University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations

2001

Inner product computational architectures using the double base number system.

Stanley Jonathan. Eskritt
University of Windsor

Follow this and additional works at: http://scholar.uwindsor.ca/etd

Recommended Citation

Eskritt, Stanley Jonathan., "Inner product computational architectures using the double base number system." (2001). *Electronic Theses and Dissertations.* Paper 2453.

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

"Inner Product Computational Architectures Using the Double Base Number System"

by

"Stanley Jonathan Eskritt"

A Thesis
Submitted to the Faculty of Graduate Studies and Research through the Department of Electrical and Computer Engineering in partial fulfillment of the requirements for the Degree of Master of Applied Science at the University of Windsor

Windsor, Ontario, Canada

2001

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-62212-6

Canada

© 2001 "Stanley Jonathan Eskritt"

All Rights Reserved. No part of this document may be reproduced, stored or otherwise retained in a retrieval system or transmitted in any form, on any medium or by any means without the prior written permission of the author.

To my parents for their encouragement, and to Daniella for her support.

List of Symbols

m	*mantissa*
r	*base or radix*
SNR	*Signal to Noise Ratio*
Φ	*Look-up table operator for DBNS addition*
Ψ	*Look-up table operator for DBNS subtraction*
t	*DBNS ternary exponent*
b	*DBNS binary exponent*

List of Abbreviations

ALU	*Arithmetic Logic Unit*
CMOS	*Complementary Metal-Oxide Semiconductor*
CNN	*Cellular Neural Network*
dB	*Decibels*
DBNS	*Double Base Number System*
DFT	*Discrete Fourier Transform*
DSP	*Digital Signal Processing*
FIR	*Finite Impulse Response*
HDL	*Hardware Description Language*
IC	*Integrated Circuit*
IIR	*Infinite Impulse Response*
LNS	*Logarithmic Number System*
MAC	*Multiply ACcumulate*
MOSFET	*Metal-Oxide Semiconductor Field Effect Transistor*
PE	*Processing Elements*
ROM	*Read Only Memory*
RAM	*Random Access Memory*
SFG	*Signal Flow Graph*
SNR	*Signal to Noise Ratio*
VLSI	*Very Large Scale Integration*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*

Acknowledgments

There are several people who deserve my sincere thanks for their generous contributions to this project.

I would first like to thank my supervisor Dr. G. A. Jullien for his guidance, advice and for bringing this challenging project to my attention. I am grateful to Gennum Corp. for providing funding. I would also like to thank my committee members Dr. Vassil Dimitrov, Dr. Alioune Ngom.

I would also like to recognize the following individuals and corporations for their contributions: Paul Horbal for his time and comments on the first drafts of this thesis, CMC for providing and supporting the design software and computing hardware which made this project possible, Micronet R&D for providing financial and networking support, Roberto Muscedere for his aid with HDLs, Cadence tools and helping in producing the hybrid DBNS microchip.

Abstract

Digital signal processing (DSP) permeates many of the products we see around us today. DSP systems manipulate signals as sequences of numbers, and require massive arithmetic computations to perform algorithmic processing such as filtering. Traditionally these systems are designed using the binary number system to perform the computations. This thesis presents an exploration into the construction of finite impulse response (FIR) filters using a recently introduced number system. The number system uses two orthogonal bases and has been referred to in literature as the Double-Base Number System (DBNS). We use an index calculus implementation of the DBNS to take advantage of the logarithmic-like properties of the associated arithmetic.

Using the index calculus form of the Double Base Number System (DBNS), instead of the classical binary representation, a number of advantages are gained. The logarithmic-like properties of the index calculus DBNS allow for reduced complexity multiplication and division, expensive operations using binary arithmetic; the orthogonal nature of the index computations and the multi-digit extensions of the representation reduce the complexity of computations compare to the Logarithmic Number System. Finally, the non-linear nature of the representation allows for a natural mapping for some special systems, such as hearing devices, that require non-linear compression operations.

This thesis investigates a recently disclosed architecture for inner product computations in the DBNS. This architecture is expanded from the initial single-digit DBNS form to a hybrid (mix of 1-digit and 2-digit forms) and finally a full 2-digit form, and we show the efficiencies obtained in using the full 2-digit form. We target the implementation of large tap length Finite Impulse Response (FIR) filters, and we examine both the problem of finite precision index coefficient design and the design and fabrication of a systolic

Abstract

architecture using DBNS processors. In particular we introduce a mapping scheme that allows the output of full-precision filter design packages to be easily converted to finite precision DBNS indices with close to optimum results, and we fabricate and successfully test a 15-tap filter design using a 0.35μ CMOS process. We finally examine the use of asymmetrical dynamic ranges on the 2 orthogonal indices in order to reduce the area and power of the filter architecture with minimal reduction in precision.

Publications and Presentations

The following refereed conference publications and invited talks were generated from the work presented in this thesis.

[1] V.S.Dimitrov, J.Eskritt, L.Imbert, G.A.Jullien and W.C.Miller. "*The use of the multi-dimensional logarithmic number system in DSP applications.*" 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, June 2001. Accepted for presentation.

[2] J. Eskritt, "*2-digit, 2-Dimensional Logarithms for Efficient FIR Filter Architectures*", invited presentation at the Micronet Annual Workshop, April 2001 (accepted for presentation).

[3] J. Eskritt, "*Hybrid and 2-digit DBNS Filter Architectures*", invited presentation, Gennum Corporation, February 2001.

[4] J. Eskritt, "*Inner Product Computional Architectures Using the Double Base Number System*", VLSI Research Group, University of Windsor, December 2000.

[5] J. Eskritt, R. Muscedere, G.A. Jullien, V.S. Dimitrov and W.C. Miller. "A *2-Digit DBNS Filter Architecture.*" Proceedings of the 2000 IEEE Workshop on Signal Processing Systems (SiPS 2000), Lafayette, LA, October. Pages 447-456.

[6] J. Eskritt, "*Inner Product DBNS Computation*", VLSI Research Group, University of Windsor, September 2000.

[7] J. Eskritt, "*DBNS Research for the Micronet Project S.1.WII*", invited presentation, Gennum Corporation, March 2000.

Table of Contents

Chapter 1 Introduction **1**

1.1 Introduction 1
1.2 VLSI Technology 3
1.3 Double Base Number System 5
1.4 Thesis Objectives 6
1.5 Thesis Organization 6

Chapter 2 The Double Base Number System **8**

2.1 Introduction 8
2.2 Representation 8
2.3 Index Calculus 13
2.3.1 Multiplication and Division 13
2.3.2 Addition and Subtraction 14
2.3.3 Multi-digit DBNS Arithmetic 16
2.4 Asymmetric Exponents 16
2.5 Logarithmic Number System 18
2.6 Different Odd Bases 20
2.7 Conversion Between Binary and DBNS 21
2.7.1 Binary to DBNS Conversion 22
2.7.2 DBNS to Binary conversion [9] 22
2.8 Summary 23

Chapter 3 DBNS Finite Precision FIR Filter Design **24**

3.1 Introduction 24
3.2 Coefficient Design Techniques 24
3.2.1 Genetic Algorithms 25
3.2.2 Integer Programming 28

3.2.3 Direct mapping 29
3.3 Summary 37

Chapter 4 DBNS FIR Filter Architectures 38

4.1 Introduction 38
4.2 Systolic Arrays 38
4.2.1 Pipelining 40
4.3 Basic Processing Element 41
4.4 Hybrid 2 digit DBNS Architecture 43
4.5 2 Digit DBNS Architecture 44
4.5.1 Two Digit Channel Reduction 46
4.6 Comparison to Binary 50
4.7 Summary 52

Chapter 5 A DBNS FIR Filter Case Study 53

5.1 Introduction 53
5.2 Verilog Simulation 53
5.3 VHDL Synthesis 56
5.3.1 Placement and Routing 58
5.4 Final Chip Assembly 61
5.5 Testing the Chip 62
5.6 Comparison to 2 digit DBNS 63
5.7 Summary 65

Chapter 6 Conclusions and Future Work 66

6.1 Conclusions 66
6.2 Contributions 67
6.3 Suggestions for Future Work 67

REFERENCES 69

Appendix A VHDL Code for Hybrid DBNS FIR Filter 71

A.1 Introduction 71
A.2 Binary to DBNS Coversion Stage 71
A.3 DBNS MAC Cell 73

Appendix B VHDL Code for 2 Digit DBNS FIR Filter 85

B.1 Introduction 85
B.2 MAC Cell 85

Appendix C C Code for Mapping into DBNS.. 89

C.1 Introduction 89
C.2 Optimal Mapping Listing 89
C.3 Modified Greedy Mapping Listing 96

Appendix D Genetic Algorithm Code.. 102

D.1 DBNS Coefficient Generation 102

Appendix E A Brief Tutorial in High Level Chip Design.................................. 113

E.1 Introduction 113
E.2 System Design 113
E.3 Synthesizing the System 115

List of Figures

Figure 1.1 Block diagram of a digital filter 2
Figure 2.1 A 2D DBNS Representation of 79 10
Figure 2.2 A 2D DBNS Representation of 7.25 11
Figure 2.3 Magnitude response comparison for asymmetric exponents 17
Figure 2.4 Error for symmetric and asymmetric responses 18
Figure 2.5 Comparison between LNS and DBNS filter responses 20
Figure 2.6 Magnitude response of DBNS filters with different odd bases 21
Figure 3.1 Crossover 26
Figure 3.2 Frequency response of a genetic algorithm produced DBNS FIR filter 28
Figure 3.3 Result curve due to optimization 29
Figure 3.4 A histogram showing the coefficient values for over 200 filters 30
Figure 3.5 Filter response of a floating point coefficient filter 32
Figure 3.6 Magnitude response for the mapped filters 34
Figure 3.7 Mapped filters' error 34
Figure 3.8 Error of mapped DBNS responses 35
Figure 3.9 Error between designed response and binary and DBNS mappings 36
Figure 3.10 Error between designed coefficient values and mapped coefficients 37
Figure 4.1 A unidirectional systolic array convolver and processing element 39
Figure 4.2 A DBNS Multiply Accumulate Cell 42
Figure 4.3 Hybrid 2 digit DBNS Architecture 44
Figure 4.4 Four channel filter architecture for 2 digit DBNS 45
Figure 4.5 2 digit reduced channel architecture 48
Figure 4.6 Comparison between 3 channel filtered signal and floating point signal 48
Figure 4.7 Error between input signal and filtered signals 49
Figure 4.8 Binary filter responses 50
Figure 4.9 Binary multiplier cell (660x925mm) 51
Figure 5.1 Hierarchal diagram for HDL description of a DBNS filter 54
Figure 5.2 Magnitude response of genetic algorithm designed DBNS FIR filter 55

Figure 5.3 Input chirp signal ranging from 1kHz to 512kHz 56
Figure 5.4 Filtered chirp signal 57
Figure 5.5 Input Stage. Binary to DBNS Conversion ROM (2969x666mm) 60
Figure 5.6 DBNS MAC cell (3526x857 mm) 60
Figure 5.7 Output Stage. Accumulator for both Filter Channels (328x113 mm) 60
Figure 5.8 Final Chip Layout (16x9 mm) 62
Figure 5.9 15 tap. hybrid DBNS FIR filter (left) and a 53 tap. reduced channel. 2 digit DBNS FIR filter (right) 64

List of Tables

Table 2.1 Stopband Attenuation (in Decibels) for Mapped DBNS Filter Coefficients with Varying Bases 21
Table 3.1 Mapped Filter Specifications 33
Table 4.1 Unidirectional systolic convolver results 40
Table 4.2 53DBNS bandpass FIR filter coefficients 47
Table 4.3 Area comparison between binary and DBNS filter taps 52

Chapter 1

Introduction

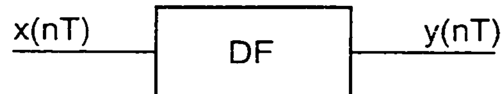
1.1 Introduction

Digital signal processing (DSP) increasingly permeates our lives. Almost every field of science and engineering relies on processing signals. Signals can be received and analyzed through test equipment or generated through simulation hardware. Signals are responsible for allowing us to communicate over large distances and for entertaining us through radio and television. DSP comes into play when these signals are discrete. Discrete signals are signals that exist only at specified points in time and can only assume a finite number of values [16]. DSP is used in many areas, such as digital filtering, image processing and even computer systems.

The representation of signals as discrete is not a large constraint, as non-discrete bandlimited signals are easily made discrete. Non-discrete signals, called continuous signals, exist at every point in a specified time or space. For example, the speed of a car or a person talking into a telephone. These signals can be quantized and converted into digital signals. To quantize a signal, it is measured at specific time intervals and the value is recorded to within a certain numerical precision.

A fundamental operation in DSP is filtering. By filtering a signal, unwanted portions of the signal spectrum can be removed or the signal can be modified and reshaped. For example, filtering is used to provide the selected channels for display on a television or for selecting stations on a radio. A basic block diagram of a digital filter is shown in Figure 1.1, where $x(nT)$ is the discrete input and $y(nT)$ is the discrete response of the filter.

Figure 1.1: Block diagram of a digital filter



This can be represented mathematically as:

$$y(nT)) = \Re[x(nT)] \tag{1.1}$$

where $\Re$ is an operator representing the transfer function of the filter. Furthermore, filters are classified by whether or not they are time-invariant, linear or causal [16].

<u>Definition 1.1</u> *Time-invariance* – A filter is time-invariant if the filter's response to a given input signal does not depend on the time the input signal is applied.

<u>Definition 1.2</u> *Linearity* – A filter is considered linear if it can satisfy the following two conditions.

$$\Re[\alpha x(nT)] = \alpha\Re[x(nT)] \tag{1.2}$$

$$\Re[x_1(nT) + x_2(nT)] = \Re[x_1(nT)] + \Re[x_2(nT)] \tag{1.3}$$

<u>Definition 1.3</u> *Causality* – A filter is said to be causal if the response of the filter, at a given moment, does not depend on any future value of the input.

One of the most basic structures used in DSP is the Finite Impulse Response (FIR) filter, or nonrecursive filter. A FIR filter is a filter that does not depend on any past output values

of the filter. If it is assumed that the FIR filter is linear, time-invariant and causal, then the response of the filter, $y(nT)$, can be expressed as a difference equation.

$$y(nT) = \sum_{i=0}^{N} a_i x(nT - iT) \tag{1.4}$$

where a_i represents constants, and N is the order of the filter. A benefit of using nonrecursive filters is that they are guaranteed to be linear and have a linear phase response.

Another type of filter is the Infinite Impulse Response (IIR) filter or recursive filter. An IIR filter uses past outputs to influence the current response of the filter. Recursive filters usually require a much lower order of filter to produce the same magnitude response of a FIR filter, but IIR filters are not guaranteed to be stable or have linear phase. The output response, $y(nT)$, of a recursive filter can be described by the difference equation:

$$y(nT) = \sum_{i=0}^{N} a_i x(nT - iT) - \sum_{i=1}^{M} b_i y(bT - iT) \tag{1.5}$$

where a_i and b_i represent constants. Therefore, at any given moment, the response of the filter is dependent on the past N values of the input and the past M values of the output.

For applications that require a linear phase response, such as video and audio applications, FIR filters are typically used. This is due to the guarantee of linear phase response, ease of design and stability.

1.2 VLSI Technology

VLSI (Very Large Scale Integration) technology has changed the lives of almost everyone in the world today. It is the technology that produces high density microchips. Incredibly sophisticated systems can be built on a piece of silicon the size of a fingernail. These

microchips exist in almost every device or product we buy, from computers, stereos and cellular phones to children's toys and household appliances.

VLSI was first introduced in 1977 to describe commercial products being sold at that time. It was used to describe any integrated circuit of greater than 2000 logic blocks [29]. Today's microchips consist of considerably more logic blocks than 2000. In the past 20 years the transistors used to form VLSI circuits have increase in switching speeds by over 20 times, while decreasing the area used by the transistor to less than 1% [28].

The transistor was invented in 1947. This transistor was a bipolar transistor. Bipolar transistors were a great advance to electronic circuits that previously used vacuum tubes, and circuits were reduced to a fraction of their previous size and power requirements. It took eleven years (1958) before the first integrated circuit was made, this further reduced the size and power requirements of electronic circuits. In 1960, the first MOSFET transistors (both p-channel and n-channel) were invented. The MOSFET transistors were easier to manufacture, and had better packing densities than the bipolar transistors, but the MOSFETs had slower switching speeds. In 1963, Complementary Metal-Oxide Semiconductor (CMOS) circuits were invented. CMOS offered the ease of manufacture and packing densities of MOSFET transistors, but had the added benefit of negligible standby power dissipation. The only time power is consumed is when a circuit is switched. This offered a tremendous advantage over bipolar transistors which dissipated a lot of power. Bipolar transistors are still in use today, in circuits that require the raw circuit speed of bipolar transistors, but are typically used in small stand alone bipolar chips, or in a BiCMOS process that uses CMOS for the majority of the circuits and only a small number of bipolar circuits [28].

Microchips are produced by forming multiple specialized layers. CMOS transistors are formed by diffusing gaseous boron and phosphorous into a wafer of single crystal silicon and using polycrystalline silicon as the gate electrode. Connections are made by layers of metal traces that are insulated from each other. These traces are usually made of aluminum. Current processes have up to seven layers of metal available. The more layers

of metal available to the process, the more densely packed the system can be, allowing for larger systems to be produced in the same area.

The current trend in VLSI design is to produce high quality libraries of basic cells, logic gates and basic operations (adders, counters, multiplexers, etc.) and design the system using a hardware description language (HDL). HDLs allow a system to be designed using a technology independent, high level, programming-like language. Once the system is fully described in an HDL, it can be compiled from components of the libraries and fabricated. This process leads to more modular designs and greatly speeds up the design and testing time of a chip, at the expense of not being optimally designed.

1.3 Double Base Number System

In the area of digital signal processing considerable demand exists for compact, high speed/real time digital filters for use in applications such as radar image processing. However, available real-time digital filters are often too slow, too costly, too complex or require too much power. To correct for this, the attention of some researchers has been directed to different techniques for designing high speed digital filters making use of special coding schemes or number systems. Traditional algorithms used in digital signal processing do not "parallelize" nor "modularize" easily, hence not allowing the maximum use of the VLSI technology. It becomes clear that different design approaches are needed that can incorporate modularity and parallelism. One such approach is a multi-digit Double Base Number System (DBNS), whose arithmetic is broken up into independent sub-terms, providing a modular hardware implementation.

The DBNS is a recently developed number system [8], though it does have some relation to the Logarithmic Number System (LNS), and can be viewed as a multi-dimensional form of the LNS [10]. DBNS offers a highly redundant number system with a very sparse representation, through the used of two orthogonal bases.

A large number of DSP functions, such as FIR filtering and Discrete Fourier Transform (DFT), require the use of inner product computations. Inner product computations are based on multiply-accumulate (MAC) operations. Therefore, specialized DSP hardware, such as FIR filter microchips, heavily rely on optimized MAC operations. Through the use of DBNS, a MAC architecture can be built without the need of binary multipliers, thus reducing the overhead of MAC dependent systems such as filters. DBNS also has a non-linear number representation that can be advantageous in some systems.

1.4 Thesis Objectives

There are two main thrusts of the work presented in this thesis. The first is to investigate and expand upon architectures for use with DBNS and FIR filters and, using these architectures, make a comparison to equivalent binary architectures. The second thrust is to provide proof of concept by producing a working microchip using the VHSIC Hardware Description Language (VHDL). Also investigated is the mapping of floating point numbers into the DBNS representation.

1.5 Thesis Organization

This thesis is organized into six chapters and four appendices. The first chapter (this chapter) is an introduction. Chapter 2 covers the Double Base Number System, and the arithmetic associated with it. The second chapter also makes a comparison between the DBNS and the LNS number systems. Chapter 3 presents the methods used to generate the coefficients for the DBNS filters. Chapter 4 looks at the architectures used in producing a FIR filter that uses DBNS for filter coefficients and data. Chapter 5 provides verification of the DBNS architectures through the production of a working microchip of a 15-tap, hybrid DBNS, FIR filter, designed using VHDL. Finally, Chapter 6 presents conclusions and future work for this thesis.

The appendices include additional information that did not fit neatly into the thesis body. Appendix A and Appendix B contain the VHDL code used to produce the hybrid and two

digit DBNS microchip described in Chapter 5. Appendix C contains the code for the decimal to DBNS mapping of data and coefficients. Appendix D contains the code for the genetic algorithm used for generating DBNS FIR filter coefficients. Finally, Appendix E provides a brief introduction to high level microchip design, and points the reader to more complete tutorials and information.

Chapter 2

The Double Base Number System

2.1 Introduction

The Double Base Number System (DBNS) is a recently introduced [8], highly redundant number system with a very sparse representation. The DBNS has a number of properties that can be advantageous to DSP applications, such as the possible overhead reduction for a DSP system heavily reliant on multiplications. Also the DBNS, in an index calculus form, has a non-linear number representation that may benefit certain specialized applications, such as those requiring accurate representation of both large and small signals. Recently, the index calculus Double Base Number System has been compared to the Logarithmic Number System, as a multi-dimensional form of that number system [10].

2.2 Representation

The DBNS representation of a number differs slightly from the traditional fixed radix form of representation. Typically a number is represented in the form:

$$x = \sum_{i=0}^{N} m_i r^i \qquad (2.1)$$

where $m \in \{0, 1, \ldots, r-1\}$, i is an integer and r is the radix. For example in the decimal system r=10, and in the binary system r=2.

The DBNS representation varies from this by replacing the radix, r, with two orthogonal bases, restricting the range of the mantissa, m, to represent the sign. The equation for DBNS number representation is:

$$x = \sum_{i=0}^{N} m_i 2^{b_i} 3^{t_i} \qquad (2.2)$$

where b_i, t_i and i are integers. Since m is signed, both positive and negative numbers are representable. The mantissa, or sign component, must also sometimes equal zero. This is required because a DBNS digit, using the exponential representation $2^{b_i} \cdot 3^{t_i}$, cannot express a value of zero.

Equation (2.2) shows the binary number system to be a subset of the DBNS if t_i=0 for all i. Equation (2.2) also leads to a very simple two dimensional representation [15].

Figure 2.1: A 2D DBNS Representation of 79

	2^0	2^1	2^2	2^3	2^4
3^0	■				
3^1		■			
3^2				■	
3^3					

Figure 2.1 shows one of the possible mappings of 79 into DBNS. The black squares represent the location of a DBNS digit. The digits used are $2^0 3^0 + 2^1 3^1 + 2^3 3^2 = 1 + 6 + 72 = 79$. The white squares represent the '0' digits and the black squares represent the digits that are '1'. From Figure 2.1 the very sparse representation of the DBNS number system can be seen.

If negative exponents are used, a much wider range of numbers are available and representation of real numbers, with arbitrary precision, is possible. Representation of integer numbers will require less digits if their representation allows for negative exponents. The two dimensional map of Figure 2.1 can still be used for negative exponents, but it will require a little expanding. Figure 2.2 shows the representation of the real number 7.25 using 2 DBNS digits $2^{-2}3^3 + 2^{-1}3^0 = 6.75 + 0.5 = 7.25$.

Figure 2.2: A 2D DBNS Representation of 7.25

	2^{-5}	2^{-4}	2^{-3}	2^{-2}	2^{-1}	2^{0}	2^{1}	2^{2}	2^{3}	2^{4}
3^{3}				■						
3^{2}										
3^{1}										
3^{0}					■					
3^{-1}										
3^{-2}										
3^{-3}										
3^{-4}										

Only a small number of digits are required to represent a number. For example, the number 103 can represented, approximately, by the two DBNS digits $2^{-24}3^{18} + 2^{19}3^{-8}$ and 2315 can be represented, exactly, by the two digits $2^{0}3^{7} + 2^{7}3^{0}$. Since the Double Base Number System is a redundant number system, most numbers will have more than one representation. For example 4.25 has 3 error free two digit DBNS representations, while 7.25 has a single unique, error free, two digit DBNS representation. It is provable [10] that every real number may have at most 91 error free, two digit DBNS representations, but realistically the majority will have at most 4 or 5 error free representations. This redundancy can be useful in allowing you to choose the best possible representation for your application.

The previous examples all used the form of $\pm 2^{b_1}3^{t_1} \pm 2^{b_2}3^{t_2}$ to first display the DBNS numbers. This is a more convenient and informative way of writing DBNS numbers, as apposed to the two dimensional map representation. In this form the number is represented by the sign and the exponents (also called indices). The exponent on the base of 2 is called the binary exponent, and the exponent on the base of 3 is called the ternary exponent. If we

fix the binary and ternary exponents to a signed, fixed length representation, we can represent any real number to arbitrary precision.

DBNS Two Dimensional Map Arithmetic

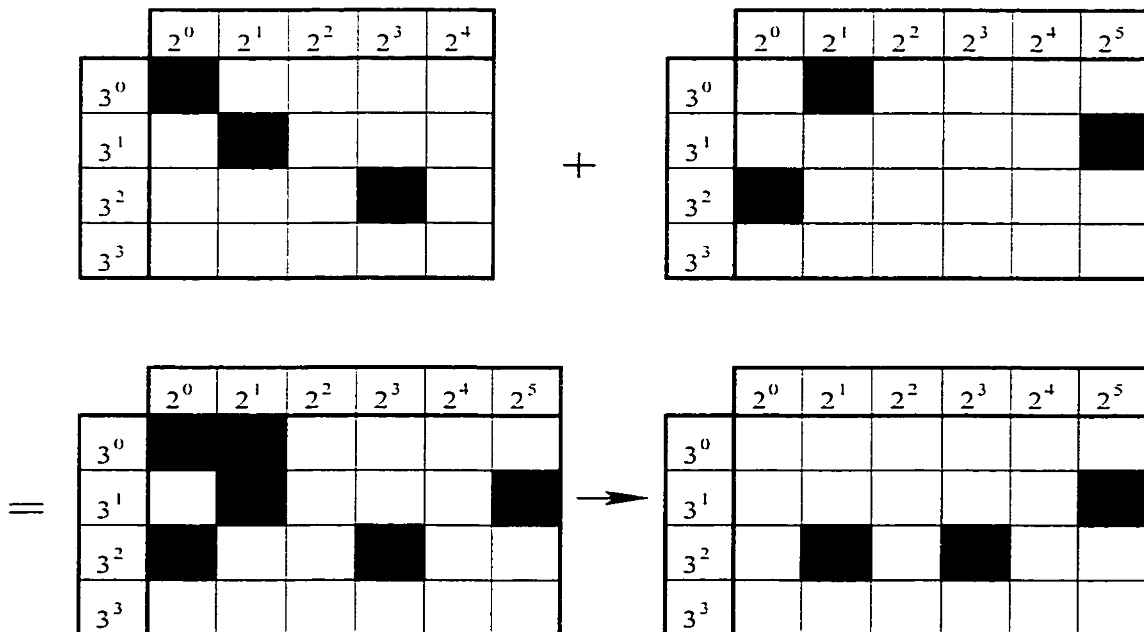
To add any two numbers using the DBNS two dimensional map representation (Figure 2.1 and Figure 2.2), simply overlap the two DBNS maps. Furthermore, a series of reductions of the DBNS map can be performed in order to reduce the number of resulting DBNS digits. These reductions can be summarized in three equations [15]:

$$2^i3^j + 2^{i-1}3^j = 2^i3^{j-1} \tag{2.3}$$

$$2^i3^j + 2^i3^{j-1} = 2^{i-2}3^j \tag{2.4}$$

$$2^i3^j + 2^i3^j = 2^{i-1}3^j \tag{2.5}$$

Example 2.1The addition of the 2D mapped DBNS representations of 79 and 107



The result of this map arithmetic is $2^1 3^2 + 2^3 3^2 + 2^5 3^1 = 18 + 72 + 96 = 186$.

Little work is being done with the two dimensional DBNS map arithmetic as it is difficult to perform in hardware, though Cellular Neural Networks (CNN) have had some success [27]. CNNs are analog circuits that can perform digital computations and produce a digital output. There is currently no way to perform multiplication using the two dimension DBNS map.

2.3 Index Calculus

Double base numbers are usually represented by their binary and ternary exponents, or indices. In order to perform mathematical operations with DBNS numbers using these exponents we need to define these operations. When performing these operations we will represent the DBNS numbers with the triple (s_x, b_x, t_x), where s_x is the sign, b_x and t_x are integers, such that $s_x 2^{b_x} 3^{t_x}$ is a DBNS representation of a number x. Performing mathematical operations using this representation of DBNS numbers is called index calculus, because the numbers are represented by their exponents, or indices.

2.3.1 Multiplication and Division

Multiplication and division are the simplest operations in index calculus DBNS, and offer very little overhead, or complexity. The equations for multiplication and division, given $x=(s_x, b_x, t_x)$ and $y=(s_y, b_y, t_y)$, are [8]:

$$x \cdot y = ((s_x + s_y) mod 2, b_x + b_y, t_x + t_y) \quad (2.6)$$

$$x / y = ((s_x + s_y) mod 2, b_x - b_y, t_x - t_y) \quad (2.7)$$

Equations (2.6) and (2.7) show that DBNS multiplication can be implemented in hardware using simple binary adders. This feature of DBNS can be used to reduce the overhead of a binary system that utilizes many multipliers by replacing them with a DBNS system, since

DBNS multipliers are two binary adders and simple logic for the sign correction. The binary adders should also require less time to evaluate than the corresponding binary multiplier.

