# VLSI Implementation of Digital Fourier Transforms
## Final Report

*A. Despain*
*C. Sequin*
*C. Thompson*
*E. Wold*
*D. Lioupis*

November 1, 1982

U. S. Army Research Office

Grant number: DAAG29-78-G-0167

Institution: University of California

The view, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

# Table of Contents

# List of Figures

# List of Tables

# VLSI Implementation of Digital Fourier Transforms

## 1. OVERVIEW

In the late 1970's a modular, high-throughput architecture for large scale Fourier Transform processors was developed in [1,2]. This architecture uses only a few basic modules in a highly pipelined arrangement and some serial memory for temporary storage of operands. This streamlined architecture seemed predestined for implementation with "Charge Transfer Devices", which have proven themselves in many high-speed signal processing applications and for serial memory [3]. Thus we proposed to investigate the use of charge-coupled devices (CCDs) in the implementation of pipelined FFT processors. For various reasons which are explained below, the use of CCD's was dropped at an early stage and the decision was made to design these same modules with standard silicon-gate NMOS technology.

A collection of the basic modules used in these FFT architectures have been designed and implemented; some are currently in fabrication. These modules are:

1. a specialized $\frac{\pi}{16}$ vector rotator which uses the rational approximation algorithm developed by Despain in [2],
2. a module to perform a multiplication by $\sqrt{3}$, also developed in [2],
3. a general CORDIC rotator, capable of rotating a complex vector by any angle with 16-bit precision,
4,5. two modules capable of performing the "butterfly" operation of the Fast Fourier Transform (FFT),
6. a barrel shifter which was designed for use in an iterative CORDIC module.

We also expanded the scope of our research to study the more general problem of efficiently computing the Discrete Fourier Transform with proper attention to the constraints of VLSI. Many of the results are applicable to most VLSI technologies (N- or P-channel MOS, bulk CMOS, SOS CMOS, $I^2L$). New techniques were developed for the construction of large scale FFT processors which are geared toward the use of VLSI. These techniques employ the traditional Cooley-Tukey version of the FFT [4] as well as the prime-factor algorithm of Good [5] and elements of the Winograd Fourier Transform [6].

At a lower level, we developed new results on minimum latency adders which would be useful in the design of Fourier Transform processors using the CORDIC or rational approximation rotation algorithms. On the theoretical side, studies of the computational complexity of various Fourier Transform algorithms were made using a VLSI model developed by Thompson [7].

In this report we will first review the class of highly pipelined architectures for FFT processors which are considered for implementation with VLSI (sect 2 & 3). We then review the charge-transfer techniques and the more classical MOS technologies and discuss the trade-offs for the implementation of the envisioned FFT processor architectures (sect. 4). In section 5 we present a detailed description of the hardware modules that were designed and implemented in

NMOS technology and make some performance predictions for a complete system. In section 6 we present new theoretical results concerning minimum latency adders which could be used in FFT processors where minimum latency was a design goal. Also there are results from the application of complexity theory which produce some absolute bounds of area and time for implementations of FFT processors in VLSI.

## 2. INTRODUCTION

### 2.1. Fast Fourier Transform background

The Discrete Fourier Transform (DFT), widely used in many areas of signal processing, can be expressed as

$$A_r = \sum_{k=0}^{N-1} B_k e^{-j\frac{2\pi rk}{N}} \qquad r = 0,1,\ldots,N-1 \tag{1}$$

This computation is generally used to transform the representation of a set of data samples from the time or space domain into the frequency domain. The Cooley-Tukey FFT [4] is a factorization of this equation which reduces the number of multiplications involved from $O(N^2)$ to $O(N\log_2 N)$ assuming that the radix-2 algorithm is used. This method of computing (1) consists of "butterfly" operations of the form

$$\begin{aligned} C &= A+B \\ D &= A-B \end{aligned} \tag{2}$$

and multiplications by various roots of unity. The Cooley-Tukey algorithm has a great deal of regularity which can be exploited in a VLSI implementation.

Other techniques developed by Good [5] and Winograd [6] can be utilized to reduce the number of multiplications required to compute (1) to $O(N)$ for certain values of $N$, although these techniques in general require an increase in the complexity of the interconnections involved. The reduction in multiplications is achieved by expressing small transforms as convolutions, utilizing fast algorithms to perform these convolutions, and by building up large transforms out of these smaller, relatively prime modules.

While most of the Discrete Fourier Transform algorithms which have been developed have been for existing general-purpose machines, tremendous speed-ups are possible through the development of algorithms and hardware simultaneously. Despain [1] pointed out that the complex multiplications in (1) are actually vector rotations and can thus be computed using an arithmetic technique known as the CORDIC algorithm, originally developed by Volder [8]. Depending on the factors of $N$, the transform size, computational savings can also be realized through the use of rational approximations for rotations as described in [2].

