AN IMPLEMENTATION OF DIGITAL FILTERS
USING FERMAT NUMBER TRANSFORM
ON TMS320C30 DIGITAL
SIGNAL PROCESSOR

By

YEK CHONG YEO

Bachelor of Science

Oklhahoma State University

Stillwater, Oklahoma

1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1995

OKLAHOMA STATE UNIVERSITY

AN IMPLEMENTATION OF DIGITAL FILTERS

USING FERMAT NUMBER TRANSFORM

ON TMS320C30 DIGITAL

SIGNAL PROCESSOR

Thesis Approved:

H. Lu

Thesis Adviser

Blayne E. Mayfield

K. A. Teague

Thomas C. Collins

Dean of the Graduate College

PREFACE

Precise systems design, equipment standardization, and stability of performance characteristics are among the many advantages digital techniques can offer in signal processing. Earlier research in this field of study has contributed much to many of the modern day conveniences. Many of these contributions focus on improving computational efficiency of discrete Fourier transform (DFT) calculation. However, there are many shortcomings; therefore number theoretic transform (NTT) is proposed.

This study implements three digital filters using one of the NTT, namely the Fermat number transform (FNT), and DFT. It compares the execution time, number of operation, and memory requirement for both implementations. Implementation of both types of filters employs the radix-2 fast Fourier transform (FFT). This study proposes a modified diminished-one number system in implementing FNT. The number system was originally proposed by Leibowitz.

I would like to take this opportunity to thank my major advisor, Dr. Lu, for the encouragement she has offered over the years. Her patience and constructive guidance has been very helpful. Also, I would like to thank Dr. Teague for his kindness in giving me access to his digital signal processing laboratory, where most of the work in this study was done. My appreciation also goes to my parents for their continuous support

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

CCP	cyclic convolution property
DFT	discrete Fourier transform
DIF	decimation-in-frequency
DIT	decimation-in-time
DSP	discrete signal processor
EVM	evaluation module
FFT	fast Fourier transform
FNT	Fermat number transform
IDFT	inverse discrete Fourier transform
IFNT	inverse Fermat number transform
MNT	Mersenne number transform
NTT	number theoretic transform
PFA	prime factor algorithm
RT	Rader transform
TI	Texas Instruments
WFTA	Winograd Fourier transform algorithm

CHAPTER 1

1. INTRODUCTION

C.M. Rader introduced Mersenne number transform (MNT) in 1972 [1]. His proposal started a whole new category of transform, generally termed as number theoretic transform (NTT). This opened up new possibilities in digital signal processing. Among the many claims of number theoretic transform are precision and accuracy, processing speed, and a lower memory requirement compared to its complex counterpart. As the structure of NTT is very similar to that of DFT, FFT algorithms are most suitable for improving efficiency. Interesting enough, FFT algorithms did not draw much attention until a decade earlier, when Cooley and Tukey published their paper in 1965 [2].

This study seeks to implement lowpass, bandpass, and highpass filters using NTT and DFT. The specific focus is FNT and DFT, using radix-2 FFT algorithm implementations. The software Hypersignal was used to design the filters, producing a set of filter data sequence to convolute with input signals. A modification of L.M. Leibowitz's diminished-one number system used to implement FNT is presented. The study compares the execution time and the memory requirement for the two implementations. An analysis of the computational accuracy is also presented. All these implementations are run on Texas Instruments' (TI) TMS320C30 evaluation module.

Chapter 2 contains a historical account of the development of the field of digital signal processing. The chapter first recount early 19th century discoveries. A description of the improvements in both NTT and DFT will follow. Chapter 3 gives the necessary background information regarding the implementations in this study. Chapter 4 discusses the specific implementations. A detailed description of the algorithm implemented in this study, followed by a discussion of the modified diminished-one operations is also presented. Chapter 5 presents an analysis of the results collected from the two implementations. This paper will then end with a summary in chapter 6. It reiterates the findings and wraps up this study.

CHAPTER 2

2. LITERATURE REVIEW

2.1 Introduction

The contribution of the French mathematician Jean Baptiste Joseph Fourier (1768 - 1830) is a major milestone to the study of signal processing [3]. Fourier first demonstrated the method of representing periodic functions in infinite harmonic series. This method, now known as the Fourier series, has become a valuable tool for the study of digital signal processing. Later, the definition of the Fourier transform is developed from the foundation of this work. Just as the Fourier series describes a periodic function in terms of the frequency-domain attributes of amplitude and phase, the Fourier transform extends this frequency-domain description to aperiodic functions. However, digital signal processing did not catch on until Cooley and Tukey published their work in 1965 [2]. Their work was indeed a turning point in digital signal processing and in certain areas of numerical analysis. With their method, the order of complexity of performing DFT drops from $O(N^2)$ to a significantly lower order at $O(N \log_2 N)$.

Less known to many researchers, however, is the treatise written by the eminent German mathematician Carl Friedrich Gauss (1777 - 1855) [4]. This treatise described an algorithm similar to the Cooley-Tukey FFT more than one hundred and fifty years earlier, in 1805. As noted by H. Burkhardt [5] in 1904 and H. H. Goldstine [6] in 1977, this work predated even Fourier's 1807 work on harmonic analysis.

Although DFT is useful in today's digital electronic era, time-consuming complex operations and roundoff errors due to finite word length plague its usefulness. Many researchers have begun to explore similar transforms exhibiting cyclic convolution property (CCP), which is having the transform of the cyclic convolution of two sequences equal to the product of their transforms. In 1971, Pollard published a paper [18] discussing the conditions for having transforms showing CCP and defined transforms in a finite (Galois) field. Transforms in the rings of integers appeared after Rader proposed the transforms in the rings of integers modulo a Mersenne number or a Fermat number. Researchers call these transforms in a ring of integers NTT.

This chapter first recounts the development behind modern day digital signal processing since early 1800. Then, an account of the progress in DFT follows. Thereafter, the chapter discusses progress in NTT.

2.2 Theoretical Development

2.2.1 Early Developments

In a well-documented article [7], M.T. Heideman referred to two earlier papers [5,6], confirming Gauss's contribution to the study of signal processing. He quoted Herman H. Goldstine [6], "This fascinating work of Gauss was neglected and was rediscovered by Cooley and Tukey in an important paper in 1965." Here, he was referring to Gauss's treatise, "Theoria Interpolations Methodo Nova Tractata." Gauss published this treatise, most likely written in 1805, in 1866. H. Burkhardt, who wrote in 1904 [5], also noted the contribution of Guass.

While a student at Göttingen, Gauss became familiar with the works of Leonhard Euler (1707 - 1783) and Joseph Louis Lagrange (1736 - 1813) on the analysis of trigonometric series [7]. He later extended these works on trigonometric interpolation to periodic functions. Nevertheless, most of his important publications are in a nineteenth-century version of Latin, called neo-Latin. Unfortunately, it is difficult for a casual student of classical Latin to translate neo-Latin accurately. Gauss's notation in describing his method also posts an obstacle to modern readers. As such, his work went largely unnoticed. Heideman wrote, "Burkhardt pointed out this algorithm in

1904 and Goldstine suggested the connection between Gauss and the FFT in 1977, but both of these went largely unnoticed, presumably because they were published in books dealing primarily with history." So, the world did not benefit from Gauss's method until Cooley and Tukey came along in 1965.

"Gauss's algorithm is as general and powerful as the Cooley-Tukey common-factor algorithm and is, in fact, equivalent to a decimation-in-frequency algorithm adapted to a real data sequence," wrote Heideman. Although Gauss did not go on to quantify the computational requirements of his method, clearly his algorithm performs in the order of complexity at the now familiar $O(N \log_2 N)$.

Although Gauss's contribution precedes that of Fourier, Fourier's work in harmonic series commands more attention. It is fascinating to see that the knowledge acquired in one field of study became useful in another. In fact, Fourier was studying and analyzing the heat flow in metal rods when he discovered the trigonometric series representation of a periodic function. He did not know the importance of his work to modern day signal processing. Essentially, Fourier showed that any periodic function is expressible as a function of harmonic frequencies of the fundamental frequency, which we now refer to as the Fourier series. Such mathematical abstraction of periodic functions is a useful tool for objective observation of periodic signals. Beside the physical interpretation to view periodic functions as a sum of component functions with harmonic frequencies, Fourier series allow us to describe such functions in their frequency-domain attributes. Further development introduces the idea of Fourier transform, giving us the abstract mathematical tools to study aperiodic functions.

Over the years, independent works that were unrelated to Gauss's work appeared, but were not as general or as well formulated as his work. Many of the methods did not handle computation above the fourth harmonics until Runge published his work [7] in the early 1900's.

2.2.2 Discrete Fourier Transform

At the turn of the century, Runge proposed an algorithm for lengths equal to powers of two [7], which was later generalized to powers of three as well. Apparently, his work was well known and was cited in the popular textbook written by Whittaker and Robinson [8]. His influence did not survive after the war, however.

After the war, in 1958, another important contribution appeared. Good developed an index mapping that facilitated the division of a problem into subproblems of smaller length [9]. This mapping itself is an application of the Chinese remainder theorem, which dates back to the Chinese mathematician Sun-Tsu some time between 200 BC and 200 AD [10]. When Cooley and Tukey presented their fast Fourier transform in 1965, they claimed to base their work on Good's mapping, apparently unaware of Gauss's algorithm that the world had forgotten more than a century ago. However, there are major differences between Good's algorithm and the Cooley-Tukey FFT. The former does not require auxiliary complex multiplications, referred to as twiddle factors, while the later does; thus, the two different classes of FFT.

The development of FFT, without twiddle factors, did not become popular with Good's algorithm, which is suitable when factors of the transform length are coprime, or are indivisible by each other. Good's algorithm requires a set of efficient small-length DFT algorithms. Paradoxically, in 1968, Rader published a paper that showed how to map a prime length N DFT into circular convolution of length $N-1$ [11]. Yet, not until Winograd's study on complexity theory [12] that the two foundation works above are ready for efficient applications on signal processing. Winograd published his paper in 1977, presenting his complex theory, particularly on the number of multiplications required for computing polynomial products or convolution. His work is crucial to this class of FFT. With Good's mapping, coupled with

Rader's fast convolution scheme, a first algorithm makes use of the intimate structure of these convolution schemes to obtain a nesting of the various multiplications. This algorithm is now known as Winograd Fourier transform algorithm (WFTA). If the nesting is not used, the resulting algorithm is known as prime factor algorithm (PFA).

The development in FFT with twiddle factors took off with the Cooley-Tukey FFT. Unlike Good's algorithm, the Cooley-Tukey FFT can have transform length of any composite length. With the growing interest in the theoretical aspects of digital signal processing motivated by technical improvements in the semiconductor industry, and the availability of reasonable computing power, the Cooley-Tukey FFT quickly became an interest of research. They employ the divide and conquer approach by separating input sequences to process. When the sequence length is a power of two, their algorithm becomes what is known as the radix-2 decimation-in-time (DIT) algorithm. Emphasis on a dual approach leads to decimation-in-frequency (DIF) algorithm. Later, Bergland noted that the algorithm can be more efficient with higher radices like radix-8 [13]. In 1984, there were four proposals [14,15,16,17] submitted at about the same time that leads to split-radix algorithm. This approach uses a different radix for the even part (radix-2) and the odd part (radix-4).

This research uses a radix-2 algorithm for comparison with the NTT implementation. Thus, it is the author's intention to leave the rest of the development in this thread, for example polynomial transform, to the reader to explore. This discussion will continue with the motivations and history of NTT.

2.2.3 Number Theoretic Transform

In his paper published in 1971 [18], J.M. Pollard defined a transform in finite (Galois) field analogous to the DFT. Replacing the complex roots of unity in DFT with r^k, where r is a member of any field F, of order d and k is an integer, he showed that CCP holds for the transform in F. This is under the condition that r has finite order d in the multiplicative group F^* of F. Generalizing his definition, he suggested transforms in a ring of integers modulo m, where m is an integer. C.M. Rader picked up from here and introduced a transformation defined in the rings of integers modulo a Mersenne number [1]. We now call this transform Mersenne number transform (MNT).

Unlike DFT, the only arithmetic operations MNT needs are that of additions and circular shifts of bits within a word. One other advantage the MNT has over DFT is the accuracy of the transformation it can attain; virtually no roundoff errors. Rader also suggested a transformation using Fermat numbers. In time, researchers coined the term 'number theoretic transform' to refer to similar transformations. Later in 1974, R.C. Argarwal and C.S. Burrus [19] worked on a transform using Fermat numbers, named Fermat number transforms (FNT). In their paper, they formalized the conditions for CCP and showed its relation to the transform length. Unlike Rader, they defined the roots of unity as α^r instead of 2^r. Thus giving more flexibility to the transform. Since these transformations are similar to the DFT, therefore FFT algorithms are perfectly suitable on NTT to achieve better efficiency. As the topic of interest is on implementation of FNT, many of the later developments in NTT are left to the readers to explore. The emphasis will now turn to issues in the implementation of FNT.

Argarwal and Burrus proposed implementations of the various basic arithmetic operations modulo a Fermat number, $F_t = 2^b + 1$, in their paper [19]. However, this method does not have the convenience of circular shifts of bits when performing scaling. The method also involves the representation of the number -1, requiring $b+1$ bits. The integer b is equal to 2^t. Argarwal and Burrus decided to ignore this extra bit in order to simplify modular arithmetic operations. While McClellan developed a technique of exact computation using a new binary code representation of $b+1$ bits in 1976 [20], L.M. Leibowitz proposed a similar binary representation he called diminished-one number system the same year [21]. His method shows less mathematical complexity than that of McClellan. It also allows circular bit shifts for scaling by powers of 2. This paper builds on the foundation of Leibowitz's diminished-one number system and presents modifications to the various diminished-one number operations.

2.3 Hardware Environment

2.3.1 TMS320C30 Evaluation Module

The TMS320C30 Evaluation Module (EVM) is a tool for application development. This module allows the execution and debugging of application programs. With one of the fastest digital signal processors (DSP), the 33-MFLOP TMS320C30 floating-point DSP, this EVM provides a lot of computing power. This section will give a brief description of the hardware environment used in this study.

As mentioned earlier, the TMS320C30 EVM employs the TI TMS320C30 DSP for its brain power. This DSP has a 60-ns single-cycle instruction execution time (33Mhz). The processor has one 4K x 32-bit single-cycle, dual-access on-chip ROM block, and two 1K x 32-bit single-cycle dual-access on-chip RAM blocks. It also includes a 64 x 32-bit instruction cache. The instruction and data words are 32-bit, while the addresses are 24-bit. Both the multiplier and the ALU have 40-bits floating-point operations and 32-bits for integer operations. It has a 32-bit barrel shifter. Among the many advances in this processor, it can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle. Another main feature of this processor is its internal dual-access memory capability.

