cCross_pnt_sync_sig      <= Cross_pnt_sync_sig;
    Binary_search_signal_update; -- Procedure handling assignment
                                of low & high signals

-----
-- Update Index Matrix Address
-----
When state_5 =>
    cHigh_sig                <= High_sig;
    cLow_sig                  <= Low_sig;
    cMult_adr_sig            <= Mult_adr_sig;
    cState0_f_sig            <= State0_f_sig;
    cState2_f_sig            <= State2_f_sig;
    cState3_cnt_sig          <= State3_cnt_sig;
    cPol_MAC_dataa_sig       <= Pol_MAC_dataa_sig;
    cPol_MAC_datab_sig       <= Pol_MAC_datab_sig;
    cPol_MAC_clr_sig         <= '1';           -- MAC disabled
    cPol_amp_temp_sig        <= Pol_amp_temp_sig;
    cComp_f_sig              <= Comp_f_sig;
    cSaw_adr_sig             <= Saw_adr_sig;
    cSaw_tooth_sig           <= Saw_tooth_sig;
    cHigh_data_sig           <= High_data_sig;
    cLow_data_sig            <= Low_data_sig;
    cLatch_pol1_sig          <= Latch_pol1_sig;
    cLatch_pol2_sig          <= Latch_pol2_sig;
    cState6_f_sig            <= '0';
    cIndex_sig                <= Index_sig;
    cIndex_out_sig           <= Index_out_sig;
    cPol_coef_adr_sig         <= Pol_coef_adr_sig+1;
    cInt_mat_adr_sig         <= Index_sig*eight_cnst-eight_cnst; -- Update matrix
                                                                address

    cEnd_int_sig             <= End_int_sig;
    cStart_int_sig           <= Start_int_sig;
    cCrosspnt_start_f_sig    <= Crosspnt_start_f_sig;
    cCross_pnt_sync_sig      <= Cross_pnt_sync_sig;
    next_state              <= state_2;

END CASE;
END PROCESS Combinational_process;
END ARCHITECTURE interval_calc;

```

## F4. Crosspoint calculation

```
-----
-- Design unit : Crosspoint derivation
-- File name   : crosspoint.vhd
-- Description  : Calculation of the interval crosspoint between a saw-tooth and polynomial signal
-- System      : VHDL'93
-- Author      : Deon Jacobs
--
--              : Department of Electrical Engineering
--              : University of Stellenbosch
--              : Deonj@sun.ac.za
-- Revision    : Version 1.0 26/09/2005
-----
```

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

```
ENTITY crosspoint IS
```

```
PORT (
    clk_60Meg_in       : IN STD_LOGIC;
    pulse_352_in      : IN STD_LOGIC;
    reset_in           : IN STD_LOGIC;
    sync_in            : IN STD_LOGIC;
    Index512_in        : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    Saw_tooth_in       : IN STD_LOGIC_VECTOR(24 DOWNTO 0);
    Pol_amp1_in        : IN STD_LOGIC_VECTOR(24 DOWNTO 0);
    Pol_amp2_in        : IN STD_LOGIC_VECTOR(24 DOWNTO 0);
    PWM_out            : OUT STD_LOGIC_VECTOR(23 DOWNTO 0)
);
```

```
END ENTITY crosspoint;
```

```
ARCHITECTURE derivation OF crosspoint IS
```

```
-- State declaration
```

```
TYPE State_machine IS (state_0,state_1,state_2,state_3,state_4);
SIGNAL current_state,next_state : State_machine;
```

```
-- Combinational process signals
```

```
SIGNAL cIndex512_sig : signed(9 DOWNTO 0);
SIGNAL cIndexNew_sig : signed(16 DOWNTO 0);
SIGNAL cLow_sig      : signed(16 DOWNTO 0);
SIGNAL cHigh_sig     : signed(16 DOWNTO 0);
SIGNAL cSaw_tooth_sig : signed(24 DOWNTO 0);
SIGNAL cSaw_amp_sig  : signed(24 DOWNTO 0);
SIGNAL cPol_amp_sig  : signed(41 DOWNTO 0);
SIGNAL cPol_amp1_sig : signed(24 DOWNTO 0);
SIGNAL cPol_amp2_sig : signed(24 DOWNTO 0);
SIGNAL cPol_amp_div_sig : signed(24 DOWNTO 0);
SIGNAL cGrad_f_sig   : STD_LOGIC;
SIGNAL cPWM_frac_sig : signed(16 DOWNTO 0);
SIGNAL cPWM_sig      : signed(23 DOWNTO 0);
```

```
-- Memory process signals
```

```
SIGNAL Index512_sig : signed(9 DOWNTO 0);-- Index value from previous binary search
SIGNAL IndexNew_sig : signed(16 DOWNTO 0);-- Index value for current binary search
SIGNAL Low_sig      : signed(16 DOWNTO 0);-- Low value for binary search procedure
SIGNAL High_sig     : signed(16 DOWNTO 0);-- High value for binary search procedure
SIGNAL Saw_tooth_sig : signed(24 DOWNTO 0);-- 1st amplitude coordinate of saw-tooth signal
SIGNAL Saw_amp_sig  : signed(24 DOWNTO 0);-- Calculated saw-tooth wave amplitude at
                                         index value
SIGNAL Pol_amp_sig  : signed(41 DOWNTO 0);-- Polynomial amplitude result through Linear
                                         interpolation
```

