# SOME SOFTWARE AND HARDWARE IMPLEMENTATIONS OF THE FAST HARTLEY TRANSFORM 

By<br>FU, YAN KIT<br>B.Sc.Eng., University of New Brunswick, 1986<br>\title{ A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF APPLIED SCIENCE in<br><br>THE FACULTY OF GRADUATE STUDIES<br><br>DEPARTMENT OF ELECTRICAL ENGINEERING }<br>We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1990
(c) FU Yan Kit, 1990

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical Engineering
The University of British Columbia
2356 Main Mall
Vancouver, Canada

Date:

$$
\text { Aug. } 13,1990
$$


#### Abstract

The fast Hartley transform (FHT) is a new tool for converting data between time and frequency domains.

In this thesis, some speed-optimized software implementations of the radix- 2 and split-radix FHT algorithms are presented initially, and then applied to the problems of convolution, computation of power spectra, image degradation, and image restoration.

Subsequent work involved the development of a new bit-reversal algorithm. This algorithm is fast and efficient, and can be used to increase the throughput of the FHT.

Finally, several hardware implementations are presented for the discrete Hartley transform (DHT) and the FHT with architectures using a single butterfly unit, pipelining and superparallelism. The advantages of each implementation are stressed. The data processing rates of these hardware implementations are analyzed.


## Table of Contents

Abstract ..... i
List of Tables ..... v
List of Figures ..... vii
Acknowledgement ..... viii
1 Introduction ..... 1
1.1 Digital signal processing ..... 1
1.2 Overview of the following chapters ..... 2
2 The fast Hartley transform (FHT) software implementation ..... 4
2.1 The Hartley transform ..... 4
2.2 The discrete Hartley transform (DHT) ..... 4
2.3 The FHT ..... 5
2.3.1 The radix-2 fast algorithm ..... 5
2.3.2 The split-radix fast algorithm ..... 10
2.4 Comparison with the complex FFT ..... 14
2.5 Comparison with the real FFT ..... 18
2.6 To get the FFT from the FHT ..... 19
2.7 FHT convolution and power spectrum ..... 21
2.8 An image processing example of the application of the FHT ..... 22
2.9 Summary ..... 26
3 The bit-reversal algorithms ..... 27
3.1 Some existing algorithms ..... 27
3.2 A new bit-reversal algorithm ..... 31
3.3 Comparison of the bit-reversal algorithms ..... 34
3.4 Summary ..... 35
4 The FHT hardware implementation ..... 37
4.1 Introduction ..... 37
4.2 Hardware implementation of the DHT ..... 37
4.3 Hardware implementation of the FHT ..... 40
4.3.1 Single butterfly unit FHT architecture ..... 40
4.3.2 Pipeline FHT architecture ..... 45
4.3.3 Superparallel FHT architecture ..... 49
4.4 Data unscrambling in hardware implementation ..... 51
4.5 Comparison of the hardware implementations ..... 52
4.6 Summary ..... 53
5 Conclusion ..... 54
5.1 The work done in this thesis ..... 54
5.2 Future work ..... 56
References ..... 57
Appendices ..... 61
A The radix-2 decimation-in-frequency FHT programs listing ..... 61
A. 1 Radix-2 FHT implemented with the "three-loop method" ..... 62
A. 2 Radix-2 FHT implemented with the "table-lookup method" ..... 64
B The split-radix decimation-in-frequency FHT programs listing ..... 66
B. 1 Split-radix FHT implemented with the "three-loop method" ..... 67
B. 2 Split-radix FHT implemented with the "table-lookup method" ..... 70
C The radix-2 decimation-in-frequency FFT programs listing ..... 73
C. 1 Radix-2 forward FFT implemented with the "three-loop method" ..... 74
C. 2 Radix-2 forward FFT implemented with the "table-lookup method" ..... 76
D The split-radix decimation-in-frequency FFT programs listing ..... 78
D. 1 Split-radix forward FFT implemented with the "three-loop method" ..... 79
D. 2 Split-radix forward FFT implemented with the "table-lookup method" ..... 82
E Bit-reversal programs listing ..... 85
E. 1 The implementation of Evans' bit-reversal algorithm ..... 86
E. 2 The implementation of Gold and Rader's bit-reversal algorithm ..... 88
E. 3 The implementation of Rodríguez's bit-reversal algorithm ..... 89
E. 4 The implementation of Buneman's bit-reversal algorithm ..... 90
E. 5 The implementation of the new bit-reversal algorithm ..... 91
E. 6 The implementation of the new algorithm using table-lookup method ..... 92
F Image processing example using the FHT ..... 93
F. 1 Using the FHT in the processing of image degradation and restoration ..... 94

## List of Tables

2.1 Number of real arithmetic operations for the fast algorithms. ..... 15
2.2 Comparison of the fast algorithms in CPU execution time (in seconds) on
a Sun $3 / 60$ workstation, where $\mathrm{r}-2$ stands for the radix- 2 algorithm and s-r stands for the split-radix algorithm. ..... 15
3.3 Bit-reversing of a 16 -point array ..... 28
3.4 Exclusive-oring a bit-reversed index in column three and the related operand in column four produces the next bit-reversed index in the data sequence. ..... 32
3.5 CPU execution time (in seconds) of the bit-reversal algorithms running on a Sun $3 / 60$ workstation. ..... 34
4.6 Comparison of the three FHT hardware implementations for $N=1024$ and memory access time of 60 ns . ..... 53

## List of Figures

2.1 Radix-2 decimation-in-frequency and decimation-in-time butterflies. ..... 7
2.2 Flow diagram of radix-2 decimation-in-frequency decomposition from a 16-point FHT into two 8-point FHT blocks. ..... 8
2.3 Flow diagram of an 8 -point radix-2 decimation-in-frequency FHT ..... 8
2.4 Decomposition pattern of a 64 -point radix-2 FHT. ..... 9
2.5 Flow diagram of split-radix decimation-in-frequency decomposition from a 16 -point FHT into one 8 -point FHT block and two 4 -point FHT blocks. ..... 12
2.6 Flow diagram of an 8-point split-radix decimation-in-frequency FHT. ..... 12
2.7 Decomposition pattern of a 64 -point split-radix FHT. ..... 13
2.8 FHT requires less CPU time than the complex FFT in the radix- 2 soft- ware implementation in both the "three-loop (lp)" and "table-lookup (tb)" methods. ..... 16
2.9 FHT requires less CPU time than the complex FFT in the split-radix software implementation in both the "three-loop (lp)" and "table-lookup (tb)" methods. ..... 16
2.10 Image degradation and restoration processes. ..... 23
2.11 Original image. ..... 24
2.12 Degraded image. ..... 24
2.13 Restored image. ..... 25
3.14 Gold-Rader's bit-reversal algorithm (after Rabiner and Gold). ..... 28
3.15 The table-lookup bit-reversal algorithm (after Buneman). ..... 30
3.16 Improved Gold-Rader's algorithm (after Rodríguez) ..... 30
3.17 The new bit-reversal algorithm. ..... 33
3.18 Plotting of Table 3.5. ..... 35
4.19 Hardware structure of a 4-point DHT. ..... 38
4.20 Architecture of a single butterfly unit FHT processor. ..... 41
4.21 Radix-2 FHT butterfly unit hardware structure. ..... 41
4.22 Single butterfly unit FHT hardware structure. ..... 42
4.23 Data sequence entering the radix-2 FHT butterfly unit. ..... 43
4.24 Timing diagram for the single butterfly unit processor. ..... 44
4.25 Pipeline architecture of a 16 -point FHT. ..... 46
4.26 Pipeline hardware structure of a 16 -point FHT. ..... 46
4.27 Hardware structure of the butterfly unit in the combined last two stages. ..... 47
4.28 Data sequence for each butterfly unit. ..... 48
4.29 Timing diagram of the pipeline FHT processor. ..... 48
4.30 Superparallel hardware structure of a 16 -point FHT. ..... 50
4.31 Hardware structure of butterfly unit BU0. ..... 50
4.32 Pre-permutation 16 -point radix-2 FHT data flow diagram. ..... 51
4.33 Data sequence entering the pre-permutation algorithm. ..... 52

## Acknowledgement

I would like to express my gratitude to my supervisor Dr. Malcome Wvong for his valuable suggestion and encouragement. I would like to thank Mr. Robert Ross for his technical advice in the use of the departmental computer facilities. Thanks also to Dr. M. Robert Ito for pointing out some interesting properties of the fast Fourier transform, that led to an improvement in this thesis.

## Chapter 1

## Introduction

### 1.1 Digital signal processing

Along with the progress of technology, the rapid development of the digital computer has made it a very powerful tool for computation and analysis. Digital signal processing technology takes advantage of the tremendous power of the digital computer to provide much faster and more accurate solutions to scientists and engineers than in the past. Much tedious work in the fields of engineering, biomedicine, physics, astronomy, etc. previously based on the analogue signal processing technology, now can be easily handled with digital signal processing techniques.

Digital signal processing is usually carried out for a mixture of data in both the time and frequency domains. The time domain allows processing in real-time systems, while the frequency domain facilitates filtering and spectrum analysis. During regular digital signal processing, data are usually transformed between these two domains frequently. In fact, the transformation is very time consuming, the throughput of most processes depends heavily on the efficiency of the way that the transformation is implemented. The fast Fourier transform (FFT) and the fast Hartley transform (FHT) are solutions to this bottleneck.

The FFT is a widely used tool in digital signal processing, and provides a fast numerical method to convert data from time domain to frequency domain, and vice versa. The classical Fourier transform is designed for the general case when the data sequence
is complex, that is, it has real and imaginary parts. For the case when the data sequence is real, its Fourier transform is of complex conjugate nature, this redundancy has been taken advantage of, and several FFT algorithms which make use of this redundancy have been devised [1]-[5], resulting in computer time savings of approximately fifty percent.

The FHT is an alternative to the FFT for real-valued sequences, and is well-known for the characteristics that, it has only real arithmetic, and the forms of its forward and inverse transforms are identical. The FHT has a speedup factor of two over the complex FFT [6]-[17], in the case when the complex FFT processes a real sequence as complex by adding an redundant zero-valued imaginary part. The FHT is useful in real data processing in different areas, for example, in one-dimension: spectral analysis, in twodimension: image processing, and also in multi-dimensional digital signal processsing.

### 1.2 Overview of the following chapters

This thesis presents some software and hardware implementations of the one-dimensional FHT.

Chapter 2 gives an introduction to the FHT. The radix-2 and split-radix FHT algorithms are implemented in software using several approaches. The FHT software implementaions are evaluated and compared with the complex and real FFT's. Application of the FHT is shown through an image processing example, where an image is first degraded and then restored. Some properties of the FHT are discussed.

Chapter 3 discusses the scrambled output problem inherent in the fast radix algorithms. A new bit-reversal algorithm is proposed and its efficiency is compared with several other bit-reversal algorithms.

Chapter 4 shows some hardware implementations of the DHT and the FHT. Architectures such as single butterfly unit, pipelining and superparallelism are compared. The
data processing rates are analyzed.
Chapter 5 draws conclusions on the implementation results.

## Chapter 2

## The fast Hartley transform (FHT) software implementation

### 2.1 The Hartley transform

In 1942 R.V.L. Hartley introduced a pair of real integral transforms [18] as an alternative to the Fourier transform pair. This pair of formulas have strict reciprocal characteristics for converting signals between time and frequency domains, and are known as the Hartley transform and its inverse.

If $t$ is a time variable and $f$ is a frequency variable, the Hartley transform ${ }^{1}[6]$ for a real function $h(t)$ is

$$
\begin{equation*}
H(f)=\int_{-\infty}^{\infty} h(t) \operatorname{cas}(2 \pi f t) d t \tag{2.1}
\end{equation*}
$$

and the inverse Hartley transform is

$$
\begin{equation*}
h(t)=\int_{-\infty}^{\infty} H(f) \operatorname{cas}(2 \pi f t) d f \tag{2.2}
\end{equation*}
$$

where $\operatorname{cas} \theta \triangleq \cos \theta+\sin \theta$.

### 2.2 The discrete Hartley transform (DHT)

In 1983 R.N. Bracewell introduced the discrete Hartley transform (DHT) [19]. The DHT uses discrete time and frequency variables instead of continuous variables as in the Hartley transform, and thus allows numerical analysis on digital computers.

[^0]The DHT of a real data sequence $h(t)$ of size $N$ is defined as

$$
\begin{equation*}
H(f)=\sum_{t=0}^{N-1} h(t) \operatorname{cas}(2 \pi f t / N) \tag{2.3}
\end{equation*}
$$

where $f=0, \ldots, N-1$.
The inverse DHT is defined as

$$
\begin{equation*}
h(t)=\frac{1}{N} \sum_{f=0}^{N-1} H(f) \operatorname{cas}(2 \pi f t / N) \tag{2.4}
\end{equation*}
$$

where $t=0, \ldots, N-1$.

### 2.3 The FHT

The DHT requires $N^{2}$ multiplications and $N(N-1)$ additions for an $N$-element data sequence, then the total arithmetic operations equals $2 N^{2}-N$ or of order $O\left(N^{2}\right)$. The number of arithmetic operations which grows at a rate of $N^{2}$ is highly undesirable for the digital computer, and led to the development of the FHT. The FHT is a set of techniques (algorithms) used on digital computers to improve the efficiency of the DHT.

In 1984 Bracewell introduced the decimation-in-time radix-2 FHT [7] of order $O\left(N \log _{2} N\right)$, and soon after that, many other FHT's of the same order have been presented, for example, Cooley-Tukey radix-2(decimation-in-frequency), radix-4, radix-8, split-radix, prime factor, Winograd, Walsh-Hadamard, etc. [9], [10], [20]-[24]. In fact, most of the fast algorithms applied to the discrete Fourier transform (DFT) basically can also be applied to the DHT because of their similarities.

### 2.3.1 The radix-2 fast algorithm

One of the earliest and most popularly used fast algorithm is the Cooley-Tukey radix-2 algorithm. The radix-2 algorithm has a simple flow structure that can be easily implemented in software.

The radix-2 FHT is listed in the following, with the general notations $n$ and $k$ instead of $t$ and $f$ to eliminate the specific reference to time and frequency, since the same FHT algorithm can be applied to both forward and inverse transforms, with only a scaling factor difference between them.

The radix-2 decimation-in-time FHT [7] is

$$
\begin{equation*}
H(k)=H_{2 n}(k)+H_{2 n+1}(k) \cos \left(\frac{2 \pi}{N} k\right)+H_{2 n+1}(N-k) \sin \left(\frac{2 \pi}{N} k\right) \tag{2.5}
\end{equation*}
$$

where $k=0, \ldots, N-1$, and $H_{2 n}$ is the DHT of the even-numbered data and $H_{2 n+1}$ is the DHT of the odd-numbered data.

The radix-2 decimation-in-frequency FHT [10] is

$$
\begin{align*}
H(2 k)= & \sum_{n=0}^{\frac{N}{2}-1}\left[x(n)+x\left(\frac{N}{2}+n\right)\right] \operatorname{cas}\left(\frac{2 \pi}{N} 2 k n\right)  \tag{2.6}\\
H(2 k+1)= & \sum_{n=0}^{\frac{N}{2}-1}\left\{\left[x(n)-x\left(\frac{N}{2}+n\right)\right] \cos \left(\frac{2 \pi}{N} n\right)\right. \\
& \left.+\left[x\left(\frac{N}{2}-n\right)-x(N-n)\right] \sin \left(\frac{2 \pi}{N} n\right)\right\} \operatorname{cas}\left(\frac{2 \pi}{N} 2 k n\right) \tag{2.7}
\end{align*}
$$

where $k=0, \ldots, \frac{N}{2}-1$.
The radix- 2 decimation-in-time and decimation-in-frequency approaches are distinguished according to the way they process the data sequence, either by doing the multiplication of the trigonometric coefficients firstly or lastly in the butterfly ${ }^{2}$ as shown in Figure 2.1. However, these approaches are symmetric in flow structures and require the same number of arithmetic operations [25] [26], where the number of nontrivial real additions, $A$, and the number of nontrivial real multiplicatons, $M$, required for an $N$-point radix-2 FHT [10] are:

$$
\begin{align*}
A & =(3 / 2) N \log _{2} N-(3 / 2) N+2  \tag{2.8}\\
M & =N \log _{2} N-3 N+4 \tag{2.9}
\end{align*}
$$

[^1]
decimation-in-frequency

decimation-in-time
$$
\theta=2 \pi / N
$$

Figure 2.1: Radix-2 decimation-in-frequency and decimation-in-time butterflies.

For simplicity, only the decimation-in-frequency approach is discussed in the following.
The operation of the radix-2 decimation-in-frequency FHT is first to split the input data into two FHT blocks, where one contains the even-numbered points and another contains the odd-numbered points. Similarly, these two FHT blocks are then split into four FHT blocks, and repeatedly until one-point FHT blocks are obtained.