2.3.2 Addition and Subtraction

Unfortunately, as with logarithms, addition and subtraction operations are not as simple as muitiplication and division operations. Addition and subtraction must be handled through a set of identities and look-up tables. The identities are [8][9]:

$$\begin{aligned} 2^a 3^b + 2^c 3^d &= 2^a 3^b (1 + 2^{c-a} 3^{d-b}) \\ &\approx 2^a 3^b \Phi(c-a, d-b) \end{aligned} \quad (2.8)$$

$$\begin{aligned} 2^a 3^b - 2^c 3^d &= 2^a 3^b (1 - 2^{c-a} 3^{d-b}) \\ &\approx 2^a 3^b \Psi(c-a, d-b) \end{aligned} \quad (2.9)$$

The operators Φ and Ψ are lookup tables that must be precomputed and store the approximate values of:

$$\Phi(x, y) = 1 + 2^x 3^y \approx 2^\alpha 3^\beta \quad (2.10)$$

$$\Psi(x, y) = 1 - 2^x 3^y \approx 2^\gamma 3^\delta \quad (2.11)$$

The use of large look-up tables, implemented through the use of ROMs, for the evaluation of addition and subtraction operations is the traditional approach in systems such as the Logarithmic Number System [24][25]. This technique is only feasible for very small ranges of DBNS numbers. It is more practical to convert the DBNS numbers to binary and perform the addition and subtraction using binary representation (see Section 2.7.2).

In order to design efficient DBNS hardware, an algorithm should be designed to have all additions and subtractions operations either before or after the multiplication operations, so conversion to and from DBNS will only have to take place one time each. This is easily done in architectures such as FIR filters. The conversions from DBNS to binary will still

require a look-up table, but one that is much smaller than required for handling DBNS addition and subtraction. The look-up table is used to convert the ternary portion of the DBNS number into a binary representation. Therefore the size of the look-up table is dependent on the number of bits used to represent the ternary exponent.

Example 2.2The addition of the index representation of 72 and 48

$$\begin{aligned} & 2^3 3^2 + 2^4 3^1 \\ &= 2^3 3^2 \cdot \Phi(4-3, 1-2) \\ &= 2^3 3^2 \cdot 2^{-4} 3^3 \\ &= 2^{3-4} 3^{2-3} \\ &= 2^{-1} 3^5 \\ &= (121)_{10} \end{aligned}$$

Note that the answer 121 is incorrect, it should be 120. However, this is a close approximation, when performing integer arithmetic. This also shows how unsuited DBNS is for index calculus addition, justifying a conversion to binary before addition.

Example 2.3The multiplication of the index representation of 72 and 48

$$\begin{aligned} & 2^3 3^2 \cdot 2^4 3^1 \\ &= 2^{3-4} 3^{2-1} \\ &= 2^7 3^3 \\ &= (3456)_{10} \end{aligned}$$

Note how much easier multiplication is in index calculus DBNS than addition. The results are as accurate as the numbers being used. This is because a look-up table of approximate values is not required.

2.3.3 Multi-digit DBNS Arithmetic

Multi-digit DBNS arithmetic is simply an extension of the single digit DBNS arithmetic, and is necessary when numbers are represented by more than one DBNS digit (see Equation (2.2)). When performing a computation using multi-digit DBNS each digit can be treated as an independent DBNS number and the operations handled separately. For example, if X and Y are 2 digit DBNS numbers such that $X = x_1 + x_2$ and $Y = y_1 + y_2$ then:

$$X \cdot Y = (x_1 + x_2)(y_1 + y_2) = x_1 y_1 + x_1 y_2 + x_2 y_1 + x_2 y_2 \qquad (2.12)$$

where x_i and y_i are single digit DBNS numbers.

Example 2.4 An example of 2 digit DBNS multiplication of 27 and 23

$$\begin{aligned}
&(2^2 3^2 - 2^0 3^2) \cdot (2^5 3^0 - 2^0 3^2) \\
&= 2^2 3^2 \cdot 2^5 3^0 - 2^2 3^2 \cdot 2^0 3^2 - 2^0 3^2 \cdot 2^5 3^0 + 2^0 3^2 \cdot 2^0 3^2 \\
&= 2^{2+5} 3^{2+0} - 2^{2+0} 3^{2+2} - 2^{0+5} 3^{2+0} + 2^{0+0} 3^{2+2} \\
&= 2^7 3^2 - 2^2 3^4 - 2^5 3^2 + 2^0 3^4 \\
&= (1152)_{10} - (324)_{10} - (288)_{10} + (81)_{10} \\
&= (621)_{10}
\end{aligned}$$

The independence of the arithmetic operations is very important, as it allows for parallel architectures. Furthermore, these parallel structures can be asynchronously clocked, reducing the power requirement of the circuit. This parallel architecture is not restricted solely to multi-digit DBNS, but extends to hybrid DBNS systems. A hybrid DBNS system is one that combines single digit and multi-digit DBNS.

2.4 Asymmetric Exponents

It is not necessary for the exponents or indices to be of the same number of bits. There are advantages to having asymmetric indices. As will be seen in Chapter 4, the area

complexity of a DBNS multiply accumulator is heavily based on the size of the ternary exponent. due the ternary binary conversion ROM necessary for addition. and binary exponent size has very little effect on the area of the cell. Therefore decreasing the ternary exponent is an important factor in minimizing the overall area. It is possible. to a limited extent. to decrease the ternary exponent and increase the binary exponent. while still maintaining similar accuracy in the number representation. Such asymmetric exponents can greatly decrease area while still maintaining high accuracy.

Figure 2.3 shows the magnitude responses of two DBNS mapped FIR filters. The lowpass filter chosen for the example is a typical interpolation filter used for audio and video applications. One filter was mapped into 5 bit exponents for both the binary and ternary exponents. The other filter was mapped into DBNS having 6 bit binary and 4 bit ternary exponents. Note that there is very little difference between these two filter responses. The error. from the designed response. can be seen in Figure 2.4.

Figure 2.3: Magnitude response comparison for asymmetric exponents

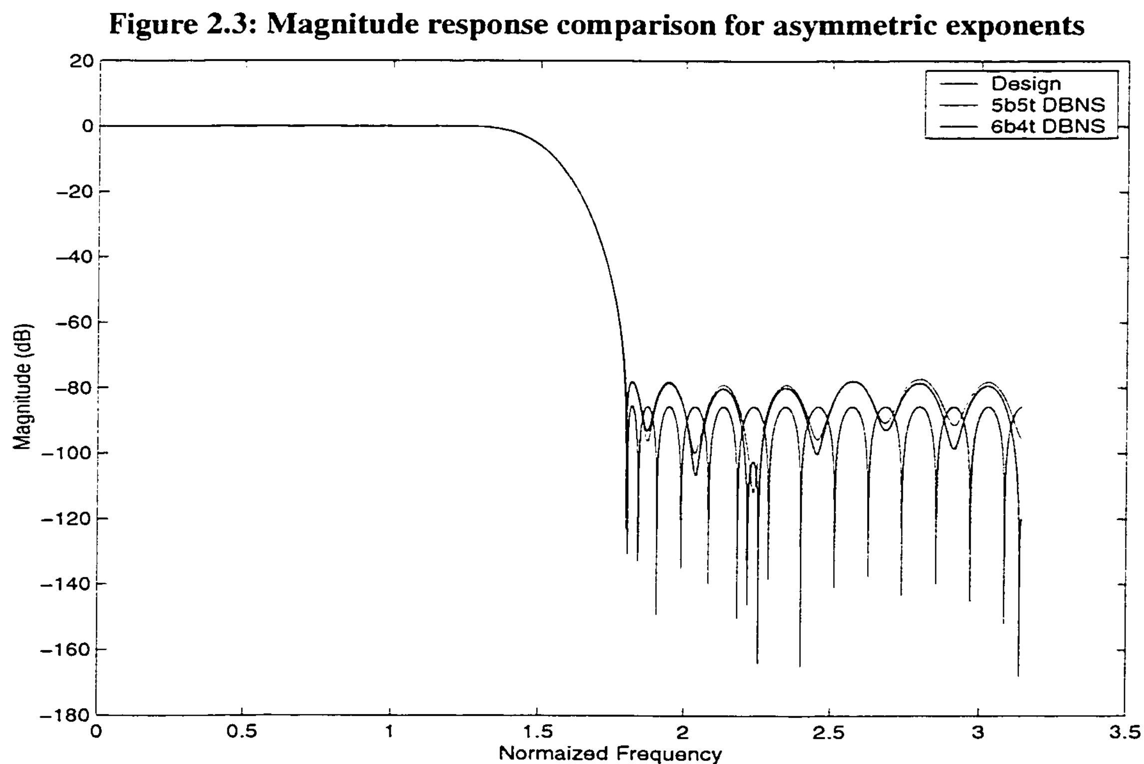
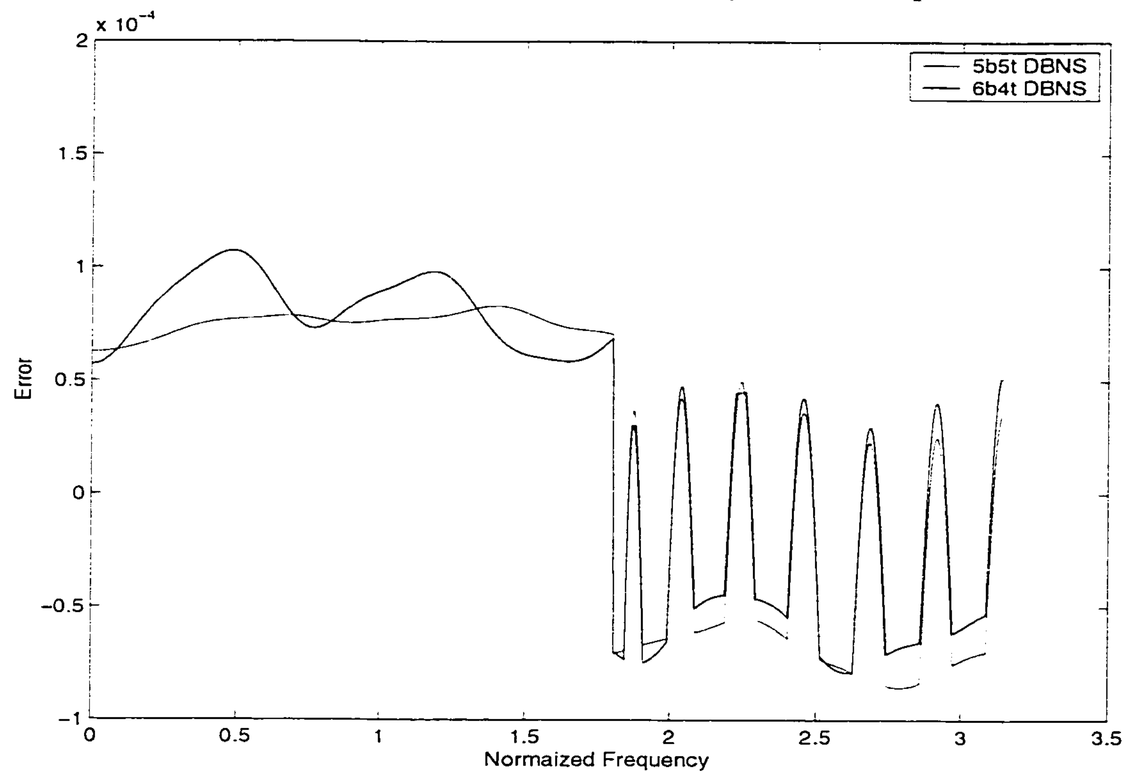


Figure 2.4: Error for symmetric and asymmetric responses



2.5 Logarithmic Number System

The Logarithmic Number System (LNS) is a more mature and documented number system than the DBNS. In the LNS a number, n, is represented by:

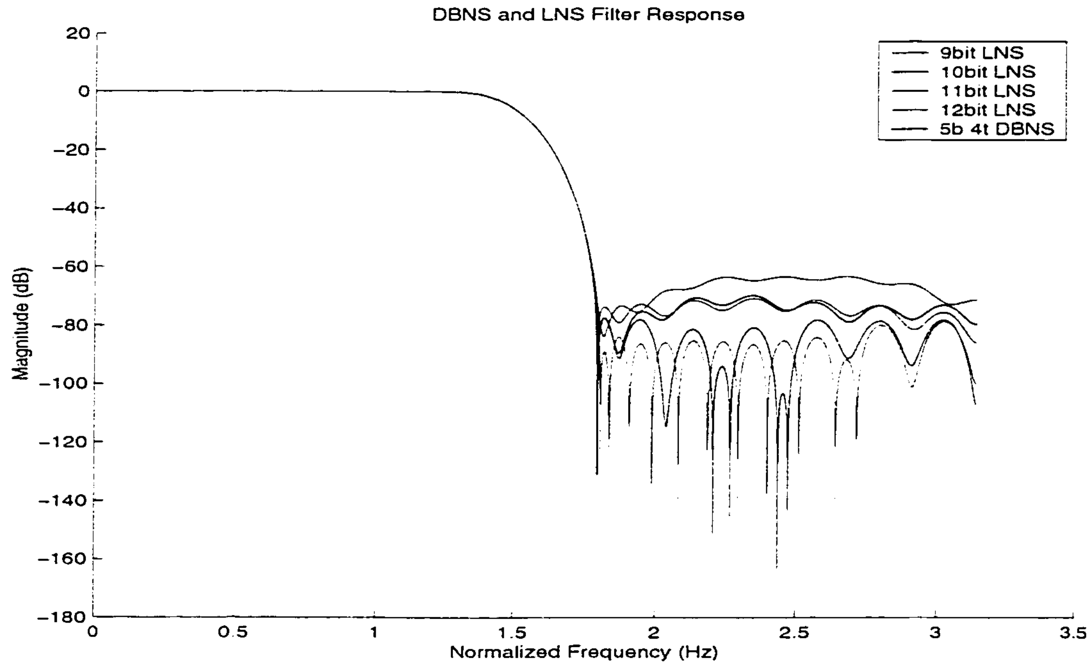
$$n = s_i 2^{a_i} \tag{2.13}$$

where a_i is an arbitrary real number. This representation is very similar to the DBNS number representation. In fact it could be said that DBNS is a multi-dimensional form of the logarithmic number system. Multi-dimensional forms of LNS (like DBNS) offer many advantages over the more traditional single dimensional LNS [10]:

- The area complexity in DBNS systems is based on the size of the ternary exponent. By using asymmetric exponents, the area can be decreased while maintaining accuracy.
- Multi-digit LNS does not offer substantial area complexity reduction. Multi-digit DBNS does offer substantial area complexity reduction.
- Linear data maps to the DBNS better and, more often, has error free representations. This is especially true when using multi-digit DBNS.
- The DBNS is a highly redundant number system, and the LNS is not. A redundant number system has multiple error free representations. While this redundancy is not exploited in the DBNS filters described here, it can be very useful. One "redundancy" that is useful for DBNS filters is the ability to accurately represent a floating point number with DBNS numbers containing widely varying exponents. This will be shown later with channel reduction.
- When performing mathematical operations on DBNS numbers whose exponent values are close to overflow, the numbers can be multiplied by an approximation of 1 to reduce the value of the exponents and prevent overflow.
- The DBNS architecture is extremely flexible in that it allows the odd base (the 2nd base) to be any convenient odd base in accordance with any special requirements. This base change will not affect the architecture, though the values in the look-up tables will have to be changed.
- The DBNS corresponds to a much smaller hardware representation as opposed to classical LNS

Some filter magnitude response comparisons have been made between LNS and DBNS (Figure 2.5). They show that it takes 12bit LNS to exceed the response of a 2 digit, 5bit binary, 4bit ternary, direct mapped DBNS filter, and that 11bit LNS does not have enough accuracy to match the DBNS response.

Figure 2.5: Comparison between LNS and DBNS filter responses



2.6 Different Odd Bases

Typically when using the DBNS the bases used are 2 and 3. The odd base is not restricted to 3, however, and any convenient odd base can be used. Switching the odd base can be done to fulfill a special requirement of the design. For example using a 10 bit exponent for an odd base of 35 has shown good characteristics in mapping hearing aid data. Switching to a new odd base does not introduce any major design changes. DBNS systems use the same hardware and architecture no matter the odd base. Only exponent sizes may change with a change of bases.

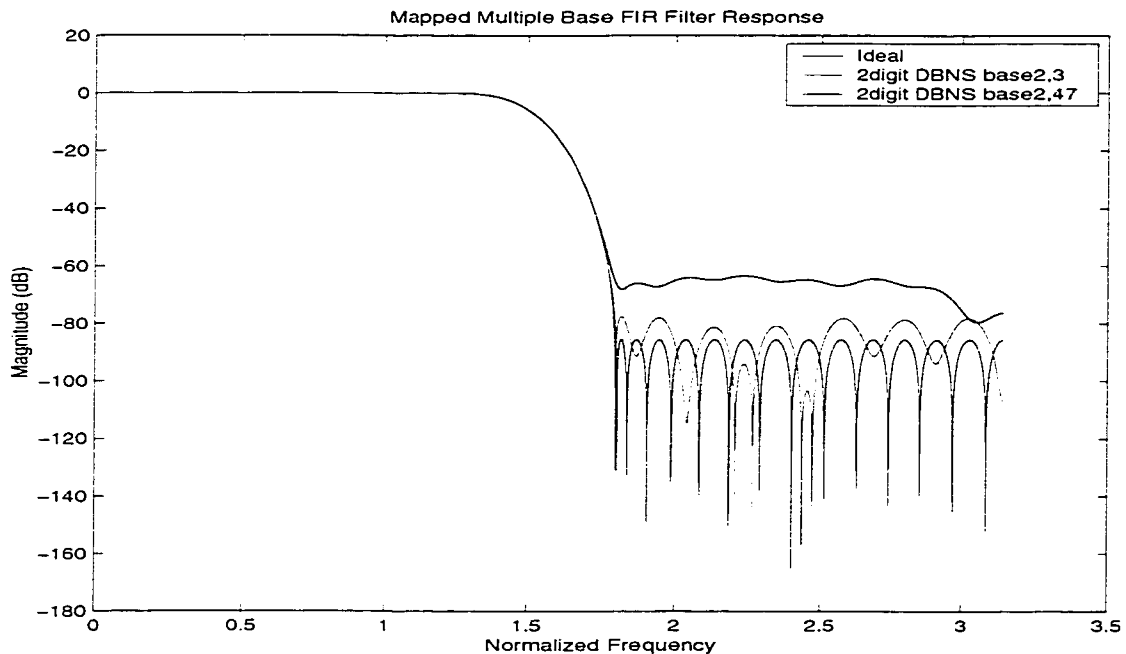
Overall, a base of 3 offers the best results for filter coefficients. Table 2.1 shows the stopband attenuation of DBNS filters using different odd bases. A filter was designed in Matlab using floating point coefficients, and mapped using a modified Greedy Algorithm (see Modified Greedy Mapping in Chapter 3) into DBNS coefficients. The DBNS coefficients are two digit DBNS, with 5bit binary and 4bit odd base exponents. Figure 2.6

shows a graph of the magnitude responses of the ideal, designed, filter and of the two digit DBNS filter response for the filters with an odd base of 3 and 47.

Table 2.1: Stopband Attenuation (in Decibels) for Mapped DBNS Filter Coefficients with Varying Bases

Base	Ideal	3	5	7	11	47
Stopband (dB)	85.5	78	70	64.5	71.3	63.3

Figure 2.6: Magnitude response of DBNS filters with different odd bases



2.7 Conversion Between Binary and DBNS

The majority of systems today rely on the use of the binary number system. In order for any system utilizing the Double Base Number System to be compatible with existing systems, conversions to and from binary will be required.

2.7.1 Binary to DBNS Conversion

There are two methods that can be used to convert from binary representation to DBNS representation. The first method is the most straightforward method. It relies on a ROM to convert a binary input into DBNS exponents and signs. While the ROM look-up table is very easy to produce, the size of the ROM can become overwhelmingly large if the number of bits in the binary representation becomes too high.

The second method of binary to DBNS conversion is the use of a binary to DBNS converter system. This system converts binary numbers into DBNS numbers based on the repetitive patterns in the DBNS number system [26]. This process relies on a number of very small look-up tables and can take many stages. Fortunately these stages can be pipelined to increase the throughput of the conversion systems. There are two pre-requisites before this system can be used. First, the number of bits used for the ternary exponent must be equal to or larger than the number of bits used for the binary exponent. Secondly the DBNS exponents must be of at least 6 bits in length.

2.7.2 DBNS to Binary conversion [9]

Converting DBNS to binary is three step process. The first step is to convert the ternary exponent into a binary manissa and a binary shift value:

$$3^t \rightarrow m \cdot 2^s \tag{2.14}$$

Where t is the ternary exponent, m is the mantissa and s is the shift factor. For example the ternary 3^1 could be translated into $11_b \cdot 2^0$, where the subscript b denotes a binary representation. This conversion is handled through the use of a look-up table ROM.

In the second step the shift value is added to the binary exponent. This will produce an "overall" shift value. This result is used in the third stage as a shift value for the mantissa. Once the mantissa has been shifted, the final result is a binary number that approximates

the DBNS representation. For example, to convert the DBNS representation of 6 into binary would require:

$$\begin{aligned} n &= 2^1 \cdot 3^1 \qquad\qquad (3^1 = 11_b \cdot 2^0) \\ &= 2^1 \cdot 11_b \cdot 2^0 \\ &= 11_b \cdot 2^{1-0} \\ &= 110_b \end{aligned}$$

In this case the converted value exactly matches the DBNS representation. This is not always true, but the approximation can be quite accurate depending on the number of bits used to represent the mantissa.

2.8 Summary

This chapter discussed advantages and flexibility of the Double Base Number system. DBNS is well suited for tasks where overhead, due to multiplications, can be reduced. The difficulties that arise when performing arithmetic can be avoided by converting the DBNS into binary, through the use of a ROM, binary adder and a barrel shifter. The size of the ROM is governed by the number of bits in the ternary exponent. The size of the ternary exponent can be reduced through the use of asymmetric coefficients and by using two digit DBNS. The non-linear nature of the single digit DBNS makes it ideal for mapping non-linear data, such as data for hearing devices. The odd base in DBNS is not restricted to a base of three, but can be changed to any odd base to suit any specialized needs. This change of the odd base does not change the DBNS architecture in any way, only the contents of the any conversion ROMs.

Chapter 3

DBNS Finite Precision FIR Filter Design

3.1 Introduction

Good hardware design is not the only requirement for good filter design. Another important requirement, besides good hardware design, is good coefficient design. The design of coefficients determines the quality of the filter, such as stopband attenuation and passband ripple, as well as the order of the filter. The order of the filter determines the number of hardware stages required by the filter. Poor coefficient design can produce the desired filter response, but at a much higher hardware cost. The technique used to design the filter coefficients may restrict the hardware design, or the hardware design may restrict the choice of coefficient design algorithms.

3.2 Coefficient Design Techniques

There are many different techniques to design filter coefficients. Many of which require the use of special number representations, such as canonic signed digit binary representation. For Finite Impulse Response (FIR) filters the main categories of these design techniques are:

1. Approximation digital filter design [16]: Techniques to approximate digital filters, such as using numerical analysis, Fourier series and windowing functions [19], or the Discrete Fourier Transform for non recursive filter design.
2. Linear programming: Filter design through the use of linear optimization methods. These techniques will produce results closer to the optimal design, but at the cost of taking more time. Includes methods such as Remez Exchange Algorithm [16].
3. Non-linear programming: Filter design through the use of non-linear optimization techniques. This method is used for very complicated systems where it would be difficult to use linear programming. One method used is genetic algorithms [13][23].

The techniques that were investigated for the design of filter coefficients for use with the Double Base Number System (DBNS) filters were genetic algorithms, integer programming and direct mapping. The filter designed through these was a lowpass filter suitable for interpolation of video signals.

3.2.1 Genetic Algorithms

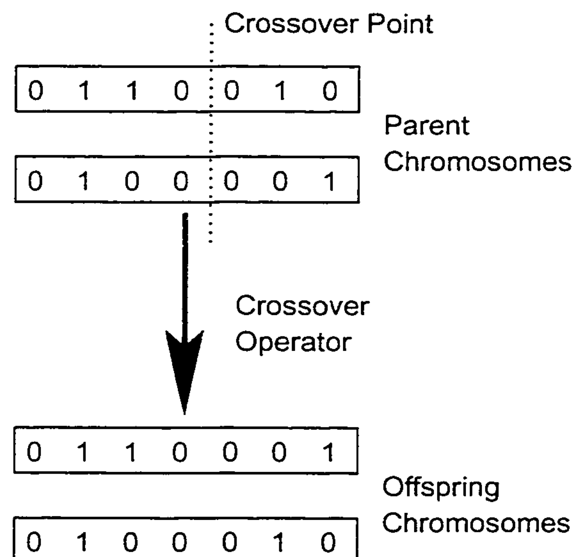
Designing coefficients for a DBNS filter at first appears very difficult. Only linear and non-linear programming techniques can be used because there are no standard methods for DBNS filter design. The DBNS coefficients are to be in index form, meaning the coefficients have 3 parts, representing the sign and the two exponents of the DBNS number. A small change to the value of a coefficient is not a straightforward process. A small change in one of the exponents of the coefficients can mean a very large change in the coefficient's value. This makes a linear programming approach more difficult and a non-linear programming approach much more appealing. Genetic Algorithms (GA) [13][23] were chosen as the preferred non-linear method.

Genetic algorithms are a relatively recent optimization technique, initially disclosed in 1975 [21]. Genetic Algorithms are a non-linear programming technique that use the natural selection properties of evolution to produce a desired result. Multiple random starting points are selected, referred to as chromosomes. Each chromosome is a series of

coefficients that are encoded into sub-chromosomes. These sub-chromosomes are a single bit sequence representing a DBNS digit. The chromosomes are then subject to a set of predetermined fitness tests. These fitness tests require the decoded chromosomes to conform to a set of filter specifications. The chromosomes that produce the best results are then combined through a process called crossover.

Crossover is used to produce the next generation of chromosomes from parent chromosomes of the previous generation. Keeping with the random nature of evolution, crossover selects pairs of parent chromosomes randomly, using the results of the fitness test as a weighting factor. Once a chromosome pair is selected, a pair of offspring chromosomes are produced by switching a random number of contiguous bits between the two parent chromosomes (see Figure 3.1). It can take hundreds of thousands of generations to produce the desired result.

Figure 3.1: Crossover



To keep the population of chromosomes from evolving into a non optimal state, there are two precautions that can be taken. The first is the use of a mutation factor. The mutation factor introduces a random change into the chromosomes. There is a very small chance

that a random bit of a chromosome will be switched. The mutation factor affects the offspring chromosomes immediately after the crossover operation is performed. The second technique is to set conditions where low performing chromosomes will be discarded and replaced with new, random coefficients. The genetic algorithm used for designing DBNS filter coefficients used a mutation factor.

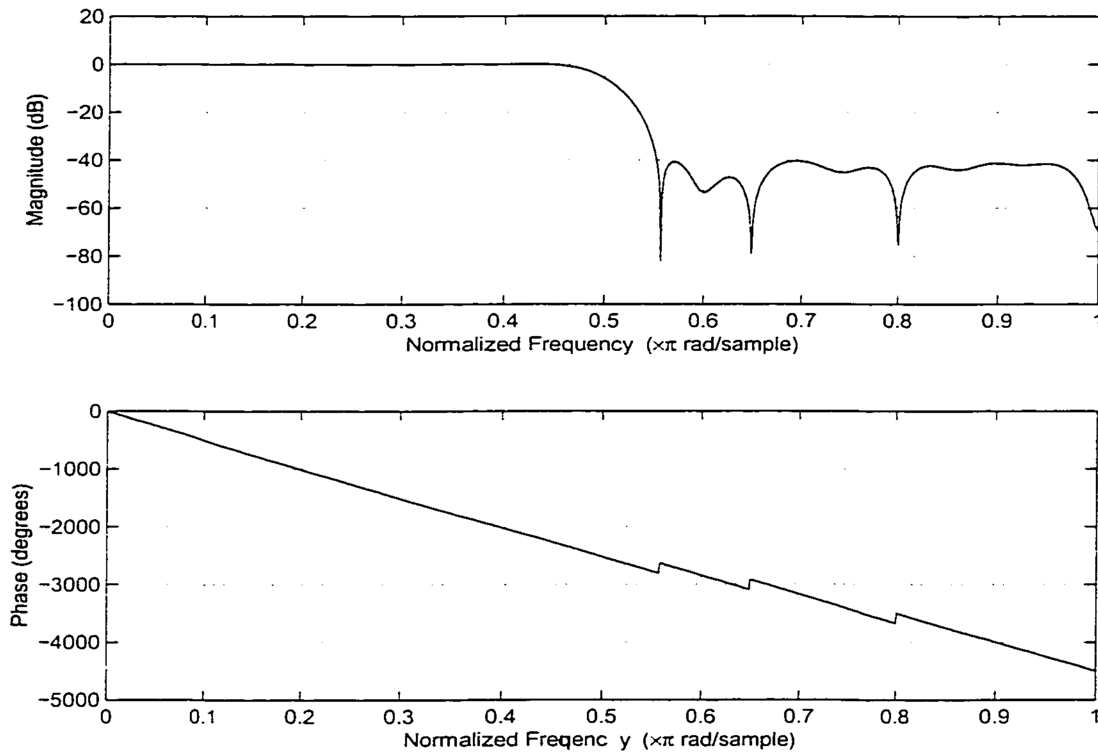
Another technique used to speed up the convergence of the genetic algorithm is multi-point crossover and mutation [22]. Because the chromosomes are of such length and complexity, they can be broken up into segments and have crossover and mutation on these segments. A good design choice would be to have these segments designed into natural groupings. For example, in the DBNS genetic algorithm, these segments are the individual chromosomes, which were each a distinct sub-chromosome. Therefore the crossover and mutation are performed on a coefficient by coefficient basis.