```

SIGNAL Pol_amp1_sig : signed(24 DOWNT0 0); -- 1st amplitude coordinate of polynomial signal
SIGNAL Pol_amp2_sig : signed(24 DOWNT0 0);-- 2nd amplitude coordinate of polynomial signal
SIGNAL Pol_amp_div_sig: signed(24 DOWNT0 0);-- Difference between 2nd & 1st polynomial
amplitudes
SIGNAL Grad_f_sig : STD_LOGIC; -- Indicates if polynomial gradient is positive or
negative
SIGNAL PWM_frac_sig : signed(16 DOWNT0 0); -- Fractional value of 24-bit PWM output
Signal
SIGNAL PWM_sig : signed(23 DOWNT0 0);-- Signal holding complete 24-bit PWM
output signal

```

```
BEGIN
```

```
-----
Register_process:
-----
```

```

PROCESS (clk_60Meg_in,reset_in) IS
BEGIN
  IF (reset_in = '1') THEN
    Index512_sig <= "0000000000";
    IndexNew_sig <= "0000000000000000";
    Low_sig <= "0000000000000000";
    High_sig <= "0000000000000000";
    Saw_tooth_sig <= "000000000000000000000000";
    Saw_amp_sig <= "000000000000000000000000";
    Pol_amp_sig <= "000000000000000000000000000000000000000000000000";
    Pol_amp1_sig <= "000000000000000000000000000000";
    Pol_amp2_sig <= "000000000000000000000000000000";
    Pol_amp_div_sig<= "000000000000000000000000000000";
    Grad_f_sig <= '0';
    PWM_frac_sig <= "0000000000000000";
    PWM_sig <= "000000000000000000000000000000";
    current_state <= state_0;

  ELSIF (rising_edge(clk_60Meg_in)) THEN

    -- Assignment of internal signals
    Index512_sig <= cIndex512_sig;
    IndexNew_sig <= cIndexNew_sig;
    Low_sig <= cLow_sig;
    High_sig <= cHigh_sig;
    Saw_tooth_sig <= cSaw_tooth_sig;
    Saw_amp_sig <= cSaw_amp_sig;
    Pol_amp_sig <= cPol_amp_sig;
    Pol_amp1_sig <= cPol_amp1_sig;
    Pol_amp2_sig <= cPol_amp2_sig;
    Pol_amp_div_sig<= cPol_amp_div_sig;
    Grad_f_sig <= cGrad_f_sig;
    PWM_frac_sig <= cPWM_frac_sig;
    PWM_sig <= cPWM_sig;

    -- Assignment of output signals
    PWM_out <= std_logic_vector(cPWM_sig(23 DOWNT0 0));
    current_state <= next_state;

  END IF;
END PROCESS Register_process;

```

```
-----
Combinational_process:
-----
```

```

PROCESS(current_state,pulse_352_in,sync_in,PWM_sig,PWM_frac_sig,Grad_f_sig,Index512_in,
Saw_tooth_in,Pol_amp1_in,Pol_amp2_in,Index512_sig,Saw_tooth_sig,Pol_amp1_sig,Pol_amp2_sig,
Pol_amp_div_sig,IndexNew_sig,High_sig,Low_sig,Pol_amp_sig,Saw_amp_sig) IS

```

```

PROCEDURE Binary_search IS

    VARIABLE Low_var   : signed(16 DOWNT0 0);
    VARIABLE High_var  : signed(16 DOWNT0 0);

BEGIN
    -- Comparison of polynomial amplitude with saw-tooth amplitude
    IF (to_integer(signed(Pol_amp_sig(41 DOWNT0 15))) >
        to_integer(signed(Saw_amp_sig))) THEN

        IF (Grad_f_sig = '1') THEN

            cLow_sig   <= IndexNew_sig;
            Low_var    := IndexNew_sig;
            cHigh_sig  <= High_sig;
            High_var   := High_sig;

        ELSE

            cLow_sig   <= Low_sig;
            Low_var    := Low_sig;
            cHigh_sig  <= IndexNew_sig;
            High_var   := IndexNew_sig;

        END IF;

    ELSE

        IF (Grad_f_sig = '1') THEN

            cLow_sig   <= Low_sig;
            Low_var    := Low_sig;
            cHigh_sig  <= IndexNew_sig;
            High_var   := IndexNew_sig;

        ELSE

            cLow_sig   <= IndexNew_sig;
            Low_var    := IndexNew_sig;
            cHigh_sig  <= High_sig;
            High_var   := High_sig;

        END IF;

    END IF;

    -- Determine if binary search has completed
    IF (to_integer(High_var - Low_var) = 1) THEN
        cIndexNew_sig <= IndexNew_sig;
        cPWM_frac_sig <= Low_var; -- PWM_fraction equal to lower index value
        next_state    <= state_4;
    ELSE
        -- has not completed update index value
        cIndexNew_sig <= Low_var + shift_right((High_var - Low_var),1); -- Update
                                                                    index
        cPWM_frac_sig <= PWM_frac_sig;
        next_state    <= state_2;
    END IF;

END PROCEDURE Binary_search;

BEGIN
    CASE (current_state) IS
        -----
        -- Reset state & Latch relevant input signals
        -----
        When state_0 =>

            IF ((pulse_352_in = '1')and(sync_in = '1')) THEN

                cIndex512_sig <= signed(Index512_in(9 DOWNT0 0)); -- Cast
                                                                    from std_logic_vector to signed
                cIndexNew_sig <= IndexNew_sig;
                cLow_sig      <= Low_sig;
                cHigh_sig     <= High_sig;
            
