Number theoretic techniques applied to algorithms and architectures for digital signal processing

Jeremy S. Ward

Number Theoretic Techniques Applied to Algorithms and Architectures for Digital Signal Processing

Jeremy S. Ward B.Sc (Dunelm)

Department of Applied Physics and Electronics

University of Durham

August 1983

A Thesis submitted for the degree of

Doctor of Philosophy of the University of Durham

Abstract

Number Theoretic Techniques Applied to Algorithms and Architectures for Digital Signal Processing

J.S.Ward

Many of the techniques for the computation of a two-dimensional convolution of a small fixed window with a picture are reviewed. It is demonstrated that Winograd's cyclic convolution and Fourier Transform Algorithms, together with Nussbaumer's two-dimensional cyclic convolution algorithms, have a common general form. Many of these algorithms use the theoretical minimum number of general multiplications.

A novel implementation of these algorithms is proposed which is based upon one-bit systolic arrays. These systolic arrays are networks of identical cells with each cell sharing a common control and timing function. Each cell is only connected to it's nearest neighbours. These are all attractive features for implementation using Very Large Scale Integration (VLSI). The throughput rate is only limited by the time to perform a one-bit full addition.

In order to assess the usefulness to these systolic arrays a 'cost function' is developed to compare them with more conventional techniques, such as the Cooley-Tukey radix-2 Fast Fourier Transform (FFT). The cost function shows that these systolic arrays offer a good way of implementing the Discrete Fourier Transform for transforms up to about 30 points in length. The cost function is a general tool and allows comparisons to be made between different implementations of the same algorithm and between dissimilar algorithms.

Finally a technique is developed for the derivation of Discrete Cosine Transform (DCT) algorithms from the Winograd Fourier Transform Algorithm. These DCT algorithms may be implemented by modified versions of the systolic arrays proposed earlier, but requiring half the number of cells.

Acknowledgements

I would like to thank my supervisor at Durham, Dr B.J.Stanier, and my supervisor at the Royal Signals and Radar Establishment (RSRE), Dr J.G.McWhirter for their help and encouragement throughout the course of this project.

I should also like to thank Dr J.G.B.Roberts and Dr J.V.McCanny, both of RSRE, for many helpful discussions and constructive comments.

This work was supported by a Science and Engineering Reseach Council Co-operative Award in Science and Engineering. The sponsoring body was the Royal Signals and Radar Estabishment.

Table of Contents

Chapter 1

Aperiodic and Cyclic Convolutions

An increasingly important digital signal processing function is the calculation of two dimensional convolutions. Such convolutions are widely used, for example, in image processing and synthetic aperture radar. This thesis addresses the question, what is the 'best' method of calculating a two-dimensional convolution? In order to place some bounds upon the problem the case where one of the arrays is fixed and is small compared to the other is considered. A typical problem might be the convolution of a fixed 15x15 window with a varying picture of, say, 500x500 points.

The thesis falls into two parts. The first part, which consists of the chapters one to five, is a detailed review of many of the techniques and algorithms for calculating convolutions. The close relationship between Fourier transforms and convolutions is discussed. In particular a general form is derived which describes the most computationally efficient algorithms for both one- and two-dimensional convolutions and Discrete Fourier Transforms (DFTs). The final chapter of this first part compares all these algorithms in terms of the number of multiplications and additions performed for a given convolution.

The second part of the thesis proposes a novel implementation of algorithms having the general form developed in part one. This implementation is based upon one-bit systolic arrays and is extremely well suited for realisation using Very Large Scale Integration (VLSI) techniques. Finally a much more searching comparison is made between algorithms and different implementations of these algorithms. The comparison is made by developing a Cost Function. This Cost Function accounts for potential throughput, number of gates used, power consumption and control overheads.

The conclusion, reached at the end of the thesis, is that minimal complexity algorithms of the general cannonical form implemented using one-bit systolic arrays offer a very good solution to the problem of calculating one- and two-dimensional convolutions and DFTs.

1.1 Introduction to Chapter 1

Chapter 1 deals firstly with partitioning techniques which split long convolutions into a series of shorter ones and secondly with the relationship between DFTs and convolutions. It is shown that the convolution of two sequences is the inverse transform of the product of the foward transforms of the two sequences. Rader's theorem is then introduced. It shows that Fourier-like transforms can be re-expressed as convolutions. So there is an apparent paradox of a convolution being calculated using transforms and these transforms themselves being calculated by convolutions. The computational advantage of this nested structure will become apparent in Chapter 2.

1.2 Aperiodic and Cyclic Convolution

Many applications calling for the use of convolution demand the aperiodic convolution of two sequences. However most 'fast' algorithms apply only to periodic functions and yield cyclic or circular convolutions. The term 'fast' means using a smaller number of arithmetic operations. This section defines aperiodic and cyclic convolutions and shows how to calculate aperiodic convolutions from cyclic ones.

In the one dimensional case the aperiodic convolution y_l of a sequence x_n of N terms and the sequence h_m of L terms is defined as

$$y_n = \sum_{k=0}^{L-1} h_k x_{n-k} \qquad n = 0,1,\ldots,N+L-2$$

$$x_{n-k} = 0 \text{ if } n-k<0 \qquad (1.1)$$

This is generalised to the two-dimensional case with the arrays being $N_1 x N_2$ and $L_1 x L_2$, giving

$$y_{n_1,n_2} = \sum_{k_1=0}^{L_1-1} \sum_{k_2=0}^{L_2-1} h_{k_1,k_2} x_{n_1-k_1,n_2-k_2} \qquad (1.2)$$

where $n_1 = 0,1,\ldots,N_1+L_1-2$ and $n_2 = 0,1,\ldots,N_2+L_2-2$ with $x_{n_1-k_1,n_2-k_2}=0$ if $k_1>n_1$ or $k_2>n_2$.

In many instances only one-dimensional functions are discussed in this chapter but the results, for the most part, are easily generalisable to two-dimensions.

The aperiodic convolution of two sequences of lengths N and L (1.1) results in a sequence of N+L-1 values.

In contrast cyclic convolution convolves two length N sequences gving a length N sequence as the result. Cyclic convolution is defined by

$$y_n = \sum_{k=0}^{N-1} h_k x_{<n-k>} \qquad n=0,1,\ldots,N-1 \qquad (1.3)$$

In (1.3) the notation <.> denotes reduction modulo N. Thus to calculate the aperiodic convolution of two sequences of lengths L and N using a cyclic convolution algorithm a N+L-1 point cyclic convolution is needed. The two input sequences are extended to length N+L-1 by appending N-1 zeroes to the h sequence and L-1 zeroes to the x sequence.

To take advantage of the various fast cyclic convolution algorithms to speed-up the calculation of extremely long convolutions we need a technique to split a long convolution into a series of cyclic convolutions. There are two methods, Overlap-Add and Overlap-Save

[1.1]. The methods are similar and yield results of comparable complexity.

1.2.1 Overlap-Save

Suppose we have a very long sequence x_m, M values long with which we wish to convolve a sequence h_n, N values in length. In the Overlap-Save method the resultant convolution y_l is sectioned into blocks, each block containing K values. That is

$$y_l = y_{l_1+l_2K} \qquad l_1 = 0,1,\ldots,K-1 \text{ and } l_2 = 0,1,\ldots. \qquad (1.4)$$

$$y_{l_1+l_2K} = \sum_{n=0}^{N-1} h_n x_{l_1+l_2K-n} \qquad (1.5)$$

Consider the first y_l in any given block, i.e. $l_1=0$ and l_2 fixed, then the x_m and h_n terms which contribute to this y_l are

$$y_{l_2K} = \sum_{n=0}^{N-1} h_n x_{l_2K-n} \qquad l_1=0 \quad l_2 \text{ fixed} \qquad (1.6)$$

so the first x_m term which contributes to the result is x_{l_2K-N+1}. Now turn to the last value of y_l in the block,

$$y_{l_2K+K-1} = \sum_{n=0}^{N-1} h_n x_{l_2K+K-1-n} \qquad l_2 \text{ fixed} \qquad (1.7)$$

the last x_m term contributing to (1.7) is x_{l_2K+K-1}. So the total number of different x_m terms contained in the block is $-N+1 \rightarrow K-1$ i.e. N+K-1 terms in total. Thus to calculate each of the aperiodic convolutions in (1.5) N+K-1 point cyclic convolutions are needed. K-1 zeroes are appended to the h_m sequence. The output of each of these cyclic convolutions is given by

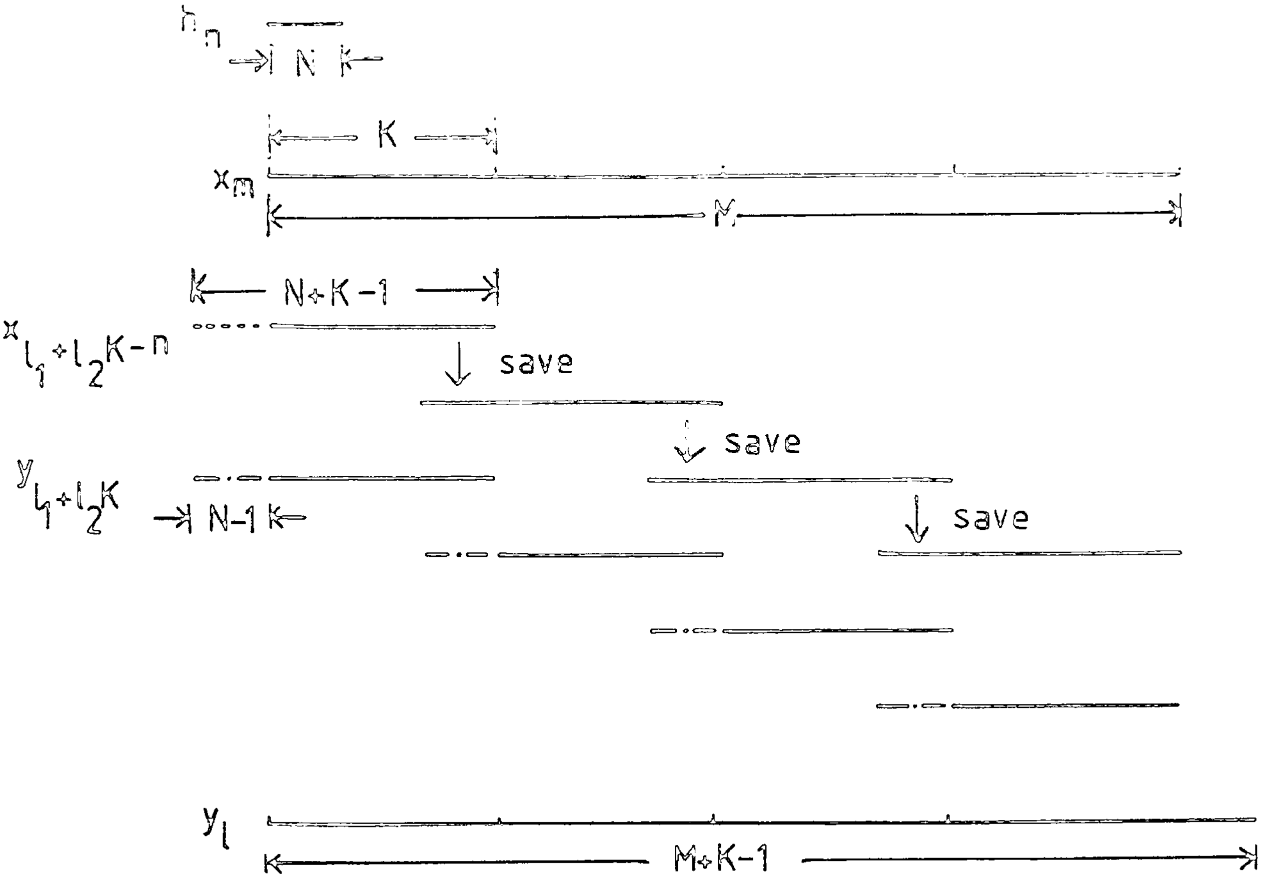


Fig 1.1 Overlap-Save

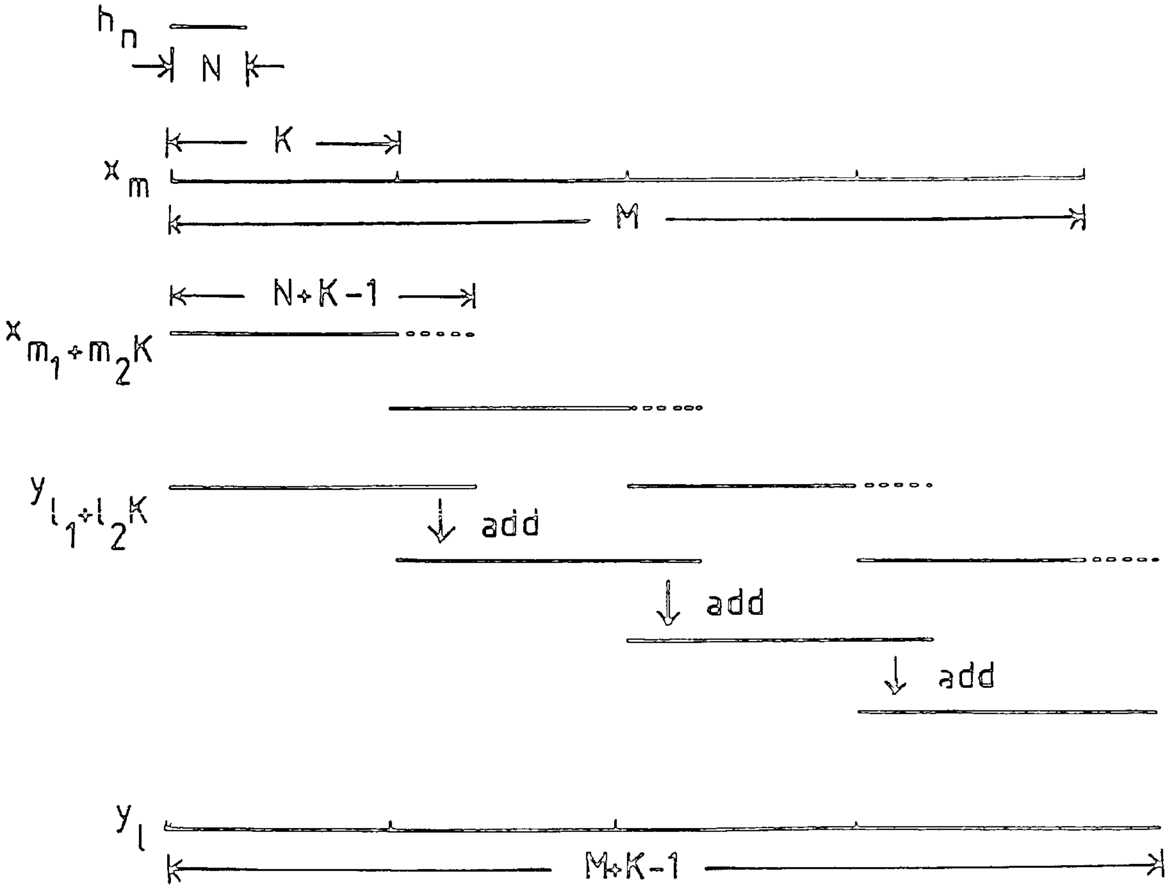


Fig 1.2 Overlap-Add

$$y_{l_1+l_2K} = \sum_{n=0}^{N+K-2} h_n x_{<l_1-n>+l_2K} \qquad \begin{array}{l} l_1 = 0,1,\ldots,K-1 \\ l_2 = 0,1,\ldots\ldots \end{array} \qquad (1.8)$$

where $<l_1-n>$ is evaluated modulo N+K-1. Only the last K output samples from each of the cyclic convolutions are used. The other values are discarded. As the last value of x_n used in a block is x_{l_2K+K-1} and the first value of x_m used in the next block is $x_{(l_2+1)K-N+1}$ the blocks into which the x_m series is divided into overlap by (N-1) terms. The Overlap-Save algorithm derives it's name from these N-1 terms of x_m 'saved' between successive cyclic convolutions. Figure 1.1 illustrates the Overlap-Save principle.

1.2.2 Overlap-Add

In the Overlap-Add technique the input sequence x_m is sectioned in contiguous blocks of length K such that

$$x_m = x_{m_1+m_2K} \qquad \begin{array}{l} m_1 = 0,1,\ldots,K-1 \\ m_2 = 0,1,\ldots \text{ for successive blocks} \end{array} \quad (1.9)$$

The aperiodic convolution of each of these blocks $x_{m_1+m_2K}$ with the sequence h_n is then computed and yields an output sequence $y_{m_2,l}$ of K+N-1 samples. The successive aperiodic convolutions $y_{m_2,l}$ are computed using K+N-1 cyclic convolutions. The input sequences are extended by appending K-1 zeroes to them.

The Overlap-Add method derives it's name from the fact that the output of each section overlaps it's neighbours by K-1 samples. These samples must be added to find the desired y_l. Thus the Overlap-Add method requires an additional K-1 additions when compared with the Overlap-Save method. Consequently the Overlap-Save method is often perfered. The Overlap-Add method is illustrated in figure 1.2.

1.3 The Cyclic Convolution Property

The Discrete Fourier Transform is one of the most important mathematical aids to signal processing. One of the more important properties of the DFT is that the inverse DFT of the product of the DFTs of two sequences is the cyclic convolution of those two sequences. This property is known as the Cyclic Convolution Property. The existence of fast DFT algorithms enables convolutions to be calculated more efficiently than by direct computation.

The Discrete Fourier Transform X_k of a sequence x_m, of N terms is defined by

$$X_k = \sum_{m=0}^{N-1} x_m W_N^{mk} \qquad k=0,1,\ldots,N-1 \qquad (1.10)$$

$$W_N = e^{-j2\pi/N}, \quad j=\sqrt{-1}$$

x_m and X_k are uniquely related by a transform pair with the foward transform given by (1.10) and the inverse transform given by

$$y_l = \frac{1}{N}\sum_{k=0}^{N-1} X_k W_N^{-lk} \qquad l=0,1,\ldots,N-1 \qquad (1.11)$$

It can be verified that (1.11) is the inverse of (1.10) by substituting for X_k in (1.11). This gives

$$y_l = \sum_{m=0}^{N-1} x_m \frac{1}{N} \sum_{k=0}^{N-1} W_N^{(m-l)k} \qquad (1.12)$$

However $W_N^N = 1$ so m-l is defined modulo N. For m-l=0 modulo N

$$\sum_{k=0}^{N-1} W_N^{(m-l)k} = N \qquad (1.13)$$

If $m-l \not\equiv 0$ mod N, then

$$\sum_{k=0}^{N-1} W_N^{(m-l)k} = \frac{W_N^{(m-l)N}-1}{W_N^{m-l}-1} \qquad (1.14)$$

Then, since $W_N^{m-l} \neq 1$ the sum (1.14) is zero. So the only non-zero case corresponds to $\underline{l=m}$, which gives $y_l = x_l$.

The DFT can be used to compute a cyclic convolution y_l of N terms with

$$y_l = \sum_{n=0}^{N-1} h_n x_{l-n} \qquad l=0,1,\ldots,N-1 \tag{1.3}$$

This is done by computing the DFTs H_k and X_k of h_n and x_m, by multiplying H_k by X_k and by computing the inverse transform C_l of H_kX_k.

$$C_l = \frac{1}{N} \sum_{k=0}^{N-1} \left\{ \sum_{m=0}^{N-1} x_m W_N^{mk} \right\} \left\{ \sum_{n=0}^{N-1} h_n W_N^{nk} \right\} W_N^{-lk} \tag{1.15}$$

rearranging gives

$$C_l = \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} h_n x_m \frac{1}{N} \sum_{k=0}^{N-1} W_N^{(m+n-l)k} \tag{1.16}$$

By using a similar argument to the inverse DFT case

$$S = \sum_{k=0}^{N-1} W_N^{(m+n-l)k}$$

becomes $\underline{S=0}$ for $m+n-l \neq 0$ Mod N

and $\underline{S=N}$ for $m+n-l=0$ Mod N

so $m=l-n$ Mod N and substituting into (1.16) gives $C_l = y_l$. The calculation of convolutions using DFTs is of little value if the DFTs are evaluated using N^2 complex multiplications. However the existence of many fast DFT algorithms makes the approach given in (1.15) an extremely useful technique.

Transforms having the property that $C_l = y_l$ in (1.15) are said to have the Cyclic Convolution Property (CCP). The DFT is not the only transform having the CCP. The above proof of the CCP relied on the existence of the value N^{-1} and the existence of the Nth root of unity in the field under consideration. This is the complex field in the case of the DFT. Provided that N^{-1} and the Nth root of unity exist other fields may be used to calculate cyclic convolutions. Pollard [1.2] was the first to suggest the use of finite fields, in particular the Galois field of p^n elements, where p is prime and n a positive integer. Such transforms are known as Number Theoretic transforms (NTTs). No further discussion of NTTs is given in this thesis. More recently Nussbaumer [1.3], and Nussbaumer and Quandalle [1.4] have introduced a transform based upon a ring of polynomials. These polynomial transforms are an important adjunct to the development to two-dimensional convolution algorithms and are considered in more detail in chapter 3.

An alternate way [1.11] to express the cyclic convolution of two sequences x and h is as follows. The transforms of x and h are defined by

$$\begin{aligned} \mathbf{X} &= A\mathbf{x} \\ \mathbf{H} &= B\mathbf{h} \end{aligned} \tag{1.17}$$

where A and B are matrices representing the foward DFT. Then the point by point product of the two sequences is obtained

$$\mathbf{Y} = \mathbf{H} \times \mathbf{X} \tag{1.18}$$

Finally the inverse DFT of $\mathbf{Y}$ is obtained

$$\mathbf{y} = C\mathbf{Y} \tag{1.19}$$

C is the matrix representing the inverse DFT. The processes given by the equations (1.17)-(1.19) can be described by the single equation

$$\mathbf{y} = C\ (\ A\mathbf{x} \times B\mathbf{h}\ ) \tag{1.20}$$

1.4 Rectangular Transforms having the CCP

Agarwal and Cooley [1.5] have shown that other transforms, besides the DFT-like transforms discussed above, have the CCP. This is provided that the A, B and C matrices are non-square. Agarwal and Cooley called these transforms 'Rectangular Transforms'.

Suppose that the A and C matrices are of dimensions MxN and the C matrix MxM. Here N is the length of the cyclic convolution and M is the number of points in the 'transform' domain, M>N. So (1.17) can be rewritten as

$$\mathbb{X}_k = \sum_{q=0}^{N-1} A_{k,q} x_q \quad \text{and} \quad \mathbb{H}_k = \sum_{p=0}^{N-1} B_{k,p} h_p \qquad k=0,1,\ldots,M-1 \tag{1.21}$$

Equation (1.18) becomes

$$y_n = \sum_{k=0}^{M-1} C_{n,k} \mathbb{X}_k \mathbb{H}_k \qquad n=0,1,\ldots,N-1 \tag{1.22}$$

Substituting for $\mathbb{X}_k$ and $\mathbb{H}_k$ from (1.21) into (1.22)

$$y_n = \sum_{k=0}^{M-1} C_{n,k} \left\{ \sum_{q=0}^{N-1} A_{k,q} x_q \right\} \left\{ \sum_{p=0}^{N-1} B_{k,p} h_p \right\} \tag{1.23}$$

Reordering the summations gives

$$y_n = \sum_{p=0}^{N-1} \sum_{q=0}^{N-1} x_q h_p \left\{ \sum_{k=0}^{M-1} C_{n,k} A_{k,q} B_{k,p} \right\} \tag{1.24}$$

A necessary and sufficient condition for (1.24) to represent cyclic convolution is

$$\sum_{k=0}^{M-1} C_{n,k} A_{k,q} B_{k,p} \qquad \begin{aligned} &=1 \text{ if } p=n-q \text{ Mod } N \\ &=0 \text{ otherwise} \end{aligned} \tag{1.25}$$

In the case where M=N Agarwal and Burrus [1.6] have shown that the matrices must have the DFT structure. However if M>N then many different choices are possible. As M is increased the entries in the A, B and C matrices become simpler. In the extreme case where $M=N^2$ each row of the A and B matrices will have only one non-zero element and the algorithm is equivalent to the direct computation of cyclic convolution. The remainder of this thesis is concerned with algorithms in which $N<M<N^2$. In particular, convolution algorithms of this form in which the entries in the A and C matrices are restricted to the values +1, -1 and 0 are considered in detail. Chapter 6 gives a novel implementation of algorithms of this form which is based upon the use of one-bit systolic arrays.

1.5 Rader's Theorem N=p

This section and the subsequent sub-section show how certain DFTs can be converted into circular convolutions (or correlations) by a method proposed by Rader in 1968 [1.7]. The first and least complex case is of a DFT for N=p, where p is an odd prime. Then the DFT is given by

$$X_k = \sum_{n=0}^{p-1} x_n W_N^{nk} \qquad k = 0,1,\ldots,p-1$$

$$W_N = e^{-j2\pi/p}, \; j=\sqrt{-1} \qquad (1.10)$$

For k=0 X_k is simply the summation

$$X_0 = \sum_{n=0}^{p-1} x_n \qquad (1.26)$$

For k≠0

$$X'_k = \sum_{n=1}^{p-1} x_n W_N^{nk}$$

$$X_k = x_0 + X'_k \qquad k = 1,2,\ldots,p-1 \qquad (1.27)$$

The indices n and k are defined modulo p. As N is prime there is some number, not necessarily unique, such that there is a one-to-one mapping of the integers i=1,2,...,N-1 to the integers j=1,2,...,N-1 given by

$$j = g^i \text{ Mod } N$$

The integer g is known as a primitive root of N [1.8]. So for n,k≠0 we can replace n and k by the transformations

$$n = g^u \text{ mod } p$$

$$k = g^v \text{ mod } p \qquad u,v=0,1,\ldots,p-2 \qquad (1.28)$$

so (1.27) becomes

$$X'_{g^v} = \sum_{u=0}^{p-2} x_{g^u} W^{g^{u+v}} \qquad v=0,1,\ldots,p-2 \qquad (1.29)$$

In (1.29) the exponents of g are taken modulo p-1. This may be rewritten as

$$\bar{X}'_v = \sum_{u=0}^{p-2} \bar{x}_u h_{u+v} \qquad v=0,1,\ldots,p-2 \qquad (1.30)$$

where

$$\left\{\bar{x}_u\right\} = \left\{x_{g^u}\right\}, \qquad \left\{\bar{X}'_v\right\} = \left\{X'_{g^v}\right\} \qquad \text{and} \qquad \left\{h_{u+v}\right\} = \left\{W^{g^{u+v}}\right\}.$$

Equation (1.30) represents a circular correlation. To obtain a circular convolution we change the sign of u in (1.29). This corresponds to fixing x_{g^0} and reversing the remainder of the permuted input sequence. The convolution is

$$X'_{g^v} = \sum_{u=0}^{p-2} x_{g^{-u}} W^{g^{v-u}} \qquad v=0,1,\ldots,p-2 \qquad (1.31)$$

By combining (1.31) and (1.26) a DFT of prime length p can always be calculated as a p-1 point convolution with some additions.

As an example consider the calculation of a seven point DFT. This may be written in matrix vector notation as

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 \\ 1 & W^2 & W^4 & W^6 & W^1 & W^3 & W^5 \\ 1 & W^3 & W^6 & W^2 & W^5 & W^1 & W^4 \\ 1 & W^4 & W^1 & W^5 & W^2 & W^6 & W^3 \\ 1 & W^5 & W^3 & W^1 & W^6 & W^4 & W^2 \\ 1 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} \qquad W^7 = 1 \qquad (1.32)$$

For N=7 a suitable primitive root is 3. Thus, excluding x_0, and by using the mapping given in (1.28) with g=3, the input vector is rearranged in the sequence 1,5,4,6,2,3 and the output vector in the sequence 1,3,2,6,4,5. This gives

$$\begin{bmatrix} X'_1 \\ X'_3 \\ X'_2 \\ X'_6 \\ X'_4 \\ X'_5 \end{bmatrix} = \begin{bmatrix} W^1 & W^5 & W^4 & W^6 & W^2 & W^3 \\ W^3 & W^1 & W^5 & W^4 & W^6 & W^2 \\ W^2 & W^3 & W^1 & W^5 & W^4 & W^6 \\ W^6 & W^2 & W^3 & W^1 & W^5 & W^4 \\ W^4 & W^6 & W^2 & W^3 & W^1 & W^5 \\ W^5 & W^4 & W^6 & W^2 & W^3 & W^1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_5 \\ x_4 \\ x_6 \\ x_2 \\ x_3 \end{bmatrix} \qquad W^7 = 1 \qquad (1.33)$$

Equation (1.33) has the form of a cyclic convolution.

With a fast algorithm for calculating p-1 point convolutions it is possible to use Rader's theorem to calculate p point DFTs - this is one of the key points in the derivation of the Winograd Fourier Transform Algorithm. The case where $N=p^r$ is more complex and is described below.

1.5.1 Rader's Theorem $N=p^r$

The case when $N=p^r$, p prime, has been discussed by several authors; Kolba and Parks [1.9], Nussbaumer [1.10] and by McClellan and Rader in the introduction to [1.11]. It is possible to convert a DFT of $N=p^r$ points into a series of convolutions. The first step is to change the index k so that

$$k = pk_1 + k_2 \qquad k_1 = 0,1,\ldots,p^{r-1}-1$$
$$k_2 = 0,1,\ldots,p-1 \qquad (1.34)$$

Subsequently, for $k_2=0$ we have $k \equiv 0$ modulo p and X_k becomes

$$X_{pk_1} = \sum_{n=0}^{p^r-1} x_n W^{pnk_1} \qquad (1.35)$$

By way of an illustration of the decomposition of the $N=p^r$ point case a 9-point is used as an example. The 9-point DFT matrix is

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 & W^8 \\ 1 & W^2 & W^4 & W^6 & W^8 & W^1 & W^3 & W^5 & W^7 \\ 1 & W^3 & W^6 & 1 & W^3 & W^6 & 1 & W^3 & W^6 \\ 1 & W^4 & W^8 & W^3 & W^7 & W^2 & W^6 & W^1 & W^5 \\ 1 & W^5 & W^1 & W^6 & W^2 & W^7 & W^3 & W^8 & W^4 \\ 1 & W^6 & W^3 & 1 & W^6 & W^3 & 1 & W^6 & W^3 \\ 1 & W^7 & W^5 & W^3 & W^1 & W^8 & W^6 & W^4 & W^2 \\ 1 & W^8 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} \qquad W^9 = 1 \tag{1.36}$$

Equation (1.35) describes the rows corresponding to X_0, X_3 and X_6.

$$\begin{pmatrix} X_0 \\ X_3 \\ X_6 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W^3 & W^6 & 1 & W^3 & W^6 & 1 & W^3 & W^6 \\ 1 & W^6 & W^3 & 1 & W^6 & W^3 & 1 & W^6 & W^3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} \tag{1.37}$$

Then in (1.35) since W^{pnk_1} defines n modulo p^{r-1}, i.e. Modulo 3 in (1.37), we change the index n to

$$n = p^{r-1}n_1 + n_2 \qquad \begin{aligned} n_1 &= 0,1,\ldots,p-1 \\ n_2 &= 0,1,\ldots,p^{r-1}-1 \end{aligned} \tag{1.38}$$

Then for $k_2=0$, X_k becomes a DFT of p^{r-1} points,

$$X_{pk_1} = \sum_{n_2=0}^{p^{r-1}-1} \sum_{n_1=0}^{p-1} x_{p^{r-1}n_1+n_2} \, W^{pk_1n_2} \tag{1.39}$$

So in the 9-point DFT example (1.37) can be rewritten as a 3-point DFT,

$$\begin{bmatrix} X_0 \\ X_3 \\ X_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & W^3 & W^6 \\ 1 & W^6 & W^3 \end{bmatrix} \begin{bmatrix} x_0 + x_3 + x_6 \\ x_1 + x_4 + x_7 \\ x_2 + x_5 + x_8 \end{bmatrix} \qquad (1.40)$$

We now return to the k index given by (1.34). For $k \not\equiv 0$ modulo p, we compute separately the terms corresponding to $n \equiv 0$ mod p (A_k) and to $n \not\equiv 0$ mod p (B_k),

$$X_k = A_k + B_k \qquad (1.41)$$

Firstly, for $n \equiv 0$ mod p

$$n = pn_1 \qquad n_1 = 0,1,\ldots,p^{r-1}-1 \qquad (1.42)$$

So by reordering k, we have

$$k = p^{r-1}k_1 + k_2 \qquad k_1 = 0,1,\ldots,p-1$$

$$k_2 \not\equiv 0 \text{ mod } p \qquad k_2 = 1,\ldots,p^{r-1}-1 \qquad (1.43)$$

so

$$A_{p^{r-1}k_1+k_2} = \sum_{n_1=0}^{p^{r-1}-1} x_{pn_1} W^{pn_1k_2} \qquad (1.44)$$

The righthand side of (1.44) is independant of k_1 so A_k is a DFT of p^{r-1} points in which the terms corresponding to $k_2 \equiv 0$ are not calculated, In the 9-point DFT example we have (equation (1.42)),

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \\ A_8 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & W^3 & W^6 \\ 1 & W^6 & W^3 \\ 1 & 1 & 1 \\ 1 & W^3 & W^6 \\ 1 & W^6 & W^3 \\ 1 & 1 & 1 \\ 1 & W^3 & W^6 \\ 1 & W^6 & W^3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_3 \\ x_6 \end{bmatrix} \qquad (1.45)$$

Using this example (1.44) describes

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \end{bmatrix} = \begin{bmatrix} A_3 \\ A_4 \\ A_5 \end{bmatrix} = \begin{bmatrix} A_6 \\ A_7 \\ A_8 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & W^3 & W^6 \\ 1 & W^6 & W^3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_3 \\ x_6 \end{bmatrix} \qquad (1.46)$$

The condition that $k_2 \not\equiv 0$ mod p in (1.43) is because the values of X_{pl} $l=0,1,\ldots,p^{r-1}-1$, have already been calculated in (1.39).

Finally we consider B_k of (1.41). This consists of terms for which $n,k \not\equiv 0$ mod p. B_k is of length $p^{r-1}(p-1)$ points. Then, in a similar manner to the N=p case above, the indices n and k can be generated by a primitive root g defined modulo p^r with

$$n = g^u \bmod p^r$$

$$k = g^v \bmod p^r \qquad u,v = 0,1,\ldots,p^{r-1}(p-1)-1 \qquad (1.47)$$

Thus a correlation of length $p^{r-1}(p-1)$ is obtained

$$B_{g^v} = \sum_{u=0}^{p^{r-1}(p-1)-1} x_{g^u} W^{g^{u+v}} \qquad (1.48)$$

For the 9-point DFT the 6-point convolution is obtained using a primitive root equal to 2. This gives (1.48) as

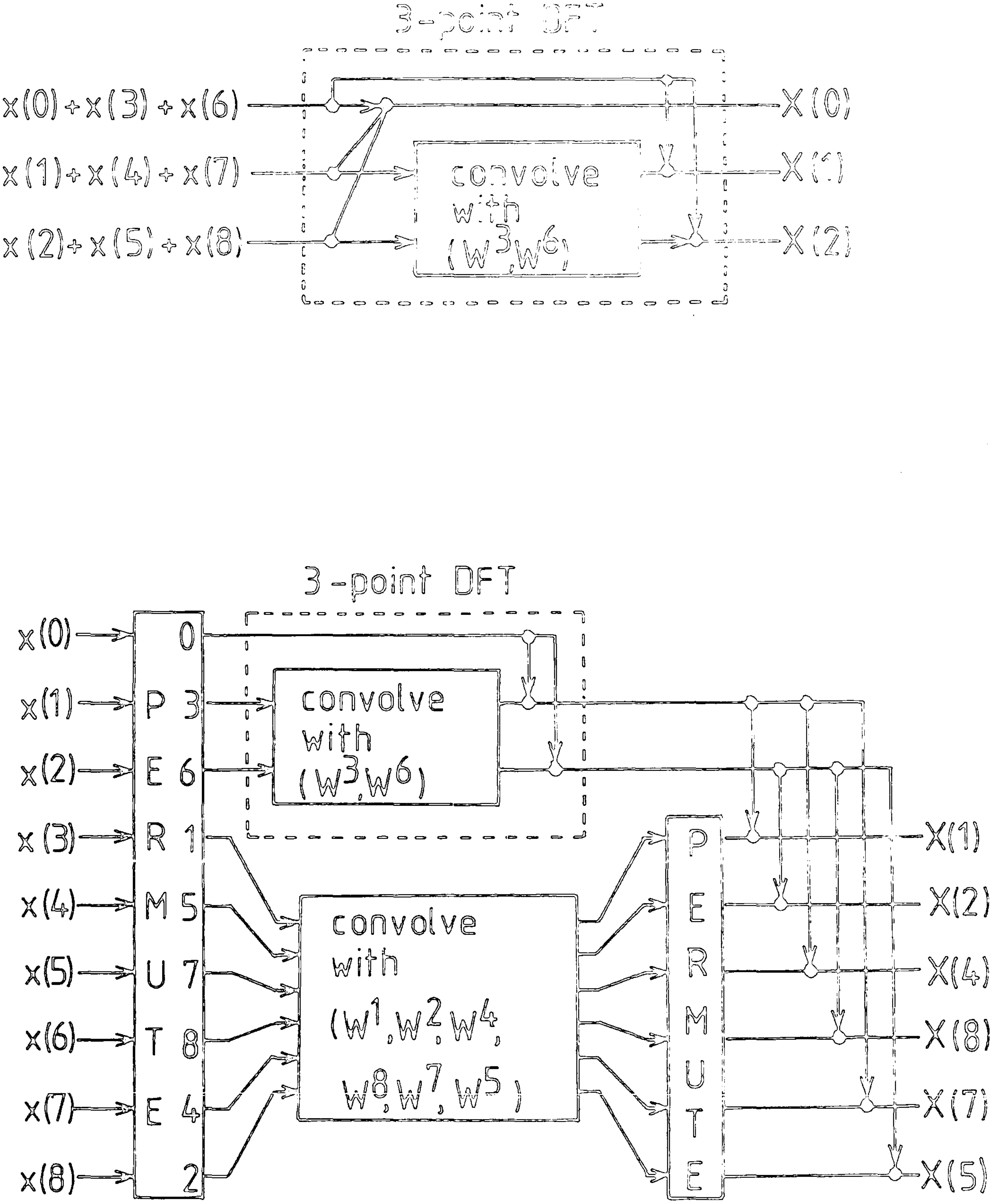


Fig 1.3 9-point DFT Decomposition

using Rader's Theorem

$$\begin{bmatrix} B_1 \\ B_2 \\ B_4 \\ B_8 \\ B_7 \\ B_5 \end{bmatrix} = \begin{bmatrix} W^1 & W^5 & W^7 & W^8 & W^4 & W^2 \\ W^2 & W^1 & W^5 & W^7 & W^8 & W^4 \\ W^4 & W^2 & W^1 & W^5 & W^7 & W^8 \\ W^8 & W^4 & W^2 & W^1 & W^5 & W^7 \\ W^7 & W^8 & W^4 & W^2 & W^1 & W^5 \\ W^5 & W^7 & W^8 & W^4 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_5 \\ x_7 \\ x_8 \\ x_4 \\ x_2 \end{bmatrix} \qquad W^9 = 1 \qquad (1.49)$$

Each of the subsiduary DFTs (1.39) and (1.44) in the calculation of a p^r point DFT can themselves be decomposed into convolutions. So as noted by Kolba and Parks [1.9], this 9-point DFT requires a 6-point cyclic convolution (1.49) and two 3-point DFTs (1.40) and (1.46). Figure 1.3 illustrates this decomposition of the 9-point DFT.

When N is a power of two, the N-point DFT is partitioned into DFTs of size N/2 by the same method, and the DFTs corresponding to n and k odd are computed as correlations. However, there are no primitive roots for N>4. For N>4 the composite root $(-1)^{n_1}3^{n_2}$, with $n_1=0,1$ and $n_2=0,\ldots,(N/4-1)$ should be used. These roots generate a two-dimensional correlation of size 2x(N/4).

So Rader's decomposition yields a method of computation, based upon convolutions, for p and p^r point DFTs. The major significance of Rader's algorithm is that it allows one to compute large DFTs very efficiently when it is combined with other techniques.

1.6 Summary Chapter 1

This chapter has provided a review of some of the fundamental techniques for calculating convolutions and DFTs. Firstly the techniques of Overlap-Add and Overlap-Save were introduced to show how long aperiodic convolutions can be calculated from a series of shorter cyclic convolutions. Then the Cyclic Convolution Property (CCP) was derived to show how cyclic convolutions may be calculated using transform techniques. However Rader's theorem shows how these tranforms themselves may be expressed as convolutions. The use of this structure of convolutions within convolutions is used in latter chapters concerned with the development of two-dimensional convolution and Fourier Transform algorithms.

Chapter 2

'Short-N' DFT and Convolution Algorithms

This chapter deals with the derivation of fast convolution algorithms by the means of polynomial algebra and the Chinese Remainder Theorem (CRT). These short convolution algorithms are then applied to the DFT using Rader's Theorem as outlined at the end of chapter one.

2.1 Convolution and Polynomial Algebra

Consider the aperiodic convolution y_n of two sequences h_k and x_m, each of N terms,

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k} \qquad n=0,1,\ldots,2N-2 \qquad (2.1)$$

Now suppose that the N elements of the h_k and x_m are assigned to be coefficients of the polynomials $H(z)$ and $X(z)$ of degree N-1 in z. i.e.

$$H(z) = \sum_{k=0}^{N-1} h_k z^k$$

$$X(z) = \sum_{k=0}^{N-1} x_k z^k \qquad (2.2)$$

Taking the product $H(x)X(z)$ the resulting polynomial $Y(z)$ will be of order 2N-2. Thus

$$Y(z) = H(z)X(z) = \sum_{n=0}^{2N-2} a_n z^n \qquad (2.3)$$

In this polynomial multiplication each coefficient, a_n of z^n, is found by summing all the products $h_k x_m$ so that $n=k+m$, i.e. $m=n-k$. It follows that

$$a_n = \sum_{k=0}^{N-1} h_k x_{n-k} = y_n$$

and so $\quad Y(z) = \sum_{n=0}^{2N-2} y_n z^n \qquad (2.4)$

The implication of this is that the multiplication of two polynomials is equivalent to the convolution of two sequences. Moreover, if the convolution defined by (2.1) is cyclic the indices are all defined modulo N. Thus in a length N cyclic convolution we have $N \equiv 0$, implying that $z^N \equiv 1$. So a cyclic convolution is the product of two polynomials modulo the polynomial $z^N - 1$.

$$Y(z) = H(z)X(z) \bmod (z^N - 1) \qquad (2.5)$$

2.1.1 The Toom-Cook Algorithm

The Toom-Cook algorithm provides a method of constructing the polynomial product (2.3) by using the Lagrange interpolation formula. The Toom-Cook algorithm is a special case of a method for constructing polynomial products using the Chinese Remainder Theorem. The more general case is considered later.

Knuth [2.1] discusses the use of the Toom-Cook algorithm for multiplications. Agarwal and Cooley [2.2] discuss the use of the Toom-Cook algorithm for the calculation of aperiodic convolutions, i.e. polynomial products.

Theorem (The Toom-Cook Algorithm)

The polynomial product (2.3) can be computed in 2N-1 general multiplications.

A general multiplication is one where both multiplicands depend upon the data. The theorem is proved by constructing the algorithm.

Suppose the polynomials given by (2.2) are formed and the product $Y(z)$ (2.3) found. $Y(z)$ is a polynomial of degree $2N-2$. To determine the $2N-1$ a_ns of (2.3) one can select $2N-1$ distinct numbers α_j $j=0,1,\ldots,2N-2$, and substitute them for z in (2.3) and obtain the $2N-1$ products

$$Y(\alpha_j) = E(\alpha_j)X(\alpha_j) \qquad j=0,1,\ldots,2N-2 \qquad (2.6)$$

Then the Lagrange interpolation formula may be used to uniquely determine the $2N-2$ degree polynomial

$$Y(z) = \sum_{j=0}^{2N-2} Y(\alpha_j)L_j(z) \qquad (2.7)$$

where the interpolating polynomials are

$$L_j(z) = \prod_{\substack{k=0\\k\neq j}}^{2N-2} \cdot \frac{(z-\alpha_k)}{(\alpha_j-\alpha_k)} \qquad (2.8)$$

So the polynomial product (2.3) is found using the $2N-1$ multiplications in (2.6). As an example of the use of the Toom-Cook algorithm consider the product of two polynomials

$$H(z) = h_2z^2 + h_1z + h_0 \quad \text{and} \quad X(z) = x_2z^2 + x_1z + x_0$$

giving

$$Y(z) = y_4z^4 + y_3z^3 + y_2z^2 + y_1z + y_0 \qquad (2.9)$$

As both these polynomials are of degree two the Toom-Cook algorithm states that their product may be found with five general multiplications. Suppose the α_j are chosen as

$$\alpha_0 = 0, \quad \alpha_1 = 1, \quad \alpha_2 = 2, \quad \alpha_3 = 3 \quad \text{and} \quad \alpha_4 = 4.$$

This is a purely arbitrary choice, negative values of α_j could have been employed. Then the $Y(\alpha_k)$ of (2.6) are found by substituting the values of α_j above into (2.9).

$$Y(0) = H(0)X(0) = h_0x_0$$

$$Y(1) = (h_2 + h_1 + h_0)(x_2 + x_1 + x_0)$$

$$Y(2) = (4h_2 + 2h_1 + h_0)(4x_2 + 2x_1 + x_0)$$

$$Y(3) = (9h_2 + 3h_1 + h_0)(9x_2 + 3x_1 + x_0)$$

$$\mathbb{Y}(4) = (16h_2 + 4h_1 + h_0)(16x_2 + 4x_1 + x_0) \qquad (2.10)$$

The interpolating polynomials (2.8) are functions of the α_ks (2.10), not the coefficients of the polynomials $\mathbb{H}(z)$ and $\mathbb{X}(z)$ (2.9). These interpolation polynomials are,

$$\mathbb{L}_0(z) = \frac{(z-1)(z-2)(z-3)(z-4)}{(0-1)(0-2)(0-3)(0-4)} = 1/24\ (z^4-10z^3+35z^2-50z+24)$$

$$\mathbb{L}_1(z) = \frac{(z-0)(z-2)(z-3)(z-4)}{(1-0)(1-2)(1-3)(1-4)} = -1/6\ (z^4-9z^3+26z^2-24z)$$

$$\mathbb{L}_2(z) = 1/4\ (z^4-8z^3+19z^2-12z)$$

$$L_3(z) = -1/6\ (z^4-7z^3+14z^2-8z)$$

$$\mathbb{L}_4(z) = 1/24\ (z^4-6z^3+11z^2-6z) \qquad (2.11)$$

Collecting terms in like powers of z in (2.11) gives

$$y_0 = \mathbb{Y}(0)$$

$$y_1 = -\frac{25}{12}\mathbb{Y}(0) + 4\mathbb{Y}(1) - 3\mathbb{Y}(2) + \frac{4}{3}\mathbb{Y}(3) - \frac{1}{4}\mathbb{Y}(4)$$

$$y_2 = \frac{35}{24}\mathbb{Y}(0) - \frac{13}{3}\mathbb{Y}(1) + \frac{19}{4}\mathbb{Y}(2) - \frac{7}{3}\mathbb{Y}(3) + \frac{11}{24}\mathbb{Y}(4)$$

$$y_3 = -\frac{5}{12}\mathbb{Y}(0) + \frac{3}{2}\mathbb{Y}(1) - 2\mathbb{Y}(2) + \frac{7}{6}\mathbb{Y}(3) - \frac{1}{4}\mathbb{Y}(4)$$

$$y_4 = \frac{1}{24}\mathbb{Y}(0) - \frac{1}{6}\mathbb{Y}(1) + \frac{1}{4}\mathbb{Y}(2) - \frac{1}{6}\mathbb{Y}(3) + \frac{1}{24}\mathbb{Y}(4) \qquad (2.12)$$

Although this polynomial product algorithm, described by (2.9),(2.10) and (2.12), may appear to contain more than 5 multiplications, the only 'general' multiplications are those in (2.10). All the multiplications by constants in (2.12) do not count as general multiplications.

As mentioned above the choice of the 2N-1 distinct values of α_j is arbitary. The example shows how the use of integers other than +1,-1 and 0, in the choice of the α_js, quickly involves multiplications by inconvenient constants which are not counted as general multiplications. These constants limit the usefulness of the Toom-Cook algorithm to very short polynomial products.

Nussbaumer [2.3 pp 27-29] explains how the Overlap-Add method can be derived from the Toom-Cook algorithm by choosing the $\alpha_j = W^j_{2N-1}$ $j=0,1,\ldots,2N-1$. Agarwal and Burrus [2.2] show that the Toom-Cook algorithm is of the general canonical form

$$\mathbf{Y} = C\ (A\mathbf{x}\ \mathrm{x}\ B\mathbf{h}) \tag{1.20}$$

The polynomial representation of convolution is an extremely useful tool for developing convolution algorithms. An essential part of this development is the use of the Chinese Remainder Theorem.

2.2 The Chinese Remainder Theorem

This theorem, which was first known in ancient China [2.1] allows you to construct the solution to the following type of problem. Given

$$x = 4 \bmod 5$$

$$x = 1 \bmod 4$$

$$x = 2 \bmod 3$$

find the smallest integer x that satisfies these conditions. Expressing the same problem more formally the Chinese Remainder Theorem (CRT) allows you to construct a unique solution from a set of congruences in mutually prime moduli. Whilst the integer version of the CRT is used in chapter 4 here, we are more concerned with a polynomial version of it. Before the polynomial form is introduced the concepts of residue polynomials and irreducibility are discussed.

2.2.1 The Chinese Remainder Theorem for Integers

Let m_1, m_2, ..., m_r be positive integers which are relatively prime in pairs, i.e.,

$$(m_j, m_k) = 1 \qquad \text{when } j \neq k \tag{2.13}$$

where (.) denotes greatest common divisor. Let a, u_1, ..., u_r be integers. Then there is exactly one integer u which satisfies the conditions

$$a \leq u < a+M \quad \text{and} \quad u \equiv u_j \mod m_j \text{ for } 1 \leqslant j \leqslant r \qquad (2.14)$$

Proof: If $u \equiv v \mod m_j$ for $1 \leqslant j \leqslant r$, then u-v is a multiple of m_j for all j. So (2.13) implies that u-v is a multiple of $M = m_1 m_2 \ldots m_r$. This argument shows that there is at most one solution to (2.14). The proof is completed by showing the existence of at least one solution.

As u runs through the M distinct values $a \leq u < a+M$, the r-tuples ($u \mod m_1$, ..., $u \mod m_r$) must also run through M distinct values since (2.14) has at least one solution. There are however $m_1 m_2 \ldots m_r$ possible r-tuples. Therefore each r-tuple must occur exactly once, and there must be some value of u for which ($u \mod m_1$, ..., $u \mod m_r$) = (u_1, ..., u_r).

Given the residues the integer u may be determined by means of the formula

$$u = \left\langle \frac{M}{m_1} \left\langle \frac{M}{m_1} \right\rangle_{m_1}^{-1} u_1 + \frac{M}{m_2} \left\langle \frac{M}{m_2} \right\rangle_{m_2}^{-1} u_2 + \ldots + \frac{M}{m_r} \left\langle \frac{M}{m_r} \right\rangle_{m_r}^{-1} u_r \right\rangle_M \qquad (2.15)$$

where $\langle . \rangle_k$ denotes reduction modulo k. The use of this formula is now illustrated.

First the quantities M_i are found, where

$$M_1 = \frac{M}{m_1}, \qquad M_2 = \frac{M}{m_2}, \qquad \ldots, \quad M_r = \frac{M}{m_r} \qquad (2.16)$$

Each M_i is relatively prime to its corresponding m_i. It is possible to find (see [2.4]) number N_i solving

$$N_i M_i = 1 \qquad \text{modulo } m_i \qquad (2.17)$$

Now consider the quantity

$$U = u_1 N_1 M_1 + u_2 N_2 M_2 + \ldots + u_r N_r M_r \qquad (2.15)$$

Then taking the residue of U modulo m_1, each of the factors containing M_j $j \neq 1$ has m_1 as a common factor, so

$$U = u_1 N_1 M_1 \quad \text{modulo } m_1 \qquad (2.18)$$

But since $N_1 M_1 = 1$ modulo m_1

$$U = u_1 \text{ modulo } m_1 \qquad (2.19)$$

similarly $$U = u_i \text{ modulo } m_i \qquad (2.20)$$

Thus U is a solution to a given set of congruences. The problem stated at the beginning of this section is now solved,

$$x = 4 \bmod 5$$

$$x = 1 \bmod 4$$

$$x = 2 \bmod 3 \quad \text{Find } x.$$

The three moduli are 3,4 and 5 so M=60. The M_i are

$$M_1 = \frac{60}{5} = 12, \quad M_2 = \frac{60}{4} = 15, \quad \text{and} \quad M_3 = \frac{60}{3} = 20.$$

Then N_1 is the solution to $12N_1=1$ mod 5, which is equivalent to $2N_1=1$ mod 5, i.e. $N_1=3$. Similarly $15N_2=1$ mod 4 gives $N_2=3$ and $20N_3=1$ mod 3 gives $N_3=2$. So

$$x = 12.3.4 + 15.3.1 + 20.2.2 \bmod 60$$

$$\underline{x = 29.}$$

2.2.2 Residue Polynomials

We now turn to polynomial arithmetic and define polynomial equivalents of 'congruent' and 'remainder'. The idea of irreducibility of polynomials is introduced; the factorization of (z^N-1) into irreducible factors is of great importance in the derivation of the Winograd minimal complexity convolution algorithms.

A polynomial d(z) divides a second polynomial H(z) if a polynomial $\mathbb{p}(z)$ exists such that

$$\mathbb{H}(z) = \mathbb{d}(z)\mathbb{p}(z) \qquad (2.21)$$

A polynomial $\mathbb{H}(z)$ whose only divisors are of degree equal to zero or deg($\mathbb{p}(z)$) is said to be irreducible in the field, F, of the coefficients of $\mathbb{H}(z)$. Notice that the irreducibility of $\mathbb{H}(z)$ depends upon the field of coefficients. This is illustrated by the following example,

a) z^5-1 factors as $\prod_{k=0}^{4}(z - W_5^k)$ over the field of complex numbers.

b) As $(z - 1)(z^2 - 2\cos\frac{2\pi}{5} z + 1)(z^2 - 2\cos\frac{4\pi}{5} z + 1)$ over the real numbers

c) As $(z -1)(z^4 + z^3 + z^2 + z +1)$ over the rationals.