In addition to the TMS320C30 DSP, this EVM has 16K words of zero wait-state SRAM on the primary bus. It also has a voice quality analog data acquisition circuitry, with standard RCA jacks for line-level analog input and output. Beside an external serial port, it also has a 16-bit bidirectional PC host communication port. The unit is built on an IBM PC/AT compatible 8-bit half card that fits onto any PC compatible computer.

With this computing power, the EVM represents one of the latest technologies available in the market. It is, therefore, a suitable platform to test and to compare the various implementations for use in this study.

CHAPTER 3

3. Background Information

3.1 Fourier Series and Fourier Transform

3.1.1 Fourier Series

What Fourier discovered in his experiments on heat flow is that a periodic function is expressible as the sum of an infinite number of sinusoids with a period that is the multiple of the fundamental frequency. The equation below is the mathematical realization of this discovery.

$$x(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left[a_n \cos\frac{2\pi nt}{P} + b_n \sin\frac{2\pi nt}{P}\right]$$

Here, $x(t)$ is a periodic time function that is integrable over its period P. The a's and b's are the Fourier coefficients, while $2\pi/P$ is the fundamental frequency of the $x(t)$. The integral multiples of the fundamental frequency are the harmonic frequencies of $x(t)$.

Mathematically, Dirichlet's conditions ensure a convergent Fourier series. These conditions, as noted in [3], are

1. $x(t)$ is single-valued;

2. $x(t)$ has a finite number of discontinuities in the periodic interval;

3. $x(t)$ has a finite number of maxima and minima in the periodic interval; and

4. The integral $\int_{t_0}^{t_0+P} |x(t)| dt$ exists.

These are all sufficient conditions and not necessary conditions. Therefore, Fourier series can express any periodic functions meeting these requirements. However, periodic functions not meeting these conditions may still be expressible in Fourier series.

Fourier series leads to the definition of Fourier transform. Note the frequency-domain attributes in Fourier series in the next section, which discusses the development of Fourier transform.

3.1.2 Fourier Transform

Strictly speaking, Fourier transform is not a new transform. J.W. Nilsson [3] wrote, "It is a special case of bilateral Laplace transform with the real part of the complex frequency set equal to zero." Nevertheless, understanding the evolution from Fourier series to Fourier transform gives tremendous insights to the physical significance of this transform.

By replacing the sine and cosine terms in the Fourier series equation

$$x(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left[a_n \cos\frac{2\pi nt}{P} + b_n \sin\frac{2\pi nt}{P}\right]$$

with the identities

$\cos\theta = \frac{1}{2}\left(e^{j\theta} + e^{-j\theta}\right)$,

$\sin\theta = \frac{1}{2j}\left(e^{j\theta} - e^{-j\theta}\right)$, and collecting like terms, the Fourier series equation becomes

$$x(t) = \sum_{n=-\infty}^{\infty} \frac{1}{2}\left[a_{|n|} - jsign(n)b_{|n|}\right]e^{2\pi nt/P},$$

where $sign(n) = \left\{\frac{1, n \geq 0}{-1, n < 0}\right.$

Then let

$$X(n) = \frac{1}{2}\left[a_{|n|} - jsign(n)b_{|n|}\right]$$

such that the equation becomes

$$x(t) = \sum_{n=-\infty}^{\infty} X(n)e^{2\pi nt/P}$$

Therefore, $|X(n)|$ is the frequency magnitude at $2\pi nt/P$, where $1/P$ is the fundamental frequency of the original function. Note that the exponential functions are orthogonal as in

$$\frac{1}{P}\int_{-P/2}^{P/2} e^{-j2\pi kt/P}e^{-j2\pi lt/P}dt = \delta_{kl}$$

where δ_{kl} is the Kronecker delta function given by

$$\delta_{kl} = \begin{cases} 1, k = l \\ 0, otherwise \end{cases}$$

The summation index in the above equation can be changed to l. Then multiplying both sides of the equation by $e^{-j2\pi kt/P}$, integrating from $-P/2$ to $P/2$, and applying the orthogonality equality gives the following equation.

$$X(k) = \frac{1}{P} \int_{-P/2}^{P/2} x(t) e^{-i2\pi kt/P} dt$$

Following this, by multiplying each side with P, and taking limit on P, we get the following

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-2\pi ft} dt$$

This is the Fourier transform of $x(t)$. The inverse Fourier transform is therefore defined as

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{j2\pi ft} df \ .$$

It becomes clear that the Fourier transform of a time function represents its frequency domain counterpart, as $X(n)$ determines the frequency magnitude. The roles of the exponential terms clearly relate the Fourier transform to the frequency domain.

As explained in the previous chapter, the Fourier transform exhibits CCP. This means that the product of the transforms of two time functions is equal to the transform of their convolution. This property has no apparent usefulness unless a more efficient computation of the Fourier transform becomes available, which is very crucial to the study of signal processing as noted in the later sections. One other invaluable contribution of Fourier transform is the transformation of a time function to the frequency counterpart of the function. This essentially allows the ability to perform frequency analysis of any time function, an essential operation in most signal processing studies.

With the development in microelectronics technology, today's computing power increases many fold compared to that of only three decades ago. Combining these advances, and the many advantages of digital filter characteristics, it is undeniably practicable to conceive a digital form of the Fourier transform. Thus, the conception of DFT is a natural progression.

3.2 Signal Filtering and Convolution

The process of filtering signals involves modification of the frequency attributes of the signal. In most cases, such processes include removal or intensification of a certain range of frequency components of the signal. These processes can best be described as the multiplication of a filter sequence with the signal in the frequency-domain.

Assume $x(t)$ is the sampled signal and $h(t)$ is the filter. The transform of $x(t)$, $X(f)$, and the transform of $h(t)$, $H(f)$, are in the frequency-domain. Clearly, a term-wise multiplication of $H(f)$ to $X(f)$ modifies the frequency attributes of the signal $x(t)$. The results can then be reverse transformed to time-domain to produce the desired signal modification. This process is called convolution.

3.3 Discrete Fourier Transform

DFT is an adaptation of Fourier transform in situations where discrete quantization of a continuous function is necessary. This adaptation is most useful in digital signal processing where discrete numbers are processed. Without going into the details of the derivation, the following equation represents the DFT

$$X(k) = \frac{1}{N}\sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N}, k = 0,1,2,\ldots,N-1$$

The inverse discrete Fourier transform (IDFT) is also a summation, similar to DFT. This is shown as

$$X(n) = \sum_{n=0}^{N-1} X(k) e^{j2\pi nk/N}, k = 0,1,2,\ldots,N-1$$

3.3.1 Cooley-Tukey Algorithm

Starting with the DFT equation, as shown below, Cooley and Tukey employed a divide and conquer strategy to simplify the computation of the transform.

$$X(k) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)W^{nk}, W = e^{-j2\pi/N}$$

They first divide the transform length N into smaller length, as in $N = N_1 \bullet N_2$. Letting

$$n = n_2 N_1 + n_1, n_1 = 0,1,2,\ldots,N_1 - 1, n_2 = 0,1,2,\ldots,N_2 - 1$$

$$k = kn_1 N_2 + k_2, k_1 = 0,1,2,\ldots,N_1 - 1, k_2 = 0,1,2,\ldots,N_2 - 1$$

Cooley and Tukey defined their FFT algorithm as follows.

$$y_{n_1,k_2} = \sum_{n_2=0}^{N_2-1} x_{n_2 N_1 + n_1} W_{N_2}^{n_2 k_2}$$

$$y'_{n_1,k_2} = y_{n_1,k_2} W_N^{n_1 k_2}$$

$$X_{k_1 N_2 + k_2} = \sum_{n_1=0}^{N_1-1} y'_{n_1,k_2} W_{N_1}^{n_1 k_1}$$

This modification breaks up the computation of DFT into two smaller pieces, y_{n_1,k_2} and $x_{k_1 N_2 + k_2}$. Each of the smaller pieces can then be further broken up into even smaller pieces. In this way, the computational time complexity reduces from $O(N^2)$ to $O(N \log N)$

3.3.2 Radix-2 Fast Fourier Transform

The radix-2 FFT is a special case of the Cooley-Tukey FFT. This type of FFT has transform length $N = 2^n$. Assuming that N is the product of N_1 and N_2, where $N_1 = 2$, and $N_2 = N/2$, we get the following:

$$X_{k_2} = \sum_{n_2=0}^{N/2-1} x_{2n_2} W_{N/2}^{n_2k_2} + W_N^{k_2} \sum_{n_2=0}^{N/2-1} x_{2n_2+1} W_{N/2}^{n_2k_2}$$

$$X_{N/2+k_2} = \sum_{n_2=0}^{N/2-1} x_{2n_2} W_{N/2}^{n_2k_2} - W_N^{k_2} \sum_{n_2=0}^{N/2-1} x_{2n_2+1} W_{N/2}^{n_2k_2}$$

This form of radix-2 FFT is termed as decimation-in-time (DIT). However, owing to the transform structure, it is necessary to rearrange the order of the input sequence. This process is called bit-reverse procedure. Essentially, the order of the input sequence is arranged such that the index has its bits reversed. For example, an $N = 4$ transform sequence will have an input sequence of 0,2,1,3 (00,10,01,11).

By emphasizing the duality principle, taking $N_1 = N/2$ and $N_2 = 2$, we get the decimation-in-frequency (DIF) radix-2 FFT algorithm.

$$X_{2k_1} = \sum_{n_1=0}^{N/2-1} W_{N/2}^{n_1k_1}\left(x_{n_1} + x_{N/2+n_1}\right)$$

$$X_{2k_1+1} = \sum_{n_1=0}^{N/2-1} W_{N/2}^{n_1k_1} W_N^{n_1}\left(x_{n_1} - x_{N/2+n_1}\right)$$

Somewhat different from the DIT algorithm, the output of this algorithm is in bit-reversed order when the input is in ascending order.

3.4 Number Theoretic Transform

When Pollard published his paper in 1971 [18], he had in mind analogous transforms to Fourier transform that also exhibit the CCP. Yet, these transforms are defined in finite field. As transforms defined in the rings of integers appeared since Argarwal and Burrus published their important paper in 1974 [19], the motivation had been an attempt to do away with the inaccuracy of complex operations due to limited word length. One other motivation had been the complete removal of multiplications that such transforms promised. Today, these motivations are still as credible. Such transforms in the rings of integers are now termed as number theoretic transform. Among the more popular NTT is FNT.

3.4.1 Fermat Number Transform

FNT is defined over the rings of integers modulo a Fermat number, F_t, defined as 2^b+1, where $b=2^t$. This transform and its inverse transform are defined as follows. The FNT is defined as

$$X_k = \left\langle \sum_{n=0}^{N-1} x_n \alpha^{\langle nk \rangle_b} \right\rangle_{F_t}, where \quad k = 0,1,2,\ldots,N-1$$

$$\alpha^N = \langle 1 \rangle_{F_t}, and \quad N = 2^m$$

while the inverse Fermat number transform (IFNT) is defined as

$$x_n = \left\langle q \sum_{k=0}^{N-1} X_k \alpha^{\langle -nk \rangle_b} \right\rangle_{F_t}$$

$$q = 2^{-m} = \left\langle -2^{b-m} \right\rangle_{F_t}$$

α is an integer of order N, transform length. If α is taken as the number 3, it will have an order $N = 2^b$. However, if α is taken as 2, it has an order of $N = 2b$. In this case, the transform is also called the Rader transform (RT). The transform is most efficiently implemented in this case, since most digital devices are designed to function in base 2.

When Argarwal and Burrus proposed Fermat number transform, they also suggested a set of arithmetic operations to make this transform possible. However, multiplication by powers of 2 is not as simple. Later, L.M. Leibowitze introduced a simplified binary number system named diminished-one number system to simplify arithmetic operations in the rings of integers modulo a Fermat number. The next section describes this number system.

3.4.2 Diminished-One Number System

Leibowitz defined a number A to be represented by $[a_b,\ldots,a_1,a_0]$ where $0 \le A \le 2^b$. In this representation, the number zero is represented by $a_b = 1$, and $a_i = 0$ for $i = 0,1,2,\ldots,b-1$. All other numbers are represented by the normal binary representation of $(A-1)$.

To add two numbers in this system involves taking the sum of the two numbers and adding the complement of the carry to the sum. However, if any of the numbers to be added is zero, the sum is set to the other number.

Negation in this number system is much simpler. Remembering that all operation in this number system is to modulo a Fermat number, negation of a number is simply the binary complement of the number. The only exception in this operation is, again, the number zero. In this case, the operation is inhibited.

Subtraction in this number system is defined as a combination of the above two operations. It involves a negation of the subtrahend and an addition to the minuend.

One of the more frequently used operations in Fermat number transforms is scaling. Scaling involves the multiplication with the roots of unity. In this case, the root is always a number that is a power of 2. Thus, multiplication by powers of 2 becomes an important operation in ensuring an efficient algorithm. Fortunately, this operation involves only left bit shifts and an addition of the complement of the carry bit. In other word, "for each factor of 2, a left-circular shift of the b-lsb's is required and the bit circulated into the lsb is complemented."

The last operation that Leibowitz proposed was general multiplication. Here, he offered 3 methods. The first method involves multiplication of the two diminished-one numbers. This result is then added to the b-lsb of the diminished-one sum of the two numbers. Then one must perform a residue reduction of the result by a diminished-one subtraction of the b-msb's from the b-lsb's. The same rule regarding the number zero applies.

The second method requires translation to normal binary coding. The two numbers are then multiplied and a residue reduction as described above is applied. The result is the desired product.

The third method requires a translation of one of the numbers to normal binary coding and then doing a general diminished-one addition.

CHAPTER 4

4. Implementation

4.1 Radix-2 Fast Fourier Transform Algorithm

4.1.1 Decimation-In-Time Algorithm

As presented earlier, the DIT FFT equation is as follows:

$$X_{k_2} = \sum_{n_2=0}^{N/2-1} x_{2n_2} W_{N/2}^{n_2 k_2} + W_N^{k_2} \sum_{n_2=0}^{N/2-1} x_{2n_2+1} W_{N/2}^{n_2 k_2}$$

$$X_{N/2+k_2} = \sum_{n_2=0}^{N/2-1} x_{2n_2} W_{N/2}^{n_2 k_2} - W_N^{k_2} \sum_{n_2=0}^{N/2-1} x_{2n_2+1} W_{N/2}^{n_2 k_2}$$

This form of the Cooley-Tukey FFT divides a transform sequence into two smaller sequences, containing either the set of odd, or the set of even sequences. This corresponds to the first and second term of the above equations. Recursive application of the equation leads to further division of the two sequences into smaller sequences. This continues until each set has only one element remaining. Essentially, this requires recursive application to the first and second term in the equation.

Taking a cue from this, the algorithm for DIT FFT begins by pairing two elements at a time. This produces $N/2$ two element pairs. To perform this operation, called the butterfly operation, multiply a root of unity to the second element, add and subtract from the first, and then store the results. The result produces $N/2$ terms. Combining two terms at a time, and repeating the same processes, results in $N/4$ terms. Continue this until only one term remains. The following figure illustrates this process using a length eight transform sequence.