```

```

cSaw_tooth_sig <= signed(Saw_tooth_in(24 DOWNT0 0));

cSaw_amp_sig <= Saw_amp_sig;
cPol_amp_sig <= Pol_amp_sig;
cPol_amp1_sig <= signed(Pol_amp1_in(24 DOWNT0 0));

cPol_amp2_sig <= signed(Pol_amp2_in(24 DOWNT0 0));

cPol_amp_div_sig<= Pol_amp_div_sig;
cGrad_f_sig <= Grad_f_sig;
cPWM_frac_sig <= PWM_frac_sig;
cPWM_sig <= PWM_sig;
next_state <= state_1;
ELSE
cIndex512_sig <= "0000000000";
cIndexNew_sig <= "000000000000000000";
cLow_sig <= "000000000000000000";
cHigh_sig <= "000000000000000000";
cSaw_tooth_sig <= "000000000000000000000000000000";
cSaw_amp_sig <= "000000000000000000000000000000";
cPol_amp_sig <=
"00000000000000000000000000000000000000000000000000000";
cPol_amp1_sig <= "00000000000000000000000000000000";
cPol_amp2_sig <= "00000000000000000000000000000000";
cPol_amp_div_sig<= "0000000000000000000000000000000000";
cGrad_f_sig <= '0';
cPWM_frac_sig <= "000000000000000000";
cPWM_sig <= PWM_sig;
next_state <= state_0;
END IF;

```

```

-----
-- Linear interpolation gradient determination & Initial signal assignments for binary
-- search
-----

```

```

When state_1 =>

```

```

cIndex512_sig <= Index512_sig;
cIndexNew_sig <= "001000000000000000";-- Initial index assignment:
2^15/2=16384
cLow_sig <= "000000000000000000";-- Low assignment : 0
cHigh_sig <= "010000000000000000";-- High assignment: 32384
cSaw_tooth_sig <= Saw_tooth_sig;
cSaw_amp_sig <= Saw_amp_sig;
cPol_amp_sig <= Pol_amp_sig;
cPol_amp1_sig <= Pol_amp1_sig;
cPol_amp2_sig <= Pol_amp2_sig;
cPol_amp_div_sig<= (Pol_amp2_sig - Pol_amp1_sig); -- Difference calculation
giving gradient
cGrad_f_sig <= Grad_f_sig;
cPWM_frac_sig <= PWM_frac_sig;
cPWM_sig <= PWM_sig;
next_state <= state_2;

```

```

-----
-- Equation calculation
-----

```

```

When state_2 =>

```

```

cIndex512_sig <= Index512_sig;
cIndexNew_sig <= IndexNew_sig;
cLow_sig <= Low_sig;
cHigh_sig <= High_sig;
cSaw_tooth_sig <= Saw_tooth_sig;

```

```

-- Straight line amplitude approximation of polynomial
IF (Pol_amp1_sig(24)='1') THEN
  cPol_amp_sig <= (Pol_amp_div_sig*IndexNew_sig)+
    ("11"&Pol_amp1_sig&"0000000000000000");
  cGrad_f_sig <= '1';
ELSE
  cPol_amp_sig <= (Pol_amp_div_sig*IndexNew_sig)+
    ("00"&Pol_amp1_sig&"0000000000000000");
  cGrad_f_sig <= '0';
END IF;
-- Straight line amplitude calculation of saw-tooth signal
cSaw_amp_sig <= IndexNew_sig + Saw_tooth_sig;
cPWM_frac_sig <= PWM_frac_sig;
cPol_amp1_sig <= Pol_amp1_sig;
cPol_amp2_sig <= Pol_amp2_sig;
cPol_amp_div_sig <= Pol_amp_div_sig;
cPWM_sig <= PWM_sig;
next_state <= state_3;

-----
-- Binary search
-----
When state_3 =>

  cIndex512_sig <= Index512_sig;
  cSaw_tooth_sig <= Saw_tooth_sig;
  cPol_amp_sig <= Pol_amp_sig;
  cSaw_amp_sig <= Saw_amp_sig;
  cPol_amp1_sig <= Pol_amp1_sig;
  cPol_amp2_sig <= Pol_amp2_sig;
  cPol_amp_div_sig <= Pol_amp_div_sig;
  cGrad_f_sig <= Grad_f_sig;
  cPWM_sig <= PWM_sig;
  -- Call binary search procedure
  Binary_search;

-----
-- PWM-width calculation
-----
When state_4 =>

  cIndex512_sig <= Index512_sig;
  cIndexNew_sig <= IndexNew_sig;
  cLow_sig <= Low_sig;
  cHigh_sig <= High_sig;
  cSaw_tooth_sig <= Saw_tooth_sig;
  cSaw_amp_sig <= Saw_amp_sig;
  cPol_amp_sig <= Pol_amp_sig;
  cPol_amp1_sig <= Pol_amp1_sig;
  cPol_amp2_sig <= Pol_amp2_sig;
  cPol_amp_div_sig <= Pol_amp_div_sig;
  cGrad_f_sig <= Grad_f_sig;
  cPWM_frac_sig <= PWM_frac_sig;
  cPWM_sig <= (Index512_sig&"0000000000000000")+
    ("0000000000"&PWM_frac_sig(15 DOWNT0 1)); -- PWM width
                                                    calculation
  next_state <= state_0;

END CASE;
END PROCESS Combinational_process;
END ARCHITECTURE derivation;

```

## F5. Noise shaping