In a similar manner to the case for integers every polynomial can be written uniquely in the form

$$p(z) = k \prod_{i=1}^{d} [p_i(z)]^{r_i} \tag{2.22}$$

where k is a constant, $p_i(z)$ are irreducible monic polynomials and d is the number of factors of p(z). A monic polynomial is one whose leading coefficient is unity. Equation (2.22) implies that

$$\sum_{i=1}^{d} r_i\ [\deg p_i(z)] = \deg[p(z)].$$

For two polynomials H(z) and D(z) it is always possible to write

$$H(z) = P(z)D(z) + R(z) \tag{2.23}$$

where deg[R(z)]<deg[D(z)], R(z) is known as the remainder or residue polynomial. The representation in (2.23) is unique [see 2.4 p55].

Two polynomials $p_1(z)$ and $p_2(z)$ are said to be congruent modulo d(z) if they have the same residue modulo d(z).

2.2.3 The Chinese Remainder Theorem for Polynomials

There exists a unique polynomial Y(z) satisfying

a) $Y(z) \equiv Y_i(z) \mod M_i(z)$, $i=1,2,\ldots,d$

b) $$0<\deg[Y(z)]< \sum_{i=1}^{d} \deg\ [M_i(z)] \tag{2.24}$$

Provided that the monic polynomials $M_i(z)$ are relatively prime in pairs.

A proof of this theorem is given in [2.4]. Y(z) may be constructed from the congruences by means of the formula,

$$Y(z) = \sum_{i=1}^{d} S_i(z)Y_i(z) \qquad \text{mod } M(z) \tag{2.25}$$

where the auxilary polynomials $S_i(z)$ are defined by

$$S_j(z) \equiv \left\{\left\{\prod_{\substack{i=1\\i\neq j}}^{d} M_i(z)\right\} \Big/ \left\{\prod_{\substack{i=1\\i\neq j}}^{d} M_i(z) \text{ mod } M_j(z)\right\}\right\} \quad \text{mod } M(z) \tag{2.26}$$

where $M(z) = M_1(z)M_2(z)....M_d(z)$

The d auxilary polynomials are such that

$$\begin{aligned} S_j(z) &\equiv 0 \text{ mod } M_i(z) \quad i\neq j \\ &\equiv 1 \text{ mod } M_j(z) \end{aligned} \tag{2.27}$$

Reducing $S_j(z)$ defined by (2.26) modulo $M_j(z)$ gives (2.27) provided that

$$\prod_{\substack{i=1\\i\neq j}}^{d} M_i(z) \quad \not\equiv 0 \text{ mod } M_j(z)$$

This last condition is ensured by the polynomials $M_j(z)$ being relatively prime.

As an example, which is applied to a 4-point convolution algorithm in the latter parts of this chapter, the $S_j(z)$s are constructed for the three polynomials (z-1), (z+1) and (z^2+1).

$$M_1 = (z-1) \quad M_2 = (z+1) \quad M_3 = (z^2+1) \quad M=M_1M_2M_3 = z^4-1$$

then $S_1(z) =\left\{[(z+1)(z^2+1)] \Big/ \left\{ [(z+1)(z^2+1)] \quad \text{mod } z-1\right\}\right\} \text{ mod } (z^4-1)$

but $(z+1)(z^2+1) = z^3+z^2+z+1$

and $z^3+z^2+z+1= 4 \text{ mod } (z-1)$

giving $$\underline{S_1(z) = \tfrac{1}{4}(z^3+z^2+z+1)} \tag{2.28a}$$

$$S_2(z) = \left\{[(z-1)(z^2+1)] \Big/ \left\{[(z-1)(z^2+1)] \mod z+1\right\}\right\} \mod (z^4-1)$$

but $(z-1)(z^2+1) = z^3-z^2+z-1$

and $z^3-z^2+z-1 = -4 \mod (z+1)$

giving $\underline{S_2(z) = -\frac{1}{4}(z^3-z^2+z-1)}$ (2.28b)

$$S_3(z) = \left\{[(z-1)(z+1)] \Big/ \left\{[(z-1)(z+1)] \mod z^2+1\right\}\right\} \mod (z^4-1)$$

but $(z-1)(z+1) = z^2-1$

and $z^2-1 = -2 \mod (z^2+1)$

giving $\underline{S_3(z) = -\frac{1}{2}(z^2-1)}$ (2.28c)

This is a particularly simple set of recombination polynomials. Often the calculation of

$$\left\{[\prod_{\substack{i=1\\i\neq j}}^{d} M_i(z)] \mod M_j(z)\right\}^{-1} \mod M_j(z)$$

is more difficult and involves the long division of polynomials to determine the remainder.

In the Toom-Cook algorithm, $M_i(z) = z-z_i$ and the Lagrangian interpolation formula provides a method of constructing $Y(z)$.

2.3 Computation of Convolutions using the Chinese Remainder Theorem

We now consider the problem of calculating

$$Y(z) = H(z)X(z) \mod P(z) \tag{2.29}$$

In section 2.1 above, (2.29) was shown to be a circular convolution when $P(z)$ is chosen as z^N-1. The problem in (2.29) can be re-expressed if the polynomial $P(z)$ can be factored as

$$P(z) = \prod_{i=1}^{d} P_i(z) \tag{2.30}$$

The major simplification comes from noting that if

$$Y_i(z) = H(z)X(z) \mod P_i(z) \quad i=1,\ldots,d \tag{2.31}$$

Then the Chinese Remainder Theorem can be used to construct $Y(z)$ from the $Y_i(z)$ provided that the $P_i(z)$ are mutually prime in pairs.

As shown in section 2.2.2 the factorization of $P(z)$ depends upon the field of coefficients. For the moment this field is left unspecified.

The convolution (2.29) can now be represented as d subproblems. Each subproblem being to reduce $X(z)$ and $H(z)$ to find $X_i(z)$ and $H_i(z)$. The product $H_i(z)X_i(z)$ is then found modulo $P_i(z)$. This product can be calculated using the Toom-Cook algorithm derived earlier. Since $X_i(z)$ and $H_i(z)$ will be of degree $(\deg[P_i(z)]-1)$, the Toom-Cook algorithm will need $2\deg[P_i(z)]-1$ general multiplications. As stated earlier some multiplications by fixed constants in the field of coefficients of $X_i(z)$ and $H_i(z)$ may be needed - these are <u>not</u> counted. The residue reduction modulo $P_i(z)$ will require additions, subtractions and multiplications by fixed constants in the field of coefficients. So each of the d subproblems can be computed with $2\deg[P_i(z)]-1$ multiplications. The final step of the computation of (2.29) is the reconstruction of $Y(z)$ from the $Y_i(z)$ using the Chinese Remainder Theorem. The general form of this being

$$Y(z) = \sum_{i=1}^{d} S_i(z)Y_i(z) \mod P(z) \tag{2.25}$$

The coefficients of each polynomial $S_i(z)$ lie in the same field as the coefficients of the $P_i(z)$. So (2.25) requires only additions, subtractions and multiplications from the field of coefficients of the $P_i(z)$.

Then the total number of general multiplications used in the d subproblems for the calculation of (2.29) is

$$\sum_{i=1}^{d} 2\deg[P_i(z)]-1 = 2\deg[P(z)]-d \tag{2.32}$$

This does <u>not</u> include multiplications by constants. For the case of cyclic convolution deg $\mathbb{P}(z) = N$ so the number of general multiplications needed is

$$2N-d \qquad (2.33)$$

This result is a case of a theorem due to Winograd [2.5].

Winograd's Minimum Complexity Theorem

The polynomial product $\mathbb{Y}(z)=\mathbb{H}(z)\mathbb{X}(z)$ modulo $\mathbb{P}(z)$ may be computed with a minimum of $2N-d$ general multiplications. Where $N =$ deg $\mathbb{P}(z)$ and d is the number of irreducible factors of $\mathbb{P}(z)$ over the field F.

The preceeding section showed how an algorithm using $2N-d$ general multiplications can be found. In [2.5] Winograd shows that at least $2N-d$ multiplications are needed and that any algorithm using the minimum number of multiplications must use the Chinese Remainder Theorem.

2.3.1 The Canonical Form of Winograd's Algorithms

Winograd [2.6,2.7] derived a series of 'short N' convolution algorithms using the Chinese Remainder Theorem recombination method outlined in the previous section. These algorithms were derived with the field of coefficients as the rational numbers. It is interesting to consider the more general form of the minimal algorithms in other fields.

Winograd's algorithms all have the same form

$$\mathbb{Y} = C\,(\,Ax \times Bh\,) \qquad (1.20)$$

The A, B and C matrices are rectangular and their entries are constrained to lie in the field, F. For 'minimal' algorithms the A and B matrices are of dimension $(2N-d)xN$ and the C matrix $Nx(2N-d)$. The general multiplications are contained in the step ($Ax \times Bh$).

McClellan and Rader [2.4] show that it is possible to derive the Cyclic Convolution Property (CCP) from Winograd's theorem. Furthermore, by choosing the field of coefficients as the complex numbers, they show that the A and B matrices must be foward DFTs and that the C matrix is constrained to be the inverse DFT. Notice that in this case all the multiplications associated with the DFTs are not counted as 'general' multiplications as they are constants in the field!

Winograd's minimal algorithms are only of interest if the fixed constants in the field are simple. Choosing the field of constants as the rational numbers, whilst eliminating complex values, could still leave awkward values to implement. There is no way to force the multipliers to be simple, but the use of the Rational numbers, at least for small algorithms, does give simple valued multipliers.

When the field of the coefficients is the field of rational numbers, z^N-1 factors into polynomials whose coefficients are rational numbers. These polynomials are called cyclotomic polynomials [2.8]. In view of their importance the next section is devoted to cyclotomic polynomials.

2.3.2 Cyclotomic Polynomials

This section deals with a few of the properties of cyclotomic polynomials. There is an excellent section on these polynomials in [2.4]. Some of the properties of these polynomials are listed here. Many of the results are proved in [2.4].

The kth cyclotomic polynomial $C_k(z)$ is defined to be

$$C_k(z) = \prod_{\substack{(m,k)=1 \\ 0<m<k}} (z - W_k^m) \qquad (2.34)$$

Some important properties of these polynomials are:

(i) Cyclotomic polynomials are irreducible over the field of rational numbers. This is proved in [2.8].

(ii) For a given N the number, d, of distinct polynomial factors of z^N-1 is equal to the number of divisors of N including 1 and N.

(iii) The degree of $\mathbb{C}_k(z)$ is $\emptyset(k)$ where $\emptyset(k)$ is Euler's totient function. The value of this function is the number of integers that are smaller than k and that are relatively prime to k.

(iv) For p prime

$$\mathbb{C}_p(z) = z^{p-1} + z^{p-2} + \dots + z + 1 \qquad (2.35)$$

(v) For any choice of integers m and p

$$\mathbb{C}_{mp^k}(z) = \mathbb{C}_{mp}(z^{p^{k-1}}) \qquad (2.36)$$

(vi) For p prime and if p does not divide m (i.e. (p,m)=1) then

$$\mathbb{C}_{pm}(z) = \frac{\mathbb{C}_m(z^p)}{C_m(z)} \qquad (2.37)$$

(vii) For n odd and $n \geq 3$.

$$\mathbb{C}_{2n}(z) = C_n(-z) \qquad (2.38)$$

(viii) The coefficients of the $C_k(z)$ are integers.

(ix) $\mathbb{C}_k(z)$ has coefficients from the set 0, 1, -1 when k has at most two different odd prime factors. Since 105 = 3.5.7 it is the smallest integer to be divisible by three odd primes and so all cyclotomic polynomials will have this property for k<105.

Examples of some cyclotomic polynomials are given in table 2.1.

$C_1(z) = z-1$	$C_8(z) = z^4 + 1$
$C_2(z) = z+1$	$C_9(z) = z^6 + z^3 + 1$
$C_3(z) = z^2 + z + 1$	$C_{10}(z) = z^4 - z^3 + z^2 - z + 1$
$C_4(z) = z^2 + 1$	$C_{12}(z) = z^4 - z^2 + 1$
$C_5(z) = z^4 + z^3 + z^2 + z + 1$	$C_{16}(z) = z^8 + 1$
$C_6(z) = z^2 - z + 1$	

Table 2.1
Some Cyclotomic Polynomials

2.4 Construction of Short Convolution and DFT Algorithms of the Winograd Type

This section deals with the derivation of short convolution and DFT algorithms by the method discussed in the preceeding section. An important result is derived in section 2.4.2. This result is the Matrix exchange or Transpose system.

2.4.1 A 4-point Cyclic Convolution Algorithm

As an example of the method of deriving cyclic convolution algorithms a 4-point convolution algorithm is found.

A 4-point cyclic convolution is equivalent to polynomial product modulo z^4-1. The cyclotomic factors of z^4-1 are $z-1$, $z+1$ and z^2+1. The first step in the construction is to reduce the input polynomials $X(z)$ and $H(z)$ by each of these moduli in turn.

$$X(z) = x_3z^3+x_2z^2+x_1z+x_0 \qquad H(z) = h_3z^3+h_2z^2+h_1z+h_0$$

$$X_1(z) = X(z) \text{ Mod } (z-1) \qquad H_1(z) = H(z) \text{ Mod } (z-1)$$

$$= x_0+x_1+x_2+x_3 \qquad = h_0+h_1+h_2+h_3$$

$$= x_{10} \qquad = h_{10}$$

$$X_2(z) = X(z) \text{ Mod } (z+1) \qquad H_2(z) = H(z) \text{ Mod } (z+1)$$

$$= x_0-x_1+x_2-x_3 \qquad = h_0-h_1+h_2-h_3$$

$$= x_{20} \qquad = h_{20}$$

$$X_3(z) = X(z) \text{ Mod } (z^2+1) \qquad H_3(z) = H(z) \text{ Mod } (z^2+1)$$

$$= (x_1-x_3)z + (x_0-x_2) \qquad = (h_1-h_3)z + (h_0-h_2)$$

$$= x_{31}z + x_{30} \qquad = h_{31}z + h_{30}$$

The second stage of the algorithm is to perform the three polynomial products

(i) $Y_1(z) = X_1(z)H_1(z) \text{ mod } (z-1) = (x_{10}h_{10}) = y_{10}$

(ii) $Y_2(z) = X_2(z)H_2(z) \text{ mod } (z-1) = (x_{20}h_{20}) = y_{20}$

(iii) $Y_3(z) = X_3(z)H_3(z) \text{ mod } (z^2+1) = (x_{31}z+x_{30})(h_{31}z+h_{30}) = y_{31}z+y_{30}$

The first two polynomial products are accomplished using one multiplication each as the reductions modulo (z-1) and (z+1) yield scalars. An algorithm for the product of the two polynomials $X_3(z)$ and $H_3(z)$ using 3 multiplications is given below. This is a general algorithm but is expressed using the notation of this example.

$$a_0 = x_{30} + x_{31} \qquad b_0 = h_{30}$$

$$a_1 = x_{31} \qquad b_1 = h_{30} + h_{31}$$

$$a_2 = x_{30} \qquad b_2 = h_{31} - h_{30}$$

$$m_k = a_k b_k \quad k=0,1,2$$

$$y_{30} = m_0 - m_1 \qquad y_{31} = m_0 + m_2$$

The quantities a_i, b_i and m_i $i=0,1,2$ are intermediate values.

The penultimate step in the construction of the algorithm is the Chinese Remainder Theorem reconstruction. The recombination polynomials for this example were derived in section 2.2.3. They were

$$S_1(z) = \frac{1}{4}(z^3 + z^2 + z + 1) \qquad (z-1) \qquad (2.28a)$$

$$S_2(z) = -\frac{1}{4}(z^3 - z^2 + z - 1) \qquad (z+1) \qquad (2.28b)$$

$$S_3(z) = -\frac{1}{2}(z^2 - 1) \qquad (z^2+1) \qquad (2.28c)$$

The final algorithm is given by

$$\mathbb{Y}(z) = \sum_{i=1}^{3} \mathbb{S}_i(z)\mathbb{Y}_i(z) \qquad \bmod (z^4-1) \qquad (2.39)$$

These polynomial products are

$$\begin{aligned}
\mathbb{S}_1(z)\mathbb{Y}_1(z) &= \tfrac{1}{4}\, y_{10}\ (z^3 + z^2 + z + 1) && \bmod (z^4-1)\\
\mathbb{S}_2(z)\mathbb{Y}_2(z) &= -\tfrac{1}{4}\, y_{20}\ (z^3 - z^2 + z - 1) && \bmod (z^4-1)\\
\mathbb{S}_3(z)\mathbb{Y}_3(z) &= -\tfrac{1}{2}\ (z^2 - 1)(y_{31}z + y_{30}) && \bmod (z^4-1)\\
&= -\tfrac{1}{2}\ (y_{31}z^3 + y_{30}z^2 - y_{31}z - y_{30})
\end{aligned}$$

The final step is to collect terms of equal powers of z together, giving

$$\begin{aligned}
\mathbb{Y}(z) &= \tfrac{1}{4}((-2y_{31} - y_{20} + y_{10})z^3 + (-2y_{30} + y_{20} + y_{10})z^2\\
&\qquad (2y_{31} - y_{20} + y_{10})z + (2y_{30} + y_{20} + y_{10}))\\
&= y_3z^3 + y_2z^2 + y_1z + y_0
\end{aligned}$$

This completes the derivation of the 4-point cyclic convolution algorithm. This derivation can be represented in terms of matrices so that it has the general form of (1.20). The algorithm is split into two stages, pre- and post-multiplication operations. In the matrices below which represent this 4-point cyclic convolution algorithm the first row of the A and B matrices corresponds to $\mathbb{X}_1(z)\mathbb{H}_1(z)$, the second to $X_2(z)H_2(z)$ and the last three rows to the multiplications needed to find $\mathbb{X}_3(z)\mathbb{H}_3(z)$.

$$A\mathbb{x} = \begin{bmatrix} 1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 0 & 1 & 0 & -1\\ 1 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_0\\ x_1\\ x_2\\ x_3 \end{bmatrix} \qquad B\mathbb{h} = \begin{bmatrix} 1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 0 & -1 & 0\\ 1 & 1 & -1 & -1\\ -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} h_0\\ h_1\\ h_2\\ h_3 \end{bmatrix}$$

$$\mathbb{M} = (Ax \times Bh) \text{ and } \mathbb{M} = (m_0, m_1, m_2, m_3, m_4)^T$$

and the C matrix is

$$y = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 2 & -2 & 0 \\ 1 & -1 & 2 & 0 & 2 \\ 1 & 1 & -2 & 2 & 0 \\ 1 & -1 & -2 & 0 & -2 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_3 \\ m_4 \end{bmatrix}$$

This algorithm gives a cyclic convolution exactly as defined by (1.3) with N=4.

As 4 has 3 factors, including 1 and itself, Winograd's theorem gives the minimum number of multiplication for a 4-point convolution as 5. Thus, in this case we have designed an algorithm with the minimum number of multiplications. In general it is not possible to attain the minimum number of multiplications due to difficulties in writing minimal polynomial product algorithms.

2.4.2 The Transpose System

Consider the general canonical form of these minimal algorithms,

$$\mathbf{Y} = C\ (\ \mathbf{A}\mathbf{x}\ \mathrm{x}\ \mathbf{B}\mathbf{h}\ ) \qquad (1.20)$$

When a minimal algorithm is constructed using the Chinese Remainder Theorem the A and B matrices correspond to the reductions into each of the moduli $C_i(z)$ and the first part of the polynomial product algorithms. The C matrix corresponds to the latter part of the polynomial products and to the CRT reconstruction. Whilst the CRT reconstruction can be regarded as the inverse of the reduction process it is, in general, more complex. Consequently the entries in the C matrix are not as simple as those in the A and B matrices. This can be seen in the matrices for the 4-point convolution. Here the C matrix contains twos but the A and B matrices do not.

In many digital filtering applications one of the sequences, h is fixed and the matrix product Bh may be precalculated. It would be most desirable if the C and B matrices could be interchanged. Winograd [2.7] introduced the 'Transpose System' to accomplish this. The same idea is described by Nussbaumer as the 'Matrix Exchange Algorithm', see [2.3].

The form (1.20) can be described by

$$y_n = \sum_{p=0}^{N-1} \sum_{q=0}^{N-1} h_p x_q \left\{ \sum_{k=0}^{M-1} C_{n,k} A_{k,q} B_{k,p} \right\} \qquad (1.24)$$

We have already noted that a necessary and sufficient condition for (1.24) to represent cyclic convolution is

$$T = \sum_{k=0}^{M-1} C_{n,k} A_{k,q} B_{k,p} \quad \begin{aligned} &= 1 \text{ if } p = n-q \text{ Mod } N \\ &= 0 \text{ otherwise} \end{aligned} \qquad (1.25)$$

The $B_{k,p}$ matrix is now replaced by a matrix $B'_{k,p}$ which has the elements $C_{N-p,k}$, and the $C_{n,k}$ matrix is replaced by $C'_{n,k}$ with elements $B_{k,N-n}$. The summation is now,

$$T' = \sum_{k=0}^{M-1} C_{N-p,k} A_{k,q} B_{k,N-n} \qquad (2.41)$$

Comparison between T' and T (1.25) shows that the subscripts of (1.25) are replaced by

$$n \rightarrow (N-p) \qquad \text{and} \qquad p \rightarrow (N-n) \qquad (2.42)$$

substitution of these new variables in the equality

$$p = n-q \mod N$$

gives $\quad (N-n) = (N-p)-q \mod N$

i.e. $\quad p = n-q \mod N$

Thus the transposed summation T' still represents a cyclic convolution provided that

$$T = 1 \text{ if } p = n-q \mod N$$

and $\quad = 0$ otherwise.

So we have developed a method of exchanging the elements of the B and C matrices. The procedure for this exchange may be summarised as follows,

(i) Exchange and transpose the B and C matrices.

(ii) Leaving the first column of the new B matrix (i.e. C^t) in place the remaining columns should be reversed.

(iii) Leaving the first row of the new C matrix (i.e. B^t) in place the remaining rows should be reversed.

In the case of the 4-point convolution algorithm the transpose system is,

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} \qquad B = \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 2 & -2 & -2 & 2 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & 2 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 1 & 1 & 1 & -1 \\ 1 & -1 & 0 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & 0 & 1 & 1 \end{bmatrix} \qquad (2.43)$$

The transpose system can be extended to two or more dimensions for cyclic convolutions by the use of suitable indexing. The idea is also applicable to cyclic correlations. For the cyclic correlation case the condition (1.25) becomes

$$T = 1 \quad \text{if} \quad n=p+q$$
$$= 0 \quad \text{otherwise} \qquad (2.44)$$

Table 2.2 gives the number of multiplications needed for various short convolution algorithms, and the minimum number of additions. The figures are for real data.

Convolution Length	No. of Multiplications	No. of Additions
2	2	4
3	4	11
4	5	15
5	10	31
6	8	34
7	16	70
8	14	46
9	22	98
16	41	141
16	35	155

Table 2.2

There are three sources of 'small N' convolution algorithms in the literature. Winograd [2.7] gives cyclic correlation algorithms for 2,3,4,5 and 6 points. All Winograds algorithms are of the form of (1.20) and have A and C matrices containing only +1, -1 and 0. Agarwal and Cooley [2.2] give algorithms for 2,3,4,5,6,7,8 and 9 points. Again all are of the form of (1.20) but with matrices containing other values besides +1,-1 and 0. Nussbaumer [2.3] gives algorithms for 2,3,4,5,7,8 and 9 point convolutions. All his algorithms are of the general form of (1.20). His 8-point algorithm has the +1, -1 and 0 form and his 9-point algorithm may be altered, at the expense of 3 additional multiplications so that it also has A and C matrices containing only +1, -1 and 0. Nussbaumer states that it is possible to derive a variety of 16-point algorithms with different numbers of multiplications. The 16-point cyclic convolution algorithm given in Appendix I uses 41 multiplications. So in Table 2.2 above all the algorithms have A and C with entries restricted to +1, -1 and 0 except for the 7-point and 35 multiplication 16-point algorithms.

2.4.3 The Application of Short Convolution Algorithms to DFTs

Chapter 1 showed how Rader's theorem could be applied to express DFTs of p or p^r points, p prime, as convolutions. Consequently it is possible to apply Winograd's short convolution algorithms to the calculation of short length DFTs. As an example a 5-point DFT algorithm is found from the 4-point convolution algorithm which has just been derived.

The 5-point DFT can be written as

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & W^1 & W^2 & W^3 & W^4 \\ 1 & W^2 & W^4 & W^1 & W^3 \\ 1 & W^3 & W^1 & W^4 & W^2 \\ 1 & W^4 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \qquad W^5 = 1 \qquad (2.46)$$

Then the non-unity multiplications are arranged, using the mappings given in (1.28), to form the convolution

$$\begin{bmatrix} \overline{X}_1 \\ \overline{X}_2 \\ \overline{X}_4 \\ \overline{X}_3 \end{bmatrix} = \begin{bmatrix} W^1 & W^3 & W^4 & W^2 \\ W^2 & W^1 & W^3 & W^4 \\ W^4 & W^2 & W^1 & W^3 \\ W^3 & W^4 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \\ x_4 \\ x_2 \end{bmatrix} \qquad (2.47)$$

where $\overline{X}_i = X_i - x_0$. It is, however, better to calculate $X_i - X_0$ $i = 1,2,3,4$ and having found X_0 add it to each term to find the X_is. $\overline{X}_i - \overline{X}_0 = X_i - X_0$ and

$$\begin{bmatrix} X_1 - X_0 \\ X_2 - X_0 \\ X_4 - X_0 \\ X_3 - X_0 \end{bmatrix} = \begin{bmatrix} W^1-1 & W^3-1 & W^4-1 & W^2-1 \\ W^2-1 & W^1-1 & W^3-1 & W^4-1 \\ W^4-1 & W^2-1 & W^1-1 & W^3-1 \\ W^3-1 & W^4-1 & W^2-1 & W^1-1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \\ x_4 \\ x_2 \end{bmatrix} \qquad (2.48)$$

Notice this does not disrupt the form of the convolution. The 4-point cyclic convolution algorithm may now be applied to (2.48). The 5-point DFT algorithm may now be calculated with 6 multiplications. The 5-point DFT A matrix is

$$\text{A matrix} \qquad \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & 1 \\ 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 & -1 \end{bmatrix} \qquad (2.49a)$$

The top row of this matrix calculates X_0. The bottom right-hand block of 5 rows and 4 columns is the 4-point cyclic convolution, transpose system, A matrix with it's columns reordered 1,3,4,2, i.e. the order of the input sequence in (2.49).

The B matrix is replaced by a list of 6 precalculated coefficients, they represent the product of the transpose system B matrix and the series W^1-1, W^3-1, W^4-1, W^2-1 . The first coefficient in this list preserves the value X_0. These coefficients are,

$$
\begin{aligned}
M_0 &= 1 \\
M_1 &= \tfrac{1}{4}(W^1+W^2+W^3+W^4) = \tfrac{1}{2}(\cos(u) + \cos(2u))-1 \\
M_2 &= \tfrac{1}{4}(W^1-W^2+W^4-W^3) = \tfrac{1}{2}(\cos(u) - \cos(2u)) \\
M_3 &= \tfrac{1}{2}(W^1-W^2-W^4+W^3) = j\ (\sin(u) - \sin(2u)) \\
M_4 &= \tfrac{1}{2}(-W^1+W^4) = -j\sin(u) \\
M_5 &= \tfrac{1}{2}(-W^2+W^3) = -j\sin(2u) \qquad u = 2\pi/5 \qquad (2.49b)
\end{aligned}
$$

Finally the C matrix for this 5-point DFT algorithm is

$$
\text{C matrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & 0 & -1 & -1 \\ 1 & 1 & -1 & 0 & 1 & 1 \\ 1 & 1 & 1 & -1 & -1 & 1 \end{pmatrix} \qquad (2.49c)
$$

In this matrix the top row preserves X_0 and the first column represents the addition of X_0 to each of the X_i-X_0 $i=1,2,3,4$. The other terms are the 4-point cyclic convolution transpose system C matrix (2.43) with it's rows rearranged in the order 1,2,4,3, i.e. the order of the output sequence in (2.48).

This completes the derivation of a 5-point DFT algorithm. Note that this algorithm is of the general form of (1.20) and that it's A and C matrices contain only +1, -1 and 0. Furthermore note that the coefficients are either real or imaginary numbers, never general complex numbers. This is a general property of DFT algorithms derived in this manner. A proof is given by Winograd in [2.7]. The essence of the proof is to show that the coefficients are always either of the form $(W^k + W^{-k})$ or $(W^k - W^{-k})$.

Table 2.3 gives the number of general multiplications for various 'short N' DFT algorithms. Except for the 9, 11 and 13-point DFTS all the algorithms are due to Winograd [2.7]. All of these algorithms are listed in Appendix I. All the algorithms are of the general form of (1.20) and have A and C matrices which contain only +1, -1 and 0. The 9-point DFT algorithm, taken from Nussbaumer [2.3], has been modified at the expense of one additional multiplication so that it's matrices have the required form. Table 2.3 gives the number of multiplications and additions for the algorithms for real only input data.

Transform Length	No. of Multiplications	No. of Additions
2	2	2
3	3	6
4	4	8
5	6	17
7	9	36
8	8	26
9	11	45
11	21	84
13	21	106
16	18	74

Table 2.3

2.5 Summary

This chapter has been devoted to the derivation of short convolution and DFT algorithms. Firstly a cyclic convolution was shown to be equivalent to the product of two polynomials modulo z^N-1. As z^N-1 can be factored by Cyclotomic polynomials, $C_k(z)$, over the rationals, the Chinese remainder theorem for polynomials can be used to reconstruct the product modulo z^N-1 from a series of products modulo $C_k(z)$. Winograd has shown that convolutions constructed in this manner use the minimum number of multiplications possible. The minimum number of 'general' multiplications for a N-point cyclic convolution is 2N-d, where d is the number of factors of N including 1 and itself.

These cyclic convolution algorithms are then applied to the DFT using Rader's theorem which was developed in chapter 1.

A key point to note is that all these algorithms, both for convolutions and DFTs, have the general form of

$$\mathbb{Y} = C\,(\,\mathbb{A}\mathbb{x} \times \mathbb{B}\mathbb{h}\,). \tag{1.20}$$

Furthermore these algorithms can be derived so that their $\mathbb{A}$ and C matrices only contain +1, -1 and 0.

Chapter 3

Polynomial Transforms

The techniques discussed in chapters One and Two dealt mainly with one-dimensional convolution and DFT algorithms, particularly those algorithms based upon the work of Winograd and Rader. This chapter is devoted to the derivation of two-dimensional convolution algorithms. These two-dimensional convolution algorithms fall into two categories. The first and more important group are those algorithms derived by Nussbaumer [3.1] using 'Polynomial Transforms'. Nussbaumer's derivation may be viewed as an extension of Winograd's method for deriving one-dimensional convolution algorithms. Nussbaumer's algorithms have the general form of equation (1.20). The second set of two-dimensional convolution algorithms, whilst still based upon polynomial transforms, do not have the general form of (1.20).

This chapter starts by showing that two-dimensional convolutions may be represented in terms of polynomials.

3.1 Two-Dimensional Convolutions Expressed using polynomials

The non-cyclic convolution y_{n_1,n_2} of the arrays x_{k_1,k_2} (N_1xN_2 points) and the array h_{k_1,k_2} (L_1xL_2 points) is defined by

$$y_{n_1,n_2} = \sum_{k_1=0}^{L_1-1} \sum_{k_2=0}^{L_2-1} h_{k_1,k_2} x_{n_1-k_1,n_2-k_2} \qquad (1.2)$$

where $n_1 = 0,1,\ldots,N_1+L_1-2$ and $n_2 = 0,1,\ldots,N_2+L_2-2$ with $x_{n_1-k_1,n_2-k_2}=0$ if $k_1>n_1$ or $k_2>n_2$. Then in a similar manner to the previous chapter, the two arrays x and h are assigned to be the coefficients of the two-dimensional polynomials $X(u,v)$ and $H(u,v)$.

$$\mathbb{X}(u,v) = \sum_{l_1=0}^{N_1-1} \sum_{l_2=0}^{N_2-1} x_{l_1,l_2} u^{l_1} v^{l_2}$$

$$\mathbb{H}(u,v) = \sum_{k_1=0}^{L_1-1} \sum_{k_2=0}^{L_2-1} h_{k_1,k_2} u^{k_1} v^{k_2} \tag{3.1}$$

So $\mathbb{H}(u,v)$ and $\mathbb{X}(u,v)$ are polynomials of degree (N_1-1) and (L_1-1) in u and of degree (N_2-1) and (L_2-1) in v. If the polynomials $\mathbb{X}(u,v)$ and $\mathbb{H}(u,v)$ are multiplied together the resulting polynomial will have the form

$$\mathbb{Y}(u,v) = \sum_{n_1=0}^{N_1+L_1-2} \sum_{n_2=0}^{N_2+L_2-2} y_{n_1,n_2} u^{n_1} v^{n_2} \tag{3.2}$$

Each coefficient y_{n_1,n_2} of (u,v) is found by summing all the products of h_{k_1,k_2} and x_{l_1,l_2} such that $n_1=l_1+k_1$ and $n_2=l_2+k_2$. Substituting $l_1=n_1-k_1$ and $l_2=n_2-k_2$ gives (1.2). Thus a two dimensional convolution may be written as a polynomial product.

As the degree of $\mathbb{Y}(u,v)$ is of (N_1+L_1-2) in u and (N_2+L_2-2) in v, (3.2) may be replaced by

$$\mathbb{Y}(u,v) = \mathbb{H}(u,v)\mathbb{X}(u,v) \bmod f(u) \bmod g(v) \tag{3.3}$$

Provided that degree $[f(u)]=d_1>N_1+L_1-2$ and degree $[g(u)]=d_2>N_2+L_2-2$. For cyclic convolutions of two NxM point sequences it is necessary that $f(u)=u^N-1$ and $g(v)=u^M-1$.

3.1.1 Convolutions of Polynomials

Equation (3.2) shows that a two-dimensional convolution is equivalent to a two dimensional polynomial product. Nussbaumer and Quandalle [3.2] show that a two-dimensional convolution is equivalent to a one-dimensional convolution of polynomials. Consider a two-dimensional cyclic convolution of NxN points,

$$y_{u,l} = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} h_{n,m} x_{u-n,l-m} \qquad u,l=0,1,\ldots,N-1 \tag{3.4}$$

The subscripts in (3.4) are evaluated modulo N. Then expressing this as as polynomial product,

$$Y_l(z) = \sum_{m=0}^{N-1} H_m(z)X_{l-m}(z) \bmod (z^N-1) \qquad (3.5)$$

where

$$H_m(z) = \sum_{n=0}^{N-1} h_{n,m}z^n \qquad m = 0,1,\ldots,N-1$$

$$X_r(z) = \sum_{s=C}^{N-1} x_{s,r}z^s \qquad r = 0,1,\ldots,N-1 \qquad (3.6)$$

and $y_{u,l}$ is obtained from the N polynomials by taking the coefficients of z^u in $Y_l(z)$.

$$Y_l(z) = \sum_{u=0}^{N-1} y_{u,l}z^u \qquad r = 0,1,\ldots,N-1 \qquad (3.7)$$

For convenience this example is a two-dimensional cyclic convolution, the method could be extended to non-cyclic convolutions by changing the modulus in which (3.5) is evaluated.

3.2 Two-Dimensional Convolution Algorithms using the CRT

Since the two-dimensional cyclic convolution (3.4) is expressed by $Y_l(z)$ modulo (z^N-1) in (3.5), $Y_l(z)$ may be computed by reducing $H_m(z)$ and $X_r(z)$ modulo each of the cyclotomic factors of (z^N-1), computing the polynomial convolutions in each of the moduli and finally reconstructing $Y_l(z)$ using the Chinese Remainder Theorem. This is analogous to Winograd's method of the previous chapter.

The difficulty with this method is the calculation of the polynomial convolutions modulo cyclotomic polynomials. The calculation of two-dimensional convolutions by this method is first considered for N=p, p an odd prime.

3.2.1 A pxp Cyclic Convolution, p an odd prime

When p is an odd prime z^p-1 has two cyclotomic factors,

$$(z^p-1) = (z-1)C_p(z) \tag{3.8}$$

where $$C_p(z) = z^{p-1} + z^{p-2} + \dots + z + 1 \tag{2.35}$$

Then the polynomial convolutions, which correspond to the reductions modulo (z-1) and $C_p(z)$ are $Y_{1,l}(z) = Y_l(z)$ Mod $C_p(z)$ and $Y_{2,l}(z) = Y_l(z)$ mod (z-1) respectively. The final reconstruction using the Chinese Remainder Theorem is

$$Y_l(z) = S_1(z)Y_{1,l}(z) + S_2(z)Y_{2,l}(z) \mod (z^p-1) \tag{2.25}$$

The auxillary polynomials $S_1(z)$ and $S_2(z)$ are found from (2.26), they are

$$S_1(z) = [p-C_p(z)]/p \tag{3.9}$$

$$S_2(z) = C_p(z)/p \tag{3.10}$$

$Y_{2,l}(z)$ is found relatively easily because it is defined modulo (z-1). So it's calculation reduces to that of a single one-dimensional scalar convolution with

$$Y_{2,l}(z) = \sum_{m=0}^{p-1} H_{2,m}X_{2,l-m} \qquad l=0,1,\dots,p-1 \tag{3.11}$$

where $$H_{2,m} = \sum_{n=0}^{p-1} h_{n,m} \qquad \text{and} \qquad X_{2,r} = \sum_{s=0}^{p-1} x_{s,r} \tag{3.12}$$

The p-point one-dimensional cyclic convolution of (3.11) may be calculated by the techniques of the previous chapter.

To complete the calculation of $Y_l(z)$ we still need to find $Y_{1,l}(z)$, i.e. a polynomial convolution modulo $C_p(z)$. The technique used to calculate $Y_{1,l}(z)$ involves the use of Polynomial transforms. These are discussed in the next section.

3.2.2 Generalised Polynomial Transforms

Consider a one-dimensional cyclic convolution of order N, in a residue class polynomial ring R/f(z), where R is a ring or field. The polynomial algebra is performed mod f(z), i.e. mod $\mathbb{C}_p(z)$ in the above case, whilst the coefficients of the polynomials are taken to lie in R - the real numbers for most cases. Suppose $\mathbb{A}_i(z)$ and $\mathbb{B}_i(z)$ are one-dimensional sequences of length N whose elements are polynomials in R/f(z), then their cyclic convolution $\mathbb{D}_l(z)$ is given by

$$\mathbb{D}_l(z) = \sum_{i=0}^{N-1} \mathbb{A}_{<l-i>}(z)\mathbb{B}_i(z) \quad \text{mod } f(z) \qquad l=0,1,\ldots,N-1 \qquad (3.13)$$

The notation <.> denotes modulo N. f(z) is assumed to be a monic polynomial. Then as proposed by Nussbaumer [3.1], Nussbaumer and Quandalle [3.2,3.3] and by Arambepola and Rayner [3.8] this convolution may be calculated by DFT-like discrete transforms. Provided that p(z) is an Nth primitive root of unity in the polynomial ring, then these transforms are defined by

$$\overline{\mathbb{A}}_k(z) = \sum_{i=0}^{N-1} \mathbb{A}_i(z)\, p^{ik}(z) \quad \text{mod } f(z) \qquad (3.14a)$$

$$\overline{\mathbb{B}}_k(z) = \sum_{j=0}^{N-1} \mathbb{B}_j(z)\, p^{jk}(z) \quad \text{mod } f(z) \qquad k=0,1,\ldots,N-1 \qquad (3.14b)$$

Then multiplying term by term

$$\overline{\mathbb{D}}_k(z) = \overline{\mathbb{A}}_k(z)\overline{\mathbb{B}}_k(z) \quad \text{mod } f(z) \qquad k=0,1,\ldots,N-1 \qquad (3.15)$$

The sequence $\mathbb{D}_l(z)$ is recovered from $\overline{\mathbb{D}}_k(z)$ using the inverse transform given by

$$\mathbb{D}_l(z) = N^{-1} \sum_{k=0}^{N-1} \overline{\mathbb{D}}_k(z)\, p^{-lk}(z) \quad \text{mod } f(z) \qquad (3.16)$$

Provided that

$$\text{(i)}\quad S \equiv \sum_{k=0}^{N-1} (p(z))^{(i+j-1)k} \mod f(z) \begin{array}{ll} \equiv N \text{ if } i+j=1 & \text{mod } N \\ \equiv 0 \text{ if } i+j\neq 1 & \text{mod } N \end{array} \quad (3.17a)$$

(ii) N has an inverse in R. (3.17b)

The condition (3.17a) may be refined by noting that

$$\sum_{k=0}^{N-1} p(z)^{tk} = \frac{p(z)^{tN} - 1}{p(z)^{t} - 1} \quad (3.18)$$

Hence (3.17a) will be true provided that

$$(p(z)^{t} - 1 \not\equiv 0) \quad \text{for all } t \not\equiv 0 \mod N \quad (3.19)$$

Having defined the general case the next section returns to the problem of calculating $Y_{1,1}(z)$, a cyclic polynomial convolution of length p, modulo $C_p(z)$, p an odd prime.

3.2.3 A Polynomial Transform, length p, root z, modulo $C_p(z)$

Using the results of the previous section we now calculate $Y_{1,1}(z)$ using polynomial transforms. The simplest root for the foward polynomial transform length p mod $C_p(z)$ is z. This may be seen by noting

a) $z^N \equiv 1 \mod C_p(z)$ and $z^t \not\equiv 1 \mod C_p(z)$ for $t \not\equiv 0 \mod p$

b) $\sum_{k=0}^{p-1} z^{tk} = p \mod C_p(z)$ for $t = 0 \mod p$.

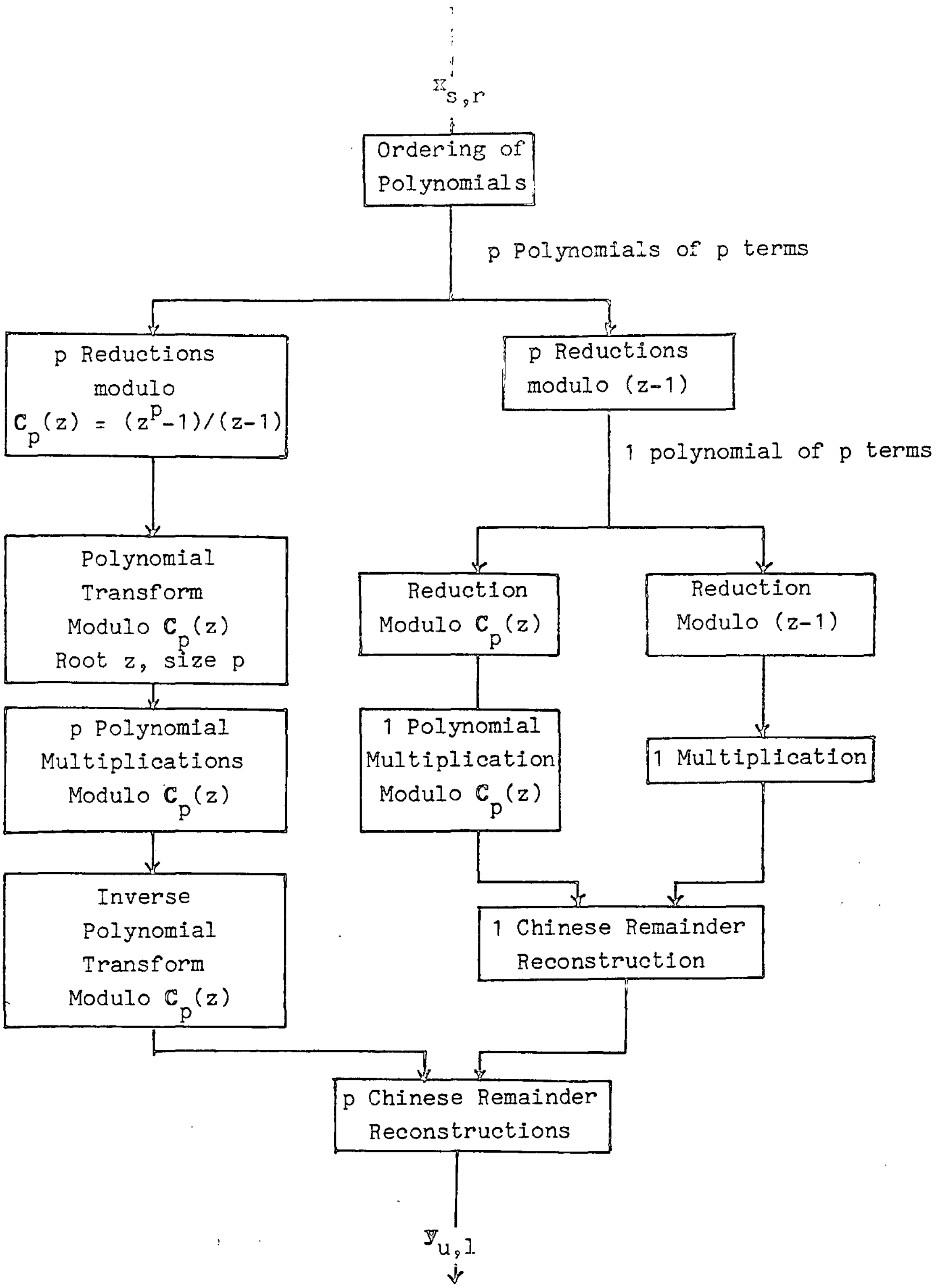
c) For $t \not\equiv 0 \mod p$, the set of exponents tk, defined mod p, is a permutation of the integers 0,1,...,p-1. Thus

$$S \equiv \sum_{k=0}^{p-1} z^{tk} = \sum_{k=0}^{p-1} z^{t} \equiv C_p(z) \equiv 0 \mod C_p(z) \quad t \not\equiv 0 \mod p$$

d) p^{-1} exists as the field of coefficients is the real numbers.

Thus all the conditions of the previous section are met and z is a suitable root for a polynomial transform of length p mod $C_p(z)$.

Using this method $Y_{1,1}(z)$ is computed with three polynomial transforms and p polynomial multiplications $\bar{H}_{1,k}(z)\bar{Y}_{1,k}(z)$ defined



$x_{s,r}$
Ordering of Polynomials
p Polynomials of p terms
p Reductions modulo $C_p(z) = (z^p-1)/(z-1)$
p Reductions modulo (z-1)
1 polynomial of p terms
Polynomial Transform Modulo $C_p(z)$ Root z, size p
Reduction Modulo $C_p(z)$
Reduction Modulo (z-1)
p Polynomial Multiplications Modulo $C_p(z)$
1 Polynomial Multiplication Modulo $C_p(z)$
1 Multiplication
Inverse Polynomial Transform Modulo $C_p(z)$
1 Chinese Remainder Reconstruction
p Chinese Remainder Reconstructions
$y_{u,l}$

Figure 3.1

A pxp point Cyclic Convolution Computed Using Polynomial Transforms

modulo $C_p(z)$. In many digital filtering applications one of the input sequences, $h_{n,m}$, is fixed and its polynomial transform $\overline{H}_{1,k}(z)$ can be precomputed. In this case only two polynomial transforms are required.

Figure 3.1 illustrates the computation of a two-dimensional cyclic convolution of size pxp, p an odd prime, by the use of polynomial transforms.

Using the polynomial product algorithms given by Nussbaumer [3.6] 3x3 and 5x5 cyclic convolution algorithms were derived. These algorithms have the general form of (1.20). Having applied the transpose system derived in chapter 2 to these 3x3 and 5x5 algorithms their A and C matrices contained only +1, -1 and 0. The use of the transpose system obviates the use of a technique suggested by Nussbaumer [3.6] to simplify the Chinese Remainder Theorem reconstruction.

Provided that the root of a polynomial transform is simple it may be calculated without multiplications. In particular when the root is z, or a power of z, the polynomial transform may be calculated using additions, subtractions and word-shifts between the words of the polynomial corresponding to different powers of z.

3.3 Other Applications of Polynomial Transforms

So far only one class of two-dimensional convolution algorithms has been discussed, pxp points, p an odd prime. However the derivation of polynomial transforms in section 3.2.2 was of a far more general nature. This section considers the application of polynomial transforms to other convolution sizes.

3.3.1 Polynomial Transforms with Roots in a field of Polynomials

As we have seen a two-dimensional cyclic convolution of NxN points can be represented as a polynomial convolution where all the polynomials are defined modulo (z^N-1). For the pxp point case z^N-1 has only two cyclotomic factors. In general z^N-1 has d factors, where d is the number of factors of N, including 1 and itself. When N is not prime how should the calculation proceed?

Both Nussbaumer [3.6] and Arambepola and Rayner[3.8] show that a polynomial transform of length N and root z always exists modulo $C_N(z)$, the largest cylotomic factor of z^N-1.

Thus the convolution $y_{u,l}$ of dimension NxN might be computed by ordering the input array as N polynomials of N terms which are reduced modulo $C_N(z)$ and modulo $C(z) = \prod_{i=1}^{d-1} C_i(z)$. $C_N(z)$ is the largest cyclotomic factor of z^N-1. The result $y_{u,l}$ is found using a Chinese Remainder Theorem reconstruction from the polynomial convolutions $Y_{1,l}(z)$ mod $C_N(z)$ and $Y_{2,l}(z)$ mod $C(z)$. The polynomial convolution $Y_{1,l}(z)$ mod $C_N(z)$ is found using polynomial transforms of length N, root z and N polynomial products modulo $C_N(z)$, which, as noted above, always exist. There are two possible ways of calculating $Y_{2,l}(z)$.

The first method is to reduce $Y_{2,l}(z)$ modulo the various cyclotomic factors of C(z), d-1 of them, and define the corresponding polynomial transforms, when they exist, to calculate each of these further polynomial convolutions. Unfortunately polynomial transforms of length N and root z will not exist in these other moduli. Some transforms with simple roots exist, this approach is discussed further in section 3.3.3.

The second possibility for the calculation of $Y_{2,l}(z)$ is to consider $Y_{2,l}(z)$ as a two-dimensional polynomial product modulo

$C(z), z^N-1$. This approach is illustrated for the case of a convolution of $p^2 x p^2$ points, p an odd prime.

3.3.2 A Cyclic Convolution of $p^2 x p^2$ points, p an odd prime

The cyclotomic factorization of $z^{p^2}-1$ is given by

$$z^{p^2}-1 = \mathbb{C}_{p^2}(z) \cdot C_p(z) \cdot C_1(z) \qquad (3.20)$$

where $$\mathbb{C}_{p^2}(z) = z^{p(p-1)} + z^{p(p-2)} + \ldots + z^p + 1 \qquad (3.21)$$

and $$C_p(z) \cdot C_1(z) = z^p - 1 \qquad (3.22)$$

Equations (3.21) and (3.22) may be deduced from equations (2.35) and (2.36) in the list of properties of cyclotomic polynomials.

Then proceeding as outlined in the previous section, $\mathbb{Y}_{1,1}(z)$ is computed by polynomial transforms of length p^2 and root z defined modulo $\mathbb{C}_{p^2}(z)$, while $\mathbb{Y}_{2,1}(z)$ is a convolution of size $p^2 x p$. This second convolution can be viewed as a polynomial convolution of length p on polynomials of p^2 terms. It, in turn, may be evaluated as a polynomial convolution of length p defined modulo $C_{p^2}(z)$ and a convolution of pxp points. The length p convolution defined modulo $\mathbb{C}_{p^2}(z)$ can be calculated by polynomial transforms of length p with root z^p. The pxp point convolution evaluation using polynomial transforms has been discussed above. This whole procedure is illustrated in figure 3.2.

A 9x9 cyclic convolution algorithm was derived using this method. A FORTRAN listing of this derivation is given in Appendix II. This algorithm, which uses 229 multiplications, yields A and C matrices which contain only +1, -1 and 0 after the application of the transpose system.

The $p^2 x p^2$ example is the simplest case of a more general algorithm for $p^c x p^c$ points, p an odd prime. The $p^c x p^c$ case was originally mentioned by Nussbaumer and Quandalle [3.2] but explained in

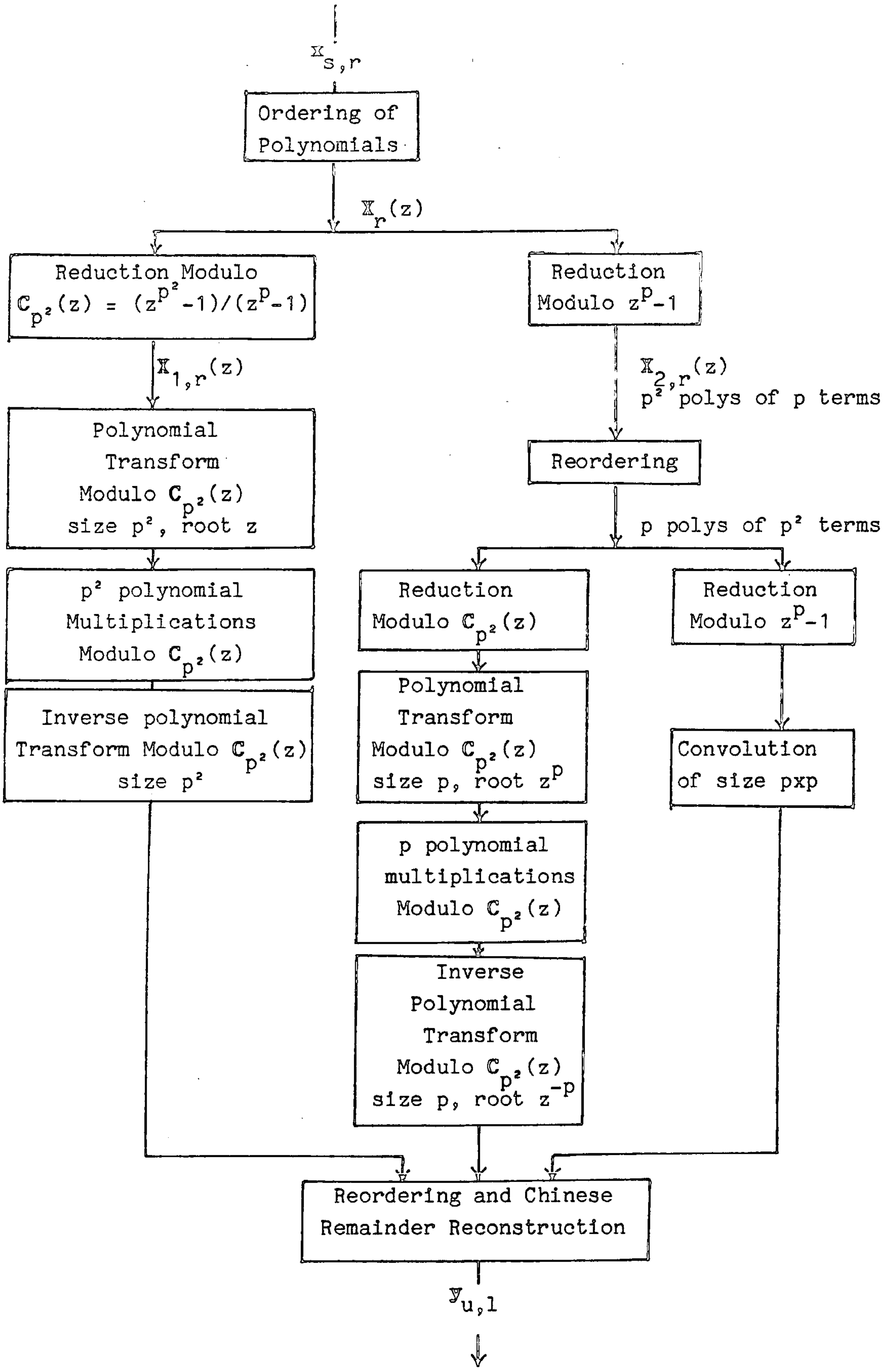

$x_{s,r}$
Ordering of Polynomials
$X_r(z)$
Reduction Modulo $C_{p^2}(z) = (z^{p^2}-1)/(z^p-1)$
Reduction Modulo z^p-1
$X_{1,r}(z)$
$X_{2,r}(z)$
p^2 polys of p terms
Polynomial Transform Modulo $C_{p^2}(z)$ size p^2, root z
Reordering
p polys of p^2 terms
p^2 polynomial Multiplications Modulo $C_{p^2}(z)$
Reduction Modulo $C_{p^2}(z)$
Reduction Modulo z^p-1
Inverse polynomial Transform Modulo $C_{p^2}(z)$ size p^2
Polynomial Transform Modulo $C_{p^2}(z)$ size p, root z^p
Convolution of size pxp
p polynomial multiplications Modulo $C_{p^2}(z)$
Inverse Polynomial Transform Modulo $C_{p^2}(z)$ size p, root z^{-p}
Reordering and Chinese Remainder Reconstruction
$y_{u,l}$

Figure 3.2
Computation of Convolution of $p^2 x p^2$ points

more detail by Nussbaumer in [3.6].