There were some problems with the genetic algorithm used. The first was that the GA is very slow. Secondly, there is no guarantee that the GA will converge, producing the correct answer. Finally, the results produced were not close to optimal. The best filter design achieved had the following statistics:

- 57th order
- 40.35dB stopband attenuation
- 1.75dB pass band ripple
- 0.09 Hz transition band (normalized to 1 Hz)

Figure 3.2 shows the frequency response of the genetic algorithm filter, using a normalized frequency range.

Figure 3.2: Frequency response of a genetic algorithm produced DBNS FIR filter



3.2.2 Integer Programming

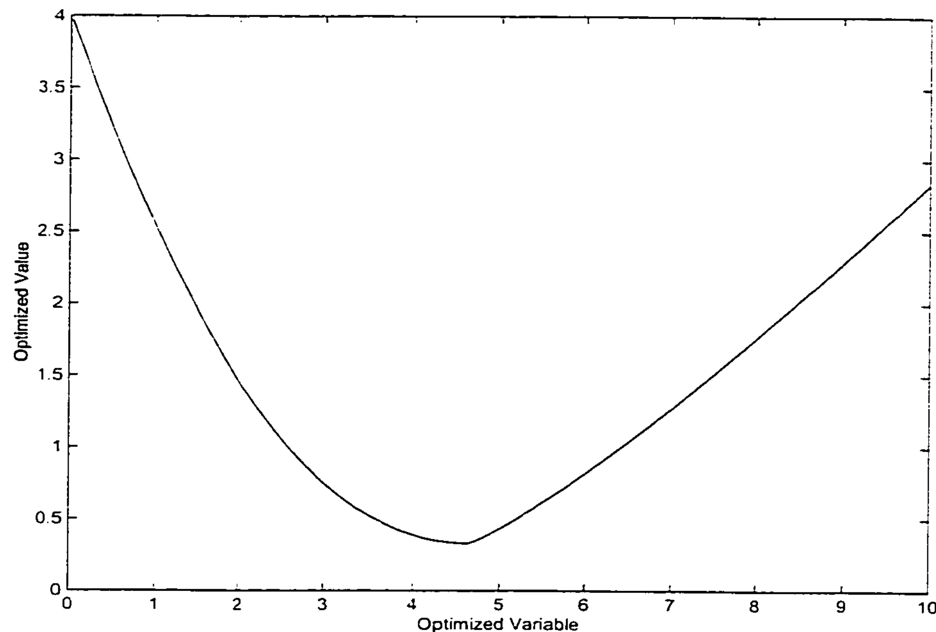
Finding the best DBNS approximation of a designed filter's coefficients is an integer programming task. This is due to the fact that the DBNS exponents, both binary and ternary, are integers. In general, integer programming is a NP-complete problem. That is the reason why genetic algorithms were tried first for generating the coefficients. There are, however, some purely integer programming methods that can be used as well.

The integer programming method used was to map floating point decimal numbers into DBNS. There are four variables to optimize for each number, one for each DBNS exponent.

$$n = \pm 2^a 3^b \pm 2^c 3^d \quad (3.1)$$

To simplify the solution, the first three variables, a, b and c, are set using an exhaustive search, and only the fourth variable, d, is optimized. An important concern, when dealing with fractional results, is to ensure the more optimal choice is selected.

Figure 3.3: Result curve due to optimization

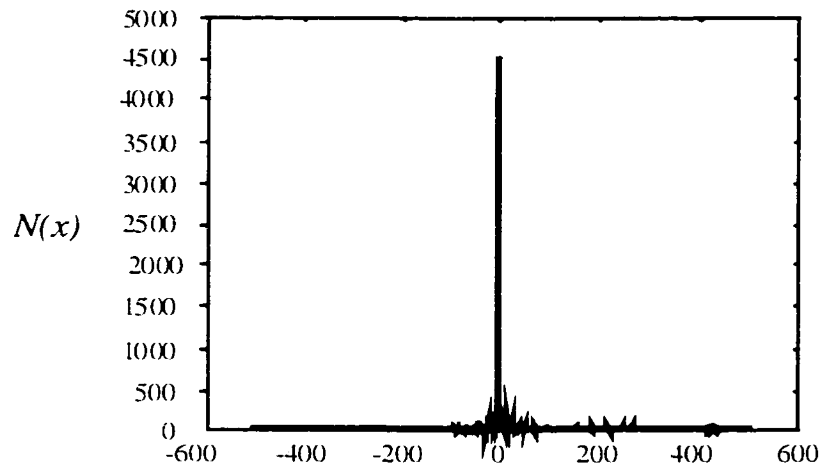


For example if the result curve due to optimization is similar to Figure 3.3, then rounding a fractional result would produce the result of 5, and not the most optimal result of 4. The results of the integer programming were very similar to that of the direct mapping scheme described next, though the required computational time was much higher.

3.2.3 Direct mapping

A study of filter coefficients can be performed to show the typical number range of filter coefficients for a large set of designs. A histogram showing a study of the coefficient distribution of over 200 filters is shown in Figure 3.4 [14]

Figure 3.4: A histogram showing the coefficient values for over 200 filters



Looking at Figure 3.4, it can be seen that the vast majority of the coefficients are zero or very close to zero. This corresponds very well with the DBNS representation accuracy.

In the DBNS, approximately one half of all representable numbers lie in the range between zero and one. This means that if a number is less than one it can be very accurately represented in DBNS. This accurate representation of numbers less than one allows for a direct mapping of these numbers into DBNS.

Filter coefficients can be designed using standard filter design tools, for example MatLab. Filter design functions in MatLab, such as *remez()*, can be used to design filter coefficients in the internal floating point representation. The floating point coefficients can then be converted directly into a DBNS representation. The conversion, or mapping, can be performed in two different ways. The first method is the modified Greedy Mapping algorithm, and the second is optimal mapping using a form of linear programming (in our case an exhaustive search was conducted).

Modified Greedy Mapping

The Greedy Algorithm was first proposed in [15]. For a single digit, the DBNS number closest to the target number is selected. Multi-digit DBNS works in a similar way, the first digit is selected to be as close to the target number as possible. The following digits are selected to be as close as possible to the remaining value after subtracting the values of the previous DBNS digits.

The modified greedy mapping algorithm is very similar to the Greedy Algorithm. It was designed for use with one and two digit DBNS mapping. For one digit it behaves exactly as the Greedy Algorithm. For two digits, it selects the two closest numbers to the target value, then fine tunes the mapping with a second digit. The best result of the two possible is then selected. This produces a more accurate mapping than the Greedy Algorithm alone.

One advantage of the modified greedy mapping method over the optimal mapping is speed. The modified greedy mapping time increases linearly with an increased representation range (i.e. more bits per exponent). Another possible advantage to the modified greedy mapping will be discussed in Chapter 4.

Optimal Mapping

Optimal mapping is performed just as the name implies. The entire representation range is searched for the best possible match to the target value. This method becomes more efficient if a sorted list is used. The mapping time increases approximately along a square law with increased representation range. Despite the square law increase in speed, the mapping of fifty-three, 2 digit DBNS numbers, using 12bit exponents requires less than a minute on a Sun Ultra 10 workstation.

An Example of Mapping

A filter was designed using MatLab's *remez()* function. The filter designed had the following specifications:

- 53rd order
- 85.36 dB stopband attenuation
- 0.000468 dB pass band ripple
- 0.12 Hz transition band (normalized to 1 Hz)

Figure 3.5: Filter response of a floating point coefficient filter

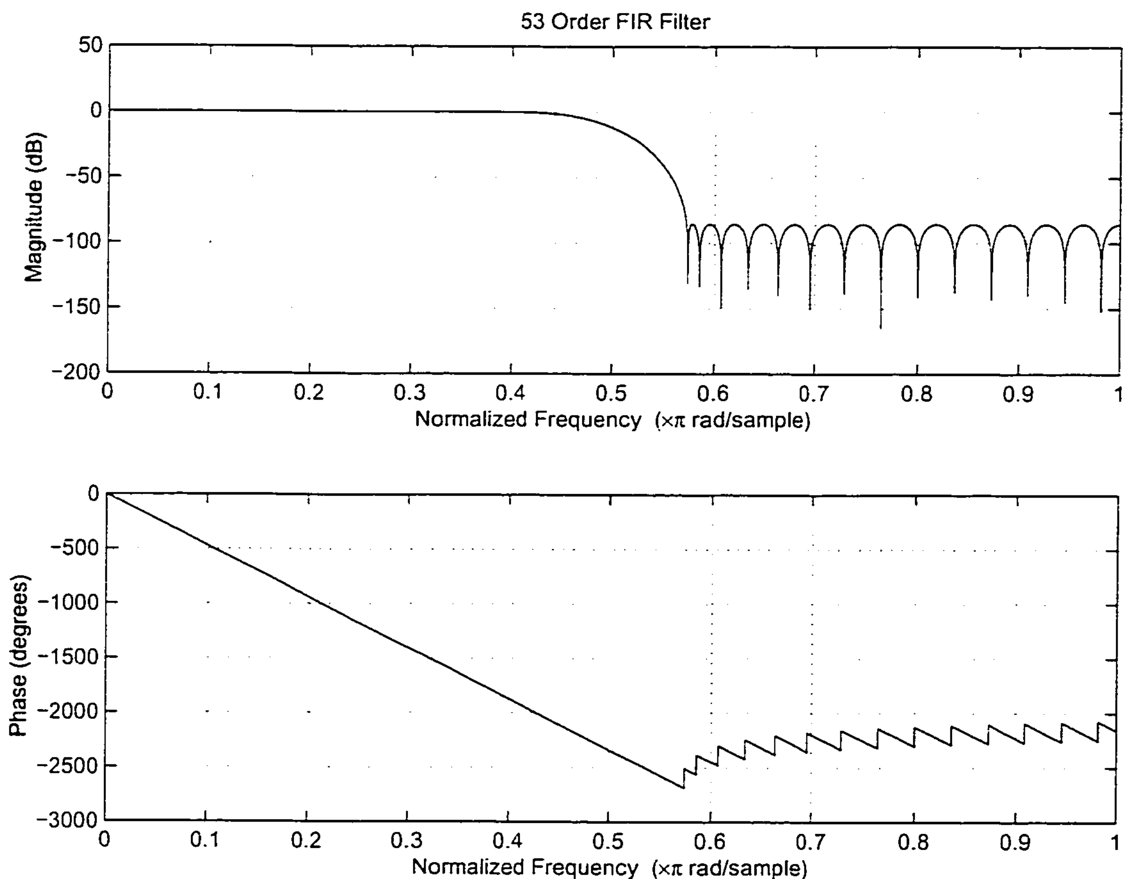


Figure 3.5 shows the frequency response of this filter. The filter coefficients are all in floating point representation. This gives the filter coefficients an accuracy hard to achieve in hardware.

The designed filter is mapped using both the optimal mapping and the modified greedy mapping. The DBNS representation used for these mappings is two digit, with 5 bits being used for the binary exponent and 4 bits used for the ternary exponent. These bit widths

include the sign bit for the exponents. The resulting filters had the following specifications as shown in Table 3.1.

Table 3.1: Mapped Filter Specifications

	Designed	Optimal	M. Greedy
Order	53	53	53
Passband Ripple (dB)	0.000468	0.001339	0.001341
Stopband Attenuation (dB)	85.36	77.98	77.52
Transitionband Width (normalized)	0.12	0.12	0.12

The filters' magnitude response is shown in Figure 3.6, and the error between the design response and the mapped response is shown in Figure 3.7.

As can be seen from both the table and the figures, there is very little error between the designed and mapped filters. The optimal mapping method does not offer a significant accuracy advantage over the modified Greedy Algorithm. The entire design, from generating the designed filter, to producing the output graphs and statistics took less than a minute on a Sun Ultra 10 workstation. This is very fast and produces very good filters, especially when compared to the genetic algorithm approach.

Figure 3.6: Magnitude response for the mapped filters

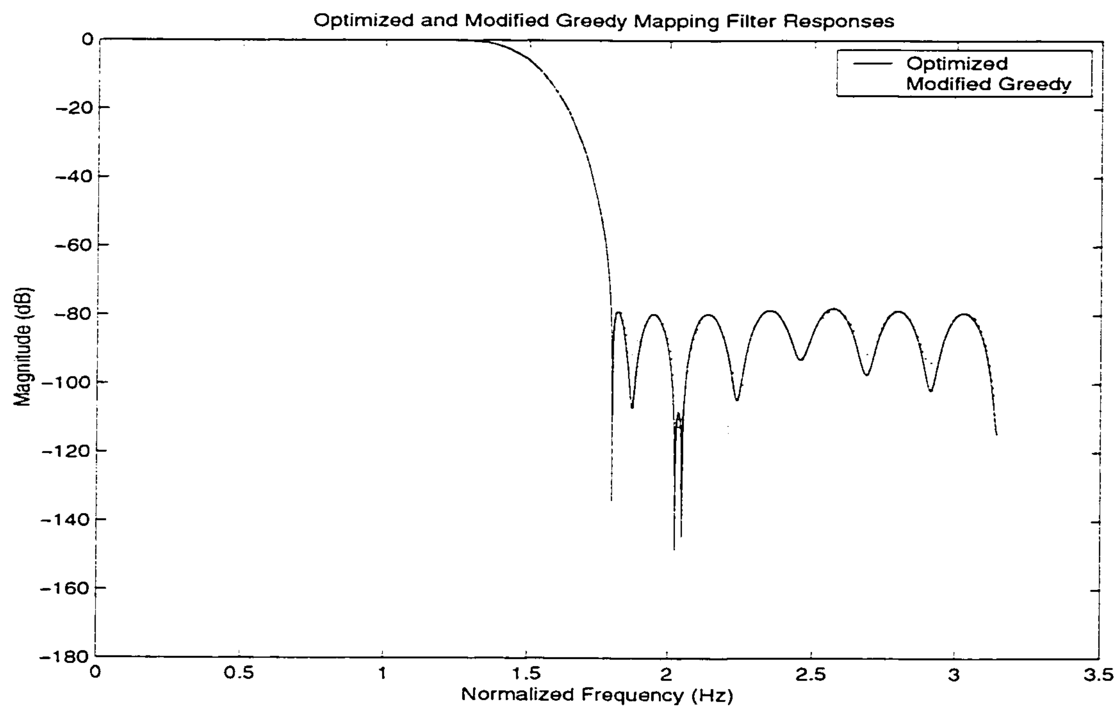
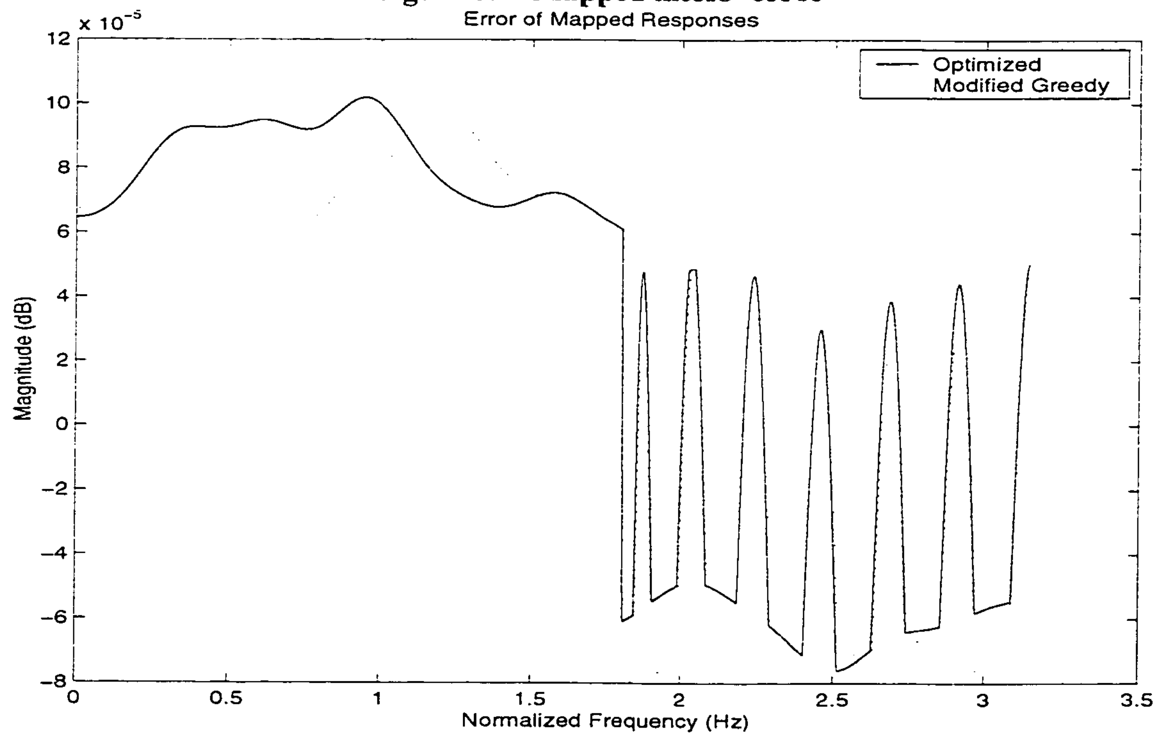


Figure 3.7: Mapped filters' error



Comparing DBNS and Binary Direct Mapping

Direct mapping of DBNS can be accurate. as seen in the last section. Consider the direct mapping scheme to binary and DBNS of one and two digits. For the comparison. the *remez()* FIR filter from the previous section (Figure 3.5) was used as the base filter to be mapped. For the DBNS mapping 9 bit binary and ternary exponents were used for the single digit DBNS. and 5 bit binary. 4 bit ternary were used for the two digit DBNS. All the DBNS mappings used the modified Greedy Algorithm. The number of binary bits used for the mapping was 10 and 12 bit.

The error between the mapped DBNS responses and the designed magnitude response can be seen in Figure 3.8. The two digit DBNS mapping is much more accurate. by an order of magnitude. than the single digit DBNS mapping.

Figure 3.8: Error of mapped DBNS responses

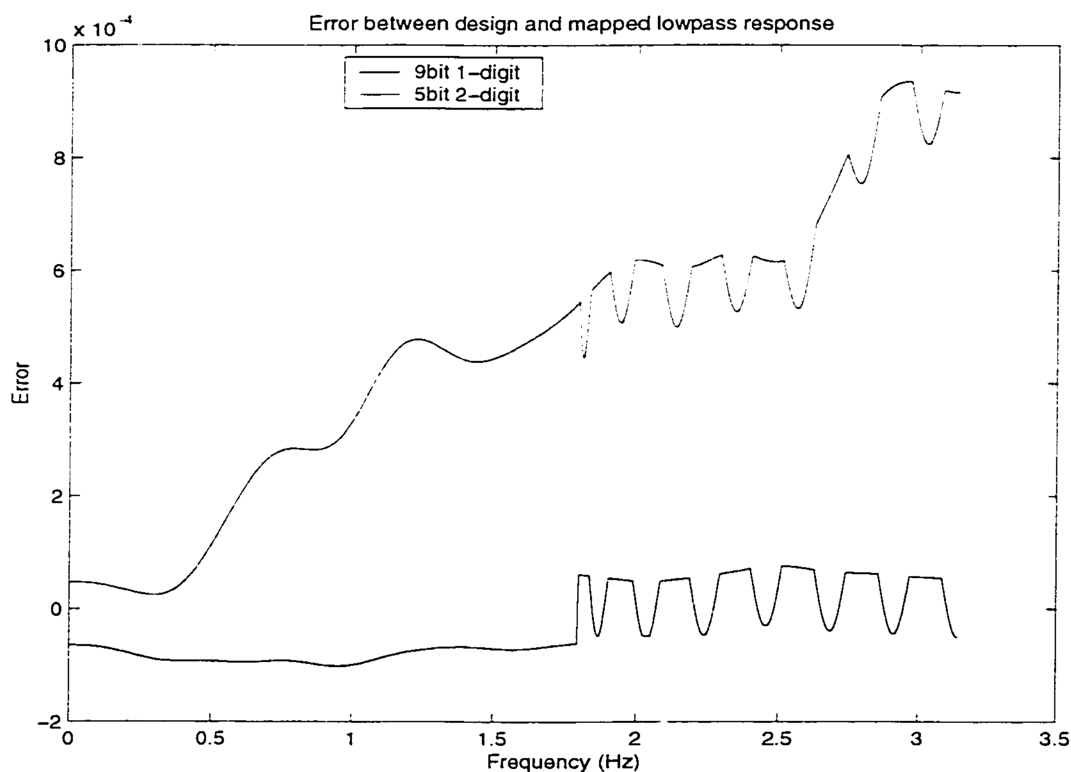


Figure 3.9 shows the error between the designed magnitude response and the magnitude response of both the binary mappings and both the DBNS mappings. The stopband attenuation achieved by these mappings are as follows: 10 bit binary 45dB, 12 bit binary 55dB, single digit DBNS 60dB and 2 digit DBNS is 77dB. It can be clearly seen that the DBNS mappings offer a more accurate mapping, particularly in the case of the two digit mapping.

Figure 3.9: Error between designed response and binary and DBNS mappings

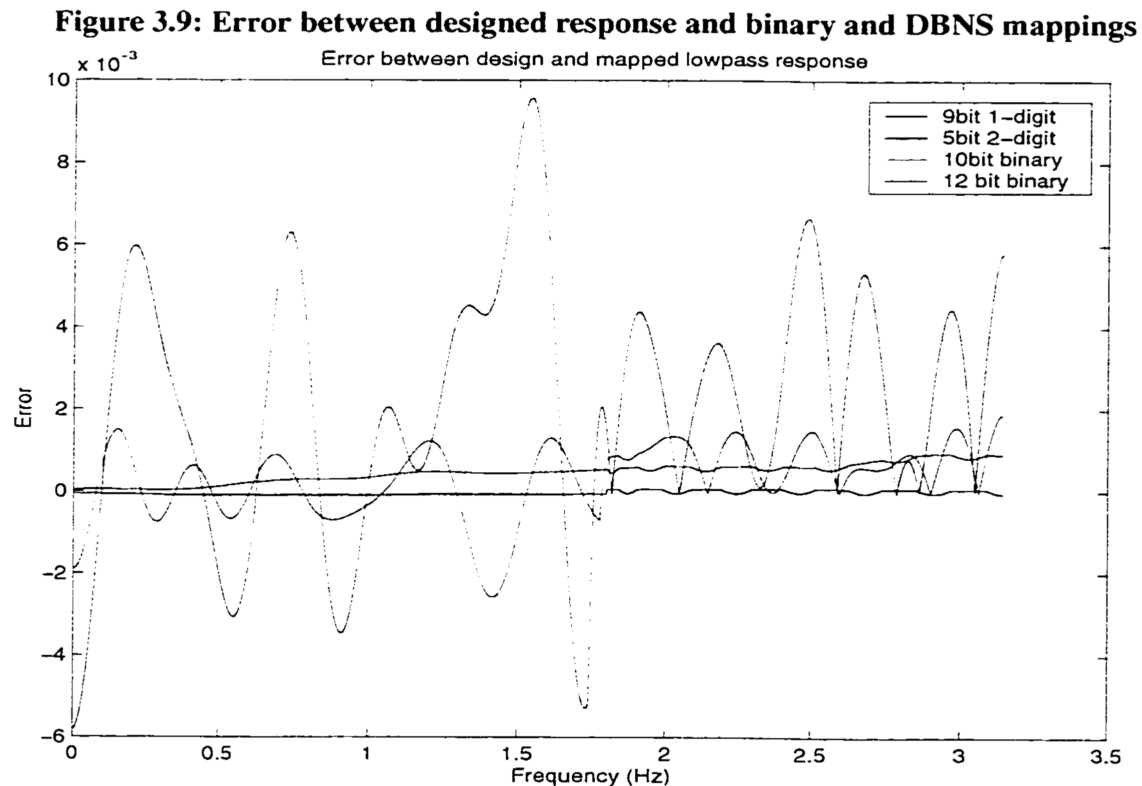
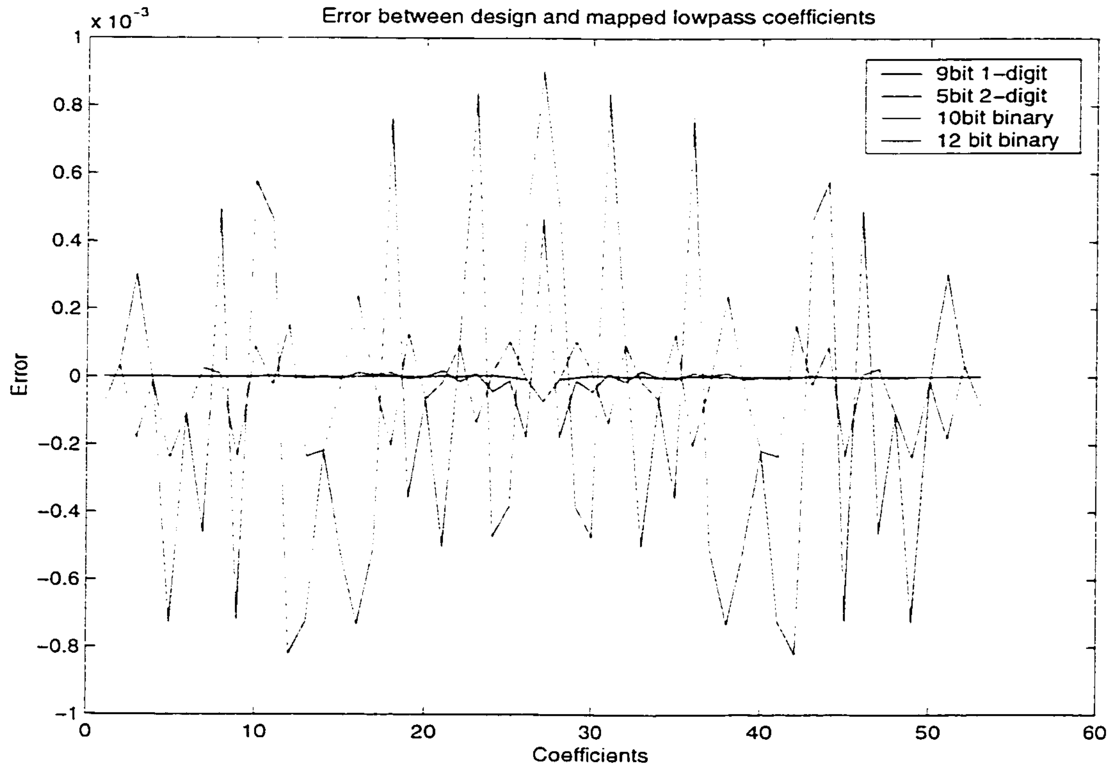


Figure 3.10 shows the error between the designed coefficient values and the mapped coefficients. The binary design has varying errors across all the filter's coefficients. The largest error in the DBNS mappings is focused at the central coefficients, and rapidly drops off for the outer coefficients. As typical in a lowpass FIR filter, the central coefficients have the largest value and the value of the coefficients decreases towards the outer coefficients [16]. The smaller the number, the more accurately the DBNS can represent it, thus the reason there is more error in the central coefficients. The larger

central coefficients have more weight in determining the output. therefore if their error could be decreased it would greatly decrease the error in the mapped response.

Figure 3.10: Error between designed coefficient values and mapped coefficients



3.3 Summary

This chapter covered different techniques to generate coefficients for DBNS FIR filters. While it is possible to use complex and time consuming techniques, such as genetic algorithms, it is not necessary. The results produced by the use of direct mapping are better quality filters (i.e. better stopband attenuation) than that of the genetic algorithm, and are achieved much more quickly than the genetic algorithm or other iterative optimization techniques. By using standard, well-defined filter design tools, and mapping the results into DBNS we achieve better results in a fraction of the time.