```

-----
-- Design unit : Noise shaper
-- File name   : Noise shaper.vhd
-- Description: This module executes the noise shaper filter. The function of
--             : this filter is to attenuate the requantized noise present in
--             : the audio band by moving it into the supersonic band.
-- System      : VHDL'93
-- Author       : Deon Jacobs
--             : Department of Electrical Engineering
--             : University of Stellenbosch
--             : Deonj@sun.ac.za
-- Revision    : Version 1.0 10/10/2005
-----

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY noise_shaper IS
PORT( clk60Meg_in      : IN STD_LOGIC;
      Reset_in        : IN STD_LOGIC;
      pulse352_in     : IN STD_LOGIC;
      PWM_in          : IN STD_LOGIC_VECTOR(23 DOWNTO 0);
      PWM_out         : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
END ENTITY noise_shaper;

ARCHITECTURE filter OF noise_shaper IS

    COMPONENT MAC_shaper
    PORT
    (
        dataa      : IN STD_LOGIC_VECTOR (13 DOWNTO 0);
        datab     : IN STD_LOGIC_VECTOR (30 DOWNTO 0);
        clock0     : IN STD_LOGIC := '1';
        aclr0      : IN STD_LOGIC := '0';
        result     : OUT STD_LOGIC_VECTOR (44 DOWNTO 0)
    );
    END COMPONENT;

    COMPONENT buf_shaper
    PORT
    (
        address    : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        clock      : IN STD_LOGIC ;
        data       : IN STD_LOGIC_VECTOR (13 DOWNTO 0);
        wren       : IN STD_LOGIC ;
        q          : OUT STD_LOGIC_VECTOR (13 DOWNTO 0)
    );
    END COMPONENT;

    COMPONENT Coef_shaper
    PORT
    (
        address    : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        clock      : IN STD_LOGIC ;
        q          : OUT STD_LOGIC_VECTOR (30 DOWNTO 0)
    );
    END COMPONENT;

```

```

-----
-- State Machine declerations
-----

TYPE State_machine IS (state_0,state_1,state_2,state_3);
SIGNAL current_state, next_state : State_machine;

-----

--Combinational signals
-----

-- MAC shaper
    SIGNAL cMAC_dataa_sig : STD_LOGIC_VECTOR(13 DOWNTO 0);
    SIGNAL cMAC_clr_sig   : STD_LOGIC;

-- Buffer shaper
    SIGNAL cBuf_adr_sig   : unsigned(3 DOWNTO 0);
    SIGNAL cBuf_dat_sig   : STD_LOGIC_VECTOR(13 DOWNTO 0);
    SIGNAL cBuf_wren_sig  : STD_LOGIC;

-- Coefficient shaper
    SIGNAL cCoef_adr_sig  : unsigned(3 DOWNTO 0);

-- Misc
    SIGNAL cState1_f_sig  : STD_LOGIC;
    SIGNAL cState2_f_sig  : STD_LOGIC;
    SIGNAL cState3_cnt_sig : unsigned(2 DOWNTO 0);
    SIGNAL cNshaper_filtout_sig : unsigned(23 DOWNTO 0);
    SIGNAL cPWM_out_sig   : STD_LOGIC_VECTOR(9 DOWNTO 0);

-----

-- Memory (latched) signals
-----

-- MAC shaper
    SIGNAL MAC_dataa_sig : STD_LOGIC_VECTOR(13 DOWNTO 0);
    SIGNAL MAC_datab_sig : STD_LOGIC_VECTOR(30 DOWNTO 0);
    SIGNAL MAC_clr_sig   : STD_LOGIC;
    SIGNAL MAC_result_sig : STD_LOGIC_VECTOR(44 DOWNTO 0);

-- Buffer shaper
    SIGNAL Buf_adr_sig   : unsigned(3 DOWNTO 0);
    SIGNAL Buf_dat_sig   : STD_LOGIC_VECTOR(13 DOWNTO 0);
    SIGNAL Buf_wren_sig  : STD_LOGIC;
    SIGNAL Buf_out_sig   : STD_LOGIC_VECTOR(13 DOWNTO 0);

-- Coefficient shaper
    SIGNAL Coef_adr_sig  : unsigned(3 DOWNTO 0);

-- Misc
    SIGNAL State1_f_sig  : STD_LOGIC;
    SIGNAL State2_f_sig  : STD_LOGIC;
    SIGNAL State3_cnt_sig : unsigned(2 DOWNTO 0);
    SIGNAL Nshaper_filtout_sig : unsigned(23 DOWNTO 0);
    SIGNAL PWM_out_sig   : STD_LOGIC_VECTOR(9 DOWNTO 0);

-----

-- Constants
-----

    CONSTANT Buf_adr_max : integer := 13;

```



```

BEGIN

MAC_shaper_inst : MAC_shaper PORT MAP (
    dataa    => MAC_dataa_sig,
    datab   => MAC_datab_sig,
    clock0   => clk60Meg_in,
    aclr0    => MAC_clr_sig,
    result   => MAC_result_sig
);

buf_shaper_inst : buf_shaper PORT MAP (
    address  => std_logic_vector(unsigned(Buf_adr_sig(3 DOWNT0 0))),
    clock    => clk60Meg_in,
    data     => Buf_dat_sig,
    wren     => Buf_wren_sig,
    q       => Buf_out_sig
);