The case of convolutions of size $2^t x 2^t$ is very similar to the $p^c x p^c$ case. The cyclotomic factorization of $z^{2^t}-1$ is simpler because

$$\mathbb{C}_{2^t}(z) = z^{2^{t-1}}+1 \qquad (3.23)$$

This gives $z^{2^t}-1$, t-1 cyclotomic factors of the general form of (3.23).

3.3.3 Polynomial Transforms with Composite Roots

The derivation of polynomial transforms in section 3.2.2 relied on the existance of a primitive Nth root p(z) defined in the ring of polynomials. Nussbaumer and Quandalle [3.2] suggested a way of extending the size of polynomial transforms by taking advantage of the field of coefficients. If a N_1-point polynomial transform supports cyclic convolution mod f(z) with root p(z), it is possible to use roots of unity of order N_2 in the field of coefficients for the definition of transforms of length N_1N_2 which also have the cyclic convolution property.

Suppose the field of coefficients is the complex numbers, so DFTs of length N_2, root $W= \exp(-2\pi j/N_2)$ are defined which support cyclic convolution. Then, if $(N_1,N_2)=1$ the polynomial transform of root Wp(z) defined modulo f(z) supports a cyclic convolution of length $N=N_1N_2$.

This is verified by considering the conditions given in section 3.2.2. Firstly, since $W^{N_2} = 1$ and $p(z)^{N_1} \equiv 1 \bmod f(z)$,

$$[Wp(z)]^N = (W^{N_2})^{N_1}[p(z)^{N_1}]^{N_2} \equiv 1 \mod f(z) \qquad (3.24)$$

Condition (3.17b) that N has an inverse, is met because N_1 and N_2 both have inverses. We now consider condition (3.17a),

$$S \equiv \sum_{k=0}^{N-1} [Wp(z)]^{tk} \mod f(z) \qquad (3.25)$$

As the exponents tk are defined modulo N, $S \equiv N$ for $t \equiv 0$ modulo N. For $t \not\equiv 0$ mod N, S may be mapped into a two-dimensional summation because N_1 and N_2 are mutually prime.

$$k \equiv N_1 k_2 + N_2 k_1 \quad \text{Mod } N$$
$$k_1 = 0,1,\ldots,N_1-1 \qquad k_2 = 0,1,\ldots,N_2-1 \tag{3.26}$$

giving

$$S = \sum_{k_2=0}^{N_2-1} W^{tN_1k_2} \sum_{k_1=0}^{N_1-1} p(z)^{tN_2k_1} \quad \text{mod } f(z) \tag{3.27}$$

The existence of the two transforms of length N_1 and N_2 with roots p(z) and W implies that $S \equiv 0$ for $k_1 \not\equiv 0$ mod N_1 and $k_2 \not\equiv 0$ mod N_2. Therefore $S \equiv 0$ for $k \not\equiv 0$ mod N. Thus (3.17a) is satisfied. So polynomial transforms with composite roots may be defined.

The condition that $(N_1,N_2)=1$ means that this method is of most value when N_1 is odd and $N_2=2$ or 4. In these cases $W=-1$ or j, $j=\sqrt{-1}$. So if the N_1-point polynomial transform has a simple root then the $2N_1$-point and $4N_1$-point polynomial transforms will also have simple roots.

A 6x6 cyclic convolution algorithm was derived using a composite polynomial transform of length 6, root -z, mod $C_3(z)=z^2+z+1$. Utilising the transpose system the algorithm was of the general form of (1.20) with the A and C matrices containing only +1, -1 and 0.

Table 3.1, taken from Nussbaumer [3.6] lists all the polynomial transforms that can be calculated without multiplications together with their associated rings.

Transform ring	Transform		Size of convolution
	Length	Root	
$(z^p-1)/(z-1)$	p	z	pxp
$(z^p-1)/(z-1)$	$2p$	$-z$	$2pxp$
$(z^{2p}-1)/(z^2-1)$	$2p$	$-z^{p+1}$	$2px2p$
$(z^{p^2}-1)/(z^p-1)$	p	z^p	pxp^2
$(z^{p^2}-1)/(z^p-1)$	p^2	z	p^2xp^2
$(z^{2p^2}-1)/(z^{2p}-1)$	$2p^2$	$-z^{p^2+1}$	$2p^2x2p^2$
$(z^{p_1p_2}-1)/(z^{p_2}-1)$	p_1	z^{p_2}	$p_1xp_1p_2$
$z^{2^{t-1}}+1$	2^t	z	2^tx2^t

Table 3.1 p, p_1 and p_2 odd primes.

Multiplication free polynomial transforms

3.4 Operation Counts

Table 3.2, extracted from Nussbaumer's work [3.2,3.6], gives the number of multiplications and additions for a variety of convolution sizes. The number of additions is the minimum required to evaluate the algorithm. It is not equivalent to the number used when performing the matrix operations for the canonical form (1.20).

Convolution Size	Number of multiplications	Number of Additions	Mults. per point	Adds. per point
3x3	13	70	1.44	7.78
4x4	22	122	1.38	7.62
5x5	55	369	2.20	14.76
6x6	52	424	1.44	11.78
7x7	121	1163	2.47	23.73
8x8	130	750	2.03	11.72
9x9	193	1382	2.38	17.06
10x10	220	1876	2.20	18.76
14x14	484	5436	2.47	27.73
16x16	634	4774	2.48	18.65
18x18	772	6576	2.38	20.30
30x30	2860	31088	3.18	34.54
32x32	3658	24854	3.57	24.27
64x64	17770	142902	4.34	34.89
128x128	78250	720502	4.78	43.98

Table 3.2
Operation count for two dimensional convolutions evaluated using polynomial transforms

3.5 Other Approaches

As mentioned at the beginning of the chapter the approach adopted by Nussbaumer and Quandalle is not the only possible technique for applying applying polynomial transforms to the calculation of cyclic convolutions. Here three other schemes are considered.

3.5.1 Fast Biased Polynomial Transforms

Pei and Wu [3.12] proposed a method of performing two-dimensional cyclic convolutions using biased polynomial transforms in the ring z^N-1 itself. This technique eliminates the reductions into the various moduli and the Chinese Remainder Theorem recombinations.

The derivation of these biased polynomial transforms is the same as that for the general polynomial transform given in section 3.2.2, i.e, equations (3.13-3.16) inclusive with $f(z)=z^N-1$ and root $p(z)=z$.

The biased polynomial transforms of the two polynomial sequences $A_i(z)$ and $B_i(z)$ are

$$\bar{A}_k(z) = \sum_{i=0}^{N-1} A_i(z)z^{ik} \quad \text{mod } (z^N-1)$$

$$\bar{B}_k(z) = \sum_{j=0}^{N-1} B_j(z)z^{jk} \quad \text{mod } (z^N-1) \quad k=0,1,\ldots,N-1 \qquad (3.28)$$

Multiplying term by term and performing the inverse biased polynomial transform yields the sequence $C'_l(z)$

$$C'_l(z) = \frac{1}{N}\sum_{k=0}^{N-1} \bar{A}_k(z)\bar{B}_k(z)z^{-lk} \quad \text{mod } (z^N-1) \quad k=0,1,\ldots,N-1 \quad (3.29)$$

giving

$$C'_l(z) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} A_i(z)B_j(z)\,\frac{1}{N}\sum_{k=0}^{N-1} z^{(i+j-l)k} \quad \text{mod } (z^N-1) \quad (3.30)$$

There are now several possible cases depending upon N. Just one case, N an odd prime, is considered.

For $t \equiv (i+j-1) \equiv 0 \bmod N$

$$\sum_{k=0}^{N-1} z^{tk} = N \qquad (3.31)$$

For $t \equiv (i+j-1) \not\equiv 0 \bmod N$, the set of exponents $tk \bmod N$ is a permutation of the integers $0, 1, \ldots, N-1$ and

$$\sum_{k=0}^{N-1} z^{tk} = \sum_{k=0}^{N-1} z^{k} = 1 + z + \ldots + z^{N-1} \mod (z^N-1) \qquad (3.32)$$

Then

$$C_l'(z) = \underbrace{\sum_{\substack{n=0 \\ n+m-l\equiv 0 \bmod N}}^{N-1} B_n(z) A_{<l-n>}(z)}_{\text{convolution term}} + \underbrace{\sum_{m=0}^{N-1} \sum_{\substack{n=0 \\ n+m-l\not\equiv 0 \bmod N}}^{N-1} B_n(z) A_m(z)(1+z+\ldots+z^{N-1})}_{\text{biased term}} \qquad (3.33)$$

The righthand term of (3.33) forms a constant bias to each term of the final results. For N an odd prime the biases for each row of the result are always equal to a constant, the value depending upon the input data. These biases must be subtracted from (3.33) to obtain the correct convolution. Pei and Wu suggest that the biases may be calculated by extending the input sequences with zeroes so that otherwise zero terms in the convolution will be biased and easily identified.

3.5.2 Polynomial Transforms in Modified Rings

A polynomial transform of length N and root z^2 can always be defined modulo (z^N+1) but the modulus (z^N+1) does not support cyclic convolution. However Arambepola and Rayner [3.7, 3.8] developed a mapping to change a convolution modulo (z^N-a) into one modulo (z^N-b). A special case of this mapping is the mapping of cyclic convolutions modulo (z^N-1) into 'skew-circular' convolutions modulo (z^N+1). The skew circular convolution can then be evaluated using polynomial transforms of length N and root z^2 modulo (z^N+1) and N polynomial products modulo (z^N+1). This way of performing two-dimensional cyclic convolutions may be described as follows

$$y_{u,l} = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} h_{n,m} x_{u-n,l-m} \qquad u,l = 0,1,\ldots,N-1 \tag{3.14}$$

$$H_m(z) = \sum_{n=0}^{N-1} h_{n,m} W^n z^n \qquad W = e^{-j\pi/N} \quad j=\sqrt{-1} \tag{3.34}$$

$$X_r(z) = \sum_{s=0}^{N-1} x_{s,r} W^s z^s \qquad m,r = 0,1,\ldots,N-1 \tag{3.35}$$

Then

$$A_l(z) = \sum_{m=0}^{N-1} H_m(z) X_{l-m}(z) \quad \mathrm{mod}\ (z^N+1) \tag{3.36}$$

with

$$A_l(z) = \sum_{u=0}^{N-1} a_{u,l} z^u \qquad l=0,1,\ldots,N-1 \tag{3.37}$$

and

$$y_{u,l} = a_{u,l} W^{-u} \tag{3.38}$$

This calculation is illustrated in figure 3.3. When compared with some of the previous methods this algorithm trades computational efficiency for structural simplicity. Furthermore the multiplications in (3.34), (3.35) and (3.38) by W^n involve complex arithmetic.

Whilst offering some advantages this algorithm is not the most computationally efficient method of calculating two-dimensional convolutions and does not have the general form of (1.20).

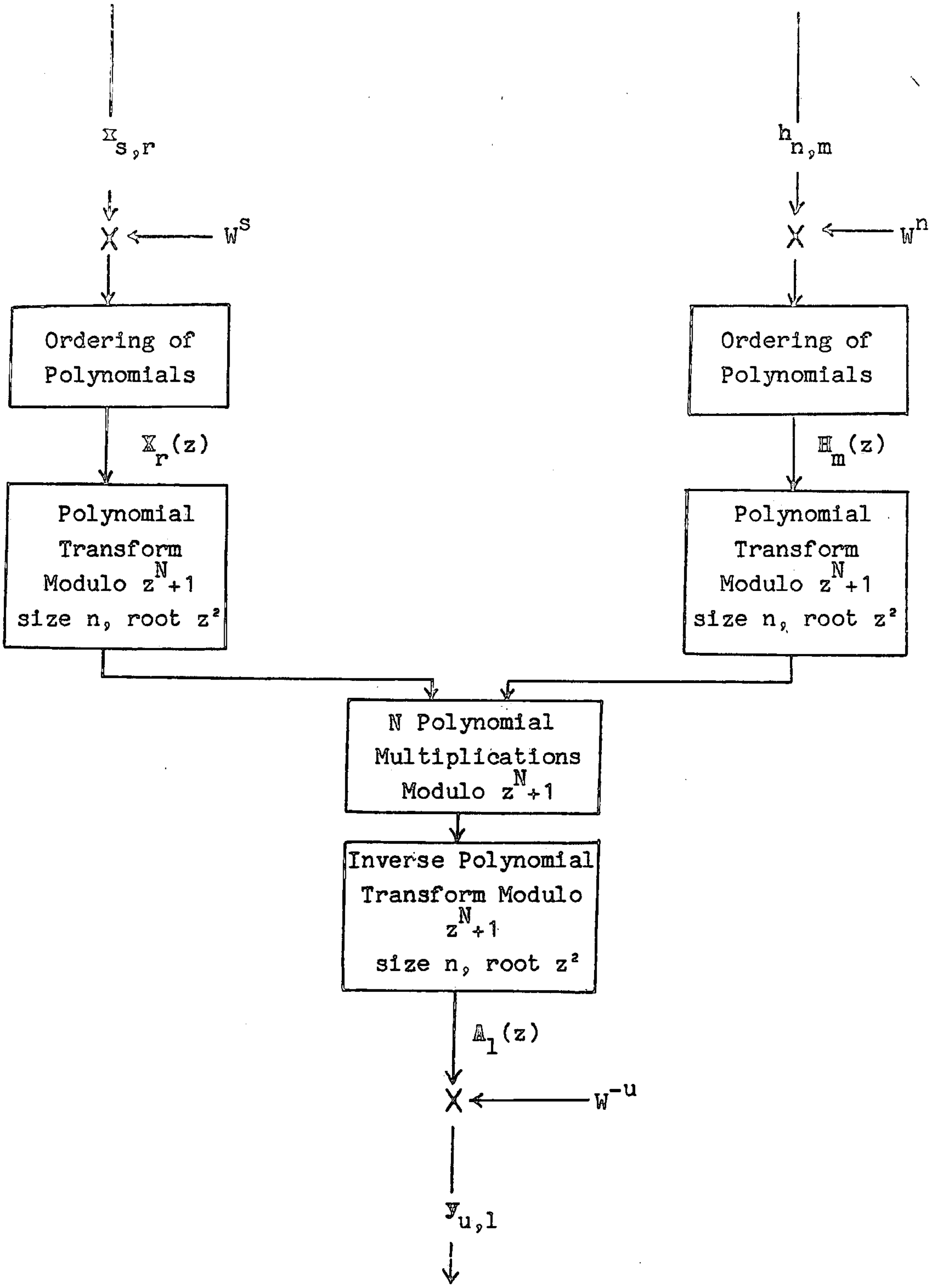

$x_{s,r}$
W^s
Ordering of Polynomials
$X_r(z)$
Polynomial Transform Modulo z^N+1 size n, root z^2
$h_{n,m}$
W^n
Ordering of Polynomials
$H_m(z)$
Polynomial Transform Modulo z^N+1 size n, root z^2
N Polynomial Multiplications Modulo z^N+1
Inverse Polynomial Transform Modulo z^N+1 size n, root z^2
$A_l(z)$
W^{-u}
$y_{u,l}$

Figure 3.3

Convolution of NxN points
Calculated by polynomial transforms
in modified rings

3.5.3 Other Two-Dimensional Cyclic Convolution Algorithms

Truong, Reed, Lipes and Wu [3.9], Reed, Shoa and Truong [3.10] suggest a procedure for calculating a cyclic convolution of $d_1 x d_2$ points with $d_2=2^m$ and $d_1=2^{m-r+1}$ for $1<r<m$. The essence of their technique is to factorize $z^{d_2}-1$, the longer dimension, into r+1 factors so that

$$z^{d_2}-1 = (z^{d_2/2}+1)(z^{d_2/2^2}+1)\ldots(z^{d_2/2^r}+1)(z^{d_2/2^r}-1) \qquad (3.39)$$

The point of this factorization is that each of the factors, except $(z^{d_2/2^r}-1)$ supports length d_1-point polynomial transforms with root $z^{2^{r-i}}$ for the ith factor. So the first r factors maybe evaluated using polynomial transforms and N polynomial products each. The polynomial convolution corresponding to the last factor $(z^{d_{2/2}^r}-1)$ is more difficult to calculate. It may be evaluated by using Arambeopla's mapping to convert it into a convolution modulo $(z^{d_{2/2}^r}+1)$ for which a suitable polynomial transform exists.

In the original paper by Truong et al. [3.9] the way of calculating the polynomial products using the FFT is inefficient. Although an improvement is suggested by Martens [3.11], cyclic convolutions calculated by this method are very inefficient when compared to Nussbaumer and Quandalle's method for the convolution of $2^t x 2^t$ points discussed at the end of section 3.3.2.

3.6 Summary of Chapter 3

This chapter has dealt with two-dimensional convolutions calculated using polynomial transforms. The method involves treating the two-dimensional convolutions as one-dimensional convolutions of polynomials. These polynomials are then reduced in each of the cyclotomic factors of z^N-1. Polynomial transforms can be used to calculate these polynomial convolutions in certain of these moduli.

The Chinese Remainder Theorem is used to reconstruct the result from each of the polynomial convolutions in the compontent moduli. It is possible to derive some of these algorithms in the general form of (1.20) with their A and C matrices containing only +1, -1 and 0. In particular algorithms for 2x2, 3x3, 4x4, 5x5, 6x6 and 9x9 point cyclic convolutions exist and have this form.

Chapter 4

Multi-dimensional Mappings

The preceeding chapters considered algorithms for specific transform and convolution lengths. This chapter deals with multi-dimensional mapping techniques which allow longer transforms and convolutions to be built up from a set of smaller ones. From these mappings the fully nested form of the Winograd Fourier Transform Algorithm (WFTA) is built up, together with the equivalent nested convolution algorithms.

Throughout the chapter the number of operations required to compute the various algorithms is given, allowing a preliminary comparison to be made between algorithms.

4.1 The DFT

Consider the calculation of a one-dimensional DFT

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} \qquad k=0,1,\ldots,N-1 \qquad (1.10)$$

where W_N is the Nth root of unity and the index nk is evaluated modulo N. A direct implementation of (1.10) for complex data would require

$$\begin{array}{ll} 4N^2 & \text{Real multiplications} \\ 2N(N-1) + 2N^2 & \text{Real additions} \end{array} \qquad (4.1)$$

assuming 4 real multiplications and 2 real additions per complex multiplication. It is possible to perform a complex multiplication in 3 real multiplications and 5 real additions [4.1]; however this is not usually done. This is because the 3 multiplication algorithm is less well suited to parallel hardware implementations.

4.2 The Fast Fourier Transform (FFT)

If N is composite, then it is possible to map a one-dimensional sequence into a multi-dimensional one. These are many such mappings but only a few have suitable properties. Perhaps the simplest set of mappings applicable to the DFT occurs when $N=N_1N_2$ and the indices m and k in (1.10) are redefined as

$$m = n_1 + N_1n_2 \qquad n_1,k_1 = 0,1,\ldots,N_1-1 \qquad (4.2)$$

$$k = N_2k_1 + k_2 \qquad n_2,k_2 = 0,1,\ldots,N_2-1 \qquad (4.3)$$

There are no restrictions on the values of N_1 and N_2. Substituting (4.2) and (4.3) into (1.10) gives

$$X_{N_2k_1+k_2} = \sum_{n_1=1}^{N_1-1} W_N^{N_2n_1k_1} W_N^{n_1k_2} \sum_{n_2=0}^{N_2-1} x_{n_1+N_1n_2} W_N^{N_1n_2k_2} \qquad (4.4)$$

Define two $N_1 x N_2$ arrays $\hat{x}_{n_1,n_2}$ and $\hat{X}_{k_1,k_2}$ as

$$\hat{x}_{n_1,n_2} = x_{N_1n_2+n_1} \quad \text{and} \quad \hat{X}_{k_1,k_2} = X_{N_2k_1+k_2} \qquad (4.5)$$

Further note that $W_N^{N_2n_1k_1} = W_{N_1}^{n_1k_1}$ and $W_N^{N_1n_2k_2} = W_{N_2}^{n_2k_2}$ then (4.4) may be written as

$$\hat{X}_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} W_{N_1}^{n_1k_1} \left[W_N^{n_1k_2} \left[\sum_{n_2=0}^{N_2-1} \hat{x}_{n_1,n_2} W_{N_2}^{n_2k_2} \right] \right] \qquad (4.6)$$

The computation of (4.6) is carried out in three stages,

i) Calculate N_1, N_2-point DFTs along the second index of $\hat{x}_{n_1,n_2}$.

ii) Multiply the result of step (i) by

$$W_N^{n_1k_2} \qquad n_1 = 0,1,\ldots,N_1-1, \quad k_2 = 0,1,\ldots,N_2-1$$

iii) Calculate N_2, N_1-point DFTs along the first index of $\hat{x}_{n_1,n_2}$.

The multiplications in step (ii) are known as 'twiddle factors'.

The method can be extended to more than two factors of N. If there are L factors of N then there will be L-1 sets of twiddle factors. The number of operations for this two factor example is given by

$$4N(N_1+N_2+1) \qquad \text{Real multiplications}$$

$$2N(2N_1+2N_2-1) \qquad \text{Real additions} \qquad (4.7)$$

This is clearly an advance over (4.1). The computational advantage is increased with a greater number of factors.

When $N_1=N_2=2$ the mappings (4.2) and (4.3) become the basis for the decimation-in-time radix-2 FFT. The Fast Fourier Transform (FFT) was first proposed by Cooley and Tukey in 1965 [4.2]. When $N_1=N_2=2$ or 4 the DFTs in stages (i) and (iii) above may be calculated without multiplications using only additions and subtractions. The operation count for complex data for the radix-2 FFT is

$$2N\mathrm{Log}_2 N \qquad \text{Real multiplications}$$

$$3N\mathrm{Log}_2 N \qquad \text{Real additions} \qquad (4.8)$$

Equation (4.8) includes one extra set of twiddle factors (all ones) to increase the regularity of the algorithm. Equation (4.8) also includes some other multiplications by unity. There are many variants of the FFT, see, for example, Brigham [4.3].

4.3 Prime Factor Mappings

The mappings (4.2) and (4.3) for n and k placed no restrictions on the values of N_1 and N_2. By constraining N_1 and N_2 to be relatively prime many other mappings are possible. Some of these possibilities are now discussed. Burrus [4.4] considers the general case of mapping a one-dimensional sequence of length $N=N_1N_2$ into a two-dimensional array that is N_1 by N_2 in size. He considers in detail the conditions for the mapping

$$n = K_1 n_1 + K_2 n_2 \mod N \qquad \begin{array}{l} n_1 = 0,1,\ldots,N_1-1 \\ n_2 = 0,1,\ldots,N_2-1 \end{array} \tag{4.9}$$

which is cyclic in N, to be cyclic in N_1 and N_2 as well as being one-to-one (unique). The conditions for this to be true when $(N_1,N_2)=1$ are that

$$K_1 = \alpha N_2 \text{ and } K_2 = \beta N_1 \quad \text{and } (\alpha,N_1) = (\beta,N_2) = 1 \tag{4.10}$$

These conditions are now applied to a $N=N_1N_2$ point DFT. Suppose the indices n and k are mapped as

$$n = \alpha N_2 n_1 + \beta N_1 n_2 \mod N \tag{4.11}$$

$$k = \gamma N_2 k_1 + \delta N_1 k_2 \mod N \tag{4.12}$$

Then substituting these mappings into (1.10) and defining

$$\bar{x}_{n_1,n_2} = x_{\alpha N_2 n_1 + \beta N_1 n_2} \quad \text{and} \quad \bar{X}_{k_1,k_2} = X_{\gamma N_2 k_1 + \delta N_1 k_2} \tag{4.13}$$

gives

$$\bar{X}_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \left[\sum_{n_2=0}^{N_2-1} \bar{x}_{n_1,n_2} W_{N_2}^{\alpha\delta N_1 n_2 k_2} \right] W_{N_1}^{\beta\gamma N_2 n_1 k_1} \tag{4.14}$$

Thus the mappings (4.11) and (4.12) completely separate the one-dimensional DFT (1.10) into a two-dimensional function. Note that the inner summation, in brackets, only involves n_1 as an index. Some possible choices of α, β, γ and δ are now considered.

4.3.1 Good's Algorithm

Good [4.5] considered using

$$\alpha = \beta = 1 \qquad \delta = N_1^{-1} \mod N_2 \quad \text{and} \quad \gamma = N_2^{-1} \mod N_1 \tag{4.15}$$

Then the mapping (4.12) becomes the Chinese Remainder Theorem. Substituting (4.15) into (4.14) gives

$$\bar{X}_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \left[\sum_{n_2=0}^{N_2-1} \bar{x}_{n_1,n_2} W_{N_2}^{n_2 k_2} \right] W_{N_1}^{n_1 k_1} \tag{4.16}$$

Equation (4.16) represents a true two-dimensional DFT which eliminates all the twiddle factors associated with (4.6). Explicitly the mappings are

$$n = N_2 n_1 + N_1 n_2 \tag{4.17}$$

$$k = N_2 \langle N_2 \rangle_{N_1}^{-1} k_1 + N_1 \langle N_1 \rangle_{N_2}^{-1} k_2 \tag{4.18}$$

Good calls the n mapping the Ruritanian mapping and the k mapping the Sino correspondance. Equation (4.16) could also have been derived by choosing $\gamma = \delta = 1$ and using the CRT form on α and β. This would have reversed the forms of (4.17) and (4.18). The mappings as given by (4.17) and (4.18) are the same as those used Kolba and Parks [4.6,4.7]. Direct computation of (4.16) requires

$$4(N_1^2 + N_2^2) \qquad \text{Real multiplications}$$

$$2[N_1(2N_1-1) + N_2(2N_2-1)] \qquad \text{Real additions} \tag{4.19}$$

Kolba and Parks [4.6] consider the use of Winograd's 'short N' algorithms for the N_1 and N_2-point transforms. Their paper contains some 'short N' algorithms derived in a manner analogous to Winograd's. If M_i and A_i are the number of multiplications and additions for a N_i-point short-N WFTA for real data, then the number of operations for an algorithm with L factors for complex data is

$$2 \sum_{i=1}^{L} \frac{N}{N_i} M_i \qquad \text{Real Multiplications}$$

$$2 \sum_{i=1}^{L} \frac{N}{N_i} A_i \qquad \text{Real Additions} \tag{4.20}$$

where $N = \prod_{i=1}^{L} N_i$

For the worst small N algorithm $M_i \simeq 1.3N_i$ (7-pt. WFTA). Then at most a two factor algorithm would require 5.2N multiplications.

Figure 4.1 illustrates a 15-point DFT calculated by this method. The use of the mappings (4.17) and (4.18) coupled with the use of Winograd's small N DFT algorithms is often refered to as the 'Prime Factor Algorithm' (PFA).

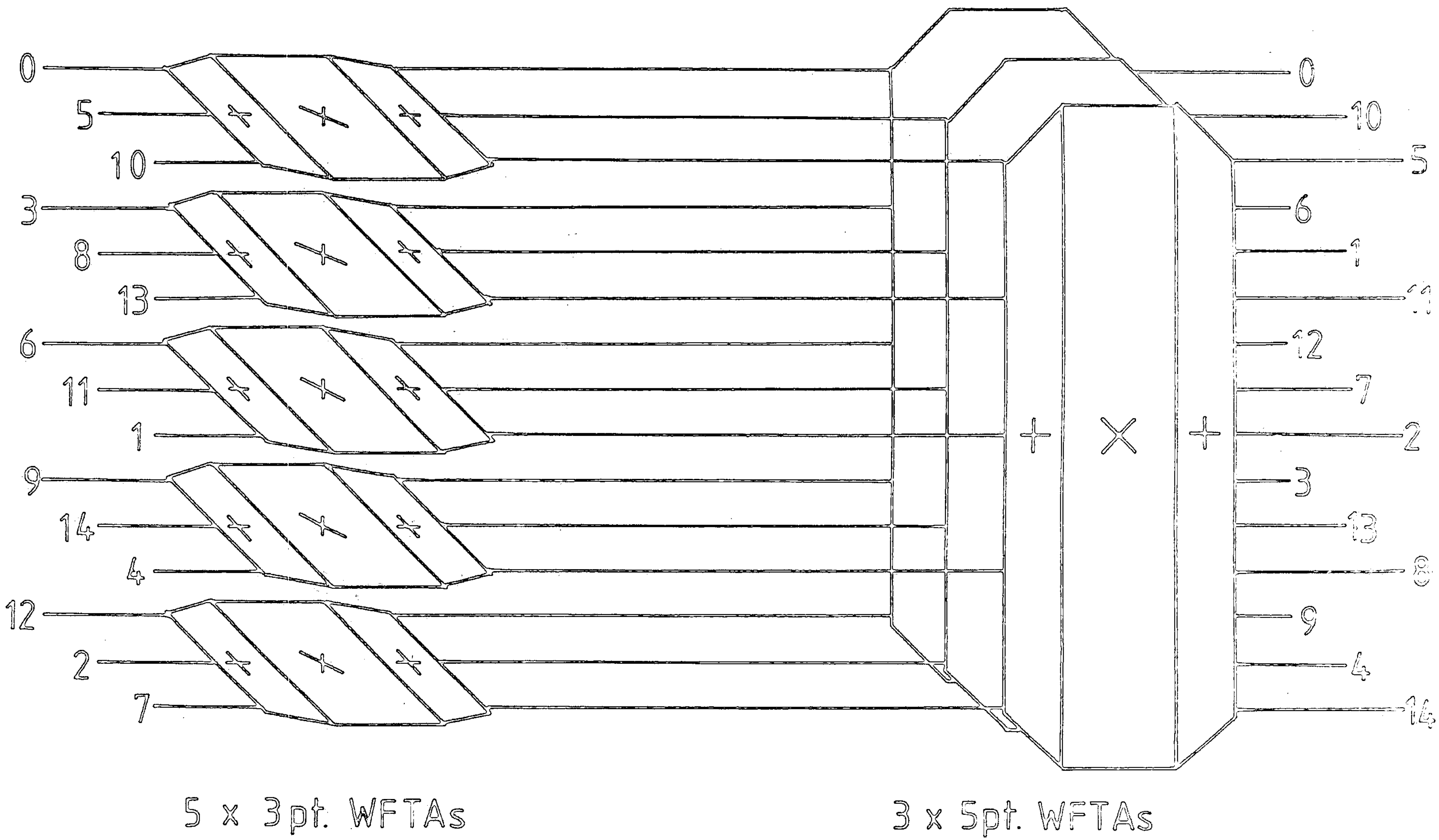


Fig 4.1 A 15-point PFA

Perhaps the most inconvenient feature of the PFA is the implementation of the data reorderings associated with (4.18). Two schemes are now introduced to ease this problem.

4.3.2 An Unscrambling Constant

Burrus and Eschenbacher [4.8] give a simple way of calculating the reordering corresponding to the Chinese Remainder Theorem mapping. The scheme is illustrated here for a two factor example. Using Good's mapping

$$n = \langle N_2 n_1 + N_1 n_2 \rangle_N \qquad (4.17)$$

$$k = \langle K_3 k_1 + K_4 k_2 \rangle_N$$

where $K_3 = \langle N_2 \langle N_2 \rangle_{N_1}^{-1} \rangle_N$ and $K_4 = \langle N_1 \langle N_1 \rangle_{N_2}^{-1} \rangle_N$ (4.18)

At the end of the calculation the location of the calculated DFT values is given by

$$n = \langle N_2 k_1 + N_1 k_2 \rangle_N \qquad (4.21)$$

corresponding to the frequency index

$$k = \langle K_3 k_1 + K_4 k_2 \rangle_N$$

But k_i=k mod N_i, so (4.21) becomes

$$n = \langle N_2 \langle k \rangle_{N_1} + N_1 \langle k \rangle_{N_2} \rangle_N \qquad (4.22)$$

or

$$n = \langle (N_1 + N_2) k \rangle_N \qquad (4.23)$$

for L factors

$$n = \left\langle \left(\sum_{i=1}^{L} \frac{N}{N_i} \right) k \right\rangle_N \qquad (4.24)$$

This gives a simple method of calculating the unscrambling of the result. This result could ease address generation problems, both in software and hardware implementations of the DFT.

4.3.3 An In-Place, In-Order PFA Algorithm

Burrus and Eschenbacher [4.8] also consider an in-order, in-place algorithm in which the n and k mappings are the same. This is done by choosing the Ruritanian correspondance for both maps, i.e.,

$$n = \langle N_2 n_1 + N_1 n_2 \rangle_N \qquad (4.17)$$

$$k = \langle N_2 k_1 + N_1 k_2 \rangle_N \qquad (4.25)$$

This is equivalent to setting $\alpha=\beta=\gamma=\delta=1$ into (4.11) and (4.12) giving

$$X_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1,n_2} W_{N_2}^{N_1 n_2 k_2} W_{N_1}^{N_2 n_1 k_1} \qquad (4.26)$$

The exponents in (4.26) are evaluated mod N_2 and N_1 respectively. Burrus and Eschenbacher point out that since N_1 and N_2 are relatively prime the operations $\langle N_1 n_2 \rangle_{N_2}$ or $\langle N_1 k_2 \rangle_{N_2}$ and $\langle N_2 n_1 \rangle_{N_1}$ or $\langle N_2 k_1 \rangle_{N_1}$ are merely permutations. So the modified DFTs required (4.26) may be obtained by reordering the inputs or outputs to each of the small N DFT modules. Burrus and Eschenbacher explicitly reorder the small N DFTs so that modules in their FORTRAN program depend upon the transform length. Rothweiler [4.9] uses the same principle but utilises a pointer to reorder the output of standard WFTA modules. Arambepola [4.10] derives a similar method to Rothweiler's for the reordering of the outputs from 'short-N' bit serial WFTA ICs.

4.4 The Kronecker Product Strucure of DFT Matrices

Consider a 10-point DFT written as matrix vector product

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \\ X_9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 & W^8 & W^9 \\ 1 & W^2 & W^4 & W^6 & W^8 & 1 & W^2 & W^4 & W^6 & W^8 \\ 1 & W^3 & W^6 & W^9 & W^2 & W^5 & W^8 & W^1 & W^4 & W^7 \\ 1 & W^4 & W^8 & W^2 & W^6 & 1 & W^4 & W^8 & W^2 & W^4 \\ 1 & W^5 & 1 & W^5 & 1 & W^5 & 1 & W^5 & 1 & W^5 \\ 1 & W^6 & W^2 & W^8 & W^4 & 1 & W^6 & W^2 & W^8 & W^4 \\ 1 & W^7 & W^4 & W^1 & W^8 & W^5 & W^2 & W^9 & W^6 & W^3 \\ 1 & W^8 & W^6 & W^4 & W^2 & 1 & W^8 & W^6 & W^4 & W^2 \\ 1 & W^9 & W^8 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} \qquad W^{10}=1 \qquad (4.27)$$

Now consider the mappings given by (4.17) and (4.18) with $N_1=2$ and $N_2=5$. Then the mappings are

$$n = 5n_1 + 2n_2 \qquad n_1 = 0,1$$
$$k = 5n_1 + 6n_2 \qquad n_2 = 0,1,2,3,4 \qquad (4.28)$$

with n_2, the innermost factor, varying most rapidly, the columns of the above matrix and the input vector, are both rearranged according to the n sequence, i.e.,

0, 2, 4, 6, 8, 5, 7, 9, 1, 3

the rows of the matrix and the output vector are rearranged by the k sequence

0, 6, 2, 8, 4, 5, 1, 7, 3, 9

this yields the following rearrangement of the 10-point DFT matrix

$$\begin{bmatrix} X_0 \\ X_6 \\ X_2 \\ X_8 \\ X_4 \\ X_5 \\ X_1 \\ X_7 \\ X_3 \\ X_9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W^2 & W^4 & W^6 & W^8 & 1 & W^2 & W^4 & W^6 & W^8 \\ 1 & W^4 & W^8 & W^2 & W^6 & 1 & W^4 & W^8 & W^2 & W^6 \\ 1 & W^6 & W^2 & W^8 & W^4 & 1 & W^6 & W^2 & W^8 & W^4 \\ 1 & W^8 & W^6 & W^4 & W^2 & 1 & W^8 & W^6 & W^4 & W^2 \\ 1 & 1 & 1 & 1 & 1 & W^5 & W^5 & W^5 & W^5 & W^5 \\ 1 & W^2 & W^4 & W^6 & W^8 & W^5 & W^7 & W^9 & W^1 & W^3 \\ 1 & W^4 & W^8 & W^2 & W^6 & W^5 & W^9 & W^3 & W^7 & W^1 \\ 1 & W^6 & W^2 & W^8 & W^4 & W^5 & W^1 & W^7 & W^3 & W^9 \\ 1 & W^8 & W^6 & W^4 & W^2 & W^5 & W^3 & W^1 & W^9 & W^7 \end{bmatrix} \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \\ x_8 \\ x_5 \\ x_7 \\ x_9 \\ x_1 \\ x_3 \end{bmatrix} \qquad (4.29)$$

This matrix exhibits a block structure, each block having the form $W_2^{nm}D_5$, where D_5 is the 5x5 DFT matrix. Suppose we define

$$y_0 = \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \\ x_8 \end{bmatrix} \quad y_1 = \begin{bmatrix} x_5 \\ x_7 \\ x_9 \\ x_1 \\ x_3 \end{bmatrix} \quad Y_0 = \begin{bmatrix} X_0 \\ X_6 \\ X_2 \\ X_8 \\ X_4 \end{bmatrix} \quad Y_1 = \begin{bmatrix} X_5 \\ X_1 \\ X_7 \\ X_3 \\ X_9 \end{bmatrix} \qquad (4.30)$$

Then (4.29) may be rewritten as

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} W_2^0 D_5 & W_2^0 D_5 \\ W_2^0 D_5 & W_2^1 D_5 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \qquad \text{since } W_{10}^5 = W_2^1 \qquad (4.31)$$

However since the values $\begin{bmatrix} W_2^0 & W_2^0 \\ W_2^0 & W_2^1 \end{bmatrix}$ are the 2x2 DFT matrix D_2. Then (4.31) can be written in terms of the Kronecker product [4.11] of D_2 and D_5

$$\tilde{X} = (D_2 \otimes D_5) \tilde{x} \qquad (4.32)$$

$\tilde{X}$ and $\check{x}$ are the reordered vectors X and x respectively. If 2 had been chosen as the inner factor, rather than 5, the matrix (4.29) would have exhibited 5x5 blocks of 2x2 matrices rather than 2x2 blocks of 5x5 matrices. The Kronecker product can be applied to any number of mutually prime factors, i.e.. in general

$$X = (D_{N_1} \otimes D_{N_2} \otimes \ldots \otimes D_{N_L}) x \tag{4.33}$$

4.5 The Nested Winograd Fourier Transform Algorithm

The way the 'small-N' DFT modules can be nested together into longer transforms has been discussed by Winograd [4.12,4.13], Silverman [4.14-4.16], Kolba and Parks [4.6] and by Agarwal and Cooley [4.17]. Perhaps the simplest approach to this nesting is that adopted by Silverman [4.14], the following discussion is based upon his approach.

As discussed in chapter 2 each of Winograd's 'small-N' DFT algorithms has the form

$$Y = C\ (Ay \times Bz) \tag{1.20}$$

This can be rewritten as

$$Y = CB'Ay \tag{4.34}$$

where the A and C matrices are as before and B' is a diagonal MxM matrix, the values along the diagonal being the M values Bz of (1.20). Substituting (4.34) into (4.33) gives

$$X = ((C_{N_1} B'_{N_1} A_{N_1}) \otimes (C_{N_2} B'_{N_2} A_{N_2}) \otimes \ldots \otimes (C_{N_L} B'_{N_L} A_{N_L}))x \tag{4.35}$$

Matrix multiplication is associative and a property of the Kronecker product [4.11 p11] is that

$$AB \otimes CD = (A \otimes B)(C \otimes D) \tag{4.36}$$

Then repeated application of (4.36) to (4.35) gives

$$X = ((C_{N_1} \otimes C_{N_2} \otimes \ldots \otimes C_{N_L})(B'_{N_1} \otimes B'_{N_2} \otimes \ldots \otimes B'_{N_L})(A_{N_1} \otimes A_{N_2} \otimes \ldots \otimes A_{N_L}))x \tag{4.37}$$

The form (4.37) has several interesting consequences. As all the B'_{N_i} matrices are diagonal, their Kronecker product will also be diagonal. As the diagonal matrices only contain real and imaginary terms, so will their Kronecker product. Similarly if the A_N and B_N matrices only contain +1, −1 and 0, their Kronecker products will also only contain +1, −1 and 0. Thus the nested form for the WFTA (4.37) is of the same general form as (1.20).

When the nested A and C matrices are generated in this manner the input and output of the algorithm are ordered by (4.17) and (4.18), exactly the same as the matrix vector product (4.29). To give a transform using data in natural order and giving an output in natural order, the columns of the nested A matrix should be reordered by (4.17) and the rows of the nested C matrix reordered by (4.18). There is no need to reorder the coefficients.

Figure 4.2, taken from Kolba and Parks [4.6], gives another way of viewing the nested form of (4.41).

4.5.1 Operations Count for the Nested WFTA

If each of the short-N WFTAs of N_i points requires M_i multiplications and A_i additions for real data, then the operation counts for the fully nested form (4.37) with L factors, for complex data, are

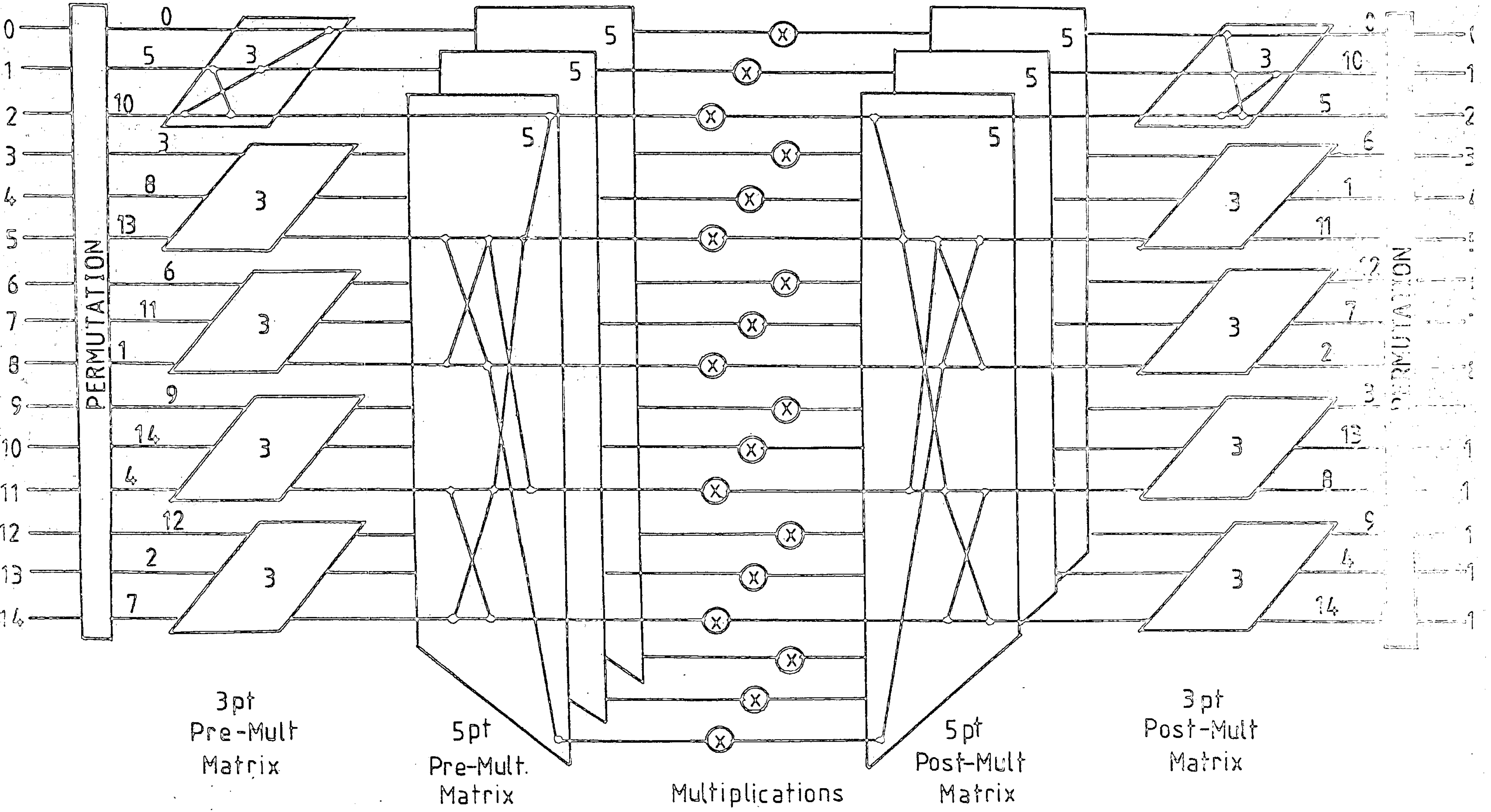
$$N = \prod_{i=1}^{L} N_i$$

$$2 \prod_{i=1}^{L} M_i \qquad \text{Real Multiplications} \qquad (4.38)$$

$$2 \quad \frac{N}{N_1} A_1 + \frac{N}{N_2} \frac{M_1}{N_1} A_2 + \dots + \frac{N}{N_L} \left(\prod_{i=1}^{L-1} \frac{M_i}{N_i} \right) A_L \qquad \text{Real Additions} \quad (4.39)$$

N_L is the innermost and most rapidly varying factor. Whilst the number of multiplications needed is independent of the order of the factors the number of additions is not. Agarwal and Cooley [4.17] discuss this problem and show that in order to achieve the minimum number of

Fig. 4.2 A 15-point WFTA



additions the factors should be ordered so that the quantity

$$T(N_i) = \frac{M_i - N_i}{A_i} \qquad (4.40)$$

increases towards the innermost factor. For example, in the 63-point DFT, $N_1=7$, $N_2=9$, $M_1=9$, $A_1=36$, $M_2=11$ and $A_2=45$. Then $T_1=\frac{1}{18}$ and $T_2=\frac{2}{45}$, Hence 7 should be the innermost factor. The number of additions needed is

$$\text{7 as the innermost factor} \quad 2[\frac{63}{9}\,45 + \frac{63}{7}\cdot\frac{11}{9}\,36\,] = 1422$$

$$\text{9 as the innermost factor} \quad 2[\frac{63}{7}\,36 + \frac{63}{9}\cdot\frac{9}{7}\,45\,] = 1458$$

As predicted, less additions are required if 7 is the innermost factor. These considerations do not apply if (4.37) is taken as a set of matrix products.

It is interesting to consider the computational savings, if any, of the nested form over the Prime Factor Algorithm. For the nested form to require less multiplications than the PFA, then

$$N_2M_1 + N_1M_2 > M_1M_2 \quad \text{or} \quad N_1/M_1 + N_2/M_2 > 1 \qquad (4.41)$$

In general $M_1>N_1$ and $M_2>N_2$. For the smaller N algorithms M_1 and M_2 are only slightly larger than N_1 and N_2. As N_i/M_i decreases only slowly with increasing N_i, condition (4.41) is almost always met. Condition (4.41) is not met for extremely large DFTs or when sub-optimal 'small-N' algorithms are used. An example of the latter case is described by Johnson and Burrus [4.19].

As the nested form computes the N_1N_2 point DFT with $N_2A_1+M_1A_2$ additions and the PFA with $N_2A_1+N_1A_2$ additions, the PFA will always require less additions than the nested form except when $M_1=N_1$. Thus there will always be a tradeoff between the nested WFTA and the PFA in terms of number of multiplications and additions.

4.6 Block Structure And Nesting of Convolution Algorithms

So far in this chapter only DFTs have been discussed. Do mappings and block structures similar to the DFT exist for convolutions?

Burrus [4.2], Agarwal and Burrus [4.18] both discuss the mapping of a one-dimensional convolution into a multi-dimensional form. Consider a length N cyclic convolution

$$y_n = \sum_{k=0}^{N-1} h_k x_{<n-k>} \tag{1.3}$$

where the subscripts are evaluated modulo N. Then substituting the mapping

$$\begin{aligned} n &= K_1 n_1 + K_2 n_2 \\ k &= K_1 k_1 + K_2 k_2 \end{aligned} \tag{4.42}$$

i.e. the same mapping for both indices, into (1.3) gives

$$y_{K_1 n_1 + K_2 n_2} = \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} h_{K_1 k_1 + K_2 k_2} x_{<K_1 n_1 + K_2 n_2 - K_1 k_1 - K_2 k_2>} \tag{4.43}$$

Then by defining suitable two-dimensional arrays

$$\hat{y}_{n_1, n_2} = \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} \hat{h}_{k_1, k_2} \hat{x}_{n_1-k_1, n_2-k_2} \tag{4.44}$$

Equation (4.44) represents a true two-dimensional convolution. This convolution is cyclic along n_1 if and only if $K_1=\alpha N_2$, and cyclic along n_2 if and only if $K_2=\beta N_1$. If N_1 and N_2 are relatively prime, it is possible for the mapping to be cyclic in both N_1 and N_2, if they have a common factor this is not so.

In the simplest case $\alpha=\beta=1$ and, for a two factor example, the map reduces to

$$n = N_2 n_1 + N_1 n_2 \quad \text{and} \quad k = N_2 k_1 + N_1 k_1 \tag{4.45}$$

As an example consider a 10-point cyclic convolution, which when written in matrix vector notation becomes

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \end{pmatrix} = \begin{pmatrix} x_0 & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 \\ x_1 & x_0 & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 \\ x_2 & x_1 & x_0 & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 \\ x_3 & x_2 & x_1 & x_0 & x_9 & x_8 & x_7 & x_6 & x_5 & x_4 \\ x_4 & x_3 & x_2 & x_1 & x_0 & x_9 & x_8 & x_7 & x_6 & x_5 \\ x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & x_9 & x_8 & x_7 & x_6 \\ x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & x_9 & x_8 & x_7 \\ x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & x_9 & x_8 \\ x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & x_9 \\ x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \end{pmatrix} \begin{pmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{pmatrix}$$

Both the rows and the columns are reordered by the mapping

$$n = 5n_1 + 2n_2 \qquad n_1=0,1 \qquad n_2=0,1,2,3,4$$

with n_2 varying most rapidly the mapping is

$$0,\ 2,\ 4,\ 6,\ 8,\ 5,\ 7,\ 9,\ 1,\ 3$$

This gives

$$\begin{pmatrix} y_0 \\ y_2 \\ y_4 \\ y_6 \\ y_8 \\ y_5 \\ y_7 \\ y_9 \\ y_1 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_0 & x_8 & x_6 & x_4 & x_2 & x_5 & x_3 & x_1 & x_9 & x_7 \\ x_2 & x_0 & x_8 & x_6 & x_4 & x_7 & x_5 & x_3 & x_1 & x_9 \\ x_4 & x_2 & x_0 & x_8 & x_6 & x_9 & x_7 & x_5 & x_3 & x_1 \\ x_6 & x_4 & x_2 & x_0 & x_8 & x_1 & x_9 & x_7 & x_5 & x_3 \\ x_8 & x_6 & x_4 & x_2 & x_0 & x_3 & x_1 & x_9 & x_7 & x_5 \\ x_5 & x_3 & x_1 & x_9 & x_7 & x_0 & x_8 & x_6 & x_4 & x_2 \\ x_7 & x_5 & x_3 & x_1 & x_9 & x_2 & x_0 & x_8 & x_6 & x_4 \\ x_9 & x_7 & x_5 & x_3 & x_1 & x_4 & x_2 & x_0 & x_8 & x_6 \\ x_1 & x_9 & x_7 & x_5 & x_3 & x_6 & x_4 & x_2 & x_0 & x_8 \\ x_3 & x_1 & x_9 & x_7 & x_5 & x_8 & x_6 & x_4 & x_2 & x_0 \end{pmatrix} \begin{pmatrix} h_0 \\ h_2 \\ h_4 \\ h_6 \\ h_8 \\ h_5 \\ h_7 \\ h_9 \\ h_1 \\ h_3 \end{pmatrix} \qquad (4.46)$$

Again this reordered matrix exhibits a block structure. Proceeding in a similar manner to the 10-point DFT case, we define

$$\mathbb{Y}_0 = \begin{bmatrix} y_0 \\ y_2 \\ y_4 \\ y_6 \\ y_8 \end{bmatrix} \quad \mathbb{Y}_1 = \begin{bmatrix} y_5 \\ y_7 \\ y_9 \\ y_1 \\ y_3 \end{bmatrix} \quad \mathbb{H}_0 = \begin{bmatrix} h_0 \\ h_2 \\ h_4 \\ h_6 \\ h_8 \end{bmatrix} \quad \mathbb{H}_1 = \begin{bmatrix} h_5 \\ h_7 \\ h_9 \\ h_1 \\ h_3 \end{bmatrix}$$

and

$$\mathbb{X}_1 = \begin{bmatrix} x_0 & x_8 & x_6 & x_4 & x_2 \\ x_2 & x_0 & x_8 & x_6 & x_4 \\ x_4 & x_2 & x_0 & x_8 & x_6 \\ x_6 & x_4 & x_2 & x_0 & x_8 \\ x_8 & x_6 & x_4 & x_2 & x_0 \end{bmatrix} \quad \mathbb{X}_2 = \begin{bmatrix} x_5 & x_3 & x_1 & x_9 & x_7 \\ x_7 & x_5 & x_3 & x_1 & x_9 \\ x_9 & x_7 & x_5 & x_3 & x_1 \\ x_1 & x_9 & x_7 & x_5 & x_3 \\ x_3 & x_1 & x_9 & x_7 & x_5 \end{bmatrix}$$

So (4.46) may be rewritten as

$$\begin{bmatrix} \mathbb{Y}_0 \\ \mathbb{Y}_1 \end{bmatrix} = \begin{bmatrix} \mathbb{X}_1 & \mathbb{X}_2 \\ \mathbb{X}_2 & \mathbb{X}_1 \end{bmatrix} \begin{bmatrix} \mathbb{H}_0 \\ \mathbb{H}_1 \end{bmatrix} \qquad (4.47)$$

This clearly shows that (4.46) is a convolution of convolutions. Thus Winograd's small-N convolution algorithms, which have the three matrix structure of (4.34), may be nested together. No intermediate form equivalent to the Prime Factor Algorithm exists for convolutions. The nested form applicable to the 10-point convolution example above is

$$\mathbb{Y} = (C_2 \otimes C_5)(B_2' \otimes B_5')(A_2 \otimes A_5) \qquad (4.48)$$

As before this nesting can be extended to any number of mutually prime factors. Using the same notation as before the number of operations for a real cyclic convolution evaluated using the nested form is given by

$$N = \prod_{i=1}^{L} N_i$$

$$\prod_{i=1}^{L} M_i \quad \text{Real multiplications} \qquad (4.49a)$$

$$\frac{N}{N_1} A_1 + N \sum_{i=2}^{L} \left[\left(\frac{A_i}{N_i}\right) \left\{ \prod_{k=1}^{i-1} \left(\frac{M_k}{N_k}\right) \right\} \right] \qquad \text{Additions} \qquad (4.49b)$$

4.7 Summary of Chapter 4

This chapter has dealt with the use of multi-dimensional mappings to construct longer algorithms for convolution and the DFT from sets of short length algorithms. For the DFT the main algorithms considered were the FFT, the Prime Factor Algorithm (PFA) and the nested Winograd Fourier Transform Algorithm (WFTA). The nested WFTA has the same form as the 'small-N' algorithms, i.e. equation (1.20), and the A and C matrices of the nested form will only contain +1, -1 and 0 provided that the 'small-N' A and C did so.