Chapter 4

DBNS FIR Filter Architectures

4.1 Introduction

One of the basic structures in DSP is the FIR filter. FIR filters are used to modify and transform discrete data in a number of applications. FIR filters exist in many devices, from CD players to cell phones to radios. FIR filters can be used in traditional filtering tasks, or for weighted or moving average calculations. The FIR filters described here are primarily designed for video rate applications (approximately 40MHz), though they can be used in any general application. This chapter will discuss two architectures used for Double Base Number System (DBNS) FIR filters. These architectures are the hybrid 2 digit DBNS and the 2 digit DBNS filters.

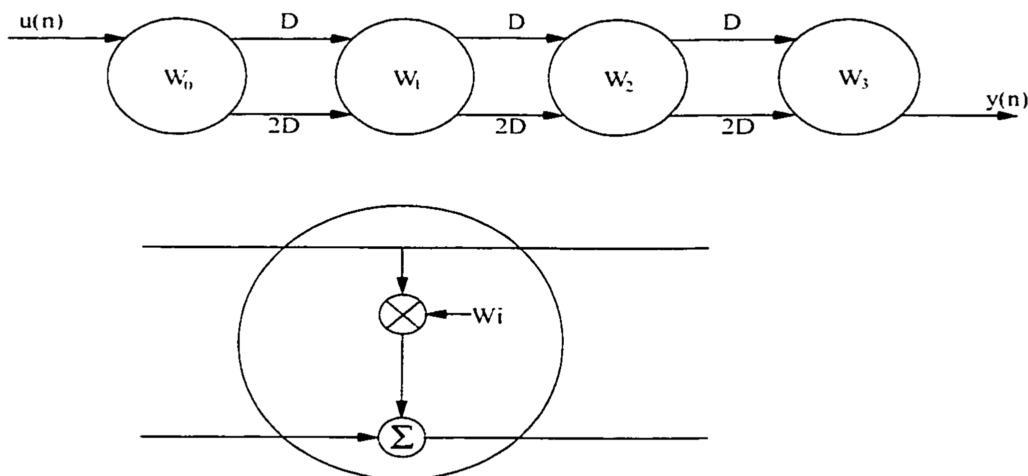
4.2 Systolic Arrays

Filter architectures require many additions and multiplications. Each multiply block represents a filter coefficient, contributing to the order of the filter. Fifteen multiplication blocks represent a fifteenth order filter. In order to arrange the multipliers and adders into a parallel architecture, where there are a minimum number of binary two operand adders, we need to use systolic arrays.

Systolic arrays are a class of pipelined array architectures, and feature the properties of modularity, local connectivity, regularity, high level of pipelining, and highly synchronized multiprocessing [11]. Systolic arrays are useful architectures for systems that have multiple operations performed on each data element in a repetitive manner. This gives the system a highly parallel processing architecture. These repeated operational blocks in the systolic array are called processing elements.

Filtering of data is performed in the time domain using convolution. A unidirectional systolic array convolver was used as the base filter architecture in all DBNS filters described herein. Figure 4.1 [11] shows the typical unidirectional systolic array convolver. The circles in the figure represent a multiply accumulate processing element.

Figure 4.1: A unidirectional systolic array convolver and processing element



The top portion of Figure 4.1 is a signal flow graph (SFG). In a SFG a "D" represents a delay element, and the processing elements (the circles) are considered to have no delay. Realistically, the processing element hardware would contain delay, and care must be taken to ensure the signals arrive at the correct time. This can be handled through the addition of delay elements, such as latches.

Example 4.1

If $u(n)$ were to consist of the elements $\{u_0, u_1, u_2, u_3\}$, the data passing through the convolver would produce the following table. The columns represent the output of the given processing element. The output of the final processing element is the output of the filter and is therefore labelled as such.

Table 4.1: Unidirectional systolic convolver results

Time	PE 0	PE 1	PE 2	Output
0	u_0w_0			
1	u_1w_0	u_0w_1		
2	u_2w_0	$u_0w_0+u_1w_1$	u_0w_2	
3	u_3w_0	$u_1w_0+u_2w_1$	$u_0w_1+u_1w_2$	u_0w_3
4		$u_2w_0+u_3w_1$	$u_0w_0+u_1w_1+u_2w_2$	$u_0w_2+u_1w_3$
5		u_3w_0	$u_1w_0+u_2w_1+u_3w_2$	$u_0w_1+u_1w_2+u_2w_3$
6			$u_2w_0+u_3w_1$	$u_0w_0+u_1w_1+u_2w_2+u_3w_3$
7			u_3w_0	$u_1w_0+u_2w_1+u_3w_2$
8				$u_2w_0+u_3w_1$
9				u_3w_0

4.2.1 Pipelining

In hardware design, a pipeline is a structure that breaks up computational tasks into smaller independent computations that can be cascaded together and computed concurrently. The importance of the pipeline is that the input data only waits for a single stage to compute, rather than the entire task.

The pipelinability of a systolic array is an important feature. Through pipelining the data can proceed through the sub-tasks without having to wait for the entire computation to be completed. Hence by breaking a computational task into multiple independent sub-tasks the throughput of the system is increased.

An example of pipelining is the systolic array convolver discussed above. Instead of waiting for the data to pass through the convolver individually, each processing element acts independently and concurrently. Real world hardware does not perform without a propagation delay, as assumed by processing elements in the SFG. Thus it makes sense to pipeline the processing elements. The processing elements can be pipelined by breaking multiply accumulate operation into an independent multiplication and addition. This has the potential effect of doubling the clock frequency, and the throughput.

4.3 Basic Processing Element

The systolic array convolver is made up of a network of identical processing elements. Each of these elements performs an inner product calculation, which is performed through a multiply accumulate (MAC) operation. That is, the processing element performs the following calculation:

$$p_i = p_{i-1} + u(n) \cdot w_i \tag{4.1}$$

where p_i is the output of the ith processing element, i is the index of the element, and n is the index of the input data being operated on. When chained together to form a systolic array filter, the output equation becomes:

$$y(n) = \sum_{i=0}^{N} u(n) \cdot w_i \tag{4.2}$$

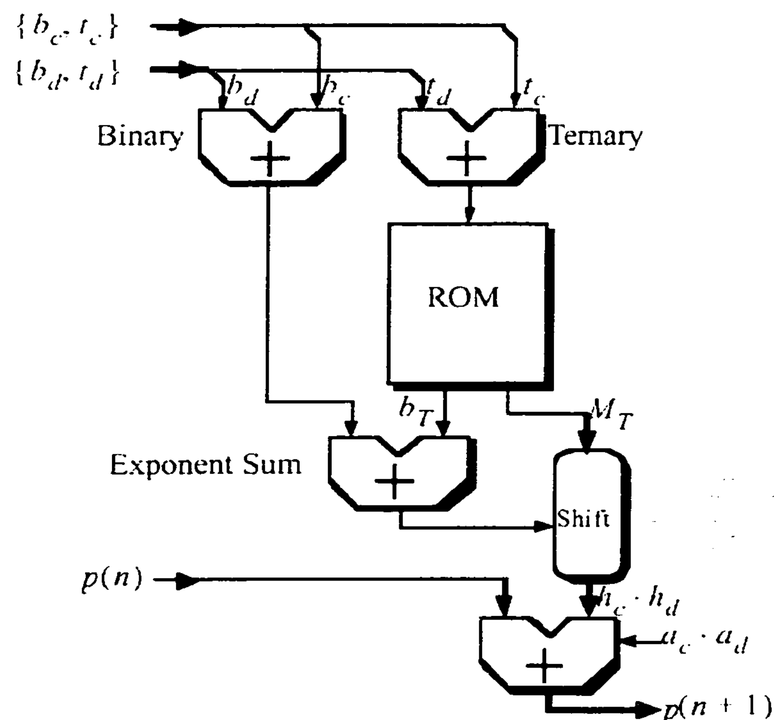
where N is the order of the filter.

The DBNS equivalent to a MAC cell is shown in Figure 4.2 [9]. Because the DBNS is represented in hardware by the triple (s_i, b_i, t_i), there are three pieces of data on which to be operated. The binary and ternary exponents of the data (b_d and t_d) are added to the exponents of the coefficients (b_c and t_c) in order to perform the multiplication portion of the MAC calculation. The two adders required for this multiplication are in the top part of

Figure 4.2. The sign bits are handled by simple logic and are used for sign correction before the final accumulation.

The accumulation stage cannot be performed directly after the multiplication, due to the difficulty in adding two DBNS numbers. It is easier to convert the DBNS value to binary for the accumulation. This conversion is handled by the ROM, shifter and adder in the middle of the figure. The ROM contains the data for converting the ternary portion of a DBNS digit into a binary value. This data is a fixed point binary mantissa and a shift value.

Figure 4.2: A DBNS Multiply Accumulate Cell



The binary portion of the DBNS digit also corresponds to a shift. Therefore the binary exponent and the shift data from the ROM are added to find the final shift value for the mantissa data from the ROM. The ROM increases size along a square law based on the number of bits required to represent the ternary exponent. Therefore it is important to keep the ternary exponent as small as possible.

Once the DBNS to binary conversion has taken place, the accumulation can be performed. The accumulation sums the previous multiplication with all the values generated by the preceding MAC cells.

4.4 Hybrid 2 digit DBNS Architecture

Architecture describes the structure of a system. It consists of the sub-blocks of the system and the interconnects or buses between each of these sub-blocks. The architecture of a system can be hierarchical. For example, the DBNS MAC can be considered a single sub-block of a system, even though it has its own architecture.

The first architecture investigated was the hybrid 2 digit DBNS architecture. This system uses two digit DBNS for the data and single digit DBNS for the coefficients. This architecture was first introduced in [9]. Two digits were used for the data to produce a pseudo-linear mapping space. This reduced the error introduced by the conversion to DBNS. Approximately one half of all numbers representable by a DBNS digit are less than 1, and the majority of all filter coefficients are also below 1 [9]. This allows for a large number of coefficient values that can be represented by a single digit.

The FIR filter architecture is based on the systolic array convolver (Figure 4.1) discussed in Section 4.2. For the multiply accumulate processing elements of the systolic array, the DBNS multiply accumulate structure from Section 4.3 can be used. This systolic array convolver forms a single filtering channel. A single channel can be used to filter single digit DBNS data with single digit DBNS coefficients.

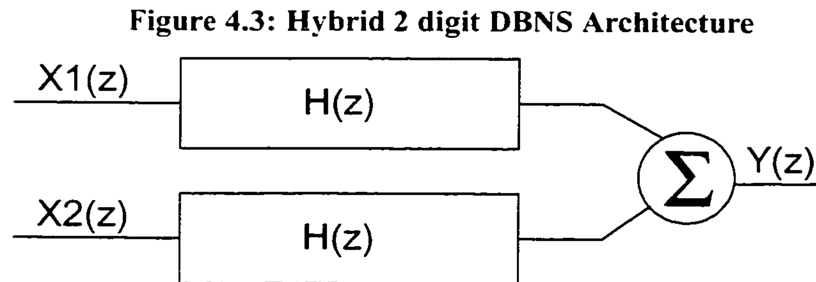
If multi-digit DBNS is to be used for a filter, more than a single filter channel must be used. Convolution in the time domain corresponds to multiplication in the frequency domain [16]. For 2 digit hybrid DBNS the frequency domain multiplication is:

$$Y(z) = H(z) \cdot X(z) = H(z) \cdot (X_1(z) + X_2(z)) = H(z) \cdot X_1(z) + H(z) \cdot X_2(z) \quad (4.3)$$

where $Y(z)$ is the output of the filter, and $X(z)$ is the 2 digit DBNS input into the filter. $H(z)$ is the transfer function of the filter, and represents a filter channel. $X_1(z)$ and $X_2(z)$ are the individual digits of the 2digit DBNS input $X(z)$.

Looking at Equation (4.3) we notice that the individual digits of the input, $X_1(z)$ and $X_2(z)$, are filtered (multiplied by $H(z)$) independently from each other. This means that each digit can be processed separately. Therefore, in order to move to multi-digit DBNS, parallel filtering channels can be used. To determine the final output of the filter, we sum the outputs of the filter channels. Figure 4.3 shows the hybrid 2 digit DBNS architecture based on Equation (4.3). The H(z) blocks represent the independent filtering channels.

Figure 4.3: Hybrid 2 digit DBNS Architecture



4.5 2 Digit DBNS Architecture

The major problem with using the DBNS filter architecture described above is the size of the DBNS to binary conversion ROM. As the size required for the coefficients increases, the size of the ROM grows according to a square law, quickly becoming the dominant feature of a MAC cell, and requiring excess space. Therefore any reduction in the size of the ternary exponent, the driving factor on the size of the ROM, is an important design consideration.

To this end a full two digit architecture was developed [17]. This architecture uses two digit input data, as in the hybrid DBNS filter, but two digits to represent the coefficients.

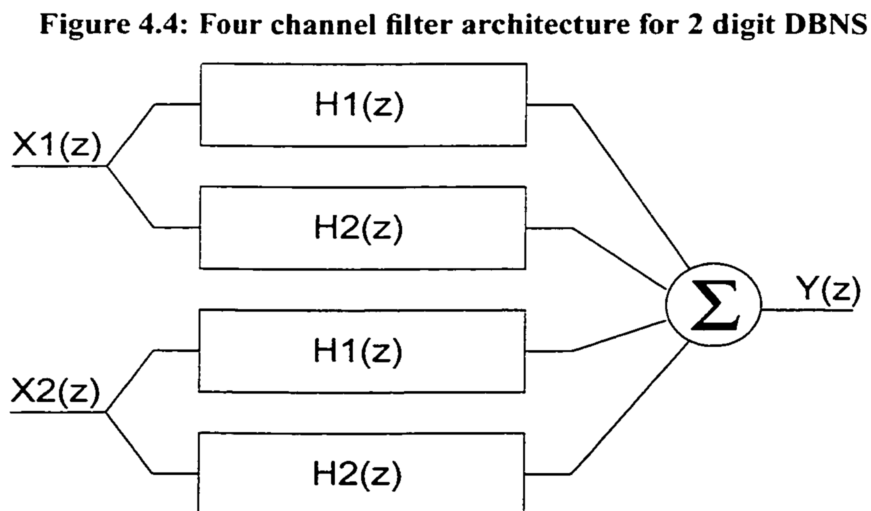
The idea behind using two digits for the coefficients is that a two digit DBNS representation requires, at most, one half the number of bits per digit as a single digit DBNS number. By halving the number of bits used for the ternary exponent, the conversion ROM is drastically reduced in size.

The data $X(z) = X_1(z) + X_2(z)$ is filtered through multiplication in the frequency domain by the filter's coefficients $H(z) = H_1(z) + H_2(z)$. The produced output is therefore described as:

$$\begin{aligned} Y(z) &= X(z) \cdot H(z) \\ &= (X_1(z) + X_2(z)) \cdot (H_1(z) + H_2(z)) \\ &= X_1(z)H_1(z) + X_1(z)H_2(z) + X_2(z)H_1(z) + X_2(z)H_2(z) \end{aligned} \tag{4.4}$$

Each multiplication in Equation (4.4) corresponds to a filtering operation with transfer function of $H_i(z)$. The additions make each of these filtering operations independent, whose final outputs are summed to produce the overall filtered output. These four independent filtering operations can be performed in parallel using a four channel architecture, as shown in Figure 4.4.

Figure 4.4: Four channel filter architecture for 2 digit DBNS



The increase in hardware area by adding the additional channels is a linear increase, but the area reduction of the ROM due to decrease of the ternary exponent bit length follows a square law. This means that for small exponent systems, the use of a hybrid DBNS system reduces the area. For systems that have exponents of nine or more bits using the hybrid DBNS scheme, switching to the two digit architecture would offer superior area optimization, due to the large decrease of ROM area.

4.5.1 Two Digit Channel Reduction

The problem with the two digit DBNS architecture is the trade off of doubling the number of channels, for reducing the exponents by a factor of two. Because of this trade-off, it can be difficult to decide when to use a two digit DBNS system and when to use a hybrid DBNS system. If a single filtering channel could be removed, the hardware savings to a two digit DBNS structure would be obvious. Then use of a two digit DBNS system would almost always be the better design choice.

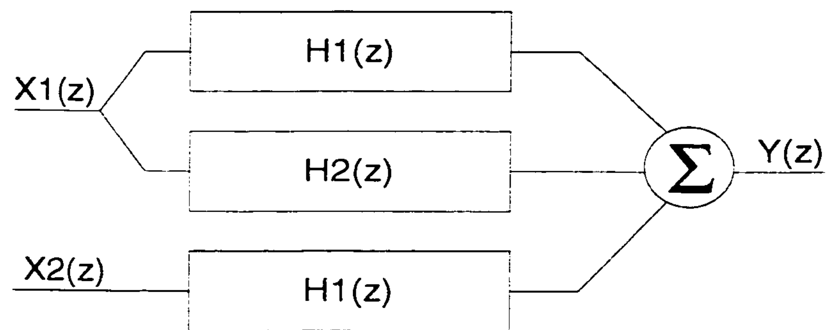
If the Greedy Algorithm, or the modified Greedy Algorithm, is used to generate both the data and the coefficients, then the following relationships can be made: $X_1 \gg X_2$ and $H_1 \gg H_2$. This is due to the first digit being as close as possible to the target number, and the second digit correcting for the error, even for the mapping of small numbers. To illustrate this point Table 4.2 shows the DBNS coefficients of a 53 tap bandpass FIR filter. The closest value between two digits is in the third entry of the first column, 8.7891e-03 and 2.1701e-04, and there is more than an order of magnitude between them. Obviously the relationship of $X_1 \gg X_2$ and $H_1 \gg H_2$ holds true for this case.

Table 4.2: 53DBNS bandpass FIR filter coefficients

1st Digit	2nd Digit	1st Digit	2nd Digit	1st Digit	2nd Digit
-2.0833e-02	-1.9290e-04	9.7546e-03	5.7156e-05	-6.2500e-02	-2.2862e-04
0	0	0	0	0	0
8.7891e-03	2.1701e-04	1.0547e-01	1.9290e-04	-9.7546e-03	-1.7147e-04
0	0	0	0	0	0
2.6367e-02	3.8580e-04	-9.7546e-03	-9.0422e-06	4.3896e-02	3.4294e-04
0	0	0	0	0	0
-9.2593e-03	-1.9052e-05	-3.1641e-01	-1.6479e-03	9.2593e-03	1.7147e-04
0	0	0	0	0	0
-3.2922e-02	-7.1445e-06	4.7461e-01	3.5156e-02	-3.2922e-02	-7.1445e-06
0	0	0	0	0	0
9.2593e-03	1.7147e-04	-3.1641e-01	-1.6479e-03	-9.2593e-03	-1.9052e-05
0	0	0	09	0	0
4.3896e-02	3.4294e-04	-9.7546e-03	-9.0422e-06	2.6367e-02	3.8580e-04
0	0	0	0	0	0
-9.7546e-03	-1.7147e-04	1.0547e-01	1.9290e-04	8.7891e-03	2.1701e-04
0	0	0	0	0	0
-6.2500e-02	-2.2862e-04	9.7546e-03	5.7156e-05	-2.0833e-02	-1.9290e-04
0	0	0	0		

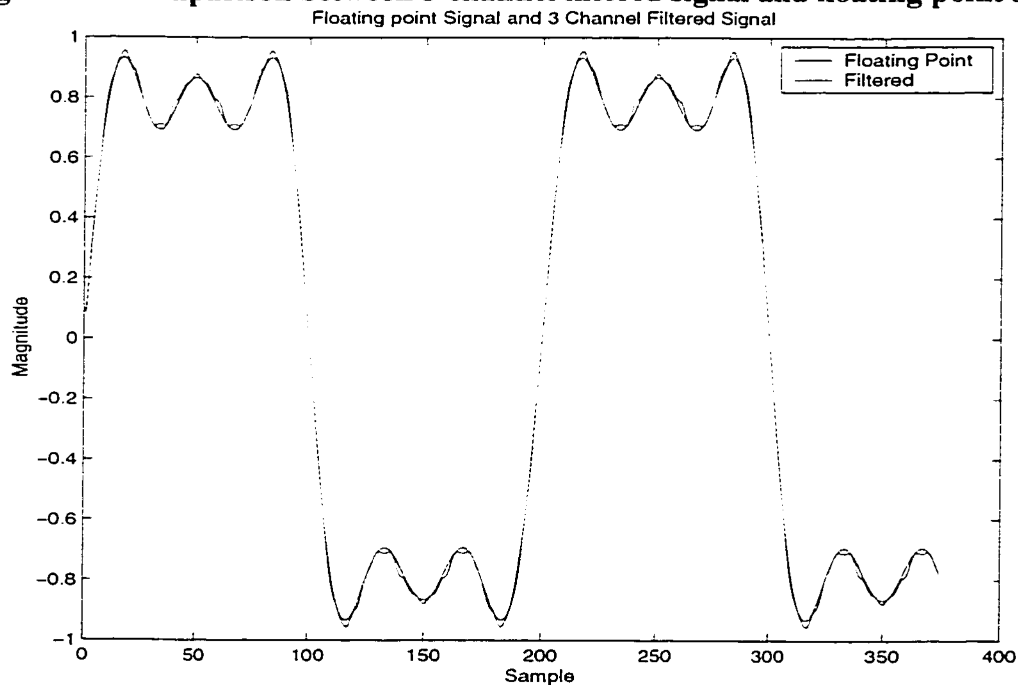
Using Equation (4.4), it can be seen that the last term, X_2H_2, becomes negligible because $X_1H_1 \gg (X_1H_2$ or $X_2H_1) \gg X_2H_2$ and $Y(z) = X_1H_1 + X_1H_2 + X_2H_1 + X_2H_2$. The last term is so small in comparison to the other three that it becomes insignificant. The new architecture, using only three channels if shown in Figure 4.5.

Figure 4.5: 2 digit reduced channel architecture



A small loss in accuracy of the filter should be expected. Figure 4.6 shows the plot of the 3 channel filtered signal compared to the input floating point signal. and Figure 4.7 shows the error in between the filtered signals and the input signal. Both the DBNS data signals and the coefficients used here are represented using two digit DBNS with binary exponents of 6 bits and ternary exponents of 5 bits.

Figure 4.6: Comparison between 3 channel filtered signal and floating point signal



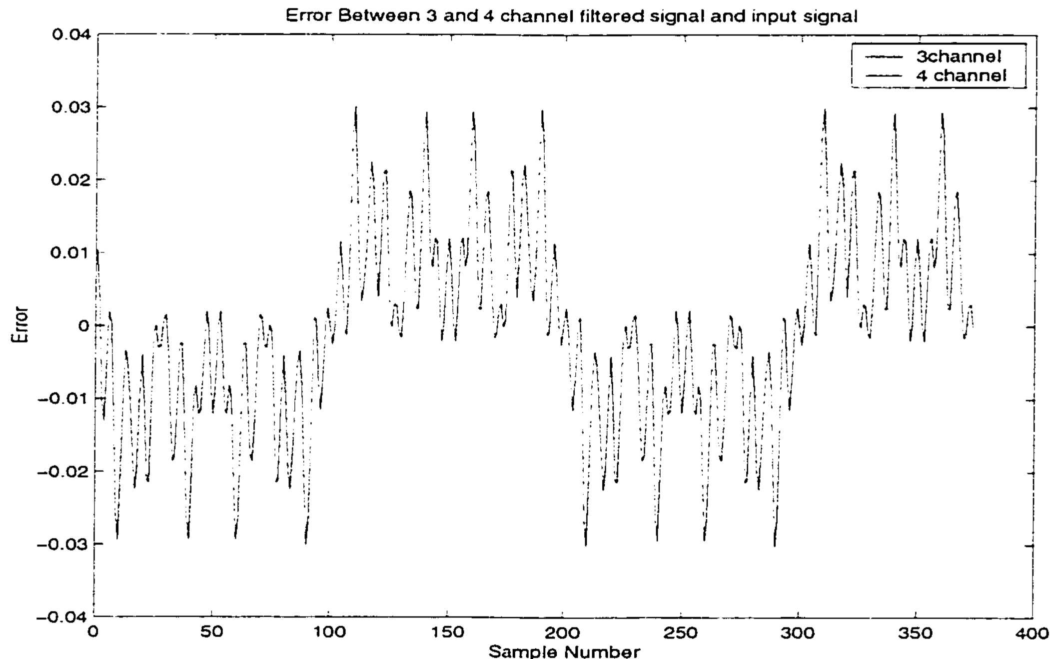
From Figure 4.7, it can be seen that the error in the 3 channel and 4 channel filtered signals are very close. The amount of noise in a signal can be characterized by its Signal to

Noise Ratio (SNR). The signal to noise ratio is calculated using the following formula [18]:

$$SNR = 20 \cdot \log\left(\frac{signal}{noise}\right) \text{ dB} \tag{4.5}$$

Using this calculation the signal to noise ratio of the filtered signal of the 3 channel filter was calculated to be 35.50dB and the SNR of the 4 channel was 35.46dB. Note that the errors are very close, and that the 3 channel filter actually has a better SNR result. This should not always be expected to occur.

Figure 4.7: Error between input signal and filtered signals



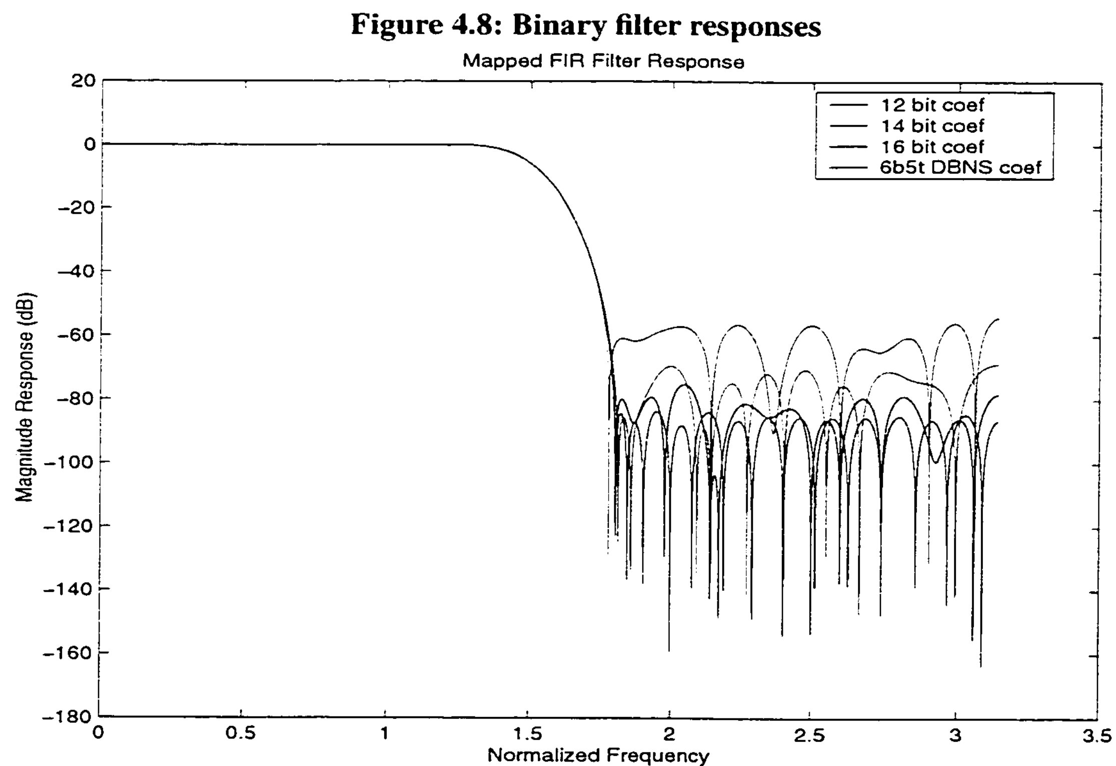
The error could be reduced, thereby increasing the SNR, by increasing the number of bits used to represent the input data. The increase in the number of bits in the input data could increase the size of the ternary ROM used in the DBNS to binary conversion, if the number of coefficient bits is similar to the number of data bits. In this case, it makes sense to also increase the size of the coefficient exponents to match the number bits used for the data bits, as it would not further increase the size of the ROM. This increase in the

dynamic range of the filter coefficients would further increase the accuracy of the filter, reducing the error and increasing the signal to noise ratio.