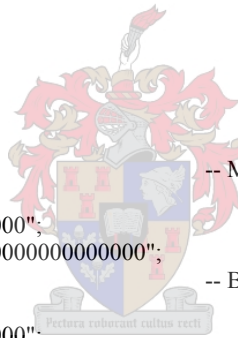
Coef_shaper_inst : Coef_shaper PORT MAP (
    address  => std_logic_vector(unsigned(Coef_adr_sig(3 DOWNT0 0))),
    clock    => clk60Meg_in,
    q       => MAC_datab_sig
);

-----
Register_process:
-----
PROCESS (clk60Meg_in,Reset_in) IS
BEGIN
    IF (reset_in = '1') THEN

        --MAC
        MAC_clr_sig    <='1';           -- MAC cleared
        --Buf
        Buf_adr_sig    <="0000";
        Buf_dat_sig    <="0000000000000000";
        Buf_wren_sig   <='0';           -- Buffer in read mode
        --Coef
        Coef_adr_sig   <="0000";
        --Misc
        State1_f_sig   <='0';
        State2_f_sig   <='0';
        State3_cnt_sig <="000";
        Nshaper_filtout_sig <="00000000000000000000000000000000";
        PWM_out_sig    <="0000000000";
        --State
        current_state  <= state_0;

    ELSIF rising_edge(clk60Meg_in) THEN

        --MAC
        MAC_dataa_sig  <=cMAC_dataa_sig;
        MAC_clr_sig    <=cMAC_clr_sig;
        --Buf
        Buf_dat_sig    <=cBuf_dat_sig;
        Buf_adr_sig    <=cBuf_adr_sig;
        Buf_wren_sig   <=cBuf_wren_sig;
        --Coef
        Coef_adr_sig   <=cCoef_adr_sig;
        --Misc
        State1_f_sig   <=cState1_f_sig;
        State2_f_sig   <=cState2_f_sig;
        State3_cnt_sig <=cState3_cnt_sig;
        Nshaper_filtout_sig <=cNshaper_filtout_sig;
        PWM_out_sig    <=cPWM_out_sig;
        PWM_out        <=cPWM_out_sig&"000000";
    
```



```

--State
current_state    <= next_state;

END IF;
END PROCESS Register_process;

-----
Combinational_process:
-----
PROCESS(current_state,PWM_in,pulse352_in,MAC_dataa_sig,MAC_clr_sig,Buf_dat_sig,Buf_adr_sig,
Buf_wren_sig,Coef_adr_sig,Buf_out_sig,State1_f_sig,State2_f_sig,State3_cnt_sig,Nshaper_filtout_sig,
MAC_result_sig,PWM_out_sig) IS

    PROCEDURE Noise_shaper_Quantization IS

        VARIABLE PWM_input_and_NShaper_output : unsigned(23 DOWNT0 0);

    BEGIN
        -- Addition of noise shaper output and 24-bit PWM_input
        IF (Nshaper_filtout_sig(23) = '1') THEN
            PWM_input_and_NShaper_output := unsigned(PWM_in(23 DOWNT0 0)) -
                ("0"&Nshaper_filtout_sig(22 DOWNT0 0));
        ELSE
            PWM_input_and_NShaper_output := unsigned(PWM_in(23 DOWNT0 0)) +
                Nshaper_filtout_sig;
        END IF;
        -- Quantization of PWM_input_and_NShaper_output variable
        cPWM_out_sig    <= std_logic_vector(
            unsigned(PWM_input_and_NShaper_output(23 DOWNT0 14)));
            -- Quantized 10-bit PWM output
        -- Quantization error
        cBuf_dat_sig    <= std_logic_vector(
            unsigned(PWM_input_and_NShaper_output(13 DOWNT0 0)));
            -- Quantization noise for filter input
    END PROCEDURE Noise_shaper_Quantization;

    BEGIN
    CASE (current_state) IS
    -----
    -- Reset State
    -----
    When state_0 =>

        --Mac
        cMAC_dataa_sig <= "00000000000000";
        cMAC_clr_sig   <= '1';
        --Buf
        cBuf_adr_sig   <= "0000";
        cBuf_dat_sig   <= "00000000000000";
        cBuf_wren_sig  <= '0';
        --Coef
        cCoef_adr_sig  <= "0000";
        --Misc
        cState1_f_sig  <= '0';
        cState2_f_sig  <= '0';
        cState3_cnt_sig <= "000";
        cNshaper_filtout_sig <= Nshaper_filtout_sig;
        cPWM_out_sig   <= PWM_out_sig;
        -- Noise shaping filter start decision
        IF (pulse352_in = '1') THEN
            next_state <= state_1;
        ELSE
            next_state <= state_0;
        END IF;
    
```

```

-----
-- PWM quantization and storage within buffer
-----
When state_1 =>

  cMAC_dataa_sig <= Buf_out_sig;
  cBuf_adr_sig   <= "0001";    -- Data value written to first buffer address value: 1h
  --Misc
  cState1_f_sig <='1';
  cState2_f_sig <='0';
  cState3_cnt_sig <="000";
  -- Coef
  cCoef_adr_sig   <= "0000";    -- First coefficient address
  cNshaper_filtout_sig <= Nshaper_filtout_sig;
  IF (State1_f_sig = '0') THEN

    -- Mac
    cMAC_clr_sig <='1';    -- Keep MAC cleared
    -- Buf
    Noise_shaper_Quantization;
    cBuf_wren_sig <= '1';    -- Write new data value to the noise shaper input buffer
    -- State
    next_state   <= state_1;

  ELSE

    -- Mac
    cMAC_clr_sig <='0';
    -- Buf
    cBuf_dat_sig <= "0000000000000000";
    cPWM_out_sig <= PWM_out_sig;
    cBuf_wren_sig <= '0';    -- Write new data value to the noise shaper input buffer
    -- State
    next_state   <= state_2;