The number of multiplications required by the WFTA will normally be less than the PFA, but the PFA will require less additions. A more detailed comparison is made between the algorithms in the next chapter.

Finally this chapter showed that 'small N' convolution algorithms can be nested to form longer convolutions. Again the nested algorithm preserves the form of (1.20) with A and C matrices containing only +1, -1 and 0.

Chapter 5

Comparison of DFT and Convolution Algorithms

We now return to the problem stated at the beginning of the thesis. "What is the 'best' way of calculating the convolution of a large picture with a small (fixed) window?" This chapter considers, in purely arithmetic terms, the best algorithm to choose so that the minimum number of multiplications are calculated.

The latter parts of the thesis are concerned with the implementation of algorithms having the general form of (1.20) with A and C matrices containing only +1, -1 and 0. An important feature of this implementation is that the dimensions of the arrays of one-bit full adder cells depends upon the word length and the number of multiplications used in the algorithm. The dimensions of the arrays are unaffected by the number of additions required to find the matrix products involving the A and C matrices. Consequently algorithms are compared with a view to minimising the number of multiplications needed for the function.

Before considering two-dimensional convolutions the number of operations for one-dimensional DFTs and convolutions, calculated using the methods outlined in Chapters 2 and 4, are compared.

5.1 Operation Counts for the One-Dimensional DFT

This section compares three algorithms for the one-dimensional DFT for complex data, the FFT, the Prime Factor Algorithm (PFA) and the nested Winograd Fourier Transform Algorithm (WFTA).

5.1.1 The FFT

The operations counts for a $N=2^n$-point radix-2 FFT, for complex data, are given by

$2N\ \mathrm{Log}_2 N$ Real Multiplications

$3N\ \mathrm{Log}_2 N$ Real Additions $N=2^n$ only (4.8)

Whilst this is not the best FFT algorithm it is taken as being representative of these types of algorithm. Table 5.1.1 gives the operations counts for some transform lengths.

Transform Length	Real Mults.	Real Additions	Mults. per point	Additions per point
8	48	72	6	9
16	128	192	8	12
32	320	480	10	15
64	768	1152	12	18
128	1792	2688	14	21
256	4096	6144	16	24
512	9216	13824	18	27
1024	20480	30720	20	30

Table 5.1.1 The Radix-2 FFT

5.1.2 The Prime Factor Algorithm

The PFA uses Good's mapping and Winograd's 'small-N' algorithms to calculate the DFTs along each of the dimensions resulting from the multi-dimensional mapping. If M_i and A_i are the number of muliplications and additions for a N_i-point 'short-N' WFTA for real data, then the number of operations for an algorithm with L factors with complex data is

$$N = \prod_{i=1}^{L} N_i$$

$$2\sum_{i=1}^{L}\left(\frac{N}{N_i}\right)M_i \qquad \text{Real Multiplications}$$

$$2\sum_{i=1}^{L}\left(\frac{N}{N_i}\right)A_i \qquad \text{Real Additions} \qquad (4.20)$$

The number of operations for the 'small-N' WFTA algorithms is given by table 2.3 in chapter 2. Table 5.1.2 gives the factorizations and operation counts for complex data. The 11 and 13-point DFT algorithms given in table 2.3 are not used as they do not offer a low number of multiplications per point.

Transform Length	Factors	Real Mults.	Real Additions	Mults. per point	Additions per point
12	3,4	48	96	4.00	8.00
20	4,5	88	216	4.40	10.80
30	2,3,5	192	384	6.40	12.80
60	3,4,5	384	888	6.40	14.80
120	3,5,8	768	2076	6.40	17.30
240	3,5,16	1596	4812	6.65	20.05
504	7,8,9	3536	13500	7.02	26.79
1008	7,8,16	7324	29772	7.27	29.54

Table 5.1.2 The Prime Factor Algorithm

The number of multiplications given in table 5.1.2 includes those involving ±1 and ±j.

5.1.3 The WFTA

Using the notation of the previous section the number of operations for the WFTA is given by

$$2\prod_{i=1}^{L} M_i \qquad \text{Real Multiplications} \qquad (4.38)$$

$$2\,\frac{N}{N_1}\,A_1 + \frac{N}{N_2}\,\frac{M_1}{N_1}\,A_2 + \ldots + \frac{N}{N_L}\left\{\sum_{i=1}^{L-1}\left(\frac{M_i}{N_i}\right)\right\}A_L \qquad \text{Real Additions} \tag{4.39}$$

Table 5.1.3 gives the operation counts, for complex data, the same transform lengths as the PFA. The factorisation is given in optimal order, with the innermost factor last.

Transform Length	Optimal Factor Order	Real Mults.	Real Additions	Mults. per point	Addition per point
12	3,4	24	96	2.00	8.00
20	4,5	48	216	2.40	10.80
30	2,3,5	72	384	2.40	12.80
60	4,3,5	144	888	2.40	14.80
120	8,3,5	288	2076	2.40	17.30
240	3,16,5	648	5016	2.70	20.90
504	8,9,7	1584	14652	3.14	29.07
1008	16,9,7	3564	34920	3.54	34.64

Table 5.1.3 The WFTA

Nussbaumer [5.1] introduces a technique called 'split-nesting' to reduce the number of additions with no change in the number of multiplications. This, however, increases the algorithm complexity and is not considered here.

Note that all these nested algorithms will have A and C matrices containing only +1, -1 and 0. As with the PFA the number of multiplications includes some apparently trivial ones by ± 1 and $\pm j$. They are included so as to preserve the general form of (1.20).

5.1.4 Comparison of DFT Algorithms

Comparison of tables 5.1.1-5.1.3 shows that the WFTA always offers the minimum number of multiplications for roughly comparable transform lengths. The PFA requires less additions than the WFTA for longer transform lengths. All three algorithms use roughly the same number of additions for comparable transform lengths.

Much has been written [5.2-5.8] in the recent literature on the comparison between programs and algorithms to implement DFTs on general purpose computers. There appears to be no firm conclusion to be drawn from these studies. However it seems that on general purpose machines the 'in-order in-place' PFA may be the fastest algorithm [5.2].

One point in common in these studies is that the arithmetic savings of the WFTA are largely outweighed by it's relative complexity.

5.2 One-Dimensional Convolutions

There are two main alternatives for evaluating one-dimensional cyclic convolutions. These are the nesting of 'short-N' convolution algorithms, such as those derived in Chapter 2, or the the calculation of convolutions by transforms having the cyclic convolution property. In the comparison that follows it is assumed that one of the sequences to be convolved is fixed - the filter coefficients - and that two real convolutions are computed by each complex DFT. Finally the optimum block length for one-dimensional filters involving a fixed filter tap length and a semi-infinite sequence is discussed.

5.2.1 Nesting 'Short-N' Convolution Algorithms

Using the same notation as the PFA for the number of operations for each of the factors, the operation count for the nested algorithm is given by

$$\prod_{i=1}^{L} M_i \qquad \text{Multiplications} \qquad (4.49a)$$

$$\frac{N}{N_1} A_1 + N\left[\sum_{i=2}^{L}\left(\frac{A_i}{N_i}\right)\left\{\prod_{k=1}^{i-1}\left(\frac{M_k}{N_k}\right)\right\}\right] \qquad \text{Additions} \qquad (4.49b)$$

Table 5.2.1 gives the number of operations for a variety of convolution lengths. The number of operations for the 'small-N' convolutions are taken from table 2.2. The factorisation is given in optimal order to produce the minimum number of additions, with the innermost factor being given last.

Convolution Length	Optimal Factorisation	Real Mults.	Real Additions	Mults. per point	Additions per point
12	4,3	20	100	1.67	8.33
20	4,5	50	230	2.50	11.50
30	6,5	80	418	2.67	13.93
60	4,3,5	200	1120	3.33	18.67
120	3,8,5	560	3096	4.67	25.80
240	3,5,16	1640	8504	6.83	35.43
504	8,9,7	5852	34678	11.61	68.81
1008	9,7,16	17138	95258	17.00	94.50

Table 5.2.1
Real Cyclic convolutions calculated using nested 'short-N' convolution Algorithms.

Again all these nested algorithms, except those containing the factor 7, will have A and C matrices containing only +1, -1 and 0.

5.2.2 Cyclic Convolutions Calculated by Transforms having the CCP

By definition transforms possessing the Cyclic Convolution Property may be used to compute cyclic convolutions! One useful property when dealing with real convolutions is to calculate two real only convolutions using one complex DFT. Assume that h_n is fixed, the convolution of h_n with the two N-point sequences x_m and x'_m is found by first constructing the complex sequence $x_m + jx'_m$. The complex convolution of h_n with $x_m + jx'_m$ is the computed by DFTs to yield the complex convolution $y_m + jy'_m$. Thus the convolution of h_n with x_m is defined by the real part of the complex convolution and the corresponding convolution of h_n and x'_m by the imaginary. Using this method the number of operations to compute a real convolution is half that of a complex convolution.

Suppose a DFT algorithm computes a N-point complex DFT with 2M real multiplications and 2A real additions. Then, assuming 4 real multiplications and 2 real additions per complex multiplication, a N-point real cyclic convolution is computed with

$$2[N + M] \quad \text{Real Multiplications} \tag{5.1a}$$

and

$$[2A + N] \quad \text{Real Additions} \tag{5.1b}$$

Table 5.2.2 below gives the number of operations for real convolutions evaluated using the WFTA and PFA (using figures from tables 5.1.1 and 5.1.2)

Convolution Length	WFTA Real Mults	WFTA Real Adds	PFA Real Mults	PFA Real Adds
12	48	108	72	108
20	88	236	128	236
30	132	414	252	414
60	264	948	504	948
120	528	2196	1008	2196
240	1128	5256	2076	5052
504	2592	15156	4544	14004
1008	5580	35928	9340	30780

Table 5.2.2
Number of real operations for real cyclic convolutions computed using the WFTA and PFA.
(2 real convolutions per DFT; one input sequence fixed)

5.2.3 Comparison of Ways of Computing 1-D Cyclic Convolutions

Comparison of tables 5.2.1 and 5.2.2 shows that in terms of numbers of multiplications the 'best' method of calculating one-dimensional cyclic convolutions depends upon the convolution length. For smaller convolutions the nested 'short-N' convolution algorithms use less multiplications than either the WFTA or PFA solutions. For long convolutions the WFTA uses less multiplications and additions than the nested 'short-N' convolution algorithms. Where is the cross-over between the two methods? Inspection of the previous two tables suggests somewhere between 60 and 120-point convolutions. Table 5.2.3 below considers a few more convolution lengths in this region.

Convolution Length	Nested 'short-N' Real Mults.	Nested 'short-N' Real Adds.	WFTA Real Mults.	WFTA Real Adds.
60	200	1120	264	948
80	410	1906	376	1432
90	440	2524	444	1918
112	779	3831	548	2444
120	560	3096	528	2196

Table 5.2.3
Comparison of Operations for 1-D convolutions of 60 to 120 points

Examination of table 5.2.3 immediately shows that there is no clear crossover between the algorithms. It also illustrates that longer algorithms may use less operations than some shorter ones - compare the 112 and 120-point cases above. In terms of multiplications the cross-over point between the two methods is around 90 points.

For convolutions up to 90-points use nested 'short-N' convolution algorithms, such as those derived in chapter 2. Convolutions of greater length should be evaluated using the WFTA. The exact crossover point depends upon the assumptions made about the way the DFT method calculates real convolutions.

5.2.4 Optimum Block Length for 1-D Filters

In many digital filtering applications one sequence, h_n often comprises of a limited number of points, N_1, and represents the impulse response of the filter. The other sequence, x_m, is often very large. The aperiodic convolution of these two sequences may be obtained by sectioning x_m, performing a series of N-point cyclic convolutions and using the Overlap-Save or Overlap-Add techniques, described in chapter 1, to reconstruct the filter output.

When using the Overlap-Save method the input data is sectioned into blocks of length N_2, each block overlapping the previous one by N_1-1 samples. The window is extended to $N=N_1+N_2-1$ points by appending N_2-1 zeroes and a cyclic convolution of N points is performed. Only N_2 samples of each block are retained, the other values are discarded.

If $M_1(N)$ a~~n~~d is the number of multiplications per output point for cyclic convolution of length N and $M_2(N,N_1)$ the number of multiplications per output point for a N_1-tap digital filter, then $M_2(N,N_1)$ is given by

$$M_2(N,N_1) = M_1(N)N/(N-N_1+1) \tag{5.2a}$$

Similarly, $A_2(N,N_2)$, the number of additions per output point of the filter, when using Overlap-Save, is given as

$$A_2(N,N_1) = A_1(N)N/(N-N_1+1) \tag{5.2b}$$

where $A_1(N)$ is the number of additions per point for a cyclic convolution of length N. If $M_1(N)$ was an increasing function of N, then, as $N/(N-N_1+1)$ is a decreasing one, there would be an optimum block size N which minimised the number of multiplications in the filter. Despite $M_1(N)$ not being a monotonically increasing function of N for the WFTA and the nested 'short-N' algorithms it is still possible to find a minimum for a given tap length. Table 5.2.4 gives the optimal block length for the minimum number of multiplications for a variety of filter tap lengths evaluated using 'short-N' convolution algorithms. The choice of possible convolution lengths was restricted to algorithms containing only +1, -1 and 0 in their A and C matrices, i.e. algorithms containing a factor of 7 were excluded.

Filter Tap Length	Optimum Block size	Multiplications per point	Additions per point
2	6	1.60	6.80
4	12	2.22	11.11
8	24	3.29	16.00
16	60	4.44	32.00
32	120	6.29	34.79
64	240	9.27	48.05
128	360	13.22	79.30
256	720	19.40	175.98
512	720	43.16	238.14

Table 5.2.4
Optimum block sixes and number of operations per point for 1-D filters computed using nested 'short-N' algorithms

5.3 Two-Dimensional Convolutions

In a similar manner to the previous section the number of operations to calculate two-dimensional cyclic convolutions is discussed in this section. There are several additional techniques for computing 2-D convolutions apart form those applied to the 1-D convolutions above. These include the use of 2-D cyclic convolution algorithms derived using polynomial transforms and 2-D DFT algorithms also derived from polynomial transforms.

The methods are discussed in the following order: nested 1-D WFTAs, 2-D DFTs based on polynomial transforms, nested 1-D 'short-N' convolution algorithms and 2-D convolution algorithms based upon polynomial transforms.

Finally the optimum block size for the window-picture convolution problem is considered.

Throughout this chapter only NxN-point convolutions are considered. It should be noted that it is often possible to derive $N_1 x N_2$-point convolutions $N_1 \neq N_2$.

5.3.1 Nested 1-D WFTAs

It follows from the derivation in chapter 4 that a nested WFTA can be built up not only from one-dimensional data which has been mapped into a multi-dimensional array, but also from data that is inherently multi-dimensional. Furthermore because the data is multi-dimensional no mappings are required and the constraint that the lengths of the dimensions be mutually prime is lifted. However there is still the restriction that the length of any one dimension may itself not contain any factors having a greater common divisor greater than one.

The above comments apply equally to other algorithms utilising multi-dimensional mappings. It should be apparent that a whole host of algorithms are possible, all with varying degrees of nesting. For example, it is possible to calculate a two-dimensional DFT using a 1-D WFTA along one dimension and a PFA along another.

The minimum number of multiplications will occur when a fully nested WFTA is used. Suppose a N_i-point WFTA requires $2M_i$ real multiplications and $2A_i$ additions (4.38,4.39) for complex data, then the number of operations for a $N_1 x N_2$-point two-dimensional DFT for complex data will be

$$2M_1M_2 \qquad \text{Real Multiplications} \qquad (5.3a)$$

$$2[N_1A_2 + M_2A_1] \qquad \text{Real Additions} \qquad (5.3b)$$

Table 5.3.1 below lists the number of operations for some NxN-point DFTs calculated by nesting 1-D WFTAs.

DFT Size	Real Multiplications	Real Additions	Mults. per point	Additions per point
12x12	288	2304	2.00	16.00
20x20	1152	9504	2.88	23.76
30x30	2592	25344	2.88	28.16
60x60	10368	117216	2.88	32.56
120x120	41472	548064	2.88	38.06
240x240	209952	2829024	3.65	49.12
504x504	1254528	18988992	4.94	74.76
1008x1008	6351048	97426800	6.25	95.88

Table 5.3.1
Operation counts for complex DFTs evaluated by Nested WFTAs

Again it should be noted that if the 1-D WFTAs have A and C matrices containing only +1, -1 and 0 then so will these 2-D DFTs.

5.3.2 Nested 2-D DFTs

Nussbaumer and Quandalle [5.9,5.10] introduced a way of computing multi-dimensional DFTs using polynomial transforms. These algorithms have the form of (1.20). Judging from their structure many of these algorithms will have A and C matrices containing only +1, -1 and 0. Table 5.3.2a, taken from Nussbaumer [5.1,p192], lists the operation counts for real data for some of these polynomial transform based algorithms and the corresponding nested 2-D WFTAs.

DFT Size	Poly.Trans. Mults.	Poly.Trans. Additions	Nested WFTA Mults.	Nested WFTA Additions
2x2	4	8	4	8
3x3	9	36	9	36
4x4	16	64	16	64
5x5	31	221	36	187
7x7	65	635	81	576
8x8	64	408	64	416
9x9	105	785	121	900
16x16	304	2264	324	2516

Table 5.3.2a
Operations for 2-D DFTs evaluated using polynomial transforms

As these polynomial transform based algorithms are of the general form of (1.20) they may be nested together in a similar manner ot the WFTA. If a $N_i x N_i$-point DFT requires M_i multiplications and A_i addition for real data then the number operation required for a nested algorithm for complex data is

$$N = \prod_{i=1}^{L} N_i \qquad \text{NxN-point DFT}$$

$$2 \prod_{i=1}^{L} M_i \qquad \text{Real Multiplications} \qquad (5.4a)$$

$$2\left[\left(\frac{N^2}{N_1^2}\right)A_1 + \sum_{i=2}^{L}\left(\frac{N^2}{N_i^2}\right)A_i\left\{\prod_{k=1}^{i-1}\left(\frac{M_k}{N_k^2}\right)\right\}\right] \qquad \text{Real Additions} \qquad (5.4b)$$

Table 5.3.2b gives the number of operations used for the same DFTs as table 5.3.1 but using the polynomial transform based 2-D DFT algorithms.

DFT Size	Real Multiplications	Real Additions	Mults. per point	Additions per point
12x12	288	2304	2.00	16.00
20x20	992	10272	2.48	25.68
30x30	2232	26712	2.48	29.68
60x60	8928	121248	2.48	33.68
120x120	35712	553392	2.48	38.43
240x240	169632	2688912	2.95	46.68
504x504	873600	16353584	3.44	64.38
1008x1008	4149600	80267312	4.08	79.00

Table 5.3.2b
Operation counts for complex 2-D DFTs computed by polynomial transforms and Nesting

Comparison of tables 5.3.1 and 5.3.2b shows that the polynomial transform based 2-D DFT algorithms use an equal number or less multiplications than the nested WFTA and less additions for DFTs greater than 120x120-points.

Nussbaumer [5.1] discusses some techniques for reducing the number of additions.

Having found, from the minimum number of multiplications point of view, the optimal way of calculating a 2-D DFT, we now consider the number of operations for 2-D cyclic convolutions evaluated using polynomial transform based 2-D DFT algorithms.

5.3.3 2-D Cyclic Convolutions Calculated using 2-D DFTs

Proceeding in a manner analogous to the one-dimensional case, the DFTs of the previous sub-section are applied to the calculation of 2-D cyclic convolutions. Again it is assumed that a complex multiplication requires 4 real multiplications together with 2 real additions and each complex DFT performs two real convolutions. The DFT operation counts are for the 2-D DFTs derived using polynomial transforms.

Convolution Size	Real Multiplications	Real Additions	Mults. per point	Additions per point
12x12	576	2448	4.00	17.00
20x20	1792	10672	4.48	26.68
30x30	4032	27612	4.48	30.68
60x60	16128	124848	4.48	34.68
120x120	64512	567792	4.48	39.43
240x240	284832	2746512	4.95	47.68
504x504	1381632	16607600	5.44	65.38
1008x1008	6181728	81283376	6.08	80.00

Table 5.3.3
Number of operations for real cyclic 2-D convolutions computed using 2-D DFT algorithms based upon polynomial transforms
(2 real convolutions per DFT, one input sequence fixed)

5.3.4 2-D Cyclic Convolutions Calculated using Nested 1-D Convolution Algorithms

Another method for the calculation of 2-D cyclic convolution algorithms is the use of nested 1-D 'short-N' convolution algorithms. As noted at the beginning of the section on 2-D DFTs a NxN-point convolution may be calculated provided that the factors of N are mutually prime. Remember that for convolutions no intermediate form similar to the PFA exists, the algorithm must be the fully nested form.

If a N_i-point 1-D cyclic convolution algorithm requires M_i multiplications and A_i additions then a $N_1 x N_2$-point cyclic convolution algorithm for real data will require

$M_1 M_2$ Multiplications (5.5a)

$N_1 A_2 + M_2 A_1$ Additions. (5.5b)

A lower number of additions may be obtained by treating a $N_1 N_2 x N_1 N_2$-point convolution as a $(N_1 x N_1) x (N_2 x N_2)$-points rather than

$(N_1xN_2)x(N_1xN_2)$-points whenever

$$N_1A_2 + M_2A_1 > A_2M_1 + A_1N_2$$

Table 5.3.4 gives the number of operations for a variety of convolution sizes using (5.5a,b). It uses the figures taken from table 5.2.1, no attempt was made to minimise the number of additions.

Consideration of the 1-D case for nested 'short-N' convolutions shows that in the 2-D case this method will only be better than the 2-D DFT solutions for $N^2 \simeq 90$. This borne out by the values in table 5.3.4.

Convolution Size	Real Multiplications	Real Additions	Mults. per point	Additions per point
12x12	400	3200	2.78	22.22
20x20	2500	16100	6.25	40.25
30x30	6400	45980	7.11	51.09
60x60	40000	291200	11.11	80.89
120x120	313600	2105280	21.78	146.20
240x240	2689600	15987520	46.69	277.56
504x504	34245904	220413368	134.82	867.71

Table 5.3.4
Operation Count for real 2-D cyclic convolutions computed using Nested 1-D 'short-N' convolution algorithms

5.3.5 2-D Cyclic Convolutions Based Upon Polynomial Transforms

We now turn to the application of the 2-D cyclic convolution algorithms derived in chapter 3 to various 2-D convolutions. Although, as shown in chapter 3, it is possible to derive quite long algorithms, in particular $2^n x2^n$ points, it can be quite difficult and involved. The range of possible convolution lengths may be increased by the use of nesting. In this method a convolution of size $N_1N_2xN_1N_2$ $gcd(N_1,N_2)=1$, is converted into the four-dimensional convolution

$(N_1xN_1)x(N_2xN_2)$. If M_1 and A_1 are the numbers of multiplications and additions for a 2-D convolution of size N_1xN_1, then the number of operations to evaluate the 2-D convolution of size $N_1N_2xN_1N_2$ is

$$M_1M_2 \qquad \text{Multiplications} \qquad (5.6a)$$

$$N_2^2A_1 + M_1A_2 \qquad \text{Additions} \qquad (5.6b)$$

Table 5.3.5 lists the number of operations for the same convolution lengths as before. The number of operations for the various small 2-D convolution algorithms are taken from table 3.2 (chapter 3). Note that with the number of multiplications used here the A and C matrices contain only +1,-1 and 0. The optimal factor order, for the minimum number of additions, is given with the innermost factor last.

Convolution Size	Optimal Factor Order	Real Mults.	Real Additions	Mults. per point	Additions per point
12x12	4,3	286	2638	1.99	18.32
20x20	4,5	1210	11168	3.03	27.92
30x30	2,3,5	2860	29788	3.18	33.10
60x60	4,3,5	15730	171484	4.37	47.63
120x120	3,5,8	92950	955258	6.45	66.34
240x240	3,16,5	453310	5040848	7.87	87.51
504x504	7,9,8	3035890	34245950	11.95	134.82
1008x1008	7,16,9	14805802	176924690	14.57	174.13

Table 5.3.5
2-D Cyclic convolutions computed by nested polynomial transform algorithms

5.3.6 Comparison of 2-D Cyclic Convolution Computation Methods

In terms of numbers of multiplications comparison of tables 5.3.3, 5.3.4 and 5.3.5 shows that nested 1-D convolution algorithms do not form a good way of computing 2-D cyclic convolutions. The choice lies between the other two methods. For 2-D cyclic convolutions up to about 60x60 points the nested polynomial transform algorithms offer the least number of multiplications. For 2-D cyclic convolutions of larger sizes the best approach is to use the 2-D DFT algorithms based upon polynomial transforms. The DFT approach usually offers the least number of additions per point.

5.3.7 Optimum Block Size for 2-D Convolutions

What is the 'best' way of calculating the convolution of a small window with a large picture? Again we take the word best to mean the minimum number of multiplications. However this may not be the overriding criterion for algorithm selection. For example, some solutions, particularly those involving large DFTs, demand knowledge of the entire picture before the computation can begin. The amount of storage required may be prohibitive. In other situations it may be appropiate to store as few lines of the picture as possible and start the convolution immediately. Clearly the 'best' solution is application depend$\overset{e}{\text{a}}$nt.

Despite these considerations we now consider the optimum block size into which a picture should be broken so as to give the minimum number of multiplications per point for a two-dimensional cyclic convolution.

If $M_1(N^2)$ is the number of multiplications per output point for a cyclic convolution of size NxN and $M_2(N^2, N_1^2)$ is the number of multiplications per output point for the $N_1 x N_1$-point aperiodic convolution, $M_2(N^2,N_1^2)$ is given by

$$M_2(N^2,N_1^2) = M_1(N^2)N^2/(N-N_1+1)^2 \qquad (5.7a)$$

similarly for the number of additions

$$A_2(N^2,N_1^2) = A_1(N^2)N^2/(N-N_1+1)^2 \qquad (5.7b)$$

Table 5.3.7 gives the optimum block size corresponding to the minimum number of multiplications per point for windows of sizes 3x3 to 15x15 points. This range of window sizes was chosen as being most suitable for image processing applications. All possible 2-D DFT and convolution algorithms were compared.

Window Size	Optimum Block Size	Algorithm Type	Mults. per point	Additions per point
3x3	12x12	2-D Conv	2.86	12.00
4x4	18x18	2-D Conv	3.43	21.75
5x5	18x18	2-D Conv	3.94	19.88
6x6	36x36	2-D Conv	4.42	44.75
7x7	48x48	2-D Conv	4.67	32.33
8x8	48x48	2-D Conv	4.90	72.75
9x9	168x168	2-D DFT	5.13	13.50
10x10	168x168	2-D DFT	5.19	17.63
11x11	168x168	2-D DFT	5.26	24.00
12x12	168x168	2-D DFT	5.33	34.56
13x13	168x168	2-D DFT	5.40	54.00
14x14	840x840	2-D DFT	5.46	96.00
15x15	840x840	2-D DFT	5.47	216.00

Table 5.3.7
Optimum Block size for 2-D Aperiodic convolutions calculated using either 2-D polynomial transform based convolutions or DFTs.

It is interesting to consider the number of additions performed. Notice that the convolution algorithms use many more additions per point. For example contrast the number of additions per point for the 8x8 and 9x9 windows.

5.4 Summary of Chapter 5

This chapter has compared the number of operations for a variety of functions using algorithms derived in the earlier chapters. The conclusions reached about the relative computational merits of these algorithms were as follows.

For the one-dimensional DFT the Winograd Fourier Transform algorithm (WFTA) was shown to use the smallest number of multiplications per point when compared with the Prime Factor Algorithm (PFA) and the Fast Fourier Transform (FFT). The number of additions may be decreased at the expense of more multiplications by the use of the PFA. The WFTA has the form of (1.20) with A and C matrices containing only +1, -1 and 0.

Two methods were compared for the calculation of one-dimensional cyclic convolutions. Firstly 'short-N' convolution algorithms were nested together, secondly the WFTA was used, taking advantage of the Cyclic Convolution Property (CCP). For up to about 90-point cyclic convolutions the nested 'short-N' convolution algorithms use less multiplications than the WFTA technique. This comparison assumes that one of the input sequences is fixed and that two real convolutions are computed per complex DFT. All the nested one-dimensional convolution algorithms have the form of (1.20). For the minimum number of multiplications convolution lengths containing factors 7, 9 and 16 will not have A and C matrices containing only +1, -1 and 0. Whilst it has proved possible to construct suitable algorithms with A and C matrices containing only +1, -1 and 0 for the 9- and 16-point convolutions the number of multiplications used is not the minimum. It has not proved possible to derive a suitable 7-point convolution algorithm.

Turning to two-dimensional functions, two ways of computing two-dimensional DFTs were compared. The two-dimensional DFT algorithms derived by Nussbaumer and Quandalle using polynomial transforms were shown to be superior to the nested one-dimensional WFTA. These two-dimensional DFTs have the general form of (1.20) and it appears that many of them will have A and C matrices containing only +1, -1 and 0.

Three methods were compared for the computation of two-dimensional cyclic convolutions. They were, computation using two-dimensional DFTs, calculation based upon nested one-dimensional convolution algorithms and finally the use of the two-dimensional convolutions derived using polynomial transforms (see chapter 3). The nested one-dimensional convolutions may be quickly discarded as not offering a good solution. For cyclic convolutions up to 60x60 points in size the polynomial transform based convolution algorithms offer the best choice in terms of numbers of multiplications used per point. For cyclic convolutions greater than this size the use of 2-D DFT algorithms requires less multiplications per point. As noted in chapter 3 these 2-D convolution algorithms are of the general form (1.20) and many have A and C matrices containing only +1, -1 and 0.

Finally the optimum block size for the aperiodic convolution of a small window, up to 15x15-points, with a large picture was considered. For windows up to and including 8x8-points two-dimensional polynomial transform based convolution algorithms should be used. Above this size use two-dimensional DFT algorithms.

Throughout this chapter many of the optimal algorithms have had the form of (1.20) with A and C matrices containing only +1, -1 and 0. This chapter concludes the first, and more theoretical, part of the thesis. The next chapter forms the start of the second part and deals with the implementation of algorithms of the general form of (1.20) which have A and C matrices containing only +1, -1 and 0.

Chapter 6

Implementing the WFTA using one-bit Systolic Arrays

In the field of digital signal processing there is a constant demand for higher and higher performance implementations of functions such as convolution and the Fourier Transform. New implementations must not only have increased performance but also consume less power and cost less! The advent of Very Large Scale Integration (VLSI) allows some of these goals to be achieved.

In high speed digital signal processing performance is gauged by the attainable throughput rather than the total time required for a function. By the use of pipelining techniques processing may proceed concurrently with input and output so that the throughput rate will be limited by the delay of the stages of the pipeline. The delay associated with the stages of the pipeline may be reduced by parallel processing in each stage of the pipeline.

The systolic arrays proposed by Kung and Leiseron [6.9] are an excellent example of such a pipelined, parallel system. In general a systolic array may be thought of as a one- or two-dimensional array of identical processing elements arranged in a regular fashion. Each module in the array is connected only to it's nearest neighbours. In the original arrays proposed by Kung and Leirseron each cell operated at the word level and the relevant circuits tend to consist of multiplier accumulator processors. However McCanny and McWhirter [6.1] have proposed a systolic array pipelined at the bit level suitable for matrix vector multiplication and a pipelined systolic multiplier. These arrays have a throughput rate limited only by the time to perform a one-bit full addition.

This chapter proposes a way of implementing algorithms of the general form of (1.20) which have A and C matrices containing only ± 1 and 0 based upon these one-bit systolic arrays.. The general form of this calculation may be described as follows,

1. The input sequence y is multiplied by the MxN A matrix which contains only ± 1 and 0.

2. The M values resulting from the product Ay are multiplied point-by-point by the M coefficients of the precalculated Bh. This only involves general complex multiplications for complex cyclic convolutions.

3. Finally the M values from the point-by-point multiplication are multiplied by the NxM C matrix, which again only contains ± 1 and 0, giving the N values of the output sequence y.

6.1 Proposed Architecture

The architecture proposed to implement these algorithms is based upon the three stages of the calculation outlined above. In this calculation there are two separate problems, firstly the matrix multiplication by the A and C matrices, secondly the multiplication by the fixed coefficients in stage two above. The circuits that are proposed here pipeline these calculations at the bit level to ensure maximum throughput and parallelism.

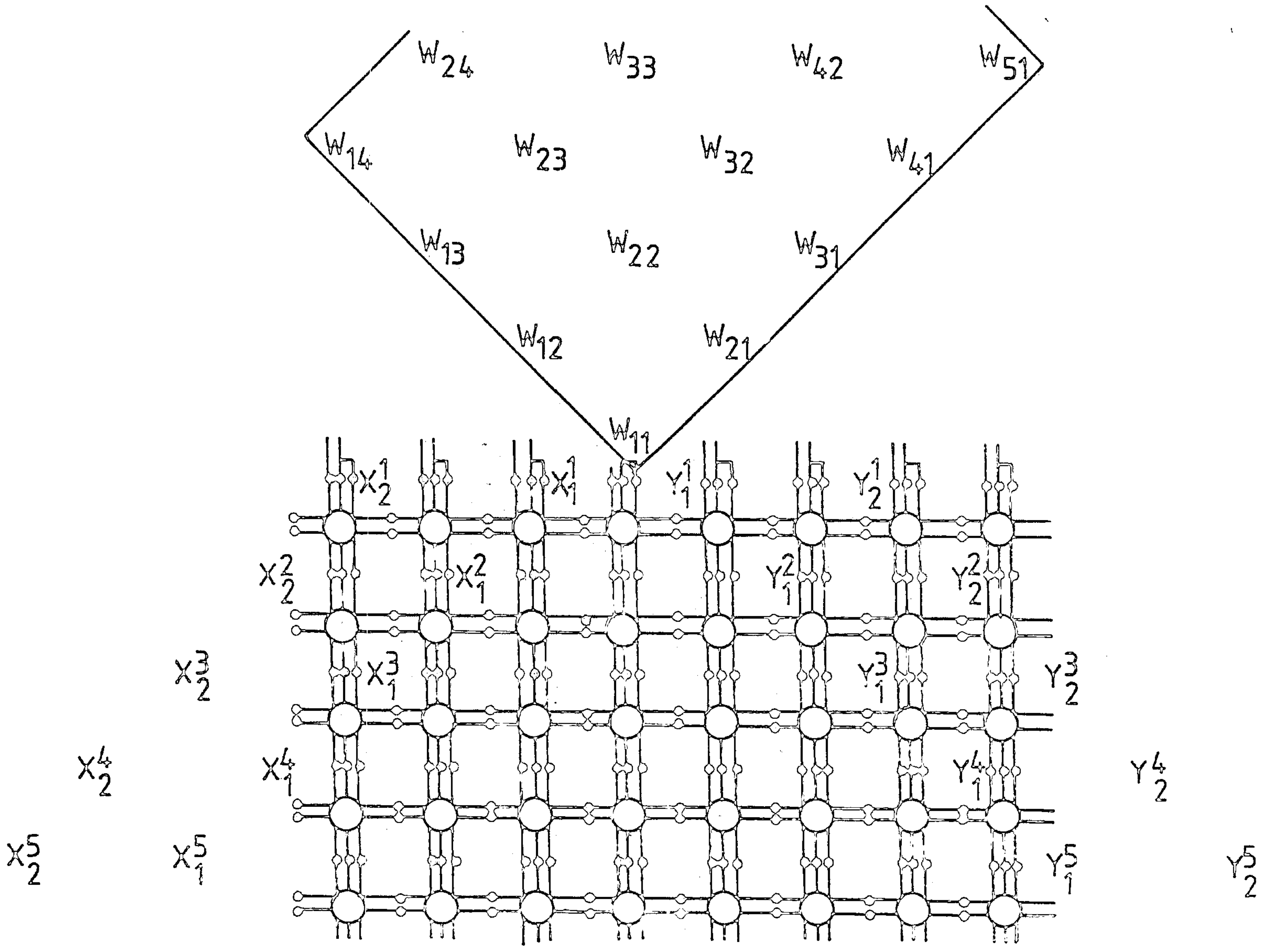
6.1.1 A and C Matrix Implementation

McCanny and McWhirter [6.1] describe a pipelined bit-slice transform array which will perform matrix vector products of the form Tx=y where T is a NxN matrix with one-bit coefficients. Ward and Stanier [6.4] have proposed a way of extending the basic cell given by McCanny and McWhirter to perform two's complement arithmetic with two-bit coefficients. The values ±1 and 0 require two bits to represent them and are shown as the inputs U and V to the extended basic cell which is illustrated in figure 6.1.1. The extended cell is a one-bit full adder with extra logic which complements the input bit, $x^{(j)}$, to the adder if the coefficient is -1, leaves the input bit unaltered if the coefficient is +1 or makes the input bit 0 if the coefficient is 0.

Figure 6.1.1 illustrates an array of these cells to perform the matrix product $Wx=y$ with W being a 5 (rows) x 4 (columns) matrix. This diagram is drawn using the convention adopted by McCanny and McWhirter with heavy dots representing latches and open circles representing the basic processing elements. The overall operation of the array is as follows: Data words, x_i, are input to the array from the left on every alternate cycle with sucessive bits staggered by one cycle. The input words move one cell to the right on every clock cycle. The input words are be sign extended to the maximum range of the answer, i.e. the number of rows in the array, upon entry into the array.

Bits representing the elements in the matrix W are organised so that they move down the array in a vertical direction. Whenever a coefficient of value -1 enters the array, 1 is added to the least significant bit of the input word via the 'carry-in' of the appropriate cell in the top row of the array and that input word x_i is complemented bit by bit as it moves through the array.

The output words, y_i, are initialised to zero on entering the array and move from right to left. Their bits are staggered by one row per bit, as are the input words x_i. This means that the kth bit of a word, $y_i^{(k)}$, meets all the terms required to form the sum



W	UV
+1	10
-1	01
0	00

$R \leftarrow XU + \bar{X}V$

$Y' \leftarrow \bar{R}\bar{C}Y + \bar{R}C\bar{Y} + R\bar{C}\bar{Y} + RCY$

$C' \leftarrow RC + RY + CY$

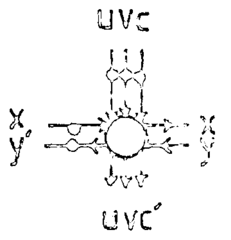


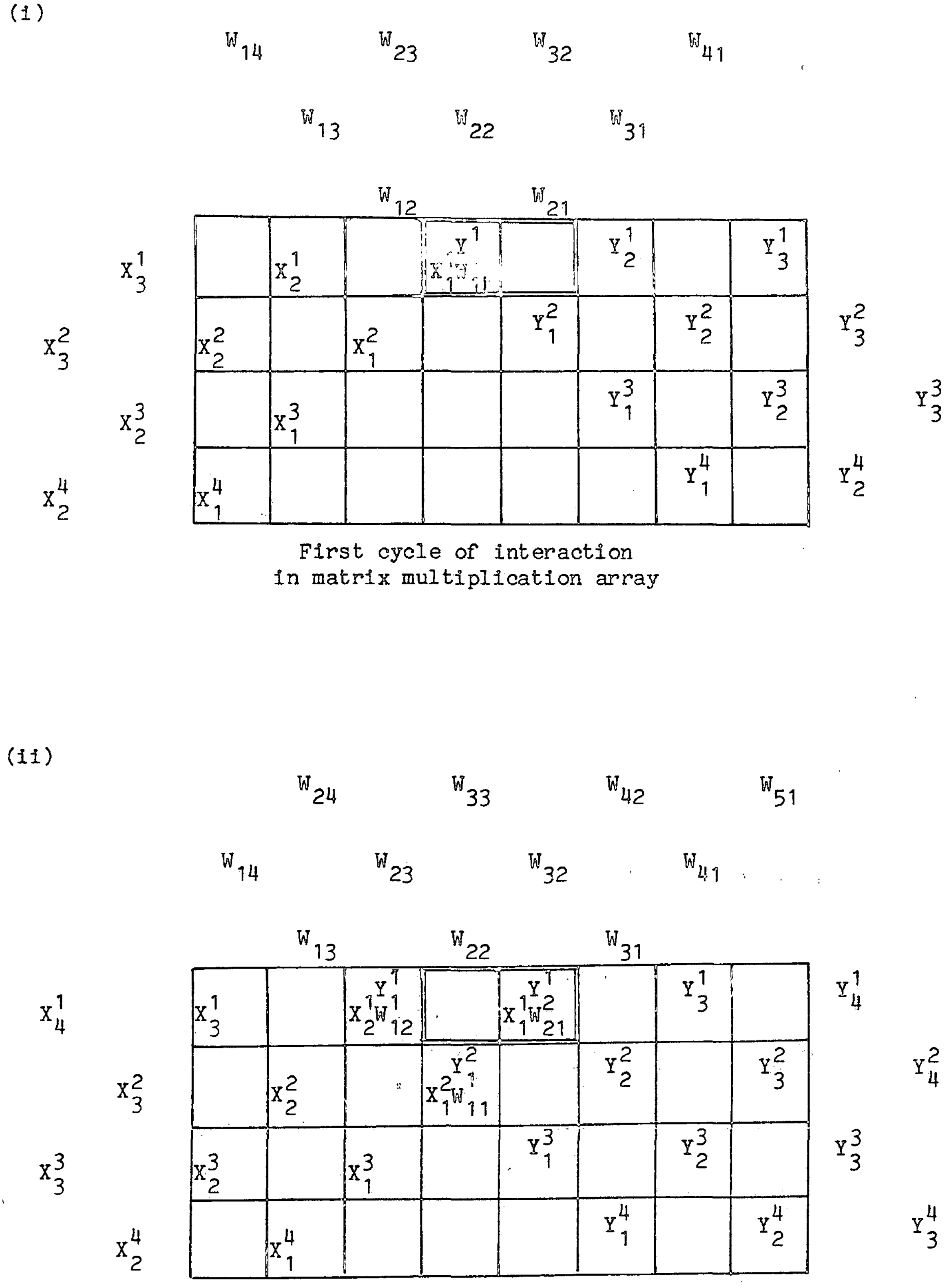
Fig 6.1.1 A & C Matrix Arrays

$$y_i^{(k)} = \sum_{j=1}^{n} W_{ij}x_j^{(k)} \tag{6.1}$$

the product $W_{ij}x_j^{(k)}$ being formed by the 'extra' logic in the cell which complements, leaves unaltered or zeroes the input to the one-bit full adder. Any carry bits which are generated in the course of this summation are latched vertically downwards as shown. The 'carry save' principle is used - this is the reason for the stagger on the bits of the words x_i and y_i. Having traversed the array the matrix vector product is complete with the output words emerging every alternate clock cycle, still in skewed form.

McCanny and McWhirter [6.3] have recently proposed several improvements to their original array. Firstly they note that as words enter and leave the array on alternate clock cycles only half the cells, on average, in the array are in use at any moment. This may easily be seen by reference to figure 6.1.2 which illustrates the first four cycles of interaction in these arrays. The layout of latches and cells has been omitted for clarity. McCanny and McWhirter propose that pairs of adjacent cells are coalesced into one with some extra data paths to preserve the regular flow within the array. This halving of the number of processors is considered in more detail in chapter 8. Here we propose that two sets of data could be interleaved in the unmodified array with two separate transforms being computed simultaneously.

(i)


(i)
W_{14} W_{23} W_{32} W_{41}
W_{13} W_{22} W_{31}
W_{12} W_{21}
X_3^1 X_2^1 Y^1 X_1W_{11} Y_2^1 Y_3^1
X_3^2 X_2^2 X_1^2 Y_1^2 Y_2^2 Y_3^2
X_2^3 X_1^3 Y_1^3 Y_2^3 Y_3^3
X_2^4 X_1^4 Y_1^4 Y_2^4
(ii)
W_{24} W_{33} W_{42} W_{51}
W_{14} W_{23} W_{32} W_{41}
W_{13} W_{22} W_{31}
X_4^1 X_3^1 Y^1 $X_2^1W_{12}^1$ Y^1 $X_1^1W_{21}^2$ Y_3^1 Y_4^1
X_3^2 X_2^2 Y^2 $X_1^2W_{11}^1$ Y_2^2 Y_3^2 Y_4^2
X_3^3 X_2^3 X_1^3 Y_1^3 Y_2^3 Y_3^3
X_2^4 X_1^4 Y_1^4 Y_2^4 Y_3^4

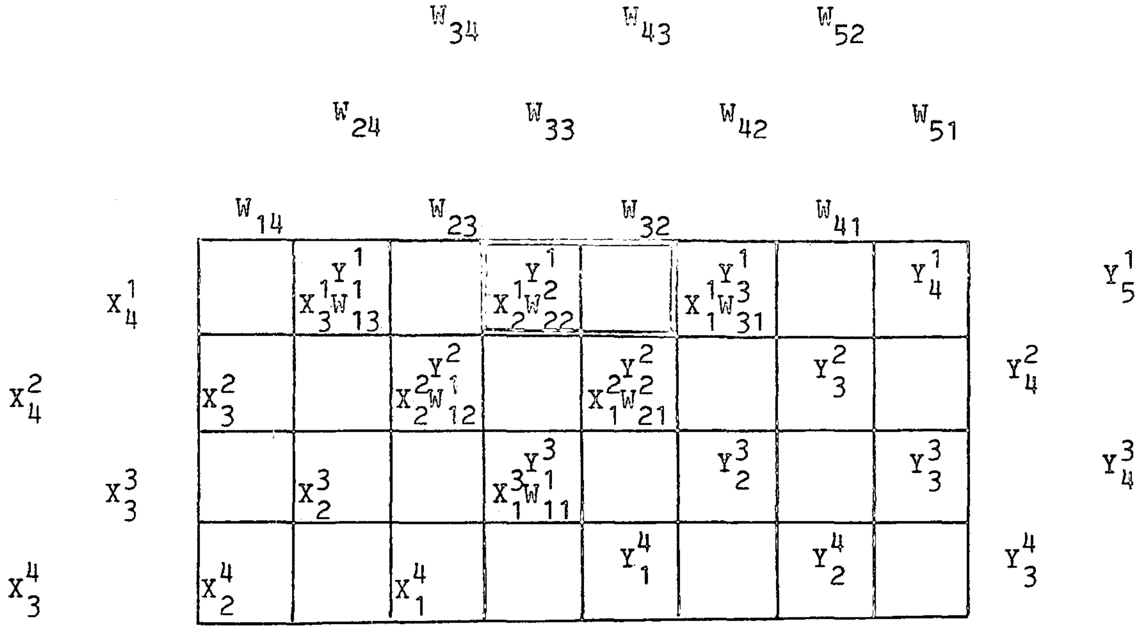
First cycle of interaction
in matrix multiplication array

(ii)

Second cycle of interaction
in matrix multiplication array

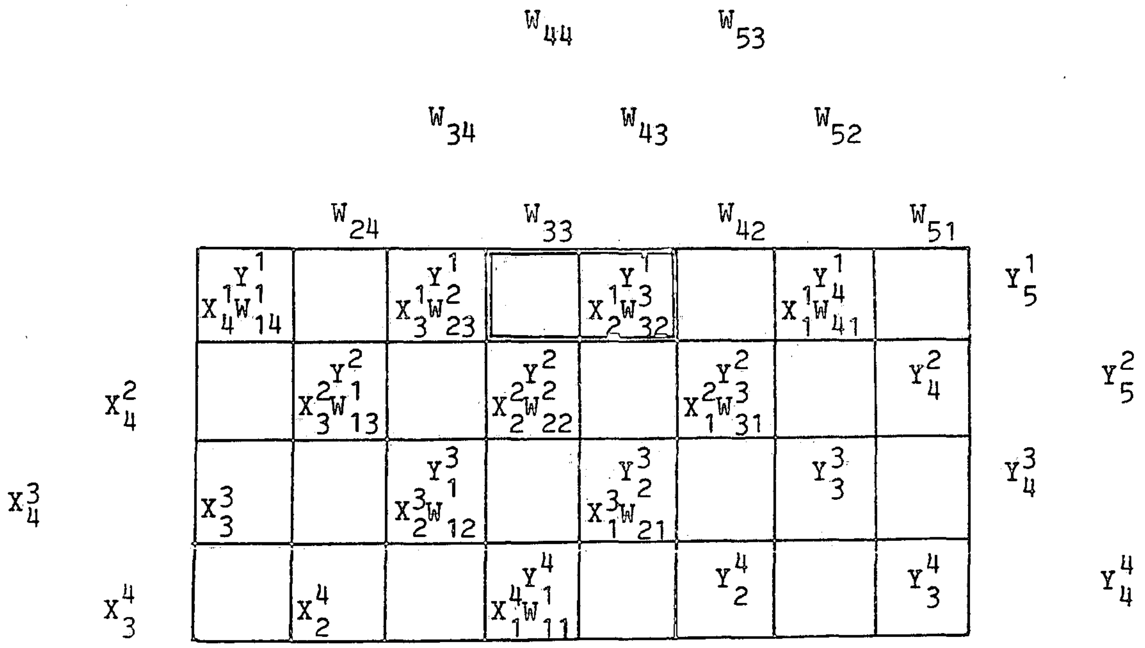
Figure 6.1.2

(iii)


W_{34} W_{43} W_{52}
W_{24} W_{33} W_{42} W_{51}
W_{14} W_{23} W_{32} W_{41}
X_4^1 $Y_3^1 \; X_3^1W_{13}^1$ $Y_2^1 \; X_2^1W_{22}^2$ $Y_1^1 \; X_1^1W_{31}^3$ Y_4^1 Y_5^1
X_4^2 X_3^2 $Y_2^2 \; X_2^2W_{12}^1$ $Y_1^2 \; X_1^2W_{21}^2$ Y_3^2 Y_4^2
X_3^3 X_2^3 $Y_1^3 \; X_1^3W_{11}^1$ Y_2^3 Y_3^3 Y_4^3
X_3^4 X_2^4 X_1^4 Y_1^4 Y_2^4 Y_3^4

, Third cycle of interaction
in matrix multiplication array

(iv)


W_{44} W_{53}
W_{34} W_{43} W_{52}
W_{24} W_{33} W_{42} W_{51}
$Y_4^1 \; X_4^1W_{14}^1$ $Y_3^1 \; X_3^1W_{23}^2$ $Y_2^1 \; X_2^1W_{32}^3$ $Y_1^1 \; X_1^1W_{41}^4$ Y_5^1
X_4^2 $Y_3^2 \; X_3^2W_{13}^1$ $Y_2^2 \; X_2^2W_{22}^2$ $Y_1^2 \; X_1^2W_{31}^3$ Y_4^2 Y_5^2
X_4^3 X_3^3 $Y_2^3 \; X_2^3W_{12}^1$ $Y_1^3 \; X_1^3W_{21}^2$ Y_3^3 Y_4^3
X_3^4 X_2^4 $Y_1^4 \; X_1^4W_{11}^1$ Y_2^4 Y_3^4 Y_4^4

Fourth cycle of interaction
in matrix multiplication array

Figure 6.1.2 Continued

One application requiring two simultaneous matrix multiplications occurs in the calculation of DFTs with complex data when using the algorithms of the form of (1.20). This would require the input sequence to be entered into the array with the real word followed by the imaginary, i.e. giving a sequence of real, imaginary, real, imaginary,... .

The second improvement to the array is to notice that the output words do not have to be initialised to zero as they enter the array from the righthand side. These words may be initialised to any value which is to be included in the final result of the matrix multiplication. If the output of the array is truncated a form of rounding may be introduced by initialising the output words to the average value of the discarded bits.

It is of interest to note that an array of these cells is capable of performing a Walsh transform with no other hardware.

To summarise this subsection, the A and C matrix multiplications can be performed by an array of modified one-bit adder cells connected together orthogonally. The interconnection pattern and logic for these cells is illustrated in figure 6.1.1. The exact size and partitioning of these arrays is discussed in section 6.2 below.

6.1.2 Pipelined Systolic Multipliers

McCanny and McWhirter [6.1,6.2] describe a two's complement pipelined systolic multiplier based upon an array of one-bit full adder cells and bit staggered inputs. This multiplier architecture is illustrated in figure 6.1.3. The two input words to this multiplier are staggered in different manners. One input word, described as $b_1(n)$ in figure 6.1.3 is staggered least significant bit first, whereas the other input word, $a_1(n)$, is staggered most significant bit first.