A decimation-in-frequency radix-2 fast Hartley transform has the flow diagram as shown in Figure 2.2 and Figure 2.3. Figure 2.2 shows the first stage of a 16 -point FHT decomposed into two 8-point FHT blocks. Figure 2.3 shows the second, third and fourth stages of decomposition in one of the 8-point FHT block. The total number of stages required for the radix- 2 FHT equals $\left(\log _{2} N\right)$, such that, a 16 -point FHT requires 4 stages, a 1024 -point FHT requires 10 stages, etc. This decomposition property of the radix-2 FHT is shown clearly in Figure 2.4.

Software implementation of the FHT in a general purpose computer has an efficiency closely related to the fast algorithm and implementation technique. The CPU time of a FHT processing is mainly spent on the multiplications and additions of floating-point numbers, and also the computation of the trigonometric functions, the former is chiefly determined by the fast algorithm used, for instant, the split-radix FHT (will be discussed


Figure 2.2: Flow diagram of radix-2 decimation-in-frequency decomposition from a 16 -point FHT into two 8 -point FHT blocks.


Figure 2.3: Flow diagram of an 8-point radix-2 decimation-in-frequency FHT.


Figure 2.4: Decomposition pattern of a 64 -point radix-2 FHT.
in the next subsection) requires less floating-point multiplications and additions than the radix- 2 FHT , the latter, is determined by the implementation technique. Some efficient and interesting methods for implementing the radix-2 decimation-in-frequency FHT are presented in the following.

The first method (as shown in Appendix A) uses three loops to implement the recursive property of the radix- 2 algorithm, and is thus termed the "three-loop method".

The first loop jumps by stage, and it loops through only $\left(\log _{2} N-2\right)$ stages, because the last two stages are not involved in the multiplication of the trigonometric functions, it is more efficient to treat them separately with special coding.

The second loop jumps by data point, and the trigonometric function values are computed in this loop. Since within the same stage, the trigonometric function values required in different blocks are identical, so that, when the third loop jumps by block, the same set of trigonometric function values can be reused in each block, and thus increases
the processing speed.
Inside the third loop, a four-point butterfly is used to process four data points at a time. A separate part for sequence with size less than four is added at the beginning of the program to complement the four-point butterfly.

The second method (also shown in Appendix A) is an improved version of the first method, and the repetitive computation of the identical trigonometric functions in different stages is avoided. This is done by first creating a trigonometric function table to store all the precalulated sine and cosine function values, so that later on, when the program requires a trigonometric function value, it will look up the table instead of calculating it repeatedly in different stages. This method is termed the "table-lookup method".

One drawback of this method is that a memory space of $(N / 2)$ words is required for the trigonometric function table. However, for a process that does fixed length FHT many times, the table needs to be created only once, and the overhead to create the table will become negligible.

### 2.3.2 The split-radix fast algorithm

Among the radix algorithms, the split-radix algorithm is the most efficient one. The splitradix algorithm was first designed for the DFT, which combines the advantages of both the radix- 2 and radix- 4 fast algorithms, and has its even-numbered part performed in radix- 2 and odd-numbered part performed in radix-4. The resulting split-radix algorithm has fewer nontrivial real multiplications and additions than the radix-2, radix-4, and even radix-8 algorithms [27] [4]. The split-radix FHT [9] [20] possesses the same advantages of the split-radix FFT.

The even-numbered part of the split-radix FHT algorithm is

$$
\begin{equation*}
H(2 k)=\sum_{n=0}^{\frac{N}{2}-1}\left[x(n)+x\left(\frac{N}{2}+n\right)\right] \operatorname{cas}\left(\frac{2 \pi}{N} 2 k n\right) \tag{2.10}
\end{equation*}
$$

where $k=0, \ldots, \frac{N}{2}-1$, and the odd-numbered part is

$$
\begin{align*}
& H(4 k+1)= \\
& \quad \sum_{n=0}^{\frac{N}{4}-1}\left\{\left[\left(x(n)-x\left(\frac{N}{2}+n\right)\right)+\left(x\left(\frac{N}{4}-n\right)-x\left(\frac{3 N}{4}-n\right)\right)\right] \cos \left(\frac{2 \pi}{N} n\right)-\right. \\
& \left.\quad\left[\left(x\left(\frac{N}{4}+n\right)-x\left(\frac{3 N}{4}+n\right)\right)-\left(x\left(\frac{N}{2}-n\right)-x(N-n)\right)\right] \sin \left(\frac{2 \pi}{N} n\right)\right\} \\
& \quad \cdot \operatorname{cas}\left(\frac{2 \pi}{N} 4 k n\right)  \tag{2.11}\\
& H(4 k+3)= \\
& \quad \sum_{n=0}^{\frac{N}{4}-1}\left\{\left[\left(x(n)-x\left(\frac{N}{2}+n\right)\right)-\left(x\left(\frac{N}{4}-n\right)-x\left(\frac{3 N}{4}-n\right)\right)\right] \cos \left(\frac{2 \pi}{N} 3 n\right)+\right. \\
& \left.\quad\left[\left(x\left(\frac{N}{4}-n\right)-x\left(\frac{3 N}{4}+n\right)\right)+\left(x\left(\frac{N}{2}-n\right)-x(N-n)\right)\right] \sin \left(\frac{2 \pi}{N} 3 n\right)\right\} \\
& \quad \cdot \operatorname{cas}\left(\frac{2 \pi}{N} 4 k n\right) \tag{2.12}
\end{align*}
$$

where $k=0, \ldots, \frac{N}{4}-1$.
The split-radix FHT has a total number of nontrivial real additions, $A$, and a total number of nontrivial real multiplicatons, $M$, for an $N$-point FHT [9], such that:

$$
\begin{align*}
A & =(2 / 3) N \log _{2} N-(19 / 9) N+3+(-1)^{\log _{2} N} \cdot(1 / 9)  \tag{2.13}\\
M & =(4 / 3) N \log _{2} N-(14 / 9) N+3+(-1)^{\log _{2} N} \cdot(5 / 9) \tag{2.14}
\end{align*}
$$

Figures 2.5 and 2.6 show the flow diagrams of the split-radix FHT. The implementation of the split-radix algorithm is more difficult than the radix-2 algorithm. Since the split-radix algorithm is a combination of the radix- 2 and the radix- 4 algorithms, in a radix-2 stage, a FHT block is decomposed into two half-sized FHT blocks, but in a radix-4 stage, a FHT block is decomposed into four quarter-sized FHT blocks, that means one radix- 4 stage is equivalent to two radix- 2 stages. This is the reason why the radix- 4 is faster than the radix- 2 and why it can only accept data sequence with size equal to multiple of four.


$$
\theta=2 \pi / 16
$$

Figure 2.5: Flow diagram of split-radix decimation-in-frequency decomposition from a 16 -point FHT into one 8 -point FHT block and two 4 -point FHT blocks.


Figure 2.6: Flow diagram of an 8 -point split-radix decimation-in-frequency FHT.
stage: $|0| 1|2| 3|4| 5 \mid$


Figure 2.7: Decomposition pattern of a 64 -point split-radix FHT.
In Figure 2.7, we can see that the 64 -point FHT has its even part decomposed by the radix- 2 into a 32 -point FHT block and its odd part decomposed by the radix- 4 into two 16-point FHT blocks. This combination of different radices makes the proceeding of the even and odd part decomposition unsymmetrical. So that the "three-loop method" applied to the radix- 2 requires some modification in order to fit into the split-radix.

The first method (as shown in Appendix B) uses the similar approach as in the "threeloop method" in Appendix A. A special part is used for the last three stages which are not involved in the multiplication of the trigonomertic function, in order to optimize the processing speed.

The first loop jumps by stage for a total $\left(\log _{2} N-3\right)$ stages. The second loop jumps by point, and the trigonometric function values are calculated in this loop, such that when the third loop jumps by the FHT blocks, each block in the same stage can reuse the trigonometric function values obtained in the second loop and so reduces the CPU
time.
The third loop is more complicated than that in the radix- 2 implementation, because of the decomposition of the unsymmetrical even-numbered and odd-numbered parts. An elegant method in [28] is utilized here to achieve the task. Inside the third loop, an eightpoint butterfly is used to process the data, therefore, a pre-examination part is presented at the beginning of the program to process the FHT with sequence size less than eight.

This method avoids the trigonometric functions being calculated repeatedly in each block in the same stage. A further speed-up can be obtained by using a lookup table (as shown in Appendix B). This is similar to that of the radix-2, where a trigonometric function table is created to store all the precalculated sine and cosine values, and thus avoid the repeated calculation of the identical trigonometric functions in different stages. This "table-lookup method" requires a table size of ( $N / 2$ ) words.

### 2.4 Comparison with the complex FFT

For comparison of the FHT with the complex FFT, the radix- 2 complex FFT and the split-radix complex FFT [28] [29] based on the three-loop and table-lookup methods are implemented and as shown in Appendix C and Appendix $\mathrm{D}^{3}$, respectively.

Table 2.1 lists the number of nontrivial real multiplications and additions required for each algorithm, where for the radix- 2 complex FFT,

$$
\begin{align*}
A & =3 N \log _{2} N-2 N+2  \tag{2.15}\\
M & =2 N \log _{2} N-4 N+4 \tag{2.16}
\end{align*}
$$

and for the split-radix complex FFT [29] [30],

$$
\begin{equation*}
A=(8 / 3) N \log _{2} N-(16 / 9) N+2-(-1)^{\log _{2} N} \cdot(2 / 9) \tag{2.17}
\end{equation*}
$$

[^2]| $N$ | FHT |  |  |  | complex FFT |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | Radix-2 |  | Split-radix |  | Radix-2 |  | Split-radix |  |
|  | adds | mults | adds | mults | adds | mults | adds | mults |
| 2 | 2 | 0 | 2 | 0 | 4 | 0 | 4 | 0 |
| 4 | 8 | 0 | 8 | 0 | 18 | 4 | 16 | 0 |
| 8 | 26 | 4 | 22 | 2 | 58 | 20 | 52 | 8 |
| 16 | 74 | 20 | 64 | 12 | 162 | 68 | 144 | 32 |
| 32 | 194 | 68 | 166 | 42 | 418 | 196 | 372 | 104 |
| 64 | 482 | 196 | 416 | 124 | 1026 | 516 | 912 | 288 |
| 128 | 1154 | 516 | 998 | 330 | 2434 | 1284 | 2164 | 744 |
| 256 | 2690 | 1284 | 2336 | 828 | 5634 | 3076 | 5008 | 1824 |
| 512 | 6146 | 3076 | 5350 | 1994 | 12802 | 7172 | 11380 | 4328 |
| 1024 | 13824 | 7172 | 12064 | 4668 | 28674 | 16388 | 25488 | 10016 |
| 2048 | 30722 | 16388 | 26854 | 10698 | 63490 | 36868 | 56436 | 22760 |
| 4096 | 67586 | 36868 | 59168 | 24124 | 139266 | 81924 | 123792 | 50976 |

Table 2.1: Number of real arithmetic operations for the fast algorithms.

| Power of two | $N$ | FHT |  |  |  | complex FFT |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | three-loop |  | table-lookup |  | three-loop |  | table-lookup |  |
|  |  | r-2 | s-r | r-2 | s-r | r-2 | S-r | r-2 | s-r |
| 8 | 256 | 0.10 | 0.08 | 0.08 | 0.07 | 0.22 | 0.20 | 0.17 | 0.15 |
| 9 | 512 | 0.22 | 0.20 | 0.17 | 0.15 | 0.48 | 0.42 | 0.37 | 0.32 |
| 10 | 1024 | 0.48 | 0.42 | 0.37 | 0.33 | 1.02 | 0.88 | 0.82 | 0.68 |
| 11 | 2048 | 1.03 | 0.88 | 0.83 | 0.70 | 2.20 | 1.88 | 1.77 | 1.48 |
| 12 | 4096 | 2.17 | 1.88 | 1.78 | 1.53 | 4.65 | 3.97 | 3.82 | 3.15 |
| 13 | 8192 | 4.57 | 3.95 | 3.87 | 3.22 | 9.73 | 8.30 | 8.02 | 6.68 |
| 14 | 16384 | 9.65 | 8.30 | 8.15 | 6.82 | 20.40 | 17.45 | 17.02 | 14.18 |
| 15 | 32768 | 20.25 | 17.42 | 17.35 | 14.48 | 42.65 | 36.43 | 36.00 | 29.93 |
| 16 | 65536 | 42.43 | 36.43 | 36.90 | 30.53 | 89.23 | 75.97 | 75.85 | 63.17 |

Table 2.2: Comparison of the fast algorithms in CPU execution time (in seconds) on a Sun $3 / 60$ workstation, where r-2 stands for the radix- 2 algorithm and s-r stands for the split-radix algorithm.


Figure 2.8: FHT requires less CPU time than the complex FFT in the radix-2 software implementation in both the "three-loop (lp)" and "table-lookup (tb)" methods:


Figure 2.9: FHT requires less CPU time than the complex FFT in the split-radix software implementation in both the "three-loop (lp)" and "table-lookup (tb)" methods.

$$
\begin{equation*}
M=(4 / 3) N \log _{2} N-(32 / 9) N+4-(-1)^{\log _{2} N} \cdot(4 / 9) \tag{2.18}
\end{equation*}
$$

The easiest method to apply the complex FFT on a real sequence, is to expand the real sequence to complex by adding redundant zero-valued imaginary part. Based on this method, the execution time ${ }^{4}$ of the FHT and the complex FFT of a real data sequence is computed and compared in Table 2.2, the graphs in Figures 2.8 and 2.9 based on Table 2.2 further clarify how the execution time increases with $N$. One important point not shown in Table 2.2 is that in the table-lookup method, the FHT requires a table size of ( $N / 2$ ), where the complex FFT requires a table size of $N$, double that of the FHT. It is seen that, under the same fast algorithm and implementation method, the FHT is about twice as fast as the complex FFT.

It is obvious that, the complex FFT based on the above method wastes one-half of the CPU time, on computing the redundant zero-valued imaginary data, and is then very inefficient. There are two other commonly used methods [31] to deal with real sequence in the complex FFT, and without the use of the redundant zero-valued imaginary part.

The first method is to transform an N-point real sequence by means of an $\mathrm{N} / 2-$ point complex FFT. First of all, an N-point real sequence is divided into two $\mathrm{N} / 2$-point sequences, where one contains the even-numbered data, and another contains the oddnumbered data of the original sequence. These two $\mathrm{N} / 2$-point sequences are input as the real and imaginary parts to an N/2-point complex FFT. The transform of the N-point sequence can be obtained by combining the real and imaginary parts of the complex FFT output with some algebraic computation.

The second method allows the transform of two N -point real sequences simultaneously by using one N-point complex FFT. In this case, the first real sequence is input as the real part and the second sequence is input as the imaginary part of an N-point complex FFT.

[^3]The transforms of these two N -point sequences can be decomposed from the N -point complex FFT output with some additional computation.

These two methods have the similar speedup factor as the FHT. However, as indicated in [13] [16], these methods cannot be used to retransform their own output, since they take only real data. Some modified approaches should be used for the inverse transform to convert the complex conjugate symmetric sequence back to a real sequence. So that the FHT is more convenient here for real sequences, since the identical forward transform program can be used for the inverse transform with only a scaling, and does not require extra computation for combining or sorting out the result.

### 2.5 Comparison with the real FFT

Some specially designed FFT's for real sequence [1]-[5], utilize the redundancy and symmetry of the complex FFT's to reduce the number of multiplications by exactly one-half, and the number of additions by more than one-half of that of the complex FFT's. So that, these real FFT's should run at around the same speed as the FHT under the same fast algorithm implementation. Also, the real FFT requires the same size N storage space as the FHT, and because the real FFT and the FHT are very similar, the other overhead costs should be almost the same.

The real FFT transforms a real sequence into a complex conjugate symmetric sequence, the inverse of the real FFT transforms a complex conjugate symmetric sequence back into a real sequence, such that, two algorithms are required for the application which needs both forward and inverse transforms. The FHT is then more convenient than the real FFT, since only one algorithm is sufficient for both forward and inverse transforms.

### 2.6 To get the FFT from the FHT

The Hartley transform, $H(f)$, can be split into its even part, $H_{\text {even }}(f)$ and odd part, $H_{\text {odd }}(f)$ [6], such that

$$
\begin{equation*}
H(f)=H_{\text {even }}(f)+H_{\text {odd }}(f) \tag{2.19}
\end{equation*}
$$

The even part is defined as

$$
\begin{align*}
H_{\text {even }}(f) & =\frac{1}{2}[H(f)+H(-f)]  \tag{2.20}\\
& =\int_{-\infty}^{\infty} x(t) \cos (2 \pi f t) d t \tag{2.21}
\end{align*}
$$

and the odd part is defined as

$$
\begin{align*}
H_{o d d}(f) & =\frac{1}{2}[H(f)-H(-f)]  \tag{2.22}\\
& =\int_{-\infty}^{\infty} x(t) \sin (2 \pi f t) d t \tag{2.23}
\end{align*}
$$

The two integrals in Equation (2.21) and (2.23) are known as the Fourier cosine transform and the Fourier sine transform respectively.