  END IF;

-----
-- Filter calculation and Buffer shift
-----
When state_2 =>

  cMAC_clr_sig <='0';    -- MAC arithmetic enabled
  cState1_f_sig <='0';
  cState3_cnt_sig <="000";
  cNshaper_filtout_sig <= Nshaper_filtout_sig;
  cPWM_out_sig <= PWM_out_sig;
  IF (to_integer(Buf_adr_sig) < Buf_adr_max+2) THEN
    IF (State2_f_sig = '0') THEN
      cMAC_dataa_sig <= Buf_out_sig;    -- Current buffer output
                                          filtered at next clock edge
      cBuf_dat_sig <= Buf_out_sig;    -- Buffer output connected
                                          to buffer input for buffer shift
      cBuf_wren_sig <= '1';    -- Overwrite current buffer value with
                                          previous buffer value
      cBuf_adr_sig <= Buf_adr_sig;    -- Keep current buffer address
      cCoef_adr_sig <= Coef_adr_sig;    -- Keep current buffer address
      cState2_f_sig <= '1';    -- Reset flag for next data storage
      next_state <= state_2;
    ELSE
      -- Give opportunity for storing previous value at current address
      -- and update buffer and filter coefficient lookup table address
      cMAC_dataa_sig <= "0000000000000000";    -- No new input data to noise shaping
                                          filter
      cBuf_dat_sig <= "0000000000000000";    -- "
      cBuf_wren_sig <= '0';    -- Write new data value to the noise
                                          shaper input buffer
      cBuf_adr_sig <= Buf_adr_sig +1;
    END IF;
  END IF;

```

```

        cCoef_adr_sig <= Coef_adr_sig +1;
        cState2_f_sig <= '0'; -- Reset flag for next data storage
        next_state <= state_2;
    END IF;
ELSE
    -- filtering finished
    cMAC_dataa_sig<= "00000000000000";
    cBuf_dat_sig <= "00000000000000"; -- Clear input signal of buffer after data store
    cBuf_wren_sig <= '0'; -- Write new data value to the noise shaper input buffer
    cBuf_adr_sig <= "0000"; -- Reset buffer address
    cCoef_adr_sig <= "0000"; -- Reset Filter coefficient lookup table
    cState2_f_sig <= '0'; -- Reset flag for next data storage
    next_state <= state_3;
END IF;
-----
-- Wait state until MAC arithmetic finished and assign noise shaping filter output to register
-----
When state_3 =>

    cMAC_clr_sig <='0'; -- MAC arithmetic enabled
    cState1_f_sig <='0';
    cMAC_dataa_sig <= "00000000000000";
    cBuf_dat_sig <= "00000000000000"; -- Clear input signal of buffer after data store
    cBuf_wren_sig <= '0'; -- Write new data value to the noise shaper input buffer
    cBuf_adr_sig <= "0000";
    cCoef_adr_sig <= "0000"; -- First coefficient address
    cState2_f_sig <= '0'; -- Reset flag for next data storage
    cPWM_out_sig <=PWM_out_sig;
    IF (to_integer(State3_cnt_sig) < 5) THEN
        IF (to_integer(State3_cnt_sig) = 4) THEN -- Time to assign filter output to Noise
            shaper filter signal
            IF (MAC_result_sig(44) = '1') THEN -- Negative MAC result
                cNshaper_filtout_sig <="1000000000"&
                    (unsigned(not(MAC_result_sig(43 DOWNT0 0)))+1);
                --Top 14 bits of MAC result assigned as lower 14 bits
            ELSE --to be added to PWM input, MAC_result(44) is a sign bit
                -- Positive MAC result
                cNshaper_filtout_sig <="0000000000"&
                    (unsigned(MAC_result_sig(43 DOWNT0 30)));
            END IF;
        ELSE -- Not time yet to assign Noise shaper filter signal (MAC not finished)
            cNshaper_filtout_sig <= Nshaper_filtout_sig;
        END IF;
        State3_cnt_sig <= State3_cnt_sig +1;
        next_state <= state_3;
    ELSE
        -- Assignment complete, wait for next 24-bit PWM signal
        cNshaper_filtout_sig <= Nshaper_filtout_sig;
        cState3_cnt_sig <="000";
        next_state <= state_0;
    END IF;
END CASE current_state;
END PROCESS Combinational_process;
END ARCHITECTURE filter;

```

## F6. PWM generator

```
-----
-- Design unit : Pulse width modulation unit
-- File name   : PWM_gen.vhd
-- Description  : File generates a pulse width signal from a 16 bit data input,
--               : the data input is compared to a saw-tooth wave having a gradient
--               : of 1. The switching frequency of the PWM signal is at 352.8Khz.
--               : The resolution of the PWM signal is 8-bits, and the method of PWM is
--               : leading edge Uniform Pulse width modulation
-- System      : VHDL'93
-- Author      : Deon Jacobs
--              : Department of Electrical Engineering
--              : University of Stellenbosch
--              : Deonj@sun.ac.za
-- Revision    : Version 1.3 21/08/2005
-----
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
```

```
ENTITY make_pwm_352 IS
PORT(  pwm_width_data_in  : in std_logic_vector(15 downto 0);
        clk_90_3Meg       : in std_logic;
        data_ready        : in std_logic;
        PWM_out           : out std_logic
      );
END ENTITY make_pwm_352;
```