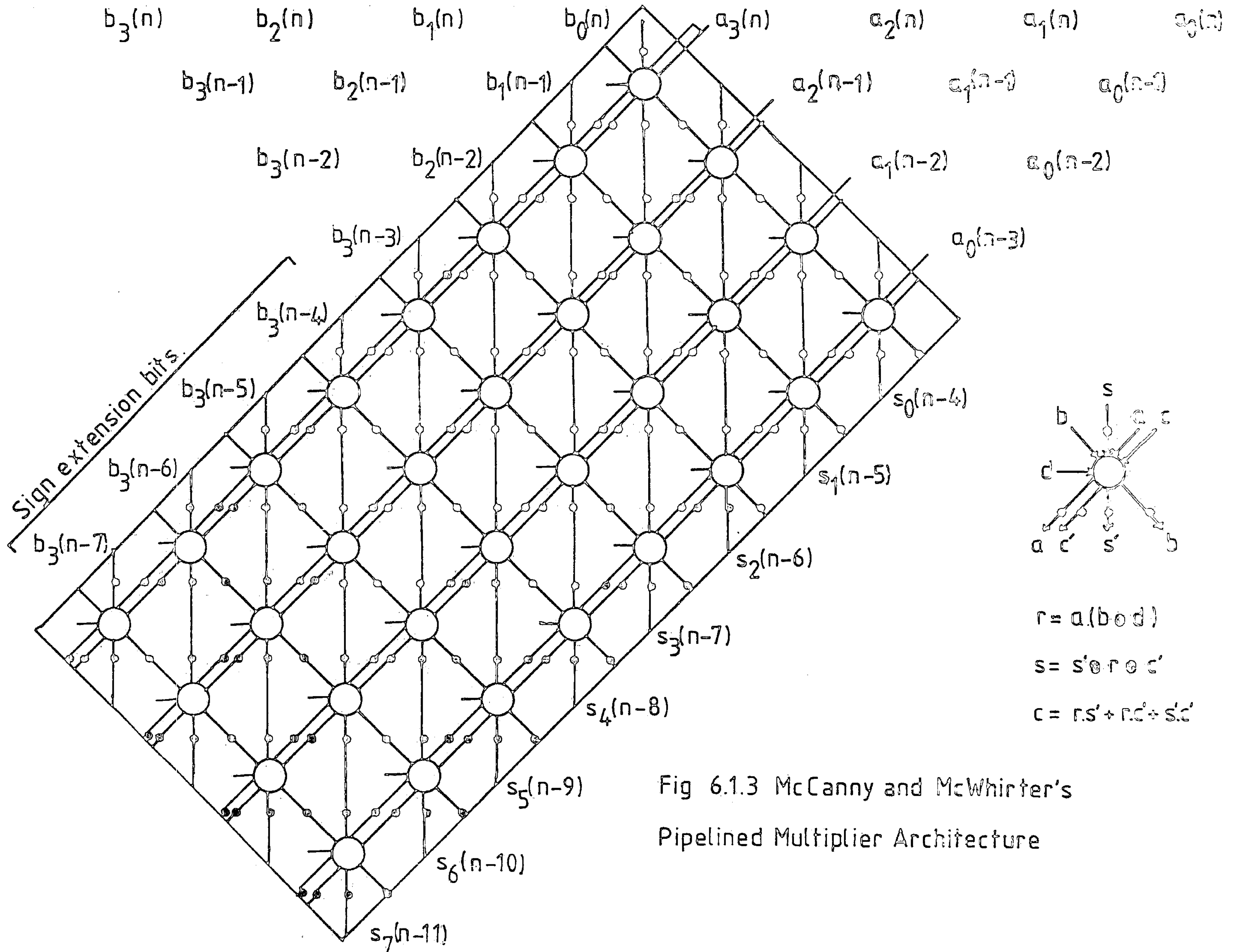


Fig 6.1.3 McCanny and McWhirter's Pipelined Multiplier Architecture

Furthermore this architecture, which is completely regular, is capable of accepting words every clock cycle. In order to achieve the two's complement operation the cells along the lefthand upper boundary have their 'd' input set to '1'. All other cells in the array have a 'd' input of '0'. Note that the value of $a_3(n)$ is fed to the carry input of the top cell, all other carry inputs are initialised to zero.

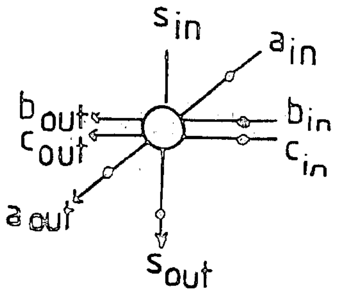
Figure 6.1.3 illustrates a 4x4 bit multiplier using an 8x4 array of cells. Notice that in this diagram the three most lefthand columns of cells do not contribute to the final result, their function is to delay the sign extended bits to the correct significance. This triangle of cells could be replaced by latches. The total number of cells in this truncated array for an NxN bit multiplication would be $\frac{1}{2}N(3N+1)$.

Another possible pipelined parallel two's complement multiplier architecture is given by Myers [6.8]. This is illustrated for a 4x4 bit example in figure 6.1.4 which is a corrected version of fig 4.1.2.1 in [6.8]. The total number of cells for a NxN bit multiplication using this architecture is N(N+1). Note that the cells along the bottom row have a slightly different logic function to the other cells with A and B being NANDed instead of ANDed together. The 'carry-in' of the bottom row of cells is set to '1' rather than the '0' of the other rows of cells.

Both architectures have the same potential throughput rate limited by the time taken for a one-bit full addition.

Many of these features may be used to advantage. Notice that the stagger on the output words of the transform arrays is exactly the same as the $b_i(n)$ input word stagger for McCanny and McWirter's multiplier. Thus their multiplier and the A matrix array may be

Basic Cell



Logic

$a_{out} = a_{in}$

$b_{out} = b_{in}$

$c_{out} = (ab)c_{in} + (ab)s_{in} + c_{in}s_{in}$

$s_{out} = (ab) \oplus s_{in} \oplus c_{in}$

Bottom Row Only

$c_{out} = (\overline{ab})c_{in} + (\overline{ab})s_{in} + c_{in}s_{in}$

$s_{out} = (\overline{ab}) \times s_{in} \times c_{in}$

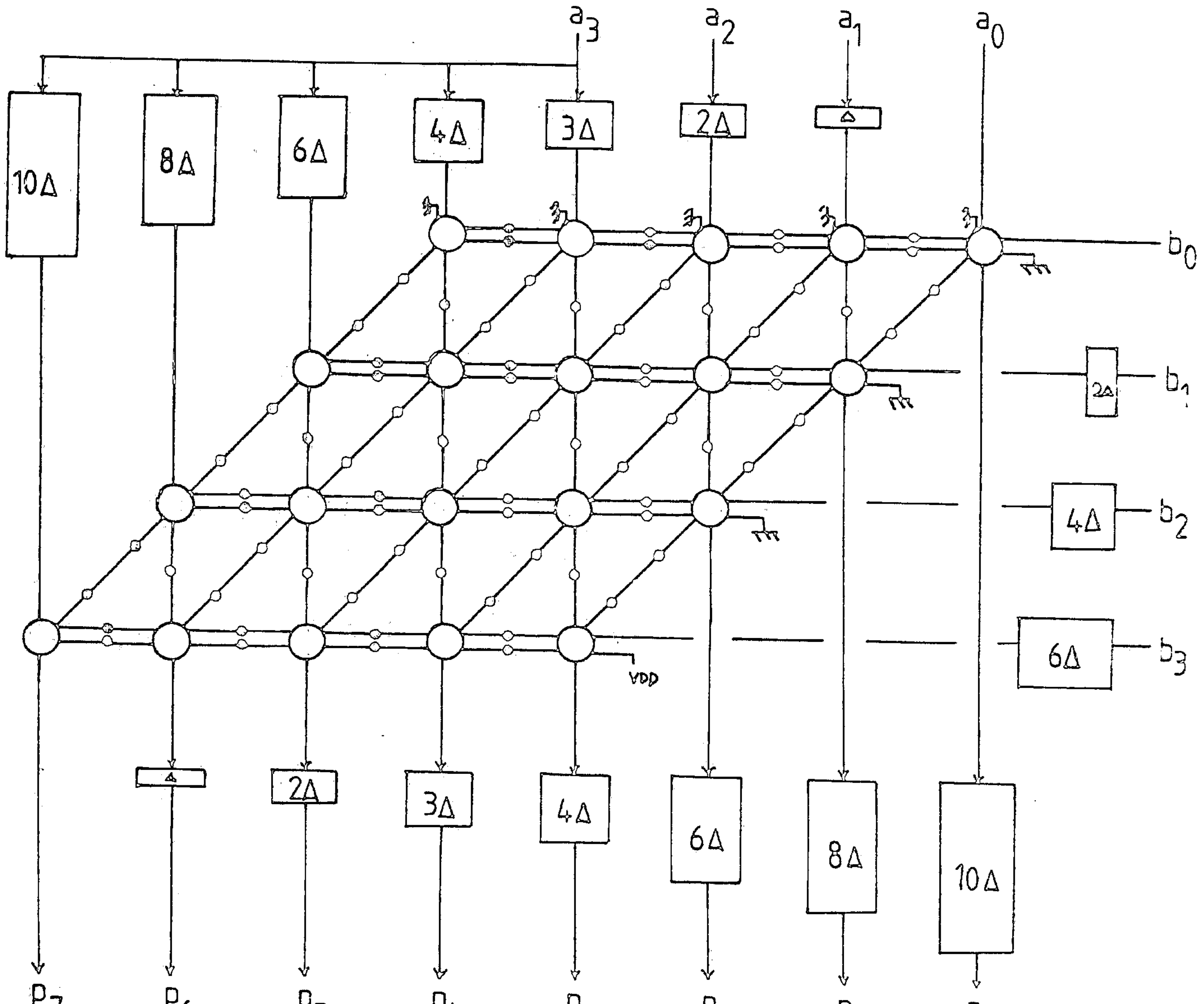


Fig 6.1.4 Myers' Pipelined Multiplier Architecture

connected directly together. The stagger on the output from their multiplier is the same stagger required by the input words for the transform arrays. This implies that their multiplier and the C matrix transform array may be connected without any need to alter the stagger of the bits. These least significant bit first staggers may easily be accomodated in Myers' multiplier by the omission of some of the latches in figure 6.1.4.

As the multipliers are capable of operating every clock cycle it is able to multiply the interleaved real and imaginary words from the A matrix array by a purely real or purely imaginary coefficient, such as found in the WFTA. In order that a system employing two matrix multiplication arrays and one multiplier maybe connected directly together a technique for reordering the real and imaginary words is needed whenever the complex word has been multiplied by an imaginary coefficient. This reordering should occur either between the multiplier and the C array or between the A array and the multiplier. A method of achieving this reordering is discussed in section 6.2.3 below. When dealing with 'real' only functions, for example convolutions, there is no need to have this word swap procedure.

A very slight simplification to the multipliers is possible by forcing the coefficients, $a_i(n)$ in figure 6.1.3, to be positive. This may be achieved by either altering the signs of a row of the A matrix or the signs of a column of the C matrix. The net effect of this is to require a multiplier capable of multiplying positive by negative or positive numbers rather than a general two's complement multiplier.

6.2 Practical Implementation of these Algorithms

The overriding limitation of any implementation based upon these one-bit full adders is the number of cells in the arrays. The silicon area occupied by these arrays is large. Whilst it is possible to split the arrays, problems are immediately encountered with pinout limitations. These problems are discussd in more detail below, firstly for the A and C arrays and then for the multiplier.

6.2.1 The A and C Matrix Arrays

This section and it's associated subsections consider the dimensions of the arrays for the A and C matrices in terms of numbers of cells and in area. The problems of pinout and coefficient storage are then discussed.

6.2.1.1 Matrix Multiplication Array Sizes

For a matrix, containing only +1, -1 and 0, of dimensions N (rows) x M (columns) a total of N+M-1 columns of cells in the array are required. Thus the A and C arrays require the same number of columns. The number of rows in the arrays is governed by the maximum word length of the accumulated sums. In the worst case the addition of N two's complement numbers, each of which may have been multiplied by -1, gives a word length growth of $\mathrm{Log}_2 N+1$ bits. The reason for the additional bit is as follows. A b-bit two's complement number represents a number in the range

$$-2^{b-1} \text{ to } 2^{b-1}-1$$

when multiplied by -1 a number in the above range falls into the range

$$-2^{b-1}+1 \text{ to } 2^{b-1}.$$

Notice that the upper limit of this new range is outside the limit for representation as a b-bit two's complement number.

Starting with b-bit inputs the size of the arrays for the A and C arrays should be:

A matrix $(b+Log_2N+1)$ x $(N+M-1)$ cells (6.2)

B matrix $(b+Log_2M+1)$ x $(N+M-1)$ cells (6.3)

Table 6.1.1 below lists the number of multiplications and the number of columns for various DFT lengths, based upon the WFTA. The number of rows is the maximum word length in the C array, based upon 16-bit input data. The longer transform lengths are constructed by the nesting techniques of chapter 4.

Transform Length	Number of Multiplications	Number of Columns	Number of Rows
2	2	3	18
3	3	5	19
4	4	7	20
5	6	10	20
8	8	15	20
9	11	19	21
11	21	31	22
13	21	33	22
15	18	32	22
16	18	33	22
30	36	65	23
60	72	131	24
120	144	263	25
240	324	563	26
504	792	1295	27
1008	1792	2798	28

Table 6.1.1
Numbers of Rows and Columns of Cells for WFTAs with 16-bit data

Table 6.1.1 shows that the number of columns in the arrays quickly grows. For transforms lengths in the region 100 to 1000 the number of columns is approximately 2.5N.

The number of columns in the arrays may be reduced slightly. Look at figure 6.1.1. If value of the coefficient W_{14} is 0 then the lefthand most column of the array never contributes anything to the calculation. In a similar fashion if all the coefficients in a particular column are zero then that column may be omitted from the array.

By rearranging the rows of the A matrix and the columns of the C matrix the maximum number of zeroes may be forced into the corners of the array. For the larger length transforms the possible savings are small. If it is acceptable to have the input and output data sequences in a non-natural order then the possibilities for saving columns from the arrays are much larger.

Clearly, in any implementation the size of the basic cell is of great importance. The following is a rough guide to the cell size.

6.2.1.2 Basic Cell Area

A suitable cell for the A and C arrays was designed in 5µ NMOS. The dimensions of this cell were 300µm x 220µm (width x height). Whilst this design was fairly well compacted the design was not as small as it might have been. Furthermore, because of other design considerations, it was not desirable to have a cell narrower in width at the expense of increased height. Thus using 3µm design rules it might not be unreasonable to expect a basic cell size of 150µm x 160µm. Patel et al.[6.6] and McCabe et al.[6.7] show a design in 3.5µm CMOS-on-Saphire for a correlator cell which measures 260µm x 240µm. This correlator cell is similar to the cell needed for these arrays. However their design does not appear to be particularly compact with all the latches being of a standard design. Further decreases in cell area may be possible by the use of two layer poly-silicon or metal processes. In particular two layers of metal would reduce the area occupied by the clock and power connections to each cell.

6.2.1.3 Matrix Array Chip Size and Pinout

Now consider the number of cells that it would desirable to integrate onto a single chip. If a flexible component is desired one chip should be able to cope with the A and C matrices for small transforms, say up to 16-points. The same component should also serve as the basis for the much larger arrays of longer transform lengths. The short transform length condition requires at least 33 columns of processors and the long transform word length dictates about 27 rows. The 27 rows would allow accumulations of up to 2048 values starting from 16-bit data. An array of 33 x 27 cells would occupy 4.95mm x 4.32mm, using the cell size projected above.

Whilst such an array is possible just on the grounds of silicon area, the pin-out of such a chip must be considered carefully. In order that the chip may be used to build up longer arrays each row requires 4 pins, 2 for input and 2 for output. Each of the columns needs 1 pin to input it's coefficients. This is achieved by noting that as each two-bit coefficient is used for two succesive cycles it may be input via one pin during two cycles. This gives a pin count of 141 for the 33 x 27 array, excluding power supplies, clocks, etc. Such a pin count is high, but not impracticable.

In the case where the chips are being used to create long arrays, the 27 rows might be connected directly chip-to-chip if the whole multi-chip array was mounted on a substrate. If the chip were being used only for small transforms the right hand edge of the array would not need to be bonded onto external pins, with the inputs being directly bonded to ground. This would reduce the need for large multi-pin packages and chip carriers.

6.2.1.4 A and C Matrix Storage and Entry

As noted above each of the of the array columns requires two pins, or one pin used for two cycles, for the input of it's coefficients. The use of off chip storage for the A and C matrices gives the greatest flexibility at the expense of extremely wide ROM. Any ROM supplying the coefficients would need to be (N+M-1) bits wide. Despite the great width of this ROM, design would be comparatively simple as the addressing could be performed by a single input pin driving a counter.

The alternative would be to provide the ROM storage on the same chips as the array cells. Reference to figure 6.1.1 shows that the maximum number of values that could be entered down a column of cells is (N+M-1). For other than comparatively short transform lengths the amount of on-chip ROM becomes large. As an example consider the amount of on-chip ROM required for the 33 x 27 array proposed above to be capable of being used to build up a 1008-point transform. The total on chip storage would need to be 33 x 2 x (1008 + 1792 - 1) = 184,734 bits. It is certainly possible to store all these values on a single chip. However it seems doubtful that a sufficiently large and fast ROM together with the array of processor cells could be integrated on a single chip using current technology. On-chip storage of coefficients may be feasible for more moderate transform lengths.

It is important to notice that when no coefficient value is being entered into the array the coefficient inputs should represent 0. That means that the leftmost and rigthmost columns of cells in figure 6.1.1 will only receive a non-zero input on one cycle per transform. This condition ensures that sucessive transforms do not interfere (see section 6.3 for a note on delays between transforms).

6.2.1.5 Summary of A and C Matrix Implementation

The way the matrix multiplication arrays are implemented depends upon the type of solution that is being aimed at. If the goal is flexibility the best course would be integrate as many cells as possible onto one chip, providing a long word length with no on-chip coefficient storage. If, however, the goal is a minimum chip count, then the policy should be to provide only the minimum acceptable word length and store all the A and C matrix coefficients on-chip. For some image processing problems this may be the best course. The provision of 33 columns of cells would provide an excellent basis for a system to implement a 15-point Discrete Cosine Transform using the algorithm given by Ward and Stanier [6.5], see chapter 8.

In terms of ease of development, the flexible approach has much to recommend it. For example commercially available ROM, or PROM, could be used initially rather than custom designed wide ROMs. (This does put the chip count up rather spectacularly!)

6.2.2 A Pipelined Systolic Multiplier

The multiplier architecture proposed by Myers [6.8] requires an array of b x (b+1) cells to implement the multiplication of two b-bit words. For the relevant case of 16-bit data, this is 16 x 17 cells. The basic cell is a one-bit full adder with some extra latches. The cell is very similar to that used in the matrix multiplication arrays. It should be possible to integrate all of this multiplier onto a single chip. However both pipelined multiplier architectures may be cascaded to build up word length if needed. However the pinout requirement quickly grows.

Storage is needed to hold the M coefficients for the multiplier. The possibilities fall into two categories. On- or off-chip storage. Off-chip storage requires the multiplier to have 4N pins (64) for data input and output - excluding clocks etc, assuming untruncated output, for 16-bit data. On-chip coefficient storage reduces the chip count but decreases flexibility; the amount of room left on the chip after design of the multiplication array will limit the size of the ROM and hence the maximum transform length. Any on-chip ROM should be as large as possible. Perhaps the most convenient solution would be to have a separate custom designed ROM addressed by a counter that was directly bonded to the multiplier within the same package.

The amount of ROM required by the multiplier is Mx(b+1) bits. The extra bit is to flag real/imaginary for each coefficient. This requires 5,508 bits for a 240-point WFTA with 16 bit coefficients.

6.2.3 Data Skew and Word Order Swapping

Before entry into the A matrix multiplication array and after leaving the C array the data needs to be skewed and deskewed respectively. The same design may be used to achieve both functions. It may be done by a triangular array of latches. Such an array would need b pins input and b pins output. To aid the development of the system and to allow for future word length growth it may be convenient to have an array of latches whose width is at least the word length of the matrix multiplication arrays.

As mentioned above, if the technique of interleaving real and imaginary words is used in a DFT, some method of interchanging the real and imaginary words is needed, either immediately before or after the multiplier. This interchange should occur each time the coefficient is imaginary. Figure 6.2.3 outlines a structure capable of doing this.

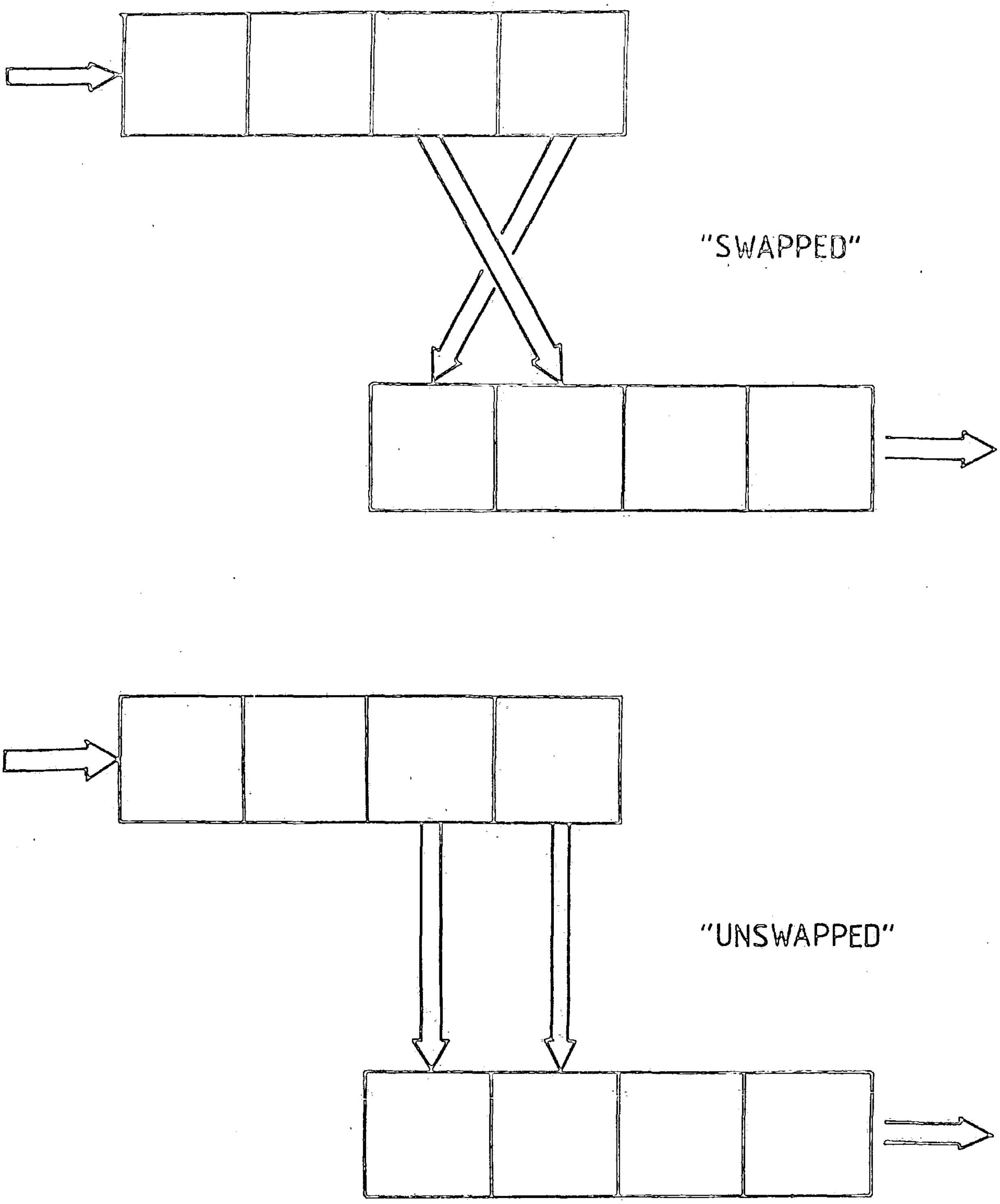


Figure 6.2.3

Word Swap Structure

Many other possibilities for doing this task exist and may be better. In order to keep the number of separate chip designs down it might be convenient to integrate this 'word-swapper' on the same design as the data skew array.

6.3 The Overall System

A possible layout of the complete system is show in figure 6.3. The data, in natural order, is input sequentially, with real and imaginary words interleaved, into the 'data skew' module followed by a gap of (M-N) zeroes. These (M-N) blank values have two functions. Firstly, as the A and C matrices are rectangular, time has to be allowed for the last accumulated sum using the last coefficient, W_{MN}, to move from the Mth column of the array past the Nth column where the first coefficient, W_{11}, enters for the first sum of the succeeding transform. This prevents the sums of two consecutive transforms from clashing. Secondly, whilst the multiplier has to perform 2M multiplications and is working continuously, only 2N words are entered into the system. Then, as the entire system will be clocked together, M-N blank values should be input to avoid swamping the multiplier.

After skewing, the data enters the A matrix array. The output from this array is fed directly into the multiplier, having been truncated back to the word length of the multiplier. Suppose the input word length and the word length of the multiplier were both b-bits. Now suppose A matrix array is D bits wide, $D>b+\mathrm{Log}_2N+1$, i.e. this array has a word length greater than the minimum. Then the $D-b-\mathrm{Log}_2N-1$ most significant bits should be discarded - they should all be the same, and the Log_2N least significant bits should be discarded to truncate the word length down to that of the multiplier. There are now M values in the sequence.

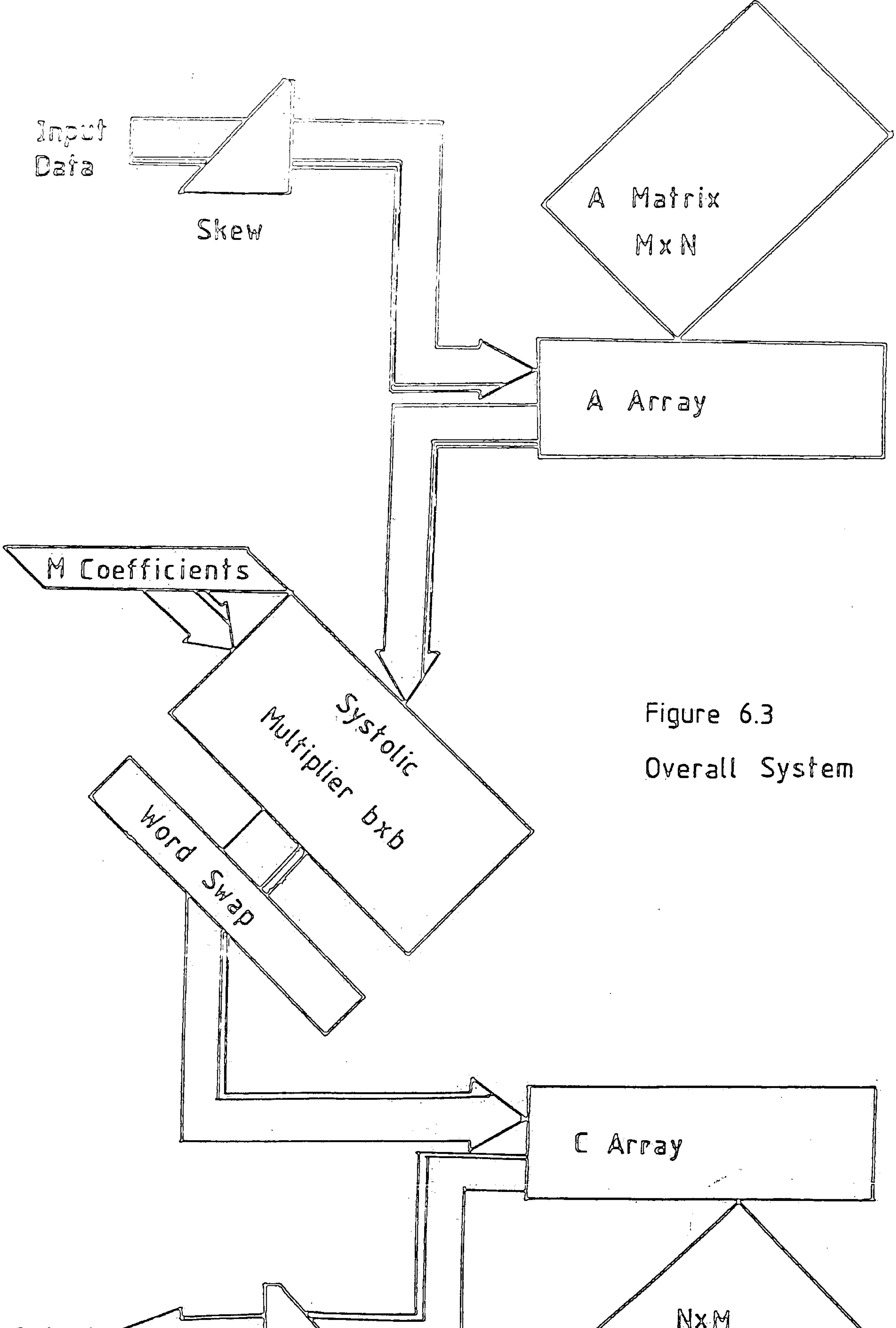


Figure 6.3

Overall System

If an array is built up from standard parts and has a number of columns greater than N+M-1 then the extra columns should be located at the righthand end of the array and the coefficient inputs tied to represent a coefficient value of 0. This will ensure that these extra columns have no effect on the operation of the array.

The truncated data words enter the multiplier where they meet the fixed coefficients. After leaving the multiplier real and imaginary words of a data value should be swapped if the coefficient is imaginary. This is shown as a separate function in figure 6.3 but could incorporated into another component. Then the M values enter the C array. N values leave the C array followed by M-N blank values. Finally, the N values are deskewed by a triangular array of latches. The transform results will be in natural order.

This architecture allows easy implementation of window functions. An extra multiplier and ROM could be inserted between the initial data skew array and the A matrix array. No loss of performance is entailed but the latency of the system will be increased.

6.3.1 Performance

As the basic cells of the transform arrays and the multiplier are very similar it will be possible to clock the entire system at the same rate. Since the basic cell is a derivative of a one-bit full adder the clocking rate may be quite high, perhaps 40MHz for 3μm CMOS [6.6]. This gives a an effective data rate for these systems of (N/M).20MHz.

6.3.2 Chip Designs

The different designs needed for this architecture are as follows:

1. Data Skew/Deskew. This is a triangular array of latches. The 'word swap' circuitry might be incorporated into this chip.

2. The matrix multiplication arrays. This might include on-chip ROM. If there is no on-chip ROM design 3 will also be needed.
3. A very wide ROM. As many bits width as possible, addressing performed by a counter driven by the system clock and reset at the start of each transform.
4. A pipelined systolic multiplier of b x b bits, perhaps with on-chip ROM if there is enough room.

As all these designs involve large arrays most of the design effort should go towards the layout of the basic cells. Since the arrays are clocked together and as the data flows regularly through the arrays very little control will be needed. The control circuitry may be summarised as follows,

1. A flag to signify the start of a transform. This control bit should propagate through the arrays with the data resetting the counters which address the coefficient storage ROMs.
2. Circuitry to ensure that M-N blank values (complex) are appended after each data set.

6.4 AnA Matrix Systolic Array Design

A 5μm NMOS integrated circuit was designed to perform the A matrix multiplication for a 6-point WFTA.

The size of the design was limited to 4mm x 4mm, including frame, bonding pads and a test strip. At most 40 interconnection pins were available. In view of these constraints it was decided to design a small A matrix multiplication array. So as to keep the external circuitry as simple as possible, all coefficients were to be stored 'on-chip'.

As mentioned above the basic cell had dimensions of 300µm x 200µm. This enabled an array of at most 10 processors in width to be put onto the chip. These 10 processors are sufficient to perform the A matrix multiplication for a 6-point WFTA. A 6-point WFTA requires 6 multiplications. As there is a zero in one corner of the matrix only 10 columns are needed.

Having designed the coefficient storage there was room for 8 rows of cells. This allows 4-bit input data and 4 bits of accumulation. However, by mistake, the chip was designed for 5 input bits. The coefficients are stored in a ROM which is loaded in parallel (i.e. all cells together) into a 20-bit wide shift register at the start of each transfrom. This shift register is then clocked downwards to enter the coefficients into the array.

There was not sufficient room to provide skewing or deskewing latches. The total chip contains approximately 4500 transistors and was estimated to dissipate about 0.4 Watts. There is a photograpgh of the design inside the rear cover. Unfortuneately none of the bonded devices did anything. This was due to a substrate biasing problem.

6.5 Summary of Chapter 6

This chapter has proposed an implementation of many of the algorithms from the previous chapters using systolic arrays of one-bit full adder cells. These systolic arrays offer several advantages. Firstly as they are composed of arrays of identical cells the design of these arrays should be straightfoward in comparison to other custom VLSI components. Secondly since these cells are only connected to their nearest neigbours there are no long word lines across the chip with their associated high capacitances to slow down the operation. Thirdly as the arrays are pipelined down to the bit level the maximum throughput rate is only limited by the time taken to perform a one-bit full add.

There are two basic components in this implementation. The arrays for performing the A and C matrix multiplications and the pipelined mutliplier. The A and C matrix arrays are capable of accomodating complex data for WFTAs without alteration. The size of these arrays limits the size of transform (or convolution) that may be performed. Indeed the most useful application of this technique may be to implement 'small-N' WFTAs and then to use a Prime Factor Type Algorithm to build up longer transforms. These 'small-N' transforms might be all implemented using the same chip set with different coefficients.

Chapter 7

A Cost Function for the Comparison of Algorithm Implementations

Is the implementation of DFTs and convolutions proposed in the previous chapter any 'better' than conventional implementations of the FFT? Do these one-bit systolic arrays offer any architectural advantages besides the regularity and ease of design mentioned previously?

This question is a particular example of the problem of comparing different ways of achieving the same signal processing function. Ward, Barton, Roberts and Stanier [7.1] have developed a cost function to make quantitative comparisons between digital algorithms implementations including control and overheads. This cost function provides a tool allowing different implementations of the same algorithm, and also different algorithms for the same function, to be compared. As an example of the use of this cost function five implementations of the FFT and four of the WFTA, including the one-bit systolic arrays of chapter 6, are compared.

This work was originally begun as a contribution to a joint industry-MOD working party on high performance logic. The work on the underlying assumptions on the number of logic gates involved and on the first four FFT implementations was provided by Dr P. Barton of STL [7.2].

7.1 Introduction to Cost Function

The implications of developing complex integrated circuits for real time digital signal processing include the need for a reappraisal of the algorithms which are most efficient for convolution, correlation, spectral analysis, beamforming etc. This is because the previously 'high-cost' operations of multiplication and data storage can now be realised with specialised, high speed, low cost, VLSI components.

As new algorithms and processing architectures evolve, simple arithmetic comparisons between them, such as those made in chapter 5, are no longer adequate. Hardware complexity, ease of control, physical size and power consumption must all be considered when comparing different realisations of a given processing function.

Consequently, it is important to measure algorithm effectiveness in a standard manner well related to the available hardware technologies. Similarly it is appropriate for each technology to be given a figure of merit, which, in conjunction with an algorithm efficiency, will indicate the 'cost' of realising a given processing function. Some algorithm implementation factors of practical importance are difficult to quantify, in particular the applicability of standard components rather than custom ICs is vital to a short term project. The ease of reprogramming major parameters in the process is frequently important, for instance the ease of changing the length of a discrete transform or convolution. The financial cost of the required components does not feature strongly in the rating of most high performance DSP designs and is expected to become increasingly less important as the component cost per function continues to fall. Furthermore we have chosen to ignore devclopment costs in our proposed model.

In section 7.2 a 'cost' function is derived which aims to separate the technology dependence from the algorithm efficiency and which is easily applicable to different types of implementation. Sections 7.4 and 7.5 show the cost function applied to implementations of the radix-2 FFT and Winograd Fourier Transform algorithm. These two sections illustrate why it is misleading to compare only the arithmetic complexities of digital signal processing algorithms.

7.2 The Cost Function

In order to keep algorithm properties separate from technology attainments we prefer to reduce each algorithm to an implied architecture measured in terms of the highest common technology independent factors. So that

$$\text{System Cost} = \text{Network Cost} \times \text{Technology Price} \qquad (7.1)$$

with, for example, the system cost measured in Watts, the network cost in GatesMHz and the technology price in Watts per gate per MHz. For current technologies these technology independent factors may be chosen as the number of logic gates, G, and the number of memory locations, B, integrally involved in the algorithm.

Some previous measures of complexity [7.3, 7.4] only considered logic elements and memory cells involved in the calculation. Here we reduce all features of a fixed algorithm, including control functions and address generation, to the cost elements of logic gates and memory locations.

The benefit of an implementation reduces to the available word throughput rate, R, measured at the input, for a specified word length. To make this independent of technology it has to be normalised by the gate speed and memory access time. Stating the 'cost' of an implementation as the number of logic gate operations and memory accesses needed should be regarded as the cost of the algorithm without regard for the architecture, whose inefficiency is measured by the proportion of time during which gates do not potentially change state and memory is not accessed.

In practice an algorithm is encapsulated within the architecture used to realise it. We therefore find it easier to discuss the cost function of the combined entity which we term a network.

Network = Algorithm + Architecture

Consider a logic-only network with no memory involved. The algorithm requies N gate operations per input word and the architecture is measured as G, the number of gates in the network, each of potential speed f_L Hz. So

$$NR \leqslant Gf_L \tag{7.2}$$

and the efficiency of the architecture is measured as

$$E_{arch} = \frac{NR}{Gf_L} \leqslant 1 \tag{7.3}$$

More usefully, the cost of the network in GateHz per unit throughput is

$$C_N = \frac{Gf_L}{R} = \frac{N}{E_{arch}} \tag{7.4}$$

For an implementation combining logic and memory the cost function is modified to

$$C_N = \frac{Gf_L + K_{RAM}B_{RAM} + K_{ROM}B_{ROM}}{R} \tag{7.5}$$

where B_{RAM},B_{ROM} are the number of bits of memory used and K_{RAM},K_{ROM} are the relative costs, for example in Watts, of one bit of memory to one logic gate.

Hence the overall system cost, C_S, for a given throughput is given by

$$C_S = C_N \; R \; P_{tech}. \tag{7.6}$$

where C_S is measured in Watts or Silicon Area if P_{tech} is the technology price in units of Watts per GateHz or Silicon Area per GateHz. In (7.6) C_N is found from (7.5).

Whilst the cost function in (7.5) does not properly take into account some real world costs, such as design, construction, testing time and, as noted above, programmability, it does seem a fair basis for comparing implementations of specified processing functions. It correctly takes into account, for instance, simple versus complex control features, and the advantages of pipelined, parallel and systolic structures designed to keep all logic gates continuously active. C_N is not affected by choices which achieve high throughput rates by the use of more or faster hardware components.

As examples of the usefulness of costing in terms of C_N, we consider the cost of several different networks for computing complex DFTs in the range of 8 to about 1000 points. We then translate C_N in GateHz per unit throughput into power costs per unit throughput for a current (bipolar) technology.

7.3 Assumptions about Arithmetic and Memory Functions

The following simple assumptions were made about arithmetic and memory functions.

(i) An ALU, for an input word length of b-bits, has a complexity of 16b logic gates and an operation time, utilising 'carry look-ahead', equivalent to 5 gate delays (t).

$$G_{ALU} = 16b \qquad T_{ALU} = 5t \tag{7.7}$$

(ii) A typical multiplier architecture (TRW) is assumed with b-bit inputs and a 2b-bit output which has a logic gate complexity of

$$G_{Mult} = 10b^2 + 30b + 80 \tag{7.8}$$

The multiplication time is taken as

$$T_{Mult} = 4bt \tag{7.9}$$

(iii) The random access memory time T_{RAM} is assumed to be adequately approximated by

$$T_{RAM} = [5 + \text{Log}_2(\text{Number of addresses})]t_{RAM} \tag{7.10}$$

where t_{RAM} is the appropriate gate delay and is kept separate from that for logic, t, to allow for case where technology dependence affects the relative speeds of logic and RAM.

(iv) It is assumed the ROM does not affect the throughput rate; hence its access time is not considered in the assessment. It does, however, contribute to chip area, power etc, and is consequently included in the cost function.

(v) Parallel-access latches are assumed to be characterised by

$$G_{Latch} = 10b \qquad\qquad T_{Latch} = 5t \qquad\qquad (7.11)$$

(vi) A 2 to 1 b-bit multiplexer is described by

$$G_{Mux} = 4b + 3 \qquad\qquad T_{Mux} = 3t \qquad\qquad (7.12)$$

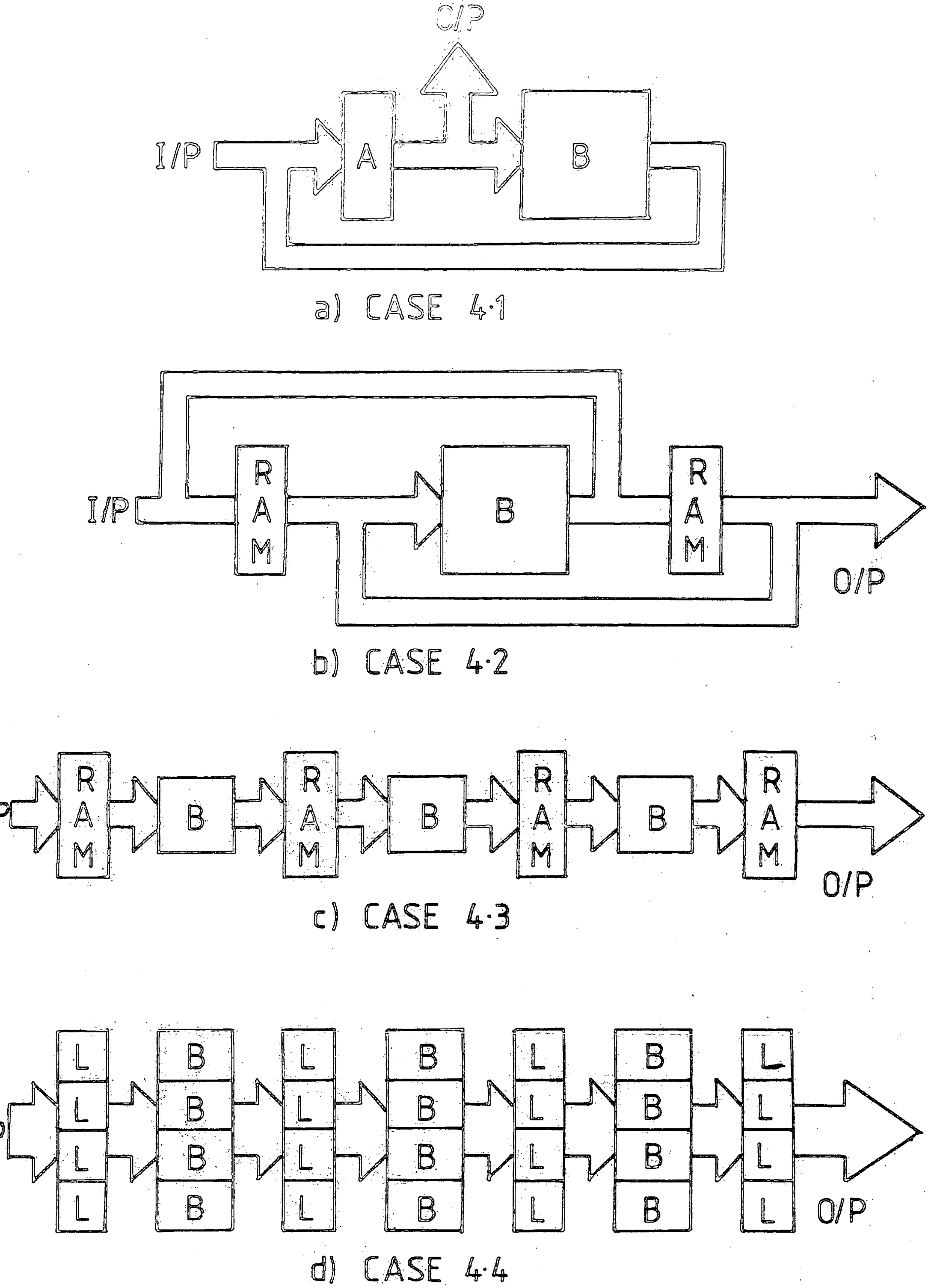
Some futher assumptions are needed about the number of gates in pipelined multipliers and for the one-bit systolic arrays of chapter 6. These assumptions are given in the relevant sections below.

7.4 Implementing the FFT

Five different networks for implementing the Cooley-Tukey Fast Fourier Transform Algorithm [7.5, 7.6] are discussed in this section in terms of radix-2 butterfly elements. The first four of these structures represent increasing parallelism offering trade-offs between processing speed and power dissipation and are sketched in figure 7.4.1. The fifth implementation uses a pipelined butterfly built up from four pipelined multipliers.

7.4.1 A Four-Cycle 'In-Place' Butterfly

This implementation is aimed at low power consumption at the expense of computation speed. Two multipliers and two ALUs are used to perform a butterfly in four clock cycles. Only one scratch pad memory



a) CASE 4·1

b) CASE 4·2

c) CASE 4·3

d) CASE 4·4

FIG. 7.4.1 Different FFT Hardware Configurations

is employed, the output of the multiplier-accumulator combination being returned to the input memory. Since there is only one memory, time must be allowed to set up the data array prior to each transform and an equal amount of time to output data after each transform. For a transform of length N the number of complex multiplications is $(\mathrm{NLog}_2/2)$. These are all done by the same butterfly element using in-place computations. In each FFT implementation the RAM and ROM are accessed whilst the multiplier-accumulator combinations are busy.

The time to complete one transform is

$$T_{4.1} = (2N\log_2 N)(T_{\mathrm{Mult}} + 2T_{\mathrm{ALU}} + T_{\mathrm{RAM}}) + 2NT_{\mathrm{RAM}}$$

$$= 2N(4b + 10)\mathrm{Log}_2 Nt + 2N(1 + \mathrm{Log}_2 N)(5 + \mathrm{Log}_2 N)t_{\mathrm{RAM}} \quad (7.13)$$

The total number of logic gates in the circuit is given by

$$G_{4.1} = 2G_{\mathrm{Mult}} + 2G_{\mathrm{ALU}} + G_{\mathrm{Control}}$$

$$= 20b^2 + 92b + 560 \quad (7.14)$$

A total of 400 logic gates has been assumed for the FFT control. This figure has been gauged from recent hardware design projects.

Finally, considering the storage requirement, if all the 'twiddle factor' values are precalculated, the trignometrical table has to be $(\mathrm{NLog}_2 N)/2$ words deep, with real and imaginary values each of b-bits. The node address table, which selects the words from the scratch pad, has as many words as complex multiplications, $(\mathrm{NLog}_2 N)/2$, and a word length of $2\mathrm{Log}_2 N$ bits to select the desired pair of words from the scratch pad. Thus

$$(B_{\mathrm{RAM}})_{4.1} = 2Nb \text{ bits} \quad (7.15)$$

$$(B_{\mathrm{ROM}})_{4.1} = Nb\mathrm{Log}_2 N + N(\mathrm{Log}_2 N)^2 \text{ bits} \quad (7.16)$$

7.4.2 A Single Cycle 'In Place' Butterfly

Four multipliers and six ALUs are used to form one complex butterfly element, with the same element being used for 'in-place' computations on all passes. This single cycle butterfly element is used N/2 times for each of $\mathrm{Log}_2 N$ passes. Two scratch pad memories are utilised, functioning alternately as input and output memories. The output and input to the external system can take place simultaneously involving only N sequential RAM accesses rather than 2N as above. Then,

$$\begin{aligned} T_{4.2} &= (N\mathrm{Log}_2 N)/2)(T_{\mathrm{Mult}} + 2T_{\mathrm{ALU}} + T_{\mathrm{RAM}}) + NT_{\mathrm{RAM}} \\ &= ((4b+10)((N\mathrm{Log}_2 N)/2))t + (N(2+\mathrm{Log}_2 N)(5+\mathrm{Log}_2 N)/2)t_{\mathrm{RAM}} \end{aligned} \tag{7.17}$$

$$\begin{aligned} G_{4.2} &= 4G_{\mathrm{Mult}} + 6G_{\mathrm{ALU}} + G_{\mathrm{Control}} \\ &= 40b^2 + 216b + 720 \end{aligned} \tag{7.18}$$

$$(B_{\mathrm{RAM}})_{4.2} = 4Nb \tag{7.19}$$

$$(B_{\mathrm{ROM}})_{4.2} = Nb\mathrm{Log}_2 N + N(\mathrm{Log}_2 N)^2 \tag{7.20}$$

7.4.3 $\mathrm{Log}_2 N$ Butterflies

One butterfly element per pass is employed to give an improvement of $\mathrm{Log}_2 N$ times in calculation speed over implementation 4.2. Double buffered memory is used and data is passed from one butterfly to alternate input memories of the next stage. One memory receives data from the preceeding stage whilst butterfly computations are performed on data supplied by the other memory. Then,

$$\begin{aligned} T_{4.3} &= N(T_{\mathrm{Mult}} + 2T_{\mathrm{ALU}} + T_{\mathrm{RAM}})/2 \\ &= N(4b + 10)/2t + N(5 + \mathrm{Log}_2 N)/2t_{\mathrm{RAM}} \end{aligned} \tag{7.21}$$

$$\begin{aligned} G_{4.3} &= (4G_{\mathrm{Mult}} + 6G_{\mathrm{ALU}})\mathrm{Log}_2 N + G_{\mathrm{Control}} \\ &= (40b^2 + 216b + 320)\mathrm{Log}_2 N + 400 \end{aligned} \tag{7.22}$$

$$(B_{\mathrm{RAM}})_{4.3} = 4Nb(1 + \mathrm{Log}_2 N) \tag{7.23}$$

$$(B_{\mathrm{ROM}})_{4.3} = Nb\mathrm{Log}_2 N + N(\mathrm{Log}_2 N)^2 \tag{7.24}$$

7.4.4 A Totally Parallel FFT

This is the ultimate structure for a radix-2 algorithm with one butterfly element per butterfly computation performed. The throughput is fast at the expense of a very high power consumption. As each butterfly has a dedicated function no trigonometrical or node address tables are required.

As in section 7.4.3 a double-buffered memory scheme is needed to make full use of the pipelined structure. Since memory access for all the butterflies takes place simultaneously, individual parallel-access latches have replaced RAM. Similarly pairs of latches are interchanged between stages of the pipeline.

$$T_{4.4} = T_{Mult} + 2T_{ALU} + T_{Latch}$$
$$= (4b + 15)t \qquad (7.25)$$

$$G_{4.4} = (4G_{Mult} + 6G_{ALU})(NLog_2N)/2 + G_{Control} + 4N(1 + Log_2N)G_{Latch}$$
$$= NLog_2N(20b^2 + 148b + 160) + 40Nb + 400 \qquad (7.26)$$

7.4.5 A Pipelined Butterfly

This section proposes using one pipelined butterfly built up from four pipelined multipliers and six pipelined adders. This single pipelined butterfly is used in a similar manner to the single conventional butterfly of section 7.4.2. The multiplier architecture is that proposed by Myers, using b(b+1) adder cells. Suppose these pipelined multipliers were arranged so as to give their output skewed least significant bit first. Then it would be possible to add the products from two of these multipliers together using b clocked adder cells. On the first cycle the least significant bits could be added together, on the second the two least significant bits together with the carry from the previous summation would be summed and so on for sucessive cycles. This pipelined adder would the consist of b full adder cells and give an output skewed least significant bit first.

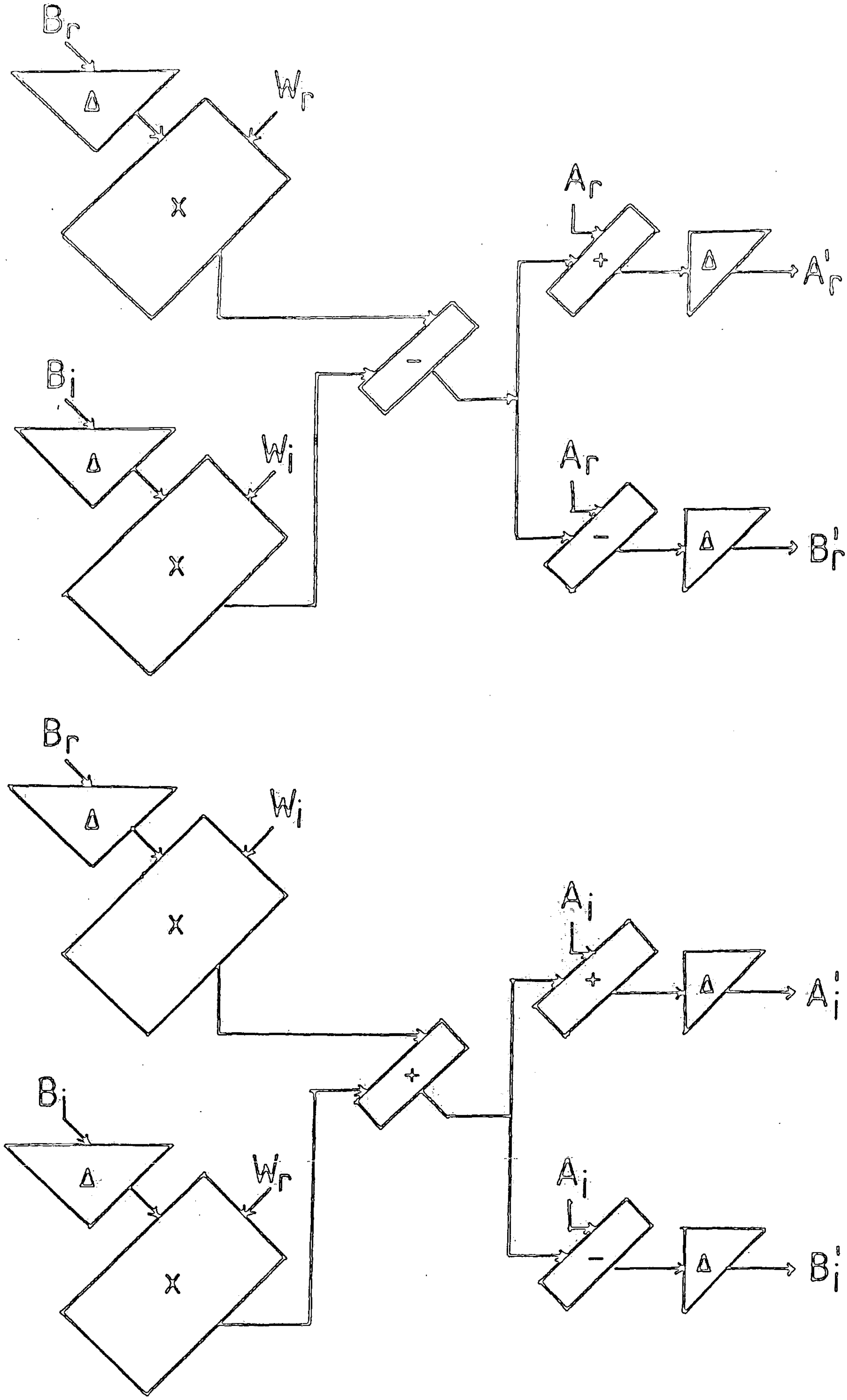


Fig. 7.4.5. A Pipelined FFT Butterfly

Further pipelined adders could be added immediately after the first with no intervening delays. This feature is utilised in the pipelined butterfly illustrated in figure 7.4.5.