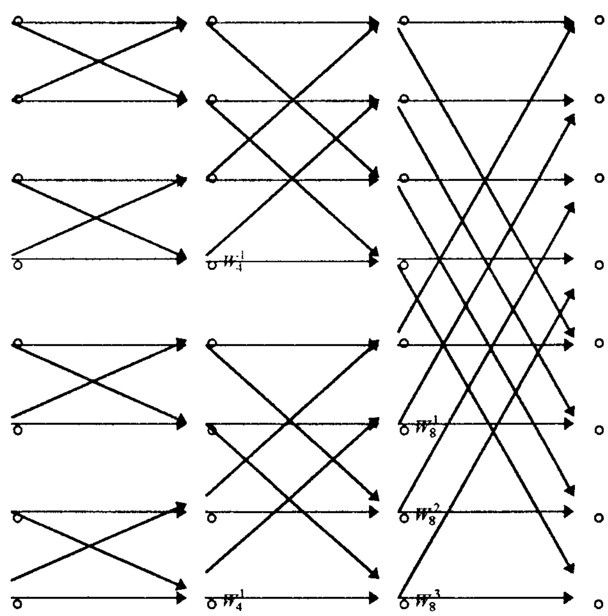

Figure 4. 1: DIT FFT algorithm (Extracted from [10])

As such, the construct of the following algorithm shows the iterative process. Each iteration corresponds to one layer in the above figure, that is a recursive application of the equation on itself. Within each iteration, perform butterfly operations on each element of the separate sets in the sequence. This is how we get the two loops as shown in the algorithm below.

$n = length$

for $l = 1, \log_2 n\{$

$m = 2^l$

$W_m =$ root of unity

$W = 1$

for $j = 0, m/2 - 1\{$

for $k = j, n-1, m\{$

$t = Wx(k + m/2)$

$u = x(k)$

$x(k) = u + t$

$x(k + m/2) = u - t\}$

$W = WW_m\}$

$\}$

return X

Figure 4. 2: DIT FFT pseudo code

Since this algorithm works best with bit reversed input, it makes sense to implement it on IFFT and allow the DIF algorithm to take care of FFT. In this way, there is less wastage in resources to modify input sequence and gives a more efficient implementation.

4.1.2 Decimation-In-Frequency Algorithm

The DIF algorithm is similar to that of the DIT except for the emphasis on duality. Instead of separating the transform sequence into sets of odd and even sequences, the algorithm pairs up elements of the set.

$$X_{2k_1} = \sum_{n_1=0}^{N/2-1} W_{N/2}^{n_1 k_1}\left(x_{n_1} + x_{N/2+n_1}\right)$$

$$X_{2k_1+1} = \sum_{n_1=0}^{N/2-1} W_{N/2}^{n_1 k_1} W_N^{n_1}\left(x_{n_1} - x_{N/2+n_1}\right)$$

The above equations show the pairing of elements half the sequence length apart. Repeated application of the equation to itself leads to the efficient DIF FFT algorithm. The figure below depicts this process.

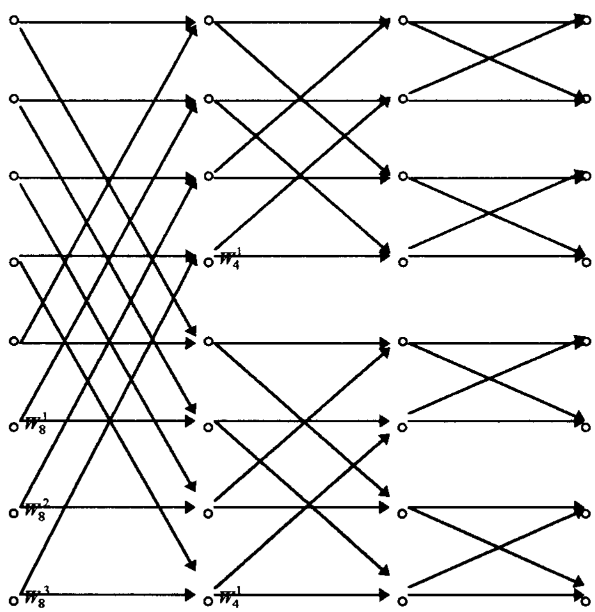

Figure 4. 3: DIF FFT algorithm (Extracted from [10])

Again, taking cue from the figure, the following algorithm begins by performing butterfly operations that are half the sequence length apart. The butterfly operation in this case consists of addition and subtraction of the two elements, followed by the multiplication of the root of unity to the difference of the two elements. This same process continues as the algorithm proceeds to the next layer.

$n = length$

for $l = 1, \log_2 n\{$

$\quad m = n/2^{l-1}$

$\quad W_m =$ root of unity

$\quad W = 1$

$\quad$ for $j = 0, m/2 - 1\{$

$\quad\quad$ for $k = j, n-1, m\{$

$\quad\quad\quad u = x(k) - x(k + m/2)$

$\quad\quad\quad x(k) = x(k) + x(k + m/2)$

$\quad\quad\quad x(k + m/2) = uW\}$

$\quad\quad W = WW_m\}$

$\quad\}$

return X

Figure 4. 4: DIF FFT pseudo code

Note that this algorithm will produce a bit reversed output sequence. It is, therefore, more suitable to implement the FFT. When performing a convolution, this bit reversed output sequence is fed into IFFT implemented in DIT algorithm. Such is a perfect combination that takes advantage of the characteristic of both DIT and DIF FFT algorithms.

4.2 Modified Diminished-One Operations

The idea behind the diminished-one number system is useful because it simplifies arithmetic operations when implementing FNT. This system eradicates the need to perform multiplication. Although not true in most modern digital signal processing processors like the one in use with this research, multiplication is usually the most expensive operation in most processors. All operations in this system consist of simple additions and bit shift operations. However, one of the disadvantages of diminished-one number system is the need to convert from an ordinary binary number before any such operations are useful. Moreover, scaling operations or multiplication by powers of 2 requires multiple steps to produce the desired result for higher powers of scaling factor. It defeats the purpose of employing FNT when this most frequently used operation in a transform demands expensive processing resources.

The modified diminished-one algorithm employs the best of both worlds, normal binary and diminished-one arithmetic operations. Although the theoretical foundation is based on the diminished-one number system, this method attempts to solve the shortcomings of the diminished-one number system by turning back to ordinary binary operations whenever it is most convenient. Scaling has become a multiple bit rotation operation instead of a series of single bit rotations. These are explained in the following sections.

4.2.1 Negation

Negation of a number remains the same as in the diminished-one number system operation. Performing a binary complement on all the bits of a diminished-one number produces its negative counterpart.

4.2.2 Addition

This modification to diminished-one addition eliminates the need to check for the number zero. Therefore reducing the number of branching operations, which is the most expensive operation for the digital signal processor used in this research. The operation assumes all numbers are in the rings of integers modulo a Fermat number, F_t. Therefore, the algorithm disregards all cases of adding negative numbers. An addition begins with normal binary addition and a condition check to determine if the number is indeed greater than or equal to the modulo. When the result is indeed greater than, or equal to the modulo, a conversion to the diminished-one number system enables easy modulo F_t operation. The following equation shows that the sum of two numbers is one more than the value of the sum in the diminished-one number system.

$$A + B = [(A + B) - 1] + 1$$

With this insight, and taking that the range of any addition is no larger than twice the modulo, adding one to the diminished-one number is equivalent to adding the complement of the msb to the b lsb's of the number. Since the maximum number of bits is b+1, and that the msb is 2^b, the above conclusion is true as $\langle 2^b \rangle_{F_t} = -1$.

$k = l + m$

$if(k \geq 2^b)\{$

$k = k - 1$

(subtract 1 to get diminished-one representation)

complement msb and add to b lsb's

(perform conversion from diminished-one to binary)

$\}$

return k

Figure 4. 5: Modified diminished-one addition

Therefore, the algorithm for this operation is shown above in Figure 4.5.

Notice the saving of one conditional check compared to the original diminished-one algorithm. This algorithm also has the advantage of not needing to convert a number into diminished-one number system in order to obtain the desired result.

4.2.3 Subtraction

The subtraction algorithm is similar to the addition algorithm. The rationale is the same, except that this algorithm is much simpler to implement. Like the addition algorithm, the following equation shows the connection between normal binary representation and the diminished-one representation of the difference of two numbers.

$$A - B = [(A - B) - 1] + 1$$

This algorithm assumes that all inputs are in the ring of integers modulo F_t, as in the addition operation. Thus the largest difference is $F_t - 1$ and the smallest difference is $-(F_t - 1)$. When the difference is a positive number, the algorithm does nothing more. When the difference is a negative number, the algorithm retains only the b lsb's of the number.

Employing the same rationale as the addition algorithm, theoretically this algorithm converts the difference to diminished-one representation when it is negative. This eliminates the conditional checks for the number zero and thus the inevitable branch operations. When using the diminished-one representation, a residue reduction on the number accomplishes the modulo F_t operation. Residue reduction involves the addition of the complement of the b msb's to the b lsb's.

Since a negative number has all b msb's equal to 1, its complement is always zero. For all possible negative numbers except the smallest, this involves subtracting one from the b lsb's and adding zero. Following this is a code conversion to normal binary number, which is the addition of the complement of the b+1th bit to the b lsb's. For this case, it is the addition of the number one. This algorithm removes the redundancy of subtracting and later adding one to the b lsb's by simply retaining the b lsb's of a negative number. For the smallest possible negative number, such as the binary number with b ones followed by b zeros, it is different. After converting to diminished-one representation and performing residue reduction, all bits except the b+1th bit are one. A code conversion to normal binary representation gives the same results as in retaining the b lsb's. Therefore, the algorithm is as shown below.

$k = l - m$
$if(k < 0)$

    $k =$ b lsb's of k

return k

Figure 4. 6: Modified diminished-one subtraction

As in the addition operation, this algorithm also removes the need for number system conversions and the need to check for the number zero.

4.2.4 Scaling

The scaling operation is essentially an extension to the diminished-one scaling operation. Noted in the previous chapter, scaling in the diminished-one number system involves rotating the complement of the msb to the lsb. This is easily seen in the following equality.

$$(2A-1)=2(A-1)+1$$

However, this operation is a single bit operation. Therefore, a multiplication of higher powers of 2 requires several scaling operations to accomplish. The proposed algorithm extends the diminished-one scaling operations to multiple bit operation. Noting that the maximum power of 2 in the multiplication of the scaling factor is $2b$, or the maximum transform length for FNT. This operation needs only be concerned with a maximum of $2b$ bit rotations. Subsequent paragraphs will explain the operation using a combination of bit shift, 'xor', and 'or' logic operations. This is so because rotation is not a native operation of the digital signal processor used in this research,.

Referring to the following figures, the left figures represent a number while the right figures represent the same number, d2, with bit extension, d1. A single bit rotation in the left figures is equivalent to a left bit shift and an addition of the complement of the bit shifted into the d1 field. Thus a multiple bit rotation in the left figures is then the same operation as multiple bit shift and addition of the complement of the bits shifted into d1 to d2. The scaling algorithm takes advantage of the above idea to simplify the diminished-one scaling operation.

Assuming scaling by 2^x, and that $mask = 2^b -$, the following figures show the inner working of this algorithm.

Figure 4. 7: Diminished-one number A

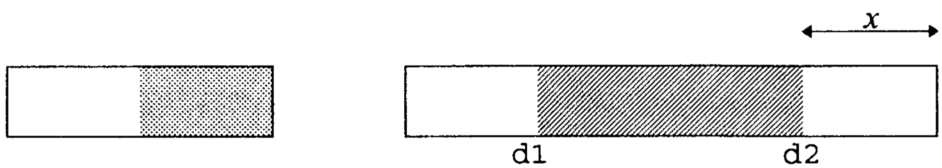

Figure 4. 8: Diminished-one number A after less than b bits rotation

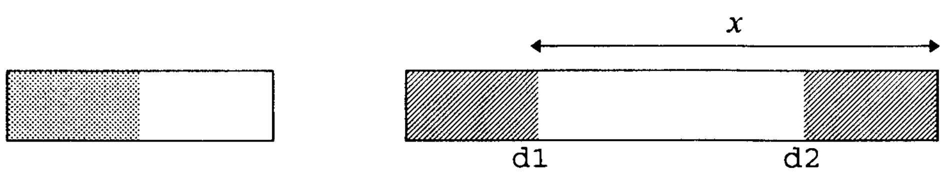

Figure 4. 9: Diminished-one number A after more than b bits rotation

In the above figures on the left, the shaded areas are the complemented bits rotated. The shaded areas in the right figures represent the original bit sequence. When a rotation is less than b bits, figure 2 shows that the msb's of a number is rotated into the d1 zone. However, when the bit rotation is more than b bits, the lsb's of the number gets into the d1 zone. Whenever the bits are in zone d1, take their complement, and 'or' them to the bits in zone d2 to give you the diminished-one result of the scaling operation.

convert A to diminished-one representation

d1 = d2 left shift $x-b$ bits

$mask = mask$ left shift $x-b$ bits

d1 = d1 xor $mask$

if (bit shift $<b$)

  d2 = d2 << x

else

  d2 = d2 << $x-2b$

d2 = d1 or d2

convert d2 to normal binary representation

Figure 4. 10: Diminished-one scaling

Therefore, the above algorithm first converts a number into its diminished-one representation. A left shift of $x-b$ bits gives the bits shifted into zone d1. Perform the same number of bit shifts to the mask and then do a 'xor' with it to produce the complements of the bits shifted into zone d1. The result is stored in a register conveniently named d1. In the case when the number of bit rotation is more than b, the msb's will have been complemented twice.

Therefore, this method does not perform these complements and simply left shifts the original number by $x-2b$. Take note that a negative left shift implies a right shift. The result is then stored in register d2. This is then 'or' with the value in d1 to produce the desired result.

4.2.5 Multiplication

Multiplication remains the same as method 2 proposed by Leibowitz. A normal binary multiplication result in a product, which is the desired product in diminished-one representation. This is followed by a residue reduction. The last step in this algorithm is a code conversion of the product to obtain the normal binary product. Figure 4.6 shows the process of this operation.

Since the algorithm is essentially the same as Leibowitz's diminished-one algorithm, the author will not discuss the algorithm any further. However, as this implementation involves 33 bit operation, note that this implementation assumes three 16 bit parts in a number. This is because the multiplication operation in the digital signal processor used involves 16 bits numbers and produces a product with 24 bits. By separating a number into three 16 bit parts, multiplication becomes a multiple process.

$$
\begin{array}{cccccc}
 & & & A_2 & A_1 & A_0 \\
\mathrm{X} & & & B_2 & B_1 & B_0 \\
\hline
 & & & & \multicolumn{2}{c}{A_0 \circ B_0} \\
 & & & \multicolumn{2}{c}{A_1 \circ B_0} & \\
 & & & \multicolumn{2}{c}{A_0 \circ B_1} & \\
 & & \multicolumn{2}{c}{A_1 \circ B_1} & & \\
 & \multicolumn{3}{c}{B_1 B_0 \circ A_2} & & \\
+ & \multicolumn{3}{c}{A_1 A_0 \circ B_2} & & \\
\hline
 & C_5 & C_4 & C_2 & C_1 & C_0 \\
 & & & & \overline{C_4} & \overline{C_2} \\
+ & & & & & C_5 \\
\hline
 & & & D_2 & D_1 & D_0 \\
+ & & & & & \overline{D_2} \\
\hline
 & & & E_2 & E_1 & E_0
\end{array}
$$

Figure 4. 11: Diminished-one multiplication

The figure above shows the process of multiplying two numbers. Each part of a number is multiplied to the components of the other number. The results are combined to produce the actual product. A residue reduction follows, giving the diminished-one product. This number then goes through a code conversion to obtain the binary representation of the desired result.