It is apparent that

$$
\begin{align*}
H_{\text {even }}(f)-j H_{\text {odd }}(f) & =\int_{-\infty}^{\infty} x(t)[\cos (2 \pi f t)-j \sin (2 \pi f t)] d t  \tag{2.24}\\
& =\int_{-\infty}^{\infty} x(t) e^{-j 2 \pi f t} d t \tag{2.25}
\end{align*}
$$

is in fact the Fourier transform, $F(f)$. That is

$$
\begin{equation*}
F(f)=H_{e v e n}(f)-j H_{o d d}(f) \tag{2.26}
\end{equation*}
$$

In other words, $H_{\text {even }}(f)$ is equal to the real part, $F_{\text {real }}(f)$ of the Fourier transform and $H_{o d d}(f)$ is equal to the sign-reversed imaginary part, $F_{\text {imag }}(f)$ of the Fourier transform, then the Hartley transform can be written as

$$
\begin{equation*}
H(f)=F_{\text {real }}(f)-F_{i m a g}(f) \tag{2.27}
\end{equation*}
$$

Equations (2.26) and (2.27) show that from either the Fourier transform or the Hartley transform, one can easily obtain the other transform.

Similarly, in the discrete case, for a data sequence of $N$ elements, the FFT of a real function can be obtained from the even and odd part of the FHT. Since $H(f)$ is considered periodic with period $N$, then

$$
\begin{align*}
H_{\text {even }}(f) & =\frac{1}{2}[H(f)+H(N-f)]  \tag{2.28}\\
H_{\text {odd }}(f) & =\frac{1}{2}[H(f)-H(N-f)] \tag{2.29}
\end{align*}
$$

and from 2.26

$$
\begin{align*}
F(f) & =H_{\text {even }}(f)-j H_{\text {odd }}(f) \\
& =\frac{H(N-f)+H(f)}{2}+j \frac{H(N-f)-H(f)}{2} \tag{2.30}
\end{align*}
$$

where $f=0, \ldots, N-1$, and $H(N) \triangleq H(0)$.
However, Equation (2.30) is only appropriate for real-input data, since neither $H(N-$ $f$ ) nor $H(f)$ can deal with complex numbers, so that, a method to obtain the complex FFT by using the FHT is developed in the following.

Suppose $y(t)$ is a complex function such that $y(t)=a(t)+j b(t)$, where $a(t)$ and $b(t)$ are real functions, then the FFT of $y(t)$ is $Y_{F}(f)$. By using the linearity of the FFT, the FFT of $a(t)+j b(t)$ is $A_{F}(f)+j B_{F}(f)$, where $A_{F}(f)$ and $B_{F}(f)$ are complex. If $A_{F}(f)$ and $B_{F}(f)$ are expressed in their real and imaginary parts, then

$$
\begin{align*}
Y_{F}(f) & =A_{F}(f)+j B_{F}(f) \\
& =\left[A_{F_{r}}(f)+j A_{F_{i}}(f)\right]+j\left[B_{F_{r}}(f)+j B_{F_{i}}(f)\right] \\
& =\left[A_{F_{r}}(f)-B_{F_{i}}(f)\right]+j\left[A_{F_{i}}(f)+B_{F_{r}}(f)\right] \tag{2.31}
\end{align*}
$$

since

$$
\begin{align*}
A_{F_{r}}(f) & =\frac{1}{2}\left[A_{H}(N-f)+A_{H}(f)\right]  \tag{2.32}\\
A_{F_{\mathrm{i}}}(f) & =\frac{1}{2}\left[A_{H}(N-f)-A_{H}(f)\right] \tag{2.33}
\end{align*}
$$

where $A_{H}(f)$ is the FHT of $a(t)$, and

$$
\begin{align*}
B_{F_{\mathrm{r}}}(f) & =\frac{1}{2}\left[B_{H}(N-f)+B_{H}(f)\right]  \tag{2.34}\\
B_{F_{\mathrm{i}}}(f) & =\frac{1}{2}\left[B_{H}(N-f)-B_{H}(f)\right] \tag{2.35}
\end{align*}
$$

where $B_{H}(f)$ is the FHT of $b(t)$, Equation (2.31) becomes

$$
\begin{align*}
Y_{F}(f)= & \frac{1}{2}\left[A_{H}(N-f)+A_{H}(f)-B_{H}(N-f)+B_{H}(f)\right]+ \\
& j \frac{1}{2}\left[A_{H}(N-f)-A_{H}(f)+B_{H}(N-f)+B_{H}(f)\right] \tag{2.36}
\end{align*}
$$

With Equation (2.36), one can obtain the FFT of complex-input data through the FHT.

### 2.7 FHT convolution and power spectrum

Two of the most common applications of the FHT is for convolution and to obtain power spectrum for filtering.

The FHT convolution theorem [7] states that if $h(t)$ is the convolution of two real functions, $h_{1}(t)$ and $h_{2}(t)$, that is

$$
\begin{equation*}
h(t) \triangleq h_{1}(t) * h_{2}(t)=\sum_{t^{\prime}=0}^{N-1} h_{1}\left(t^{\prime}\right) h_{2}\left(t-t^{\prime}\right) \tag{2.37}
\end{equation*}
$$

then

$$
\begin{equation*}
H(f)=H_{1}(f) H_{2 e}(f)+H_{1}(N-f) H_{2 o}(f) \tag{2.38}
\end{equation*}
$$

where $H(f), H_{1}(f)$ and $H_{2}(f)$ are the FHT's of $h(t), h_{1}(t)$ and $h_{2}(t)$, respectively, and $H_{2}(f)=H_{2 e}(f)+H_{2 o}(f)$, the sum of its even and odd parts.

From Equations (2.28) and (2.29), Equation (2.38) becomes

$$
\begin{align*}
H(f)= & \frac{1}{2}\left[H_{1}(f) H_{2}(f)+H_{1}(f) H_{2}(N-f)\right. \\
& \left.+H_{1}(N-f) H_{2}(f)-H_{1}(N-f) H_{2}(N-f)\right] \tag{2.39}
\end{align*}
$$

However, when either $h_{1}(t)$ or $h_{2}(t)$ is symmetrical (even function), that is, either $h_{1}(t)=$ $h_{1}(N-t)$ or $h_{2}(t)=h_{2}(N-t)$, then the convolution theorem in Equation (2.39) can be simplified to a single real multiplication of the two FHT's:

$$
\begin{equation*}
H(f)=H_{1}(f) H_{2}(f) \tag{2.40}
\end{equation*}
$$

The power spectrum, $P_{s}(f)$, of a real function $h(t)$, calculated with the FHT is

$$
\begin{align*}
P_{s}(f) & =|H(f)|^{2}  \tag{2.41}\\
& =\frac{1}{2}\left\{[H(f)]^{2}+[H(N-f)]^{2}\right\} \tag{2.42}
\end{align*}
$$

where $H(f)$ is the FHT of $h(t)$.

### 2.8 An image processing example of the application of the FHT

An application of the FHT is presented in this section through an image processing example.

In this example, an image is first degraded, and then restored. The theorems for addition, convolution, power spectrum, etc. on the FHT are applied.

Since the image is a two-dimensional signal, in order to use the fast one-dimensional algorithm in Appendix B, the input image is transformed into one dimension by joining each row of the two-dimensional image matrix serially with the previous row ${ }^{5}$. This is similar to TV image processing.

[^4]

Figure 2.10: Image degradation and restoration processes.
The image processing is carried out by passing the image signal through a linearmotion blur operator and with the addition of some white noise, the degraded image signal is then restored by a Wiener filter. The processes are as shown in Figure 2.10.

If an image $x(t)$, passes through a linear-motion blur operator $m(t)$ and interfered by a white noise $n(t)$, the degraded image $g(t)$ can be expressed as:

$$
\begin{equation*}
g(t)=x(t) * m(t)+n(t) \tag{2.43}
\end{equation*}
$$

where "*" is the convolution operator.
A simple method to restore the image is to pass the degraded image through a Wiener filter. The restored image $r(t)$ can be expressed as:

$$
\begin{equation*}
r(t)=w(t) * g(t) \tag{2.44}
\end{equation*}
$$

where $w(t)$ is the Wiener filter function in time domain. The Wiener filter function in frequency domain, $W(f)$ is

$$
\begin{equation*}
W(f)=\frac{M(N-f)}{|M(f)|^{2}+N S R} \tag{2.45}
\end{equation*}
$$

where $M(f)$ is the linear-motion blur operator in frequency domain, and $N S R$ is the noise to signal ratio, which is defined as the ratio of the power spectrum of noise to the power spectrum of signal.


Figure 2.11: Original image.


Figure 2.12: Degraded image.


Figure 2.13: Restored image.

The degradation starts with a portrait image as shown in Figure 2.11. Each pixel in the image has a gray level between 0 to 255 . A pixel in gray level 0 is black, in gray level 255 is white, and in a gray level between 0 and 255 has different gray scale. The image is first input to a linear-motion blur operator, which stretches each pixel in the image to a distance of nine pixels to the right, then the blurred image is further degraded with white noise of random gray level from 0 to 10 . The degraded image is shown in Figure 2.12.

The degraded image is restored with a Wiener filter, which does the inverse of the blur operator and compensates the noise level. The restored images as shown in Figure 2.13 has a root-mean-square error of 8.4 gray levels with respect to the original image.

### 2.8 Summary

The radix-2 and split-radix FHT's are implemented in the "three-loop method" and "table-lookup method", and compared with the FFT on real sequence. It is found that although the FFT can be as fast as the FHT, the FHT is more convenient in performing the inverse transform on the complex conjugate symmetric data sequence back to a real sequence. The methods to obtain the FFT from the FHT and vice versa are discussed. The applications of the FHT in convolution and computation of power spectra are shown through an image processing example.

## Chapter 3

## The bit-reversal algorithms

In the radix FHT algorithms, the data after processing are stored back in the original place where the data were obtained from, in order to economize on data memory usage. However, this "in-place" storing method generates a transform with output data sequence in bit-reversed order. For the applications which require the transform sequence in the correct order, an extra permutation stage should be used to unscramble the data. For the decimation-in-time approach, the permutation stage is placed in front of the transform, and is called the "pre-permutation". For the decimation-in-frequency approach, the permutation stage is placed after the transform, and is called the "post-permuation". However, the permutation stages in both approaches are exactly the same and can be performed with the bit-reversal algorithm.

### 3.1 Some existing algorithms

One of the most widely used bit-reversal algorithm is introduced by Gold and Rader. An implementation of this algorithm [32] is repeated in pseudo-code form in Figure 3.14.

This algorithm uses a "bit-reversed counter" to generate bit-reversed index sequence which contains the reversed bits of the index sequence from zero to (N-2). A comparsion is made between each index and the corresponding bit-reversed index generated, and a swap of the values in these two indices is required whenever the bit-reversed index is greater than the original index in value. The swapping required for a 16 -point data sequence is shown in Table 3.3.

```
subroutine reverse( \(x[], N\) )
    reversed_index \(=0\)
    \(\mathrm{N} 2=\mathrm{N} / 2\)
    for index \(=0\) to \(\mathrm{N}-2\)
        if (index < reversed_index) then swap( \(x\) [index], \(x\) [reversed_index])
        \(\mathrm{k}=\mathrm{N} 2\)
        while ( \(k\) < reversed_index)
            reversed_index \(=\) reversed_index \(-\mathbf{k}\)
            \(\mathrm{k}=\mathrm{k} / 2\)
        end while
        reversed_index \(=\) reversed_index \(+\mathbf{k}\)
    next index
end subroutine
```

Figure 3.14: Gold-Rader's bit-reversal algorithm (after Rabiner and Gold).

| Index | in binary | reversed | swap |
| :---: | :---: | :---: | :---: |
| 0 | 0000 | 0000 |  |
| 1 | 0001 | 1000 | YES |
| 2 | 0010 | 0100 | YES |
| 3 | 0011 | 1100 | YES |
| 4 | 0100 | 0010 |  |
| 5 | 0101 | 1010 | YES |
| 6 | 0110 | 0110 |  |
| 7 | 0111 | 1110 | YES |
| 8 | 1000 | 0001 |  |
| 9 | 1001 | 1001 |  |
| 10 | 1010 | 0101 |  |
| 11 | 1011 | 1101 | YES |
| 12 | 1100 | 0011 |  |
| 13 | 1101 | 1011 |  |
| 14 | 1110 | 0111 |  |
| 15 | 1111 | 1111 |  |

Table 3.3: Bit-reversing of a 16 -point array.

The operation of the bit-reversed counter starts from bit-reversed index zero, the counter generates the next bit-reversed index by incrementing one in the most significant bit of the present bit-reversed index, and a rightward carry is carried out if necessary. This procedure is repeated in the counter until the second last bit-reversed index is obtained, where the last bit-reversed index is not required to be generated simply because no swapping is involved ${ }^{1}$. To increment by one in the most significant bit, means adding a value equivalent to ( $N / 2$ ), where carry is added rightwardly. A rightward carry can be obtained by checking for the first zero bit starting from the most significant bit. If a "one" is found then change it to zero, and check the next bit on the right hand side until a zero is found, then change the zero to one. To change a one to a zero can be done by substracting ( $N / 2$ ), or ( $N / 4$ ), or ( $N / 8$ ), etc. for each one bit that is found. Similarly, a zero can be changed to a one by adding ( $N / 2$ ), or ( $N / 4$ ), or ( $N / 8$ ), etc. Since the last index which has all one-bits need not be checked, there must be at least one zero bit in each index, so that overflow will not occur.

Buneman [8] shows an equivalent bit-reversal algorithm but with a lookup table to store the (N/2), (N/4), (N/8), etc. values to reduce the CPU time spent on the repeated integer divisions. Since in some high-level computer languages, the integer division cannot be replaced by simple right-bitwise shifting as in the assembly language, instead, the integer division is performed in time consuming long division. The Buneman's algorithm is shown in pseudo-code form in Figure 3.15.

Rodriguez [34] presents an improvement to the Gold and Rader's algorithm by setting an upper bound on the index sequence so as to eliminate the CPU time on checking the indices beyond the upper bound, where the upper bound means the last index of swapping, which has a value equal to ( $N-1-\sqrt{R * N}$ ), where $R$ is "one" if the power

[^5]```
subroutine reverse(x[], power, N)
    \(\mathrm{p}=1\);
    for index \(=0\) to power
        table[index] \(=p\)
        \(\mathrm{p}=\mathrm{p}+\mathrm{p}\)
    next index
    reversed_index \(=0\)
    for index \(=1\) to \(N-2\)
        \(\mathrm{p}=\mathrm{power}\)
        do \{
            \(\mathrm{p}=\mathrm{p}-1\)
            reversed_index \(=\) reversed_index - table[p]
        \} while (reversed_index \(>=0\) )
        reversed_index \(=\) reversed_index + table[p + 1]
        if (index > reversed_index) then swap(x[index], \(\left.x\left[r e v e r s e d \_i n d e x\right]\right)\)
    next index
end subroutine
```

Figure 3.15: The table-lookup bit-reversal algorithm (after Buneman).