The total number of gates used in this butterfly is assessed using the following assumptions,

(i) Myers' multiplier architecture for a bxb bit multiplication with least significant bit first staggered output is characterised by

$$G_{pipe} = 18b^2 + 14b \qquad T_{pipe} = 5t \qquad (7.27)$$

This assumes that a clocked carry save adder cell requires 9 gates and a delay of 2 gate delays.

(ii) Each of the pipelined adders uses b cells each cell containing 9 gates

(iii) The final 4 deskewing arrays are assumed to have no delay on the most significant bit and use $\frac{1}{2}b(b-1)$ delays cells, each delay cell containing 2 gates.

(iv) The latency of the pipelined multiplier is (3b-1) cycles, giving an overall latency to the butterfly of (3b+1) cycles. This latency means that the butterfly must be run for (3b+1) empty cycles between each of the Log_2N passes through the algorithm.

(v) Again 400 gates are allowed for control.

These assumptions give the total gate count for this pipelined butterfly implementation as

$$G_{4.5} = 76b^2 + 106b + 400 \qquad (7.28)$$

There are 4 sets of complex RAM and allowance is made to the transform time for transfering the results to the external system, as in implementation 4.2 above. Since each butterfly cycle requires two input words to be read from RAM this limits the maximum throughput rate. As mentioned above the butterfly pipeline must be emptied at the end of each pass through the transform. This gives the total time taken for the transform as

$$T_{4.5} = 2T_{RAM}\ (N/2 + \text{latency})\ \text{Log}_2N + NT_{RAM}$$

$$= [2(5 + \text{Log}_2N)((N/2 + 3b + 1)\ \text{Log}_2N + (N/2))]t_{RAM} \qquad (7.29)$$

$$(B_{RAM})_{4.5} = 4Nb \qquad (7.30)$$

The pipelined butterfly could be clocked faster than this but a more elaborate memory organisation would be needed to keep up the throughput rate.

The number of bits of ROM required is similar to implementation 4.3 but two sets of addressing ROMs will be needed because of the latency of the butterfly. These ROMs will also have to cope with the additional number of cycles due to the butterfly latency.

$$(B_{ROM}) = 2\text{Log}_2N(N/2 + 3b + 1)(2\text{Log}_2N + b)\text{ bits} \qquad (7.31)$$

7.5 Implementing the WFTA

This section considers 4 implementations of the WFTA which have widely different hardware complexity and throughput rates. They range from a simple scheme which uses two ALUs and a multiplier to the one-bit systolic arrays of chapter 6.

7.5.1 A Low Power WFTA

Two ALUs and one multiplier are used with a scratchpad memory in an implementation aimed at low power consumption. The two ALUs are arranged in parallel to perfom the complex additions and subtractions. All the arithmetic components have latched inputs and outputs giving 9 b-bit latches in total. Again, allowing 400 gates for control, the logic gate complexity is

$$G_{5.1} = G_{Mult} + 2G_{ALU} + 9G_{Latch} + G_{Control}$$

$$= 10b^2 + 152b + 480 \qquad (7.32)$$

The scratchpad memory must accomodate all intermediate values in the calculation. The intermediate values arise because M>N and the algorithms to evalate A and C matrices require some intermediate additions, the greatest number being 17. The the RAM size is at most

$$(B_{RAM})_{5.1} = 2(M + 17)b \text{ bits} \tag{7.33}$$

Each operation involves two operands read from RAM, latched, the result formed, latched and subsequently read back into RAM. Again time is allowed to set up the input data array and to output the transformed data. With 2 ALUs working in parallel the transform time is

$$\begin{aligned} T_{5.1} &= A(3T_{RAM} + 3T_{Latch} + T_{ALU}) + 2M(3T_{RAM} + 3T_{Latch} + T_{Mult}) + 2NT_{RAM} \\ &= (20A + 30M + 32Mb)t + (5 + Log_2(M+17))(3A + 6M + 2N)t_{RAM} \end{aligned} \tag{7.34}$$

ROM is assumed to hold all addressing and control functions. Each arithmetic operation requires three addresses, two for the operands and one for the result. The number of ROM words is 3(A+2M). As each word of ROM should address all the RAM, both real and imaginary parts, as well as providing 4 control bits, its width is $Log_2(M+17)+5$ bits. The four control bits are to select READ/WRITE, 1st/2nd operand, Add/Subtract and ALU/Multiplier.

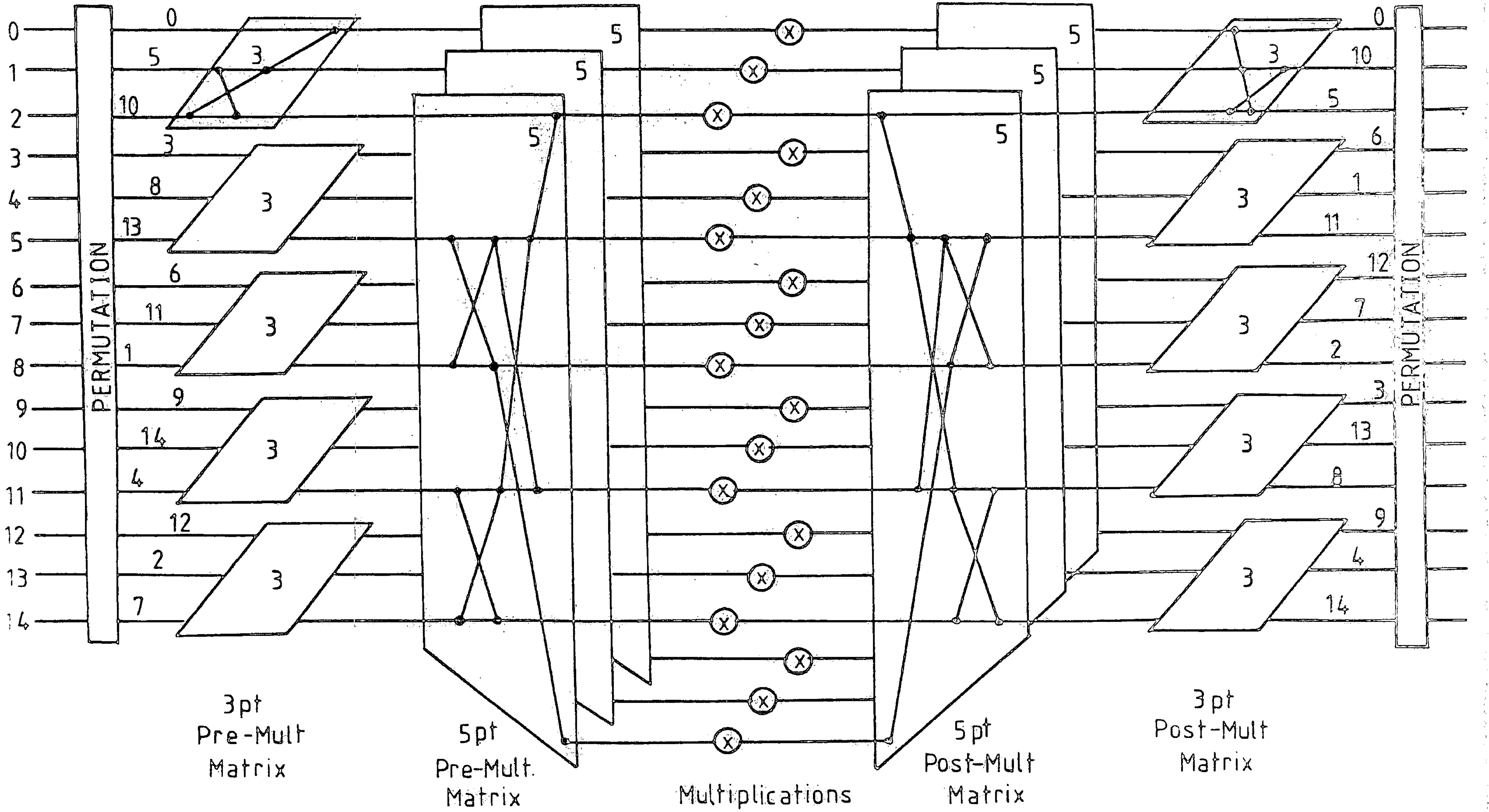
Finally the transform coefficients need Mb bits of storage. It is possible to arrange the coefficients and data so that the same address can be used for both, thus avoiding a separate coefficient addressing procedure.

$$(B_{ROM})_{5.1} = 3(A+2M)(Log_2(M+17)+5)+Mb \text{ bits} \tag{7.35}$$

7.5.2 WFTA: 'Groups of ALUs'

Here groups of ALUs are wired together to perform the additions for the 'small-N' A and C matrices. Figure 4.2, reproduced from chapter 4, illustrates a 15-point WFTA and gives one possible indexing scheme. In the 15-point tranform, which is a two factor example, 4 separate groups of ALUs are needed to handle complex data. A five stage pipeline is used, the stages being input, pre-

Fig. 4 .2 A 15-point WFTA



multiplication additions, the multiplication stage, post-multiplication additions and output. Figure 7.5.2 gives a schematic layout for this implementation.

The pipeline has 8 sets of RAM each of which has to contain M complex values. The total RAM is

$$(B_{RAM})_{5.2} = 16MB \text{ bits} \tag{7.36}$$

As each RAM reads and writes onto separate buses in two stages of the pipeline 4 tri-state latches per RAM are needed. These latches are controlled by the ROMs containing the RAM addressing.

The 2M multiplies are performed by 2 multipliers, each calculating M products. The real and imaginary parts of the product are interchanged by a multiplexer if the coefficient is imaginary. The coefficients are each stored with a 1-bit real/imaginary flag and are addressed in a similar manner to 5.1. The delay due the the multiplication stage of the pipeline would be

$$M(2T_{RAM} + 2T_{Latch} + T_{Mult} + T_{Mux}) \text{ Gate delays.} \tag{7.37}$$

The pre- and post-multiplication stages can each be broken down into k substages, each substage corresponding to a factor. Within a substage groups of n_i numbers (m_i for the post-multiply stage) are read from RAM and are latched at the inputs to an ALU group. When all n_i (m_i) values have been latched the results ripple through the ALUs and the results are latched. These latches are enabled sequentially and the sums are read back into the other RAM of that stage. By using a two bus structure input and output to the ALU group can take place simultaneously. With an even number of factors a further set of data transfers between RAMs will be needed a the end of each substage.

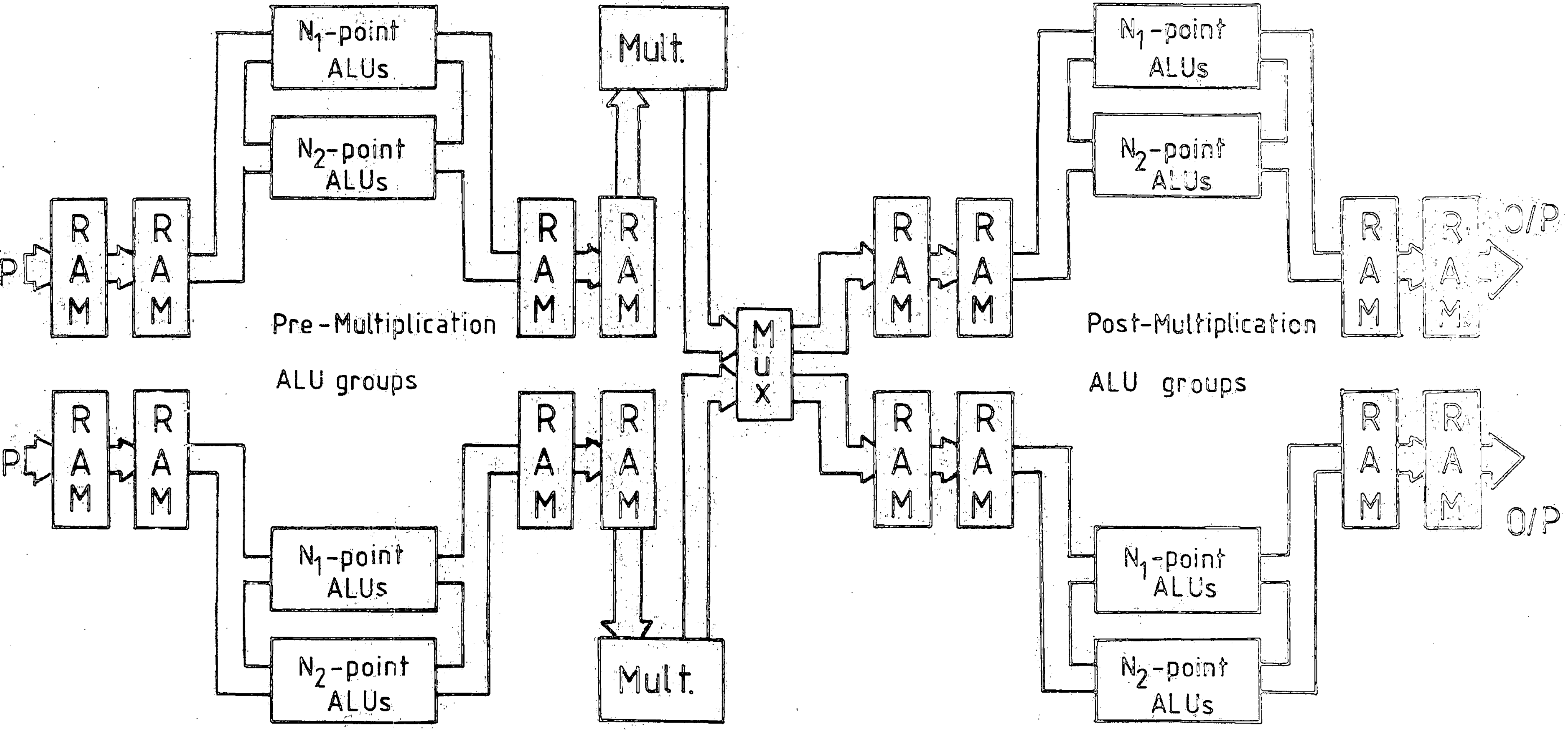


Fig. 7.5.2 Groups of ALUs WFTA Implementation

All Winograd's 'small-N' algorithm networks have a ripple through time of at most three addition times. Thus if one 'cycle' is taken as

$$T_{RAM} + 2T_{Latch} = (5 + \mathrm{Log}_2 M)t_{RAM} + 10t \tag{7.38}$$

and $t_{RAM} \simeq t$, the ripple through delay of any ALU block is less than one cycle. Consequently the time taken for this stage of the pipeline is the number of RAM access cycles with an additional cycle for each ALU block delay. The total number of cycles is

$$n_1 + \frac{N}{n_1}(m_1+1) + n_2 + \frac{N\ m_1}{n_2 n_1}(m_2+1) + n_3 + \frac{N\ m_1 m_2}{n_3 n_1 n_2}(m_3+1) + \ldots \tag{7.39}$$

A further M cycles are needed if the number of factors is even.

The throughput rate for this type of implementation is limited by the multiplication stage for transforms up to lengths of around 300 points. Larger transforms are limited by the ALU stage.

The 4 ROMs addressing the 4 RAMs in the Pre- and Post-multiply stages need as many words as the number of cycles. The 2 ROMs addressing RAM within the multiplication stage each contain M words, whereas the ROMs addressing the input and output buffers each contain N words. Each word is $\mathrm{Log}_2 M+2$ bits in width. The 2 extra bits are to indicate READ/WRITE and the pipeline stage. The total ROM is

$$(B_{ROM})_{5.2} = (4\mathrm{x}(\text{No. of cycles})+2M+2N)(\mathrm{Log}_2 M+2)+M(b+1) \text{ bits} \tag{7.40}$$

The logic gate complexity is found from

2 b bit multipliers	$20(b^2+3b+8)$
64 latches for RAM I/O	640b
No. of ALUs = $2\sum_{i=1}^{k} a_i$	$32b\sum_{i=1}^{k} a_i$
No. of latches for ALU blocks = $4\sum_{i=1}^{k}(n_i+m_i)$	$40b\sum_{i=1}^{k}(n_i+m_i)$
2 b-bit 2:1 multiplexers	$2(4b+3)$

Estimate of control 1000

Total

$$G_{5.2} = 20b^2 + \sum_{i=1}^{k}(32a_i + 40(n_i+m_i)) + 708b + 1166 \qquad (7.41)$$

7.5.3 Completely Parallel WFTA

This implementation is similar in style to the final unpipelined FFT implementation, separate multipliers and ALUs being used for each operation, giving a total of 2A ALUs and 2M multipliers. A three stage pipeline is used. No RAM is needed and completely parallel inputs and outputs are assumed. Two sets of latches are used between stages of the pipeline with latches acting as input and output buffers. As all multipliers and ALUs are dedicated to a single function no ROM is needed. The logic gate complexity is given by

$$\begin{aligned} G_{5.3} &= 2AG_{ALU} + 2MG_{Mult} + 12(N+M)G_{Latch} \\ &= 20Mb^2 + (32A + 180M + 120N)b + 160M \qquad (7.42) \end{aligned}$$

The throughput is limited either by the multiplication time or the ripple through delay of the ALU network. The ripple through delay of a composite transform is the sum of the ripple through delays of each of the smaller factors. The ripple through delay of a 'small-N' algorithm is, at most, 3 addition times. Hence throughput is limited by the multiplication delay if the number of factors, k, obeys

$$k < \frac{4b}{15} \qquad (7.43)$$

For 16-bit data this is 4 or less factors. As the transform lengths considered here have 4 or less factors, each of their throughputs is governed by the multiplication time rather than the ALU network delay. Hence

$$\begin{aligned} T_{5.3} &= 2T_{Latch} + T_{Mult} \\ &= (4b+10)t \qquad (7.44) \end{aligned}$$

7.5.4 A WFTA Implementation using One-bit Systolic Arrays

This uses the ideas presented in chapter 6 for a WFTA implementation based upon one-bit systolic arrays with complex interleaved data and Myers' multiplier architecture. The following assumptions were made

(i) Myers' pipelined multiplier architecture is assumed using b(b+1) full adder cells. With input words that are already skewed least significant bit first and output words similarly skewed the gate count for this pipelined multiplier is

$$17b^2 + 13b \qquad (7.45)$$

This assumes, as before, 9 gates for the CSA adder cell and 2 gates per delay element.

(ii) The total number of cells in the A and C arrays is

$$(N + M - 1)(\mathrm{Log}_2N + \mathrm{Log}_2M + 2b + 2) \qquad (8.46)$$

The number of gates in the basic cell is assessed by assuming 9 gates for the adder, 6 gates for the delays associated with the two control bits and input data bit with 4 gates for 'extra logic' which performs the one-bit product. Thus the basic cell contains a total of 19 gates. In addition to the adder delay the extra logic is assumed to require a further 3 gate delays giving a cell delay of 8 gate delays.

(iii) The initial data skew and final result skew will each need a triangular array of delays. The number of delay cells is $\frac{1}{2}b(b+1)$. Each delay cell needing 2 gates.

(iv) Both the A and C arrays need to have their coefficients stored. Each element of the matrices requires 2 bits. Each column of the A and C arrays is assumed to have the full (N+M-1) possible values stored for it. The coefficients for the multiplier will need a one bit flag to indicate real/imaginary. The total ROM storage is

$$(B_{ROM})_{5.4} = 4(N+M-1)^2 + M(b+1) \text{ bits} \qquad (7.47)$$

(v) RAM storage is assumed to be nil as the data enters the system in natural order and is clocked out in natural order.

(vi) The gap between the starts of successive transforms is 2M cycles. Thus with a cell delay of 8 gate delays the throughput rate of the system is

$$N/16M \text{ words per gate delay} \qquad (7.48)$$

7.6 Comparison of Algorithms

This section seeks not only to test the validity of the use of the 'cost function' in comparing implementations but also to compare different implementations and algorithms. Only the broadest of conclusions are drawn as changes to the assumptions made in the previous sections can alter the relative merits of some implementations which have very similar values of the cost function.

7.6.1 Comparison of FFT Algorithms

As a way of assessing the value of the cost function the four 'conventional' ways of the implementing the FFT are compared. Figure 7.6.1 illustrates a power cost function for these four FFT implementations. This power cost function, defined by (7.5), is plotted for 16-bit data. These cost functions assume that $t_{RAM} \simeq t$, power consumption per bit of RAM = 0.3 x power consumption of one logic gate and the power consumption of one bit of ROM as 0.03 x that of one logic gate.

In general the cost function of these four FFT implementations gradually increases with transform length. For a given transform length the cost function decreases as the hardware parallelism and throughput increase. The most 'efficient' conventional FFT implementation is that given in section 7.4.4. This implementation has

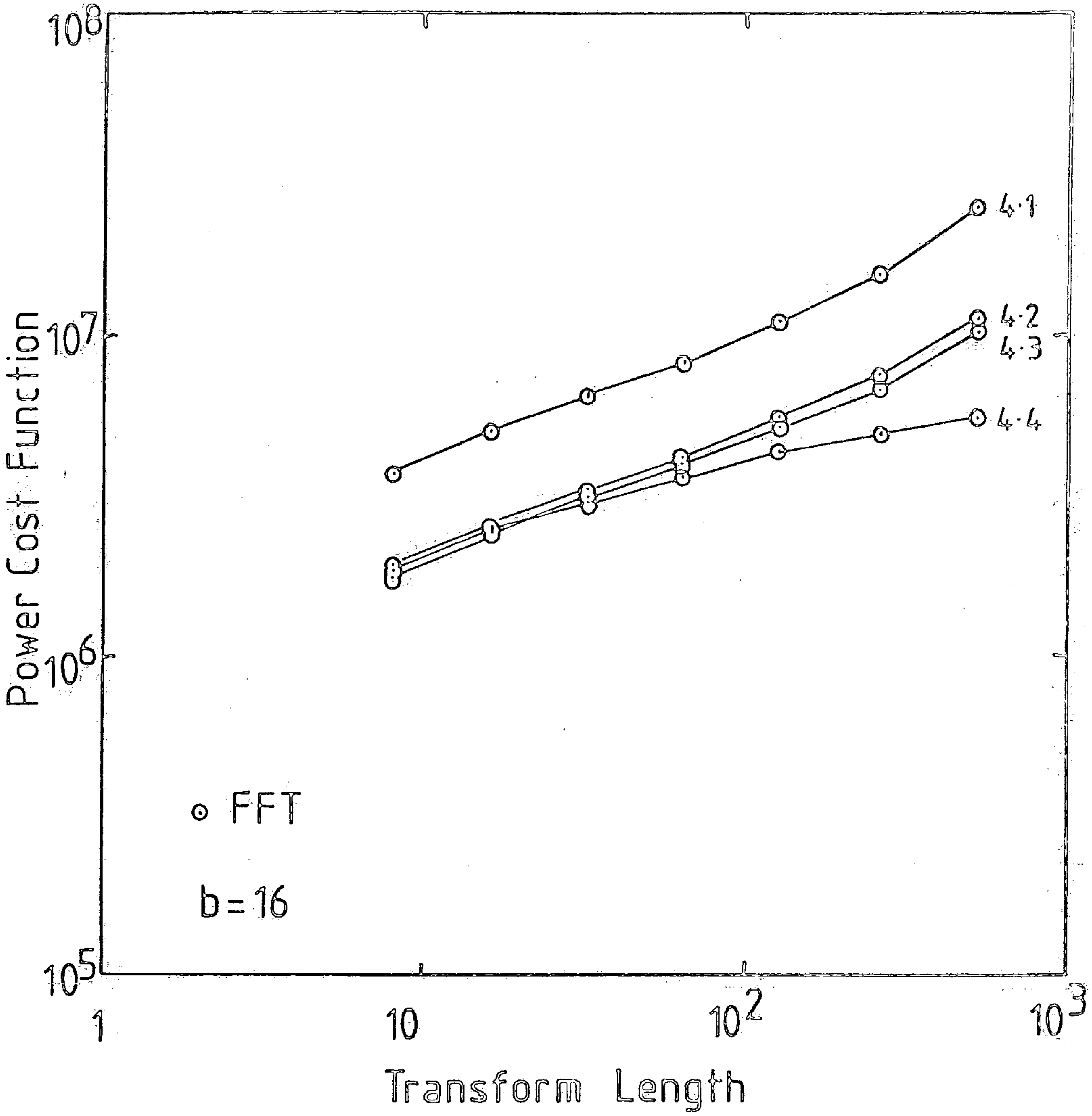


Fig. 7.6.1 4 FFT Cost Functions

the least overheads to support requiring no RAM or ROM. These seem reasonable conclusions to draw from the cost function.

Figure 7.6.2 gives the cost functions for the most parallel FFT and the pipelined butterfly implementation. It can be seen that on these assumptions the pipelined butterfly produces a similar cost function to the most parallel FFT for transform lengths above 100 points or so. For much longer transform lengths the pipelined butterfly cost function becomes larger than the fully parallel FFT cost function. Note that the pipelined butterflies cost function has a minimum, this is because of the effect of the latency of the butterfly. Improvements to the way of utilising the pipelined butterfly are almost certainly possible. The scheme given above does not use the butterfly at it's maximum throughput rate and an implementation with a more elaborate memory organisation may well have a lower cost function.

7.6.2 Comparison of WFTA Implementations

Figure 7.6.3 gives the same cost function plotted for the three conventional WFTA implementations. The cost function for the most parallel FFT is also included. It is immediately apparent that first two ways of implementing the WFTA offer few architectural advantages - they have very similar cost functions to the low power FFT implementation. The fully parallel WFTA implementation offers the lowest cost function of the architectures considered so far. Comparison with the fully parallel FFT shows the fully parallel WFTA to require less gates, because of the less operations performed, and to have a higher throughput.

Figure 7.6.4 shows the cost functions plotted for the most parallel FFT and WFTA implementations together with the cost functioon for the systolic WFTA implementation. The systolic WFTA has the lowest cost function for short transform lengths. The gradient of the

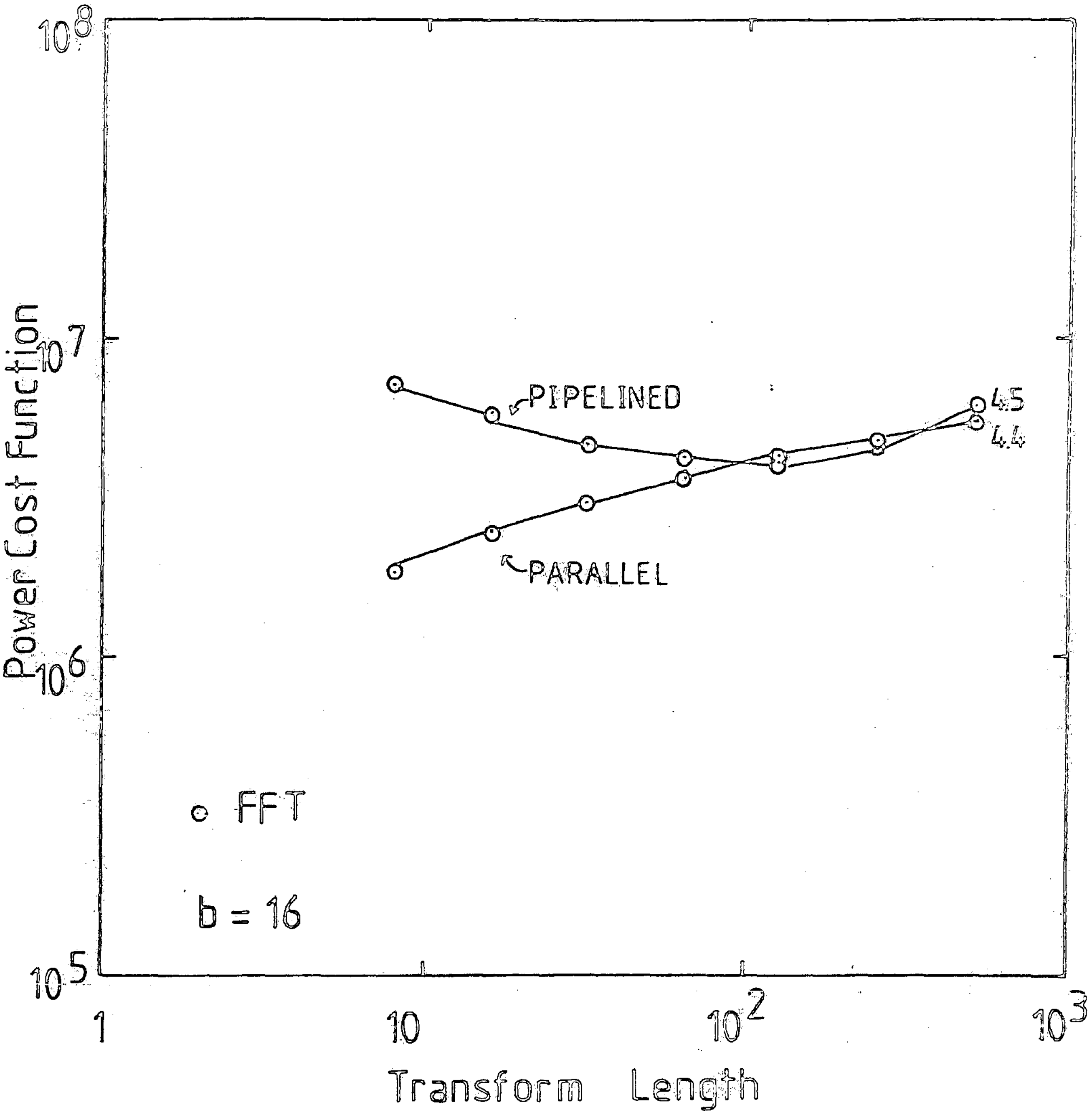


Fig. 7.6.2 Pipelined and Parallel FFT Cost Functions

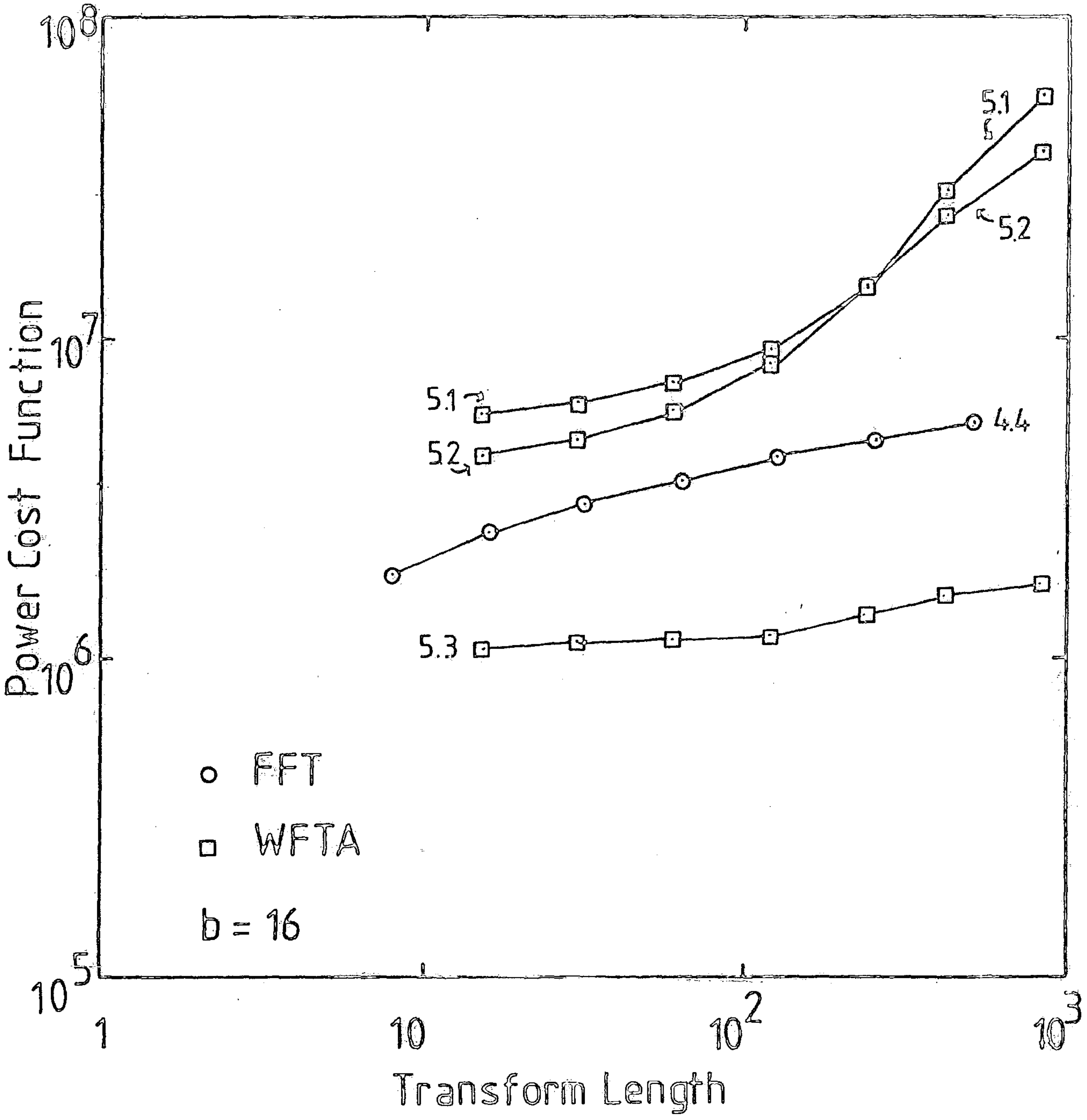


Fig. 7.6.3 WFTA and Parallel FFT Cost Functions

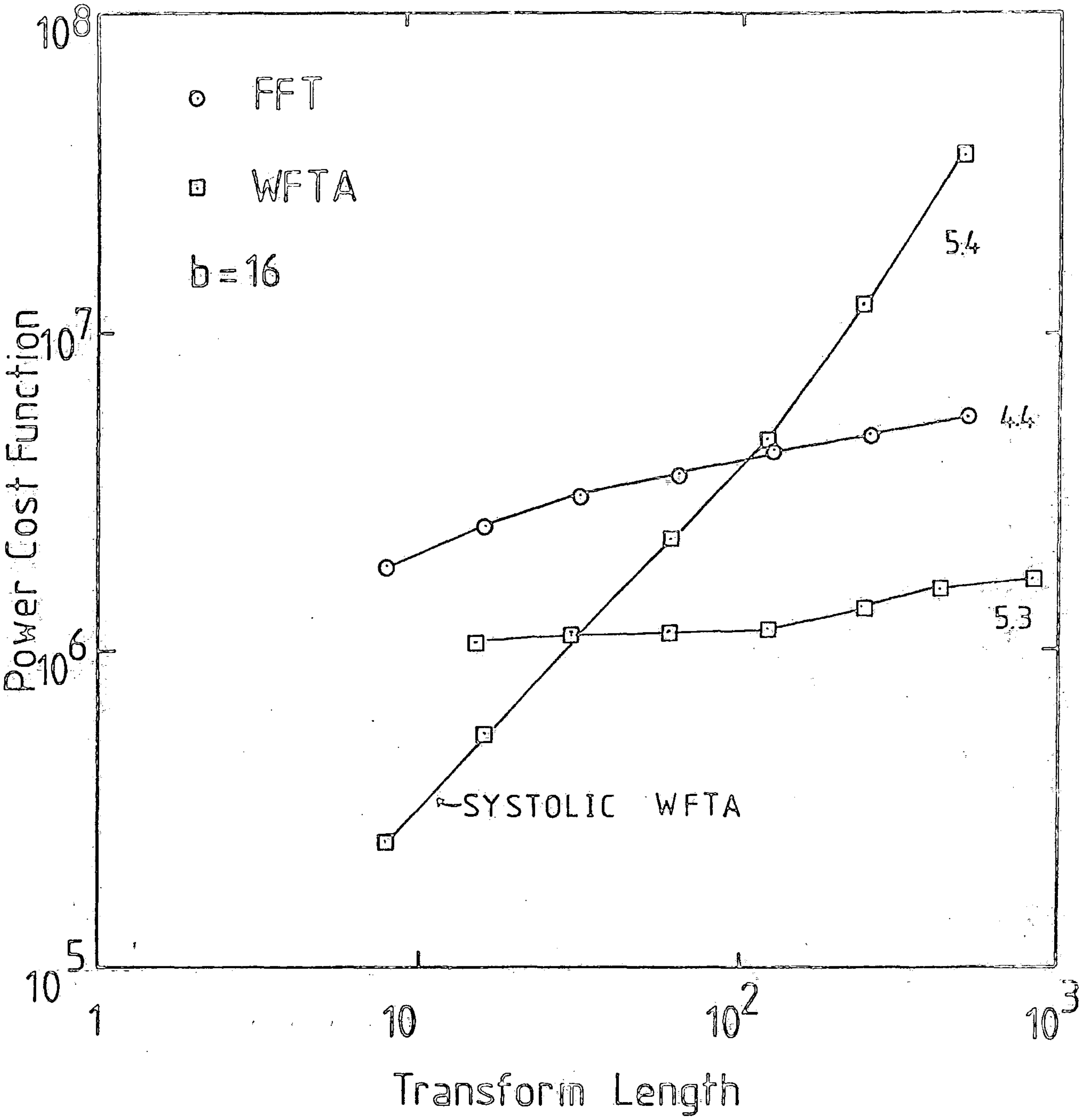


Fig. 7.6.4 Parallel FFT, WFTA and Systolic WFTA Cost Functions

systolic WFTA cost function is very steep in comparison to the other architectures. This is because of the huge numbers of gates in the A and C arrays.

7.6.3 Conclusion to Algorithm Comparison

The cost function allows both individual implementations of an algorithm to be compared as well as dissimilar algorithms for the same function. The results derived indicate that a reliable judgement on algorithm suitability demands a detailed analysis of specific designs. The results presented above indicate that the less overheads required the lower the potential cost function. Thus applications requiring relatively slow transform times should be implemented by the most efficient architecture but should use a slower technology.

As an example of the application of the cost function, the power consumption for a 120-point complex DFT with a 10MHz data rate using implementation 7.5.3, the most fully parallel WFTA is calculated. A current bipolar technology is assumed which has a power dissipation of 1.5mW/Gate and t=2.5ns, giving $P_{Tech}=3.8x10^{-12}$Watts/GateHz.

$$C_S = C_N R P_{Tech} \qquad (7.6)$$

From figure 7.6.3 $C_N=1.2x10^6$ Gate operations. This gives the power consumption for this implementation as 46W. Notice that the time available for the multiplication is 12 µsec and that the implementation uses 244 multipliers! It is as well to remember that the cost function is a measure of architectural efficiency.

In terms of the cost function the systolic WFTA offers the 'best' way of implementing short transform lengths and the fully parallel WFTA for longer transforms. This suggests that an implementation of the DFT based upon the Prime Factor Algorithm which uses the systolic WFTA to implement the factors may have some advantages.

Chapter 8

A Discrete Cosine Transform Algorithm

So far only convolutions and DFTs have been considered in this thesis. This chapter shows that another function, the Discrete Cosine Transform, can be evaluated using an algorithm of the form of (1.20). The algorithm is derived from the Winograd Fourier Transform Algorithm and may be implemented using the one-bit systolic arrays proposed in chapter 6. However as the DCT only involves real arithmetic it is shown that these arrays of cells may be halved in area.

8.1 The Discrete Cosine Transform

Several authors [8.1,8.2] have shown that the Discrete Cosine Transform is a good technique to adopt for the data reduction of video signals. Earlier methods of realising the DCT have, for the most part, been based upon the Fast Fourier Transform (FFT) [8.3-8.5]. An alternative algorithm, using Hadamard sparse matrices, has been proposed by Hein and Ahmed [8.6]. This has been implemented by Ghanbari and Pearson [8.7].

The Discrete Cosine Transform may be defined as

$$C(0) = \sqrt{\frac{1}{N}} \sum_{n=0}^{N-1} x(n) \qquad (8.1)$$

$$C(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x(n) \cos \frac{k\pi(2n+1)}{2N} \qquad k=1,\ldots,N-1 \qquad (8.2)$$

with inverse

$$x(n) = \sqrt{\frac{1}{N}} C(0) + \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} C(k) \cos \frac{n\pi(2k+1)}{2N} \qquad n=1,\ldots,N-1 \qquad (8.3)$$

Notice that unlike the DFT the DCT cannot be used as it's own inverse.

The DFT of R points is defined as,

$$F(q) = \sqrt{\frac{1}{R}} \sum_{p=0}^{R-1} x'(p) \left[\cos\left(\frac{2\pi pq}{R}\right) + j\sin\left(\frac{2\pi pq}{R}\right)\right] \quad q=0,1,\ldots,R-1 \quad (8.4)$$

with inverse

$$x'(q) = \sqrt{\frac{1}{R}} \sum_{q=0}^{R-1} F(q) \left[\cos\left(\frac{2\pi pq}{R}\right) - j\sin\left(\frac{2\pi pq}{R}\right)\right] \quad p=0,1,\ldots,R-1 \quad (8.5)$$

Then a DCT may be calculated by a DFT by noticing that

$$\cos\frac{k\pi(2n+1)}{2N} = \text{Real}\left[\cos\left(\frac{2\pi pq}{R}\right) + j\sin\left(\frac{2\pi pq}{R}\right)\right]$$

Provided that $R = 4N$

and $x'(2n+1) = x(n) \qquad n=0,1,\ldots,N-1$

$$x'(l) = 0 \qquad l\neq 2n+1 \qquad (8.6)$$

Then the calculation of the DCT may be described as placing the N terms of the DCT input sequence in the first N odd points (1,3,5,...,2N-1) of a 4N-point sequence. All other terms are zero. Then a Fourier like transform is performed by multiplying by the real part of exp(-2πpq/R). The DCT, except C(0), is found as the first N-1 terms of F(q). Some normalisation coefficients are needed.

8.2 Application of the WFTA

A WFTA algorithm for 4N-points is easily modified to calculate a N-point DCT. The procedure is described below.

1. Only the columns corresponding to the first N odd samples of the A matrix are retained. All other columns are discarded as they operate on zero inputs.

2. Since the DCT involves no complex arithmetic - it has no imaginary or complex coefficients - all the imaginary WFTA coefficients may be removed. The corresponding rows of the A matrix and corresponding columns of the C matrix are removed.

3. Some rows of the A matrix may now be all zero. The coefficients for each of these rows may be removed together with the appropriate columns from the C array.

4. The second to Nth rows of the C are retained. These represent the DCT, all other rows should be discarded.

5. By inspection it may be possible to simplify the C matrix. For example a column of zeroes represents an unused coefficient. Such coefficients and the equivalent rows of the A matrix should be removed from the algorithm.

6. Finally the C(0) term is added to the algorithm. The exact arrangement of the normalisation coefficients in (8.1) and (8.2) determines the number of additional multiplications to be added to the algorithm. With the normalisation coefficients as given above one extra multiplication is needed.

The procedure results in an efficient algorithm for the calculation of the DCT using a small number of multiplications.

The same principles can be used to derive an inverse DCT from a foward WFTA. In this case the first N-1 columns of the WFTA A matrix and the first N odd columns of the C matrix are used, i.e. the oposite way round to the foward DCT. Equally well the above procedure could have been used to calculate the Discrete Sine Transform. The next section illustrates the derivation of a 4-point DCT algorithm.

8.3 Derivation of a 4-point DCT

As an example of the procedure suggested above a 4-point DCT is derived from the 16-point WFTA given by Winograd [8.8].

The A matrix for the 16-point WFTA is

$$\begin{array}{cccccccccccccccc|c}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & \\
1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & \\
0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & i \\
1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & i \\
0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & i \\
0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & i \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & i \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & -1 & 0 & -1 & i \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & i \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & i \\
0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & \\
0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 &
\end{array}$$

The vertical columns removed (0,2,4,...) correspond to zero data. The rows marked with an i correspond to imaginary coefficients in the 16-point WFTA. These rows can also be removed.

Of the remaining rows 3, 5, 9 and 12 are all zeroes. These coefficients may be removed. Row 2 is minus row 1, this allows one of them to be removed. This means a slight change to the C matrix.

Thus coefficients 1, 8, 16, 17 and 18 remain. This implies that only columns 1, 8, 16, 17 and 18 are of interest in the 16-point WFTA C matrix. Examination of rows 2, 3, and 4 of the C matrix, i.e. those rows corresponding to the DCT output, shows that coefficient 1 is not used, it may be deleted. These last operations are illustrated on the 16-point WFTA C matrix given below.

The 16-point WFTA C matrix is

$$\begin{array}{cccccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & 0 & -1 & 1 & 0 & -1 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & -1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & 1 & 0 & 1 & -1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 0 & 1 & -1 & 0 & 1 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & -1 & 0 & 1 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 & 0 & 1 & 1 & 0
\end{array}$$

Hence a possible A matrix for the 4-point DCT algorithm is

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ 1 & 0 & 0 & -1 \\ 0 & -1 & 1 & 0 \end{pmatrix}$$

The C matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{pmatrix}$$

The coefficients are

$1/2\sqrt{2}$
$\frac{1}{2}\cos\pi/4$
$\frac{1}{2}\cos 3\pi/4$
$\frac{1}{2}(\cos 3\pi/8+\cos\pi/8)$
$\frac{1}{2}(\cos 3\pi/8-\cos\pi/8)$.

8.4 Implementing the DCT

As the algorithms derived by the above method have the same form as the WFTA algorithms the methods proposed in chapter 6 can be used to implement these algorithms. Since the DCT involves no complex arithmetic and the data is real there is no need for the data interleaving techniques used in the previous chapter to handle complex data. This allows the arrays to be simplified.

8.4.1 A andC Matrix Reduced Arrays

Figure 6.1.2 illustrated the first four cycles of interaction between the data, coefficients and summation terms in the A and C matrix arrays. For clarity the layout of cells and latches was omitted. This diagram made it clear that any one processor is empty on alternate cycles. This suggests that one processor could be made to perform the work of two.

Consider the contents of the two processors outlined in each of the four cycles of figure 6.1.2. On the even cyles (0,2,4,...) the lefthand processor is busy and the righthand one on odd cycles. These two processor may be implemented in one cell with some additional feedback paths. The data X_k^b enters the cell on even cycles and is reused on the next odd cycle. The summation terms Y_k^b enter on odd cycles and are reused on subsequent even cycles. New coefficients are required on each cycle. Inspection of figure 6.1.2 shows that rows of cells must be out of phase, i.e. if row one is 'even' row two is 'odd'. With this arrangement the coefficients and carries propagate down through the array with no delay.

The compacted array and new coefficient layout are shown in figure 8.1. The feedback paths which enable the data to recirculate are shown as being clocked on either 'odd' or 'even' cycles.

8.4.2 A Reduced Pipeline Multiplier

The two multiplier architectures considered in the last chapter are both suitable for the DCT. As noted in the previous chapter a slight simplification is possible to these multipliers by forcing all the coefficients to be positive so that a general two's complement multiplier is not needed.

For this particular application a second simplification comes

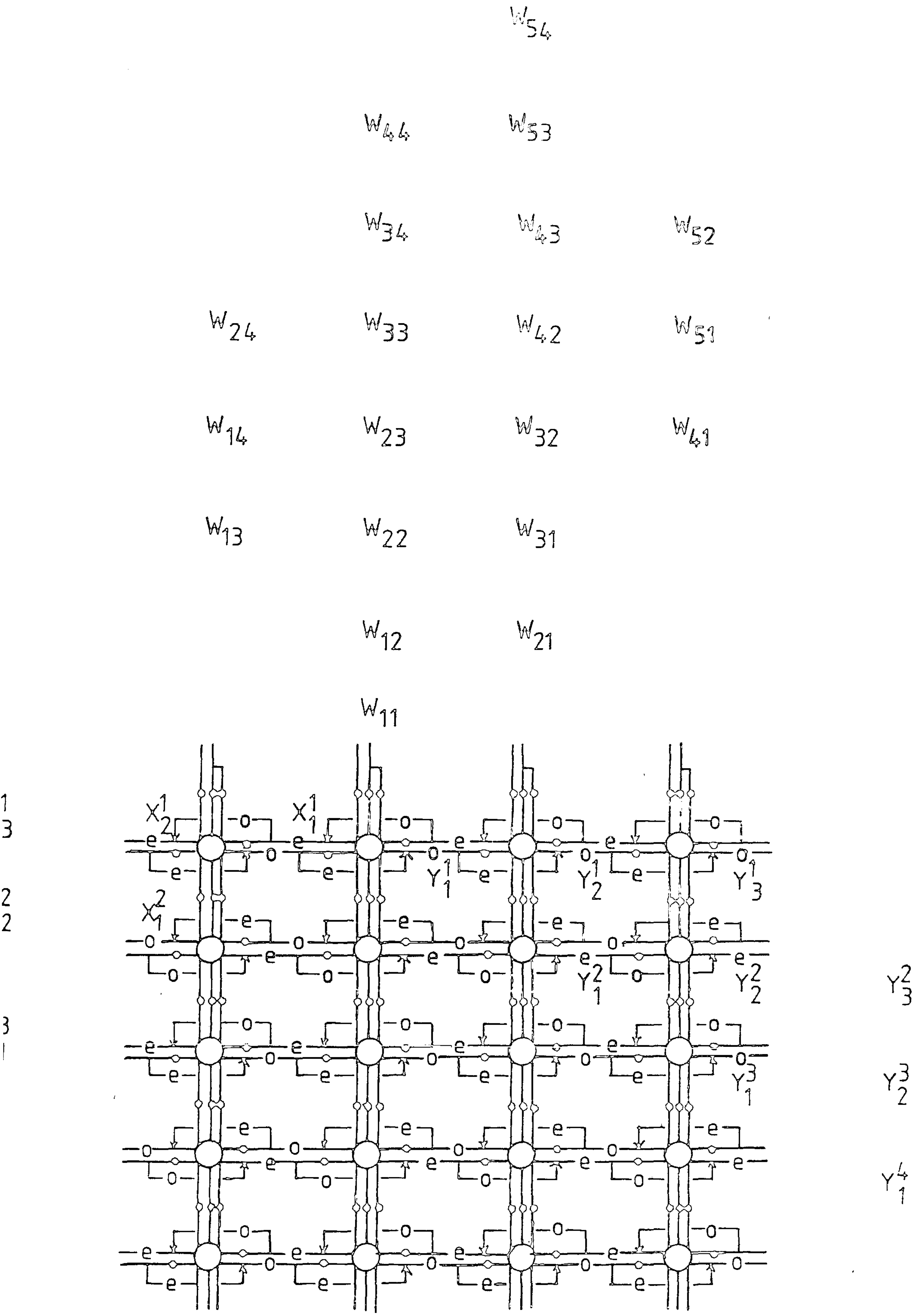


Fig 8.1 Reduced A and C Matrix Arrays

from noting that new words would be presented to the pipelined multiplier on alternate cycles from the A matrix tranform array. As in the case of the transform arrays alternate cells in the multiplier are unused. This reduced interaction is illustrated in figure 8.2 for McCanny and McWhirter's multiplier architecture [8.9]. The multiplier exhibits a similar pattern of 'odd' and 'even' cells. With one important difference a similar halving of the number of cells is possible. This multiplier architecture requires separate paths for the summation terms on 'odd' and 'even' cycles. This is because the summation terms propagate diagonally through the array rather than parallel to the sides.

This simplification yields a mxm cell array for a m-bit x m-bit multiplication from McCanny and McWhirter's multiplier architecture. The reduced array is illustrated in figure 8.3. The diagram also reveals that by multiplexing 2M pins could be used for this multiplier rather than the more normal 4M.

Similar simplifications are possible to the architecture proposed by Myers [8.10] yielding a compacted array of [(M+1)/2]M cells. The [.] denote rounding up to the nearest integer. Which architecture is 'better' will depend upon wordlength and the relative areas of the extra cells needed by McCanny and McWhirter's approach compared with the extra latches needed in Myers' architecture.

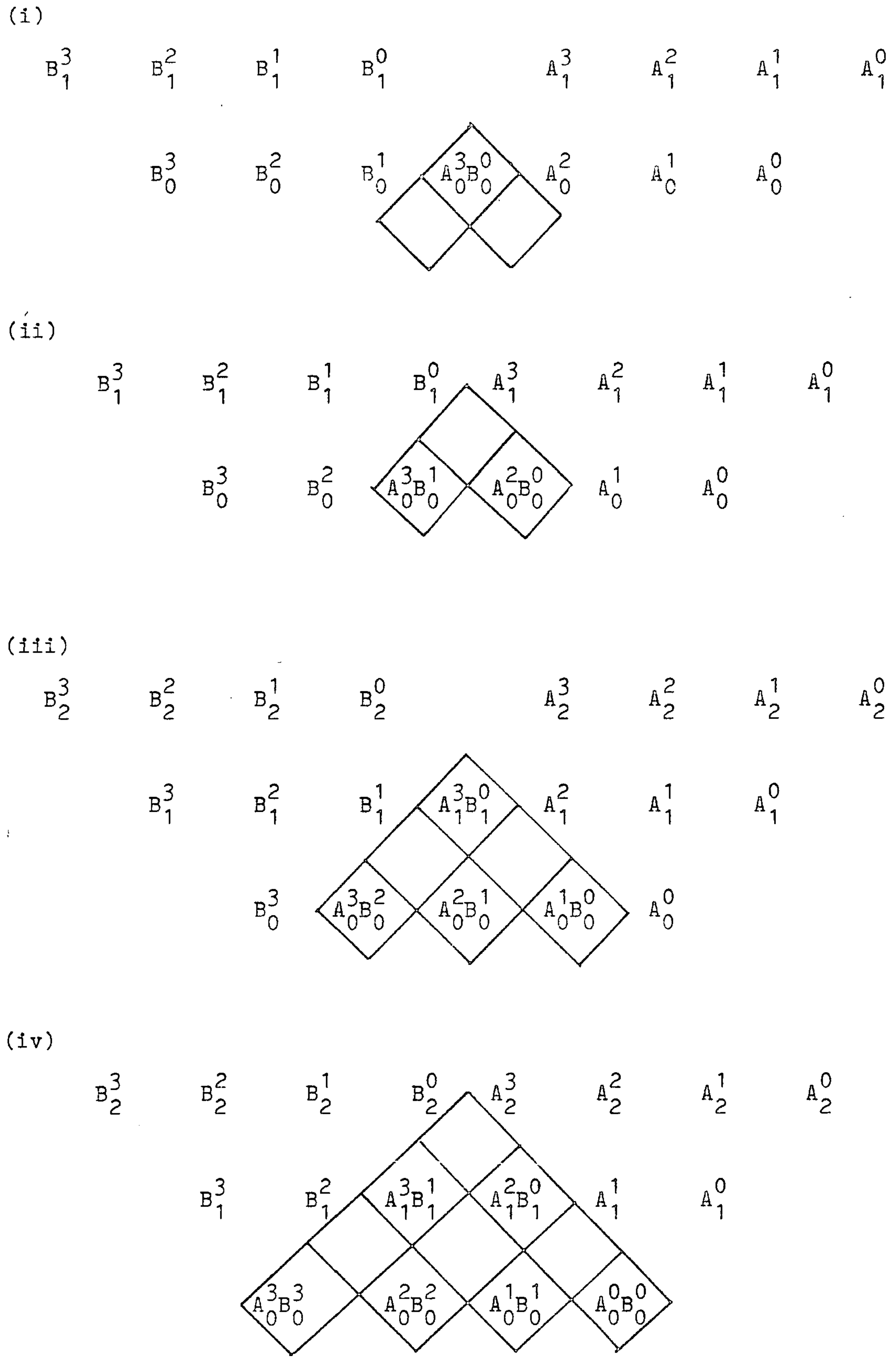

(i)
B_1^3 B_1^2 B_1^1 B_1^0 A_1^3 A_1^2 A_1^1 A_1^0
B_0^3 B_0^2 B_0^1 $A_0^3B_0^0$ A_0^2 A_0^1 A_0^0
(ii)
B_1^3 B_1^2 B_1^1 B_1^0 A_1^3 A_1^2 A_1^1 A_1^0
B_0^3 B_0^2 $A_0^3B_0^1$ $A_0^2B_0^0$ A_0^1 A_0^0
(iii)
B_2^3 B_2^2 B_2^1 B_2^0 A_2^3 A_2^2 A_2^1 A_2^0
B_1^3 B_1^2 B_1^1 $A_1^3B_1^0$ A_1^2 A_1^1 A_1^0
B_0^3 $A_0^3B_0^2$ $A_0^2B_0^1$ $A_0^1B_0^0$ A_0^0
(iv)
B_2^3 B_2^2 B_2^1 B_2^0 A_2^3 A_2^2 A_2^1 A_2^0
B_1^3 B_1^2 $A_1^3B_1^1$ $A_1^2B_1^0$ A_1^1 A_1^0
$A_0^3B_0^3$ $A_0^2B_0^2$ $A_0^1B_0^1$ $A_0^0B_0^0$

Figure 8.2
The First four cycles of the reduced multiplier interaction.