```
ARCHITECTURE Compare OF make_pwm_352 IS
```

```
    TYPE PWM_state_machine IS (setup_1,cycles_high_2,cycles_low_3);
    SIGNAL present_state,next_state : PWM_state_machine;
```

```
    -- Pulse duration execution process constants
    CONSTANT pulse_resolution : integer:=256;
```

```
    -- Data_ready_detect process Signals
    SIGNAL dr_detect_f_sig      : std_logic;      -- Flag Set when data is detected
```

```
    -- PWM calculate logic signals
    SIGNAL cpwm_calc_f_sig      : std_logic;      -- Flag set when PWM is calculated
    SIGNAL cClk_cycles_high_sig : unsigned(7 downto 0); -- Clock instances PWM signal is high
    SIGNAL cClk_cycles_low_sig  : unsigned(7 downto 0); -- Clock instances PWM signal is low
    SIGNAL cPWM_sig             : std_logic;      -- PWM ouput
    SIGNAL cWhen_flag_f_sig     : std_logic;      -- Indicates when cpwm_calc_f_sig needs to be toggled
    SIGNAL cCnt_cycles_high_sig : unsigned(7 downto 0); -- Counter for PWM high clocks
    SIGNAL cCnt_cycles_low_sig  : unsigned(7 downto 0); -- Counter for PWM low clocks
    SIGNAL cLEDS_sig           : std_logic_vector(2 downto 0);
```

```
    -- PWM generate latch signals
    SIGNAL pwm_calc_f_sig       : std_logic;      -- Flag set when PWM is calculated
    SIGNAL Clk_cycles_high_sig  : unsigned(7 downto 0); -- Clock instances PWM signal is high
    SIGNAL Clk_cycles_low_sig   : unsigned(7 downto 0); -- Clock instances PWM signal is low
    SIGNAL clk_inst_sig         : std_logic_vector(7 downto 0); -- Signal holding PWM clock width in
                                                                # of clk-cycles
    SIGNAL PWM_out_sig          : std_logic;      -- PWM ouput
    SIGNAL When_flag_f_sig     : std_logic;      -- Indicates when cpwm_calc_f_sig needs to be toggled
    SIGNAL Cnt_cycles_high_sig  : unsigned(7 downto 0); -- Counter for PWM high clocks
    SIGNAL Cnt_cycles_low_sig   : unsigned(7 downto 0); -- Counter for PWM low clocks
    SIGNAL LEDES_sig           : std_logic_vector(2 downto 0);
```

```

BEGIN
-----
Data_ready_detect:
-----
    PROCESS (clk_90_3Meg) IS
    BEGIN
        IF rising_edge(clk_90_3Meg) THEN
            IF (data_ready = '1') THEN
                IF ((pwm_calc_f_sig = '0') and (dr_detect_f_sig = '0')) THEN
                    dr_detect_f_sig <= '1';
                    clk_inst_sig    <= pwm_width_data_in(15 DOWNT0 8);

                ELSIF ((pwm_calc_f_sig = '1') and (dr_detect_f_sig = '1')) THEN
                    dr_detect_f_sig <= '0';
                    clk_inst_sig    <= pwm_width_data_in(15 DOWNT0 8);

                ELSE
                    dr_detect_f_sig <= dr_detect_f_sig;
                    clk_inst_sig    <= clk_inst_sig;

                END IF;
            END IF;
        END IF;
    END PROCESS Data_ready_detect;
-----

PWM_gen_latch:
-----
    PROCESS (clk_90_3Meg) IS
    BEGIN
        IF rising_edge(clk_90_3Meg) THEN

            -- Internal signals
            present_state <= next_state;           -- Latch new state result
            PWM_out_sig   <= cPWM_sig;             -- Latch new PWM output
            pwm_calc_f_sig <= cpwm_calc_f_sig;    -- Latch PWM process sync flag
            Clk_cycles_high_sig <= cClk_cycles_high_sig;
            Clk_cycles_low_sig <= cClk_cycles_low_sig;
            When_flag_f_sig <= cWhen_flag_f_sig;
            Cnt_cycles_high_sig <= cCnt_cycles_high_sig;
            Cnt_cycles_low_sig <= cCnt_cycles_low_sig;
            LEDS_sig       <= cLEDS_sig;

            -- Output registers
            --clk_inst_out   <= clk_inst_sig;
            PWM_out         <= PWM_out_sig;
            --Clk_cycles_high_out <= std_logic_vector(Clk_cycles_high_sig(7 downto 0));
            --Clk_cycles_low_out  <= std_logic_vector(Clk_cycles_low_sig(7 downto 0));
            --Cnt_cycles_high_out <= std_logic_vector(Cnt_cycles_high_sig(7 downto 0));
            --LEDS_out        <= LEDS_sig;

        END IF;
    END PROCESS PWM_gen_latch;
-----

PWM_gen_logic:
-----
    PROCESS(present_state,When_flag_f_sig,Cnt_cycles_low_sig,Cnt_cycles_high_sig,dr_detect_f_sig,
    PWM_out_sig,pwm_calc_f_sig,clk_inst_sig,Clk_cycles_high_sig,Clk_cycles_low_sig) IS