4.6 Comparison to Binary

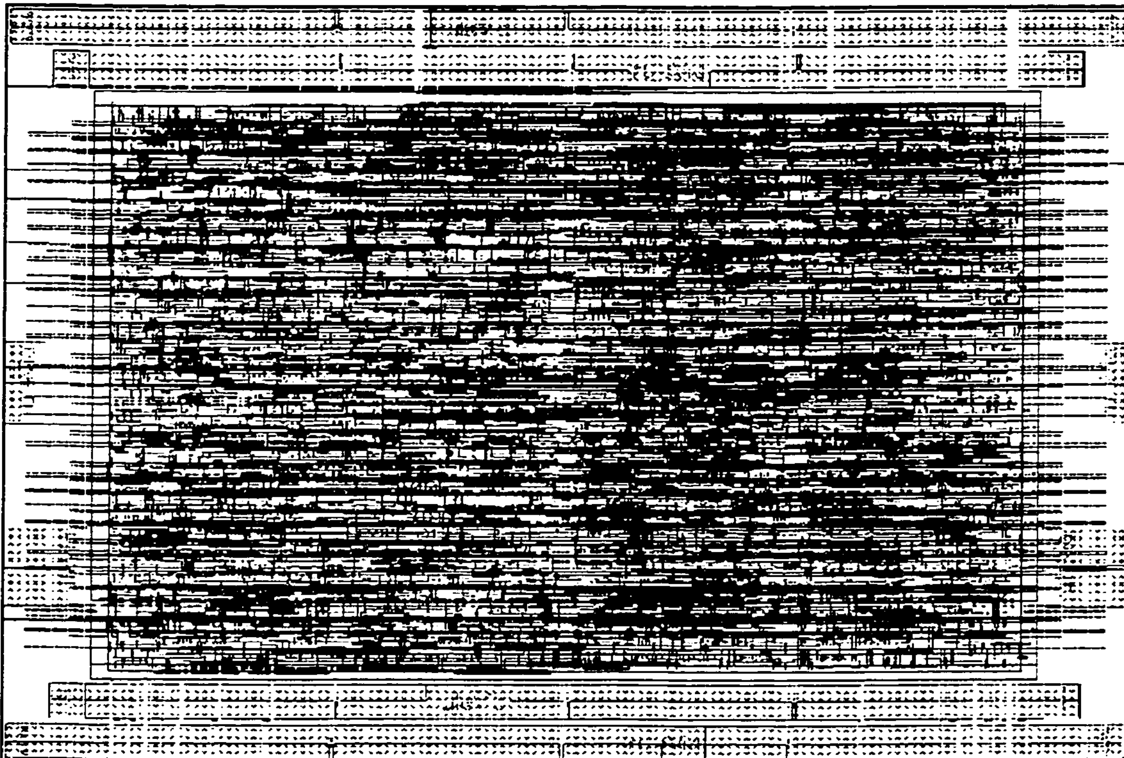
A comparison to a binary system is necessary to show how DBNS compares to a standard design. The binary system follows the same systolic array architecture, using a pipeline of MAC cells, as the DBNS filter described above. The binary MAC cell was approximated using the same design flow used for both the hybrid and two digit DBNS FIR filter designs (as discussed in Chapter 5). The approximation of the binary MAC cell area was produced by adding the area of a 16 bit binary multiplier (see Figure 4.9) and the binary accumulator from the DBNS design (see Figure 5.7). A 16 bit binary MAC cell was used because it requires 16 bits for a direct mapped binary filter to produce a similar frequency response to that of the two digit DBNS filter. The frequency responses of a range of direct mapped binary filters compared to the two digit DBNS filter can be seen in Figure 4.8.

Figure 4.8: Binary filter responses



Unfortunately the design flow and tools used lacked access to any ROM libraries. This disadvantages the DBNS designs as they rely heavily upon the use of ROMs, which will have to be made from combinational logic, greatly increasing the area required for a look-up table. Therefore the size of the DBNS MAC cells are larger than they would normally be, if designed with access to ROM libraries. The binary system has the advantage of not using any ROMs at all.

Figure 4.9: Binary multiplier cell (660x925μm)



The binary MAC cell is composed of a binary multiplier and an accumulator. The DBNS MAC cells have the structure as shown in Figure 4.2. The binary filter requires only one MAC cell per tap, whereas the hybrid and two digit DBNS require two and three MAC cells per tap, respectively. The filtering channels occupy the majority of the area of a chip so only a filter tap will be compared. Note that the DBNS MAC cells include the area necessary for the latches required for the loadable coefficients and the systolic array,

where the binary MAC cell does not. Table 4.3 summarizes the area required for a single filter tap.

Table 4.3: Area comparison between binary and DBNS filter taps

	Binary	Hybrid DBNS	2 Digit DBNS
MAC area (μm^2)	647,564	3,021,782	215,985
MAC/tap	1	2	3
Tap area (μm^2)	647,564	6,043,564	647,955

The two digit DBNS tap is only slightly larger than the binary equivalent. Though if the DBNS MAC was designed using ROM libraries it would be much smaller, making the overall tap area smaller than the binary tap. Also the area of the binary MAC is missing area required for the systolic array and loadable coefficient latches. These two factors show that DBNS FIR filters are a viable alternative to traditional binary filters.

4.7 Summary

This chapter introduced and discussed the architectural design of DBNS FIR filters. Two architectures were described, the hybrid 2 digit DBNS and the 2 digit DBNS. The 2 digit DBNS offers superior reductions in area complexity over the hybrid DBNS, particularly when using channel reduction and larger exponents. Also, despite the disadvantage of having the ROMs built from combinational logic, the 2 digit DBNS design is of comparable size to the binary design.

Chapter 5

A DBNS FIR Filter Case Study

5.1 Introduction

The Double Base Number System (DBNS) FIR filter has been shown to be theoretically sound. This was done through theory, mathematics and MatLab simulations. To fully test and prove the theory behind the DBNS FIR filter a physical microchip should be built and tested.

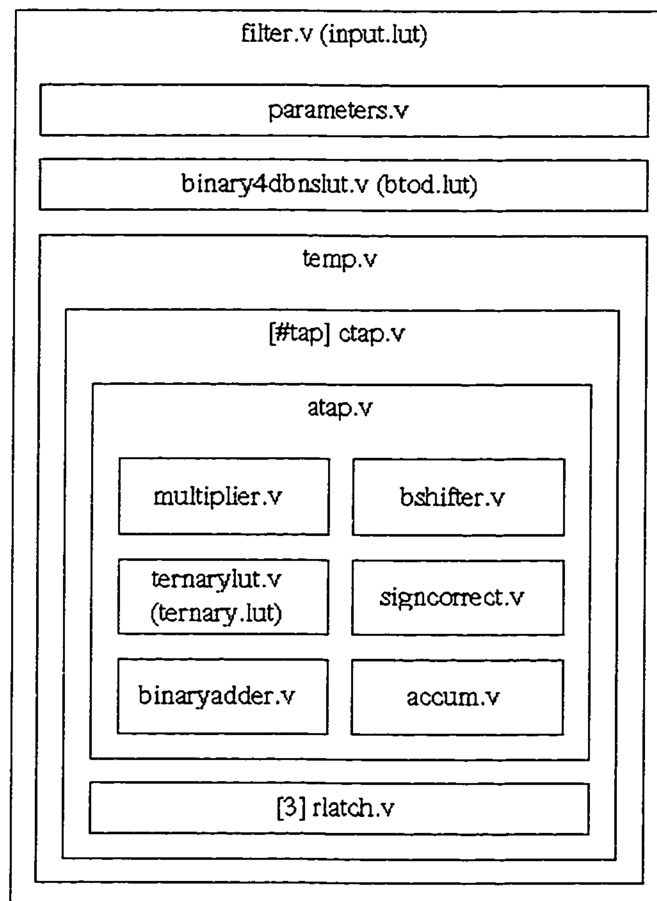
A microchip was built and tested early during the research to test the validity of the DBNS FIR filter architecture. The designed filter was a 15 tap hybrid DBNS FIR filter with single digit, 10 bit binary and ternary exponents for the coefficients and 2 digit, 5bit binary and 4 bit ternary exponents for the data. The design methodology is described below. The Hardware Description Languages (HDLs) used for the design were Verilog for simulation and VHDL for synthesis. The work presented here was the combined effort of the author and Mr. Roberto Muscedere.

5.2 Verilog Simulation

The first step was to ensure that the design behaves as expected. Verilog is the tool of choice for this stage, because the simulation

results are easier to interpret and understand than simulation in VHDL. The Verilog code used in our chip design was built in a hierarchical structure. Using a top-down approach, the overall filter was described in Verilog. Next a single tap was described, and finally the individual components of a tap. Figure 5.1 shows the overall Verilog hierarchy. The simulation was performed for a 57 tap FIR filter.

Figure 5.1 Hierarchal diagram for HDL description of a DBNS filter

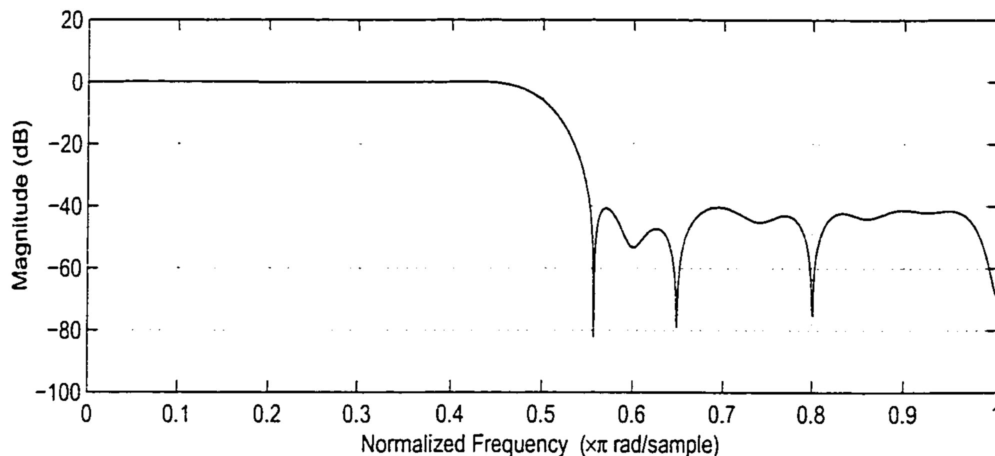


The filter is designed to have three distinct stages. The first stage is a binary to DBNS converter. This allows the data to be input in binary format. The converter was implemented through the use of a large look-up table ROM. A binary to DBNS converter [26] could not be used because the size of the data exponents, 5 bit binary and 4 bit ternary, are too small for the conversion algorithm to work. Next is the filtering stage made up from 2 parallel filtering channels. These channels are made entirely from DBNS MAC

cells. Finally there is the output stage which sums the binary output of the two filtering channels for a final output. The second and third stage together form the hybrid DBNS filter as shown in Figure 4.3. Note that the input and the output of this filter chip is in binary, while the inner workings use DBNS arithmetic. This allows the chip to be used in existing systems, or for use without needing to know anything about DBNS.

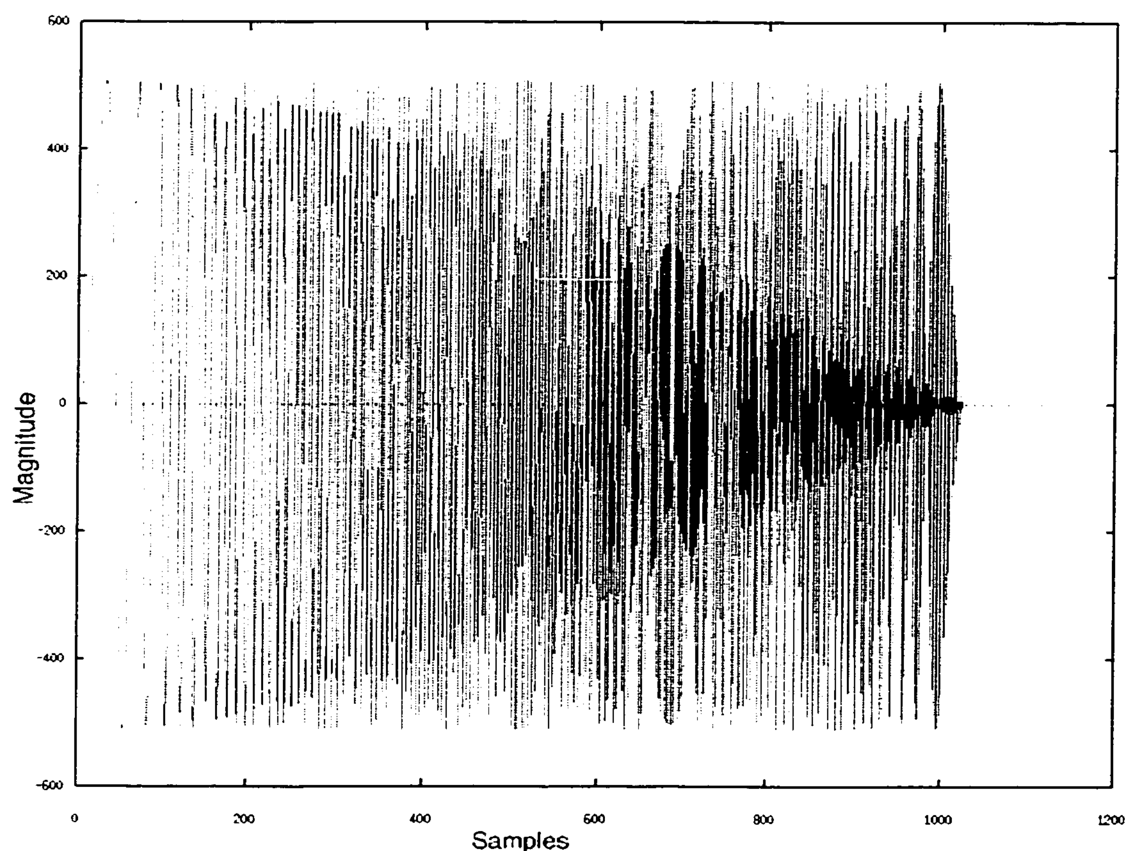
The Verilog code was written to use loadable/programmable coefficients. These coefficients are loaded directly into the filtering channels as the DBNS coefficients' exponents in 2's complement binary. A genetic algorithm was used to produce the coefficients used for the simulation. The order of the filter was 57 with a stopband attenuation of 40dB and a passband ripple of 0.0853dB, and the magnitude response can be seen in Figure 5.2.

Figure 5.2 Magnitude response of genetic algorithm designed DBNS FIR filter



Of the signals used in testing, the signal that shows the filtering effect best is a chirp signal. The chirp signal used was a sine wave whose frequency ramped from 1kHz to 512kHz over the course of 1000 samples. A plot of this chirp signal can be seen in Figure 5.3. This signal is supplied to the simulation in binary format, converted to 2 digit DBNS, by the first stage of the chip, and then filtered. The final output of the filter can be seen in Figure 5.4. Note that the output signal repeats, using the beginning of the 1kHz sine wave and the end of the 512kHz sine wave as the period of the input signal.

Figure 5.3 Input chirp signal ranging from 1 kHz to 512 kHz



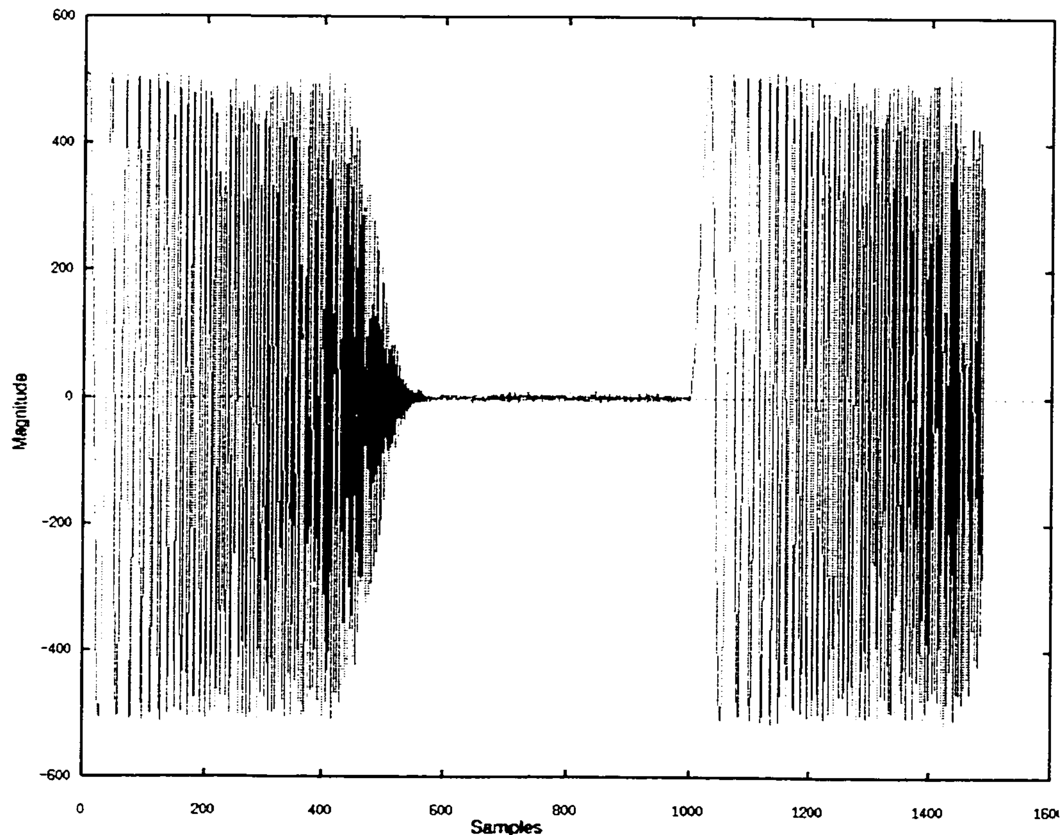
The output chirp signal shows the chirp signal rapidly dropping of and then almost flat around zero, this represents the transition and stop bands of the filter. This output signal and other test signals mean that the simulation results demonstrate that the DBNS architecture performs as the filter designed.

5.3 VHDL Synthesis

Design synthesis is what allows for rapid chip design. Synthesis takes a Hardware Description Language code file and compiles it into a physical design. This allows a person to describe a system using a programming like interface and yet produce hardware. VHSIC HDL (VHDL) was the HDL chosen for synthesis. VHDL is more suited to this task than Verilog. This is because VHDL understands concepts such as sign binary

arithmetic and sign extension and handles them automatically. Additionally, our synthesis tool, Synopsys, has better support for VHDL.

Figure 5.4 Filtered chirp signal



The VHDL was written at the behavioural level. This means that only the behaviour of the system needs to be described, such as addition, multiplication or shifting. Synopsys handles converting this description into gates or other cells supplied by a library of standard cells. The VHDL coding of the system proceeded very rapidly, due to already having the Verilog code for the system to convert into VHDL. A quick, simple simulation using the VHDL code was done to make sure that the VHDL code was correct.

Synopsys was used to take the behavioral description of the system and produce a description using the necessary library cells and connections for layout. The output was constrained to the desired timing specifications and for low area. The timing constraint

was set for 20MHz, due to the maximum frequency of the testing hardware used. The output format from Synopsys was a Verilog file for use with the Cadence tool Design Floorplanner, and a system timing file (.sdf) used in Silicon Ensemble for routing.

There was one major problem encountered during synthesis. There were no ROM libraries for use with Synopsys. This meant that the ROMs in the design, one for the input stage and one for each DBNS MAC cell, must be created from combinational logic. This greatly increases the area of the components that contain ROMs.

Due to the very large nature of the system, there were too many gates to make synthesizing the system all at once practical. Therefore another approach was needed. The system was made up of 3 main components: the input stage, DBNS MAC cells, and the output accumulator. To make synthesizing feasible each component was processed as if it were a separate chip. This produced a macro block of each component used later for manual chip assembly.

5.3.1 Placement and Routing

Placement and routing are used to take the gate level description of a system and produce a physical layout. This layout, with only a few minor modifications, is used to fabricate the microchip. Placement is used to place the library cells in a location of the microchip in such a way as to optimize routing and minimize timing. Routing finds the best way to connect the cells in a manner that minimizes delay, especially for the clock signals and critical paths. Throughout both placement and routing, the timing constraint was set for 20MHz, due to the limitations of testing equipment used.

Normally all the cells for the chip would be placed, and then the entire chip would be routed. For the hybrid DBNS filter this was not possible. There were too many gates for the workstations used to handle at one time. Therefore, each of the main components were processed as if it were a chip.

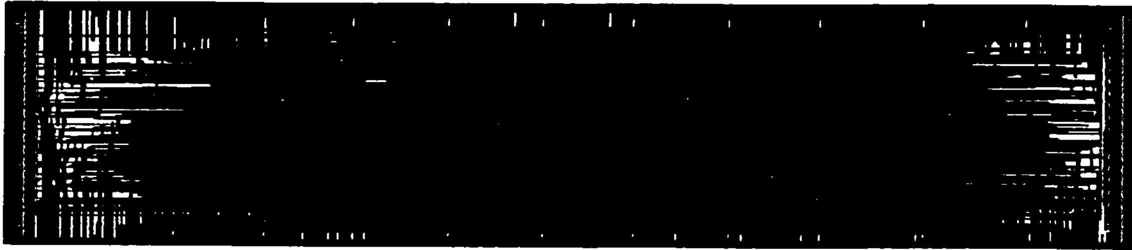
Design Floorplanner was the tool used to perform the placement of the library cells. Before the library cells are placed, there are a few steps that need to be taken. First the pads must be setup and placed in an efficient manner (i.e. the inputs on the top, outputs on the bottom and the power and ground on the sides), then the cells must be grouped. This places cells that should be in close proximity together. Next the groups are arranged on the die and some placement parameters must be set. These parameters include placement of the power and ground rails and the channel width between rows of cells and the clock tree must be generated. The clock tree is a very important factor of the placement. It ensures that the clock signal will arrive at all cells as close as together as possible and with a minimum of skew. Once these parameters are set and the clock tree is generated, the cells can be placed.

When the cells are placed, the placement is checked to see the congestion of the wire routing. If the congestion is too high, the placement must be performed again, adjusting parameters to allow for more space between the cells. When the congestion in the design is lowered to acceptable levels, the layout is exported for use in Silicon Ensemble.

The routing for the design was handled through Silicon Ensemble. The output from Design Floorplanner and the *.sdf file from Synopsys are imported into Silicon Ensemble. Silicon Ensemble is used to first route the power and ground, next the clock is routed to ensure the clock gets the best possible routing. Finally, the remainder of the chip is routed.

Sometimes the automated routing will violate design rules. The best method for dealing with these errors is with the auto fix and replace tool, Froute. When using Froute, it is best to fix the violations on the pads first, then on the core. When using Froute on the core it is best to start with small areas and expand the area if the violation cannot be fixed. Even very large numbers of violations can be repaired in this manner. On the DBNS ALU macro block over 27,000 rule violations were repaired.

Figure 5.5 Input Stage, Binary to DBNS Conversion ROM (2969x666μm)



Once all the violations are repaired, the design is imported into Cadence Design Frameworks II (or Cadence). The macro blocks of the three macro blocks can be seen in Figure 5.5, Figure 5.6 and Figure 5.7. Only one input and one output stage are required for the filter, but the chip has two processing channels each composed of 15 DBNS ALUs, for a total of 30 DBNS ALUs. The input stage is 2969x666μm, the DBNS ALU is 3526x857μm and the output stage is 328x113μm.

Figure 5.6 DBNS MAC cell (3526x857 μm)

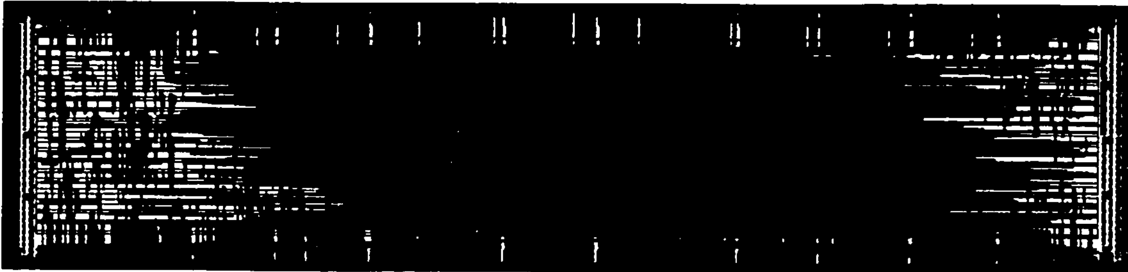
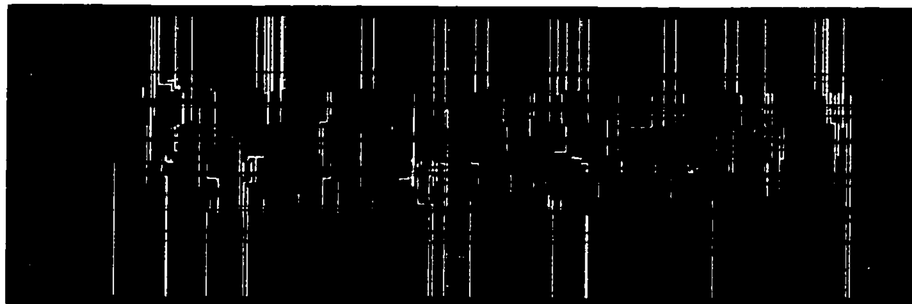


Figure 5.7 Output Stage, Accumulator for both Filter Channels (328x113 μm)



5.4 Final Chip Assembly

The overall system was not synthesized all at once, but broken into smaller blocks. This meant that the final chip had to be manually assembled. The tool for manually assembling the chip is Cadence Design Frameworks II.

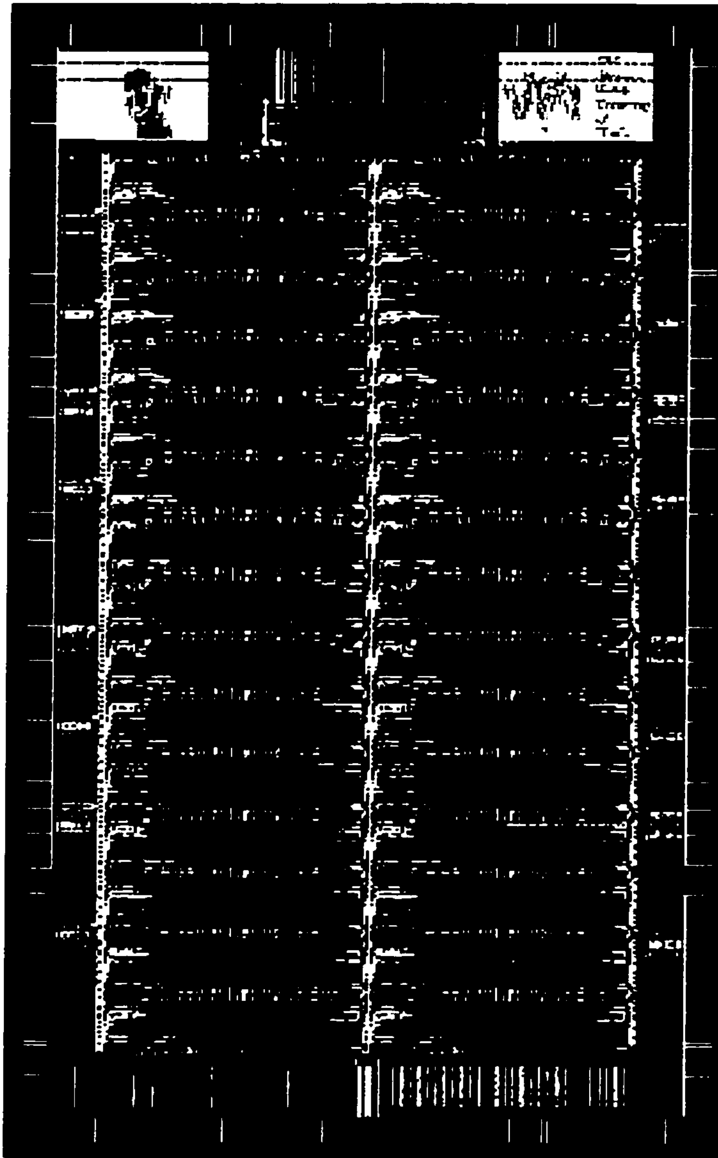
The macro blocks require some minor modifications. These modifications are performed so that when the blocks are abutted the proper connections are formed. Once the modifications are made, the pads and macro cells can be manually placed and the pads connected to the core.

The second from last step is to check if the design meets the process requirements. The process may require a certain percentage of the area to be covered by each process layer. For example the hybrid DBNS filter design had 19% of the area polysilicon and 24% of the area metal 3, where the process required 30% of both for the area. Extraneous blocks of polysilicon and all metal layers had to be placed in order to meet these process requirements.

The final step before sending the chip for fabrication is a Design Rule Check (DRC). This final check makes sure that the design does not violate any of the design rules that could cause fabrication errors.

The final size of the chip was 16x9mm. A picture of the final chip can be seen in Figure 5.8.

Figure 5.8 Final Chip Layout (16x9 mm)



5.5 Testing the Chip

The first step in testing is to setup the test head. This involves wirewrapping the testhead pins connected to the chip to the pins that connect to the test fixture connectors. Once the wirewrapping is done a power up test can be performed.

The power up test is the first test to perform on a new chip. It checks to see if there are any short circuits in the chip or test head. By performing the power up test, a problem in the chip package was uncovered. The bonding wires between the chip die and the chip package had too much slack and were pressed against the package cover or another wire. This caused short circuits between the bonding wires, destroying the chip. To recover from this problem, the chip had to be tested with the cover removed and the bonding wires carefully separated ahead of time.

The next test performed was a functional test. For this test, test vectors were generated from the data used during the Verilog simulations. Because the test vectors were generated from previous simulation data, the desired output of the filter is known. The functional test produced results similar to the Verilog simulation results.

Both the power up and the functional tests of the chip were performed successfully and with positive results. Therefore the DBNS FIR filter and the chip behaved as expected, and can be considered a successful design.