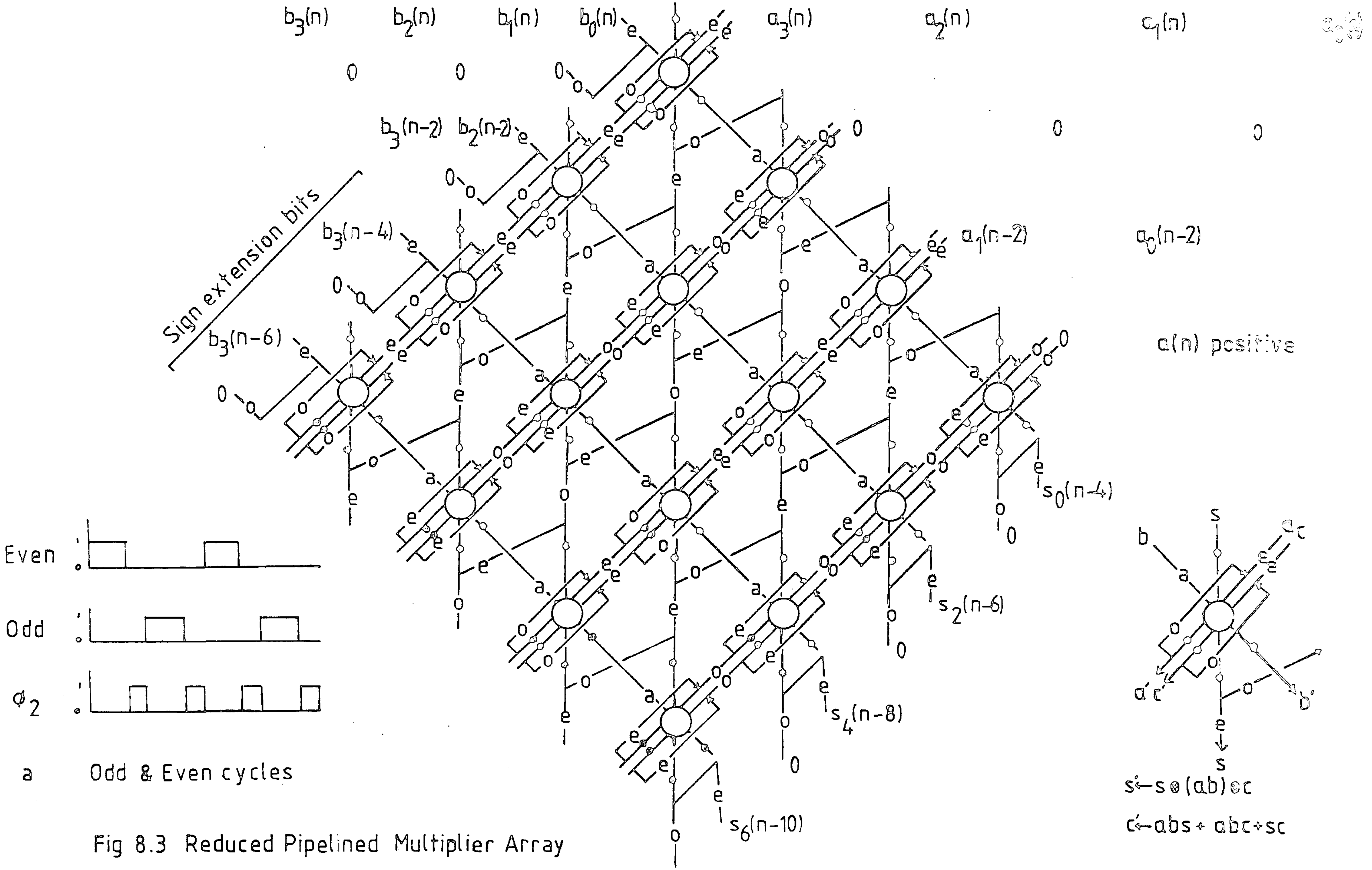


Fig 8.3 Reduced Pipelined Multiplier Array

8.5 Implementing a 15-point DCT

It appears that short length DCTs, around 16-points, are of most value in image compression. For example [8.9] is concerned with an elaborate implementation of a 16-point DCT. Appendix III gives a 15-point DCT algorithm derived from a 60-point WFTA.

This 15-point DCT algorithm uses 19 multiplications. So 19+15-1=33 single processors would be required in the A and C arrays. With the arrangement of the matrices as given a single column may be saved from the A array and 9 single columns from the C array. These savings come from considering the number of diagonals in the corners of these arrays which only contain 0. These figures imply 16 double cells for the A array and 12 double cells for the C array. An alternate arrangement yields 15 double cells in each array.

In many image processing applications a restricted word length is sufficient, say 8 bits. A 15-point DCT with 8-bit data requires

A array	16x13	Double Cells	
C array	12x12*	Double cells	
Multiplier	8x8	Double Multiplier Cells	(8.7)

* Inspection of the 15-point DCT C matrix shows that no sum involves more than 8 values, thus restricting the maximum wordlength to 12-bits. These figures assume that the wordlength is truncated back to 8 bits at the end of the A array and the multipler.

Even allowing for the increased cell sizes it seems feasible to integrate this entire 8-bit 15-point DCT onto a single chip using, say, 3μ CMOS. With 3μ CMOS such a chip would be capable of accepting domestic TV data rates.

8.6 Summary of Chapter 8

This chapter has shown the development of a fast algorithm for the Discrete Cosine Transform based upon the Winograd Fourier Transform Algorithm. This algorithm may be implemented using compacted versions of the one-bit systolic arrays discussed in chapter 6.

Chapter 9

Conclusions and Suggestions for further work

9.1 Conclusions and Summary

The first five chapters of this thesis concentrated on the development of algorithms for the computation of cyclic convolutions and DFTs. An important general form derived in chapter one is

$$y = C\ (Ax \times Bh\ ) \qquad (1.20)$$

Equation (1.20) describes the general form of Winograd's cyclic convolution and DFT algorithms which are discussed in chapter 2. Nussbaumer's two-dimensional cyclic convolution algorithms based upon polynomial transforms are also of the general form of (1.20), see chapter 3. A feature of many of these algorithms is that the A and C matrices contain only +1, -1 and 0. The product Bh is precalculated and given as series of coefficients. The Cyclic Convolution Property (CCP) can also be described by (1.20).

Chapter 4, which discusses multi-dimensional mappings, shows that algorithms of the form of (1.20) may be nested together to perform longer convolutions and DFTs with the nested structures still retaining the general form of (1.20).

Chapter 5 compared the number of arithmetic operations for a variety of convolution and DFT algorithms. In terms of the number of multiplications the following conclusions were drawn.

(i) One-dimensional DFTs should be calculated by the WFTA.

(ii) One-dimensional cyclic convolutions for real data up to 90 points in length should be calculated by nested one-dimensional 'short-N' cyclic convolution algorithms.

(iii) One-dimensional cyclic convolutions of real data for lengths greater than 90 points should be computed using complex WFTAs to calculate two real convolutions simultaneously.

(iv) Two-dimensional DFTs should be calculated using two-dimensional DFT algorithms based upon polynomial transforms. These algorithms are not discussed in detail, see below.

(v) Two-dimensional cyclic convolutions of real data up to 60x60 points should be computed using two-dimensional cyclic convolution algorithms based upon polynomial transforms.

(vi) Two-dimensional cyclic convolutions of real data for greater than 60x60 points should be calculated by two-dimensional complex DFTs based upon polynomial transforms.

(vii) Finally in chapter 5 the optimal algorithm and block size for the convolution of a picture with a fixed window was found. For up to 8x8 windows convolution algorithms should be employed, for greater size windows two-dimensional DFTs algorithms should be employed.

Chapter 6 proposed an implementation of algorithms of the form of (1.20) based upon the one-bit systolic arrays of McWhirter and McCanny. This architecture has several attractive features for implementation in VLSI. These features include regularity and short interconnection between cells. The throughput rate of this implementation is limited by the time to perform a one-bit full addition.

Chapter 7 then considers whether or not this implementation offers any advantages over more conventional techniques, such as the FFT. A Cost Function is developed which allows comparisons between different implementations of the same algorithm and between dissimilar algorithms. The cost function is based upon the number of gates used to realise the function and the throughput rate. All features of an implementation are considered, including address generation, control and other overheads. When evaluated the cost function shows that the less overheads, in the form of ROM, RAM, latches etc., the better. The systolic WFTA offers the lowest cost function of all the algorithms considered for transform lengths up to about 30 points.

Finally, in chapter 8 an algorithm for the Discrete Cosine Transform (DCT) is derived from the WFTA. This algorithm has the general form of (1.20). As the DCT involves no complex arithmetic it may be implemented by modified versions of the systolic arrays of chapter 6. These modifications allow the number of cells required in the arrays to be halved.

9.2 Suggestions for further work

An area not investigated was the derivation of two- and multi-dimensional DFTS by techniques other than the nesting of one-dimensional WFTAs. Nussbaumer [3.2-3.6] has done much work in this area and it appears that DFT algorithms derived using polynomial transforms will have the general form of (1.20), perhaps with A and C matrices containing only +1, -1 and 0. Recently Auslander, Feig and Winograd [3.13] have published a different way of deriving multi-dimensional DFT algorithms. It would be interesting to consider the relationship between the two techniques.

Much further work needs to be done on systolic arrays, particularly in ascertaining the relative merits of different systolic implementations of the same function. For example, is it better to treat a DFT as a matrix vector product and use a more general purpose array rather than the WFTA implementation of Chapter 6? Or would it be better ~~to~~ to use a systolic correlator and Rader's theorem to calculate DFTs? There are many possibilities.

Another interesting comparison to make would be between bit-serial architectures and the bit parallel structures discussed here.

Chapter 1 References

1.1 Gold B. and Rader C.M.,
Digital Processing of Signals,
McGraw-Hill, New York, 1969, pp 203-213.

1.2 Pollard J.M.,
The Fast Fourier Transform in a finite field,
Mathematics of Computation, Vol. 25, No. 114, 1971,
pp 365-374.

1.3 Nussbaumer H.J.,
Digital Filtering Using Polynomial Transforms,
Electronics Letters 23rd June 1977, Vol. 13, No. 13,
pp 386-387

1.4 Nussbaumer H.J. and Quandalle P.,
Computation of Convolutions and Discrete Fourier Transforms by Polynomial Transforms,
IBM Journal of Research and Development, Vol. 22, No. 2, March 1978, pp 134-144.

1.5 Agawal R.C. and Cooley J.W.,
Algorithms for Digital Convolution,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-25, No. 5, 1977, pp 392-410.

1.6 Agarwal R.C. and Burrus C.S.,
Fast Convolution using Fermat Number Transforms with Applications to Digital Signal Processing,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-22, April 1974, pp87-99.

1.7 Rader C.M.,
Discrete Fourier Transforms when the number of samples is prime,
Proc. of the IEEE, Vol. 56, No. 6, 1968, pp 1107-1108.

1.8 Nagell T.,
Introduction to Number Theory,
Chelsea, New York, 1964.

1.9 Kolba D.F. and Parks T.W.,
A Prime Factor FFT Algorithm using High-Speed Convolution,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-25, No. 4, 1977, pp 281-294.

1.10 Nussbaumer H.J.,
Fast Fourier Transform and Convolution Algorithms,
Springer Series in Information Sciences, Vol. 2,
Springer-Verlag, Berlin 1981.

1.11 McClellan J.H. and Rader C.M.,
Number Theory in Digital Signal Processing,
Prentice-Hall, Englewood Cliffs, N.J., USA. 1979.

Chapter 2 References

2.1 Knuth, D.E.,
"Seminumerical Algorithms" in "The Art of Computer Programming"
Vol. 2, Addison-Wesley, Reading, Mass., USA, 1971.

2.2 Agarwal, R.C. and Cooley, J.W.,
New Algorithms for Digital Convolution.
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-25, No. 5, 1977, pp 392-410.

2.3 Nussbaumer, H.J.,
Fast Fourier Transform and Convolution Algorithms,
Springer-Verlag, Berlin, 1981.

2.4 McClellan, J.H. and Rader, C.M.,
Number Theory in Digital Signal Processing,
Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

2.5 Winograd, S.,
Some Bilinear forms whose multiplicative complexity depends upon the field of coefficients.
Mathematical Systems Theory, Vol. 10, 1977, pp169-180.

2.6 Winograd, S.,
On Computing the Discrete Fourier Transform,
Proc. National Academy of Sciences of the USA, Vol. 73, 1976, pp1005-1006.

2.7 Winograd, S.,
On Computing the Discrete Fourier Transform
Mathematics of Computation, Vol. 32, No. 141, 1978, pp 175-199.

2.8 Nagel, T.,
Introduction to Number Theory,
Chelsea, New York, 1964.

Chapter 3 References

3.1 Nussbaumer H.J.,
Digital filtering using polynomial transforms,
Electronics Letters 23rd June 1977, Vol. 13, No. 13, pp386-387.

3.2 Nussbaumer H.J. and Quandalle P.,
Computation of Convolutions and Discrete Fourier Transforms by Polynomial transforms,
IBM Journal of research and Development, Vol. 22, No. 2, 1978, pp134-144.

3.3 Nussbaumer H.J. and Quandalle P.,
Fast Computation of Discrete Fourier Transforms using Polynomial Transforms,
IEEE Trans. on Acoustics, Speech and Signal Processing,
Vol. ASSP-27, No. 2, 1979, pp169-181.

3.4 Nussbaumer H.J.,
Fast Polynomial Transform Algorithms for Digital Convolution,
IEEE Trans. on Acoustics, Speech and Signal Processing,
Vol. ASSP-28, No. 2, 1980, pp205-215.

3.5 Nussbaumer H.J.,
New Polynomial Transform Algorithms for Multidimensional DFTs and Convolutions,
IEEE Trans. on Acoustics, Speech and Signal Processing,
Vol. ASSP-29, No. 11, 1981, pp74-83.

3.6 Nussbaumer H.J.,
Fast Fourier Transform and Convolution Algorithms,
Springer-Verlag, Berlin, Heidelberg, New York, 1981.

3.7 Arambepola, B. and Rayner P.J.W.,
Efficient Transforms for Multidimensional Convolutions,
Electronics Letters, March 15th 1979, Vol. 15, pp189-190.

3.8 Arambepola, B. and Rayner P.J.W.,
Discrete Transforms over Polynomial Rings with Applications in Computing Multidimensional Convolutions.
IEEE Trans. on Acoustics, Speech and Signal Processing,
Vol. ASSP-28, No. 4, 1980, pp407-414.

3.9 Truong T.K., Reed I.S., Lipes R.G. and Wu C.,
On the Application of a Fast Polynomial Transform and the Chinese Remainder Theorem to Compute a Two-dimensional Convolution.
IEEE Trans. on Acoustics, Speech and Signal Processing,
Vol. ASSP-29, No. 1, 1981, pp91-97.

3.10 Reed I.S., Shao H.M. and Truong T.K.
Fast Polynomial transform and its implementation by computer
IEE Proc., Vol. 128, Pt. E, No. 1, March 1981, pp50-60

3.11 Martens J.B.,
Fast Polynomial Transforms for Two-dimensional Convolution,
IEEE Trans. on Acoustics, Speech and Signal Processing,
Vol. ASSP-30, No. 6, 1982, pp1007-1010,

3.12 Pei, Soo-Chang and Wu, Ja-Ling,
Fast Biased Polynomial Transforms for Two-Dimensional Convolutions,
Electronics Letters 23rd July 1981, Vol. 17, No. 15, pp547-548.

3.13 Auslander L., Feig E. and Winograd S.,
New Algorithms for the Multidimensional Discrete Fourier Transform,
IEEE Trans. on Acoustics, Speech and Signal Processing,
Vol. ASSP-31, No. 2, 1983, pp388-403.

Chapter 4 References

4.1 Winograd S.,
Some Bilinear forms whose Multiplicative Complexity Depends on the Field of Coefficients
Mathematical Systems Theory, Vol. 10, 1977, pp 169-180.

4.2 Cooley J.W. and Tukey J.W.,
An Algorithm for the Machine Calculation of Complex Fourier Series,
Mathematics of Computing, Vol. 19, 1965, pp 297-301.

4.3 Brigham, E.O.,
The Fast Fourier Transform,
Prentice-Hall, 1974.

4.4 Burrus C.S.,
Index Mappings for Multidimensional Formulation of the DFT and Convolution,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-25, No. 3, 1977, pp 239-242.

4.5 Good I.J.,
The Relationship between Two Fast Fourier Transforms,
IEEE Trans. on COmputers,
Vol. C-20, 1971, pp 310-317.

4.6 Kolba D.P. and Parks T.W.,
A Prime Factor FFT Algorithm using High-Speed Convolution,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol ASSP-25, No. 4, 1977, pp 281-294.

4.7 Agarwal R.C.,
Comments on 'A Prime Factor FFT Algorithm using High-Speed Convolution,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-26, No. 3, 1978, p 254.

4.8 Burrus C.S. and Eschenbacher P.W.,
An In-Place, In-Order Prime Factor FFT Algorithm,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-29, No. 4, 1981, pp 806-816.

4.9 Rothweiler J.H.,
Implementation of the In-Order Prime Factor Transform for various sizes,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-30, No. 11, 1982, pp 105-107.

4.10 Arambepola B.,
Discrete Fourier Transform Processor Based on the Prime Factor Algorithm,
Accepted for publication in the Proceedings of the IEE.

4.11 Roa C.R. and Mitra S.K.,
Generalised Inverse of Matrices and its Applications,
Wiley, New York, 1971.

4.12 Winograd S.,
On Computing the Discrete Fourier Transform,
Proceedings of the National Academy of Sciences of the USA,
Vol. 73, 1976, pp 1005-1006.

4.13 Winograd S.,
On Computing the Discrete Fourier Transform,
Mathematics of Computation, Vol. 32, No. 141, 1978, pp 175-199.

4.14 Silverman H.F.,
An Introduction to Programming the Winograd Fourier Transform Algorithm (WFTA),
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-25, No. 2, 1977, pp 152-165.

4.15 Silverman H.F.,
Correction and Addendum to 4.14,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-26, 1978, p 268.

4.16 Silverman H.F.,
A Method for programming the Complex Genera-N Winograd Fourier Transform,
IEEE International Conference on Acoustics Speech and Signal Processing, Hartford, Conn., USA, May 9-11, 1977, pp 369-72.

4.17 Agarwal R.C. and Cooley J.W.,
New Algorithms for Digital Convolution,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-25, NO. 5, 1977, pp 392-410.

4.18 Agarwal R.C. and Burrus C.S.,
Fast One-dimensional Digital Convolution by Multidimensional Techniques,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-22, No. 1, 1974, pp 1-10.

4.19 Johnson H.W. and Burrus C.S.,
The Design of Optimal DFT Algorithms Using DYnamic Programming,
IEEE Trans. on Acoustics Speech and Signal Processing,
Vol. ASSP-31, No. 2, 1983, pp 378-387.

Chapter 5 References

5.1 Nussbaumer H.J.,
Fast Fourier Transform and Convolution Algorithms,
Springer-Verlag, Berlin, Heidelberg, New York, 1981.

5.2 Burrus C.S. and Eschenbacher P.W.,
An In-place, In-order Prime Factor FFT Algorithm,
Trans. of the IEEE on Acoustics Speech and Signal Processing
Vol. ASSP-29, No. 4, 1981, pp806-817.

5.3 Rothweiler J.H.,
Implementation of the In-order Prime Factor Transform for variable sizes,
Trans. of the IEEE on Acoustics Speech and Signal Processing
Vol. ASSP-30, No. 1, 1982, pp105-107.

5.4 Johnson H.W. and Burrus C.S.,
The Design of Optimal DFT Algorithms using Dynamic Programming,
Trans. of the IEEE on Acoustics Speech and Signal Processing
Vol. ASSP-31, No.2, 1983, pp378-387.

5.5 Morris L.R.,
A Comparative study of time efficient FFT and WFTA programs for general purpose computers.
Trans. of the IEEE on Acoustics Speech and Signal Processing
Vol. ASSP-26, 1978, pp141-150.

5.6 Nawab H. and McClellan J.H.,
Bounds on the minimum number of data tranfers in WFTA and FFT programs,
Trans. of the IEEE on Acoustics Speech and Signal Processing
Vol. ASSP-27, No. 4, 1979, pp394-398.

5.7 Nawab H. and McClellan J.H.,
Corrections to [5.6],
Trans. of the IEEE on Acoustics Speech and Signal Processing
Vol. ASSP-28, No. 4, 1980, pp480-481.

5.8 Blanken J.D. and Rustan D.L.,
Selection Criteria for Efficient Implemenation of FFT Algorithms,
Trans. of the IEEE on Acoustics Speech and Signal Processing
Vol. ASSP-30, No. 1, 1982, pp107-109.

5.9 Nussbaumer H.J. and Quandalle P.,
Computation of Convolutions and Fourier transforms by Polynomial transforms,
IBM Research and Development Journal, Vol. 22, 1978, pp134-144.

5.10 Nussbaumer H.J. and Quandalle P.,
Fast Computation of Discrete Fourier Transforms using polynomial transforms,
Trans. of the IEEE on Acoustics Speech and Signal Processing
Vol. ASSP-27, 1979, pp169-181.

Chapter 6 References

6.1 McCanny J.V. and McWhirter J.G.,
Implementation of Signal Processing Functions using 1-bit Systolic Arrays,
Electronics Letters, 18th March 1982, Vol. 18, pp241-243.

6.2 McCanny J.V. and McWhirter J.G.,
Completely iterative, pipelined multiplier suitable for VLSI,
IEE Proc., Vol. 129, Pt.G, No. 2, April 1982, pp40-46.

6.3 McCanny J.V. and McWhirter J.G.,
A bit level systolic array for matrix x vector multiplication,
To be published.

6.4 Ward J.S. and Stanier B.J.,
Implementaion of Convolution and Fourier Transform Algorithms using 1-bit systolic arrays,
Electronics Letters 2nd Sept. 1982, Vol. 18, pp799-801.

6.5 Ward J.S. and Stanier B.J.,
A Fast Discrete Cosine Transform for Systolic arrays,
Electronics Letters 20th Jan. 1983, Vol. 19, No. 2, pp58-60.

6.6 Patel K.K, Corry A.G. and McCabe A.P.H.,
A high performance Correlator based upon bit level systolic arrays,
To be published

6.7 McCabe M.M., McCabe A.P.H., Arambeploa B., Robinson I.N. and Gorry A.G.,
New Algorithms and Architectures for VLSI,
GEC Journal of Research and Development, Vol. 48, No. 2, 1982, pp68-75.

6.8 Myers D.J.,
Multipliers for LSI and VLSI Signal Processing Applications,
Edinburgh University MSc Project Report MSP5.
30th Sepember, 1981.

6.9 Kung H.T. and Leiserson C.E.,
"Algorithms for VLSI Processor Arrays",
Section 8.3 in [6.10] below.

6.10 Mead C. and Conway L.,
Introduction to VLSI systems,
Addison-Wesley, 1980, ISBN 0-201-04358-0.

Chapter 7 References

7.1 Ward J.S., Barton P., Roberts J.G.B. and Stanier B.J.,
Figures of Merit for VLSI Implementations of Digital Signal Processing Algorithms,
Submitted to IEE, Part F.

7.2 Barton P.,
Algorithms for VLSI DSP; Section 2.1. "FFT Discussion",
Presented at 'High Performance Logic Consortium - Working Group on Architectures and Algorithms', 2nd December 1981.

7.3 Savage J.E.,
Complexity of decoders: I-Classes of Decoding Rules
IEEE Transactions on Information Theory,
Vol. IT-15, 1969, pp 689-695.

7.4 Bajoga B.G. and Walbesser W.J.,
Decoder Complexity for BCH Codes
Proc. IEE, Vol. 120, 1973, pp 429-431.

7.5 Cooley J.W. and Tukey J.W.,
An Algorithm for the Machine Computation of Complex Fourier Series,
Mathematics of Computation, 1965, pp 297-301.

7.6 Gold B. and Bailly T.B.,
Parallelism in Fast Fourier Transform Hardware,
IEEE Transactions on Audio and Electroacoustics,
Vol. AU-21, No. 1, 1973, pp 5-16.

Chapter 8 References

8.1 Chen W. and Smith C.H.,
Adaptive coding of monchrome and colour images,
Transactions of the IEEE on Communications,
Vol. COM-25, 1977, pp1285-1292.

8.2 Rose J.A., Pratt W.K. and Robinson A.S.,
Interframe cosine transform image coding,
Transactions of the IEEE on Communications,
Vol. COM-25, 1977, pp1329-1339.

8.3 Ahmed N., Natarajan T. and Rao K.R.,
On image coding and the discrete cosine transform
Transactions of the IEEE on Computers,
Vol. C-23, 1974, pp90-93.

8.4 Haralick R.M.,
A storage efficient way to implement the discrete cosine transform,
Transactions of the IEEE on Computers,
Vol. C-25, 1976, pp764-765.

8.5 Narashim M.J. and Peterson A.M.,
On the Computation of the Discrete Cosine Transform,
Transactions of the IEEE on Communications,
Vol. COM-26, 1977, pp934-936.

8.6 Hein D. and Ahmed N.,
On a real-time Walsh-Hadamard/Cosine transform image processor,
IEEE Tran., 1978, Vol. EMC-20, pp453-457.

8.7 Ghanbari M. and Pearson D.E.,
Fact Cosine Transform implementation for television signals,
IEE Proc. Vol. 129, Part F, No. 1, 1982, pp59-68.

8.8 Winograd S.,
On Computing the Discrete Fourier transform,
Mathematics of Computation, Vol. 32, 1978, No. 141,
pp 175-199.

8.8 McCanny J.V. and McWhirter J.G.,
Completely iterative pipelined multiplier array suitable for VLSI,
IEE Proc. Vol. 129, Part G, No. 2, 1982, pp 40-46.

Appendix I

This appendix contains short-N convolution and DFT algorithms. All these algorithms have the general form of (1.20) with A and C matrices containing only +1, -1 and 0. Cyclic convolution algorithms for lengths 2, 3, 4, 5, 8, 9 and 16-points are given. The convolution algorithms are given as three matrices. The general scheme of the calculation is illustrated for the 2-point convolution algorithm. Note that the convolution algorithms calculate

$$y_l = \sum_{n=0}^{N-1} h_n x_{l-n} \qquad l=0,1,\ldots,N-1$$

A1.1 A 2-point Cyclic Convolution (M=2)

$$Ax = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \qquad\qquad Bh = \tfrac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix}$$

$$m = (Ax) \times (Bh)$$

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \end{bmatrix}$$

A1.2 A 3-point Cyclic Convolution (M=4)

$$A_3 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \qquad B_3 = \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ -1 & 2 & -1 \\ -2 & 1 & 1 \\ 1 & 1 & -2 \end{bmatrix} \qquad C_3 = \begin{bmatrix} 1 & -1 & -1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & -1 \end{bmatrix}$$

A1.3 A 4-point Cyclic Convolution (M=5)

$$A_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} \qquad B_4 = \tfrac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 2 & -2 & -2 & 2 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & 2 \end{bmatrix} \qquad C_4 = \begin{bmatrix} 1 & 1 & 1 & 1 & -1 \\ 1 & -1 & 0 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & 0 & 1 & 1 \end{bmatrix}$$

A1.4 A 5-point Cyclic Convolution (M=10)

$$A_5 = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 1 & 0 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad B_5 = \frac{1}{5} \begin{pmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 4 & -1 & -1 & -1 \\ 2 & 2 & -3 & 2 & -3 \\ -1 & -1 & 4 & -1 & -1 \\ -1 & -1 & -1 & 4 & -1 \\ 2 & -3 & 2 & 2 & -3 \\ 2 & 2 & 2 & -3 & -3 \\ -3 & 2 & 2 & 2 & -3 \\ 1 & 1 & 1 & 1 & -4 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$C_5 = \begin{pmatrix} -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & -1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -1 & 1 \end{pmatrix}$$

A1.5 A 8-point Cyclic Convolution (M=14)

$$A_8 = \begin{pmatrix} 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 \\ 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 \\ 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \end{pmatrix} \qquad B_8 = \frac{1}{8} \begin{pmatrix} -4 & -4 & 4 & 4 & 4 & 4 & -4 & -4 \\ -4 & 4 & 4 & 4 & 4 & -4 & -4 & -4 \\ 4 & 4 & 4 & 4 & -4 & -4 & -4 & -4 \\ 4 & 4 & 4 & -4 & -4 & -4 & -4 & 4 \\ -2 & 2 & 2 & -2 & -2 & 2 & 2 & -2 \\ 2 & 2 & -2 & -2 & 2 & 2 & -2 & -2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 4 & 0 & 0 & -4 & -4 & 0 & 0 & 4 \\ 4 & 4 & 0 & 0 & -4 & -4 & 0 & 0 \\ 4 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 4 & 0 & 4 & 0 & -4 & 0 & -4 & 0 \\ -4 & 0 & 4 & 0 & 4 & 0 & -4 & 0 \\ 2 & 0 & -2 & 0 & 2 & 0 & -2 & 0 \end{pmatrix}$$

$$C_8 = \begin{pmatrix} 0 & 0 & 0 & -1 & 0 & -1 & 1 & 1 & 1 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 & 1 & 0 & 1 & -1 & 0 & 1 & -1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & -1 \\ 1 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & -1 & -1 \\ 0 & 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & 0 & -1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & -1 & 0 & -1 & 1 & -1 & 0 & 1 \\ 0 & -1 & 0 & 0 & 0 & 1 & 1 & 1 & -1 & 0 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 1 & -1 \end{pmatrix}$$

A1.6 A 9-point Cyclic Convolution Algorithm (M=22)

$$A_9 = \left(\begin{array}{rrrrrrrrr}
0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 & 0 \\
1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 \\
0 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 \\
0 & 0 & 0 & 1 & 0 & 1 & -1 & 0 & -1 \\
1 & 0 & 1 & -1 & 0 & -1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 \\
0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\
0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\
0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 \\
1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 \\
1 & 0 & -1 & 1 & 0 & -1 & 1 & 0 & -1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}\right)
\qquad
B_9 = \frac{1}{18}\left(\begin{array}{rrrrrrrrr}
0 & 0 & -6 & 0 & 0 & 3 & 0 & 0 & 3 \\
0 & 0 & -3 & 0 & 0 & -3 & 0 & 0 & 6 \\
0 & 0 & -3 & 0 & 0 & 6 & 0 & 0 & -3 \\
-6 & 0 & 0 & 3 & 0 & 0 & 3 & 0 & 0 \\
-3 & 0 & 0 & -3 & 0 & 0 & 6 & 0 & 0 \\
-3 & 0 & 0 & 6 & 0 & 0 & -3 & 0 & 0 \\
0 & -6 & 0 & 0 & 3 & 0 & 0 & 3 & 0 \\
0 & -3 & 0 & 0 & -3 & 0 & 0 & 6 & 0 \\
0 & -3 & 0 & 0 & 6 & 0 & 0 & -3 & 0 \\
3 & 6 & 6 & -6 & -3 & -3 & 3 & -3 & -3 \\
6 & 3 & 3 & -3 & 3 & 3 & -3 & -6 & -6 \\
-3 & 3 & 3 & -3 & -6 & -6 & 6 & 3 & 3 \\
6 & -6 & 6 & -3 & 3 & -3 & -3 & 3 & -3 \\
3 & -3 & 3 & 3 & -3 & 3 & -6 & 6 & -6 \\
3 & -3 & 3 & -6 & 6 & -6 & 3 & -3 & 3 \\
6 & 6 & 3 & -3 & -3 & 3 & -3 & -3 & -6 \\
3 & 3 & -3 & 3 & 3 & 6 & -6 & -6 & -3 \\
3 & 3 & 6 & -6 & -6 & -3 & 3 & 3 & -3 \\
1 & -2 & 1 & 1 & -2 & 1 & 1 & -2 & 1 \\
2 & -1 & -1 & 2 & -1 & -1 & 2 & -1 & -1 \\
-1 & -1 & 2 & -1 & -1 & 2 & -1 & -1 & 2 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}\right)$$

$$C_9 = \left(\begin{array}{rrrrrrrrrrrrrrrrrrrrrr}
1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & -1 & -1 & 1 \\
0 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & -1 & 0 & 1 & 1 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & -1 & -1 & 1 \\
-1 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 & 1 \\
-1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 1
\end{array}\right)$$

A1.7 A 16-point Cyclic Convolution Algorithm (M=41)

$$A_{16} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\
1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\
0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\
0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\
0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\
1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 \\
1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\
0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 \\
0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

$$B_{16} = \frac{1}{16}\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\
-2 & 0 & 2 & 0 & -2 & 0 & 2 & 0 & -2 & 0 & 2 & 0 & -2 & 0 & 2 & 0 \\
-2 & -2 & 2 & 2 & -2 & -2 & 2 & 2 & -2 & -2 & 2 & 2 & -2 & -2 & 2 & 2 \\
2 & -2 & -2 & 2 & 2 & -2 & -2 & 2 & 2 & -2 & -2 & 2 & 2 & -2 & -2 & 2 \\
-4 & 0 & 0 & 4 & 4 & 0 & 0 & -4 & -4 & 0 & 0 & 4 & 4 & 0 & 0 & -4 \\
-4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & -4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\
-4 & -4 & 0 & 0 & 4 & 4 & 0 & 0 & -4 & -4 & 0 & 0 & 4 & 4 & 0 & 0 \\
-4 & -4 & -4 & 4 & 4 & 4 & 4 & -4 & -4 & -4 & -4 & 4 & 4 & 4 & 4 & -4 \\
-4 & 0 & -4 & 0 & 4 & 0 & 4 & 0 & -4 & 0 & -4 & 0 & 4 & 0 & 4 & 0 \\
-4 & -4 & -4 & -4 & 4 & 4 & 4 & 4 & -4 & -4 & -4 & -4 & 4 & 4 & 4 & 4 \\
4 & -4 & -4 & -4 & -4 & 4 & 4 & 4 & 4 & -4 & -4 & -4 & -4 & 4 & 4 & 4 \\
4 & 0 & -4 & 0 & -4 & 0 & 4 & 0 & 4 & 0 & -4 & 0 & -4 & 0 & 4 & 0 \\
4 & 4 & -4 & -4 & -4 & -4 & 4 & 4 & 4 & 4 & -4 & -4 & -4 & -4 & 4 & 4 \\
-8 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 8 & 0 & 0 & 0 & 0 & 0 & -8 & 0 \\
-8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-8 & 0 & -8 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\
-8 & 0 & -8 & 0 & -8 & 0 & 8 & 0 & 8 & 0 & 8 & 0 & 8 & 0 & -8 & 0 \\
-8 & 0 & 0 & 0 & -8 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\
-8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & 8 & 0 & 8 & 0 & 8 & 0 & 8 & 0 \\
8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & 8 & 0 & 8 & 0 & 8 & 0 \\
8 & 0 & 0 & 0 & -8 & 0 & 0 & 0 & -8 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\
8 & 0 & 8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & 8 & 0 & 8 & 0 \\
-8 & -8 & 0 & 0 & 0 & 0 & 8 & 8 & 8 & 8 & 0 & 0 & 0 & 0 & -8 & -8 \\
-8 & -8 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\
-8 & -8 & -8 & -8 & 0 & 0 & 0 & 0 & 8 & 8 & 8 & 8 & 0 & 0 & 0 & 0 \\
-8 & -8 & -8 & -8 & -8 & -8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & -8 & -8 \\
-8 & -8 & 0 & 0 & -8 & -8 & 0 & 0 & 8 & 8 & 0 & 0 & 8 & 8 & 0 & 0 \\
-8 & -8 & -8 & -8 & -8 & -8 & -8 & -8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\
8 & 8 & -8 & -8 & -8 & -8 & -8 & -8 & -8 & -8 & 8 & 8 & 8 & 8 & 8 & 8 \\
8 & 8 & 0 & 0 & -8 & -8 & 0 & 0 & -8 & -8 & 0 & 0 & 8 & 8 & 0 & 0 \\
8 & 8 & 8 & 8 & -8 & -8 & -8 & -8 & -8 & -8 & -8 & -8 & 8 & 8 & 8 & 8 \\
0 & -8 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 8 & 0 & 0 & 0 & 0 & 0 & -8 \\
0 & -8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -8 & 0 & -8 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 8 & 0 & 0 & 0 & 0 \\
0 & -8 & 0 & -8 & 0 & -8 & 0 & 8 & 0 & 8 & 0 & 8 & 0 & 8 & 0 & -8 \\
0 & -8 & 0 & 0 & 0 & -8 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 8 & 0 & 0 \\
0 & -8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & 8 & 0 & 8 & 0 & 8 & 0 & 8 \\
0 & 8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & 8 & 0 & 8 & 0 & 8 \\
0 & 8 & 0 & 0 & 0 & -8 & 0 & 0 & 0 & -8 & 0 & 0 & 0 & 8 & 0 & 0 \\
0 & 8 & 0 & 8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & -8 & 0 & 8 & 0 & 8
\end{pmatrix}$$

C_{16} first 20 columns

$$\begin{pmatrix}
1 & -1 & -1 & 1 & C & -1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 & 0 \\
1 & 1 & -1 & 0 & -1 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 & 0 \\
1 & -1 & 1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 1 & -1 & 0 & -1 & 1 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 1 & -1 \\
1 & -1 & -1 & 1 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & 0 & -1 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & -1 & 1 & 0 & 0 & 0 \\
1 & -1 & -1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 & 0 \\
1 & 1 & -1 & 0 & -1 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 & 0 \\
1 & -1 & 1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & 1 & 0 & 1 & -1 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & -1 & 1 \\
1 & -1 & -1 & 1 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & 0 & -1 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0
\end{pmatrix}$$

C_{16} columns 21 to 41

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 \\
0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 \\
-1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
0 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 \\
-1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1
\end{pmatrix}$$

The second half of Appendix I contains DFT algorithms for 2, 3, 4, 5, 7, 8, 9, 11, 13 and 16-points. All these algorithms are given as two matrices and list of coefficients. The coefficients represents the precalculated product (Bh). The number of multiplications is that for real data.

A1.8 A 2-point DFT Algorithm (M=2)

$$A_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad C_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{aligned} m_0 &= 1 \\ m_1 &= 1 \end{aligned}$$

A1.9 A 3-point DFT Algorithm (M=3)

$$A_3 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & -1 \end{bmatrix} \qquad C_3 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \qquad \begin{aligned} u_3 &= 2\pi/3 \\ m_0 &= 1 \\ m_1 &= (\cos u_3 - 1) \\ m_2 &= i \sin u_3 \end{aligned}$$

A1.10 A 4-point DFT Algorithm (M=4)

$$A_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \qquad C_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \qquad \begin{aligned} m_0 &= 1 \\ m_1 &= 1 \\ m_2 &= 1 \\ m_3 &= i \end{aligned}$$

A1.11 A 5-point DFT Algorithm (M=6)

$$A_5 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & 1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & -1 & 1 & 0 \end{bmatrix} \qquad C_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & -1 & 0 \\ 1 & 1 & -1 & 0 & 1 & 1 \\ 1 & 1 & -1 & 0 & -1 & -1 \\ 1 & 1 & 1 & -1 & 1 & 0 \end{bmatrix} \qquad \begin{aligned} u_5 &= 2\pi/5 \\ m_0 &= 1 \\ m_1 &= \tfrac{1}{2}(\cos u_5 + \cos 2u_5) - 1 \\ m_2 &= \tfrac{1}{2}(\cos u_5 - \cos 2u_5) \\ m3 &= i(\sin u_5 + \sin 2u_5) \\ m_4 &= i \sin 2u_5 \\ m5 &= i(\sin u_5 - \sin 2u_5) \end{aligned}$$

A1.12 A 7-point DFT Algorithm (M=9)

$$A_7 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & -1 & 1 & -1 & -1 \\ 0 & 1 & 0 & 1 & -1 & 0 & -1 \\ 0 & 0 & -1 & -1 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 & -1 & 1 \end{bmatrix} \qquad C_7 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 0 & -1 & 1 & -1 & 0 & -1 \\ 1 & 1 & 0 & -1 & 1 & -1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 & 1 & 1 & 0 & -1 & 1 \\ 1 & 1 & -1 & 0 & -1 & -1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & -1 & -1 & -1 & 0 \end{bmatrix}$$

$u_7 = 2\pi/7$

$m_0 = 1$

$m_1 = \frac{1}{3}(\cos u_7+\cos 2u_7+\cos 3u_7)-1$

$m_2 = \frac{1}{3}(2\cos u_7-\cos 2u_7-\cos 3u_7)$

$m_3 = \frac{1}{3}(\cos u_7-2\cos 2u_7+\cos 3u_7)$

$m_4 = \frac{1}{3}(\cos u_7+\cos 2u_7-2\cos 3u_7)$

$m_5 = i\frac{1}{3}(\sin u_7+\sin 2u_7-\sin 3u_7)$

$m_6 = i\frac{1}{3}(2\sin u_7-\sin 2u_7+\sin 3u_7)$

$m_7 = i\frac{1}{3}(\sin u_7-2\sin 2u_7-\sin 3u_7)$

$m_8 = i\frac{1}{3}(\sin u_7+\sin 2u_7+2\sin 3u_7)$

A1.13 A 8-point DFT Algorithm (M=8)

$$A_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 \end{pmatrix} \qquad C_8 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \end{pmatrix}$$

$u_8 = 2\pi/8$

$m_0 = 1$

$m_1 = 1$

$m_2 = 1$

$m_3 = i \sin 2u_8$

$m_4 = 1$

$m_5 = i \sin 2u_8$

$m_6 = i \sin u_8$

$m_7 = \cos u_8$

A1.14 A 9-point DFT Algorithm (M=12)

$$A_9 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & 0 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix} \quad C_9 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & 1 & 1 & 0 & 0 & -1 & -1 & -1 & 0 & 0 \\ 1 & -1 & 1 & 0 & -1 & 1 & 0 & 1 & 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 1 & -1 & 0 & -1 & 0 & -1 & 1 & 0 & 1 & 0 \\ 1 & -1 & 1 & -1 & 0 & -1 & 0 & 1 & -1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 1 & 0 & -1 & 1 & 0 & -1 & 0 & 1 & -1 & 0 \\ 1 & -1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

$u_9 = 2\pi/9$

$m_0 = 1$

$m_1 = 3/2$

$m_2 = -1$

$m_3 = \frac{1}{3}(2\cos u_9-\cos 2u_9-\cos 4u_9)$

$m_4 = \frac{1}{3}(\cos u_9+\cos 2u_9-2\cos 4u_9)$

$m_5 = \frac{1}{3}(\cos u_9-2\cos 2u_9+\cos 4u_9)$

$m_6 = -i \sin 3u_9$

$m_7 = -i \sin 3u_9$

$m_8 = i \sin u_9$

$m_9 = i \sin 4u_9$

$m_{10} = i \sin 2u_9$

$m_{11} = -3/2$

A1.15 A 11-point DFT Algorithm (M=21)

$$A_{11} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & 1 & 0 \\
0 & -1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 1 & -1 & -1 & 1 & -1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & 0 \\
0 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 \\
0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & -1 & 0 & -1 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 & 1 & 0 & 0 & -1 & 1 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 1 & -1 & 1 & -1 & 1 & 0 & -1 \\
0 & 1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1
\end{pmatrix}$$

$$B_{11} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & -1 & 1 & -1 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & -1 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & -1 & 1 \\
1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & -1 & 1 & -1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & -1 \\
1 & 0 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & -1 & -1 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & -1 & 1 \\
1 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1
\end{pmatrix}$$

$$u_{11} = 2\pi/11$$

$$m_0 = 1$$

$$m_1 = \frac{1}{5}(4\cos u_{11} - \cos 2u_{11} - \cos 3u_{11} - \cos 4u_{11} - \cos 5u_{11})$$

$$m_2 = -\frac{1}{5}(\cos u_{11} + \cos 2u_{11} + \cos 3u_{11} + \cos 4u_{11} - 4\cos 5u_{11})$$

$$m_3 = \frac{1}{5}(2\cos u_{11} - 3\cos 2u_{11} - 3\cos 3u_{11} + 2\cos 4u_{11} + 2\cos 5u_{11})$$

$$m_4 = -\frac{1}{5}(\cos u_{11} + \cos 2u_{11} - 4\cos 3u_{11} + \cos 4u_{11} + \cos 5u_{11})$$

$$m_5 = -\frac{1}{5}(\cos u_{11} + \cos 2u_{11} + \cos 3u_{11} - 4\cos 4u_{11} + \cos 5u_{11})$$

$$m_6 = \frac{1}{5}(2\cos u_{11} - 3\cos 2u_{11} + 2\cos 3u_{11} + 2\cos 4u_{11} - 3\cos 5u_{11})$$

$$m_7 = \frac{1}{5}(2\cos u_{11} - 3\cos 2u_{11} + 2\cos 3u_{11} - 3\cos 4u_{11} + 2\cos 5u_{11})$$

$$m_8 = -\frac{1}{5}(3\cos u_{11} + 2\cos 2u_{11} - 2\cos 3u_{11} - 2\cos 4u_{11} - 2\cos 5u_{11})$$

$$m_9 = \frac{1}{5}(\cos u_{11} - 4\cos 2u_{11} + \cos 3u_{11} + \cos 4u_{11} + \cos 5u_{11})$$

$$m_{10} = \frac{1}{5}(\cos u_{11}+\cos 2u_{11}+\cos 3u_{11}+\cos 4u_{11}+\cos 5u_{11})-1$$

$$m_{11} = i\frac{1}{5}(4\sin u_{11}+\sin 2u_{11}-\sin 3u_{11}-\sin 4u_{11}-\sin 5u_{11})$$

$$m_{12} = -i\frac{1}{5}(\sin u_{11}-\sin 2u_{11}+\sin 3u_{11}+\sin 4u_{11}-4\sin 5u_{11})$$

$$m_{13} = i\frac{1}{5}(2\sin u_{11}+3\sin 2u_{11}-3\sin 3u_{11}+2\sin 4u_{11}+2\sin 5u_{11})$$

$$m_{14} = -i\frac{1}{5}(\sin u_{11}-\sin 2u_{11}-4\sin 3u_{11}+\sin 4u_{11}+\sin 5u_{11})$$

$$m_{15} = -i\frac{1}{5}(\sin u_{11}-\sin 2u_{11}+\sin 3u_{11}-4\sin 4u_{11}+\sin 5u_{11})$$

$$m_{16} = i\frac{1}{5}(2\sin u_{11}+3\sin 2u_{11}+2\sin 3u_{11}+2\sin 4u_{11}-3\sin 5u_{11})$$

$$m_{17} = i\frac{1}{5}(2\sin u_{11}+3\sin 2u_{11}+2\sin 3u_{11}-3\sin 4u_{11}+2\sin 5u_{11})$$

$$m_{18} = -i\frac{1}{5}(3\sin u_{11}-3\sin 2u_{11}-2\sin 3u_{11}-2\sin 4u_{11}-2\sin 5u_{11})$$

$$m_{19} = i\frac{1}{5}(\sin u_{11}+4\sin 2u_{11}+\sin 3u_{11}+\sin 4u_{11}+\sin 5u_{11})$$

$$m_{20} = i\frac{1}{5}(\sin u_{11}-\sin 2u_{11}+\sin 3u_{11}+\sin 4u_{11}+\sin 5u_{11})$$

A1.16 A 13-point DFT Algorithm (M=21)

$$A_{13} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & -1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & 1 & -1 \\
0 & 0 & 1 & 1 & -1 & 0 & -1 & -1 & 0 & -1 & 1 & 1 & 0 \\
0 & -1 & 1 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 1 & 1 & -1 \\
0 & -1 & 0 & 0 & 1 & -1 & 1 & 1 & -1 & 1 & 0 & 0 & -1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & -1 & 0 \\
0 & -1 & 1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 \\
0 & -1 & -1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & 1 & 1 \\
0 & 0 & 1 & -1 & 1 & 0 & -1 & -1 & 0 & 1 & -1 & 1 & 0 \\
0 & 1 & 1 & -1 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & 1 & 1 \\
0 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & 0 & 0 & -1 \\
0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & -1 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & -1 & 1 & 0 & -1 & 0 & -1 & -1 \\
0 & -1 & 0 & 1 & 0 & -1 & 1 & -1 & 1 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\
0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1
\end{pmatrix}$$

$$C_{13} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & -1 & 0 & -1 & -1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 1 & -1 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & 0 & 1 & 1 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & -1 & 1 & -1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & -1 & 0 & -1 & 1 & 0 & 1 & 0 & -1 & 1 & -1 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & -1 & 0 & 1 & 0 & 1 & -1 & 0 & -1 & 0 & -1 & 1 & 0 & -1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & -1 & 0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 0 & 0 \\
1 & 1 & 1 & -1 & 0 & -1 & -1 & 0 & -1 & 0 & -1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & -1 & 1 & -1 & 0 & -1 & -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & -1 & -1 & 0 & -1 & -1 & 0 \\
1 & 1 & -1 & 0 & 1 & 1 & 1 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & -1 & 0 & -1 \\
1 & 1 & 1 & 0 & 1 & 1 & -1 & 0 & -1 & 1 & 1 & 0 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & -1 & 0 & -1 & 1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & -1 & 1
\end{bmatrix}$$

$$u_{13} = 2\pi/13$$

$$m_0 = 1$$

$$m_1 = \frac{1}{6}(\cos u_{13}+\cos 2u_{13}+\cos 3u_{13}+\cos 4u_{13}+\cos 5u_{13}+\cos 6u_{13})-1$$

$$m_2 = \frac{1}{6}(\cos u_{13}-\cos 2u_{13}+\cos 3u_{13}+\cos 4u_{13}-\cos 5u_{13}-\cos 6u_{13})$$

$$m_3 = \frac{1}{6}(\cos u_{13}-2\cos 2u_{13}-2\cos 3u_{13}+\cos 4u_{13}+\cos 5u_{13}+\cos 6u_{13})$$

$$m_4 = -\frac{1}{6}(\cos u_{13}+\cos 2u_{13}+\cos 3u_{13}-2\cos 4u_{13}+\cos 5u_{13}-2\cos 6u_{13})$$

$$m_5 = \frac{1}{6}(2\cos u_{13}-\cos 2u_{13}-\cos 3u_{13}-\cos 4u_{13}+2\cos 5u_{13}-\cos 6u_{13})$$

$$m_6 = -j\frac{1}{3}(\sin u_{13}+\sin 2u_{13}+\sin 3u_{13}-\sin 4u_{13}+\sin 5u_{13}+\sin 6u_{13})$$

$$m_7 = j\frac{1}{3}(\sin u_{13}+\sin 3u_{13}-\sin 4u_{13})$$

$$m_8 = -j\frac{1}{3}(\sin 2u_{13}+\sin 5u_{13}+\sin 6u_{13})$$

$$m_9 = -\frac{1}{6}(2\cos u_{13}+\cos 2u_{13}-\cos 3u_{13}-\cos 4u_{13}-2\cos 5u_{13}+\cos 6u_{13})$$

$$m_{10} = \frac{1}{6}(\cos u_{13}+2\cos 2u_{13}-2\cos 3u_{13}+\cos 4u_{13}-\cos 5u_{13}-\cos 6u_{13})$$

$$m_{11} = \frac{1}{6}(\cos u_{13}-\cos 2u_{13}+\cos 3u_{13}-2\cos 4u_{13}-\cos 5u_{13}+2\cos 6u_{13})$$

$$m_{12} = -j\frac{1}{3}(\sin u_{13}-\sin 2u_{13}-2\sin 3u_{13}-\sin 4u_{13}-\sin 5u_{13}+2\sin 6u_{13})$$

$$m_{13} = j\frac{1}{3}(2\sin u_{13}-2\sin 2u_{13}-\sin 3u_{13}+\sin 4u_{13}+\sin 5u_{13}+\sin 6u_{13})$$

$$m_{14} = -j\frac{1}{3}(\sin u_{13}-\sin 2u_{13}+\sin 3u_{13}+2\sin 4u_{13}+2\sin 5u_{13}-\sin 6u_{13})$$

$$m_{15} = -j\frac{1}{3}(\sin 2u_{13}+\sin 5u_{13}-2\sin 6u_{13})$$

$$m_{16} = j\frac{1}{3}(2\sin 2u_{13}-\sin 5u_{13}-\sin 6u_{13})$$

$$m_{17} = -j\frac{1}{3}(\sin 2u_{13}-2\sin 5u_{13}+\sin 6u_{13})$$

$$m_{18} = -j\frac{1}{3}(2\sin u_{13}-\sin 2u_{13}-\sin 3u_{13}+\sin 4u_{13}-\sin 5u_{13}+2\sin 6u_{13})$$

$$m_{19} = j\frac{1}{3}(\sin u_{13}-2\sin 2u_{13}+\sin 3u_{13}+2\sin 4u_{13}+\sin 5u_{13}+\sin 6u_{13})$$

$$m_{20} = j\frac{1}{3}(\sin u_{13}+\sin 2u_{13}-2\sin 3u_{13}-\sin 4u_{13}-2\sin 5u_{13}+\sin 6u_{13})$$

A1.17 A 16-point DFT ALgorithm (M=18)

$$A_{16} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\
0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\
1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\
0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 \\
0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & -1 & 0 & -1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0
\end{pmatrix}$$

$$C_{16} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & 0 & -1 & 1 & 0 & -1 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 0 & -1 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & 1 & 0 & 1 & -1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 0 & 1 & -1 & 0 & 1 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & -1 & 0 & 1 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & -1 & 0 & -1 & 1 & 0
\end{pmatrix}$$

$u_{16} = 2\pi/16$

$m_0 = 1$

$m_1 = 1$

$m_2 = 1$

$m_3 = i \sin 4u_{16}$

$m_4 = 1$

$m_5 = i \sin 4u_{16}$

$m_6 = i \sin 2u_{16}$

$m_7 = \cos 2u_{16}$

$m_8 = 1$

$m_9 = i \sin 4u_{16}$

$m_{10} = i \sin 2u_{16}$

$m_{11} = \cos 2u_{16}$

$m_{12} = i \sin 3u_{16}$

$m_{13} = i(\sin u_{16} - \sin 3u_{16})$

$m_{14} = i(\sin u_{16} + \sin 3u_{16})$

$m_{15} = \cos 3u_{16}$

$m_{16} = (\cos u_{16} + \cos 3u_{16})$

$m_{17} = (\cos 3u_{16} - \cos u_{16})$

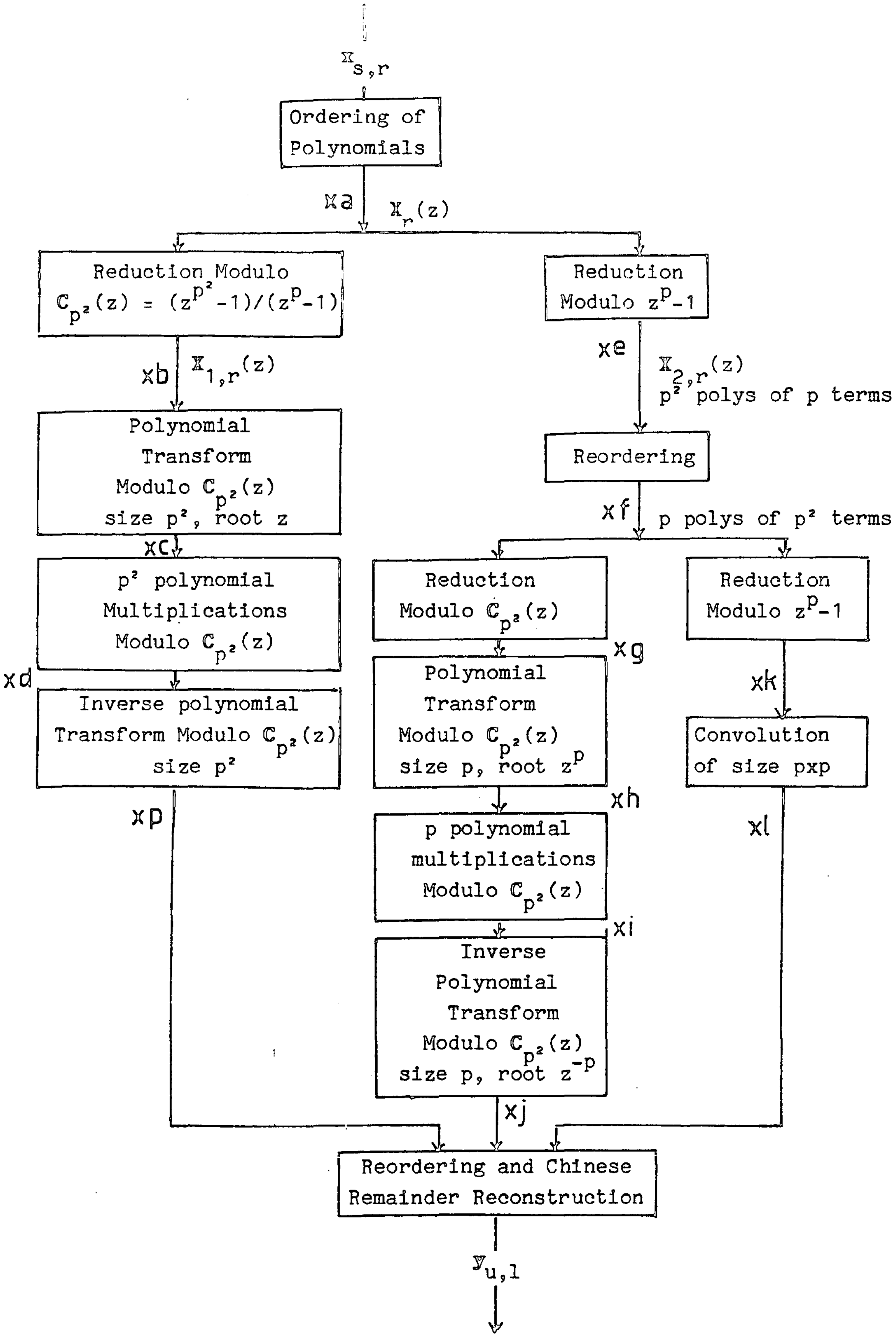
Appendix II

This appendix gives a FORTRAN program for the derivation of the matrices for a 9x9 cyclic convolution algorithm evaluated using polynomial transforms. The general scheme of the program is illustrated in the figure overleaf. In this figure the letters between boxes give the name of the array of the values at that point in the calculation. For example the input signal is initially contained in XA but after reduction modulo $C_9(z)$ is contained in array XB.