```
subroutine reverse(x[ ], power, N)
    upper_bound = N - 1-2 ** floor{(power + 1) / 2}
    /* floor{s} = the greatest integer smaller or equal to s */
    reversed_index = 0
    N2 = N / 2
    for index = 1 to upper_bound
        k = N2
        while (k <= reversed_index)
            reversed_index = reversed_index - k
            k = k / 2
            end while
            reversed_index = reversed_index + k
            if (index < reversed_index) swap(x[index], x[reversed_index])
    next index
end subroutine
```

Figure 3.16: Improved Gold-Rader's algorithm (after Rodríguez).
of two is even or "two" if the power of two is odd. The Rodriguez's improved algorithm is shown in pseudo-code form in Figure 3.16.

Evans [35] introduces a more efficient but also more complex bit-reversal algorithm than all the algorithms shown above. This algorithm uses a precalculated seed table to enhance the speed in generating the bit-reversed sequence, and employs some sophisticated rules to govern the swapping. An implementation of this algorithm is shown in Appendix E.1.

Additional work by Evans in [36] claims that with the addition of an extra rule, this algorithm can run slightly faster in most computers.

### 3.2 A new bit-reversal algorithm

Gold and Rader designed the bit-reversal algorithm based on the bit-reversed counter some two decades ago. The Gold-Rader's bit-reversed counter utilizes integer division on the operations of incrementation and rightward carry, since probably at that time, no suitable high-level language could be used to handle the bitwise operations, and the assembly language programming is tedious.

By taking advantage of the bitwise operators of the C language, a new bit-reversed counter is developed. Table 3.4 shows the operation of a bit-reversed counter of $N=16$ based on the use of the bitwise "exclusive-or" operator.

By inspection, one can see that the reversed bits of the odd-numbered indices can be generated by exclusive-oring the reversed bits of the previous even-numbered indices with the operand " 1000 ", whereas the reversed bits of the even-numbered indices can be obtained by exclusive-oring the reversed-bits of the previous odd-numbered indices with the operand " 1100 ", " 1110 " and " 1111 ", respectively.

The rule for choosing the appropriate "exclusive-or" operand is that, always choose

| Index | in binary | reversed | xor operand | xor result |
| :---: | :---: | :---: | :---: | :---: |
| 0 | 0000 | 0000 | 1000 | 1000 |
| 1 | 0001 | 1000 | 1100 | 0100 |
| 2 | 0010 | 0100 | 1000 | 1100 |
| 3 | 0011 | 1100 | 1110 | 0010 |
| 4 | 0100 | 0010 | 1000 | 1010 |
| 5 | 0101 | 1010 | 1100 | 0110 |
| 6 | 0110 | 0110 | 1000 | 1110 |
| 7 | 0111 | 1110 | 1111 | 0001 |
| 8 | 1000 | 0001 | 1000 | 1100 |
| 9 | 1001 | 1001 | 1100 | 0101 |
| 10 | 1010 | 0101 | 1000 | 1101 |
| 11 | 1011 | 1101 | 1110 | 0011 |
| 12 | 1100 | 0011 | 1000 | 1011 |
| 13 | 1101 | 1011 | 1100 | 0111 |
| 14 | 1110 | 0111 | 1000 | 1111 |
| 15 | 1111 | 1111 |  |  |

Table 3.4: Exclusive-oring a bit-reversed index in column three and the related operand in column four produces the next bit-reversed index in the data sequence.
the operand with width equals $\log _{2} N$, and with one or contiguous " 1 " bits in the most significant bits and some contiguous or no " 0 " bit in the least significant bits, such that the operand has a value just greater than the value of the bit-reversed index. For instant, in the 16 -point sequence example, the operands to be chosen are in the format: " 1000 ", " 1100 ", " 1110 " and " 1111 ", where the operand to be used for " 1001 ", the reversed bits of index 9 , in order to generate the reversed bits of index 10 is " 1100 ". The operand required for " 1101 ", the reversed bits of index 11 , in order to gernerate the reversed bits of index 12 is " 1110 ".

An alternative rule governing the use of the operands is to use the operand " 1000 ", starting from index $\left(2^{0}-1\right)$ and every $\left(2^{1}\right)$-index times, the operand " $1100^{\text {" }}$ is used every $\left(2^{2}\right)$-index times starting from index $\left(2^{1}-1\right)$, the operand " 1110 " is used every $\left(2^{3}\right)$-index times starting from index ( $2^{2}-1$ ), and the operand " 1111 " is used only once starting