CHAPTER 5

5. Data Analysis

As described in the preceding chapter, the filter implementations use both NTT and DFT. Both implementations use the Texas Instruments' TMS320C3X digital signal processor. The three filters are lowpass, bandpass, and highpass filters. Applying these signals to a filter, as presented in the subsequent section, modifies the signal accordingly. The following sections will look at these data and compare them, in the frequency domain, with the original signal. This serves to verify the validity of the algorithm. Analysis of execution time and memory requirements of each implementation will then follow.

5.1 General Data Output

The following sub-sections will look at an input sample and the filtered samples, both in time-domain and in frequency-domain. The input signal obtained via the input port of the TMS320C3X evaluation module has a transform length of 64 words. Performing convolution with the three filter sequences produces the filtered outputs. These output sequences are then compared with the original input data.

5.1.1 Sample Data

The following diagram shows the input data in its time-domain representation. Performing a Fourier transform on this data sequence produces the frequency-domain sequence of the input data, as shown in the next figure. Both figures show the real and the imaginary portion of the signal.

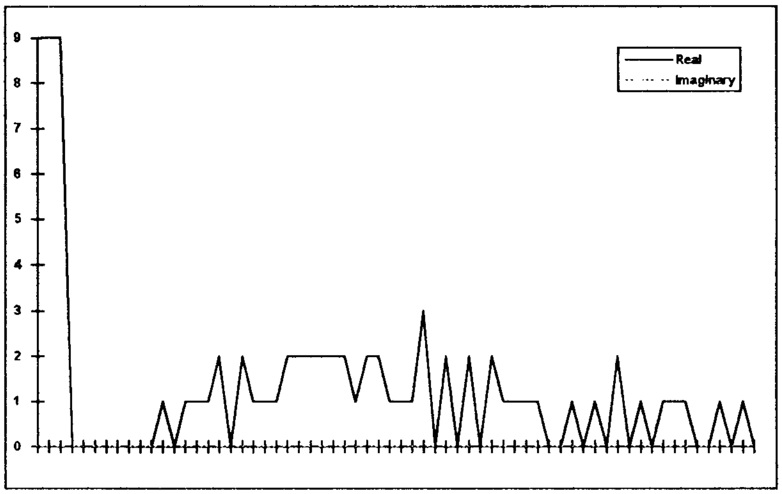

Figure 5. 1: Input data sequence

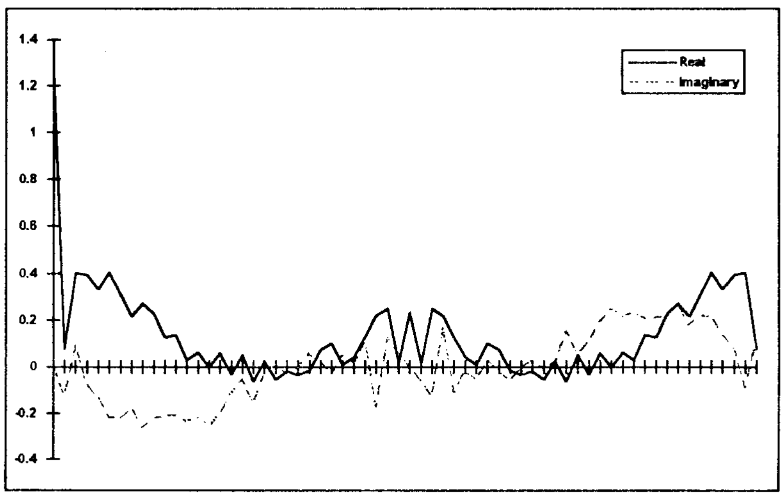

Figure 5. 2: Input data sequence in frequency-domain

Note that the frequency-domain data sequence shows a significant amount of lower frequency components. Also note the symmetry in the figure. Referring to the definition of the Fourier transform in chapter 2, one can find the term $|n|$, and that the summation spans from negative n to positive n. This accounts for the symmetry seen here. Thus, the center of this figure represents the highest frequency components.

5.1.2 Lowpass Filtering

Having looked at the original input sequence, the following figure presents the lowpass filtered sequence.

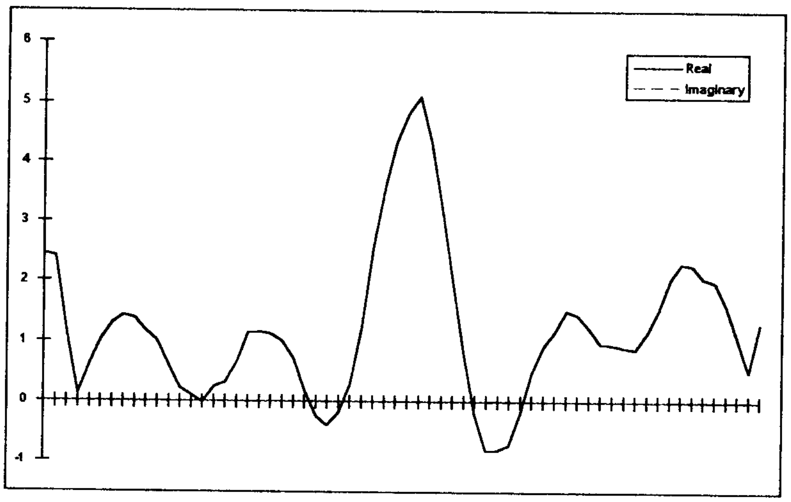

Figure 5. 3: Lowpass filtered sequence using DFT

Figure 5.3 shows the filtered sequence using DFT and Figure 5.4 shows the filtered sequence using NTT. Notice the similarity of the two figures.

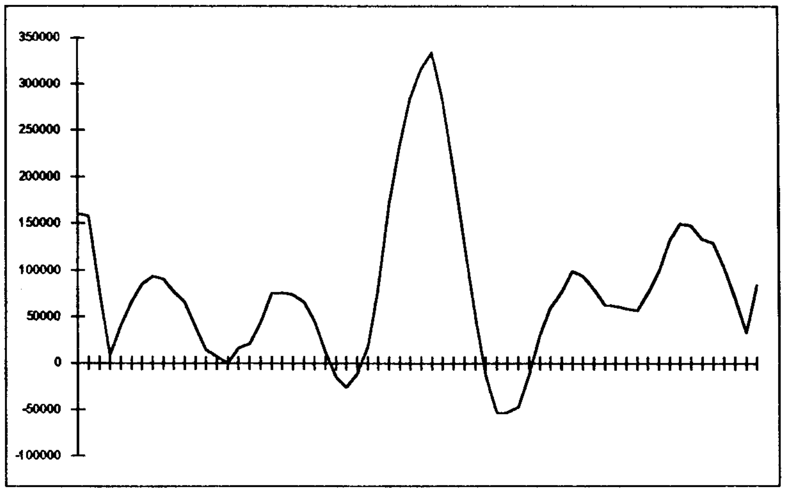

Figure 5. 4: Lowpass filtered sequence using NTT

The values of the output from the NTT version are significantly higher. However, this is due to the fact that the lowpass filter for the DFT method is scaled to unity, while that for the NTT remains in its original form.

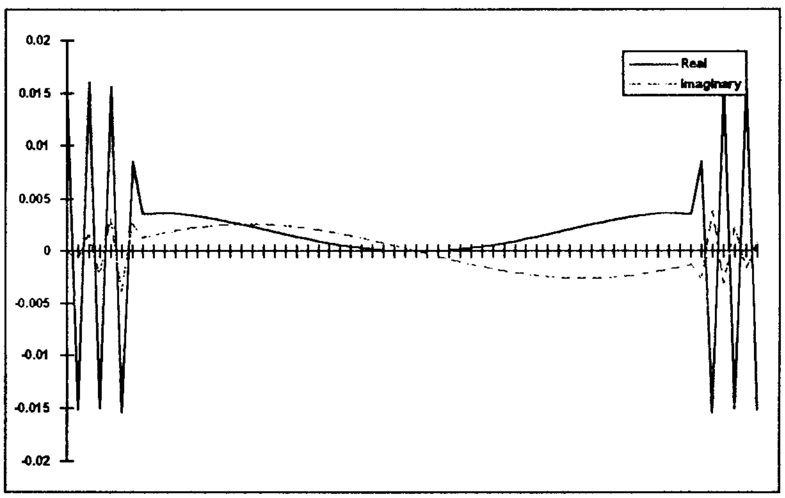

Figure 5. 5: Lowpass filter sequence in frequency-domain

The above figure shows the lowpass filter in frequency-domain. This filter has significantly larger low frequency components and tapers towards the higher frequency. Therefore, the filtered result of the input sequence is expected to contain a fair amount of lower frequency components, while the higher frequency components diminish after filtering.

Looking at the filtered sequence below, the results are just as expected. The higher frequency components of the input sequence are diminished, while the lower frequency region remained.

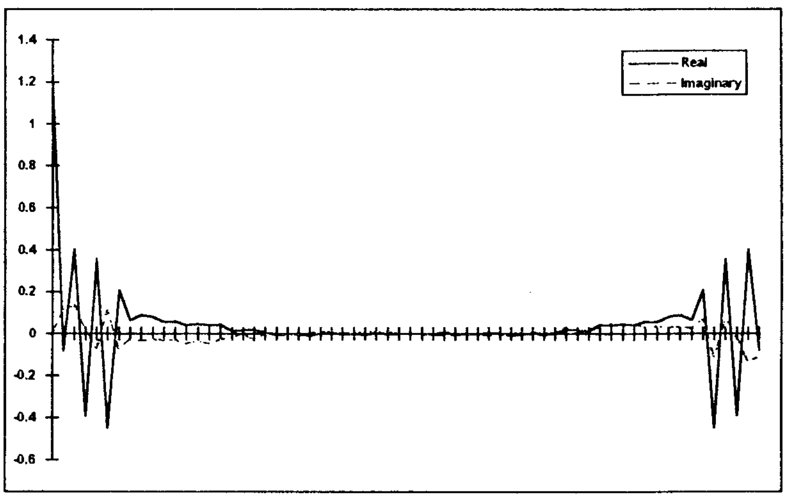

Figure 5. 6: Lowpass filtered output sequence in frequency-domain

5.1.3 Bandpass Filtering

The following two figures show the bandpass filtered output using DFT and NTT convolution respectively. Again, the output sequences have exactly the same shape, with varying amplitudes. The range of the data value is, again, different since the DFT bandpass filter is scaled to unity.

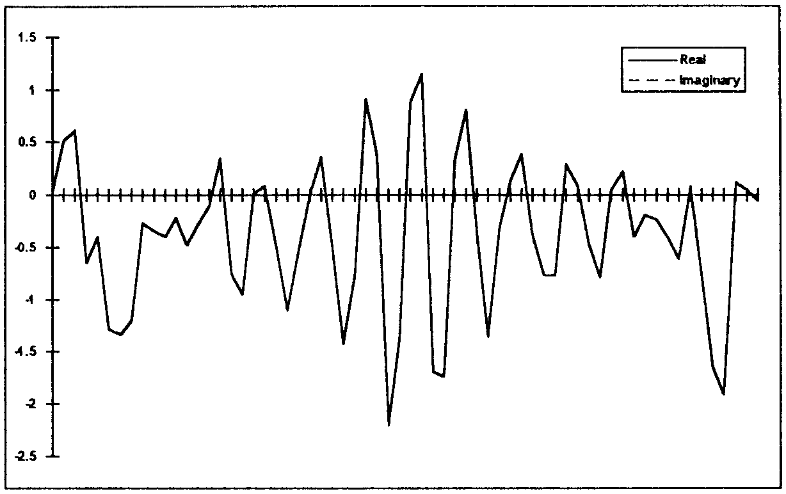

Figure 5. 7: Bandpass filtered sequence using DFT

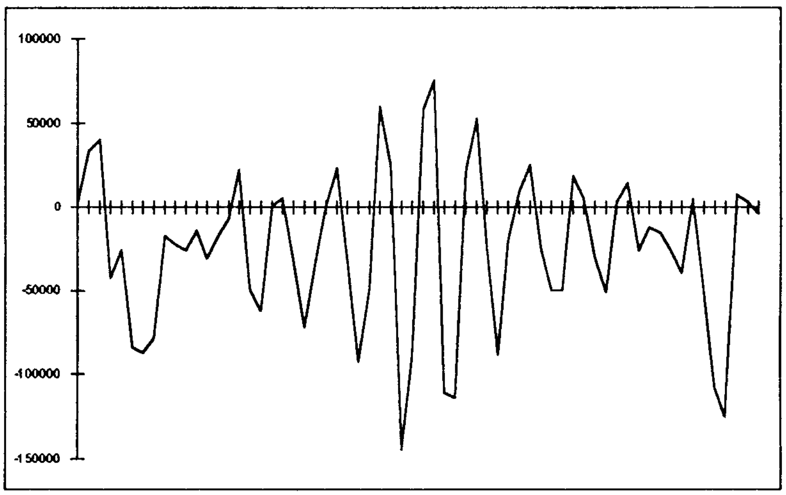

Figure 5. 8: Bandpass filtered sequence using NTT

After observing the similarity between the above two outputs, the following two figures present the frequency-domain output for analysis. The bandpass filter shows a large amount of mid-range frequency components and a fair amount of these components consist of lower frequencies. This being so, the filtered output should show a significant decrease in high frequency components, while retaining most of the mid-range and some of the lower frequency components, which is demonstrated in figure 5.10.