The program generates matrices for an untransposed system, i.e. the C matrix contains values other than +1, -1 and 0. The result should be normalised by 1/81. In the interests of size some of the statements have been compacted, particularly the DATA statements and as written may not compile. The program should be read down the lefthand column and then down the righthand column of each page.

```
C
C     TO GENERATE MATRICES FOR A  9X9  CYCLIC
C     CONVOLUTION   J.S.WARD 6/6/83
C
      INTEGER XA(9,9), HA(9,9), HB(9,6),
     &XC(9,6), HC(9,6), ROW, COL, POLY,
     &POWER, TEMPX6(6), TEMH6(6), TMPX18(18),
     &TMPH18(18), TEMPX9(9), TEMP9(9),
     &TMPX13(13), TMPH13(13), XE(9,3),
     &HE(9,3), XG(3,6), HG(3,6), XK(3,3),
     &HK(3,3), M(229),COEF(229), MP(229),
     &XL(3,3), XI(3,6), XJ(3,6), XM(3,9),
     &XN(9,3), XD(9,6),XP(9,6), XF(3,9),
     &HF(3,9), TEMPX3(3), TEMPH3(3),
     &TEMP13(13), HH(3,6), XH(3,6),
     &AM(229,81), BM(229,81), CM(81,229),
     &OUTROW, OUTCOL, ROW1, COL1, XB(9,6)
C
C     SET ONLY ONE INPUT VALUE TO 1
C
      DO 190 OUTROW = 1, 9
        DO 180 OUTCOL = 1, 9
          DO 20 ROW= 1, 9
            DO 10 COL = 1, 9
              XA(ROW,COL) = 0
              HA(ROW,COL) = 0
10          CONTINUE
20        CONTINUE
C
          XA(OUTROW,OUTCOL) = 1
          HA(OUTROW,OUTCOL) = 1
C
C     REDUCE X AND H MOD C9(Z)
```

```
C
          CALL REDC9(XA,XB)
          CALL REDC9(HA,HB)
C
C     FOR C9(Z) BRANCH POLY. TRANSFORM
C
          CALL POLYT1(XB, XC, .TRUE.)
          CALL POLYT1(HB, HC, .TRUE.)
C
C     PREMULTIPLICATION PART OF POLYNOMIAL
C         PRODUCT MOD C9(Z)
C
          DO 50 POLY = 1, 9
            DO 30 POWER = 1, 6
              TEMPX6(POWER) = XC(POLY,POWER)
              TEMPH6(POWER) = HC(POLY,POWER)
30          CONTINUE
C
      CALL PREC9(TEMPX6,TEMPH6,TMPX18,TEMPH18)
C
            DO 40 MULT = 1, 18
              M(18*(POLY-1)+MULT)=
     1                           TMPX18(MULT)
              COEF(18*(POLY-1)+MULT)=
     1                           TMPH18(MULT)
40          CONTINUE
50        CONTINUE
C
C     START THE C3(Z) C1(Z) PART
C     C3(Z) REDUCTIONS FIRST
C
          CALL REDC3(XA, XE)
          CALL REDC3(HA, HE)
```



$x_{s,r}$
Ordering of Polynomials
xa
$X_r(z)$
Reduction Modulo $C_{p^2}(z) = (z^{p^2}-1)/(z^p-1)$
Reduction Modulo z^p-1
xb
$X_{1,r}(z)$
xe
$X_{2,r}(z)$
p^2 polys of p terms
Polynomial Transform Modulo $C_{p^2}(z)$ size p^2, root z
Reordering
xf
p polys of p^2 terms
xc
p^2 polynomial Multiplications Modulo $C_{p^2}(z)$
Reduction Modulo $C_{p^2}(z)$
Reduction Modulo z^p-1
xg
xd
Inverse polynomial Transform Modulo $C_{p^2}(z)$ size p^2
Polynomial Transform Modulo $C_{p^2}(z)$ size p, root z^p
xk
Convolution of size pxp
xh
xp
p polynomial multiplications Modulo $C_{p^2}(z)$
xl
xi
Inverse Polynomial Transform Modulo $C_{p^2}(z)$ size p, root z^{-p}
xj
Reordering and Chinese Remainder Reconstruction
$y_{u,l}$

Computation of Convolution of p^2xp^2 points

```
C
C    TRANSPOSE THE ARRAYS
C
        DO 70 ROW = 1, 9
          DO 60 COL = 1, 3
            XF(COL,ROW) = XE(ROW,COL)
            HF(COL,ROW) = HE(ROW,COL)
60        CONTINUE
70      CONTINUE
C
C    FURTHER REDUCTION OF XF & HF MOD C9(Z)
C            AND C3(Z)
C
        DO 110 ROW = 1, 3
          DO 80 COL = 1, 9
            TEMPX9(COL) = XF(ROW,COL)
            TEMPH9(COL) = HF(ROW,COL)
80        CONTINUE
C
          CALL C9(TEMPX9, TEMPX6)
          CALL C9(TEMPH9, TEMPH6)
          CALL C3(TEMPX9, TEMPX3)
          CALL C3(TEMPH9, TEMPH3)
C
          DO 90 COL = 1, 6
            XG(ROW,COL) = TEMPX6(COL)
            HG(ROW,COL) = TEMPH6(COL)
90        CONTINUE
C
          DO 100 COL = 1, 3
            XK(ROW,COL) = TEMPX3(COL)
            HK(ROW,COL) = TEMPH3(COL)
100       CONTINUE
110     CONTINUE
C
C    POLY.TRANSFORM LENGHT 3 ROOT Z**3
C            MOD C9(Z)
C
        CALL POLYT2(XG, XH, .TRUE.)
        CALL POLYT1(HG, HH, .TRUE.)
C
C    3 POLY MULTS MOD C9(3)
C
        DO 140 POLY = 1, 3
          DO 120 POWER = 1, 6
            TEMPX6(POWER) = XH(POLY,POWER)
            TEMPH6(POWER) = HH(POLY,POWER)
120       CONTINUE
C
   CALL PREC9(TEMPX6,TEMPH6,TMPX18,TMPH18)
C
          DO 130 POWER = 1, 18
            M(162+(18*(POLY-1))+POWER)=
   &        TMPX18(POWER)
            COEF(162+18*(POLY-1)+POWER)=
   &        TMPH18(POWER)
130       CONTINUE
140     CONTINUE
```

```
C
C    CALL PRE-MULT PART OF 3X3 CONVOLUTION
C
        CALL PRE3X3(XK,HK,TMPX13,TMPH13)
C
        DO 150 I = 1, 13
          M(216 + I) = TMPX13(I)
          COEF(216 + I) = TMPH13(I)
150     CONTINUE
C========================================
C
C    TRANSFER VALUES TO MATRICES
C
        DO 170 ROW = 1, 22
      AM(ROW,9*(OUTROW-1)+OUTCOL)=M(ROW)
      BM(ROW,9*(OUTROW-1)+OUTCOL)=COEF(ROW)
170     CONTINUE
180   CONTINUE
190 CONTINUE
C=======================================
C
C    WRITE OUT MATRICES (BEWARE SYSTEM
C    LIMITS ON NUMBER OF CHARACTERS PER
C    LINE)
     WRITE(6,200)
200  FORMAT('19X9 CYCLIC CONV. A MATRIX',/)
     WRITE(6,210)((AM(I,J),J=1,81),I=1,229)
210  FORMAT(' ',81I3)
     WRITE(6,220)
220  FORMAT('1 B MATRIX ** UNTRANSPOSED
    & ** SYSTEM',/)
     WRITE(6,210)((BM(I,J),J=1,81),I=1,229)
C=======================================
C
C    START ON THE C MATRIX
C
     DO 420 OUTROW = 1, 229
       DO 230 ROW = 1, 229
         MP(ROW) = 0
230    CONTINUE
       MP(OUTROW) = 0
C
C    POST MULT 3X3 CONV OPERATIONS
C
       DO 240 I = 1, 13
         TEMP13(I) = MP(216 + I)
240    CONTINUE
C
       CALL PST3X3(TEMP13, XL)
C
C    POST MULTIPLICATION POLYNOMIAL
C    PRODUCTS MOD C9(Z)
C
       DO 270 POLY = 1 ,3
         DO 250 POWER = 1, 18
           TMPX1(POWER)=
    &      MP(162+(POWER-1)*18+POWER)
250      CONTINUE
```

```
C
         CALL POSTC9(TMPX18,TEMPX6)
C
         DO 260 POWER = 1, 6
           XI(POLY,POWER) = TEMPX6(POWER)
260      CONTINUE
270    CONTINUE
C
C   INVERSE POLY. TRANS. LENGTH 3 MOD C9(Z)
C
       CALL POLYT2(XI, XJ, .FALSE.)
C
C    CRT RECOMBINATION OF Z**3-1 BRANCH
C
       CALL CRT1(XJ, XL, XM)
C
C    TRANSPOSE ARRAY
C
       DO 290 POLY = 1,3
         DO 280 POWER = 1,9
           XN(POWER,POLY) = XM(POLY,POWER)
280      CONTINUE
290    CONTINUE
C
C    FINISH C9(Z) MAIN BRANCH
C
       DO 320 POLY = 1, 9
         DO 300 POWER = 1, 18
          TMPX18(POWER)=MP((POLY-1)*18+POWER
300      CONTINUE
C
         CALL POSTC9(TMPX18, TEMPX6)
C
         DO 310 POWER = 1, 6
           XD(POLY,POWER) = TEMPX6(POWER)
310      CONTINUE
320    CONTINUE
C
C   INVERSE POLY. TRANS. LENGHT 9 MOD C9(Z)
C
       CALL POLYT1(XD, XP, .FALSE.)
C
C    FINAL CRT RECOMBINATION
C
       DO 340 POLY = 1, 9
         DO 330 POWER = 1, 9
           OUT(POLY,POWER) = 0
330      CONTINUE
340    CONTINUE
C
       DO 390 POLY = 1, 9
         DO 350 POWER = 1, 6
           TEMPX6(POWER) = XP(POLY,POWER)
350      CONTINUE
C
         CALL R1(TEMPX6, TEMPX9)
C
         DO 360 POWER = 1, 9
```

```
           OUT(POLY,POWER)=
    1      OUT(POLY,POWER)-(3*TEMPX9(POWER))
360      CONTINUE
C
         DO 370 POWER = 1, 3
           TEMPX3(POWER) = XN(POLY,POWER)
370      CONTINUE
C
         CALL R2(TEMPX3, TEMPX9)
C
         DO 380 POWER = 1, 9
           OUT(POLY,POWER)=OUT(POLY,POWER)+
    1      TEMPX9(POWER)
380      CONTINUE
390    CONTINUE
C=========================================
C
       DO 410 ROW = 1, 9
         DO 400 COL = 1, 9
           CM(9*(ROW-1)+COL,OUTROW)=
    &      OUT(ROW,COL)
400      CONTINUE
410    CONTINUE
420  CONTINUE
C
     WRITE (6,430)
430  FORMAT('1 C MATRIX ** UNTRANSPOSED
    &SYSTEM',/)
     WRITE (6,440)((CM(I,J),J=1,229),I=1,81)
440  FORMAT(' ',229I3)
     STOP
     END
C****************************************************
C
C    REDUCTIONS MOD C9(Z) - CALLS NEXT
C             ROUTINE
C
     SUBROUTINE REDC9(IN, OUT)
     INTEGER IN(9,9),OUT(9,6),ROW,COL
    &,I(9),O(6)
C
     DO 30 ROW= 1, 9
       DO 10 COL= 1, 9
         I(COL) = IN(ROW,COL)
10     CONTINUE
C
     CALL C9(I,O)
C
       DO 20 COL = 1, 6
         OUT(ROW,COL)= O(COL)
20     CONTINUE
30   CONTINUE
     RETURN
     END
C+++++++++++++++++++++++++++++++++++++++++
C
C    ACTUAL REDUCTIONS MOD C9(Z)
C
```

```
      SUBROUTINE C9(IN, OUT)
      INTEGER IN(9), OUT(6)
      OUT(1) = IN(1) - IN(7)
      OUT(2) = IN(2) - IN(8)
      OUT(3) = IN(3) - IN(9)
      OUT(4) = IN(4) - IN(7)
      OUT(5) = IN(5) - IN(8)
      OUT(6) = IN(6) - IN(9)
      RETURN
      END
C+++++++++++++++++++++++++++++++++++++++++
C
C     REDUCTIONS MOD C3(Z)
C     CALLS NEXT ROUTINE
C
      SUBROUTINE REDC3(IN, OUT)
      INTEGER IN(9,9), OUT(9,3), ROW, COL
     &,I(9), O(3)
C
      DO 30 ROW= 1, 9
        DO 10 COL = 1, 9
          I(COL) = IN(ROW,COL)
10      CONTINUE
C
      CALL C3(I,O)
C
        DO 20 COL = 1, 3
          OUT(ROW,COL) = O(COL)
20      CONTINUE
30    CONTINUE
      RETURN
      END
C+++++++++++++++++++++++++++++++++++++++++
C
C     ACTUAL REDUCTIONS MOD C3(Z)
C
      SUBROUTINE C3(IN, OUT)
      INTEGER IN(9), OUT(3)
C
      OUT(1) = IN(1) + IN(4) + IN(7)
      OUT(2) = IN(2) + IN(5) + IN(8)
      OUT(3) = IN(3) + IN(6) + IN(9)
      RETURN
      END
C+++++++++++++++++++++++++++++++++++++++++
C
C     PRE-MULTIPICATION POLYNOMIAL
C     MULTIPLICATION  MOD C9(Z) (M=18)
C
      SUBROUTINE PREC9(X, H, M, COEF)
      INTEGER X(6), H(6), M(18), COEF(18)
C
      M(1) = X(4) + X(5)
      M(2) = (X(1) + X(2)) - (X(4) + X(5))
      M(3) = X(1) + X(2)
C
      M(4) = X(5) + X(6)
      M(5) = ((X(2) + X(3)) - (X(5) + X(6))
      M(6) = X(2) + X(3)
C
      M(7) = X(4) + X(6)
      M(8) = ((X(1) + X(3)) - (X(4) + X(6))
      M(9) = X(1) + X(3)
C
      M(10) = X(4)
      M(11) = X(1) - X(4)
      M(12) = X(1)
C
      M(13) = X(5)
      M(14) = X(2) - X(5)
      M(15) = X(2)
C
      M(16) = X(6)
      M(17) = X(3) - X(6)
      M(18) = X(3)
C
      COEF(1) = (H(1) + H(2)) - (H(4) + H(5))
      COEF(2) = H(1) + H(2)
      COEF(3) = H(4) + H(5)
C
      COEF(4) = (H(2) + H(3)) - (H(5) + H(6))
      COEF(5) = H(2) + H(3)
      COEF(6) = H(5) + H(6)
C
      COEF(7) = (H(1) + H(3)) - (H(4) + H(6))
      COEF(8) = H(1) + H(3)
      COEF(9) = H(4) + H(6)
C
      COEF(10) = H(1) - H(4)
      COEF(11) = H(1)
      COEF(12) = H(4)
C
      COEF(13) = H(2) - H(5)
      COEF(14) = H(2)
      COEF(15) = H(5)
C
      COEF(16) = H(3) - H(6)
      COEF(17) = H(3)
      COEF(18) = H(6)
      RETURN
      END
C+++++++++++++++++++++++++++++++++++++++++
C
C       POST MULT PART OF MOD C9(Z) POLY PROD
C
      SUBROUTINE POSTC9(IN, OUT)
      IMPLICIT INTEGER (P, Q, R, S, T)
      INTEGER IN(18), OUT(6), F(18)
C
      DO 10 I = 1, 6
        F(2*I-1) = IN(3*I-2) + IN(3*I-1)
        F(2*I)   = IN(3*I-2) + IN(3*I)
10      CONTINUE
C
      T0 = F(7)
      S0 = F(1) - F(7) - F(9)
      R0 = F(5) - F(7) + F(9) - F(11)
      Q0 = F(3) - F89) - F(11)
      P0 = F(11)
C
      T1 = F(8)
      S1 = F(2) - F(8) - F(10)
      R1 = F(6) - F(8) - F(12) + F(10)
      Q1 = F(4) - F(10) - F(12)
      P1 = F(12)
C
      OUT(1) = -Q1 + T0
      OUT(2) = -P1 + S0
      OUT(3) = R0
      OUT(4) = Q0 - Q1 + T1
      OUT(5) = P0 - P1 + S1
      OUT(6) = R1
      RETURN
      END
C+++++++++++++++++++++++++++++++++++++++++
C
C     POLYNOMIAL TRANSFORM LENGTH 9 ROOT Z
C     MOD C9(Z)
C
      SUBROUTINE POLYT1(IN, OUT, FWD)
      INTEGER IN(9,6), OUT(9,6), TEMP(9,6),
     & POLY, POLY1, POWER, INDEX
      LOGICAL FWD
C
      DO 20 I = 1, 9
        DO 10 J =1, 6
          OUT(I,J) = 0
10      CONTINUE
20    CONTINUE
C
      DO 80 POLY1 = 1, 9
        DO 70 POLY = 1, 9
          DO 60 POWER = 1, 6
            INDEX=(POLY1-1)*(POLY-1)+POWER-1
30          IF (INDEX .LT. 9) GO TO 40
            INDEX = INDEX - 9
            GO TO 30
40          IF (INDEX .GT. 5) GO TO 50
            OUT(POLY1,INDEX+1)=
     &      OUT(POLY1,INDEX+1)+IN(POLY,POWER)
            GOTO 60
50          INDEX = INDEX - 3
            OUT(POLY1,INDEX+1)=
     &      OUT(POLY1,INDEX+1)-IN(POLY,POWER)
            INDEX = INDEX - 3
            OUT(POLY1,INDEX+1)=
     &      OUT(POLY1,INDEX+1)-IN(POLY,POWER)
60        CONTINUE
70      CONTINUE
80    CONTINUE
C
C     SELECT FORWARD OR INVERSE TRANSFORM
C
      IF (FWD) RETURN
```

```
C
C     INVERSE TRANSFORM SELECTED
C
      DO 100 POLY = 2, 9
        DO 90 POWER = 1, 6
          TEMP(11-POLY,POWER)=OUT(POLY,POWER)
90      CONTINUE
100   CONTINUE
      DO 120 POLY = 2, 9
        DO 110 POWER = 1, 6
          OUT(POLY,POWER)=TEMP(POLY,POWER)
110     CONTINUE
120   CONTINUE
      RETURN
      END
C+++++++++++++++++++++++++++++++++++++++++
C
C     PRE-MULT 3X3 CYCLIC CONVOLUTION ALG.
C
      SUBROUTINE PRE3X3(X, H, M, COEF)
      INTEGER X(3,3), H(3,3), M(13),
     & COEF(13), AM(13,9), BM(13,9),ROW, COL,
     & ROW1, COL1
C
C     GIVEN IN MATRIX FORM
C
      DATA ((AM(ROW,COL),COL=1,9),ROW=1,13) /
     X-1, 1, 0,-1, 1, 0,-1, 1, 0,
     X 0,-1, 1, 0,-1, 1, 0,-1, 1,
     X-1, 0, 1,-1, 0, 1,-1, 0, 1,
     X-1, 1, 0, 0,-1, 1, 1, 0,-1,
     X 0,-1, 1, 1, 0,-1,-1, 1, 0,
     X-1, 0, 1, 1,-1, 0, 0, 1,-1,
     X-1, 1, 0, 1, 0,-1, 0,-1, 1,
     X 0,-1, 1,-1, 1, 0, 1, 0,-1,
     X-1, 0, 1, 0, 1,-1, 1,-1, 0,
     X-1,-1,-1, 1, 1, 1, 0, 0, 0,
     X 0, 0, 0,-1,-1,-1, 1, 1, 1,
     X-1,-1,-1, 0, 0, 0, 1, 1, 1,
     X 1, 1, 1, 1, 1, 1, 1, 1, 1/
C
      DATA (BM(ROW,COL),COL=1,9),ROW=1,13) /
     X 0,-1, 1, 0,-1, 1, 0,-1, 1,
     X-1, 0, 1,-1, 0, 1,-1, 0, 1,
     X-1, 1, 0,-1, 1, 0,-1, 1, 0,
     X 0,-1, 1, 1, 0,-1,-1, 1, 0,
     X-1, 0, 1, 1,-1, 0, 0, 1,-1,
     X-1, 1, 0, 0,-1, 1, 1, 0,-1,
     X 0,-1, 1,-1, 1, 0, 1, 0,-1,
     X-1, 0, 1, 0, 1,-1, 1,-1, 0,
     X-1, 1, 0, 1, 0,-1, 0,-1, 1,
     X 0, 0, 0,-1,-1,-1, 1, 1, 1,
     X-1,-1,-1, 0, 0, 0, 1, 1, 1,
     X-1,-1,-1, 1, 1, 1, 0, 0, 0,
     X 1, 1, 1, 1, 1, 1, 1, 1, 1/
C
```

```
      DO 30 ROW = 1, 13
        M(ROW) = 0
        COEF(ROW) = 0
        DO 20 ROW1 = 1, 3
          DO 10 COL1 = 1, 3
            M(ROW) = M(ROW) + X(ROW1,4-COL1)*
     &                   AM(ROW,3*(ROW1-1)+COL1)
            COEF(ROW) = COEF(ROW) +
     &BM(ROW,3(ROW1-1)+COL1) * H(ROW1,4-COL1)
 10       CONTINUE
 20     CONTINUE
 30   CONTINUE
      RETURN
      END
C++++++++++++++++++++++++++++++++++++++++
C
C     POST MULT. 3X3 CYCLIC CONVOLUTION
C
      SUBROUTINE PST3X3(M, Y)
      IMPLICIT INTEGER (G,L)
      INTEGER M(13), Y(3,3), POLY, POWER
C
      G00 = M(1)+M(2)+M(4)+M(5)+M(7)+M(8)
      G01 = M(1)+M(3)+M(4)+M(6)+M(7)+M(9)
C
      G10 = M(1)+M(2)+M(6)-M(5)-M(7)-M(9)
      G11 = M(1)+M(3)-M(4)-M(5)-M(9)+M(8)
C
      G21 = M(1)+M(3)-M(6)+M(5)-M(7)-M(8)
      G20 = M(1)+M(2)-M(4)-M(6)+M(9)-M(8)
C
      L0 = 2*M(10) + M(11) + M(12) - M(13)
      L1 = -M(10) + M(11) - 2*M(12) - M(13)
      L2 = -M(10) - 2*M(11) + M(12) - M(13)
C
      Y(1,3) = -(G00 + G01 +L1)
      Y(1,2) = -(-2*G01 + G00 + L1)
      Y(1,1) = -(-2*G00 + G01 + L1)
C
      Y(2,3) = -(G10 + G11 + L2)
      Y(2,2) = -(-2*G11 + G10 + L2)
      Y(2,1) = -(-2*G10 + G11 + L2)
C
      Y(3,3) = -(G20 + G21 + L0)
      Y(3,2) = -(-2*G21 + G20 + L0)
      Y(3,1) = -(-2*G20 + G21 + L0)
C
      RETURN
      END
C++++++++++++++++++++++++++++++++++++++++
C
C     POLY TRANSFORM LENGTH 3 ROOT Z**3
C             MOD C9(Z)
C
      SUBROUTINE POLYT2 (I, OUT, FWD)
      INTEGER I(3,6), OUT(3,6), TEMP(3,6),
     & POLY, POWER, POLY1, POWER
      LOGICAL FWD
```

```
C
      DO 20 POLY = 1, 3
        DO 10 POWER = 1, 6
          OUT(POLY, POWER) = 0
 10     CONTINUE
 20   CONTINUE
C
      DO 40 POWER = 1, 6
        DO 30 POLY = 1, 3
          OUT(1,POWER)=
     &    OUT(1,POWER)+I(POLY,POWER)
 30     CONTINUE
 40   CONTINUE
C
      OUT(2,1)=I(1,1)-I(2,4)+I(3,4)-I(3,1)
      OUT(2,2)=I(1,2)-I(2,5)+I(3,5)-I(3,2)
      OUT(2,3)=I(1,3)-I(2,6)+I(3,6)-I(3,3)
      OUT(2,4)=I(1,4)-I(2,4)+I(2,1)-I(3,1)
      OUT(2,5)=I(1,5)-I(2,5)+I(2,2)-I(3,2)
      OUT(2,6)=I(1,6)-I(2,6)+I(2,3)-I(3,3)
C
      OUT(3,1)=I(1,1)+I(2,4)-I(2,1)-I(3,4)
      OUT(3,2)=I(1,2)+I(2,5)-I(2,2)-I(3,5)
      OUT(3,3)=I(1,3)+I(2,6)-I(2,3)-I(3,6)
      OUT(3,4)=I(1,4)-I(2,1)-I(3,4)+I(3,1)
      OUT(3,5)=I(1,5)-I(2,2)-I(3,5)+I(3,2)
      OUT(3,6)=I(1,6)-I(2,3)-I(3,6)+I(3,3)
C
      IF (FWD) RETURN
C
      DO 50 POWER = 1, 6
        TEMP(2,POWER) = OUT(2,POWER)
        TEMP(3,POWER) = OUT(3,POWER)
 50   CONTINUE
      DO 60 POWER = 1, 6
        OUT(3,POWER) = TEMP(2,POWER)
        OUT(2,POWER) = TEMP(3,POWER)
 60   CONTINUE
      RETURN
      END
C**************************************************
C
C       CHINESE REMAINDER THEOREM
C     RECOMBINATIONS THIS ROUTINE USES THE
C             NEXT TWO
C
      SUBROUTINE CRT1(IN1, IN2, OUT)
      INTEGER IN1(3,6), IN2(3,3), OUT(3,9),
     & TEMP6(9), TEMP9(9), TEMP3(3),
     & POLY, POWER
C
      DO 20 POLY = 1, 3
        DO 10 POWER = 1, 9
          OUT(POLY,POWER) = 0
 10     CONTINUE
 20   CONTINUE
C
```

```
      DO 70 POLY = 1, 3
        DO 30 POWER = 1, 6
          TEMP6(POWER) = IN1(POLY,POWER)
30      CONTINUE
C
        CALL R1(TEMP6, TEMP9)
C
        DO 40 POWER = 1, 9
          OUT(POLY,POWER) = OUT(POLY,POWER) -
      &                   3*TEMP9(POWER)
40      CONTINUE
C
        DO 50 POWER = 1, 3
          TEMP3(POWER) = IN2(POLY,POWER)
50      CONTINUE
C
      CALL R2(TEMP3, TEMP9)
C
        DO 60 POWER = 1, 9
          OUT(POLY,POWER) = OUT(POLY,POWER) +
     &                    TEMP9(POWER)
60      CONTINUE
70    CONTINUE
      RETURN
      END
C++++++++++++++++++++++++++++++++++++++++++
C
C       POLYNOMIAL MULTIPLICATION BY
C     RECOMBINATION POLYNOMIAL FOR C9(Z)
C
      SUBROUTINE R1(A, OUT)
      INTEGER A(6), OUT(9)
C
      OUT(1) = A(4) - 2*A(1)
      OUT(2) = A(5) - 2*A(2)
      OUT(3) = A(6) - 2*A(3)
      OUT(4) = A(1) - 2*A(4)
      OUT(5) = A(2) - 2*A(5)
      OUT(6) = A(3) - 2*A(6)
      OUT(7) = A(1) + A(4)
      OUT(8) = A(2) + A(5)
      OUT(9) = A(3) + A(6)
      RETURN
      END
C++++++++++++++++++++++++++++++++++++++++++
C
C     POLY MULT FOR Z**3-1 RECOMBINATION
C                POLYNOMIAL
C
      SUBROUTINE R2(B, OUT)
      INTEGER B(3), OUT(9), POLY
C
      DO 10 POLY = 1, 3
        OUT(POLY) = B(POLY)
        OUT(POLY+3) = B(POLY)
        OUT(POLY+6) = B(POLY)
10    CONTINUE
      RETURN
      END
C++++++++++++++++++++++++++++++++++++++++++
```

Appendix III

This appendix contains a 15-point Discrete Cosine Transform (DCT) algorithm. The algorithm is for data in natural order, there are 19 coefficients all of which are positive. The coefficients are arranged so as to give the minimum total number of columns in the one-bit systolic arrays discussed in chapter 8.

The A matrix is

$$\left(\begin{array}{rrrrrrrrrrrrrrr}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 1 & 0 \\
-1 & -1 & 0 & 1 & 1 & 1 & 1 & 0 & -1 & -1 & -1 & -1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
-1 & 1 & 0 & 1 & -1 & 1 & 1 & 0 & 1 & -1 & -1 & 1 & 0 & 1 & -1 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
-1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & 1 \\
-1 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & -1 & 0 & 0 & -1 \\
1 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & -1 \\
1 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 1 & 0 & 0 & -1 \\
-1 & 0 & 0 & -1 & 0 & -1 & 1 & 0 & -1 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
-1 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & -1 \\
-1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 1 \\
-1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0
\end{array}\right)$$

The C matrix is:

$$\left(\begin{array}{rrrrrrrrrrrrrrrrrrr}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & -1 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & -1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & -1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & -1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 & 1 & 1 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & 1
\end{array}\right)$$

The coefficients for the above algorithm are found as follows (all values are positive and are to be multiplied by $\sqrt{(2/15)}$.

$D0 = \sin u_3$ $\qquad$ $D1 = \sin 2u_5$

$D2 = \sin u_5 + \sin 2u_5$ $\qquad$ $D3 = \sin u_5 - \sin 2u_5$

$D4 = \cos u_3 - 1$ $\qquad$ $D5 = \frac{1}{2}(\cos u_5 + \cos 2u_5) -1$

$D6 = \frac{1}{2}(\cos u_5 - \cos 2u_5)$

where $u_3 = 2\pi/3$ and $u_5 = 2\pi/5$. The coefficients are

$m_0 = 1/\sqrt{2}$

$m_1 = D0.D3$

$m_2 = -D4.D2$

$m_3 = D2$

$m_4 = D1$

$m_5 = -D5$

$m_6 = D6$

$m_7 = -D4$

$m_8 = -D4.D1$

$m_9 = D0.D1$

$m_{10} = D0$

$m_{11} = -D0.D5$

$m_{12} = D0.D6$

$m_{13} = 1$

$m_{14} = D4.D5$

$m_{15} = -D4.D6$

$m_{16} = D3$

$m_{17} = -D4.D3$

$m_{18} = D0.D2$

IMPLEMENTATION OF CONVOLUTION AND FOURIER TRANSFORMS USING 1-BIT SYSTOLIC ARRAYS

Indexing terms: Signal processing, Systolic arrays

The use of systolic arrays of 1-bit cells to implement circular convolution and DFTs is described. The architecture is very well suited to VLSI implementation. It is shown that considerable simplification of the architecture is possible for real-valued DFTs.

Introduction: Recently McCanny and McWhirter[1,2] have demonstrated how arrays of identical 1-bit processing elements can be used to perform pipelined multiplication and a bit-sliced transform. Both of these are regular structures particularly suited to implementation in VLSI. This letter shows how these two structures may be combined to perform circular convolution and Fourier transforms by using algorithms presented by Winograd.[4]

Convolutions and the Winograd transform algorithm (WFTA): Agarwal and Cooley[3] and Winograd[4] showed how fast convolution algorithms could be derived using a small, often the minimum, number of multiplications. Winograd applied a theorem due to Rader[5] to use these convolutions to perform DFTs. For the convolutions and the WFTA the general form of the algorithm is

$$\mathbb{Y} = \mathbb{C}(\mathbb{A}y \times \mathbb{B}z) \qquad (1)$$

where $\mathbb{Y}$ is the transform of the input date y. $\mathbb{A}$ and $\mathbb{B}$ are $N \times M$ rectangular matrices, N being the transform length and M the number of multiplications. $\mathbb{C}$ is an $M \times N$ matrix. $\times$ denotes pointwise multiplication and the product $\mathbb{B}z$ is precalculated as a series of M coefficients. For the WFTA these coefficients are either purely real or purely imaginary.

In all the convolution and DFT algorithms given by Winograd,[4] except the 9-point DFT, the $\mathbb{A}$ and $\mathbb{C}$ matrices contain only the elements -1, 0 and $+1$. A 9-point DFT of this form can be constructed at the expense of an extra multiplication. It is also possible, using polynomial transforms due to Nussbaumer,[6,7] to construct two-dimensional convolution algorithms of the same form as eqn. 1 in which the $\mathbb{A}$ and $\mathbb{C}$ matrices contain only -1, 0 and $+1$. Thus the method of computation is the same in each of these cases—a multiplication by a matrix containing 0, -1 and $+1$ and a multiplication, followed by a further multiplication by a matrix containing 0, -1 and $+1$.

Proposed architecture: McCanny and McWhirter[1] described a pipelined bit-slice transform array which performs a matrix times vectors transform in the form $\mathbb{W}x = y$, where W is an $n \times n$ matrix of 1-bit coefficients. Utilising two control bits and two's complement arithmetic, their basic cell can be extended to operate with coefficients of -1, 0 and $+1$. Whenever W_{ij} is -1 the input word is complemented bit by bit and unity is added to the LSB via the carry-in and the top row of cells.

An $N \times M$, or $M \times N$, matrix requires $N + M - 1$ columns and $b + \log_2(M)$, or $b + \log_2(N)$, rows of processor cells, where b is the number of bits of input data. As the array is no longer square, a gap of $M - N$, $(M > N)$, cycles is needed between successive sets of data. This extended array can now perform the $\mathbb{A}$ and $\mathbb{C}$ matrix multiplications.

The input and output words for this transform array are staggered bit by bit. The stagger on the output from the bit-slice transform array is the same as that needed for one of the input words for the two's complement pipelined multipler described in Reference 2. Furthermore, the stagger on the output from the multiplier is the same as the input into the bit-slice transform array. Thus, if a system of transform array, multiplier and transform array is used in calculations of the form of eqn. 1, the arrays of cells can be laid end to end with no need for immediate data skew. The throughput of such a cell system is limited by the propagation delay through one cell. The transform array and multiplier cells are very similar.

In the transform array only every alternate processor is active and data enters and leaves the array on alternate cycles. However, with the same clocking rate, the multiplier architecture can handle data on every cycle. So the multiplier can process two sets of data to the transform arrays' one. If the 'spare' processors in the transform array were utilised, these two sets could be the real and imaginary parts of a complex DFT.

Simplifications to the $\mathbb{C}$ matrix for real DFTs: Parsons[8] showed that each of the $\mathbb{C}$ matrices in the small-n WFTAs can be simplified by noting that for a DFT of real data $\mathbb{Y}_{N-k} = \mathbb{Y}_k^*$, where * denotes complex conjugation. Thus there is some redundancy in the $\mathbb{C}$ matrices. This is most clearly seen if the real and imaginary coefficients are separated into two distinct groups: e.g. for $N = 5$,

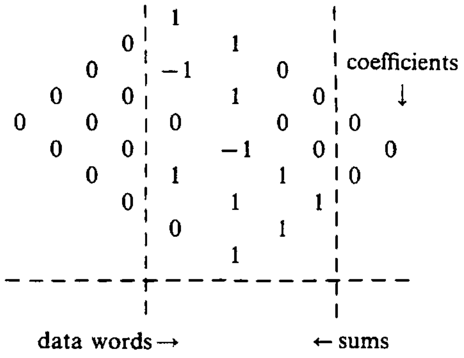
$$\mathbb{C} = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 1 & 1 & -1 & 0 \\ 1 & 1 & -1 & 0 & 1 & 1 \\ \hline 1 & 1 & -1 & 0 & -1 & -1 \\ 1 & 1 & 1 & -1 & 1 & 0 \end{array}\right] = \begin{bmatrix} \phi & 0 \\ \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

For a real input the vertical partition separates those elements contributing to the real and imaginary parts of the transform. The horizontal partitions mark off the $n = 0$ term and the first and second halves of the transform. (For even N the $n = N/2$ term is also partitioned.)

γ and δ generate the complex conjugates of the terms resulting from α and β. Therefore we define a new matrix $\mathbb{C}'$ in which γ and δ are discarded and the remaining coefficients rearranged:

$$C' = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \phi & 0 \\ \alpha & 0 \\ 0 & \beta \end{bmatrix}$$

This places the real part of the first half of the transform into the first half of the output vector, and the imaginary part of the first half into the remainder of the output vector. The coefficients are entered into the transform arrays in the shape of a parallelogram. For $N = 5$, with data words moving left to right and sums right to left, the arrangement is



Note that, because of the redundancy and the coefficient rearrangement, the three left-hand and the two right-hand columns contain only zeros. So the corresponding columns of processors never contribute to the matrix product and may be omitted. This halves the array size needed. Additionally, in this example, a further column can be deleted. Also, at least one column can be saved in each of the DFT $\mathbb{A}$ matrices.

Conclusion: We have proposed a general architecture which can perform convolutions and DFTs. This architecture is very easy to implement in VLSI with only three cell types needed: a transform array, a multiplier and a shift register for the initial and final data skewing. Furthermore, the architecture has all the inherent features of systolic arrays. There is a high throughput, and transform size is limited only by the level of integration possible. Partitioning the $\mathbb{C}$ matrix avoids the need for complex arithmetic, and the $\mathbb{C}$ matrix transform array can be significantly reduced in size for real valued DFTs.

Reprinted from ELECTRONICS LETTERS 2nd September 1982 Vol.18 No.18 pp.799–801

J. S. WARD
B. J. STANIER

Department of Applied Physics & Electronics
University of Durham
South Road, Durham DH1 3LE, England

13th July 1982

References

1 MCCANNY, J. V., and MCWHIRTER, J. G.: 'Implementation of signal processing functions using 1-bit systolic arrays', *Electron. Lett.*, 1982, **18**, pp. 241–243

2 MCCANNY, J. V., and MCWHIRTER, J. G.: 'Completely iterative pipelined multiplier array suitable for VLSI', *IEEE Proc. G, Electron. Circ. & Syst.*, 1982, **129**, pp. 40–46

3 AGARWAL, R. C., and COOLEY, J. W.: 'New algorithms for digital convolution', *IEEE Trans.*, 1977, **ASSP-25**, pp. 106–124

4 WINOGRAD, S.: 'On computing the discrete Fourier transform', *Math. Comput.*, 1978, **32**, pp. 175–199

5 RADER, C. M.: 'Discrete Fourier transforms when the number of data samples is prime', *IEEE Proc.*, 1968, **56**, pp. 1107–1108

6 NUSSBAUMER, H. J.: 'Digital filtering using polynomial transforms', *Electron. Lett.*, 1977, **13**, pp. 386–387

7 NUSSBAUMER, H. J., and QUANDALLE, P.: 'Computation of convolutions and discrete Fourier transforms by polynomial transforms', *IBM J. Res. Dev.*, 1978, **22**, pp. 134–144

8 PARSONS, T. W.: 'A Winograd-Fourier transform algorithm for real-valued data', *IEEE Trans.*, 1979, **ASSP-22**, pp. 398–402

0013-5194/82/180799-03$1.50/0

FAST DISCRETE COSINE TRANSFORM ALGORITHM FOR SYSTOLIC ARRAYS

Indexing terms: Signal processing, Systolic arrays

A fast algorithm for an N-point discrete cosine transform (DCT) is derived from a $4N$-point Winograd Fourier transform algorithm (WFTA). This algorithm, which has the same form as Winograd's Fourier transform and convolution algorithms, is suitable for a high-speed implementation using one-bit systolic arrays.

Introduction: Several authors[1,2] have shown that the discrete cosine transform is a good technique to adopt for the data reduction of video signals. Earlier methods of realising the DCT have, for the most part, been based on the fast Fourier transform (FFT).[3-5] An alternative technique, using Hadamard sparse matrices, has been proposed and implemented.[6,7] Here, a method of deriving efficient DCT algorithms is proposed based on length $4N$ Winograd Fourier transform algorithms (WFTA). This technique yields algorithms using less multiplications than the method, also based on the WFTA, outlined in Reference 5.

Furthermore, an efficient implementation is proposed based on the systolic arrays given by McCanny and McWhirter.[8]

Theory: The discrete cosine transform may be defined as

$$C(0) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n)$$

$$C(k) = \sqrt{\left(\frac{2}{N}\right)} \sum_{n=0}^{N-1} x(n) \cos\left(\frac{k\pi(2n+1)}{2N}\right),$$

$$k = 1, \ldots, N-1 \qquad (1)$$

with inverse

$$x(n) = \frac{1}{\sqrt{N}} C(0) + \sqrt{\left(\frac{2}{N}\right)} \sum_{k=1}^{N-1} c(k) \cos\left(\frac{k\pi(2n+1)}{2N}\right),$$

$$n = 0, 1, \ldots, N-1 \qquad (2)$$

Notice that, unlike the discrete Fourier transform, the DCT cannot be used as its own inverse. The DFT of R points is defined as

$$F(q) = \frac{1}{\sqrt{R}} \sum_{p=0}^{R-1} x'(p)\left[\cos\left(\frac{2\pi pq}{R}\right) + j \sin\left(\frac{2\pi pq}{R}\right)\right],$$

$$q = 0, 1, \ldots, R-1 \qquad (3)$$

with inverse

$$x'(p) = \frac{1}{\sqrt{R}} \sum_{q=0}^{R-1} F(q)\left[\cos\left(\frac{2\pi pq}{R}\right) - j \sin\left(\frac{2\pi pq}{R}\right)\right],$$

$$p = 0, 1, \ldots, R-1 \qquad (4)$$

So a DCT may be calculated by a DFT by noticing that

$$\sum_{n=0}^{N-1} x(n) \cos\left(\frac{k\pi(2n+1)}{2N}\right) = \mathrm{Re} \sum_{p=0}^{R-1} x'(p)$$

$$\times \left[\cos\left(\frac{2\pi pq}{R}\right) + j \sin\left(\frac{2\pi pq}{R}\right)\right]$$

if

$$R = 4N$$

and

$$x'(2n+1) = x(n), \quad n = 0, 1, \ldots, N-1$$

otherwise

$$x'(l) = 0, \; l \neq 2n+1, \quad l = 0, 1, \ldots, 4N-1$$

Thus the calculation of the DCT may be described as placing the N terms of the DCT input sequence in the first N odd points $(1, 3, 5, \ldots, 2N-1)$ of a $4N$-length sequence. All the other terms are zero. Then a Fourier-like transform is performed by multiplying by the real part of $\exp(-2\pi pq/R)$. The DCT, except $C(0)$, is found as the first $N-1$ terms of $F(q)$. From the definitions given above, some normalisation coefficients are needed.

Application of Winograd Fourier transform algorithm: Winograd's Fourier transform algorithms[4] each have the general form

$$\mathbb{Y} = \mathbb{C}(\mathbb{A}y \otimes \mathbb{B}z) \qquad (5)$$

where $\otimes$ denotes pointwise multiplication, y is the data and $\mathbb{Y}$ its transform. $\mathbb{A}$ is an $M \times N$ (row × column) matrix and $\mathbb{C}$ an $N \times M$ matrix; both contain only $+1$, -1 and 0. The product $\mathbb{B}z$ is precalculated and given as a sequence of M coefficients. These coefficients are either real or imaginary, never complex. A WFTA of $4N$ points is easily modified to calculate an N-point DCT. The procedure is described below.

(i) Only the columns corresponding to the first N odd samples of the $\mathbb{A}$ matrix are retained. All other columns are discarded as they operate on zero inputs.

(ii) Since the DCT involves no complex arithmetic—it has no imaginary values—all the imaginary WFTA coefficients may be removed. The corresponding rows of the $\mathbb{A}$ matrix and the corresponding columns of the $\mathbb{C}$ matrix are now deleted.

(iii) Some rows of the $\mathbb{A}$ matrix may now be all zero. The coefficients for each of these rows may be removed along with the appropriate column of the $\mathbb{C}$ matrix.

(iv) The second to Nth rows of the $\mathbb{C}$ matrix are retained. These represent the DCT—all the others should be discarded.

(v) By inspection it may be possible to simplify the $\mathbb{C}$ matrix. For example, a column of zeros represents an unused coefficient. Such coefficients and the equivalent rows of the $\mathbb{A}$ matrix should be removed from the algorithm.

(vi) Finally the $\mathbb{C}(0)$ term is added to the algorithm. The exact arrangement of the normalisation coefficients in eqns. 1 and 2 determines the number of additional multiplications to be added to the algorithm. With the definitions given above one extra multiplication is needed.

The above procedure results in an efficient algorithm for the calculation of the DCT using a small number of multiplications.

The same idea can be applied to derive an inverse DCT from the forward WFTA. In this case the first $N-1$ columns of the $\mathbb{A}$ matrix and the first N odd columns of the $\mathbb{C}$ matrix are used, i.e. the opposite way round to the forward DCT.

Implementing the DCT: We have previously shown how DFT and convolution algorithms of the general form of eqn. 5 can be implemented using 1-bit systolic arrays.[11] DCT algorithms derived by the above method still have the general form of eqn. 5. A 15-point DCT algorithm is given below. This algorithm has been arranged so that all the multiplication coefficients are positive, and the coefficients are ordered so as to reduce the number of columns of 'transform array' cells required for this algorithm.

Reprinted from ELECTRONICS LETTERS 20th January 1983 Vol. 19. No. 2 pp. 58 - 60

15-point DCT matrices: The *A* matrix, for data in natural order, is

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & -1 & 0 & 1 & 0 \\
-1 & -1 & 0 & 1 & 1 & 1 & 1 & 0 & -1 & -1 & -1 & -1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
-1 & 1 & 0 & 1 & -1 & 1 & 1 & 0 & 1 & -1 & -1 & 1 & 0 & 1 & -1 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
-1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & 1 \\
-1 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & -1 & 0 & 0 & -1 \\
1 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & 0 & -1 \\
1 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & -1 & -1 & 0 & 1 & 0 & 0 & -1 \\
-1 & 0 & 0 & -1 & 0 & -1 & 1 & 0 & -1 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
-1 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & -1 \\
-1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 1 \\
-1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0
\end{pmatrix}$$

The *C* matrix, with the output in natural order, is

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & -1 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & -1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & -1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & -1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 & 1 & 1 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & 1
\end{pmatrix}$$

The coefficients for the above algorithm are found as follows (all values are positive and are to be multiplied by $\sqrt{(2/15)}$):

$$M0 = \sin U_3 \qquad M1 = \sin 2U_5$$

$$M2 = \sin U_5 + \sin 2U_5 \qquad M3 = \sin U_5 - \sin 2U_5$$

$$M4 = \cos U_3 - 1 \qquad M5 = 1/2(\cos U_5 + \cos 2U_5) - 1$$

$$M6 = 1/2(\cos U_5 - \cos 2U_5)$$

where $U_3 = 2\pi/3$ and $U_5 = 2\pi/5$. The coefficients are:

1	$1/\sqrt{2}$	11	$M0$
2	$M0 \,.\, M3$	12	$-M0 \,.\, M5$
3	$-M4 \,.\, M2$	13	$M0 \,.\, M6$
4	$M2$	14	1·0
5	$M1$	15	$M4 \,.\, M5$
6	$-M5$	16	$-M4 \,.\, M6$
7	$M6$	17	$M3$
8	$-M4$	18	$-M4 \,.\, M3$
9	$-M4 \,.\, M1$	19	$M0 \,.\, M2$
10	$M0 \,.\, M1$		

An implementation of a DCT algorithm using these systolic arrays would have a throughput high enough to cope with real-time television pictures.

Conclusion: An efficient algorithm for performing the discrete cosine transform has been derived. The algorithm is well suited to implementation using 1-bit systolic arrays. If advantage is taken of alternate cells being unused in the design for the transform arrays in Reference 8, considerable compaction in the size of the arrays is possible. Such compactions would allow small length DCTs to be implemented in systems requiring only a few VLSI chips.

J. S. WARD
B. J. STANIER

Department of Applied Physics & Electronics
University of Durham
South Road, Durham City DH1 3LE, England

19th November 1982

References

1 CHEN, W., and SMITH, C. H.: 'Adaptive coding of monochrome and colour images', *IEEE Trans.*, 1977, COM-25, pp. 1285–1292

2 ROSE, J. A., PRATT, W. K., and ROBINSON, A. S.: 'Interframe cosine transform image coding', *ibid.*, 1977, COM-25, pp. 1329–1339

3 AHMED, N., NATARAJAN, T., and RAO, K. R.: 'On image coding and a discrete cosine transform', *ibid.*, 1974, C-23, pp. 90–93

4 HARALICK, R. M.: 'A storage efficient way to implement the discrete cosine transform', *ibid.*, 1976, C-25, pp. 764–765

5 NARASHIM, M. J., and PETERSON, A. M.: 'On the computation of the discrete cosine transform', *ibid.*, 1978, COM-26, pp. 934–936

6 HEIN, D., and AHMED, N.: 'On a real-time Walsh–Hadamard/cosine transform image processor', *ibid.*, 1978, EMC-20, pp. 453–457

7 GHANBARI, M., and PEARSON, D. E.: 'Fast cosine transform implementation for television signals', *IEE Proc. F, Commun., Radar & Signal Process.*, 1982, 129, pp. 59–68

8 MCCANNY, J. V., and MCWHIRTER, J. G.: 'Implementation of signal processing functions using 1-bit systolic arrays', *Electron. Lett.*, 1982, 18, pp. 241–243

9 MCCANNY, J. V., and MCWHIRTER, J. G.: 'Completely iterative pipelined multiplier array suitable for VLSI', *IEE Proc. G, Electron. Circ. & Syst.*, 1982, 129, (2), pp. 40–46

10 WINOGRAD, S.: 'On computing the discrete Fourier transform', *Math. Comput.*, 1978, 32, pp. 175–199

11 WARD, J. S., and STANIER, B. J.: 'Implementation of convolution and Fourier transforms using 1-bit systolic arrays', *Electron. Lett.*, 1982, 18, pp. 799–801

0013-5194/83/020058-03$1.50/0

A Matrix 1-bit Systolic Array

Key:

S	Substrate Bias
ϕ	Clock Inputs
P	Coefficient Propagate Signal
I	Input Data (1 = ℓsb)
O	Output Data

Designed by J.S.Ward