5.6 Comparison to 2 digit DBNS

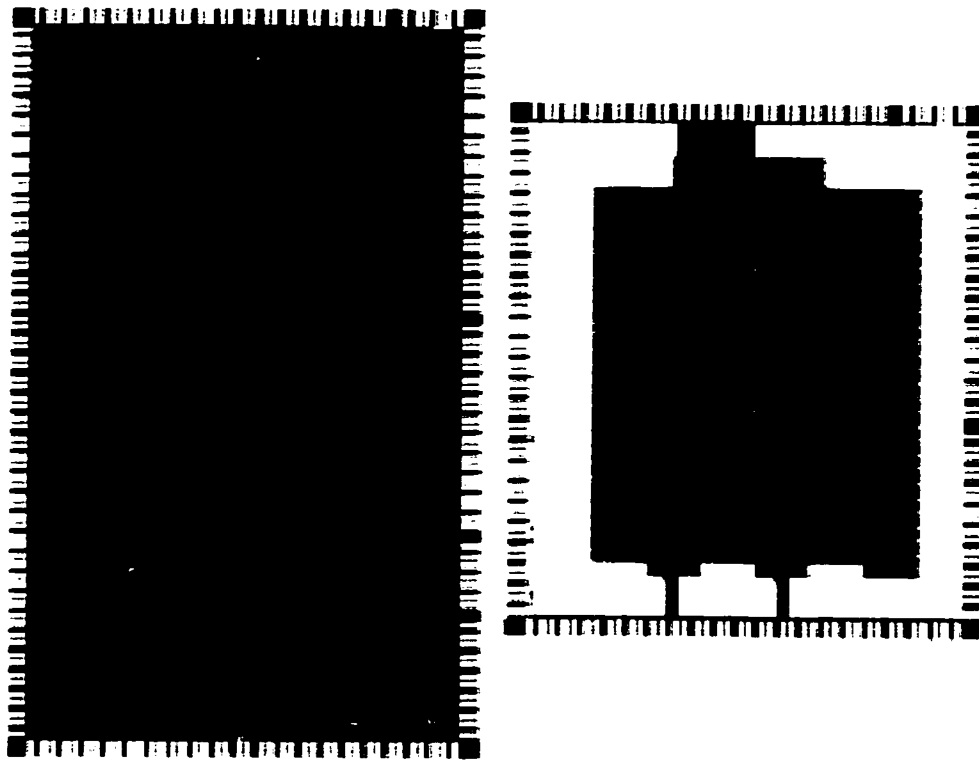
In order to compare the area savings between a hybrid DBNS design and a full two digit DBNS design, a two digit DBNS FIR filter chip had to be designed. The two digit DBNS filter used 5 bit binary and 4 bit ternary for the coefficients, and used the same number of exponent bits for the input data as the hybrid DBNS filter design. This chip followed the exact same design flow as the hybrid DBNS design described above. The only difference in the design flow was that the two digit design was not sent away for fabrication and did not have the extraneous layers placed to satisfy the process rules.

Figure 5.9 shows the 15 tap, hybrid DBNS FIR filter, described in the previous section, next to a 53 tap, channel reduced, two digit DBNS FIR filter. Note that these two chips are on the same scale, and that the two digit DBNS uses considerably less area than the hybrid DBNS filter, even though the two digit filter has 53 taps, three and a half times the number

of taps. Normally the size difference would not be so severe. but the lack of ROM libraries vastly increased the size of the ROMs used in the design. and therefore the area required by a MAC cell on the ROM. This affects the hybrid DBNS design more because the size of the required ROMs are much larger than the ROMs used in the two digit DBNS design

.

Figure 5.9 15 tap, hybrid DBNS FIR filter (left) and a 53 tap, reduced channel, 2 digit DBNS FIR filter (right)



5.7 Summary

This chapter demonstrates that the DBNS representation can be used to make practical filters. The filters designed used industry standard tools and standard libraries. The only lacking industry tool was a ROM library. If ROM libraries were used, the design would consume considerably less area, making DBNS a more viable design. The input stage would appear to grow to unreasonable size for data requiring a large number of bits. This can be avoided by using a binary to DBNS converter, that was unusable for this design since the data dynamic range was too low.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis has explored applications of the double-base number system to the implementation of systolic FIR filters. We have discovered that the DBNS number system offers significant advantages over standard binary systems, mainly through overhead reduction in area achieved by not using multipliers. This applies not only in systems with specialized needs, but in general DSP tasks. Also the design of the coefficients for DBNS filters is very easy. No computationally intensive tools or algorithms are needed to find coefficients, since floating point results can be mapped directly into a 2-digit DBNS format with a simple greedy algorithm.

The investigation into full 2 digit DBNS FIR filter architecture has shown that the 2 digit architecture is much more area efficient than the hybrid DBNS architecture. The full 2 digit DBNS filter architectures are also found to be very competitive to binary implementations in area of silicon required for the fabrication.

The microchip designed in Chapter 5 was designed to operate in a binary environment. This means that DBNS filtering systems can be

fully integrated into a binary system without any special modifications.

6.2 Contributions

This thesis has introduced a simple mapping scheme for converting floating point numbers into the DBNS representation. This mapping produces results with acceptable accuracy, allowing the use of existing floating point filter design tools.

The hybrid 2 digit DBNS FIR filter was extended into a two digit DBNS FIR filter. With this change the DBNS filter achieves a much lower area complexity than the hybrid DBNS filter. The use of our greedy 2-digit mapping scheme allows for a channel reduction, decreasing the area complexity of a DBNS filter by 25%.

We have also introduced the concept of asymmetric coefficient indices that can produce a reduction in area complexity compared to the use of symmetric indices. By increasing the binary exponent, and decreasing the ternary exponent, the number representation range remains almost constant while decreasing the size of the conversion ROM used in our multiplier/accumulator processor.

A microchip was fabricated as a proof of concept for DBNS filtering architectures. The successful testing of this chip has shown that DBNS filters can be made in practice and are a practical alternative to binary implementations.

6.3 Suggestions for Future Work

The Double Base Number System has only recently been disclosed, and there are many areas that still need to be investigated. First of all, the use of odd bases other than 3 needs to be investigated. The use of other odd bases may offer many advantages, even if only to specialized tasks.

In the filter design discussion of Chapter 3, we see that the largest errors in DBNS mapped filters, occur on the small number of largest value coefficients around the point of symmetry of the impulse response. By reducing this mapping error, the mapped filter response would be much closer to the original designed filter. Methods for reducing this error, such as scaling the coefficients to produce more favourable numbers for mapping, need investigation.

The effects of channel reduction in two digit DBNS filter needs to be examined more carefully. While favourable results were achieved in this thesis, in general this may not be the case.

An ASIC filter design of both the hybrid and two digit DBNS FIR filter needs to be produced using custom cells and ROMS. This will show the true benefit from overhead reduction caused by using DBNS. We can also use such a design study to perform a more realistic comparison with competing binary implementations.

REFERENCES

[8] V.S. Dimitrov, G.A. Jullien and W.C. Miller. "*Theory and Applications of the Double Base Number System.*" IEEE Trans. on Computers. Oct 1999, pp. 1098-1106 vol 48.

[9] G.A Jullien, V.S. Dimitrov, B. Li, W.C Miller, A. Lee, and M. Ahmadi. "*A Hybrid DBNS Processor for DSP Computation.*" Proceedings of IEEE International Symposium on Circuits and Sytems (ISCAS) 1999, Orlando, FL, May 30 - June 2. Vol 1, pages I5 - I8.

[10] V.S.Dimitrov, J.Eskritt, L.Imbert, G.A.Jullien and W.C.Miller. "T*he use of the multi-dimensional logarithmic number system in DSP applications.*" 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, June 2001. Accepted for presentation.

[11] S. Y. Kung. "*VLSI Array Processors.*" Prentice Hall 1988. Pages 198, 222.

[12] J. L. Hennessy and D. A. Patterson. "*Computer Organization and Design: The Hardware/Software Interface.*" Morgan Kaufmann Publishers 1998. Pages 436

[13] A. Lee. "*Design of 1-D and 2-D Filters with Finite-Wordlength Coefficients Using Genetic Algorithms.*" Master Thesis, University of Windsor, 2000

[14] M. Shahkarami. "*Exploiting Redundancy in Modulus Replication Product Processors.*" Ph.D. Thesis, University of Windsor, 1999

[15] V.S. Dimitrov, Saeid Sadeghi-Emamchaie, G.A. Jullien and W.C. Miller. "*A Near Canonic Double-Based Number System (DBNS) with Applications in Digital Signal Processing.*" Invited paper, SPIE Conference on Advanced Signal Processing Algorithms, Denver, 1996.

[16] A. Antoniou. "*Digital Filters: Analysis, Design and Applications.*" McGraw-Hill 1993. Pages 51, 247

[17] J. Eskritt, R. Muscedere, G.A. Jullien, V.S. Dimitrov and W.C. Miller. "*A 2-Digit DBNS Filter Architecture.*" Proceedings of the 2000 IEEE Workshop on Signal Processing Systems (SiPS 2000), Lafayette, LA, October. Pages 447-456.

[18] M. S. Roden. "Analog and Digital Communication Systems." Prentice Hall 1996. Pages 137-138.

[19] R. King, M. Ahmadi, R. Gorgui-Nguib, A. Kwabwe, M. Azimi-Sadjadi. "*Digital Filtering in One and Two Dimensions: Design and Applications.*" Plenum 1989

[20] A. Oppenhiem, R. Schafer. "*Discrete-Time Signal Processing.*" Prentice Hall 1989.

[21] J. Holland. "*Adaption in Natural and Artifical Systems.*" University of Michigan Press, 1975.

[22] D. Goldberg. "*Genetic Algorithm in Search, Optimization, and Machine Learning.*" Addison Wesley, 1989.

[23] A. Lee, M. Ahmadi, G. A. Jullien, W. C. Miller, and R. S. Lashkari, " *Design of 1-D FIR Filters with Genetic Algorithm*", Proc. 1999 Inter. Symp. on Circ. and Syst., 17.9.

[24] D.M.Lewis. "*Interleaved memory function interpolators with application to an accurate LNS arithmetic unit.*" IEEE Trans. Computer Arithmetic, Vol. 43, No. 8, pp.974-982, 1994.

[25] J.N. Coleman, E.I. Chester, C.I. Softley and J. Kaldec, Arithmetic on the European Logarithmic Microprocessor, IEEE Trans. Comp., vol 49, No 7, pp 702-715, 2000

[26] R. Muscedere, G.A. Jullien, V.S. Dimitrov, and W.C Miller. "*Non-linear signal processing using index calculus DBNS arithmetic.*" SPIE Vol. 4116, November 2000

[27] S. Sadeghi-Emamchaie, G.A. Jullien, V.S. Dimitrov and W.C. Miller. "*Cellular Neural Network Implementation of Digital Arithmetic using Symbolic Substitution.*"

[28] Y. Taur and T. Ning. "*Fundamentals of Modern VLSI Devices.*" Cambridge University Press 1998.

[29] T. Dillinger. "*VLSI Engineering.*" Prentice Hall 1988.

[30] D. J. Smith. "*HDL Chip Design.*" Doone 1998.

Appendix A

VHDL Code for Hybrid DBNS FIR Filter

A.1 Introduction

These files were originally written for the design and fabrication of the hybrid filter described in Chapter 5. The code was written by Roberto Muscedere with modification by Jonathan Eskritt. This VHDL is only suitable for synthesis. To use the code for simulation, a set of wrapper files would be required to "glue" the components into a complete filter, and a test bench and data would have to be written.

A.2 Binary to DBNS Coversion Stage

This is the VHDL code for the binary to DBNS conversion ROM. The code has directives to include btod.vhd and parameters.vhd.

A.2.1 binary4dbnslut_export.vhd

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

library DBNS;
use DBNS.btod.all;
use DBNS.DBNSfilter.all;
use DBNS.parameters.all;
```

```
entity binary4dbnslut_export is
	port(CLKI: in STD_LOGIC;
	CLKO: out STD_LOGIC;
	CRESETI: in STD_LOGIC;
	CRESETO: out STD_LOGIC;
	BINARYIN: in SIGNED(dsize downto 0);
	SIGN1: out SIGNED(1 downto 0);
	INDEXBIN1: out SIGNED(dbsize1 downto 0);
	INDEXTERN1: out SIGNED(dtsize1 downto 0);
	SIGN2: out SIGNED(1 downto 0);
	INDEXBIN2: out SIGNED(dbsize2 downto 0);
	INDEXTERN2: out SIGNED(dtsize2 downto 0) );
end binary4dbnslut_export;

architecture behaviour of binary4dbnslut_export is
begin

	U0: binary4dbnslut
		generic map ( dsize,dbsize1,dtsize1,dbsize2,dtsize2 )
		port map ( CLKI, CLKO, CRESETI, CRESETO, BINARYIN, SIGN1,
				INDEXBIN1, INDEXTERN1, SIGN2, INDEXBIN2, INDEXTERN2 );

end behaviour;
```

A.2.2 btod.vhd

This file descirbes the look-up table used for the conversion. It is truncated here. to save space, as it is over a thousand lines long.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

package btod is
	subtype ROM_WORD_5_4 is UNSIGNED(21 downto 0);
	subtype ROM_RANGE_1024 is INTEGER range -512 to 511;
	type ROM_TABLE_1024_5_4 is array (ROM_RANGE_1024'low to
ROM_RANGE_1024'high) of ROM_WORD_5_4;

	constant btod_1024_5_4: ROM_TABLE_1024_5_4 := (

"1101001000000000000000",
"1101001000001000000000",
"1101001000001000010000",
	.
	.
	.
"0101001000011000000001",
"0101001000011000010000",
"0101001000011000000000"
);
```

```
end btod;
```

A.2.3 parameters.vhd

The parameter file includes all the variable parameters. such as bit width and bus size. These parameters are stored here for global use in all VHDL throughout the design.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

package parameters is

constant dsize : INTEGER := 9;
constant dbsize1 : INTEGER := 4;
constant dtsize1 : INTEGER := 3;
constant dbsize2 : INTEGER := 4;
constant dtsize2 : INTEGER := 3;
constant cbsize : INTEGER := 9;
constant ctsize : INTEGER := 9;
constant dbsize : INTEGER := 4;
constant dtsize : INTEGER := 3;
constant mbsize : INTEGER := 9;
constant mtsize : INTEGER := 9;
constant lbsize : INTEGER := 10;
constant ltsize : INTEGER := 9;
constant absize : INTEGER := 9;
constant sbsize : INTEGER := 12;
constant asize : INTEGER := 18;
constant osize : INTEGER := 15;

constant taps : INTEGER :=  57;

end parameters;
```

A.3 DBNS MAC Cell

The DBNS MAC cell is the most fundamental cell of the DBNS filter. These cells are what make up the filter itself. The cells include all the circuitry for loadable coefficients. This VHDL file relies on the the ltap.vhd file and the parameters.vhd file from Section A.2.3.

A.3.1 ltap_export.vhd

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

library DBNS;
use DBNS.DBNSfilter.all;
use DBNS.parameters.all;

entity ltap_export is
       port(CLKI: in STD_LOGIC;
       CLKO: out STD_LOGIC;
       CRESETI: in STD_LOGIC;
       CRESETO: out STD_LOGIC;
       LRESETI: in STD_LOGIC;
       LRESETO: out STD_LOGIC;
       DATABININ: in SIGNED(dbsize downto 0);
       DATABINOUT: out SIGNED(dbsize downto 0);
       DATASIGNIN: in SIGNED(1 downto 0);
       DATASIGNOUT: out SIGNED(1 downto 0);
       DATATERNIN: in SIGNED(dtsize downto 0);
       DATATERNOUT: out SIGNED(dtsize downto 0);
       ACCUMIN: in SIGNED(asize downto 0);
       ACCUMOUT: out SIGNED(asize downto 0);
       LCLKI: in STD_LOGIC;
       LCLKO: out STD_LOGIC;
       COEFBININ: in SIGNED(cbsize downto 0);
       COEFSIGNIN: in SIGNED(1 downto 0);
       COEFTERNIN: in SIGNED(ctsize downto 0);
       COEFBINOUT: out SIGNED(cbsize downto 0);
       COEFSIGNOUT: out SIGNED(1 downto 0);
       COEFTERNOUT: out SIGNED(ctsize downto 0) );
end ltap_export;

architecture behaviour of ltap_export is

begin

       U0: ltap
              generic map ( cbsize, ctsize, dbsize, dtsize, mbsize, mtsize,
                            lbsize, ltsize, absize, sbsize, asize )
              port map ( CLKI, CLKO, CRESETI, CRESETO, LRESETI, LRESETO,
                         DATABININ, DATABINOUT, DATASIGNIN, DATASIGNOUT,
                         DATATERNIN, DATATERNOUT, ACCUMIN, ACCUMOUT, LCLKI,
                         LCLKO, COEFBININ, COEFSIGNIN, COEFTERNIN, COEFBINOUT,
                         COEFSIGNOUT, COEFTERNOUT );

end behaviour;
```

A.3.2 ltap.vhd

This is built around ctap.vhd and contains the registers necessary for the loadable coefficients.

```
library IEEE;
use IEEE.std_logic_arith.all;
```

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

library DBNS;
use DBNS.DBNSfilter.all;

entity ltap is

generic(cbsize,ctsize,dbsize,dtsize,mbsize,mtsize,lbsize,ltsize,absize,sbsize,
        asize: INTEGER);
       port(CLKI: in STD_LOGIC;
       CLKO: out STD_LOGIC;
       CRESETI: in STD_LOGIC;
       CRESETO: out STD_LOGIC;
       LRESETI: in STD_LOGIC;
       LRESETO: out STD_LOGIC;
       DATABININ: in SIGNED(dbsize downto 0);
       DATABINOUT: out SIGNED(dbsize downto 0);
       DATASIGNIN: in SIGNED(1 downto 0);
       DATASIGNOUT: out SIGNED(1 downto 0);
       DATATERNIN: in SIGNED(dtsize downto 0);
       DATATERNOUT: out SIGNED(dtsize downto 0);
       ACCUMIN: in SIGNED(asize downto 0);
       ACCUMOUT: out SIGNED(asize downto 0);
       LCLKI: in STD_LOGIC;
       LCLKO: out STD_LOGIC;
       COEFBININ: in SIGNED(cbsize downto 0);
       COEFSIGNIN: in SIGNED(1 downto 0);
       COEFTERNIN: in SIGNED(ctsize downto 0);
       COEFBINOUT: out SIGNED(cbsize downto 0);
       COEFSIGNOUT: out SIGNED(1 downto 0);
       COEFTERNOUT: out SIGNED(ctsize downto 0) );
end ltap;

architecture behaviour of ltap is

begin

       s_rlatch1: rlatch
             generic map ( 1 )
             port map ( LCLKI,LRESETI,COEFSIGNIN,COEFSIGNOUT );

       s_rlatch2: rlatch
             generic map ( cbsize )
             port map ( LCLKI,LRESETI,COEFBININ,COEFBINOUT );

       s_rlatch3: rlatch
             generic map ( ctsize )
             port map ( LCLKI,LRESETI,COEFTERNIN,COEFTERNOUT );

       s_ctap: ctap
             generic map ( cbsize, ctsize, dbsize, dtsize, mbsize, mtsize,
                          lbsize, ltsize, absize, sbsize, asize )
             port map ( CLKI, CRESETI, COEFBININ, COEFSIGNIN, COEFTERNIN,
                      DATABININ, DATABINOUT, DATASIGNIN, DATASIGNOUT,
                      DATATERNIN, DATATERNOUT, ACCUMIN, ACCUMOUT);
```

```
        process (CLKI)
        begin
              CLKO <= not ( not ( CLKI ) );
        end process;

        process (LCLKI)
        begin
              LCLKO <= not ( not ( LCLKI ) );
        end process;

        process (CRESETI)
        begin
              CRESETO <= not ( not ( CRESETI ) );
        end process;

        process (LRESETI)
        begin
              LRESETO <= not ( not ( LRESETI ) );
        end process;

end behaviour;
```

A.3.3 ctap.vhd

This description includes the latches required for the systolic array and includes the instance of the true description of the MAC cell.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

library DBNS;
use DBNS.DBNSfilter.all;

entity ctap is

generic(cbsize,ctsize,dbsize,dtsize,mbsize,mtsize,lbsize,ltsize,absize,sbsize,as
ize: INTEGER);
      port(CLK: in STD_LOGIC;
      CRESET: in STD_LOGIC;
      COEFBIN: in SIGNED(cbsize downto 0);
      COEFSIGN: in SIGNED(1 downto 0);
      COEFTERN: in SIGNED(ctsize downto 0);
      DATABININ: in SIGNED(dbsize downto 0);
      DATABINOUT: out SIGNED(dbsize downto 0);
      DATASIGNIN: in SIGNED(1 downto 0);
      DATASIGNOUT: out SIGNED(1 downto 0);
      DATATERNIN: in SIGNED(dtsize downto 0);
      DATATERNOUT: out SIGNED(dtsize downto 0);
```

```
        ACCUMIN: in SIGNED(asize downto 0);
        ACCUMOUT: out SIGNED(asize downto 0) );
end ctap;

architecture behaviour of ctap is

signal ACCUMO1: SIGNED(asize downto 0);
signal ACCUMO2: SIGNED(asize downto 0);

begin

        s_rlatch1: rlatch
                generic map ( asize )
                port map ( CLK,CRESET,ACCUMO1,ACCUMO2 );

        s_rlatch2: rlatch
                generic map ( asize )
                port map ( CLK,CRESET,ACCUMO2,ACCUMOUT );

        s_rlatch3: rlatch
                generic map ( 1 )
                port map ( CLK,CRESET,DATASIGNIN,DATASIGNOUT );

        s_rlatch4: rlatch
                generic map ( dbsize )
                port map ( CLK,CRESET,DATABININ,DATABINOUT );

        s_rlatch5: rlatch
                generic map ( dtsize )
                port map ( CLK,CRESET,DATATERNIN,DATATERNOUT );

        s_atap: atap
                generic map ( cbsize, ctsize, dbsize, dtsize, mbsize, mtsize,
                              lbsize, ltsize, absize, sbsize, asize )
                port map ( COEFBIN, COEFSIGN, COEFTERN, DATABININ, DATASIGNIN,
                           DATATERNIN, ACCUMIN, ACCUMO1);

end behaviour;
```

A.3.4 atap.vhd

This is the VHDL description of the DBNS MAC cell used to multiply the data with the coefficients and sum the results with the results of the previous stages.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

library DBNS;
use DBNS.DBNSfilter.all;
```

```
entity atap is

generic(cbsize,ctsize,dbsize,dtsize,mbsize,mtsize,lbsize,ltsize,absize,sbsize,as
ize: INTEGER);
       port(COEFBIN: in SIGNED(cbsize downto 0);
       COEFSIGN: in SIGNED(1 downto 0);
       COEFTERN: in SIGNED(ctsize downto 0);
       DATABIN: in SIGNED(dbsize downto 0);
       DATASIGN: in SIGNED(1 downto 0);
       DATATERN: in SIGNED(dtsize downto 0);
       ACCUMIN: in SIGNED(asize downto 0);
       ACCUMOUT: out SIGNED(asize downto 0) );
end atap;

architecture behaviour of atap is

signal RESULTBIN: SIGNED(mbsize downto 0);
signal RESULTTERN: SIGNED(mtsize downto 0);
signal LUTVALUE: UNSIGNED(ltsize downto 0);
signal SHIFTVALUE: SIGNED(lbsize downto 0);
signal TOTALSHIFT: SIGNED(absize downto 0);
signal NOSIGNVALUE: UNSIGNED(sbsize downto 0);
signal SIGNVALUE: SIGNED(sbsize+1 downto 0);

begin

       s_multiplier: multiplier
              generic map ( cbsize,ctsize,dbsize,dtsize,mbsize,mtsize )
              port map ( COEFBIN,COEFTERN,DATABIN,DATATERN,RESULTBIN,RESULTTERN
                        );

       s_ternarylut: ternarylut
              generic map ( mtsize,ltsize,lbsize )
              port map ( RESULTTERN,LUTVALUE,SHIFTVALUE );

       s_binaryadder: binaryadder
              generic map ( mbsize,lbsize,absize )
              port map ( RESULTBIN,SHIFTVALUE,TOTALSHIFT );

       s_bshifter: bshifter
              generic map ( ltsize,absize,sbsize )
              port map ( LUTVALUE,TOTALSHIFT,NOSIGNVALUE );

       s_signcorrect: signcorrect
              generic map ( sbsize )
              port map ( COEFSIGN,DATASIGN,NOSIGNVALUE,SIGNVALUE );

       s_accum: accum
              generic map ( asize,sbsize )
              port map ( ACCUMIN,SIGNVALUE,ACCUMOUT );

end behaviour;
```

A.3.5 rlatch.vhd

This file describes a register of latches and can be set to whatever bitwidth is required.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

entity rlatch is
        generic(wsize: INTEGER);
        port(CLK: in STD_LOGIC;
        RESET: in STD_LOGIC;
        INDATA: in SIGNED(wsize downto 0);
        OUTDATA: out SIGNED(wsize downto 0) );
end rlatch;

architecture behaviour of rlatch is
begin

        process (CLK,RESET,INDATA)
        begin
                if (RESET='1') then
                        OUTDATA <= (others => '0');
                elsif (CLK'event and CLK='1') then
                        OUTDATA <= INDATA;
                end if;
        end process;

end behaviour;
```

A.3.6 multiplier.vhd

The DBNS multiplier consists of two binary adders that add the ternary and binary exponents of the data and the coefficients

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

entity multiplier is
        generic(cbsize,ctsize,dbsize,dtsize,mbsize,mtsize: INTEGER);
        port(COEFBIN: in SIGNED(cbsize downto 0);
        COEFTERN: in SIGNED(ctsize downto 0);
        DATABIN: in SIGNED(dbsize downto 0);
        DATATERN: in SIGNED(dtsize downto 0);
        RESULTBIN: out SIGNED(mbsize downto 0);
        RESULTTERN: out SIGNED(mtsize downto 0) );
end multiplier;

architecture behaviour of multiplier is
begin
```

```
        process(COEFBIN,DATABIN,COEFTERN,DATATERN)
        begin

                RESULTBIN <= CONV_SIGNED(COEFBIN + DATABIN,mbsize+1);
                RESULTTERN <= CONV_SIGNED(COEFTERN + DATATERN,mtsize+1);

        end process;

end behaviour;
```

A.3.7 ternarylut.vhd

The ternary look-up table is used as a component in the DBNS to binary conversion. It relies on the threerom.vhd file for the ROM data.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

library DBNS;
use DBNS.threerom.all;

entity ternarylut is
        generic(mtsize,ltsize,lbsize: INTEGER);
        port(TERNARYIN: in SIGNED(mtsize downto 0);
        TERNARYOUT: out UNSIGNED(ltsize downto 0);
        BINARYOUT: out SIGNED(lbsize downto 0) );
end ternarylut;

architecture behaviour of ternarylut is
begin

        process(TERNARYIN)
        variable index: INTEGER;
        begin

                index := CONV_INTEGER(TERNARYIN);

                TERNARYOUT <= '1' &
CONV_UNSIGNED(threerom_1024_20(index)(ltsize+lbsize downto lbsize+1),ltsize);
                BINARYOUT <= CONV_SIGNED(threerom_1024_20(index)(lbsize downto
0),lbsize+1);

        end process;

end behaviour;
```

A.3.8 threerom.vhd

This module provides the data for the ternarylut.vhd ROM. The data in the ROM is truncated, as it can be generated as needed.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

package threerom is
        subtype ROM_WORD_20 is UNSIGNED(19 downto 0);
--        subtype ROM_RANGE_1024 is STD_LOGIC_VECTOR(9 downto 0);
        subtype ROM_RANGE_1024 is INTEGER range -512 to 511;
        type ROM_TABLE_1024_20 is array (ROM_RANGE_1024'low to
ROM_RANGE_1024'high) of ROM_WORD_20;
--        type ROM_TABLE_1024_20 is array (0 to 1023) of ROM_WORD_20;

        constant threerom_1024_20: ROM_TABLE_1024_20 := (

"01101001110011001011",
"00001111010011001101",
"10010111010011001110",
       .
       .
       .
"00011110001100011100",
"10101101001100011101",
"01000001101100011111",
"11100010101100100000"
);

end threerom;
```

A.3.9 binaryadder.vhd

This is a simple binary adder that is used as a component in the DBNS to binary conversion.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

entity binaryadder is
      generic(mbsize,lbsize,absize: INTEGER);
```

```
        port(BINARY1: in SIGNED(mbsize downto 0);
        BINARY2: in SIGNED(lbsize downto 0);
        BINARYOUT: out SIGNED(absize downto 0) );
end binaryadder;

architecture behaviour of binaryadder is
begin

        process(BINARY1,BINARY2)
        begin

                BINARYOUT <= CONV_SIGNED(BINARY1 + BINARY2,absize+1);

        end process;

end behaviour;
```

A.3.10bshifter.vhd

This element is a barrel shifter. That means it can shift any number of places in a single clock cycle.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

entity bshifter is
        generic(ltsize,absize,sbsize: INTEGER);
        port(TERNARYIN: in UNSIGNED(ltsize downto 0);
        SHIFT: in SIGNED(absize downto 0);
        BINARYOUT: out UNSIGNED(sbsize downto 0) );
end bshifter;

architecture behaviour of bshifter is
begin

        process(TERNARYIN,SHIFT)
        begin

                if (SHIFT = 0 ) then
                        BINARYOUT <= CONV_UNSIGNED(TERNARYIN,sbsize+1);
                elsif (SHIFT(absize)='0') then
                        BINARYOUT <= CONV_UNSIGNED(SHL(TERNA-
RYIN,CONV_UNSIGNED(SHIFT,absize+1)),sbsize+1);
                else
                        BINARYOUT <= CONV_UNSIGNED(SHR(TERNARYIN,CONV_UNSIGNED(-
SHIFT,absize+1)),sbsize+1);
                end if;

        end process;
```

```
end behaviour;
```

A.3.11signcorrect.vhd

This component ensures that the binary output of the DBNS to binary converter has the correct sign corresponding to the multiplication

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

entity signcorrect is
       generic(sbsize: INTEGER);
       port(COEFSIGN: in SIGNED(1 downto 0);
       DATASIGN: in SIGNED(1 downto 0);
       BINARYIN: in UNSIGNED(sbsize downto 0);
       BINARYOUT: out SIGNED(sbsize+1 downto 0) );
end signcorrect;

architecture behaviour of signcorrect is
begin

       process(COEFSIGN,DATASIGN,BINARYIN)
       begin

              if ( COEFSIGN = DATASIGN ) then
                     BINARYOUT <= CONV_SIGNED(BINARYIN,sbsize+2);
              elsif (COEFSIGN = 0 or DATASIGN = 0) then
                     BINARYOUT <= (others => '0');
              else
                     BINARYOUT <= - CONV_SIGNED(BINARYIN,sbsize+2);
              end if;

       end process;

end behaviour;
```

A.3.12accum.vhd

This is the accumulator used to sum the results of this MAC cell with the results of all previous results.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

```
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

entity accum is
       generic(asize, sbsize: INTEGER);
       port(ACCUMIN: in SIGNED(asize downto 0);
       TAPIN: in SIGNED(sbsize+1 downto 0);
       ACCUMOUT: out SIGNED(asize downto 0) );
end accum;

architecture behaviour of accum is
begin

       process(ACCUMIN,TAPIN)
  begin

              ACCUMOUT <= CONV_SIGNED(ACCUMIN + TAPIN,asize+1);

       end process;

end behaviour;
```

Appendix B

VHDL Code for 2 Digit DBNS FIR Filter

B.1 Introduction

The code in this appendix is a modification of the code in Appendix A. The modification was performed by Jonathan Eskritt, in order to convert the code to a two digit DBNS architecture. Only the code that differs from the code in Appendix A is listed here.