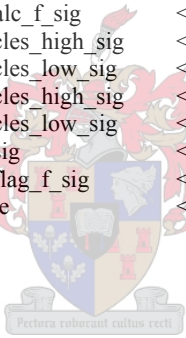
        PROCEDURE State_setup IS-- Sets up variables to execute PWM
        BEGIN
            cpwm_calc_f_sig    <= pwm_calc_f_sig;
            cClk_cycles_high_sig <= to_unsigned(to_integer(unsigned(clk_inst_sig)),8);
                                   -- Assign # of high clk cycles
            cClk_cycles_low_sig <= pulse_resolution - Clk_cycles_high_sig; -- Assign # of low
                                   clk cycles

```

```

IF (to_integer(Clk_cycles_high_sig) > 0) THEN
    cPWM_sig          <=&'1';
    cCnt_cycles_high_sig <=&"00000001";
    cCnt_cycles_low_sig  <=&Cnt_cycles_low_sig;
    next_state         <=& cycles_high_2;
ELSE -- Clk_cycles_high = 0

    cPWM_sig          <=&'1';
    cCnt_cycles_low_sig <=&"00000001";
    cCnt_cycles_high_sig <=&Cnt_cycles_high_sig;
    next_state         <=& cycles_low_3;
END IF;
END PROCEDURE State_setup;

BEGIN
CASE (Present_state) IS
WHEN setup_1 =>
    cLEDS_sig          <=& "001";
    IF ((pwm_calc_f_sig = '0') and (dr_detect_f_sig = '1')) THEN
        cWhen_flag_f_sig <=&'1'; -- pwm_calc_f_sig = 1 (later)
        State_setup;
    ELSIF ((pwm_calc_f_sig = '1') and (dr_detect_f_sig = '0')) THEN
        cWhen_flag_f_sig <=&'0'; -- pwm_calc_f_sig = 0 (later)
        State_setup;
    ELSE
        cpwm_calc_f_sig <=&pwm_calc_f_sig;
        cClk_cycles_high_sig <=&Clk_cycles_high_sig;
        cClk_cycles_low_sig <=&Clk_cycles_low_sig;
        cCnt_cycles_high_sig <=&Cnt_cycles_high_sig;
        cCnt_cycles_low_sig <=&Cnt_cycles_low_sig;
        cPWM_sig <=&PWM_out_sig;
        cWhen_flag_f_sig <=&When_flag_f_sig;
        next_state <=&Setup_1;
    END IF;
WHEN cycles_high_2 =>

    cLEDS_sig          <=& "010";
    cClk_cycles_low_sig <=& Clk_cycles_low_sig;
    cClk_cycles_high_sig <=& Clk_cycles_high_sig;
    cWhen_flag_f_sig <=& When_flag_f_sig;

    IF (to_integer(Cnt_cycles_high_sig) <=& to_integer(Clk_cycles_high_sig)) THEN

        cPWM_sig          <=& '1';
        cCnt_cycles_high_sig <=& Cnt_cycles_high_sig + 1;
        cCnt_cycles_low_sig <=& Cnt_cycles_low_sig;
        IF (to_integer(Cnt_cycles_high_sig) = (pulse_resolution-1)) THEN
            cpwm_calc_f_sig <=& pwm_calc_f_sig;
            next_state <=& setup_1;
        ELSE
            cpwm_calc_f_sig <=& pwm_calc_f_sig;

            IF (to_integer(Cnt_cycles_high_sig) = 50) THEN
                IF (When_flag_f_sig = '1') THEN
                    cpwm_calc_f_sig <=& '1';
                ELSE
                    cpwm_calc_f_sig <=& '0';
                END IF;
            ELSE
                cpwm_calc_f_sig <=& pwm_calc_f_sig;
            END IF;
            next_state <=& cycles_high_2;
        END IF;
    ELSE

```

```

        cPWM_sig          <= '0';
        cCnt_cycles_high_sig <= Cnt_cycles_high_sig;
        IF (to_integer(Clk_cycles_low_sig) = 1) THEN
            cCnt_cycles_low_sig <= Clk_cycles_low_sig;
            IF (When_flag_f_sig = '1') THEN
                cpwm_calc_f_sig <= '1';
            ELSE
                cpwm_calc_f_sig <= '0';
            END IF;
            next_state <= setup_1;
        ELSE
            cpwm_calc_f_sig <= pwm_calc_f_sig;
            cCnt_cycles_low_sig <= "00000001";
            next_state <= cycles_low_3;
        END IF;
    END IF;
WHEN cycles_low_3 =>

    cLEDS_sig          <= "100";
    cPWM_sig           <= '0';
    cClk_cycles_high_sig <= Clk_cycles_high_sig;
    cClk_cycles_low_sig <= Clk_cycles_low_sig;
    cWhen_flag_f_sig   <= When_flag_f_sig;
    cCnt_cycles_high_sig <= Cnt_cycles_high_sig;

    IF (to_integer(Cnt_cycles_low_sig) < to_integer(Clk_cycles_low_sig-1)) THEN
        IF (to_integer(Cnt_cycles_low_sig) = 50) THEN
            IF (When_flag_f_sig = '1') THEN
                cpwm_calc_f_sig <= '1';
            ELSE
                cpwm_calc_f_sig <= '0';
            END IF;
        ELSE
            cpwm_calc_f_sig <= pwm_calc_f_sig;
        END IF;
        cCnt_cycles_low_sig <= Cnt_cycles_low_sig + 1;
        next_state <= cycles_low_3;
    ELSE
        cpwm_calc_f_sig <= pwm_calc_f_sig;
        cCnt_cycles_low_sig <= Cnt_cycles_low_sig;
        next_state <= setup_1;
    END IF;
END CASE;
END PROCESS PWM_gen_logic;
END ARCHITECTURE Compare;

```