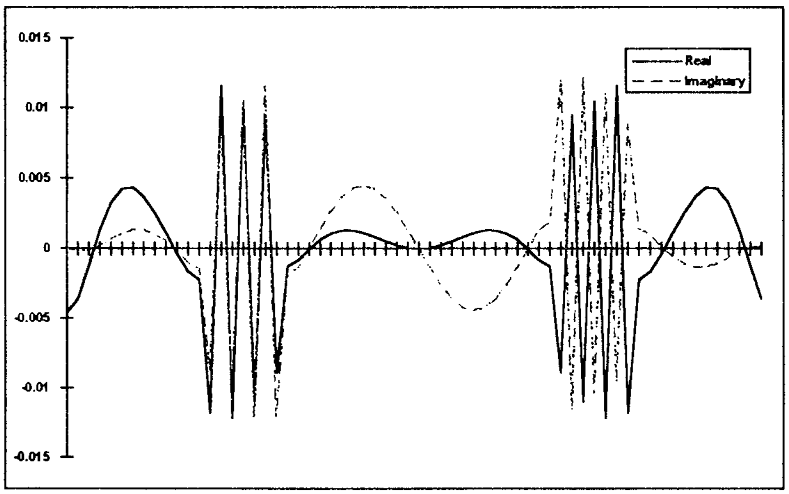

Figure 5. 9: Bandpass filter sequence in frequency-domain

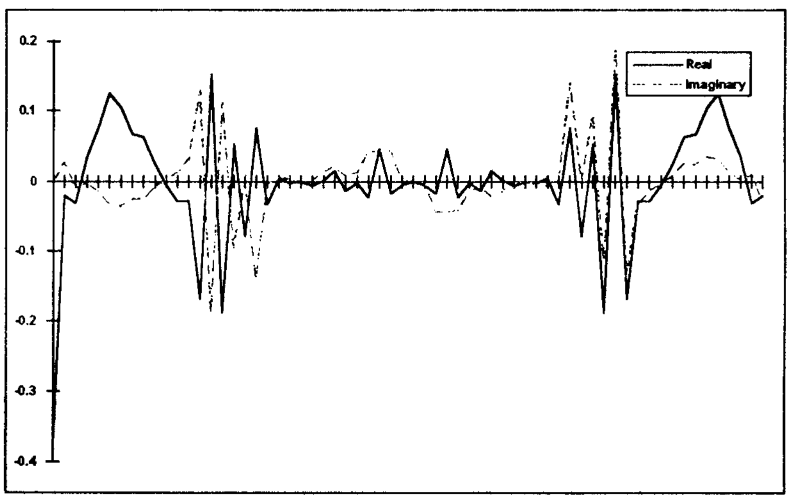

Figure 5. 10: Bandpass filtered output sequence in frequency-domain

5.1.4 Highpass Filtering

As in the above figures, the following figures show the highpass filtered output from both DFT and NTT convolution methods. Again, note that the NTT convolution methods do not produce the imaginary portion. However, although the ranges of the data values are different for the same reason as above, the shape of these sequences are the same.

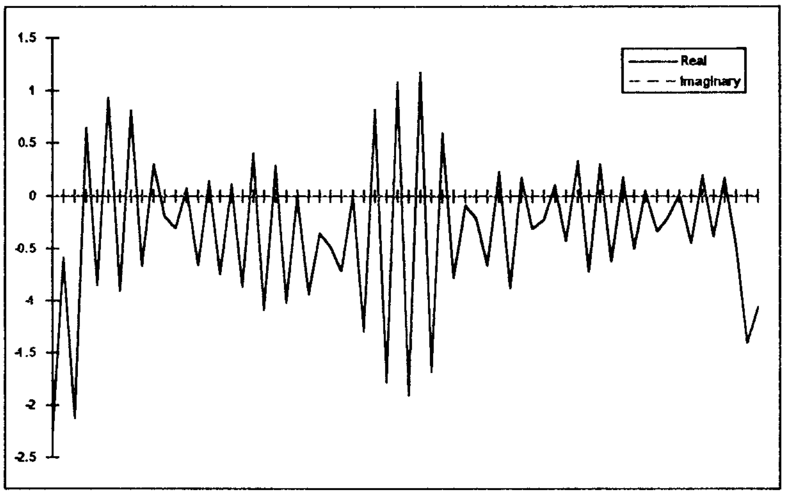

Figure 5. 11: Highpass filtered sequence using DFT

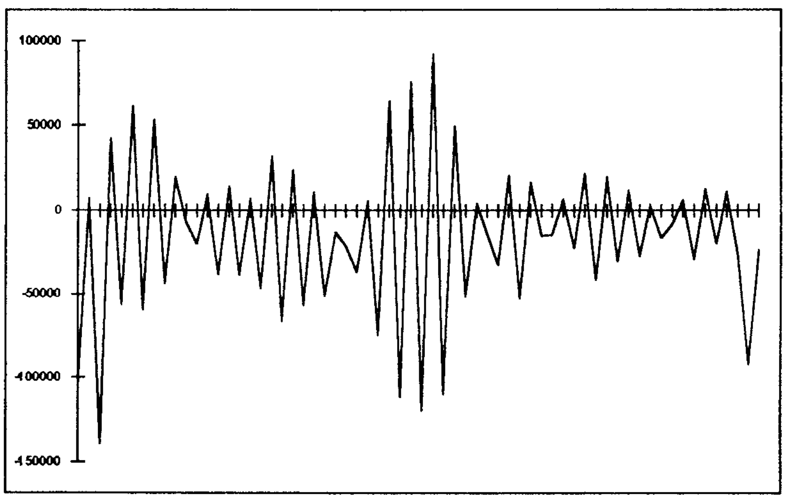

Figure 5. 12: Highpass filtered sequence using NTT

The following figure shows the highpass filter in the frequency-domain. This particular highpass filter consists of some low frequency components and has a large portion of its components in the higher frequencies. Therefore, the filtered output is expected to contain mostly high frequency components and the mid-range frequencies will be absent or filtered off. This is shown in figure 5.14, which is the filtered output in the frequency-domain.

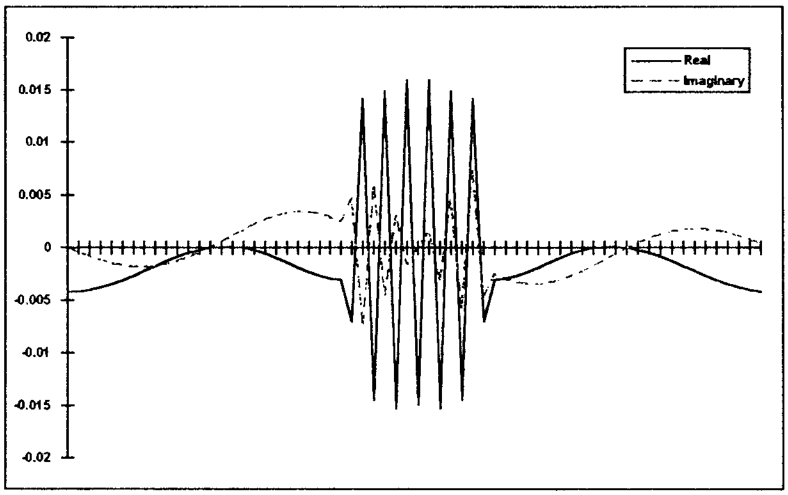

Figure 5. 13: Highpass filter sequence in frequency-domain

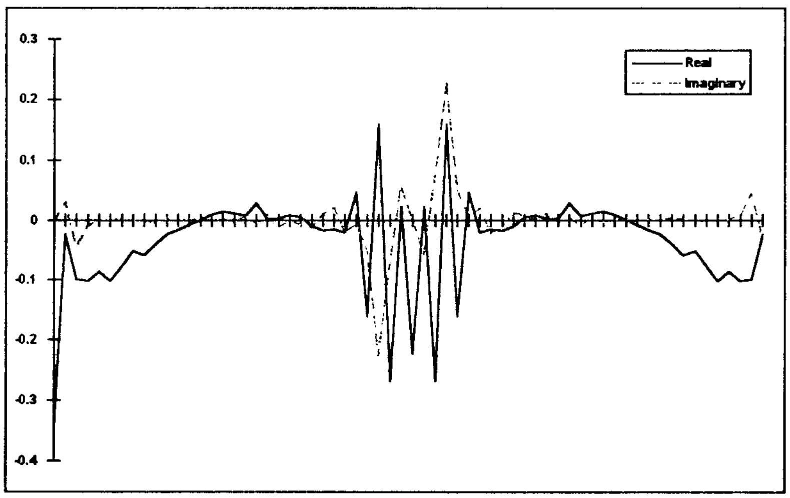

Figure 5. 14: Highpass filtered output sequence in frequency-domain

The results from the above data show that the filters perform as they are designed to do. The exact shape of both implementations shows consistency in the two implementations, therefore, laying the groundwork for comparison of both implementations.

5.2 Execution Time

The following table shows the number of clock cycle to execute each operation in both NTT and DFT implementations.

	NTT			DFT
Clock Cycle	Lowpass	Bandpass	Highpass	All cases
Filtering	16302	16308	16298	6832
Transformation	8806	8806	8806	3560

Table 5. 1: Execution clock cycle

The data indicates significantly lower number of clock cycles for DFT implementation. The conclusion is, therefore, favorable towards DFT implementation compared to NTT implementation in this environment.

A similar conclusion prevails when using the number of operation as a measure. Table 5.2 shows that the NTT implementation accounts for a significantly larger number of operations, which include addition, subtraction, multiplication, 'and', 'or', 'xor', 'not', and shift operations.

	NTT			DFT
No. Operations	Lowpass	Bandpass	Highpass	All cases
Filtering	15112	15085	15077	3852
Transformation	5159	5159	5159	1790

Table 5. 2 Number of operations

This finding differs from many research literatures on NTT. The difference is due to two reasons. Firstly, the DSP chip in this research environment performs real multiplications with 1 clock cycle, which are the same as real additions and subtractions. This is contrary to the general belief that multiplication is more expensive than other operations. Secondly, even though complex operations take a few real operations to complete, they are still better off compared to operations in modulo arithmetic. The reason is that existing DSPs do not perform modulo arithmetic. Therefore, for example, a simple modulo addition will consist of a number of integer additions, shifts, and 'xor' operations. Until there is efficient hardware implementation to modulo arithmetic in the market, NTT is not a better solution compared to DFT.

5.3 Memory Requirement

Memory requirement in the case of DFT requires a 32-bit word for the real and the imaginary portion of a complex number. Therefore, it requires 2 memory locations for each data point. An additional array of $1.25n$ words, where n is the sequence length, is required for the DFT. This array stores the scaling factors, the ω's, in order to increase computational speed to complete the DFT. While FNT requires 33 bit data point, it takes up 2 memory locations for each data point in practice. The msb of a data point will occupy one memory location, which is a waste of memory resources. However, future development may lead to more efficient use of memory. In view of this, the FNT uses less memory space compared to the DFT.

5.4 Summary

In summary, this chapter verifies the correctness of the implementations of DFT and FNT in this study. The time, or number of clock cycle taken in the FNT implementation is significantly higher than that of the DFT implementation. When counting the number of operations needed to perform the two operations, FNT shows the same disappointing results. The amount of memory requirement for the FNT, however, is lesser compared to the DFT.

CHAPTER 6

6. Analysis and Conclusion

6.1 Analysis

6.1.1 Speed

Referring to table 5.1 and table 5.2 on the number of execution clock cycle and the number of operation respectively, this study consistently indicates the poorer execution time performance of the FNT. This is contrary to the general belief that the FNT takes shorter computational time.

Agawal and Burrus commented in their 1974 paper [19], "To compute the convolution using the FFT, most of the time is taken in computing the complex multiplications required to compute the transform." They also said that, "A comparison with the RT reveals that these complex multiplications are replaced by bit shifts and subtractions which are much faster operations." This assumption that complex multiplications have poorer time performance than bit shifts and subtractions was valid in the 1970's. It is rightly so, especially when their comparison was verified on the IBM 370/155. Nevertheless, technological advances over the past twenty years have changed the validity of these statements. The results from this study strongly indicate the repercussion from this improved technology. There are two reasons the results from this study deviate from Agawal and Burrus's conclusion.

Firstly, multiplications are no longer expensive and time consuming operations. In the TMS320C30 hardware environment where this study does most of its computation, multiplications take only one clock cycle. This is the same number of clock cycle as operations like additions, subtractions, bit shifts, etc. One complex multiplication, therefore, takes only six clock cycles; four multiplications, one addition, and one subtraction. Compare this to implementing diminished-one multiple-of-two operation in the TMS320C30 environment, which takes three shifts, one 'xor', and an addition. This will take five clock cycles. Multiplication is now comparable to bit shifts and subtractions in terms of time complexity.

Secondly, the traditional practice of considering only multiplications in time complexity analysis is not justifiable. This is because multiplications need to be of equal weighting with the other arithmetic operations in time complexity analysis, since they all take the same amount of time to execute. All types of operations need to be considered in the analysis. As such time complexity analysis will also include additions, subtractions, bit shifts, etc. Since these operations are taken into consideration, and the FNT uses a lot of such operations, the time complexity of the implementation of the FNT becomes much higher compared to the DFT. The FNT clearly stands out to be more time consuming. This is verified in table 5.1 and table 5.2.

6.1.2 Memory Requirement

Memory requirement for the DFT is more than that of the FNT implementations. Since the DFT is in the complex field, it requires one word each for the real and the imaginary component. Therefore, a total of two words is needed for each element in the transform sequence. In addition to this memory requirement, the DFT implementation requires an array of scaling factors, the ω's, to speed up execution time. This array occupies $2.5n$, where n is the sequence length, words. In this implementation, the word length of the FNT is 33 bits. Since a word occupies 32 bits in this case, each element in the transform sequence for the FNT requires 2 words. Future work may aim at improving space complexity of FNT. This is because most of the bits in the most significant word are not needed. Therefore, the implementation of FNT in this study requires less memory space as the DFT implementation.

6.1.3 Accuracy

The most attractive advantage of the FNT is the degree of accuracy it offers in computing convolution. There are two sources of error in digital signal processing. They are the quantization error and the computation roundoff error.

When sampling a signal, the equipment used introduces limitations caused by the finite precision of the arithmetic units and the limited capacity of the memory. Since the sampling devices and the memory have limited word length, the sampled data in the memory represent approximations to the actual signal. These are the causes for quantization errors. Both the DFT and the FNT suffer from quantization errors. For all practicality, it is unavoidable. The degree of approximation depends on the word length of the equipment used.

Among the limitations of the FNT is the requirement that all values of the final outcome for the transform must not exceed the modulo. This limitation requires scaling of the input data in order to prevent any overflow. Errors are introduced in this scaling process. However, there are no computational roundoff errors. This is because the transforms are computed in rings of integers modulo a Fermat number. In computing the DFT, some processes of additions, subtractions, and multiplications increases the number of bits in representing the resulting complex numbers of these operations. However, the number of bits available remains unchanged within the machines. This makes it necessary to limit the wordlength throughout the calculations. As such, rounding off the results from the operations becomes necessary too. Roundoff errors occur at every operation that produces results exceeding the wordlength of the machine. Therefore, errors can be introduced at all stages of computing the transform. This lack of roundoff errors during computation for the FNT is an advantage over the DFT.

The FNT has its advantages and disadvantages. When execution speed is crucial, the DFT is a better choice than the FNT. With the current hardware technology, implementing the FNT will increase the amount of time taken to perform the transform. However, when accuracy is important, the FNT offers highly accurate results compared to the DFT.

6.2 Conclusion

The field of signal processing has matured tremendously over a little less than two centuries, most especially during the last three decades. Coupled with the advances in microelectonic industry, the many advantages in digital filtering techniques have spurred the study of digital signal processing.