B.2 MAC Cell

All the changes occurred in the MAC cell. The binary to DBNS conversion ROM, used for the first stage of the filter, remains the same since the data does not change for a two digit filter system. The third stage, the output accumulator, again stays the same, though more are needed. Only three files from Appendix A need to be changed for synthesis. This is due to designing the code around a globally sourced parameters file. Changes to the parameters file changes the bus and bitwidths throughout the design.

B.2.1 parameters.vhd

The parameter file changed to reflect the smaller bitwidths required for the two digit DBNS filter stem

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

package parameters is

constant dsize : INTEGER := 9;
constant dbsize1 : INTEGER := 4;
constant dtsize1 : INTEGER := 3;
constant dbsize2 : INTEGER := 4;
constant dtsize2 : INTEGER := 3;
constant cbsize : INTEGER := 4;
constant ctsize : INTEGER := 3;
constant dbsize : INTEGER := 4;
constant dtsize : INTEGER := 3;
constant mbsize : INTEGER := 5;
constant mtsize : INTEGER := 4;
constant lbsize : INTEGER := 5;
constant ltsize : INTEGER := 9;
constant absize : INTEGER := 9;
constant sbsize : INTEGER := 12;
constant asize : INTEGER := 18;
constant osize : INTEGER := 15;

constant taps : INTEGER :=  53;

end parameters;
```

B.2.2 ternarylut.vhd

The DBNS to binary conversion uses a much smaller ROM. Not only are there less ROM addresses, but the word size decreased as well. The ternary look-up table code had to be changed to reflect these differences.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.all;

library DBNS;
use DBNS.threerom.all;

entity ternarylut is
       generic(mtsize,ltsize,lbsize: INTEGER);
       port(TERNARYIN: in SIGNED(mtsize downto 0);
       TERNARYOUT: out UNSIGNED(ltsize downto 0);
       BINARYOUT: out SIGNED(lbsize downto 0) );
end ternarylut;

architecture behaviour of ternarylut is
```

```
begin

	process(TERNARYIN)
	variable index: INTEGER;
	begin

		index := CONV_INTEGER(TERNARYIN);

		TERNARYOUT <= '1' &
CONV_UNSIGNED(threerom_32_15(index)(ltsize+lbsize downto lbsize+1),ltsize);
		BINARYOUT <= CONV_SIGNED(threerom_32_15(index)(lbsize downto
0),lbsize+1);

	end process;

end behaviour;
```

B.2.3 threerom.vhd

The data in the ROM is not truncated in this listing. This is because the DBNS to binary conversion ROM is very small in two digit DBNS. In the hybrid design the ROM had 1024 entries, each using 20 bits.

```
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

package threerom is
	subtype ROM_WORD_15 is UNSIGNED(14 downto 0);
--	  subtype ROM_RANGE_32 is STD_LOGIC_VECTOR(4 downto 0);
	subtype ROM_RANGE_32 is INTEGER range -16 to 15;
	type ROM_TABLE_32_15 is array (ROM_RANGE_32'low to ROM_RANGE_32'high) of
ROM_WORD_15;
--	  type ROM_TABLE_32_15 is array (0 to 31) of ROM_WORD_15;

	constant threerom_32_15: ROM_TABLE_32_15 := (

"100011110011101",
"001010110011111",
"110000001100000",
"010100001100010",
"111110010100011",
"011110101100101",
"000111000100111",
"101010100101000",
"001111111101010",
"110111110101011",
"011001111101101",
"000011011101111",
"100101001110000",
"001011110110010",
"110001110110011",
"010101010110101",
```

```
"0000000000110111",
"1000000000111000",
"0010000000111010",
"1011000000111011",
"0100010000111101",
"1110011000111110",
"0110110010000000",
"0001000100000010",
"1001101000000011",
"0011001110000101",
"1100110100000110",
"0101100110010000",
"0000001100010010",
"1000010100010011",
"0010001110011101",
"1011010110011110"
);

end threerom;
```

Appendix C

C Code for Mapping into DBNS

C.1 Introduction

In Chapter 3, the concept of mapping floating point numbers directly into a DBNS representation was introduced. There were two methods that were discussed: an optimal and a modified Greedy Algorithm. These two methods were performed through the use of two programs written in C. The program listings are presented in this appendix.

C.2 Optimal Mapping Listing

This program was originally written by Roberto Muscedere, but was modified by Jonathan Eskritt to restrict the mapping range in order to prevent overflow conditions. The program generates the complete mapping range and sorts the list to reduce the mapping time.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

typedef struct _table {
  double value;
  int binexp, ternexp;
 } table;
```

```
typedef struct _total {
  double value;
  signed char s1,s2;
  int ind1, ind2;
 } total;

static int tablecompare(table *i, table *j)
{
  if((i->value) > (j->value))
    return(1);
  else if((i->value) < (j->value))
    return(-1);
  else
    return(0);
}

static int totalcompare(total *i, total *j)
{
  if((i->value) > (j->value))
    return(1);
  else if((i->value) < (j->value))
    return(-1);
  else
    return(0);
}

void binout(int value, int size)
{
       int i=value;
       int j,k;

       k=1<<j;

       if (value<0) i=value+k;

       for (j=size-1;j>=0;j--)
       {
              if ( i & (1<<j) ) printf("1");
              else printf("0");
       }
}

int main(int argc, char *argv[])
{
       table *value1,*value2;
       total *value;
       int numvals,numvals1,numvals2;
       int binbits1, ternbits1, binbits2, ternbits2;
       int i1, j1, i2;
       int count,count1,count2,i;
       int binlim1, ternlim1, binlim2, ternlim2;
       int of1h, of2h; //overflow prevention limits
       int plus=0,minus=0,one=0;
       int mode=0;
       double z;
       double minimum=0,maximum=16000;
       double uplimit=1<<24,downlimit=1>>24;
       double b,best,worst;
       int    bestindex;
```

```
	int    incfactor;
	int fz,sz;
	FILE *infile=stdin;
	char buffer[85];

	if (argc!=8)
	{
		fprintf(stderr,"Usage: %s b1 t1 over1 b2 t2 over2
mode\n\n",argv[0]);
		fprintf(stderr,"b1    - Bits for first digit binary index\n");
		fprintf(stderr,"t1    - Bits for first digit ternary index\n");
		fprintf(stderr,"over1 - max number to be added to 1st digit\n");
		fprintf(stderr,"b2    - Bits for second digit binary index\n");
		fprintf(stderr,"t2    - Bits for second digit ternary index\n");
		fprintf(stderr,"over2 - max number to be added to 2nd digit\n");
		fprintf(stderr,"mode - 0, first digit only\n");
		fprintf(stderr,"       1, first and positive second digits
only\n");
		fprintf(stderr,"       2, first and negitive second digits
only\n");
		fprintf(stderr,"       3, first and all second digits\n");

		return 1;
	}

  binbits1 = atoi(argv[1]);
  ternbits1 = atoi(argv[2]);
  of1h=atoi(argv[3]);

  binbits2 = atoi(argv[4]);
  ternbits2 = atoi(argv[5]);
  of2h=atoi(argv[6]);

	mode = atoi(argv[7]);

  numvals1 = pow(2, (binbits1+ternbits1))/2;
  binlim1 = pow(2, binbits1-1);
  ternlim1 = pow(2, ternbits1-1)-of1h;

  numvals2 = pow(2, (binbits2+ternbits2))/2;
  binlim2 = pow(2, binbits2-1);
  ternlim2 = pow(2, ternbits2-1)-of2h;

	incfactor = numvals1 = numvals1/32;

	fprintf(stderr,"First Digit Allocated    : %d\n",numvals1);

  value1 = (table *)malloc(sizeof(table)*numvals1);

	if (value1==NULL) return 1;

  count1=0;
	for(i1=-binlim1; i1<binlim1; i1++)
	   for(j1=-ternlim1; j1<ternlim1; j1++)
	    {
			value1[count1].value = exp(i1*log(2)+j1*log(3));
				if ( (value1[count1].value <= uplimit) &&
(value1[count1].value >= downlimit) )
				{
```

```
                         value1[count1].binexp = i1;
                     value1[count1].ternexp = j1;
                              count1++;
                              if (count1==numvals1)
                              {
                                     numvals1+=incfactor;
                                     value1=(table *)real-
loc(value1,sizeof(table)*numvals1);
                                     if (value1==NULL)
                                     {
                                            fprintf(stderr,"Out of mem-
ory.\n");
                                            return 1;
                                     }
                              }
                       }
              }

       fprintf(stderr,"First Digit Used          : %d\n",count1);

  value1 = (table *)realloc(value1,sizeof(table)*count1);

       incfactor = numvals2 = numvals2/32;

       fprintf(stderr,"Second Digit Allocated    : %d\n",numvals2);

  value2 = (table *)malloc(sizeof(table)*numvals2);

       if (value2==NULL) return 1;

  count2=0;
  for(i1=-binlim2; i1<binlim2; i1++)
    for(j1=-ternlim2; j1<ternlim2; j1++)
    {
      value2[count2].value = exp(i1*log(2)+j1*log(3));
                    if ( (value2[count2].value <= uplimit) &&
(value2[count2].value >= downlimit) )
                    {
             value2[count2].binexp = i1;
             value2[count2].ternexp = j1;
                           count2++;
                           if (count2==numvals2)
                           {
                                  numvals2+=incfactor;
                                  value2=(table *)real-
loc(value2,sizeof(table)*numvals2);
                                  if (value2==NULL)
                                  {
                                         fprintf(stderr,"Out of memory.\n");
                                         return 1;
                                  }
                           }
                    }
    }

       fprintf(stderr,"Second Digit Used         : %d\n",count2);

  value2 = (table *)realloc(value2,sizeof(table)*count2);
```

```
     fprintf(stderr,"Sorting First Digit Values.\n");
qsort((char *) value1, count1, sizeof(table), *tablecompare);

     fprintf(stderr,"Sorting Second Digit Values.\n");
qsort((char *) value2, count2, sizeof(table), *tablecompare);

     fz=-1;
     for(i1=0;i1<count1;i1++)
     {
            if (value1[i1].binexp==0 && value1[i1].ternexp==0)
            {
                   fz=i1;
                   break;
            }
     }
     if (fz==-1) return 1;

     sz=-1;
     for(i1=0;i1<count2;i1++)
     {
            if (value2[i1].binexp==0 && value2[i1].ternexp==0)
            {
                   sz=i1;
                   break;
            }
     }
     if (sz==-1) return 1;

numvals = (count1*count2);
     incfactor = numvals = numvals/8;
     fprintf(stderr,"Combined Digits Allocated: %d\n",numvals);

value = (total *)malloc(sizeof(total)*numvals);

     count=0;
     for(i1=0;i1<count1-1;i1++)
     {
            b = value1[i1].value;
            if (b <= maximum && b >= minimum)
            {
                   value[count].value=value1[i1].value;
                   value[count].s1=1;
                   value[count].s2=0;
                   value[count].ind1=i1;
                   value[count].ind2=sz;
                   count++;
                   if (count==numvals)
                   {
                          numvals+=incfactor;
                          value=(total *)realloc(value,sizeof(total)*numvals);
                          if (value==NULL)
                          {
                                 fprintf(stderr,"Out of memory.\n");
                                 return 1;
                          }
                   }
            }
            one++;
            if (mode==3 || mode==1)
```

```
            {
                    for (i2=0;i2<count2;i2++)
                    {
                            b=(double)(value1[i1].value+value2[i2].value);
                            if (b <= maximum && b >= minimum)
                            {
                                    value[count].value=b;
                                    value[count].s1=1;
                                    value[count].s2=1;
                                    value[count].ind1=i1;
                                    value[count].ind2=i2;
                                    count++;
                                    if (count==numvals)
                                    {
                                            numvals+=incfactor;
                                            value=(total *)real-
loc(value,sizeof(total)*numvals);
                                            if (value==NULL)
                                            {
                                                    fprintf(stderr,"Out of mem-
ory.\n");
                                                    return 1;
                                            }
                                    }
                                    plus++;
                            }
                    }
            }
            if (mode==3 || mode==2)
            {
                    for (i2=0;i2<count2;i2++)
                    {
                            b=(double)(value1[i1].value-value2[i2].value);
                            if (b <= maximum && b >= minimum)
                            {
                                    value[count].value=b;
                                    value[count].s1=1;
                                    value[count].s2=-1;
                                    value[count].ind1=i1;
                                    value[count].ind2=i2;
                                    count++;
                                    if (count==numvals)
                                    {
                                            numvals+=incfactor;
                                            value=(total *)real-
loc(value,sizeof(total)*numvals);
                                            if (value==NULL)
                                            {
                                                    fprintf(stderr,"Out of mem-
ory.\n");
                                                    return 1;
                                            }
                                    }
                                    minus++;
                            }
                    }
            }
    }
```

```
	i1=count;
	for(i=0;i<i1;i++)
	{
		value[count].value=-value[i].value;
		value[count].s1=-value[i].s1;
		value[count].s2=-value[i].s2;
		value[count].ind1=value[i].ind1;
		value[count].ind2=value[i].ind2;
		count++;
		if (count==(numvals-1))
		{
			numvals+=incfactor;
			value=(total *)realloc(value,sizeof(total)*numvals);
			if (value==NULL)
			{
				fprintf(stderr,"Out of memory.\n");
				return 1;
			}
		}
	}
	value[count].value=0;
	value[count].s1=0;
	value[count].s2=0;
	value[count].ind1=fz;
	value[count].ind2=sz;
	count++;

	fprintf(stderr,"Single Digit Combinations: %d\n",one);
	fprintf(stderr,"Positive Combinations    : %d\n",plus);
	fprintf(stderr,"Negative Combinations    : %d\n",minus);
	fprintf(stderr,"Combined Digits Used     : %d\n",count);

value = (total *)realloc(value,sizeof(total)*count);

	fprintf(stderr,"Sorting Combined Digit Values.\n");

qsort((char *) value, count, sizeof(total), *totalcompare);

	fprintf(stderr,"Done.\n");

	worst=0;
	while (feof(infile)==0)
	{
		fgets(buffer,80,infile);
		if (feof(infile)==0)
		{
			sscanf(buffer,"%le",&z);
			best=1e9;
			bestindex=-1;
			for (i=0;i<count;i++)
			{
				b=fabs(value[i].value-z);
				if ( (b<best) || (b==best && value[i].s2==0))
				{
					best=b;
					bestindex=i;
				}
			}
			if (best>worst)
```

```
                    {
                          worst=best;
                    }

printf("%d\t%d\t%d\t%d\t%d\t%d\t%g\t%g\t%g\n",value[bestindex].s1,value1[value[b
estindex].ind1].binexp,value1[value[bestindex].ind1].ternexp,value[bestindex].s2
,value2[value[bestindex].ind2].binexp,value2[value[bestindex].ind2].tern-
exp,z,value[bestindex].value,best);
              }
        }

        fprintf(stderr,"Worst Error               : %15.15e\n",worst);

  return 0;

}
```

C.3 Modified Greedy Mapping Listing

This program maps the floating point numbers into DBNS numbers using the modified Greedy Algorithm. It also allows for odd bases other than 3.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

typedef struct _table {
  double value;
  int binexp, ternexp;
  int s;
 } table;

static int tablecompare(table *i, table *j)
{
  if((i->value) > (j->value))
    return(1);
  else if((i->value) < (j->value))
    return(-1);
  else
    return(0);
}

void binout(int value, int size)
{
       int i=value;
       int j,k;

       k=1<<j;

       if (value<0) i=value+k;

       for (j=size-1;j>=0;j--)
       {
             if ( i & (1<<j) ) printf("1");
```

```
                else printf("0");
        }
}

int main(int argc, char *argv[])
{
        table *value1,*value2;
        int numvals1,numvals2;
        int binbits1, ternbits1, binbits2, ternbits2;
        int i1, j1;
        int count1,count2,i;
        int binlim1, ternlim1, binlim2, ternlim2;
        int of1h, of2h; //overflow prevention limits
        double z;
        double uplimit=1<<24,downlimit=1>>24;
        double b,best,worst;
        int    bestindex;
        int    incfactor;
        int fz,sz;
//      double p=pow(3,0x132)*pow(2,0x21b);
        FILE *infile=stdin;
        char buffer[85];
        double b2=0,val;
        int sec,tmp,bestindex2,zeroi,base;

        if (argc !=8)
        {
                fprintf(stderr,"Usage: %s b1 t1 over1 b2 t2 over2
base\n\n",argv[0]);
                fprintf(stderr,"b1    - Bits for first digit binary index\n");
                fprintf(stderr,"t1    - Bits for first digit ternary index\n");
                fprintf(stderr,"over1 - max number to be added to 1st digit\n");
                fprintf(stderr,"b2    - Bits for second digit binary index\n");
                fprintf(stderr,"t2    - Bits for second digit ternary index\n");
                fprintf(stderr,"over2 - max number to be added to 2nd digit\n");
                fprintf(stderr,"base - The value of the second base\n");

                return 1;
        }

  binbits1 = atoi(argv[1]);
  ternbits1 = atoi(argv[2]);
  of1h=atoi(argv[3]);

  binbits2 = atoi(argv[4]);
  ternbits2 = atoi(argv[5]);
  of2h=atoi(argv[6]);

  base=atoi(argv[7]);
  fprintf(stderr,"base=%d",base);

  numvals1 = pow(2, (binbits1+ternbits1)/2);
  binlim1 = pow(2, binbits1-1);
  ternlim1 = pow(2, ternbits1-1)-of1h;

  numvals2 = pow(2, (binbits2+ternbits2)/2);
  binlim2 = pow(2, binbits2-1);
  ternlim2 = pow(2, ternbits2-1)-of2h;
```

```
        incfactor = numvals1 = numvals1/32;

        fprintf(stderr,"First Digit Allocated    : %d\n",numvals1);

/*****************************************************
 * generate table of all possible values for 1st digit *
 *****************************************************/

   value1 = (table *)malloc(sizeof(table)*numvals1);

        if (value1==NULL) return 1;

   count1=0;
   for(i1=-binlim1; i1<binlim1; i1++)
     for(j1=-ternlim1; j1<ternlim1; j1++)
     {
       value1[count1].value = exp(i1*log(2)+j1*log(base));
                       if ( (value1[count1].value <= uplimit) &&
(value1[count1].value >= downlimit) )
                       {
              value1[count1].binexp = i1;
              value1[count1].ternexp = j1;
              value1[count1].s=1;
                              count1++;
                              if (count1==numvals1)
                              {
                                     numvals1+=incfactor;
                                     value1=(table *)real-
loc(value1,sizeof(table)*numvals1);
                                     if (value1==NULL)
                                     {
                                            fprintf(stderr,"Out of memory.\n");
                                            return 1;
                                     }
                              }
                       }
     }

  value1 = (table *)realloc(value1,sizeof(table)*count1*2+1);
  count2=count1;
  for (count1=0; count1<count2; count1++)
  {
       value1[count1+count2].value=-value1[count1].value;
       value1[count1+count2].binexp =value1[count1].binexp;
       value1[count1+count2].ternexp =value1[count1].ternexp;
       value1[count1+count2].s=-1;

  }
  count1=count2;
  value1[count1+count2].value=0;
  value1[count1+count2].binexp=0;
  value1[count1+count2].ternexp=0;
  value1[count1+count2].s=0;
  count1=count2*2;

       fprintf(stderr,"First Digit Used          : %d\n",count1+count2);

       incfactor = numvals2 = numvals2/32;
```

```
    fprintf(stderr,"Second Digit Allocated   : %d\n",numvals2);

/*****************************************************
 * generate table of all possible values for 2nd digit *
 *****************************************************/

  value2 = (table *)malloc(sizeof(table)*numvals2);

      if (value2==NULL) return 1;

  count2=0;
  for(i1=-binlim2; i1<binlim2; i1++)
    for(j1=-ternlim2; j1<ternlim2; j1++)
    {
      value2[count2].value = exp(i1*log(2)+j1*log(base));
                     if ( (value2[count2].value <= uplimit) &&
(value2[count2].value >= downlimit) )
                     {
             value2[count2].binexp = i1;
             value2[count2].ternexp = j1;
                         count2++;
                         if (count2==numvals2)
                         {
                                 numvals2+=incfactor;
                                 value2=(table *)real-
loc(value2,sizeof(table)*numvals2);
                                 if (value2==NULL)
                                 {
                                         fprintf(stderr,"Out of memory.\n");
                                         return 1;
                                 }
                         }
                     }
    }
  value2 = (table *)realloc(value1,sizeof(table)*count2*2+1);
  tmp=count2;
  for (count2=0; count2<tmp; count2++)
  {
      value1[count2+tmp].value=-value2[count2].value;
      value1[count2+tmp].binexp =value2[count2].binexp;
      value1[count2+tmp].ternexp =value2[count2].ternexp;
      value1[count2+tmp].s=-1;

  }
  value2[tmp+count2].value=0;
  value2[tmp+count2].binexp=0;
  value2[tmp+count2].ternexp=0;
  value2[tmp+count2].s=0;
  count2=tmp*2;

      fprintf(stderr,"Second Digit Used        : %d\n",count2+count1);

      fprintf(stderr,"Sorting First Digit Values.\n");
  qsort((char *) value1, count1, sizeof(table), *tablecompare);

      fprintf(stderr,"Sorting Second Digit Values.\n");
  qsort((char *) value2, count2, sizeof(table), *tablecompare);

      fz=-1;
```

```
        for(i1=0;i1<count1;i1++)
        {
                if (value1[i1].binexp==0 && value1[i1].ternexp==0)
                {
                        fz=i1;
                        break;
                }
        }
        if (fz==-1) return 1;

        sz=-1;
        for(i1=0;i1<count2;i1++)
        {
                if (value2[i1].binexp==0 && value2[i1].ternexp==0)
                {
                        sz=i1;
                        break;
                }
        }
        if (sz==-1) return 1;

        zeroi=0;
        while(value2[zeroi].value != 0) zeroi++;

        fprintf(stderr,"Done.\n");

        worst=0;
        while (feof(infile)==0)
        {
                fgets(buffer,80,infile);
                if (feof(infile)==0)
                {
                        sscanf(buffer,"%le",&z);
/*****************************
 * find first digit          *
 *****************************/
                        best=1e9;
                        bestindex=-1;
                        sec=1;
                        for (i=0;i<count1;i++)
                        {
                                b=fabs(value1[i].value-z);
                                if ( b<best )
                                {
                                        best=b;
                                        bestindex=i;
                                }
                        }
                        if (best>worst)
                        {
                                worst=best;
                        }
                        if ((value1[bestindex].value - z) < 0) sec=-1;
                        else if ((value1[bestindex].value - z) == 0) sec=0;
                        else sec=1;

/*****************************
 * find second digit         *
 *****************************/
```

```
                    best=1e9;
                    bestindex2=-1;
                    if (sec==0)
                    {
                          bestindex2=zeroi;
                          best=0;
                    }
                    else
                    {
                          b2=0;
                          for (i=0;i<count2;i++)
                          {
                                b=fabs((value1[bestin-
dex].value+value2[i].value)-z);
                                if ( b<best )
                                {
                                      best=b;
                                      bestindex2=i;
                                      b2=1;
                                }
                                b=fabs((value1[bestin-
dex+sec].value+value2[i].value)-z);
                                if ( b<best )
                                {
                                      best=b;
                                      bestindex2=i;
                                      b2=2;
                                }
                          }
                    }
                    if (best>worst)
                    {
                          worst=best;
                    }
                    if (b2 == 2) bestindex += sec;
                    val=value1[bestindex].value + value2[bestindex2].value;
/**************************************************
 * change printf to account for 2 differnet indices *
 **************************************************/

printf("%d\t%d\t%d\t%d\t%d\t%d\t%g\t%g\t%g\n",value1[bestindex].s,value1[bestind
ex].binexp,value1[bestindex].ternexp,value2[bestindex].s,value2[bestindex2].bin-
exp,value2[bestindex2].ternexp,z,val,best);
            }
      }

      fprintf(stderr,"Worst Error              : %15.15e\n",worst);

  return 0;

}
```

Appendix D

Genetic Algorithm Code

D.1 DBNS Coefficient Generation

A genetic algorithm was used early in the research of DBNS filters to generate the filter coefficients. The genetic algorithm, described in Section 3.2.1, was developed by Alfred Lee [13]. Heavy modifications were made that greatly decreased the simulation time.