```
subroutine reverse(x[ ], N)
    1st_operand = N / 2
    2nd_operand = (1st_operand >> 1) & (N / 2)
    reversed_index = N / 2
    for index = 1 to N - 2
        /* swapping for the odd-numbered index */
        if (index < reversed_index) then swap(x[index], x[reversed_index])
        /* generate reversed bits for the next even-numbered index */
        k = 2nd_operand
        uhile (k <= reversed_index)
            k = (k >> 1) $ (N / 2) /* next higher operand */
        end while
        reversed_index = reversed_index - k
        index = index + 1
        /* swapping for the even-numbered index */
        if (index < reversed_index) then swap(x[index], x[reversed_index])
        /* generate reversed bits for the next odd-numbered index */
        reversed_index = reversed_index " 1st_operand
    next index
end subroutine
```

Figure 3.17: The new bit-reversal algorithm.
from index $\left(2^{3}-1\right)$. In general, the operand with width equals $\log _{2} N$ and $n$ " 1 " bits in the most significant bits, where $n$ is from 1 to $\log _{2} N$, and " 0 " in the other bits, is used firstly at index position $\left(2^{n-1}-1\right)$ and repeated every $\left(2^{n}\right)$-index times.

The operands to be used for data sequence in other length can be predicted in a similar way by using either one of the rules above.

Since the implementation of the second rule may employ more complicated looping technique and hence more CPU time, the first rule is chosen for implementation, and is presented in pseudo-code form in Figure 3.17. The technique to separately process the even-numbered indices and the odd-numbered indices has a significant speedup effect.

| N | A | B | C | D | E |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 1 k | 0.008 | 0.012 | 0.013 | 0.017 | 0.010 |
| 2 k | 0.015 | 0.025 | 0.027 | 0.032 | 0.020 |
| 4 k | 0.032 | 0.050 | 0.052 | 0.063 | 0.040 |
| 8 k | 0.063 | 0.102 | 0.103 | 0.128 | 0.082 |
| 16 k | 0.128 | 0.203 | 0.210 | 0.253 | 0.163 |
| 32 k | 0.258 | 0.407 | 0.418 | 0.507 | 0.328 |

A: Evans' algorithm.
B: Gold and Rader's algorithm.
C: Rodriguez's algorithm.
D: Buneman's algorithm.
E: The new algorithm.
Table 3.5: CPU execution time (in seconds) of the bit-reversal algorithms running on a Sun $3 / 60$ workstation.

A C-language implementation based on this pseudo code is shown in Appendix E.5. A second implementation using a lookup table to store the exclusive-or operands is shown in Appendix E.6. A test by using the UNIX-C compiler and a Sun $3 / 60$ workstation shows that the implementation in Appendix E. 5 can run faster, however, in other machines or Ccompilers, where the indexing is comparatively quick, the implemenation in Appendix E. 6 may be more efficient.

### 3.3 Comparison of the bit-reversal algorithms

Table 3.5 and Figure 3.18 show the time required for different sizes of data for the above five order $O(N)$ algorithms. The result is obtained by implementing the algorithms in the C language (as shown in Appendix E.1-5), and running on a Sun $3 / 60$ workstation equipped with a floating point processor. The new algorithm is one of the fastest, and is only second to Evans' bit-reversal algorithm. However, Evans' algorithm requires a complex implementation and the use of a seed table, so that the new algorithm is very


Figure 3.18: Plotting of Table 3.5.
attractive. The Gold and Rader's algorithm is found to be simple and efficient, it is suitable for the implementation in the high-level languages lacking the bitwise operators. The algorithms by Buneman and Rodriguez do not show the expected effect, it is because the integer division ${ }^{2}$ in the Sun workstation with the UNIX-C compiler can run as fast as a right-bitwise shifting operation, the indexing and other overheads in the improvement unintentionally slow down the algorithms.

### 3.4 Summary

A new bit-reversal algorithm is presented. This new algorithm is found to be very efficient, since it is fast, has minimum memory demands and can be implemented easily in the $C$ language. Several other bit-reversal algorithms are introduced and compared with the new algorithm. A fast bit-reversal algorithm is desirable and important in the FHT, because it accounts for a few percent of the total run time of a complete radix

[^6]transform.
The usage of the bit-reversal algorithm in the hardware implementation of the FHT will be discussed in the next chapter.

## Chapter 4

## The FHT hardware implementation

### 4.1 Introduction

The software implementation of the fast algorithms on a general purpose computer or on a general purpose computer equipped with a programmable digital signal processor (DSP) chip is generally speed-limited for data processing from low to medium sampling rate, say, for a maximum of less than a million samples per second ${ }^{1}$. For some very high speed processing environment, for example, radar signal and real-time digital image processing, custom-designed hardware implementation of the transform is highly preferred. In this chapter, the hardware implementation of the DHT is proposed, and the FHT hardware implementation based on several advanced architectures is developed and evaluated.

### 4.2 Hardware implementation of the DHT

A four-point DHT based on the expression in Equation 2.3 is implemented and is shown in Figure 4.19. This hardware structure can be easily modified for data sequence of other size by appending or reducing devices in a similar layout pattern. The operation of this hardware implementation is synchronized with clock signals, namely CLK-A, CLKB, CLK-C and CLK-D. The control clock signals of the system are generated with a ring counter, where, each of the four outputs of the ring counter sends a clock signal

[^7]

Figure 4.19: Hardware structure of a 4 -point DHT.
sequentially to trigger a group of devices. The devices to be triggered synchronously, including latches and multipliers, but excluding the adders, are grouped together by the dotted-diagonal lines in Figure 4.19.

The input line is a multi-bit data bus. Data are entered through the input line to the system serially at each clock signal. Four input latches controlled by clock signals are connected to the input line to select, store and pass the corresponding data points. In each clock signal, only one input latch can pass in one data point. This data point is retained in the input latch until the next time it is triggered and another new data point is sampled. The multipliers sample data from the input latches and trigonometric
coefficients from the read-only memory (ROM). The adders are driven by the data from the multipliers, and since the two multipliers feeding data to the adder are tiggered in two consecutive clock signals, a latch is required to retain the output of the first multiplier and let the two data points arrive at the adder simultaneously. After three additions in each column, the transform output is passed to the output line in each clock signal.

The system operation starts at CLK-A. The input latch at the upper-left corner in Figure 4.19 passes in the first data point and ROM1 is addressed and provides the cas coefficient; these two data are passed to the top multiplier in the first column. At CLK-B, the multiplier is fired and the product is passed to a latch. At CLK-C the latch is selected and the datum inside is passed to an adder; meanwhile, the adder accepts another datum from the second multiplier in the first column. Since the adder is data driven, the adder outputs the sum to the second latch in column one without waiting for another clock signal. The second latch is then selected by CLK-D and the datum stored inside is passed to the next adder. Similarly, the final sum from the first column is passed to the output line through a tri-state output latch which will be triggered by the corresponding clock signal.

Just in front of the last column of multipliers, there are some other latches connected to the input latches. These are used to hold the input data, because when one of the multipliers in the last column is triggered, the input latch connected to that multiplier is being triggered simultaneously and the datum in the input latch will be lost, so that an intermediate-stage latch is required to hold the datum.

This DHT implementation requires six clock pulses of overhead, and it can convert one data point to the desired domain in one clock cycle. It has a simple structure and control, and is flexible in expansibility. However, for an $N$-point transform, $N^{2}$ word-
sized ROM, $N(N-1)$ adders and $N^{2}$ multipliers ${ }^{2}$ are required. For a large $N$, the device count is very high, and turns out to be a high hardware implementation cost. In addition, each group of devices are triggered only in the corresponding clock signal, which means a low efficiency in the use of the devices. As an alternative, hardware implementation using the fast algorithm is then desired. Several FHT hardware implementation approaches are shown in the following section.

### 4.3 Hardware implementation of the FHT

Unlike the FFT algorithms, most of the FHT algorithms, including the radix algorithms, have comparatively irregular geometry in their data-flow structure. Figures 2.2 and 2.5 are good examples to illustrate this point.

The conventional FFT hardware implementation methods [38]-[40] utilize the regular geometry of the FFT; when these methods are applied to the FHT, more complex control and data sequence rearranging are required. This is probably the reason why the FHT hardware implementation is still not so popular today.

In this section, the radix- 2 algorithm is used as an example of implementation because of its popularity and simplicity. Higher radix algorithm implementation is possible with similar technique, but a more complicated control circuit is required.

### 4.3.1 Single butterfly unit FHT architecture

Figure 4.20 shows a simple single butterfly unit architecture of the radix-2 FHT processor. The architecture shown can be configured with simple hardware. There is only one memory unit, one control unit and one butterfly unit.

[^8]

Figure 4.20: Architecture of a single butterfly unit FHT processor.

$$
\begin{array}{lc}
\text { input } & \text { output } \\
\text { registers } & \text { registers }
\end{array}
$$



Figure 4.21: Radix-2 FHT butterfly unit hardware structure.

The memory unit stores the input data, the intermediate results, the transform output and the trigonometric coefficients. The control unit generates memory addresses, memory clock cycles, butterfly unit clock cycles and other control signals. The butterfly unit performs real additions, substractions and multiplication.

Figure 4.21 shows the hardware implementation of the 16 -point radix- 2 decimation-in-frequency FHT butterfly unit based on the butterfly flow structure in Figure 2.1. The radix- 2 FHT butterfly unit can process four real data in each computation cycle,


Figure 4.22: Single butterfly unit FHT hardware structure.
Figure 4.22 shows the hardware inplementation of the single butterfly unit architecture. The operation of this implemenation can be explained with the 16 -point radix- 2 FHT in Figure 2.2 as an example. The 16-point transform has three sets of data points in its first stage, namely, the first set: points $1,7,9$ and 15 , the second set: points 2 , 6,10 and 14 , and the third set: points $3,5,11$ and 13 , which are ready to be input to the buffer registers, a, b, c and d of the butterfly unit as in Figure 4.21. The remaining points, $0,4,8$ and 12 do not require a complete operation of the butterfly unit, that is, no multiplication is required. However, it is not economical in this architecture to provide a separate butterfly unit for the processing of this special case. A solution to this is by entering these points to the butterfly unit also as a set, and inputting a one or $\cos (0)$, and a zero or $\sin (0)$ from the ROM to the multipliers, the multiplications can virtually be eliminated. Similarly, the data points in the last two stages that do not require complete operation of the butterfly unit, can be processed with the same method to eliminate the multiplication.

Figure 4.23 shows the detail of the data point sequence entering the butterfly unit. The corresponding trigonometric coefficients used in the multiplications are listed along with the data sequence, where the data sets without trigonometric coefficients shown will be multiplied by $\cos (0)$ and $\sin (0)$.


Figure 4.23: Data sequence entering the radix-2 FHT butterfly unit.

The data sequence in each stage can be fetched with the addresses generated by a pre-defined sequence counter. Therefore, for the 16 -point FHT example, four counters are required to work sequentially for each transform.

The memory unit contains three memory arrays. Memory array 1 and 2 are input buffer memory, while memory array 3 is an output buffer memory. The transform process starts with serial data input to memory array 1. In the example, sixteen data points for the first transform will be buffered. If there is more than one transform, an input switching to memory array 2 will occur, and the next sixteen data points for the second transform will enter to memory array 2 . When memory array 2 is busy in receiving the serial input data, the data already stored in memory array 1 are read into the buffer registers of the butterfly unit. The computation output of the butterfly unit is written back to memory array 1 in-place, except that the final transform result from the last stage is written to memory array 3. The output buffer memory is required to provide enough time for a slower data transferring to the next processing stage, and to allow the serial input data of the next transform to enter memory array 1 immediately in order to keep the butterfly unit running uninterruptedly.

The butterfly unit requires $\left(\frac{N}{4} \log _{2} N\right)$ butterfly cycles for an $N$-point transform. A


R: 4 read cycles, W: 4 write cycles, $\quad$ B: butterfly cycle, $\quad$ IN: 1 input cycle.
Figure 4.24: Timing diagram for the single butterfly unit processor.

16 -point transform which has four stages then requires sixteen passes in the butterfly unit. Figure 4.24 shows the timing cycles starting from memory array 2 begins to receive the second transform data. One can see that, the use of the overlapped memory and butterfly cycle, and the use of the output memory buffer increase the processing throughput and enhance the butterfly unit to run at full efficiency.

The control unit generates read and write memory cycles, butterfly cycles, device select control signals, read memory addresses, etc. Since the read memory addresses are not in sequential order, custom-designed counters are required.

Figure 4.24 shows that, when a datum is read into the buffer register $a$ in Figure 4.21 in a read memory cycle, it will wait for the equivalence of twelve memory cycles, and at the thirteenth memory cycle, the output at buffer register $a^{\prime}$ is written back to the input buffer memory in-place. That is, two sets of data points are read into the butterfly unit before one set of output is written back to the input buffer memory. Therefore, in order
to avoid the extra hardware for generating the write addresses, eight address registers are used in the control unit to retain the read addresses for twelve memory cycles.

The processing rate of this hardware structure depends on the memory access time and the processing time of the butterfly unit. In this design, the butterfly unit processing time matches the time for four read and four write cycles to optimize the efficiency. Suppose, if memory with access time of 60 ns is used, then the butterfly unit can be designed to a maximum procesing time of 480 ns or vice versa. The procesing time of a transform with length $N$, can be expressed as $8 \tau\left(\frac{N}{4}\right) \log _{2} N$, where $\tau$ is the memory access time. Therefore, if memory of 60 ns is used, a 16 -point transform will take $7.68 \mu \mathrm{~s}$, or a processing rate of 2.08 million samples per second. If $N=1024$, the processing time for one transform will take 1.23 ms , or a processing rate of 0.83 million samples per second.

The memory sometimes is the most expensive device in a circuit. If a 16 -bit number representation is used, a 16-point transform will need only 96 bytes of buffer memory, a 1024-point transform will need only 6 Kbytes of buffer memory, so that, this architecture is very economical.

### 4.3.2 Pipeline FHT architecture

Figure 4.25 shows the FHT pipeline architecture, which is similar to the single butterfly unit architecture, but instead has one butterfly unit for each stage of the FHT.

Figure 4.26 shows the pipeline hardware structure of a 16 -point FHT with two butterfly units in the first two stages and one in the last two stages. The input buffer, output buffer and the butterfly units (BU's) in the first two stages are the same as that in the single butterfly unit structure in the last subsection, except that the outputs of the butterfly units are not stored back to the input buffer, but instead are passed to the next butterfly unit through intermediate buffer memory. Since the last two stages of


Figure 4.25: Pipeline architecture of a 16 -point FHT.


Figure 4.26: Pipeline hardware structure of a 16 -point FHT.


Figure 4.27: Hardware structure of the butterfly unit in the combined last two stages.
the radix-2 FHT do not require multiplication of trigonometric coefficients, and they can process the same four data points continuously, it is more economical to combine these two stages into one with a different butterfly unit BU2 as shown in Figure 4.27.

The operation of the pipeline FHT structure is similar to the single butterfly unit structure. Each butterfly unit accepts four data points sequentially from the input buffer or the intermediate buffer, and writes the result to the next intermediate buffer or the output buffer. The input buffer and the intermediate buffer both have two memory arrays, one array being used to receive data, and the other to pass data to the next butterfly unit. A switching of input and output between these two arrays will occur when a new transform starts.

Figure 4.28 shows the data sequence entering each butterfly unit, with the trigonometric coefficients listed along. The sequence in each stage can be generated by a read address counter. Since read and write are in subsequent steps, a write address counter is avoided by using eight address registers in each stage to retain the read addresses for eight memory cycles (eight read cycles before the first write cycle), and the addresses in these address registers will be used as the write addresses.

Figure 4.29 shows the timing diagram of the pipeline structure operation. The data

B U B U B U 2


Figure 4.28: Data sequence for each butterfly unit.



| intermediate A : | W\|W|w|w | R | R | R | R | W | W | W | W | R | R | R |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| buffer 1 B: |  | W | W | W | W | R | R | R | R | W | W | W |  | w |  |
| B U: |  |  | B | B | B | B | B | B | B | B | B | B |  | B |  |



R: 4 read cycles, $\quad$ W: 4 write cycles, $\quad$ B: 1 buterfly cycle
Figure 4.29: Timing diagram of the pipeline FHT processor.
points are input to or output from the butterfly unit sequentially as in the single butterfly unit structure, and the processing time of a transform can be expressed as $4 \tau\left(\frac{N}{4}\right)$. If memory with access time of 60 ns is used, a 16 -point transform will take $0.96 \mu \mathrm{~s}$, or a processing rate of 16.7 million samples per second, while a 1024 -point tranform will take $61.4 \mu \mathrm{~s}$, or a processing rate also of 16.7 million samples per second, since the processing rate of the pipeline structure depends only on the memory read and write access time.

In this hardware structure, some very high speed buffer registers are required for the butterfly units, to allow a fast transition of data in the registers before receiving the next four data points. For example, if 60 ns memory is used for the buffer memory, 20 ns or less access time buffer register should be used. Otherwise, a wait cycle should be inserted between each four memory cycles.

### 4.3.3 Superparallel FHT architecture

In order to obtain an extremely high speed of FHT processing, a superparallel FHT processor is required. Figure 4.30 shows such a system structure for a 16 -point FHT. It is in fact the expanded structure of the pipeline FHT in Figure 4.26, enhanced with four butterfly units for each stage. A butterfly unit BU0 as shown in Figure 4.31 is used to replace butterfly unit BU in the part without the need of multiplication.

In this superparallel structure, the intermediate buffer memory can be eliminated, since it is possible to have hard-wired connections between stages. However, the superparallel structure requires a large number of butterfly units for a large $N$, for example, a long FHT transform of 1024 points, requires 2304 butterfly units. This results in a high hardware cost, and because the interconnections between stages are irregular, the input control and inter-stage connections become very complex.

By using the superparallel structure, a transform in each memory cycle is possible to obtain. That is, for any $N$, the processing time will be 60 ns if buffer memory of 60 ns


Figure 4.30: Superparallel hardware structure of a 16 -point FHT.


Figure 4.31: Hardware structure of butterfly unit BU0.


Figure 4.32: Pre-permutation 16-point radix-2 FHT data flow diagram.
access time is used. This means, theoretically, a processing rate of 267 million samples per second can be obtained for $N=16$, or 17 billion samples per second for $N=1024$. However, this depends on the data input rate as well as the butterfly processing rate.

### 4.4 Data unscrambling in hardware implementation

The transform results obtained by using the methods shown above are scrambled in bitreversed order; reordering is then required. Since $N$ is fixed in the hardware structures, the bit-reversing can be done by copying data in the output memory buffer in a bitreversed order to the output port or to the next processing stage. The bit-reversed copying sequence can be generated by a bit-reversed counter.

Another method is to employ pre-permutation algorithm. Figure 4.32 shows the prepermutation 16 -point radix- 2 FHT algorithm. If data are entered into the butterfly unit in the reversed-order as shown in Figure 4.33, using the decimation-in-time butterfly as in


Figure 4.33: Data sequence entering the pre-permutation algorithm.
Figure 2.1 in the butterfly unit, and the hardware stages in the pipeline and superparallel structures are going in the reversed direction, the transform result will be in the correct order.

### 4.5 Comparison of the hardware implementations

The DHT hardware implementation has a simple control logic and does not require the bit-reversal processing, because the data flow is very regular. Even though it involved more arithmetic than the FHT, it may be optimum for a relatively small $N$. The DHT hardware implementation can transform a point in each clock signal, therefore, it can run as fast as the FHT pipeline implementation.

A comparison of the FHT architectures in the processing rates, the number of butterfly units and the amounts of memory space required is shown in Table 4.6.

| Architecture | Number of <br> butterfly units | Speed <br> (M samples/sec.) | Memory <br> (16-bit word) |
| :---: | :---: | :---: | :---: |
| Single <br> butterfly unit | 1 |  | RAM: 6 Kbytes <br> ROM: 1 Kbytes |
| Pipelining | 9 | 0.83 | RAM: 38 Kbytes <br> ROM: 2 Kbytes |
|  |  | 16.7 | RAM: 4 Kbytes <br> ROM: 7.2 Kbytes |

Table 4.6: Comparison of the three FHT hardware implementations for $N=1024$ and memory access time of 60 ns .

### 4.6 Summary

A hardware implementation of the DHT is presented in this chapter. It has a simple control logic and flexible expansibility. However, because the number of arithmetic devices required is proportional to $N^{2}$, it is only suitable for small $N$. Solution to this is sought by the FHT implementation. Several implementation methods of the FHT are discussed, in which, the most economical method, namely, the single butterfly unit architecture, requires the fewest devices but the method is slow, whereas, the fastest method, namely, the superparallel architecture, requires the most devices. A compromise is the pipeline architecture.

## Chapter 5

## Conclusion

### 5.1 The work done in this thesis

Some properties of the fast Hartley transform (FHT) and its application in convolution and power spectrum were reviewed. The method to obtain a complex FFT through the FHT was derived.

In the software implementation, some speed-optimized FHT programs have been developed. The radix-2 and split-radix FHT programs were optimized to reduce the number of trigonometric coefficient calculations, the technique of computing the last few stages separately further improved the program execution speed. Moreover, a table-lookup version of both the radix-2 and split-radix FHT was implemented. The table-lookup version was proved to have even higher efficiency. These programs were compared with some complex FFT programs, which were implemented in an identical way.

Tests of these programs indicated that the FHT is about twice as fast as the complex FFT assigned with redundant zero-valued imaginary part. However, when some special methods are used for the complex FFT to handle the real sequence, or when the real FFT algorithm is used, the FHT does not have the speedup effect. However, the FHT is more convenient in doing the inverse transform, because only one algorithm is required for both forward and inverse transforms. On the other hand, the real sequence on the complex FFT handled with the special methods and on the real FFT, both require a separate algorithm to convert the complex conjugate symmetric transform sequence back to a real
sequence.
The application of the FHT was shown with an image processing example, where the FHT was used in the convolution and computation of power spectrum. The example used a linear-motion blur operator and white noise to degrade a protrait image, and used a Wiener filter to restore the image. The expected result was obtained as if it was done with the FFT.

A new bit-reversal algorithm was proposed based on the C-language bitwise operators. This new algorithm and several other bit-reversal algorithms were implemented and evalulated. The new algorithm is fast and simple in implementation.

In the hardware implementation, several new hardware structures were constructed based on some advanced architectures. The hardware implementation is new to the DHT and the FHT, since, from the best of the author's knowledge, no other hardware implementation of the DHT or the FHT has previously been published.

The DHT implementation requires a number of arithmetic devices proportional to $N^{2}$, but it has a simple control circuit, and a processing speed of 16.7 million samples per second for a 60 ns clock cycle, so that, the DHT hardware implementation is attractive for small $N$.

The single butterfly unit implementation is the simplest, requires the least hardware, and has a fairly high speed, namely, a data processing speed of 0.83 million samples per second for $N=1024$ and a 60 ns buffer memory access time.

The pipelining implementation is the most efficient one for either large or small $N$, which has a very high processing speed, namely, 16.7 million samples per second for $N=1024$ and a 60 ns buffer memory access time, and only requires $\left(\log _{2} N-1\right)$ butterfly units.

The superparallelism implementation is the fastest, theoretically can perform a transform in one memory cycle. It requires a hardware count of $\frac{N}{4}\left(\log _{2} N-1\right)$ butterfly units,
which turns out to be a large number if $N$ is large, but it is acceptable when $N$ is small or when a really high processing speed is necessary.

### 5.2 Future work

A regular geometry FHT algorithm is sought to ease the software and hardware implementations. To implement the FHT in software with a regular geometry algorithm, will reduce the coding effort and allow a simplier and more efficient program. To implement the FHT in hardware with a regular geometry, will greatly simplify the control circuit, and reduce the effort in designing the address generation circuit. The pipeline architecture can then be improved to accept two or even four input data points in each memory cycle, and hence increase the processing speed to two or four times. The superparallel architecture will also benefit since regular geometry allows constant geometry and repeatable interconnection design between stages.

In the hardware implementation, as $N$ increases, the hardware size will increase correspondingly, such that, for a large $N$, the chip count will be very high. It is more economical to have a custom-designed VLSI chip in case of mass production. In fact, a VLSI implementation project is currently being scheduled at UBC. In the near future, the FHT processor will very likely become a great competitor of the FFT processors.

Two dimensional and multi-dimensional FHT's have not been implemented or evaluated in this thesis; further development in this area will surely benefit two-dimensional and multi-dimensional digital signal processing.

## References

[1] G.D. Bergland, "A fast Fourier transform algorithm for real-valued series," Comm. $A C M$, vol. 11, no. 10, pp. 703-710, October 1968.
[2] J.-B. Martens, "Discrete Fourier transform algorithms for real valued sequences," IEEE Tran. ASSP, vol. 32, no. 2, pp. 390-396, April 1984.
[3] R. Kumaresan and P.K. Gupta, "A prime factor FFT algorithm with real valued arithmetic," Proc. IEEE, vol. 73, no. 7, pp. 1241-1243, July 1985.
[4] P. Duhamel, "Implementation of 'split-radix' FFT algorithms for complex, real, and real-symmetric data," IEEE Trans. ASSP, vol. 34, no. 2, pp. 285-295, April 1986.
[5] H.V. Sorensen, S.L. Jones, M.T. Heideman, and S. Burrus, "Real-valued fast Fourier transform algorithms," IEEE Trans. ASSP, vol. 35, no. 6, pp. 849-863, June 1987.
[6] R.N. Bracewell, The Hartley Transform. NY: Oxford Universtiy Press, March 1986.
[7] R.N. Bracewell, "The fast Hartley transform," Proc. IEEE, vol. 72, no. 8, pp. 10101018, August 1984.
[8] O. Buneman, "Conversion of FFT's to fast Hartley transform," SIAM J. Sci. Stat. Comput., vol. 7, no. 2, pp. 624-638, April 1986.
[9] H.V. Sorensen, D.L. Jones, C.S. Burrus and M.T. Heideman, "On computing the discrete Hartley transform," IEEE Trans. ASSP, vol. 33, no. 4, pp. 1231-1238, October 1985.
[10] H.-J. Meckleburg and P.K. Gupta, "Fast Hartley transform algorithm," Electronics Letters, vol. 21, pp. 341-343, April 1985.
[11] J. Prado, "Comments on 'The fast Hartley transform'," Proc. IEEE, vol. 73, no. 12, pp. 1862-1863, December 1985.
[12] G.E.J. Bold, "A comparison of the time involved in computing fast Hartley and fast Fourier transforms," Proc. IEEE, vol. 73, no. 12, pp. 1863-1864, December 1985.
[13] R.N. Bracewell, O. Buneman, H. Hao, and J. Villasenor, "Fast two-dimensional Hartley transform," Proc. IEEE, vol. 74, no. 9, pp. 1282-1283, September 1986.
[14] H. Hao and R.N. Bracewell, "A three-dimensional DFT algorithm using the fast Hartley transform," Proc. IEEE, vol. 75, no. 2, pp. 264-266, February 1987.
[15] M.A. O'Neill, "Faster than fast Fourier," BYTE, pp. 293-300, April 1988.
[16] R.N. Bracewell and J. Villasenor, "Hartley vs. Fourier," BYTE, vol. 13, no. 12, pp. 24-30, November 1988.
[17] T. Le-Ngoc and M.T. Vo, "Implementation and performance of the fast Hartley transform," IEEE MICRO, pp. 20-27, October 1989.
[18] R.V.L. Hartley, "A more symmetrical Fourier analysis applied to transmission problems," Proc. IRE, vol. 30, pp. 144-150, March 1942.
[19] R.N. Bracewell, "The discrete Hartley transform," J. Opt. Soc. Am., vol. 73, no. 12, pp. 1832-1835, December 1983.
[20] S.-C. Pei and J.-L. Wu, "Split-radix fast Hartley transform," Electronics Letters, vol. 22, no. 1, pp. 26-27, January 2, 1986.
[21] R.N. Bracewell, "Alternative to split-radix Hartley transform," Electronics Letters, vol. 23, no. 21, pp. 1148-1149, October 1987.
[22] H.S. Hou, "The fast Hartley transform algorithm," IEEE Trans. Computers, vol. 36, no. 2, pp. 147-156, February 1987.
[23] C.-Y. Hsu and J.-L. Wu, "Fast computation of discrete Hartley transform via WalshHadamard transform," Electronic Letters, vol. 23, no. 9, pp. 466-468, Aprl 1987.
[24] C.-Y. Hsu and J.-L. Wu, "The Walsh-Hadamard/discrete Hartley transform," Int. J. Electronics, vol. 62, no. 5, pp. 746-755, 1987.
[25] C.P. Kwong and K.P. Shiu, "Structured fast Hartley transform algorithms," IEEE Trans. ASSP, vol. 34, no. 4, pp. 1000-1002, August 1986.
[26] H. Hao, "On fast Hartley transform algorithms," Proc. IEEE, vol. 75, no. 7, pp. 961-962, July 1987.
[27] P. Duhamel and H. Hollmann, "Split radix FFT algorithm," Electronics Letters, vol. 20, no. 1, pp. 14-16, January 5, 1984.
[28] C.S. Burrus and T.W. Parks. DFT/FFT and Convolution Algorithms. NY: Wiley, 1985.
[29] H.V. Sorensen, M.T. Heideman and C.S. Burrus, "On computing the split-radix FFT," IEEE Trans. ASSP, vol. 34, no. 1, pp. 152-156, February 1986.
[30] P. Duhamel and H. Hollmann, "Existence of a $2^{n}$ FFT algorithm with a number of multiplications lower than $2^{n+1}, "$ Electronics Letteres, vol. 20, no. 17, pp. 690-692, August 1984.
[31] E.O. Brigham, The Fast Fourier Transform. Englewood Cliff, NJ: Prentice-Hall, 1974.
[32] L.R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing. Englewood Cliffs, NJ: Prentice-Hall, 1975.
[33] R. Kumaresan and P.K. Gupta, "Vector-radix algorithm for a 2-D discrete Hartley transform," Proc. IEEE, vol. 74, no. 5, pp. 755-757, May 1986.
[34] J.J. Rodriguez, "An improved FFT digit-reversal algorithm," IEEE Trans. ASSP, vol. 37, no. 8, pp. 1298-1300, August 1989.
[35] D.M.W. Evans, "An improved digit-reversal permutation algorithm for the fast Fourier and Hartley transforms," IEEE Trans. ASSP, vol. 35, no. 8, pp. 1120-1125, August 1987.
[36] D.M.W. Evans, "A second improved digit-reversal permutation algorithm for fast transforms," IEEE Trans. ASSP, vol. 37, no. 8, pp. 1288-1291, August 1989.
[37] Texas Instruments, "New 'C30 FFT benchmark," Details on Signal Processing, Issue 20, pp. 5, fall 1989.
[38] Z.M. Ali, "A high-speed FFT processor," IEEE Trans. Communications, vol. 26, no. 5, pp. 690-696, May 1978.
[39] E.E. Swartzlander, Jr. and K.W. Young, "A radix-4 delay communtator for fast Fourier transform processor," IEEE J. Solid-State Circuits, vol. 19, no. 5, pp. 702709, October 1984.
[40] K. Yamashita, et al, "A wafer-scale 170000-gate FFT processor with built-in test circuits," IEEE J. Solid-State Circuits, vol. 23, no. 2, pp. 336-342, April 1988.

## Appendix A

The radix-2 decimation-in-frequency FHT programs listing

## A. 1 Radix-2 FHT implemented with the "three-loop method"

\#include <math.h>
\#define BUTTERFLY(a, b) \{double temp =a; a = temp + b; b=temp - b; \}