Over this time, the paper published by Cooley and Tukey set an important landmark. Their method increased computational efficiency of calculating the Fourier transform many folds. Other researchers have also introduced improved algorithms to their method over the last few decades. However, in 1974, Rader introduced a new transform, generally referred to as the number theoretic transform, in the hope of eradicating the shortcomings of the Fourier transform. Number theoretic transform uses integer modulo arithmetic instead of complex arithmetic. Thus, it is possible to perform the transform without the use of multiplication, which is a traditionally expensive operation using the Fourier transform.

This study takes a look at the many claims of number theoretic transform, specifically FNT. Comparisons in execution time, number of operation, and memory requirement by both DFT and FNT using radix-2 FFT algorithm are done. All these methods are implemented on the Texas Instruments' TMS320C3X digital signal processing chip. This study also presents an analytical comparison of computational accuracy of the DFT and the FNT.

Results from this study show that the execution time, or the number of clock cycles (Table 5.1) for the FNT implementation is considerably higher than that of the DFT implementation. In order to have a fair comparison, the study also look at the number of operation for both implementations. Again, FNT shows a discouraging higher value (Table 5.2). However, based on analytical analysis, the FNT requires lesser memory space when compared to the DFT. In terms of computational accuracy, the FNT also performs much better. The only error introduced in implementing FNT is during input quantization [21]. There are no computational roundoff errors in FNT. This is not the case for DFT.

This research also introduces modified diminished-one number operations. The results from this study show the feasibility of these operations. Future development in this area should include hardware implementation of these operations. An advantage of these operations over Leibortze's original diminished-one number system proposal is the lack of number system conversion for both addition and subtraction operations. Both these operations also remove the need to check for the number zero. Scaling or multiplication by powers of 2 operation becomes a one-step operation instead of multi-step operation.

BIBLIOGRAPHY

1. C.M. Rader, "Discrete Convolution via Mersenne Transfrom," IEEE Transactions on Computers, col. c-21, pp. 1269-1273, December 1972.
2. J.W. Cooley and J.W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," Math. Comprt. vol 19, pp. 297-301, April 1965.
3. J.W. Nilson, "Electric Circuits, second edition," Addison-Wesley Publishing Co., Inc., 1986, Ch. 18, pp. 703-704.
4. P. Duhamel and M. Vetterli, "Fast Fourier Transform: A Tutorial Review And a State of The Art," Signal Processing, vol. 19, pp. 259-299, April 1990.
5. H. Burkhardt, "Trigonometrische Interpolation," Encyklopädie der Mathematischen Wissenschften, vol. 2, part 1, 1st half, ch. 9, pp. 643-693, Leipzig: B.G. Teubner, 1899-1916.
6. H.H. Goldstine, "A History of Numerical Analysis from the 16th Through the 19th Century," New York, Heidelberg, and Berlin: Springer-Veriag, pp. 249-253, 1977.
7. M.T. Heideman, "Gauss and the History of the Fast Fourier Transform," IEEE Transactions on

Acoustics,Speech, and Signal Processiing, vol. 1, No. 4, pp. 14-21, October 1984.

8. E.T. Whittaker and G. Robinson, "The Calculus of Observations," Blackie and Sons, Limited, London, pp. 260-284, 1924.

9. I.J. Good, "The Interaction Algorithm and Practical Fourier Analysis," J.R. Statist. Soc. B, vol. 20, No. 2, pp. 361-372, 1958. Addendum in J.R. Statist. Soc. B, vol. 22, No. 2, pp. 272-275, 1960

10. D.F. Elliott and K.R. Rao, "Fast Transforms: Algorithms, Analysis, Applications," Academic Press, Inc, pp. 105, 1982.

11. C.M. Rader, "Discrete Fourier Transforms When The Number of Data Samples is Prime," Proceedings of IEEE, vol. 56, pp. 1008-1107, 1968.

12. S. Winograd, "Some Bilinear Forms Whose Multiplicative Complexity Depends on The Field of Constants", Math. Systems Theory, vol 10, No. 2, pp 169-180, 1977.

13. G.D. Bergland, "A Fast Fourier Transform Algorithm Using Base 8 Iterations," Math. Comp., vol. 22, No. 2, pp. 275-279, April 1968.

14. P. Duhamel and H. Hollmann, "Split-radix FFT Algorithm," Electronics Letters, vol. 20, No. 1, pp. 14-16, January 1984.

15. J.B. Martens, "Recursive Cyclotomic Factorization - A New Algorithm for Calculating the Discrete Fourier Transform," IEEE Transactions on Accoustics, Speech, and Signal Processing, vol. ASSP 32, No. 4, pp. 750-761, August 1984.

16. R. Stasinski, "Asymmetric Fast Fourier Transform for Real and Complex Data," IEEE Transactions on Accoustics, Speech, and Signal Processing, submitted.

17. M. Vetterli and H.J. Nussbaumer, "Simple FFT and DCT Algorithm With Reduced Number of Operations," Signal Process. vol. 6, No. 4, pp. 267-278, August 1984.

18. J.M. Pollard, "The Fast Fourier Transform In a Finite Field," Math. Comp., vol. 25, pp. 365-374, April 1971.

19. R.C. Argarwal and C.S. Burrus, "Fast Convolution Using Fermat Number Transforms With Application To Digital Filtering," IEEE Trans. Acoust. Speech, Signal Processing, vol. ASSP-22, pp. 87-97, April 1974.

20. J.H. McClellan, "Hardware For the Fermat Number Transform," IEEE Trans. Acoust. Speech, Signal Processing, vol. ASSP-24, No.3, pp. 216-225, June 1976.

21. L.M. Leibowitz, "A Simplified Binary Arithmetic For the Fermat Number Transform," IEEE Trans. Acoust. Speech, Signal Processing, vol. ASSP-24, No. 5, pp. 356-359, October 1976.

22. P.R. Chevillat, "Transform-Domain Digital Filtering with Number Theoretic Transforms and Limited Word Lengths," IEEE Trans. Acoust. Speech, Signal Processing, vol. ASSP-26, No. 4, pp. 284-290, August 1978.

APPENDIX

APPENDIX A

Software User's Instruction

System Requirements:

1. IBM PC or PC compatibles with Intel 80386 or above microprocessor.
2. Texas Instruments' TMS320C30 Evaluation Module (EVM).
3. DOS 5.0 or above.
4. Windows 3.1 or above.
5. Borland custom control tools library bwcc.dll

Starting The Software:

1. Ensure that TI's TMS320C30 EVM is properly installed.
2. Ensure that the client program 'child.out', the EVM loader 'evmload.exe', and the three initialization files 'resvct.001', 'resvct.002', 'resvct.004' are in the same directory as 'host.exe'.
3. Run Windows 3.1
4. Select 'Run' from Program Manager's 'File' Menu
5. Type 'host.exe' and punch the Enter key.
6. 'host.exe' will load the TMS320C30 EVM client program, 'child.out', initiate and start the program running in the EVM.
7. 'host.exe' will then return to the Windows environment.

8. If 'host.exe' couldn't locate the TMS320C30 EVM, or the client program 'child.out', it will inform the user and prompt for further action.

Moving Around The Software:

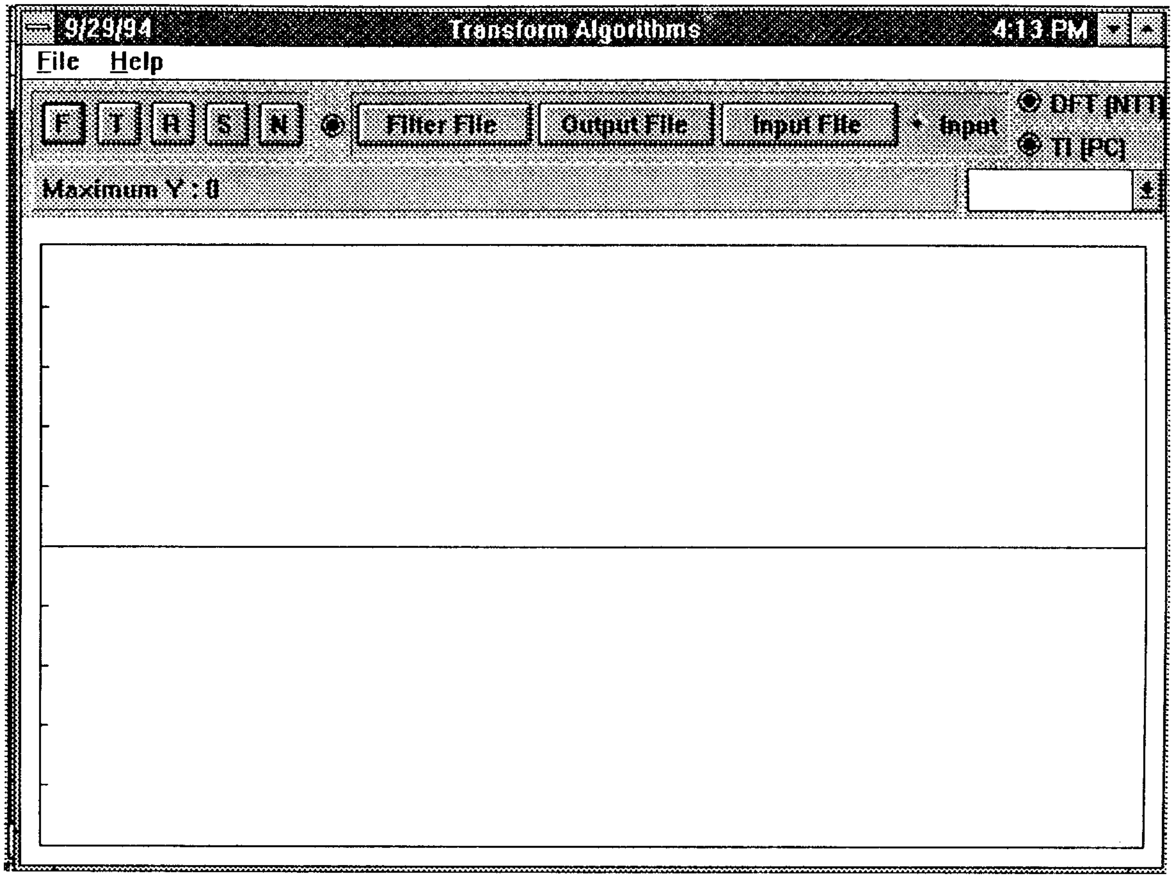

Figure 1: Host program's interface

1. Item 'Convert', under the 'File' Menu, allows user to convert filter files computed from Hypersignal to data files in the format for this program.

2. item 'About', under the 'Help' Menu, displays information about this program.

3. Button 'F' is the filter button. User must select filter, output, and input files before this function

will work. After the user points the mouse pointer at this button and click it once, the program will filter the input file using the filter file selected.

4. Button 'T' is the transform button. Selecting this function will start a transform operation on the input file using the type of transform selected. The selection of the type of transform is discussed below.
5. Button 'R' is the reset button. This function will reset the TMS320C30 EVM client program.
6. Button 'S' is the sample button. The EVM will sample input signal from the EVM input port. This 64-word data is then sent to the PC. The data is then saved using the output filename.
7. Button 'N' is the name button. When selected, the message box just below the buttons will display all the selected filenames.
8. When 'Filter File' button is selected, the user is prompted for a filter filename from a pop up window.
9. When the 'Output File' button is selected, the user is prompted for an output filename from a pop up window.
10. When the 'Input File' button is selected, the user is prompted for an input filename from a pop up window.
11. The 'Input' button, when selected, enables an input file to be used. Otherwise, the EVM will sample from

its input port and use the data collected as input data for any operation.

12. When the 'DFT(NTT)' button is highlighted, all operations are performed using DFT algorithm. Otherwise, all operations are performed using NTT algorithm.

13. When the 'TI(PC)' button is highlighted, all operations are performed using the TI's TMS320C30 EVM. Otherwise, all operations are performed using the PC.

14. Below all the buttons is the message box. This is the area where the program will display messages for the user.

15. To the right of the message box is the display list box. Pointing and clicking on the arrow will display a list of items. Selecting an item selected from the list box will display the item in the display area.