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>

#define T     1.0    //sampling frequency
#define maxFreq3.1416
#define Npoints47    //# of sampling points
#definemaxFilterOrder 50

/********************/
/* define data type */
/********************/

typedef struct {
      //the highest bit is the sign bit for double base
      unsigned int coef[maxFilterOrder];
      float fitness;
      int absfitness;
} aCHROM;

typedef struct {
      double val;
      int binExp;
      int terExp;
```

```
	int sign;
} DATA;

/********************/
/* Global Variables */
/********************/
float pmut, pcross, errZone;
DATA *coeff;
int Ncoef, Nchrom;
int *ordering;
FILE *inptr, *outptr;
DATA *data;
aCHROM *chrom, *offsprg, Bchrom;
char ob;
int chrom_len, num;
float stopbd,passbd,ALPHA; //either stopbd or ALPHA is used
unsigned int eachbit[32];
unsigned int fromright[32];
unsigned int fromleft[32];
unsigned int valuemask;
unsigned int signmask;
unsigned int wholemask;
int	*scalefactor;
double costab[Npoints+5][100]; // may have to adjust if values are large

/*************************/
/* Procedure and function */
/*************************/
int decodeChrom(int chrom_loc, int coef_loc);

void inittables()
{
	int i,j;

	for (i=0;i<32;i++)
	{
		eachbit[i]=1<<i;
		for (j=0;j<=i;j++)
		{
			fromright[i]|=1<<j;
		}
		fromleft[i] = fromright[i] ^ (unsigned int)0xffffffff;
	}

	float delta_w,w;
	int n;

  delta_w = (float)maxFreq / (float)Npoints;

	for (w = 0,n=0; w < maxFreq + delta_w; w = w + delta_w,n++){
		for ( j = 1; j < Ncoef; j++){
			costab[n][j] = 2.0 * cos(w * T * j);
		}
	}

	return;
}

int ANUMBER(){
```

```
        int i=rand();

        return( i );
}

float UNI(){
        float a = (float)rand() / (float)RAND_MAX;

        return ( a );
}

unsigned int encodeChrom(int p5_loc)
{
        unsigned int echrom=0;
        int i;

        for (i=0; i<chrom_len-1; i++)
        {
                if (p5_loc & eachbit[i]) echrom |= eachbit[i];
        }
        echrom =(unsigned int)((double)echrom*(((double)(1<<(chrom_len-1)))/
((double)num)));
        return (echrom);
}

void presetChrom(int p5_loc){
        int i,j,k,tmp;

        for (i = 0; i < Nchrom; i++){
                for (j = 0; j < Ncoef; j++){
                        chrom[i].coef[j] = 0;
                        offsprg[i].coef[j] = 0;
                }
        }

        for (i = 0; i < Nchrom; i++){
                for (j=0; j<Ncoef-1; j++){
                        if (tmp=(ANUMBER()%3)) tmp = chrom_len/tmp;
                        else tmp = chrom_len/3;
                        for (k=0; k<tmp; k++){
                                chrom[i].coef[Ncoef-j-2] |= eachbit[ANUMBER() %
chrom_len-1];
                        }
                        if (j%2==0) chrom[i].coef[Ncoef-j-2] |= signmask;
                }
                chrom[i].coef[Ncoef-1] = encodeChrom(p5_loc);
        }
}

void InsertBestChrom(int maxi, float maxFit, int mini, float minFit){
        int i, j;

        if ( maxFit > Bchrom.fitness ){
                for (j=0; j < Ncoef; j++){
                        Bchrom.coef[j] = chrom[maxi].coef[j];
                }
                Bchrom.fitness = maxFit;
        } else {
                for (j=0; j < Ncoef; j++){
```

```
                chrom[mini].coef[j] = Bchrom.coef[j];
            }
            chrom[mini].fitness = minFit;
        }
}

void scaleFitness(){
        int i,indexMax,indexMin;
        double minN, maxN;
        static int flag = 0;

        //Find maximum and minimum
        minN = 999.9;
        maxN = -999.9;
        for(i = 0; i < Nchrom; i++){
                if( minN > chrom[i].fitness ){ minN = chrom[i].fitness; indexMin =
i;}
                if( maxN < chrom[i].fitness ){ maxN = chrom[i].fitness; indexMax =
i;}
        }

        if ( flag != 0 )
                InsertBestChrom(indexMax,maxN,indexMin,minN);
        else flag++;

        //Perform scaling
        for( i = 0; i < Nchrom; i++)
        {
        chrom[i].absfitness = (int) ceil( 80.0 * (chrom[i].fitness - minN)/(maxN -
minN) ) + 5;
         }
}


void DOcrossover(int chrom1_loc, int chrom2_loc, int offsprg1_loc, int
offsprg2_loc, int coef_loc){
        int apoint, i;

        if ( UNI() < pcross ){
                apoint = ANUMBER() % (chrom_len - 1);

// High cross
//                      offsprg[offsprg1_loc].coef[coef_loc] = (
(chrom[chrom1_loc].coef[coef_loc] & fromright[apoint]) |
(chrom[chrom2_loc].coef[coef_loc] & fromleft[apoint] & wholemask) ) ^ wholemask;
//                      offsprg[offsprg2_loc].coef[coef_loc] = (
(chrom[chrom2_loc].coef[coef_loc] & fromright[apoint]) |
(chrom[chrom1_loc].coef[coef_loc] & fromleft[apoint] & wholemask) ) ^ wholemask;
/*
                        offsprg[offsprg1_loc].coef[coef_loc] = (
(chrom[chrom1_loc].coef[coef_loc] & fromright[apoint]) |
(chrom[chrom2_loc].coef[coef_loc] & fromleft[apoint] & wholemask) );
                        offsprg[offsprg2_loc].coef[coef_loc] = (
(chrom[chrom2_loc].coef[coef_loc] & fromright[apoint]) |
(chrom[chrom1_loc].coef[coef_loc] & fromleft[apoint] & wholemask) );
*/
// Low cross
```

```
                        offsprg[offsprg1_loc].coef[coef_loc] =
(chrom[chrom2_loc].coef[coef_loc] & fromright[apoint]) |
(chrom[chrom1_loc].coef[coef_loc] & fromleft[apoint] & wholemask);
                        offsprg[offsprg2_loc].coef[coef_loc] =
(chrom[chrom1_loc].coef[coef_loc] & fromright[apoint]) |
(chrom[chrom2_loc].coef[coef_loc] & fromleft[apoint] & wholemask);
        }
}


void DOmutation(int loc){
        int i,j;

        for (i=0; i < Ncoef; i++){
                for (j = 0; j < chrom_len; j++){
                  if ( UNI() < pmut )
                                offsprg[loc].coef[i] ^= eachbit[j];
                }
        }
}

void reproduce(){
        int i, count,j;
        float rawfitness, total, sum;

        total = -0.5;
        for( j = 0; j < Nchrom; j++) total = total + (float)chrom[j].absfitness;
        for( i = 0; i < Nchrom; i++){
                rawfitness = UNI() * total;
                j = ANUMBER() % Nchrom;
                sum = (float)chrom[j].absfitness;
                count = j;
                while(sum < rawfitness){
                        count++;
                        if (count == Nchrom) count = 0;
                        sum = sum + (float)chrom[count].absfitness;
                }
                ordering[i] = count;
        }
}

//decodeChrom
int decodeChrom(int chrom_loc, int coef_loc){
        int temp;

        temp = scalefactor[chrom[chrom_loc].coef[coef_loc] & valuemask];

        if (chrom[chrom_loc].coef[coef_loc] & signmask)
                return( -temp );
        else return(temp);
}



float fitness(){
        int j, count,n;
        float err,w, delta_w;
        double H, temp;
        float return_val, max_err;
```

```
    delta_w = (float)maxFreq / (float)Npoints;

    err = 0;
       max_err = 0;
       for (w = 0,n=0 ; w < maxFreq + delta_w; w = w + delta_w,n++){

              //at one frequency of the first half symmetry coefficients
        count = 0;

              H = coeff[Ncoef - 1].val;
              for ( j = 1; j < Ncoef; j++){
//                   H = H + 2.0 * coeff[Ncoef - j - 1].val * cos(w * T * j);
                     H = H + coeff[Ncoef - j - 1].val * costab[n][j];
              }

              H = fabs(H);

              //minmax or LMS
              if ( ob == '1'){

                     //exp decay transision region
                     if ( w < passbd )
                            temp = fabs( 1.0 - H);
                     else
                            temp = fabs( exp(-1.0*ALPHA*(w - passbd)) - H );

                     if (temp > errZone) err = err + temp - errZone;

              } else if(ob == 'm') {

                     //get the magnitude response error
                     if ( w < passbd )
                            err = fabs( 1.0 - H );
                     else if ( w > stopbd  )
                            err = fabs(H);
                     else err=0;

                     if (err < errZone ) err = 0;
                     else err = err - errZone;

                     if( max_err < err ) max_err = err;

              } else {
                     printf("\ncannot get the objective function\n");
                     exit(-1);
              }

       }

       if (ob == '1') return_val  =  1.0 / err;
       else return_val  =  1.0 / max_err;

       return( return_val );

}

void GETchromfit(){    //passing decoded coefficients to fitness
       int j,i, loc;
```

```
        for( j = 0; j < Nchrom; j++){
                for( i=0; i < Ncoef; i++){
                        loc = decodeChrom(j,i);
                        if (loc < 0){
                                coeff[i].val = -data[-loc].val;
                                coeff[i].binExp = data[-loc].binExp;
                                coeff[i].terExp = data[-loc].terExp;
                                coeff[i].sign = -1;
                        }else {
                                coeff[i].val = data[loc].val;
                                coeff[i].binExp = data[loc].binExp;
                                coeff[i].terExp = data[loc].terExp;
                                coeff[i].sign = 1;
                        }

                        }
                chrom[j].fitness = fitness();
    }
}

int decodeBChrom(int coef_loc){
        int temp;

        temp = scalefactor[Bchrom.coef[coef_loc] & valuemask];

        if (Bchrom.coef[coef_loc] & signmask)
                return( -temp );
        else return(temp);
}

void report(unsigned int seed){
        int i,j,sign, temp;
        float maxfitness;

        /* open output file */
        outptr = fopen( "output.dat","w+");
        if ( outptr == NULL){ printf("cannot open output file"); exit(-1); }

        /* decode the coefficient */
        for( i=0; i < Ncoef; i++){
                temp = decodeBChrom(i);
                if (temp < 0){
                        coeff[i].val = - data[-temp].val;
                        coeff[i].binExp = data[-temp].binExp;
                        coeff[i].terExp = data[-temp].terExp;
                        coeff[i].sign = -1;
                }else {
                        coeff[i].val = data[temp].val;
                        coeff[i].binExp = data[temp].binExp;
                        coeff[i].terExp = data[temp].terExp;
                        coeff[i].sign = 1;
                }
                Bchrom.fitness = fitness();
        }

        /* printout the coefficient and exponents values */
        for(i = 0; i < Ncoef; i++){
```

```
                fprintf(outptr, "%5d# %d %d",coeff[i].sign, coeff[i].binExp,
coeff[i].terExp);
                fprintf(outptr, "\n");
        }
        for(i = 1; i < Ncoef; i++){
//              if ( bestChrom.coefB2[Ncoef - i - 1][NbitB2 - 1].bit != 0 ) sign =
-1;
//              else sign = 1;
                fprintf(outptr, "%5d# %d %d",coeff[Ncoef - i - 1].sign, coeff[Ncoef
- i - 1].binExp, coeff[Ncoef - i - 1].terExp);
                fprintf(outptr,"\n");
        }

    fprintf(outptr,"\n");
        for(i = 0; i < Ncoef; i++){
                fprintf(outptr,"\n%18.15f",coeff[i].val);

        }
    for ( j = 1; j < Ncoef; j++){
                fprintf(outptr,"\n%18.15f",coeff[Ncoef - j - 1].val);
    }

        fprintf(outptr, "\n\nThe best fitness is %f\n", Bchrom.fitness);
        fprintf(outptr, "random seed is %lu\n", seed);
        fclose(outptr);
}

int ilog2(int x)
{
  int tmp,fix,loop;

  fix=0;
  tmp=x;
  if (tmp>0)
  {
    loop=0;
    while (tmp!=1)
    {
      tmp=tmp>>1;
      loop++;
    }
    if (x > (1<<loop)) fix=1;
  }
  else loop=-1;
  if (fix==1) loop++;
  return (loop);
}


/**********************
** Main Program *******
**********************/
int main(int argsize, char **arg){
        int Citer, chromctr, coefctr, MaxIter,index1,index2;
        aCHROM *temp;
         int p5_loc;
        unsigned int seed;
//      arg[1] - m for minimax or l for LMS
        if ( argsize != 10 )
```

```
        {
                fprintf(stderr,"\nwrong number of arguments!!\n\n");
                fprintf(stderr,"Usage: %s ob MaxIter Nchrom pmut pcross Ncoef
errZone pass stop\n\n", arg[0]);
                fprintf(stderr,"ob - 'l' for Least Mean Squares, 'm' for mini-
max\n");
                fprintf(stderr,"MaxIter - # of iterations     Nchrom - population
size\n");
                fprintf(stderr,"pmut - mutation probability   pcross - crossover
prob.\n");
                fprintf(stderr,"Ncoef - # of coeffiecents     errZone - max pass-
band/stopband err \n");
                fprintf(stderr,"pass - passband edge normalized to sampling rate of
2pi\n");
                fprintf(stderr,"stop - if ob=l transition band decay from passband
exp(-stop(w-pass))\n");
                fprintf(stderr,"      if ob=m stopband edge normalized to sampling
rate of 2pi\n\n");
                exit(-1);
        }

        ob           = *arg[1];    // objective function LMS or minimax
        MaxIter      = atoi(arg[2]); // # of generations
        Nchrom       = atoi(arg[3]); // population size
        pmut         = (float)atof(arg[4]);// mutation probability
        pcross       = (float)atof(arg[5]);// crossover probability
        Ncoef        = atoi(arg[6]);// number of coefficents
        errZone      = (float)atof(arg[7]);// max passband and stopband error
ripple
        passbd       = (float)atof(arg[8]);  // passband edge normalized to sam-
pling rate of 2pi
        if (ob=='m') stopbd=(float)atof(arg[9]);// stopband edge normalized to
sampling rate of 2pi
        else ALPHA=(float)atof(arg[9]);// transition band decay exp(-ALPHA(w-
passbd))

        inittables();

        seed=(unsigned)time(NULL);
        srand(seed);

        /* restriction on the population size */
        if((Nchrom % 2) != 0 ) Nchrom--;

        /* create space for chromosome */
        chrom = new aCHROM[Nchrom];
        if (chrom == NULL) {printf("don't have enough space, chrom\n"); exit(-1);}

        /* create space for offsprg */
        offsprg = new aCHROM[Nchrom];
        if (offsprg == NULL) {printf("don't have enough space, offsprg\n"); exit(-
1);}

        /* create space for ordering */
        ordering = new int[Nchrom];
        if (ordering == NULL) {printf("don't have enough space, ordering\n");
exit(-1);}

        /* create space for coeff for fitness evaluation use */
```

```
        coeff = new DATA[Ncoef];
        if ( coeff == NULL){ printf("\nNot enough memory for *coeff"); exit(-1); }

        /* read all available data */
        inptr = fopen( "dbns.dat" ,"r");
        if (inptr == NULL) { printf("cannot open readin file\n"); exit(-1); }
        fscanf(inptr,"%d",&num);
        chrom_len = ilog2(num)+1; // log2(num) +1 round up (+1 for sign)
printf("chrom length is %d\n",chrom_len);
        signmask = eachbit[chrom_len-1];
        valuemask = fromright[chrom_len-2];
        wholemask = fromright[chrom_len-1];
        data = new DATA[num];
        scalefactor = new int[eachbit[chrom_len-1]];

        if (data == NULL) {printf("don't have enough space, data\n"); exit(-1);}

        int i;
        for (i=0; i < num; i++)
        {
                fscanf(inptr,"%le\t%d\t%d",&data[i].val,&data[i].bin-
Exp,&data[i].terExp);
                if (data[i].binExp==-1 && data[i].terExp==0)
                {
                        p5_loc = i;
                        printf("0.5 loc is %d giving
%le\n",p5_loc,data[p5_loc].val);
                }
        }
        fclose(inptr);

        for (i=0;i<eachbit[chrom_len-1];i++)
                scalefactor[i]=(int)(( (double)i / (double)eachbit[chrom_len-1] ) *
num);

        //get the random chromosome
        presetChrom(p5_loc);

        //calcuate the fitness of each chromosome
        GETchromfit();
        Bchrom.fitness = -1.0;
        Bchrom.absfitness = -1;
        Citer = 0;
        scaleFitness();
        while ( (Citer < MaxIter) ){
                Citer++;
                //find Noffsprg and duplicate the chrom to offsprg
                reproduce();
                //crossover, mutation
                for (chromctr = 0; chromctr < Nchrom; chromctr = chromctr + 2){
                        index1 = ordering[chromctr];
                        index2 = ordering[chromctr + 1];
                        for (coefctr = 0; coefctr < Ncoef; coefctr++)
                                DOcrossover(index1, index2, chromctr, chromctr + 1,
coefctr);
                        DOmutation(chromctr);
                        DOmutation(chromctr + 1);
                }
```

```
            /* exchange pointer */
            temp = chrom;
            chrom = offsprg;
            offsprg = temp;

            //calcuate fitness of each offsprg
            GETchromfit();
            scaleFitness();
            if ((float)(1/Bchrom.fitness)==0.0) {report(seed); exit(0);}
            if ((Citer % 30) == 0 ) printf("\n%d%f", Citer, Bchrom.fitness );
      }

      printf("\n");
      report(seed);

      delete[] chrom;
      delete[] data;
      delete[] coeff;
      delete[] offsprg;
      delete[] scalefactor;

      return(0);
}
```

Appendix E

A Brief Tutorial in High Level Chip Design

E.1 Introduction

This appendix provides a brief introduction into high level microchip design. It is assumed that the microchips will be designed using Hardware Descriptive Languages (HDL) and compiled into their hardware representation. This is a complex topic and this tutorial should be supplemented using the many books and online documentation available to the students in the VLSI Laboratory.

The first step to high level chip design is a working knowledge of HDLs. There are many tutorials available on the internet, a quick search for "vhdl tutorial" or "verilog tutorial" should garner many results. I also recommend the book "*HDL Chip Design*" from Doone Publishers [30] (available at www.doone.com). The book has extensive examples, which are done in both VHDL and Verilog, and compares the strengths and weaknesses of both HDLs.

E.2 System Design

If the designer has a working knowledge of HDLs then one can convert a system specification into the Hardware Descriptive

Language. This can be done in two ways. The first is to start with a high level behaviour description, and test to see if the description meets the specification. If the description meets the specification, the system can be divided into smaller more detailed blocks and simulated again. This is repeated until the overall system is described at a level that is synthesizable. This is termed as "top-down" design.

The second method is to describe the blocks that are needed and simulate them in order to test if they meet specification. These blocks are then pieced together to form the complete system. This is termed "bottom-up" design.

In practice a mix of both top-down and bottom-up approaches are usually performed. For example, in an attempt at bottom-up design, it is usually better to implement a structure as a sub-block. A top-down design approach can use this block developed from a bottom-up design.

E.2.1 Choice of HDLs

Choosing the right HDL for a task is important. The purpose of the HDL description, and the tools that are available determine the best language to be used. In general, it is usually better to use Verilog for use with Cadence tools (Cadence once owned Verilog and used it as its proprietary language), or for simulation, and VHDL for synthesis. This convention is used because Verilog is not designed for system level description, and VHDL handles binary arithmetic in a better manner, such as understanding sign extension in signed binary arithmetic.

E.2.2 Putting the Design Together

Assembling the individual blocks into a system can be a long and error prone task to perform in code. There are two different ways to ease the combining of sub blocks into a system. The first, usually used for Verilog, is assembling the blocks through a graphical schematic capture program, such as Cadence Design Frameworks II. Each component is converted into a symbol with input and output pins. These symbols can then be connected

together, into the system, using graphical interface tools. The completed system can then be netlisted into a complete Verilog description.

The second method can only be used with VHDL. This method is excellent for producing repetitive, modular systems. The use of the `generate` command in VHDL allows the connection of numerous component blocks in just a few lines of code. All that is needed is to specify the components and the connections.

Example E.1 The following code will generate a filter with an arbitrary number of taps

```
GEN: for i in 0 to taps-1 generate
  U: ltap
    generic map ( cbsize, ctsize, dbsize, dtsize, mbsize, mtsize,
                  lbsize, ltsize, absize, sbsize, asize )
    port map ( ICLK(56-i), ICLK(56-i+1), ICRESET(56-i), ICRESET(56-
         i+1),ILRESET(56-i),ILRESET(56-i+1),IDATABIN(i),IDATABIN(i+1),
         IDATASIGN(i), IDATASIGN(i+1), IDATATERN(i), IDATATERN(i+1),
         IACCUM(i), IACCUM(i+1), ILCLK(56-i), ILCLK(56-i+1), ICOEF,
         BIN(i), ICOEFSIGN(i), ICOEFTERN(i), ICOEFBIN(i+1), ICOEF
         SIGN(i+1), ICOEFTERN(i+1) );
end generate GEN;
```

E.3 Synthesizing the System

Once the system is fully described using a HDL, the next step is to synthesize the design into hardware. The tool used for this is Synopsys, which can compile both VHDL and Verilog. A fully detailed tutorial in synthesizing systems, with an example producing a 16 bit multiplier, is available from CMC. The latest version, and updates is available at:

http://www.cmc.ca/Training/Digital_Flow/cmc_digflow.html

Because the CMC Digital Flow document covers the synthesizing of a HDL description, only the salient points will be covered here.

1. The first step is to create a working directory in which the design files are to reside, for example:

   ```
   mkdir synopsys
   ```

2. The working directory requires 2 files: **.synopsys_dc.setup** and **.synopsys_vss.setup**. .synopsys_vss.setup maps the compiled Synopsys libraries to system directories. The file has the format:

   ```
   Library: Directory
   ```

   It is generally good practice to have the library name and directory names in an identical format. For example the .synopsys_vss.setup file for the filter designed in this thesis was:

   ```
   DBNS: ./DBNS
   ```

   A sample .synopsys_dc.setup file is provided by CMC. This file is used to set up the location of the standard cell libraries that will be used in Synopsys. The contents of .synopsys_dc.setup are explained by the comments in the file. Something similar must appear in directory from which Synopsys is run. The CMC provided file is:

```
/* Canadian Microelectronics Corporation
 * Sample .synopsys_dc.setup file, for use with CMC DSM labs
 * November 1, 1999
 *
 * Library and Search Path variables assume links are in place so
 * $SYNOPSYS/cmc/cmosp35 points to the libraries in this design kit
 * which are compiled for the proper version of Synopsys
 */

search_path = {.}
search_path = search_path + {synopsys_root + /libraries/syn}
search_path = search_path + {synopsys_root + /cmc/cmosp35/syn}


/* Use only one of the following sets of library statements!! */

/* For Black Box (bbox) libraries */
link_library = "* tcb773pwc.db tpd773pnwc.db wsram.db"
target_library = {tcb773pwc.db tpd773pnwc.db wsram.db}
symbol_library = {tcb773p.sdb tpd773pn.sdb}

/* For the CMC developed W-Cells */
/*
link_library = "* wcells.db wsram.db"
```

```
target_library = {wcells.db wsram.db}
symbol_library = {wcells.sdb}
vhdlout_use_packages = {IEEE.std_logic_1164, wcells.components}
*/

/* Assume there is a ./Work directory */
/* define_design_lib work -path Work */
define_design_lib dbns -path DBNS

/* Try and make names compatible with Cadence dfII, from Preview man. */
bus_naming_style = "%s_%d_"
verilogout_no_tri = "true"
define_name_rules preview -allowed "A-Za-z0-9_"
/* Preview man. page says set the verilogout_single_bit = true, but to
   get the sram cells to work you may need false. */
verilogout_single_bit = "true"


/* Some usefull scripts. */
view_script_submenu_items = \
   {"Remove All Designs","remove_design find(design \"*\")", \
    "Save All Designs", "write find(design \"*\") -out save.db", \
    "set_dont_touch All Designs", "set_dont_touch find(design \"*\")", \
    "Remove dont_touch All Designs", \
       "remove_attribute find(design \"*\") dont_touch", \
    "Remove Unconnected Ports", \
      "remove_unconnected_ports -blast_buses find(-hierarchy cell, \"*\")",\
    "Fix Multiple Ports (on selected hierarchy)", \
       "set_fix_multiple_port_nets -all", \
    "Change Names for Preview", \
       "change_names -rules preview -hierarchy > change_names.out" }

/* use the following to include the TSMC name changing script */
include /CMC/kits/cmosp35/samples/TSMC_naming_rule.script
```

3. A directory must be created for each entry in the .synopsys_vss.setup file, and each directory name must be the same as those listed in .synopsys_vss.setup

4. Once the previous steps are taken care of, it is time to start Synopsys. To start Synopsys use the one of the following commands:

```
dc_shell
```

for a command line interface (no graphical interfaces) to Synopsys, or:

```
design_analyzer &
```

for the graphical interface version of Synopsys. After design_analyzer is started use the **Setup -> Command Window** menu to bring up an interface to type in commands.

If an error, such as:

```
dc_shell: Command not found
```

is encountered when trying to run one of the above commands, it means that the Synopsys tools are not in the current path and may need to be setup. At the University of Windsor this is done by executing the following command:

```
setup synopsys
```

5. In order to produce a more structured and reproducible compilation in Synopsys, a compile script can be made. This is important if the code used has errors or if a mistake is made. The script makes it easy to reproduce the compile quickly and without missing any steps. A compile script reads in each HDL file, saves them in Synopsys' internal format and then compiles the description into hardware and produces a Verilog output file for use with Cadence tools. An example of such a compile script would be:

```
remove_design "*"
analyze -f vhdl -lib DBNS parameters.vhd
analyze -f vhdl -lib DBNS packages.vhd
analyze -f vhdl -lib DBNS accum.vhd
analyze -f vhdl -lib DBNS signcorrect.vhd
analyze -f vhdl -lib DBNS bshifter.vhd
analyze -f vhdl -lib DBNS binaryadder.vhd
analyze -f vhdl -lib DBNS ternarylut.vhd
analyze -f vhdl -lib DBNS multiplier.vhd
analyze -f vhdl -lib DBNS atap.vhd
analyze -f vhdl -lib DBNS rlatch.vhd
analyze -f vhdl -lib DBNS ctap.vhd
analyze -f vhdl -lib DBNS ltap.vhd
analyze -f vhdl -lib DBNS ltap_export.vhd
analyze -f vhdl -lib DBNS binary4dbnslut.vhd
analyze -f vhdl -lib DBNS filter.vhd
elaborate -lib DBNS ltap_export
current_design "accum_asize18_sbsize12"
compile -map_effort high
current_design "rlatch_wsize1"
compile -map_effort high
current_design "rlatch_wsize3"
compile -map_effort high
current_design "rlatch_wsize4"
compile -map_effort high
current_design "rlatch_wsize18"
compile -map_effort high
current_design "binaryadder_mbsize5_lbsize5_absize9"
compile -map_effort high
current_design "bshifter_ltsize9_absize9_sbsize12"
compile -map_effort high
```

```
current_design "multiplier_cbsize4_ctsize3_dbsize4_dtsize3_mbsize5_mtsize4"
compile -map_effort high
current_design "signcorrect_sbsize12"
compile -map_effort high
current_design "ternarylut_mtsize4_ltsize9_lbsize5"
compile -map_effort high
current_design "ltap_export"
create_clock -name "CLKI" -period 25 -waveform {   "0" "12.5" } {   "CLKI" }
set_dont_touch_network find( clock, "CLKI")
set_clock_skew -plus_uncertainty 0.5 "CLKI"
set_clock_skew -minus_uncertainty 0.5 "CLKI"
uniquify
set_port_is_pad "*"
insert_pads
/* set_max_transition 2 */
/* set_min_fault_coverage 98 */
compile -map_effort high
write -for verilog -out ltap.v -hier
write_constraints -cover_design -format sdf-v2.1 -output ltap.sdf
```

To run a script type the following command into the dc_shell, or into the Command Window of the design_analyzer.

```
include script_name
```

Where **script_name** is the name of the script. The script may be generated in an iterative manner as commands are entered into Synopsys. The explanation for the script is:

1) The first line of the script clears out any existing design currently in Synopsys.
2) The **analyze** lines read in a file of type VHDL and save them into the Synopsys library DBNS (as specified in the .synopsys_vss.setup file).
3) The **elaborate** command is used on the top level cell
4) The **current_design** command is used to set the focus of Synopsys onto a specific cell. The compiling of cells should start at the bottom most level of the design hierarchy and move up through the hierarchy to the top most cell. The names can be determined from the display window in design_analyzer. The names of the components may be different than that specified in the VHDL code. This is due to the use of parameters.
5) **compile -map_effort high** converts the VHDL description into a hardware representation with trying to minimized the area used.
6) **create_clock** is used to specify which net is the clock, and the period and waveform characteristics.
7) **set_dont_touch_network** is used because another tool will be used to optimized the clock path.
8) /* ... */ are comments and were left in to show that undesired commands could be commented out, or notations can be made inside the script.

9) The **write** command is used to write out the compiled design in Verilog. At this point the Verilog code is consists only of instances for the standard cells and their connections.

10) **write_constraints** is used to produce the time constraints placed on the circuit, that will be used during routing.

6. The Verilog output from Synopsys is now ready to be used for Placement. This means that the individual library cells will be placed into a chip core and surrounded by the input/output and power pads. The placement of the cells is performed in such a manner as to place cells that are to be connected in close proximity to each other. The tool used for placement is Cadence Physical Design Planner, and is fully described in the CMC Digital Design Flow.

7. After the Placement of the library cells is completed, the Routing of the system nets is the next step. First the power nets are routed, then the clock net. This is to ensure that optimal routing is done for critical nets. Lastly, the general nets of the system are routed. The tool used for routing is Cadence Silicon Ensemble, and is fully described in the CMC Digital Design Flow.

   An important option to include while performing placement is the congestion map. The congestion map shows the likelihood of being able to fully route the design. If the map is all gray, the design can be routed. If the map contains any cautionary yellow or warning red spots, it is probably not possible to route the design. Ensuring that the congestion map is all gray can save a lot of time later.

8. Finally, the chip is brought into Cadence Design Frameworks II for the finishing touches. This involves correcting minor design rule violations and meeting any process requirements. The process requires that certain layers must cover a specified percentage of the chip's surface.

The successful completion of the above steps, with the appropriate references to the CMC Digital Design Flow, will produce a microchip from a high level description.

Vita Auctoris

"Stanley Jonathan Eskritt", born May 7, 1975 in Windsor, Ontario Canada. Jonathan attended the University of Windsor, where he studied for and obtained an Honours Bachelor of Applied Science degree in Electrical Engineering. Jonathan also pursued graduate studies at the University of Windsor where he worked towards a Master of Applied Science in the area of Electrical and Computer Engineering focusing on special DSP architectures and VSLI design.