```
* Radix-2 decimation-in-frequency FHT
                                *
* (three-loop method) *
* filename: r2.1.c *
*---------.--..-.-................-...-.---------------------
* Written by Y.K. FU, Dec. }198
```



```
* Input:
* x -- data arrray with size equals power of two.
* power -- power of two.
    size -- number of input points.
* Output:
* scrambled discrete Hartley transform stored in x. *
**********************************************************/
void FHT(double x[], unsigned power, unsigned size)
{
    unsigned n, block, stage, t1, t2, t3, t4, half, qrtr, step;
    double Rad, c, S, A, x1, x2, x3, x4;
    static double TwoPi=6.28318530717958647692;
    /* check data size < 4 */
    if (ронег < 2) (
    if (power == 1)(
        BUTTERFLY(X[0], x[1]);
            return;
        }
        else return;
    }
    /* first (power - 2) stages */
    half = size;
    for (stage = 2; stage < power; stage++) C /* jump by stage */
    step = half;
    half >>= 1;
    grtr = half >> 1;
    Rad = TwoPi / step;
    for (n = 1; n < qrir; n++) { /* jump by point */
        A = Rad * n;
            c= cos(A);
            s=sin(A);
            for (block = n; block < size; block += step) { /* jump by block */
                if (n == 1) {
                    t1 = block - 1;
                    t2 = t1 + qrtr;
                    t3 = t2 + qrtr;
                    t4 = t3 + qrtr;
                    BUTTERFLY(x[t1], x[t3]);
                    BUTTERFLY(x[t2], x[t4]);
                )
                t3 = block + half;
                t2 = t3 - n - n;
                t4 = t2 + half;
                x1 = x[block] - x[t3];
                x2 = x[t2] - x[t4];
                x[block] += x[t3];
                x[t2] += x[t4];
                x[t3] = x1 * c + x2 * s;
                x[t4] = x1 * s - x2 * c;
            )
        )
    )
```

```
    /* Last two stages */
    for (block = 0; block < size; block += 4) (
    t1 = block + 1;
    t2 = block + 2;
    t3 = block + 3;
    x1 = x[block] + x[t2];
    x2 = x[t1] + x[t3];
    x3 = x[block] - x[t2];
    x4 = x[t1] - x[t3];
    x[block] = x1 + x2;
    x[t1] = x1 - x2;
    x[t2] = x3 + x4;
    x[t3] = x3 - x4;
3
)
```


## A. 2 Radix-2 FHT implemented with the "table-lookup method"

```
#include <math.h>
#include <stdio.h>
#define BUTTERFLY(a, b) (double temp = a; a = temp + b; b = temp - b;)
/*********************************************************
* Radix-2 decimation-in-frequency FHT*
```

* (table-lookup method)

```* filename: r2.2.c
* filename: r2.2.c
\[
\text { * Uritten by Y.K. Fu, Dec. } 1989 \text { * }
\]
* -
* Input: *
* x -- data arrray with size equals power of two.
* power -- power of two.
* size -- number of input points.
* Output:
* scrambled discrete Hartley transform stored in x. *
*******************************************************/
void FHT(double x[], unsigned power, unsigned size)
{
    unsigned n, block, stage, m, t, t1, t2, t3, t4, half, qrtr, step;
    double Rad, *c, *s, A, x1, x2, x3, x4;
    static double Twopi=6.28318530717958647692;
    /* check data size < 4*/
    if (power < 2) (
    if (power == 1) (
        BUTTERFLY(x[0], x[1]);
        return;
        }
        else return;
    }
    /* Create lookup tables */
    if (size > 4) <
        Rad = TwoPi / size;
        qrtr = size >> 2;
        n= sizeof(double);
        c = (double *)calloc(qrtr, n);
        s = (double *)calloc(qrtr, n);
        if (s == NULL) <
            puts(11** run out of memory in creating lookup tables **");
            exit(1);
        )
        for (m = 1; m < qrtr; m++) (
            A = Red * m;
            c[m] = cos(A);
            s[m] = sin(A);
        )
    J
    /* first (power - 2) stages */
    half = size;
    for (stage = 2; stage < power; stage++) { /* jump by stage */
        step = half;
        half >>= 1;
        qrtr = half >> 1;
        t = stage - 2;
```

```
    for (n = 1; n < qrtr; n++) (
                                    /* jump by point */
        m= n << t;
        for (block = n; block < size; block += step) { /* jump by block */
        if (n == 1) {
                    t1 = block - 1;
                    t2 = t1 + qrtr;
                    t3 = t2 + qrtr;
                    t4 = t3 + qrtr;
                    BUTTERFLY(x[ti], x[t3]);
                    BUTTERFLY(x[t2], x[t4]);
        }
        t3 = block + half;
        t2 = t3 - n- n;
        t4 = t2 + half;
        x1 = x[block] - x[t3];
        x2 = x[t2] - x[t4];
        x[block] += x[t3];
            x[t2] += x[t4];
            x[t3] = x1* c[m] + x2 * s[m];
            x[t4] = x1 * s[m] - x2 * c[m];
            3
        3
    }
    /* Last two stages */
    for (block = 0; block < size; block += 4) (
        t1 = block + 1;
    t2 = block + 2;
    t3 = block + 3;
    x1 = x[block] + x[t2];
    x2 = x[t1] + x[t3];
    x3 = x[block] - x[t2];
    x4 = x[t1] - x[t3];
    x[block] = x1 + x2;
    x[t1] = x1 - x2;
    x[t2] = x3 + x4;
    x[t3] = x3 - x4;
    3
3
```


## Appendix B

The split-radix decimation-in-frequency FHT programs listing

## B. 1 Split-radix FHT implemented with the "three-loop method"

```
#include <math.h>
#define BUTTERFLY(a, b) {double temp = a; a = temp + b; b = temp - b;}
/*******************************************************
* Split-radix decimation-in-frequency FHT
* (three-loop method)
* filename: sr.1.c
* Written by Y.K. Fu, Dec. }198
* The looping technique is adopted from the FORTRAN
* subroutine FFT written by C.S. Burrus, Dec. 1984 in *
* "DFT/FFT and Convolution Algorithms", NY: Wiley,
* pp.141-142, 1985
* Input:
* x -- data arrray with size equals power of two.
* power -- power of two.
    size -- number of input points.
* Output:
    scrambled discrete Hartley transform stored in x.
*********************************************************/
void FHT(double }\times[],\mathrm{ unsigned power, unsigned size)
C
    unsigned qrtr, octa, stage, n, block, blocksize, start,
    step, stepsize, t11, t12, t13, t14, t21, t22, t23, t24;
    double Rad, c, s, c3, s3, A, A3, x0, x1, x2, x3, x4;
    static double Twopi=6.28318530717958647692,
                        Root2=1.41421356237309514547;
    /* Check data size < 8 */
    if (power < 3) (
        if (power == 2) {
            x1 = x[0] + x[2];
            x2 = x[1] + x[3];
            x3 = x[0] - x[2];
            x4 = x[1] - x[3];
            x[0] = x1 + x2;
            x[1] = x1 - x2;
            x[2] = x3 + x4;
            x[3] = x3-x4;
            return;
            }
            if (power == 1) (
                BUTTERFLY(x[0], x[1]);
                return;
            )
            else return;
    )
    /* first (power - 3) stages */
    blocksize = size << 1;
    for (stage = 3; stage < power; stage++) { /* jump by stage */
            stepsize = blocksize;
            blocksize >>= 1;
            qrtr = blocksize >> 2;
            octa = qrtr >> 1;
            Rad = TwoPi / blocksize;
```

```
for (n = 1; n < octa; n++) < /* jump by point */
    A = Rad * n;
    A3 = A * 3;
    c = 首(A);
    s = sin(A);
    c3 = cos(A3);
    s3 = sin(A3);
    start = n;
    step = stepsize;
    do { for f* jump by block */
        for (block = start; block < size; block += step) (
            if (n == 1) <
            t11 = block - 1;
            t12 = t11 + qrtr;
            t13 = t12 + qrtr;
            t14 = t13 + qrtr;
            x3 = x[t11] - x[t13];
            x4 = x[t12] - x[t14];
            x[t11] += x[t13];
            x[t12] += x[t14];
            x[t13] = x3 + x4;
                x[t14] = x3 - x4;
                t21 = t11 + octa;
                t22 = t21 + qrtr;
                t23 = t22 + qrtr;
                t24 = t23 + qrtr;
                    x1 = x[t21];
                    x2 = x[t22];
                x[t21] = x1 + x[t23];
                x[t22] = x2 + x[t24];
                x[t23] = (x1 - x[t23])* Root2;
                        x[t24] = (x2 - x[t24]) * Root2;
            3
            t12 = block + qrtr;
            t13 = t12 + qrtr;
            t14 = t13 + qrtr;
            t21 = t12-n-n;
            t22 = t21 + qrtr;
            t23 = t22 + qrtr;
            t24 = t23 + qrtr;
            x2 = x[block] - x[t13];
            x0 = x[t21] - x[t23];
            x1 = x2 + x0;
            x2 == x0;
            x0 = x[t12] - x[t14];
            x4 = x[t22] - x[t24];
            x3 = x4 - x0;
            x4 += x0;
            x[block] += x[t13];
            x[t21] += x[t23];
            x[t12] += x[t14];
            x[t22] += x[t24];
            x[t13] = x1 * c+x3* s;
            x[t23] = x1 * s - x3* c;
            x[t14] = x2 * c3 + x4* s3;
            x[t24] = x2 * 83 - x4 * c3;
        )
        start = (step << 1) - blocksize + n;
        step <<= 2;
    ) while (start < size);
    )
)
```

```
    /* Last 3 stages */
    /* 8-point FHT blocks */
    start = 0;
    step = 16;
    do {
    for (block = start; block < size; block += step) (
        t12 = block + 2;
        t13 = t12 + 2;
        t14 = t13 + 2;
        x3 = x[block] - x[t13];
        x4 = x[t12] - x[t14];
        x[block] = x[block] + x[t13];
        x[t12] = x[t12] + x[t14];
        x[t13] = x3 + x4;
        x[t14] = x3 - x4;
        t21 = block + 1;
        t22 = t21 + 2;
        t23 = t22 + 2;
        t24 = t23 + 2;
        x1 = x[t21];
        x2 = x[t22];
        x[t21] = x { + x[t23];
        x[t22] = x2 + x[t24];
        x[t23] = (x1 - x[t23]) * Root2;
        x[t24] = (x2 - x[t24]) * Root2;
        BUTTERFLY(x[t13], x[t23]);
        BUTTERFLY(x[t14], x[t24]);
    3
    start = (step << 1) - 8;
    step <<= 2;
    } while (start < size);
    /* 4-point FHT blocks */
    start = 0;
    step = 8;
    do C
        for (block = start; block < size; block += step) {
            t12 = block + 1;
            t13 = t12 + 1;
            t14 = t13 + 1;
            x1 = x[block] + x[t13];
            x2 = x[t12] + x[t14];
            x3 = x[block] - x[t13];
            x4 = x[t12] - x[t14];
            x[block] = x1 + x2;
            x[t12] = x1 - x2;
            x[t13] = x3 + x4;
            x[t14] = x3 - x4;
        )
        start = (step << 1) - 4;
        step <<= 2;
    ) while (start < size);
3
```


## B. 2 Split-radix FHT implemented with the "table-lookup method"

```
#include <math.h>
#include <stdio.h>
#define BUTTERFLY(a,b) (double temp = a; a = temp + b; b = temp - b;)
/***********************************************************
* Split-radix decimation-in-frequency FHT
* (table-lookup method)
* filename: sr.2.c
* Uritten by Y.K. Fu, Dec. }198
* The looping technique is adopted from the FORTRAN
* The looping technique is adopted from the FORTRAN
* subroutine FFT written by C.S. Burrus, Dec. 1984 in *
* "DFT/FFT and Convolution Algorithms", NY: Wiley *
* pp.141-142; 1985
* Input:
    x .- data arrray with size equals power of two.
    power .- power of two.
    size -- number of input points.
* Output:
* scrambled discrete Hartley transform stored in X.
********************************************************/
void FHT(double x[], unsigned power, unsigned size)
C
    unsigned qrtr, octa, stage, n, m, t, block, blocksize, start
                            step, stepsize, t11, t12, t13, t14, t21, t22, t23, t24;
    double Rad, *c, *s, *c3, *s3, A, A3, x0, x1, x2, x3, x4;
    static double Twopi=6.28318530717958647692,
                        Root2=1.41421356237309514547;
    /* Check data size < 8 */
    if (power < 3) {
        if (power == 2) <
            x1 = x[0] + x[2];
            x2 = x[1] + x[3];
            x3 = x[0] - x[2];
            x4 = x[1] - x[3];
            x[0] = x1 + x2;
            x[1] = x1 - x2;
            x[2] = x3 + x4;
            x[3] = x3-x4;
            return;
        }
        if (power == 1) {
            BUTTERFLY(x[0], x[1]);
            return;
        }
        else return;
    3
    /* Create lookup tables */
    if (size > 8) {
        Rad = TwoPi / size;
        octa = size >> 3;
        n= sizeof(double);
        c = (double *)calloc(octa, n);
        s = (double *)calloc(octa, n);
        c3 = (double *)calloc(octa, n);
        s3 = (double *)calloc(octa, n);
        if (s3 == NULL) (
            puts("** run out of memory in creating lookup tables **");
            exit(1);
        }
```

```
    for (m = 1; m < octa; m++) (
        A = Rad * m;
        A3 = A * 3;
        c[m] = cos(A);
        s[m] = sin(A);
        c3[m] = cos(A3);
        s3[m] = sin(A3);
    }
)
/* first (power - 3) stages */
blocksize = size << 1;
for (stage = 3; stage < power: stage++) < /* jump by stage */
    stepsize = blocksize;
    blocksize >>= 1;
    qrtr = blocksize >> 2;
    octa = qrtr >> 1;
    t = stage - 3;
    for (n = 1; n < octa; n++) ( /* jump by point */
        start = n;
        step = stepsize;
        m=n<< t;
        do { /* jump by block*/
            for (block = start; block < size; block += step) (
                if (n==1) {
                t11 = block - 1;
                t12 = t11 + qrir;
                t13 = t12 + qrtr;
                t14 = t13 + qrer;
                x3 = x[t11] - x[t13];
                x4 = x[t12] - x[t14];
                x[t11] t= x[t13];
                x[t12] += x[t14];
                x[t13] = x3 + x4;
                x[t14] = x3 - x4;
                    t21 = t11 + octa;
                        t22 = t21 + qrtr;
                t23 = t22 + qrtr;
                    t24 = t23 + qrir;
                    x1 = x[t21];
                    x2 = x[t22];
                        x[t21] = x1 + x[t23];
                x[t22] = x2 + x[t24];
                x[t23] = (x1 - x[t23])*Root2;
                x[t24] = (x2 - x[t24])*Root2;
                )
                    t12 = block + qrtr;
                t13 = t12 + qrtr;
                t14 = t13 + qrtr;
                t21 = t12 - n - n;
                t22 = t21 + qrtr;
                t23 = t22 + qrtr;
                t24 = t23 + qrtr;
                x2 = x[block] - x[t13];
                x0 = x[t21] - x[t23];
                x1 = x2 + x0;
                x2 == x0;
                x0 = x[t{2] - x[t14];
                x4 = x[t22] - x[t24];
                x3 = x4 - x0;
                x4 += x0;
                x[block] += x[t13];
                x[t21] += x[t23];
                x[t12] += x[t14];
                x[t22] += x[t24];
                x[t13] = x1 c[m] + x3*s[m];
                    x[t23] = x4*s[m] - x3*e[m];
```