16. The display area is below the menu and toolbar. This is the area where data is displayed.

APPENDIX B

Program Source Code

```
        .width          104
        .length         2000
        .global         SINE, N, M
        .global         iinverse, cinverse

        .data

*===================================================================
* ADDRESSES OF CONSTANTS
*===================================================================
fftsize                 .word   N
logsize                 .word   M
sinetab                 .word   SINE
IINVERSE                .word   iinverse
CINVERSE                .word   cinverse

*===================================================================
* CONSTANT VARIABLES
*===================================================================
mask                    .word   0ffffh      ;mask for ffnt
mid_carry               .word   010000h     ;carry over
shiftmask               .word   0ffffffffh ;mask for power of 2 multiplication

        .text
FP      .set            AR3
        .ref            _com_saddrl
*===================================================================
* GLOBAL DECLARATION OF FUCTIONS
*===================================================================
        .def            _fft
        .def            _fscale
        .def            _ifft
        .def            _fmult
        .def            _ffnt
        .def            _nscale
        .def            _iffnt
        .def            _nmult
        .def            _bit_reverse
```

```
*======================================================================*
* DFT using DIF FFT algorithm                                          *
*======================================================================*
*                                                                      *
*     This routine performs DFT using DIF FTT algorithm.               *
*                                                                      *
************************************************************************
_fft:
      push        FP                  ;initialize registers for
      ldi         SP,FP               ;c function interface
      ldi         *-FP(2),ar0         ;load argument
      sti         ar0,@_com_saddrl    ;initialize source address

      ldi         @fftsize,ir0        ;ir0 = 2m
      ldi         ir0,ar2
      lsh         1,ir0
      ldi         ir0,ar3
      lsh         -2,ar2              ;ar2 = N/4
      ldi         1,ir1               ;N/m
      ldi         @logsize,r6         ;log N

floop1f:
      lsh         -1,ar3              ;m
      ldi         ar3,r7

      ldi         @_com_saddrl,ar0    ;ar0->x[0]
      addi        ar3,ar0,ar1         ;ar1->x[m/2]

      ldi         ir1,rc              ;repeat N/m times
      subi        1,rc
      rptb        floop1nf

      addf        *ar1,*ar0,r1        ;r1 <- x[k] + x[k+m/2]
      subf        *ar1++,*ar0++,r2    ;r2 <- x[k] - x[k+m/2]
      addf        *ar1,*ar0,r3        ;r1 <- y[k] + y[k+m/2]
      subf        *ar1,*ar0,r4        ;r2 <- y[k] - y[k+m/2]
      stf         r3,*ar0--           ;ar0 -> y[k] + y[k+m/2]
      ||stf       r4,*ar1--           ;ar1 -> y[k] - y[k+m/2]
floop1nf: stf     r1,*ar0++(ir0)      ;ar0 -> x[k] + x[k+m/2]
      ||stf       r2,*ar1++(ir0)      ;ar1 -> x[k] - x[k+m/2]

      ldi         @sinetab,ar4        ;ar4 -> sin(0)
      mpyi        ir1,r7,r1
      lsh         -1,r1
      addi        r1,ar4              ;ar4 -> sin[m/2]
floop2f:
      subi        2,r7                ;m/2 - 1
      bz          floopnf

      ldi         @_com_saddrl,ar0    ;ar0 -> x[0]
      addi        r7,ar0              ;ar0 -> x[k]
      addi        ar3,ar0,ar1         ;ar1 -> x[k+m/2]

      ldf         *--ar4(ir1),r5      ;r5 <- sin(2*pi*j/m)
```

```
        addi        ar2,ar4,ar5         ;ar5 -> cos(2*pi*j/m)

        ldi         r1,rc               ;repeat N/m times
        subi        1,rc
        rptb        floop2nf

        subf        *ar1,*ar0,r3        ;r3 <- x[k] - x[k+m/2]
        subf        *+ar1,*+ar0,r4      ;r4 <- y[k] - y[k+m/2]
        mpyf        r3,r5,r0            ;r0 <- (x[k]-x[k+m/2])sin(2*pi*j/m)
        ||addf      *+ar1,*+ar0,r2      ;r2 <- y[k] + y[k+m/2]
        mpyf        *ar5,r4,r1          ;r1 <- (y[k]-y[k+m/2])cos(2*pi*j/m)
        ||stf       r2,*+ar0            ;y[k] <- y[k] + y[k+m/2]
        subf        r0,r1               ;r1 = b cos() - a sin()
        mpyf        *ar5,r3,r0          ;r0 <- (x[k]-x[k+m/2])cos(2*pi*j/m)
        ||stf       r1,*+ar1            ;y[k+m/2] <- b cos() - a sin()
        mpyf        r4,r5,r1            ;r1 <- (y[k]-y[k+m/2])sin(2*pi*j/m)
        ||addf      *ar1,*ar0,r2        ;r2 <- x[k] + x[k+m/2]
        addf        r0,r1               ;r1 = a cos() + b sin()
floop2nf:
        stf         r2,*ar0++(ir0)      ;x[k] = x[k] + x[k+m/2]
        ||stf       r1,*ar1++(ir0)      ;x[k+m/2] = a cos() + b sin()
        br          floop2f
floopnf:
        lsh         -1,ir0              ;m
        lsh         1,ir1               ;N/m
        subi        1,r6                ;decrement main loop count
        bp          floop1f

        pop         FP
        rets
```

```
*==================================================================*
* Floating point scaling by 1/N                                    *
*==================================================================*
*                                                                  *
*     This routine multiply every elements in the input sequence by *
*   the inverse of the transform length.  The address of the input  *
*   sequence is send to this function at location FP-2.  The value of *
*   inverse of the transform length is stored at the memory location *
*   specified at CINVERSE                                           *
*                                                                  *
********************************************************************
_fscale:
      push        FP                  ;initialize registers for
      ldi         SP,FP               ;c function interface
      ldi         *-FP(2),ar0         ;load argument
      sti         ar0,@_com_saddrl

      ldi         @CINVERSE,ar1       ;ar1 gets address of constant 1/N
      ldf         *ar1,r0             ;r0 <- 1/N
      ldi         @fftsize,rc         ;setup repeat loop number
      mpyi        2,rc
      subi        1,rc
      rptb        fscaling            ;begin loop
      mpyf        *ar0,r0,r1          ;multiply by 1/N
fscaling: stf     r1,*ar0++           ;store result

      pop         FP
      rets
```

```
*=====================================================================*
* IDFT using DIT FFT algorithm                                         *
*=====================================================================*
*                                                                      *
*     This routine performs IDFT using DIT FFT algorithm.  The         *
*  address of the memoty location of the input sequence is stored     *
*  at FP-2.                                                            *
*                                                                      *
***********************************************************************
_ifft:
      push        FP                  ;initialize registers for
      ldi         SP,FP               ;c function interface
      ldi         *-FP(2),ar0         ;load argument
      sti         ar0,@_com_saddrl

      ldi         2,ir0               ;2m
      ldi         @fftsize,ir1        ;N/m
      ldi         ir1,ar6
      lsh         -2,ar6              ;N/4
      ldi         @logsize,r7         ;log N

floop1i:
      ldi         ir0,ar3             ;m/2
      ldi         ar3,r6
      lsh         1,ir0               ;m
      lsh         -1,ir1              ;N/m

      ldi         @_com_saddrl,ar0    ;ar0->x[0]
      addi        ar3,ar0,ar1         ;ar1->x[m/2]

      ldi         ir1,rc              ;repeat N/m times
      subi        1,rc
      rptb        floop1ni

      addf        *ar0,*ar1,r0        ;r0=x[k]+x[k+m/2]
      subf        *ar1++,*ar0++,r1    ;r1=x[k]-x[k+m/2]
      addf        *ar1,*ar0,r2        ;r2=y[k]+y[k+m/2]
      subf        *ar1,*ar0,r3        ;r3=y[k]-y[k+m/2]
      stf         r2,*ar0--           ;y[k] = r2
      ||stf       r3,*ar1--           ;y[k+m/2] = r3
floop1ni:
      stf         r0,*ar0++(ir0)      ;x[k] = r0
      ||stf       r1,*ar1++(ir0)      ;x[k+m/2] = r1

      ldi         @sinetab,ar4        ;sin(0)
      mpyi        ir1,r6,r1
      lsh         -1,r1
      addi        r1,ar4              ;ar4 -> sin[m/2]
floop2i:
      subi        2,r6                ;m/2 - 1
      bz          floopni

      ldf         *--ar4(ir1),r5      ;r5 <- sin(2*pi*j/m)
      addi        ar6,ar4,ar5         ;ar5 -> cos(2*pi*j/m)
```

```
        ldi         @_com_saddr1,ar0  ;ar0->x[0]
        addi        r6,ar0            ;ar0 -> x[k]
        addi        ar3,ar0,ar1       ;ar1 -> x[k+m/2]

        ldi         ir1,rc            ;repeat N/m times
        subi        1,rc
        rptb        floop2ni

        mpyf        *+ar1,*ar5,r3     ;r3 <- y[k+m/2]*cos(2*pi*k/m)
        mpyf        *ar1,*ar4,r0      ;r0 <- x[k+m/2]*sin(2*pi*k/m)
        mpyf        *ar1,*ar5,r1      ;r1 <- x[k+m/2]*cos(2*pi*k/m)
        ||addf      r0,r3,r2          ;r2 <- b cos() + a sin()
        mpyf        *+ar1,r5,r0       ;r0 <- y[k+m/2]*sin(2*pi*k/m)
        ||addf      *+ar0,r2,r3       ;r3 <- y[k] + b cos() + a sin()
        subf        r0,r1,r0          ;r0 <- a cos() - b sin()
        subf        r2,*+ar0,r1       ;r1 <- y[k] - b cos() - a sin()
        ||stf       r3,*+ar0          ;y[k] <-  y[k] + b cos() + a sin()
        addf        *ar0,r0,r1        ;r1 <- x[k] + a cos() - b sin()
        ||stf       r1,*+ar1          ;y[k+m/2] <- y[k]-b cos()-a sin()
        subf        r0,*ar0,r1        ;r1 <- x[k]-a cos()+b sin()
        ||stf       r1,*ar0++(ir0)    ;x[k] = x[k] + a cos() - b sin()
floop2ni:
        stf         r1,*ar1++(ir0)    ;x[k+m/2] <- x[k]-a cos()+b sin()

        br          floop2i
floopni:
        subi        1,r7              ;decrement main loop count
        bnz         floop1i

        pop         FP
        rets
```

```
*===================================================================*
* Complex multiplication                                            *
*===================================================================*
*                                                                   *
*     This routine performs a term-wise multiplication of the       *
*  transformed input sequence to the transformed filter sequence.   *
*                                                                   *
*********************************************************************
_fmult:
      push        FP                  nitialize registers for
      ldi         SP,FP               ;c function interface
      ldi         *-FP(2),ar0         ;load input address
      ldi         *-FP(3),ar1         ;load filter address
      ldi         *-FP(4),ar2         ;load output address

      ldi         2,ir1

      ldi         @fftsize,rc         ;setup repeat loop number
      subi        1,rc
      rptb        fmloop              ;begin repeat loop
      mpyf        *+ar0,*ar1,r3       ;r3 <- y1*x2
      mpyf        *ar0,*+ar1,r0       ;r0 <- x1*y2
      mpyf        *+ar0,*+ar1,r0      ;r0 <- y1*y2
      ||addf      r0,r3,r2            ;r2 <- y1*x2 + x1*y2
      mpyf        *ar0++(ir1),*ar1++(ir1),r1    ;r1 <- x1*x2
      subf        r0,r1,r0            ;r0 <- x1*x2 - y1*y2
      stf         r2,*+ar2            ;store results
fmloop: stf       r0,*ar2++(ir1)

      pop         FP
      rets
```

```
*==================================================================*
* FNT using DIF FFT algorithm                                      *
*==================================================================*
*                                                                  *
*     This routine performs FNT using DIF FFT algorithm.  All      *
*   operations use the modified diminished-one number operations   *
*                                                                  *
********************************************************************
_ffnt:
      push        FP                  ;initialize registers for
      ldi         SP,FP               ;c function interface
      ldi         *-FP(2),ar0         ;load argument
      sti         ar0,@_com_saddrl

      ldi         @fftsize,ir0        ;index for increment of m
      ldi         ir0,ar4             ;N/2, for scaling comparison
      lsh         -1,ar4
      lsh         1,ir0               ;double ir0 for 2 word data
      ldi         ir0,ar5             ;m/2, index to 2nd number
      ldi         @shiftmask,ar7      ;ar7 <- 0ffffffffh
      ldi         1,ir1               ;N/m, power of 2, # bits to shift
      ldi         @logsize,r0         ;log N, number of main loops

nloopf: lsh       -1,ar5              ;update, calculate m/2
      ldi         ar5,r7              ;counter from m/2-1 to 1

      ldi         @_com_saddrl,ar0    ;ar0 -> x[0]
      addi        ar5,ar0,ar1         ;ar1 -> x[m/2]

      ldi         ir1,rc              ;repeat N/m times
      subi        1,rc
      rptb        nloop1nf
      subi        *ar1,*ar0,r3        ;r3 <- binary diff, low word
      subb        *+ar1,*+ar0,r4      ;r4 <- binary diff, high word

      bnnd        naddlf              ;if diff is positive
      ldi         1,ar2               ;register ar2 <- 1
      ldi         0,ar3               ;register ar3 <- 0
      nop

      addi        ar2,r3              ;if negative, low word is d-1,
      and         ar2,st,r4           ;thus d-1 mod operation

naddlf: addi      *ar1,*ar0,r1        ;r1 <- binary sum, low word
      addc        *+ar1,*+ar0,r2      ;r2 <- binary sum, high word

      bzd         nloop1nf            ;if sum < 2^32, store result
      sti         r4,*+ar1            ;save high word diff
      ||sti       r3,*ar1++(ir0)      ;save low word diff, and increment
                                      ;index
      subi        ar2,r1,r3           ;convert to d-1, low word
      subb        ar3,r2,r4           ;convert to d-1, high word

      xor         ar2,r4              ;A+B = [(A+B)-1]+1. Thus d-1
```

```
        addi        r4,r3,r1            ;mod operation
        and         ar2,st,r2           ;save carry

nloop1nf:
        sti         r2,*+ar0            ;save high word sum
        ||sti       r1,*ar0++(ir0)      ;save low word sum, and increment
index

nloop2f:
        subi        2,r7                ;decrement r7
        bzd         nloopnf             ;end routine if zero
        ldi         1,ar2               ;register ar2 <- 1
        ldi         @_com_saddr1,ar0    ;ar0 -> x[0]
        addi        r7,ar0              ;ar0 -> x[k]

        addi        ar5,ar0,ar1         ;ar1 -> x[k+m/2]
        ldi         ir1,rc              ;repeat N/m times
        subi        1,rc
        rptb        nloop2nf

        addi        *ar1,*ar0,r1        ;r1 <- binary sum, low word
        addc        *+ar1,*+ar0,r2      ;r2 <- binary sum, high word

        bzd         ndiff2f             ;if sum < 2^32, store result
        ldi         0,ar3               ;register ar3 <- 0
        subi        ar2,r1,r3           ;convert to d-1, low word
        subb        ar3,r2,r4           ;convert to d-1, high word

        xor         ar2,r4              ;A+B = [(A+B)-1]+1. Thus d-1
        addi        r4,r3,r1            ;mod operation
        and         ar2,st,r2           ;save carry

ndiff2f:subi        *ar1,*ar0,r3        ;r3 <- binary diff, low word
        subb        *+ar1,*+ar0,r4      ;r4 <- binary diff, high word

        bnd         nscaledf            ;if diff is negtive, r3 is in d-1
        sti         r2,*+ar0            ;save high word
        ||sti       r1,*ar0++(ir0)      ;save low word, and increment index
        mpyi        r7,ir1,r1           ;# of bit shift
        lsh         -1,r1               ;adjust bit shift for double
                                        ;counting

        subi        ar2,r3              ;convert to d-1 low word
        subb        ar3,r4              ;convert to d-1 high word
        bnn         nscaledf            ;if not zero, scale

        brd         ndonef              ;if equal zero, make r4 = 0
        ldi         0,ar2
        nop
        nop

nscaledf:
        subi        ar4,r1,r2           ;X - N/2 shift, shift d2
```

```
        bnd         nscalef            ;if shift less than N/2, next shift
                                       ;X
        lsh         r2,ar7,r6          ;shift mask by X - N/2
        lsh         r2,r3,r5           ;shift d-1 data by X - N/2
        xor         r5,r6              ;complement r5

        subi        @fftsize,r1        ;shift more than N/2, adjust next
                                       ;shift
nscalef:                               ;X - N
        lsh         r1,r3              ;left shift X or X-N bits
        or          r6,r3              ;or register to get final d-1 result
ndonef: addi        1,r3               ;convert to binary low