```
                    x[t14] = x2 * c3[m] + x4 * s3[m];
                    x[t24] = x2 * s3[m] - x4 * c3[m];
                }
                start = (step << 1) - blocksize + n;
                step <<= 2;
        ) while (start < size);
    J
3
    /* Last 3 stages */
    /* 8-point fHT blocks */
    start = 0;
    step = 16;
    do {
    for (block = start; block < size; block += step) (
        t12 = block + 2;
        t13 = t12 + 2;
        t14 = t13 + 2;
            x3 = x[block] - x[t13];
            x4 = x[t12] - x[t14];
            x[block] = x[block] + x[t13];
            x[t12] = x[t12] + x[t14];
            x[t13] = x3 + x4;
            x[t14] = x3 - x4;
            t21 = block + 1;
            t22 = t21 + 2;
            t23 = t22 + 2;
            t24 = t23 + 2;
            x1 = x[t21];
            x2 = x[t22];
            x[t21] = x1 + x[t23];
            x[t22] = x2 + x[t24];
            x[t23] = (x1 - x[t23]) * Root2;
            x[t24] = (x2 - x[t24]) * Root2;
            BUTTERFLY(x[t13], x[t23]):
            BUTTERFLY(x[t14], x[t24]);
    J
    start = (step << 1) - 8;
    step <<= 2;
    ) while (start < size);
    /* 4-point fHT blocks */
    start = 0;
    step = 8;
    do <
        for (block = start; block < size; block += step) (
            t12 = block + 1;
            t13 = t12 + i;
            t14 = t13 + 1;
            x1 = x[block] }+x[t13]
            x2 = x[t12] + x[t14];
            x3 = x[block] - x[t13];
            x4 = x[t12] - x[t14];
            x[block] = x1 + x2;
            x[t12] = x1 - x2;
            x[t13] = x3 + x4;
            x[t14] = x3 - x4;
        }
        start = (step << 1) - 4;
        step <<= 2;
    } while (start < size);
```

\}

## Appendix C

The radix-2 decimation-in-frequency FFT programs listing

## C. 1 Radix-2 forward FFT implemented with the "three-loop method"

```
#include <math.h>
#define BUTTERFLY(a, b) {double temp = a; a = temp + b; b = temp - b;)
/**********************************************************
* Radix-2 decimation-in-frequency FFT
* (three-loop method)
* filename: r2fft.c
```



```
* Written by Y.K. Fu, Dec. }198
* Input:
* x, real part and y, imaginary part of a data
* sequence with size equals power of two.
* power -- power of two.
* size -- number of input points.
* Output:
* x, real part and y, imaginary part of the
* scrambled discrete Hartley transform.
******************************************************/
void FFT(double x[], double y[], unsigned power, unsigned size)
<
    unsigned stage, block, n, half, step, t1, t2;
    double A, Rad, c, s, x_tmp, y_tmp;
    static double TwoPi = 6.28318530717958647692;
    half = size;
    for (stage = 1; stage < power; stage++) { /* jump by stage */
        step = half;
        half }>>=1
        Rad = TwoPi / step;
        for ( }n=1;n<half; n++) ( /* jump by point */
            A = Rad * n;
            c= cos(A);
            s = sin(A);
                for (block = n; block < size; block += step) ( /* jump by block */
                if (n == 1) c
                        t1 = block - 1;
                        t2 = t1 + half;
                        BUTTERFLY(x[t1], x[t2]);
                        BUTTERFLY(y[t1], y[t2]);
                        }
                t2 = block + half;
                        /* (x1 + jy1) - (x2 + jy2) = (x1 - x2) + j(y1 - y2) */
                x_tmp = x[block] - x[t2];
                y_tmp = y[block] - y[t2];
                        /* (x1 + jy1) + (x2 + jy2) = (x1 + x2) + j(y1 + y2) */
                        x[block] += x[t2];
                        y[block] += y[t2];
                        /* (x_tmp + j\mp@subsup{y}{~}{\primetmp) * (cosA - jsinA) , m}
                        /* = (x_tmp * cosA + y_tmp * sinA) +j(y_tmp * cosA - x_tmp * sinA) */
                        x[t2] = x_tmp * c + y_tmp * s;
                        y[t2] = y_tmp * c - x-tmp * s;
            }
        }
    }
```

```
/* last stage */
for (block = 0; block < size; block += 2) (
    t1 = block + 1;
    BUTTERFLY(x[block], x[t1]);
    BUTTERFLY(y[block], y[t1]);
    }
3
```


## C. 2 Radix-2 forward FFT implemented with the "table-lookup method"

```
#include <stdio.h>
#include <math.h>
#define BUTTERFLY(a, b) (double temp = a; a = temp + b; b = temp - b; )
f*********************************************************
* Radix-2 decimation-in-frequency FFT *
* (table-lookup method) *
* filename: r2fft.2.c *
```



```
* Written by Y.K. Fu, Dec. 1989 *
*---------------------------------------------------------****
* Input: *
* x, real part and y, imaginery part of a data *
* sequence with size equals power of two. *
* power -- power of two. *
* size .- number of input points. *
* Output:
* x, real part and y, imaginary part of the *
* scrambled discrete Hartley transform.
********************************************************/
void FFT(double X[], double y[], unsigned power, unsigned size)
{
    unsigned stage, block, n, m, half, step, t, t1, t2;
    double A, Rad, *c, *S, X_tmp, y tmp;
    static double rwopi = 6.'\overline{2}318530̄717958647692;
    /* Create lookup tables */
    Rad = Twopi / size;
    half = size >> 1;
    n= sizeof(double);
    c= (double *)calloc(half, n);
    s = (double *)calloc(half, n);
    if (s == NULL) (
        puts("** run out of memory in creating lookup tables **");
        exit(1);
    }
    for (m = 1;m < half; m++) (
    A = Rad * m;
    c[m] = cos(A);
    s[m] = sin(A);
    }
    half = size;
    for (stage = 1; stage < power; stage++) < /* jump by stage */
    step = half:
    half >>= 1;
    t = stage - 1;
    for (n = 1; n < half; n++) ( /* jump by point */
        m=n<< t;
        for (block = n; block < size; block += step) ( /* jump by block */
            if (n == 1) {
                t1 = block - 1;
                t2 = t1 + half;
                BUTTERFLY(x[t1], x[t2]);
                BUTTERFLY(y[t1], y[t2]);
            }
            t2 = block + half;
```

```
                /* (x1 + jy1) - (x2 + jy2) = (x1 - x2) + j(y1 - y2) */
                x_tmp = x[block] - x[t2];
                y_tmp = y[block] - y[t2];
                f* (x1 + jy1) + (x2 + jy2) = (x1 + x2) + j(y1 + y2) */
                x[block] += x[t2];
                y[block] += y[t2];
                /* (x_tmp + jy_tmp) * (cosA - jsinA) */
                /* = (x_tmp * cosA + y_tmp * sinA) +j(y_tmp * cosA - x_tmp * sinA) */
                x[t2] =- x_tmp * c[m] + y_tmp * s[m];
                y[t2] = y_tmp * c[m] - x_tmp * s[m];
            3
        3
    }
    /* last stage */
    for (block = 0; block < size; block += 2) (
        t1 = block + 1;
        BUTTERFLY(x[block], x[t1]);
        BUTTERFLY(y[block], y[t1]);
    }
3
```


## Appendix D

The split-radix decimation-in-frequency FFT programs listing

## D. 1 Split-radix forward FFT implemented with the "three-loop method"

```
#include <math.h>
#define BUTTERFLY(a,b) {double temp = a; a = temp + b; b = temp - b;}
/***************************t#tt***********************
* Split-radix decimation-in-frequency FFT *
* (three-loop method) *
* filename: srfft.c *
*-------------.-.................................................*
* This program is modified from the fORTRAN
* subroutine FFT written by C.S. Burrus, Dec. 1984 in*
* "DFT/FFT and Convolution Algorithms", NY: Wiley,
* pp.141-142, 1985
*--------------------------------
* Rewritten in C with extensive modification *
* by Y.K. Fu, Dec. }198
```



```
* Input:
* x, real part and y, imaginary part of a data
* sequence with size equals power of two.
* power -- power of two. *
* size -- number of input points. *
* Output: *
* x, real part and y, imaginary part of the *
* scrambled discrete Hartley transform. *
******************************************************/
void FFT(double x[], double y[], unsigned power, unsigned size)
{
    unsigned qrtr, stage, n, start, block, blocksize, step, stepsize, t0, t1, t2, t3;
    double c, c3, s, s3, A, A3, x1, x2, y1, y2, Re1, Re2, Im1, Im2, Rad;
    static double TwOPi=6.28318530717958647692;
    /* check data size < 4*/
    if (power < 2) (
        if (power == 1) (
            BUTTERFLY(x[0], x[1]);
            BUTTERFLY(Y[0], y[1]);
            return;
        }
        else return;
    }
    /* first (power - 2) stages */
    blocksize = size << 1;
    for (stage = 2; stage < power; stage++) & /*jump by stage */
        stepsize = blocksize;
        blocksize >>= 1;
        qrtr = blocksize >> 2;
        Rad = TwoPi / blocksize;
        for (n = 1; n < qrtr; n++) < /* jump by point */
            A = n * Rad;
            A3 = A* 3;
            c= cos(A);
            s = sin(A);
            c3 = cos(A3);
            s3 = sin(A3);
            start = n;
            step = stepsize;
```

```
    do {
    for (block = start; block < size; block += step) {
        if (n == 1) (
            t0 = block-1;
            t1 = to + qrar;
            t2 = t1 + qrtr;
            t3 = t2 + qrtr;
            x1 = x[t0] - x[t2];
            x2 = x[t1] - x[t3];
            x[t0] += x[t2];
            x[t1] += x[t3];
                    y1 = y[t0] - y[t2];
                    y2 = y[t1] - y[t3];
                    y[t0] += y[t2];
                    y[t1] += y[t3];
                    x[t2] = x1 + y2;
                    y[t2] = y1 - x2;
                    x[t3] = x1 - y2;
                    y[t3] = y1 + x2;
            )
            t1 = block + qrtr;
            t2 = t1 + qrtr;
            t3 = t2 + qrtr;
            x1 = x[block] - x[t2];
            x[block] += x[t2];
            x2 = x[t1] - x[t3];
            x[t1] += x[t3];
            y1 = y[block] - y[t2];
            y[block] += y[t2];
            y2=y[t1] - y[t3];
            y[t1] t= y[t3];
                        /* (x1 + jy1) - j(x2 + jy2) = (x1 + y2) + j(y1 - x2) */
            Re1 = x1 + y2;
            Im1 = y1 - x2;
                        /* (x1 + jy1) + j(x2 + jy2) = (x1 - y2) + j(y1 + x2) */
            Re2 = x1 - y2;
            Im2 = y1 + x2;
                        f* (Re1 + jIm1)* (\operatorname{cosA - jsinA) (Im * cosA-Re1 * sinA) */}
            x[t2] = Re1 * c+Im1 * s;
            y[t2] = Im1 * c-Re1 * s;
            /* (Re2 + jlm2)* (cos3A - jsin3A) */
            /* = (Re2 * cos3A + Im2 * sin3A) + j(Im2 * cos3A - Re2 * sin3A) */
            x[t3] = Re2 * c3 + Im2 * s3;
            y[t3] = Im2 * c3 - Re2 * s3;
        3
        /* new block start address */
        start = (step << 1) - blocksize + n;
        step <<= 2;
        } while (start < size);
    }
}
```

```
/* last 2 stages */
/* 4-point FFT blocks */
start = 0;
step = 8;
do {
    for (block = start; block < size; block += step) (
        t1 = block + 1;
        t2 = t1 + 1;
        t3 = t2 + 1;
        x1 = x[block] - x[t2];
        x2 = x[t1] - x[t3];
        x[block] += x[t2];
        x[t1] += x[t3];
        y1 = y[block] - y[t2];
        y2=y[t1] - y[t3];
        y[block] += y[t2];
        y[t1] += y[t3];
            x[t2] = x1 + y2;
            y[t2] = y1 - x2;
            x[t3] = x1 - y2;
            y[t3] = y1 + x2;
        3
    start = (step << 1) - 4;
    step <<= 2;
) while (start < size);
/* 2-point FFT blocks */
start = 0;
step = 4;
do {
    for (block = start; block < size; block += step) {
        t1 = block + 1;
        BUTTERFLY(x[block], x[t1]);
        BUTTERFLY(y[block], y[t1]);
    )
    start = (step << 1) - 2;
    step <<= 2;
} while (start < size);
}
```


## D. 2 Split-radix forward FFT implemented with the "table-lookup method"

```
#include <stdio.h>
#include <math.h>
#define BUTTERFLY(a, b) {double temp = a; a = temp + b; b = temp - b;}
l********************************************************
********************************************************/
void FFT(double x[], double y[], unsigned power, unsigned size)
<
unsigned qrir, stage, n, m, start, block, blocksize, step, stepsize, t, t0, t1, t2, t3;
double *c, *c3, *s, *s3, A, A3, x1, x2, y1, y2, Re1, Re2, Im1, Im2, Rad;
static double TwoPi=6.28318530717958647692;
/* check data size < 4 */
if (power < 2) {
    if (power == 1) {
            BUTTERFLY(x[0], x[1]);
            BUTTERFLY(Y[0], y[1]);
            return;
        }
        else return;
}
/* Create lookup tables */
Rad = ThoPi / size;
qrtr = size >> 2;
n= sizeof(double);
c=(double *)calloc(qrtr, n);
s = (double *)calloc(qrtr, n);
c3 = (double *)calloc(qrtr, n);
s3 = (double *)calloc(qrtr, n);
if (s3 == NULL) (
    puts("** run out of memory in creating lookup tables **");
    exit(1);
3
for (m = 1; m < qrtr; m++) {
    A = Rad * m;
    A3 = A* 3;
    c[m] = cos(A);
    s[m] = sin(A);
    c3[m] = cos(A3);
    s3[m] = sin(A3);
}
```

```
/* First (power - 2) stages */
blocksize = size << 1;
for (stage = 2; stage < power; stage++) (_/*jump by stage */
    stepsize = blocksize;
    blocksize >>= 1;
    qrtr = blocksize >> 2;
    t = stage - 2;
    for (n = 1; n < qrtr; n++) < /* jump by point */
        start = n;
        step = stepsize;
        m=n << t;
        do { /* jump by block */
                for (block = start; block < size; block += step) <
                if (n == 1) (
                    t0 = block - 1;
                    t1 = t0 + qrtr;
                    t2 = t1 + qrtr;
                t3 = t2 + qrtr;
                    x1 = x[t0] - x[t2];
                        x2 = x[t1] - x[t3];
                x[t0] += x[t2];
                x[t1] += x[t3];
                    y1 = y[t0] - y[t2];
                y2 = y[t1] - y[t3];
                y[t0] += y[t2];
                y[t1] += y[t3];
                    x[t2] = x1 + y2;
                y[t2] = y1 - x2;
                x[t3] = x1 - y2;
                y[t3] = y1 + x2;
                }
                t1 = block + qrtr;
                t2 = t1 + qrtr;
                t3 = t2 + qrtr;
                xi = x[block] - x[t2];
                x[block] += x[t2];
                x2 = x[t1] - x[t3];
                x[t1] += x[t3];
                y1 = y[block] - y[t2];
                y[block] += y[t2];
                y2 = y[t1] - y[t3];
                y[t1] += y[t3];
                /* (x1 + jy1) - j(x2 + jy2) = (x1 + y2) +j(y1 - x2) */
                Re1 = x1 + y2;
                Im1 = y1 - x2;
                /* (x1 + jy1) + j(x2 + jy2) = (x1 - y2) + j(y1 + x2) */
                Re2 = x1 - y2;
                Im2 = y1 + x2;
                /* (Re1 + jIm1) * (cosA - jsinA) */
                /* = (Re1 * cosA + Iml * sinA) + j(Im1 * cosA - Rei * sinA) */
                x[t2] = Re1 * c[m] + Im1 * s[m];
                y[t2] = Im\ * c[m] - Re1 * s[m];
```

```
                /* (Re2 + jIm2) * (cos3A - jsin3A)*/
                * = (Re2 * cos3A + Im2 * sin3A) + j(Im2 * cos3A - Re2 * sin3A) */
                x[t3] = Re2 * c3[m] + Im2 * s3[m];
                y[t3] = Im2 * c3[m] - Re2 * s3[m];
                    3
                    /* new block start address */
                start = (step << 1) - blocksize + n;
                step <<= 2;
        } while (start < size);
    }
}
/* last 2 stages */
/* 4-point FFT blocks */
start = 0;
step = 8;
do (
    for (block = start; block < size; block += step) {
        t1 = block + 1;
        t2 = t1 + 1;
        t3 = t2 + 1;
        x1 = x[block] - x[t2];
        x2 = x[t1] - x[t3];
        x[block] += x[t2];
        x[t1] += x[t3];
        y1 = y[block] - y[t2];
        y2 = y[t1] - y[t3];
        y[block] += y[t2];
        y[t1] += y[t3];
            x[t2] = x1 + y2;
            y[t2] = y1 - x2;
            x[t3] = x1 - y2;
            y[t3] = y1 + x2;
        }
        start = (step << 1) - 4;
        step <<= 2;
    ) while (start < size);
    /* 2-point FFT blocks */
    start = 0;
    step = 4;
    do <
        for (block = start; block < size; block += step) (
            t1 = block + 1;
            BUTTERFLY(x[block], x[t1]);
            BUTTERFLY(y[block], y[t1]);
        3
        start = (step << 1) - 2;
        step <<= 2;
    ) while (start < size);
}
```


## Appendix E

Bit-reversal programs listing

## E. 1 The implementation of Evans' bit-reversal algorithm

```
#include <stdio.h>
#define BASE 2
#define SWAP(a, b) {double temp = a; a = b; b = temp;)
/**********************************************************************
* filename: f_bitr.c
* Evans' bit-reversal permutation (for FHT)
* Written by Y.K. Fu, Dec. 1989 *
* Ref: D.M.W. Evans, "An improved digit-reversal permutation *
* algorithm for the fast Fourier and Hartley transforms," IEEE *
* Trans. ASSP, vol.ASSP-35, no.8, Pp.1120-1125, August 1987. *
```



```
* Input:
*
x -- bit-reversed data sequence.
power -- power of two, where 2'power=size.
size .- length of data swquence.
* Output:
* x =- reordered data sequence. *
```



```
void reverse(double x[], unsigned power, unsigned size)
{
unsigned SeedWidth, OffsetWidth, offset, GpNum, GpSize, OStemp,
                RevOffset, Revoffset0, width, *seed, TableSize;
    /*******************由####*
    * define seed table size *
    ***************************/
    /* get offset width and seed table width */
    SeedWidth = OffsetWidth = power >> 1;
    /* if power is odd, seed is one-bit wider */
    if (power & 1) (
        size >>= 1;
        SeedNidth += 1;
    }
    GpSize = size >> OffsetWidth;
    ノ**********************************************
    * create seed table according to seed width *
    **********************************************/
    /* allocate table space */
    seed = (unsigned *)calloc(1 << SeedWidth, sizeof(unsigned));
    if (seed == NULL) <
            puts("** run out of memory in creating seed table **");
            exit(1);
    }
    /* initialize the two basic elements */
    TableSize = BASE;
    seed [0] = 0; seed[1] = 1;
    /* expand seed table */
    for (width = 1; width < SeedWidth; width++) {
        /* append zeros to the existing seeds (become first half then) */
        /* and create the second-half seeds */
            for (offset = 0; offset < TableSize; offset++) {
                /* append a zero to the end of the seed */
                seed[offset] <<= 1;
                /* the second-half table equals the first-half table plus one */
                seed[offset + TableSize] = seed[offset] + 1;
            )
            /* for each extra bit in width, the table size will be double */
            Tablesize <<= 1;
    )
```

```
    /*********************************
    * switch bit-reversed elements *
    ********************************/
    for (offset = 1; offset < GpSize; offset++) <
        /* group zero */
        RevOffset0 = seed[offset] << OffsetWidth;
        SWAP(x[offset], x[RevOffset0]);
        /* other groups */
        OStemp = offset;
        for (GpNum = 1; GpNum < seed[offset]; GpNum++) { /* Theorem 3 */
            RevOffset = RevOffset0 + seed[GpNum];
            OStemp += GpSize;
            SWAP(x[OStemp], x[RevOffset]);
        3
    3
3
```


## E. 2 The implementation of Gold and Rader's bit-reversal algorithm

```
#define SWAP(a, b) {double temp = a; a = b; b = temp; }
/********************************************************************
* Gold-Rader's bit-reversal algorithm *
* fild
filename: f_r2.c
```



```
* Written by: Y.K. Fu, March 1990 *
```



```
* Ref: L.R. Rabiner and B. Gold, Theory and Application of *
* Digital Signal Processing, NJ: Prentice-Hall, pp.367, 1975. *
```



```
* Input:
* x -- bit-reversed data sequence.
* power -- power of two, not used here, just to match the
                                    argument format.
* N -- length of data sequence.
* Output:
* x -- reordered data sequence.
******************************************************************/
void reverse(double x[], unsigned power, unsigned W)
f
    int i, j, k, M, N2;
    j =`0;
    M=N-1;
    N2 = N >> 1;
    for (i = 0; i < M; i++) {
        if (i < j) SHAP(x[i], x[j]);
            /* generate bit-reversed index j */
            k = N2;
            while (k <= j) {
                j == k;
                k >>= 1;
            }
            j |= k;
    )
)
```


## E. 3 The implementation of Rodriguez's bit-reversal algorithm

```
#define SWAP(a, b) (double temp = a; a = b; b = temp;)
/***********************************************************************
* Rodriguez's improved (Gold and Rader's) bit-reversal algorithm *
* filename: f_r3.c
```



```
*Written by: Y.K. Fu, March 1990 *
```



```
* Ref: J.J. Rodriguez, "An improved FFT digit-reversal algorithm,"
* IEEE Trans. ASSP, vol.ASSP-37, no.8, pp1298-1300, August 1989.
```



```
* Input:
* x -- bit-reversed data sequence.
* power -- power of two, where 2"power=N.
* N -- length of data sequence.
* Output:
* x -- reordered data sequence.
*********************************************************************/
void reverse(double x[], unsigned power, unsigned N)
<
    unsigned i, j, k, N2, last;
    last = N - (1 << (power + 1 >> 1));
    j = 0;
    N2 = N >> 1;
    for (i = 1; i < last; i++) (
        /* generate bit-reversed index j */
        k = N2;
        while (k <= j) (
            j == k;
            k >>= 1;
        }
        j |= k;
        if (i < j) SWAP(x[i], x[j]);
    }
}
```


## E. 4 The implementation of Buneman's bit-reversal algorithm

```
#define SHAP(a, b) {double temp = a; a = b; b = temp;}
```



```
* Buneman's bit-reversal algorithm
* filename: f_r4.c
```



```
* Written by Y.K. Fu, March 1990 *
```



```
* Ref: O. Buneman, "Conversion of FFT's to fast Hartley
* transforms," SIAM J. Sci. Stat. Comput., vol.7, no.2, *
* pp.624-638, April 1986.
```



```
* Input: *
* x -- bit-reversed data sequence.
* power -- power of two, not used here, just
                to match the argument format.
* N -- length of data sequence.
* Output:
* x -- reordered data sequence.
************************************************************/
void reverse(double x[], unsigned power, unsigned N)
C
    int i, j, k, M, R[16];
    j = 1;
    for (k= 0; k <= power; k++) (
        R[k] = j;
        j += j;
    J
    M=N-1;
    j = 0;
    for (i = 1; i< M; i++) (
        /* generate bit-reversed index j */
        k = power;
        do {
            k = 1;
            j = R[k];
        } while (j >= 0);
        j += R[k+1];
        /* if check for (i < j), i is only from 1 to N - 3. */
        if (i > j) SWAP(x[i], x[j]);
    3
3
```


## E. 5 The implementation of the new bit-reversal algorithm

```
#define SWAP(a, b) {double temp = a; a = b; b = temp;}
```



```
* A new bit-reversal algorithm based on the use of the c
* language bitwise operators.
* filename: f_r5.c
```



```
* Written by: Y.K. Fu, March 1990 *
```



```
* This program generates the bit-reversed indices with a bit-
* reversal counter, and swap the elements x[index] and
* x[bit-reversed index] only when (index < bit-reversed index).
```



```
* Input:
* x -- bit-reversed data sequence.
* power -- power of two, not used here, just to match the
* argument format.
* N -- length of data sequence.
* Output:
* x -- reordered data sequence.
*****************************************************************/
void reverse(double x[], unsigned power, unsigned N)
{
    int i, j, k, M, N2, S;
    M=N - 2;
    j = N2 = N >> 1;
    s = (N2 >> 1) - N2;
    /* unscramble array x */
    for (i = 1; i < M; i++) (
        if (i < j) SHAP(x[i], x[j]);
        k = s;
        while (k <= j) k = (k >> 1) * N2;
        j = k;
        i += 1;
        if (i< < ) SUAP(x[i], x[j]);
        j `= N2;
    }
3
```


## E. 6 The implementation of the new algorithm using table-lookup method

```
#define SWAP(a, b) {double temp = a; a = b; b = temp;)
/******************************************************************
* A new bit-reversal algorithm based on the use of the C *
* language bitwise operators. *
* filename: f_r5.2.c *
```



```
* Written by: Y.K. Fu, March 1990 *
* This program generates the bit-reversed indices with a bit- *
* reversal counter, and swap the elements x[index] and *
* x[bit-reversed index] only when (index < bit-reversed index).
* Input: *
* -- bit-reversed data sequence. *
* power -- power of two. *
* -- length of data sequence. *
* Output: *
* x -- reordered data sequence.
```



```
void reverse(double x[], unsigned power, unsigned N
{
    int h, i, j, M, N2, S[16];
    M = N - 2;
    S[O] = j = N2 = N >> 1;
    /* generate table for "xor" operation */
    for (i = 1; i < power; i++) S[i] = (S[i-1] >> 1) * N2;
    /* unscramble array x */
    for (i = 1; i < M; i++) {
        if (i < j) SWAP(x[i], x[j]);
        h = 1;
        while (S[h] <= j) ++h;
        j = S Sh];
        i += 1;
        if (i< j) SWAP(x[i], x[j]);
        j `= N2;
    )
}
```


## Appendix $F$

Image processing example using the FHT

## F. 1 Using the FHT in the processing of image degradation and restoration


\#include <math.h>
/* Define size of data and work arrays */
\#define DataSize 16384 /* number of points */
\#define WorkSize (DataSize << 1 )
/* Program constants */
\#define H_OPERATOR 0.1111111111
\#define ZERO 0.0
/* White noise generator constants */
\#define LONG $2147483648.0 \quad / *$ the largest random number */ \#define LEVEL $10.0 \quad / *$ maximum noise gray level */ \#define seed 654321 /* random number seed */
/* Macro definitions */
\#define $\operatorname{SQR}(y)((y) *(y)) \quad$ /* square */ \#define $\operatorname{ABS}(y)(y<Z E R O) ?-(y):(y) \quad 1 *$ absolute value */

```
/* Function declarations */
void FHT(double x[], unsigned power, unsigned size);
void input(double x[], double f[], unsigned size);
void output(double x[], unsigned size);
void white(double n[], double MaxLvl, unsigned size);
void convolution(double a[], double b[], unsigned size);
void rmse(double x[], double f[], unsigned size);
void filter(double X[], double X2[], double h[], double n[], unsigned size);
void tables(unsigned size);
main()
{
    unsigned power, i;
    double f[DataSize], /* a copy of the original image */
                x[WorkSize], /* image(work) array */
                x2[WorkSize], /* a copy of the FHT of f */
                h[WorkSize], /* blur operator array */
                n[WorkSize]; /* white noise array */
    /*************************** Initialization ********************************/
    /* Compute power of two */
    i = WorkSize;
    for (роwer=0; i>1; power++) i >>= 1;
    /* Initialize arrays */
    for (i=DataSize; i<WOrkSize; i+4) x[i] = h[i] = n[i] = 2ERO;
    for (i=0; i<DataSize; i++) h[i] = 2ERO;
    /* Read in image and store it in a linear array, x */
    /* image gray level: 0 (black) - 255 (white) */
    input(x, f, DataSize);
    /* Create uniform linear motion blur operator, h */
```




```
h[0]=h[1] =h[2]=h[3]=h[4]=h[5]=h[6]=h[7] =h[8]=H_OPERATOR;
/* Create lookup tables for the FHT */
tables(WorkSize);
/*************************** Degradation *********************************/
/* Get F, FHT of image array f and H, FHT of the blur operator h */
FHT(x, power, NorkSize);
FHT(h, power, WorkSize);
/* Keep a copy of the transformed image for the NSR calculation */
for (i=0; i<WorkSize; i++) X2[i] = x[i];
/* Get blurred image, B = F * H*/
convolution(x, h. WorkSize);
/* Generate white noise, n */
white(n, LEVEL, DataSize);
/* Get N, FHT of n */
FHT(n, power, WorkSize):
/* Add noise to the blurred image, G = B + N*/
for (i=0; i<WorkSize; it+) x[i] t= n[i];
```

```
    /* Scale result */
    for (i=0; i<WorkSize; i++) x[i] /= WorkSize;
    /* Show blurred and noisy image g, IFHT of G*/
    FHT(x, power, WorkSize);
    for (i=DstaSize; i<WorkSize; i++) x[i] = ZERO;
    output(x, DataSize);
    ノ***################*********** Restoration *****************************/
    /* Get G, FHT of g*/
    FHT(x, power, HorkSize);
    /* Restore image by filtering, R = G * W*/
    filter(x, X2, h, n, WorkSize):
    /* Get r, IFHT of R */
    FHT(x, power, WorkSize);
    /* Show restored image */
    output(x, DataSize);
    /* Calculate root-mean-square error in restoration */
    rmse(x, f, DataSize);
}
/******************#************ Subroutines ********************************/
/**********************************|********t*****************
* Wiener filter: }W=r=\mp@code{rerse(H) / ( sqr(|H|) + NSR )
* Where,
reverse(H(i)) = H(WorkSize - i)
* sqr(|H(i)|)= Power(H(i)) *
* = (sqr(H(i)) + sqr(H(WorkSize - i)) / 2
* NSR = PowerNoise / PowerSignal *
* = Power(n) / Power(X2) *
***********************************************************/
C
    unsigned i, j, hsize;
    double temp, tp;
    /* Compute W*/
    hsize = size >> 1;
    h[0]/=(SQR(h[0]) + SQR(n[0])/ SQR(X2[0]));
    for (i=1; i<hsize; i++) {
        j=size - i;
        /* temp = sqr(|H(i)|) + NSR */
        temp = 0.5 * (SGR(h[i]) + SQR(h[j]))
            +(SQR(n[i]) + SQR(n[j])) / (SQR(X2[i]) + SQR(X2[j]));
        tp = h[i];
        h[i] = h[j] / temp;
        h[j] = tp / temp;
    )
    /* F*W*/
    convolution(x, h, size);
    /* scale the result */
    for (i=0; i<size; i++) x[i] /= size;
}
```

```
ノ**************************************************
* Root-mean-square error of the restored image *
************************************************/
void rmse(double x[], double f[], unsigned size)
{
    unsigned i;
    double sum;
    sum = 2ERO;
    for (i=0; i<size; i++) sum += SQR(x[i] - f[i]);
    printf ("root mean square error = %f\n", sqrt(sum/size));
}
/**************************
* White noise generator *
*************************/
void white(double n[], double MaxLvl, unsigned size)
<
    unsigned i;
    srand(seed);
    for (i=0; i<size; i++) n[i] = rand() / LONG * MaxLvl;
}
```



```
* FHT convolution
* If either a or b is symmetrical, the convolution *
* of a and b can be reduced to a * b
*.--------------------------------------------------------
* INPUT: arrays a and b in frequency domain,
* size of arrays.
* OUTPUT: convolution stored in array a.
****************************************************/
void convolution(double a[], double b[], unsigned size)
<
    unsigned i, j, hsize;
    double temp, t1, t2;
    hsize = size >> 1;
    a[0] *= b[0];
    for (i=1; i<hsize; i++) <
        j = size - i;
        t1 = b[i] + b[j];
        t2 = b[i] - b[j];
        temp = 0.5 * (a[i]*t1 +a[j]*t2);
        a[j]=0.5*(a[j]*t1 +a[i]* -t2);
        a[i] = temp;
    3
}
```


[^0]:    ${ }^{1}$ The original $1 / \sqrt{2 \pi}$ factors of the integrals in Hartley's definition are omitted, when frequency $f$ is used as the variable instead of angular frequency $\omega$.

[^1]:    ${ }^{2} \mathrm{~A}$ butterfly is a basic computation unit in the FHT or the FFT flow structure, such that the implementation of the fast algorithm can be simplified by using the butterfly repetitively.

[^2]:    ${ }^{3}$ Only the forward transform programs are listed. The inverse transform programs can be obtained easily from the forward transform programs by taking complex conjugate on the twiddle factors. A method to use a single forward FFT for both forward and inverse transforms is shown in [31] [32].

[^3]:    ${ }^{4}$ Does not include the input, output and bit-reversing time.

[^4]:    ${ }^{5}$ This method suffices only in this example. The general methods to obtain the two-dimensional FHT with the one-dimensional FHT algorithm are derived in [13]. A two-dimensional FHT algorithm is developed in [33].

[^5]:    ${ }^{1}$ In fact, if swapping is determined under the condition that the bit-reversed index is greater than the original index, then the second last bit-reversed index needs not be generated too, because the second last index must have a value greater than its reversed bits.

[^6]:    ${ }^{2}$ The integer division is replaced by right-bitwise shifting in the programs in Appendix E , just for a consistent comparison of the execution speed between different algorithms

[^7]:    ${ }^{1}$ As published in [37], a benchmark of 2.36 ms for a 1024 -point complex radix-2 FFT was recorded with the TMS320C30 DSP; this is equivalent to a processing rate of 0.434 million samples per second. The split-radix FHT implemented with the "table-lookup method" in Appendix B, requires 39.4 ms for a 1024 -point real data sequence on a Sun SPARC station 1 workstation.

[^8]:    ${ }^{2}$ In the actual implementation, the multipliers in the first column and the first row are not required, since a cas value equivalent to one is to be multiplied in these multipliers, data can be passed directly to the latches or adders instead.