        and         ar2,st,r4          ;convert to binary high

nloop2nf:
        sti         r3,*ar1++(ir0)     ;store results
        ||sti       r4,*+ar1
        br          nloop2f

nloopnf:
        lsh         -1,ir0             ;update ir0, m/2
        lsh         1,ir1              ;update ir1, N/m
        subi        1,r0               ;update r0, decrement
        bnz         nloopf

        pop         FP
        rets
```

```
*==================================================================*
* Diminished-one scaling by inverse of transform length N          *
*==================================================================*
*                                                                  *
*     This routine multiplies all terms in the input sequence by a *
*  scaling factor specified in the memory location IINVERSE.       *
*     Since 2^b is always the multiplicative inverse of itself     *
*  modulo 2^b + 1, multiplicative inverse of all other numbers     *
*  require only b bits.  The (b+1)th bit of the multiplicative     *
*  inverse of N is assumed 0 at all times to simplify calculation. *
*     Ignore all calculation of Z2, since the value of Z2 does     *
*  not effect the result.                                          *
*                                                                  *
********************************************************************
_nscale:
      push        FP                 ;initialize registers for
      ldi         SP,FP              ;c function interface
      ldi         *-FP(2),ar1        ;load argument

      ldi         @IINVERSE,r2       ;load multiplication inverse of ffnt
                                     ;size
      ldi         @fftsize,rc        ;setup repeat counter
      ldi         @mask,ar7          ;ar7 <- 0ffffh
      ldi         4,ir0              ;constant 4
      ldi         2,ir1              ;constant 5
      ldi         @mid_carry,ar6

      ldi         16,ar2             ;constant shift left 16 bits
      ldi         -16,ar3            ;constant shift right 16 bits
      ldi         1,ar4              ;constant 1
      subi        1,rc

      rptb        nscaling
      ldi         *ar1,r1            ;r1 <- Y1 Y0

      lsh         ar3,r1,r7          ;r7 <- Y1
      and         ar7,r1,r6          ;r6 <- Y0
      ldi         0,r1               ;r1 <- X2 * Y1 Y0 = 0
      ldi         *+ar1,r3           ;r3 <- Y3 Y2

      bpd         nmults             ;if Y2 = 1
      ldi         r2,r0              ;r0 <- multiplicative inverse
      lsh         ar3,r0,r5          ;r5 <- X1
      and         ar7,r0,r4          ;r4 <- X0

      ldi         0,r0               ;r0 <- Y2 * X1 X0 = 0
nmults: addi      r0,r1              ;r1 <- X1 X0 + Y1 Y0

      mpyi        r4,r6,r0           ;r0 <- X0 * Y0
      mpyi        r5,r6,r3           ;r3 <- X1 * Y0
      mpyi        r7,r4              ;r4 <- Y1 * X0

      addi        r4,r3              ;r3 <- (X1 * Y0) + (Y1 * X0)
```

```
        bncd            nmultsnxt
        lsh             ar3,r3,r4           ;r4 <- high word of r3
        lsh             ar2,r3              ;r3 <- low word of r3 left shifted
16 bits
        mpyi            r7,r5               ;r5 <- Y1 * X1

        addi            ar6,r5              ;r5 + 65536
nmultsnxt:
        addi            r3,r0               ;r0 gets low word of product
        and             ir0,st,r3           ;r3 gets zero flag
        addc            r5,r4               ;r4 gets partial high word of
product
        ldi             0,r5                ;r5 to receive carry over
        addi            r4,r1               ;r1 gets high word of product
        and             st,r3               ;r3 gets zero flag
        addc            0,r5                ;r5 gets carry
        and             st,r3               ;r3 gets zero flag
        bnz             nscaling

        not             r1                  ;(AB-1) = (AB) - 1, thus begin
residue
        addi            r1,r0               ;reduction
        and             ar4,st,r1           ;save carry
        xor             ar4,r1              ;complement carry
        addi            r1,r0               ;diminished one result

        addi            ar4,r0              ;convert to binary
        and             ar4,st,r1           ;save carry

nscaling:
        sti             r1,*+ar1
        ||sti           r0,*ar1++(ir1)      ;store result

        pop             FP
        rets
```

```
*====================================================================*
* IFNT using DIT FFT algorithm                                       *
*====================================================================*
*                                                                    *
*     This routine performs IFNT using DIT FFT algorithm.  The       *
*  starting address of the input sequence is stored in memory        *
*  location FP-2.                                                    *
*                                                                    *
**********************************************************************
_iffnt:
      push        FP                  ;initialize registers for
      ldi         SP,FP               ;c function interface
      ldi         *-FP(2),ar0         ;load argument
      sti         ar0,@_com_saddrl

      ldi         2,ir0               ;index for increment of m
      ldi         @fftsize,ir1        ;N/m, power of 2, # bits to shift
      ldi         ir1,ar4             ;N/2, for scaling comparison
      ldi         ir1,ar6             ;ar6 <- N
      lsh         -1,ar4
      ldi         @logsize,r0         ;log N, number of main loops
      ldi         @shiftmask,ar7      ;ar7 <- 0ffffffffh

nloopi:
      ldi         ir0,ar5             ;m/2, index to 2nd number
      ldi         ar5,r7              ;counter from m/2-1 to 1
      lsh         1,ir0               ;update ir0, m/2
      lsh         -1,ir1              ;update ir1, N/m

      ldi         @_com_saddrl,ar0    ;ar0 -> x[0]
      addi        ar5,ar0,ar1         ;ar1 -> x[m/2]

      ldi         ir1,rc              ;repeat N/m times
      subi        1,rc
      rptb        nloop1ni
      subi        *ar1,*ar0,r3        ;r3 <- binary diff, low word
      subb        *+ar1,*+ar0,r4      ;r4 <- binary diff, high word

      bnnd        naddli              ;if diff is positive
      ldi         1,ar2               ;register ar2 <- 1
      ldi         0,ar3               ;register ar3 <- 0
      nop

      addi        ar2,r3              ;if negative, low word is d-1,
      and         ar2,st,r4           ;thus d-1 mod operation

naddli: addi      *ar1,*ar0,r1        ;r1 <- binary sum, low word
      addc        *+ar1,*+ar0,r2      ;r2 <- binary sum, high word

      bzd         nloop1ni            ;if sum < 2^32, store result
      sti         r4,*+ar1            ;save high word diff
      ||sti       r3,*ar1++(ir0)      ;save low word diff, and increment
index
      subi        ar2,r1,r3           ;convert to d-1, low word
```

```
        subb        ar3,r2,r4           ;convert to d-1, high word

        xor         ar2,r4              ;A+B = [(A+B)-1]+1. Thus d-1
        addi        r4,r3,r1            ;mod operation
        and         ar2,st,r2           ;save carry

nloop1ni:
        sti         r2,*+ar0            ;save high word sum
        ||sti       r1,*ar0++(ir0)      ;save low word sum, and increment
index

nloop2i:
        subi        2,r7                ;decrement r7
        bzd         nloopni
        ldi         @_com_saddrl,ar0    ;ar0 -> x[0]
        addi        r7,ar0              ;ar0 -> x[k]
        addi        ar5,ar0,ar1         ;ar1 -> x[k+m/2]

        ldi         ir1,rc              ;repeat N/m times
        subi        1,rc
        rptb        nloop2ni
        subi        ar2,*ar1,r3         ;convert to d-1 low word
        subb        ar3,*+ar1,r4        ;convert to d-1 high word

        bnd         ndonei
        mpyi        r7,ir1,r1           ;# of bit shift
        lsh         -1,r1               ;adjust bit shift for double
                                        ;counting
        subi        r1,ar6,r1           ;r1 <- (-r1) mod FFTSIZE
        subi        ar4,r1,r2           ;X - N/2 shift, shift d2

        bnd         nscalei             ;if shift less than N/2, next shift
                                        ;X
        lsh         r2,ar7,r6           ;shift mask by X - N/2
        lsh         r2,r3,r5            ;shift d-1 data by X - N/2
        xor         r5,r6               ;complement r5

        subi        @fftsize,r1         ;shift more than N/2, adjust next
                                        ;shift

nscalei:                                ;X - N
        lsh         r1,r3               ;left shift X or X-N bits
        or          r6,r3               ;or register to get final d-1 result

ndonei: addi        ar2,r3,r5           ;convert to binary low
        addc        ar3,r4,r6           ;convert to binary high

        subi        r5,*ar0,r3          ;r3 <- binary diff, low word
        subb        r6,*+ar0,r4         ;r4 <- binary diff, high word

        bnnd        nadd2i              ;if diff is positive
        ldi         1,ar2               ;register ar2 <- 1
        ldi         0,ar3               ;register ar3 <- 0
        nop
```

```
        addi        ar2,r3              ;if negative, low word is d-1,
        and         ar2,st,r4           ;thus d-1 mod operation

nadd2i: addi        r5,*ar0,r1          ;r1 <- binary sum, low word
        addc        r6,*+ar0,r2         ;r2 <- binary sum, high word

        bzd         nloop2ni            ;if sum < 2^32, store result
        sti         r4,*+ar1            ;save high word diff
        ||sti       r3,*ar1++(ir0)      ;save low word diff, and increment
index
        subi        ar2,r1,r3           ;convert to d-1, low word
        subb        ar3,r2,r4           ;convert to d-1, high word

        xor         ar2,r4              ;A+B = [(A+B)-1]+1. Thus d-1
        addi        r4,r3,r1            ;mod operation
        and         ar2,st,r2           ;save carry

nloop2ni:
        sti         r2,*+ar0            ;save high word sum
        ||sti       r1,*ar0++(ir0)      ;save low word sum, and increment
index
        br          nloop2i

nloopni:
        subi        1,r0                ;update r0, decrement
        bnz         nloopi

        pop         FP
        rets
```

```
*=================================================================*
* Diminished-one multiplication                                   *
*=================================================================*
*                                                                 *
*    This routine performs term-wise modified diminished-one      *
*  multiplication on the transformed input and filter sequences.  *
*    Ignore all calculation of Z2, since the value of Z2 does not *
*  affect the result.                                             *
*    Both multiplier and multiplicant are assumed non-zero        *
*                                                                 *
*******************************************************************
_nmult:
      push         FP                 ;initialize registers for
      ldi          SP,FP              ;c function interface
      ldi          *-FP(2),ar0        ;load input address
      ldi          *-FP(3),ar1        ;load filter address
      ldi          *-FP(4),ar5        ;load output address

      ldi          @fftsize,rc        ;setup repeat counter

      ldi          16,ar2             ;16 bits left shift, get low word
      ldi          -16,ar3            ;16 bits right shift, get high word
      ldi          1,ar4              ;constant 1
      ldi          @mask,ar7          ;ar7 <- 0ffffh
      ldi          4,ir0              ;constant 4
      ldi          2,ir1              ;constant 2

      subi         1,rc

      rptb         nmloop
      ldi          *ar0,r0            ;r0 <- X1 X0
      ldi          *ar1,r1            ;r1 <- Y1 Y0
      ldi          *+ar0,r2           ;r2 <- X3 X2

      bpd          chknxt             ;if X2 = 1,
      lsh          ar3,r1,r7          ;r7 <- Y1
      and          ar7,r1,r6          ;r6 <- Y0
      nop

      ldi          0,r1               ;r1 <- X2 * Y1 Y0 = 0
chknxt: ldi        *+ar1,r3           ;r3 <- Y3 Y2

      bpd          nmultm             ;if Y2 = 1
      lsh          ar3,r0,r5          ;r5 <- X1
      and          ar7,r0,r4          ;r4 <- X0
      ldi          @mid_carry,ar6

      ldi          0,r0               ;r0 <- Y2 * X1 X0 = 0
nmultm: addi       r0,r1              ;r1 <- X1 X0 + Y1 Y0

      bncd         nmultnxt1
      mpyi         r3,r2              ;r2 <- X2 * Y2
      mpyi         r4,r6,r0           ;r0 <- X0 * Y0
      mpyi         r5,r6,r3           ;r3 <- X1 * Y0
```

```
        addi        ar4,r2              ;r2 + 1 if r0+r1 has carry
nmultnxt1:
        mpyi        r7,r4               ;r4 <- Y1 * X0

        addi        r4,r3               ;r3 <- (X1 * Y0) + (Y1 * X0)
        bncd        nmultnxt2
        lsh         ar3,r3,r4           ;r4 <- high word of r3
        lsh         ar2,r3              ;r3 <- low word of r3 left shifted
                                        ;16 bits
        mpyi        r7,r5               ;r5 <- Y1 * X1

        addi        ar6,r5              ;r5 + 65536
nmultnxt2:
        addi        r3,r0               ;r0 gets low word of product
        and         ir0,st,r3           ;r3 gets zero flag
        addc        r5,r4               ;r4 gets partial high word of
product
        addc        0,r2                ;r2 gets msb
        addi        r4,r1               ;r1 gets high word of product
        and         st,r3               ;r3 gets zero flag
        addc        0,r2                ;r2 gets msb
        and         st,r3               ;r3 gets zero flag
        bnzd        nmloop
        nop         *++ar0(ir1)
        nop         *++ar1(ir1)
        nop

        not         r1                  ;(AB-1) = (AB) - 1, thus begin
                                        ;residue
        addi        r1,r0               ;reduction
        and         ar4,st,r1           ;save carry
        addi        r2,r0               ;add msb
        addc        0,r1
        xor         ar4,r1              ;complement carry
        addi        r1,r0               ;diminished one result
        and         ar4,st,r1           ;save carry

        xor         ar4,r1              ;convert to binary
        addi        r1,r0               ;binary result
        and         ar4,st,r1           ;save carry
nmloop:
        sti         r1,*+ar5
        ||sti       r0,*ar5++(ir1)      ;store result

        pop         FP
        rets
```

```
*==================================================================*
* Bit reverse routine                                              *
*==================================================================*
*                                                                  *
*    This routine copy the data from the memory location pointed to *
*  by ar0 into the memory location pointed to by ar1 in bit-reversed *
*  order.                                                          *
*                                                                  *
********************************************************************
_bit_reverse:
      push      FP                  ;initialize registers for
      ldi       SP,FP               ;c function interface
      ldi       *-FP(2),ar0         ;load input address
      ldi       *-FP(3),ar1         ;load filter address

      ldi       @fftsize,ir0        ;initialize index
      lsh       -1,ir0
      ldi       @fftsize,rc         ;setup repeat loop
      subi      1,rc
      rptb      bitloop             ;begin loop
      ldi       *ar0,r0             ;read from input address
      sti       r0,*ar1++           ;write to output address & increment
                                    ;address
      ldi       *+ar0,r0
      sti       r0,*ar1++

      lsh       -1,ar0
      nop       *ar0++(ir0)b        ;calculate next bit-reversed address
bitloop:lsh     1,ar0               ;adjust bit reversed

      pop       FP
      rets
```

VITA

Yek Chong Yeo

Candidate for the Degree of

Master of Science

Thesis: An Implementation of Digital Filters Using Fermat Number Transform On TMS320C30 Digital Signal Processor

Major Field: Computer Science

Biographical:

Personal Data: Born in the Rep. of Singapore, on Mar 3, 1964, the son of Kee Hing Yeo and Swee Eng Thung.

Education: Graduated from Maris Stella High School, Rep. of Singapore, in Dec. 1980, and Hwa Chong Junior College, Rep. of Singapore, in Dec. 1982. Received B.S. in Electrical and Computer Engineering and B.S. in Mechanical and Aerospace Engineering from Oklahoma State University, Stillwater, Oklahoma in Jul. 1990. Completed the requirements for the M.S. degree with a major in Computer Science at Oklahoma State University in May. 1995.

Experience: Served in the Rep. of Singapore Armed Forces for two years; employed by Computer Information Service, Oklahoma State University as technical assistant from Jun. 1991 to May 1994.