### 2.2. DFT Architectures

Only a few basic functions are needed to implement a wide set of Cooley-Tukey type FFT algorithms of a given transform length and transform radix. These are:

1. A butterfly module which performs the operation in (2) above.
2. A CORDIC rotator module which performs the calculation of $Be^{j\phi}$ in (1).
3. Shift register memories for intermediate storage of data.

The pipeline structure shown in figure 1 is derived in [1], and consists only of the three modules listed above. Basically, the operation of the processor is as follows. The first butterfly (BF) module allows the input $a_i$ to pass unchanged into the shift register of length $2^{n-1}$ until it is full. At that point, the incoming data and the data in the shift register are combined in a 2-point DFT:

$$\begin{aligned} b_i &= a_i + a_{i+\frac{N}{2}} \\ b_{i+\frac{N}{2}} &= a_i - a_{i+\frac{N}{2}} \end{aligned} \qquad i = 0,\ldots,\frac{N}{2}-1 \tag{3}$$

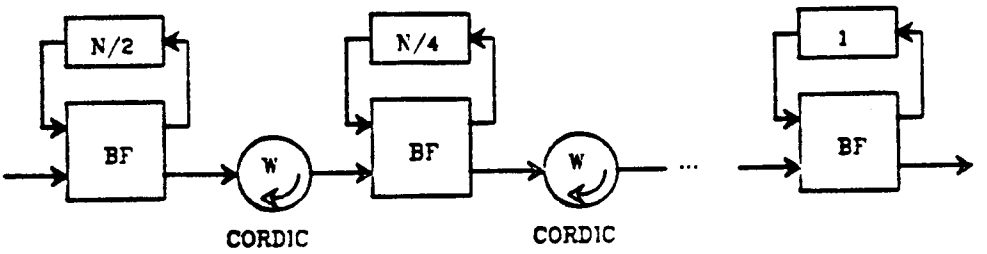



Figure 1: Despain Cascade

The $b_i$ are sent to the CORDIC rotator module which applies the proper rotations (twiddle factors) while the $b_{i+\frac{N}{2}}$ are sent back into the shift register. When all $\frac{N}{2}$ 2-point DFT's have been computed, the $b_{i+\frac{N}{2}}$ are allowed to pass out of the shift register into the CORDIC rotator. The operation of the next butterfly module is similar, except that each input datum is combined with one $\frac{N}{4}$ away. The entire computation is completed after $n = \log_2 N$ stages, where one stage consists of the butterfly module, CORDIC rotator, and shift register.

In [2], Despain points out that, for certain values of $N$, the rotations involved can be realized with less hardware than that which is required for a full CORDIC rotator. For example, the rotations involved in the computation of the FFT for $N=16$ can be performed with a set of $\frac{\pi}{2}$, $\frac{\pi}{8}$, and $\frac{\pi}{16}$ rotators. Rotations by these angles are accomplished through the use of rational approximations which can be chosen for ease of hardware implementation. An example will be given in the discussion of the $\frac{\pi}{16}$ chip below.

## 3. MODULAR CONSTRUCTION OF DFT PROCESSORS

### 3.1. Background

VLSI implementations of DFT processors can be communication-limited due to the fact that the number of pins per chip is fixed at about 100 to 200, while the number of transistors per chip is very large, about $10^5$ in 1982, and is rising rapidly. This leads to a fundamental bandwidth limitation which necessitates the development of algorithms and computational structures which minimize the amount of communication between chips.

The pipeline structure of Despain, as discussed above, lends itself to a VLSI implementation due to the ease of constructing dynamic registers necessary for pipeline computations [9]. However, for many applications, the speed of the computation needs to be higher than that which is attainable by the use of a single pipeline, and parallel structures are required. The construction of combined parallel and pipeline processors for computing the DFT is the subject of the following, although many of the results are more generally applicable to any VLSI implementation of a DFT processor which takes advantage of the inherent parallelism. These structures are designed to minimize the amount of communication necessary to compute the DFT and to make the communication hardware as simple as possible.

### 3.2. The Radix-2 Cooley-Tukey Algorithm

#### 3.2.1. Pipeline Structure

In this section, we define $N = 2^n$ to be the transform size.

Referring again to the structure in figure 1, an obvious partitioning would be to include as many stages as possible on each chip, since this structure poses no communication problems. If the number of stages per chip is denoted by $l$, we can see that the number of chips necessary to perform the computation is given by

$$C = \left\lceil \frac{n}{l} \right\rceil \tag{2}$$

Assuming that it takes $T_d$ seconds to transmit one datum through the pins of each chip and that data can be input and output simultaneously, we can process $F_t$ transforms per second where $F_t$ is given by

$$F_t = \frac{1}{T_d N} \tag{3}$$

The latency of this structure (time between the input of the first data sample and the output of the first result) is given by

$$T_l = (n-1)T_{\Psi} + nT_B + N \tag{4}$$

where $T_{\Psi}$ is the time required for a CORDIC rotation and $T_B$ is the time required for the add operation. For large transforms, the $N$ term will dominate. Despain notes that this structure is also capable of handling $2^j$ independent channels of length $N2^{-j}$ without modifying the configuration, although the results are available after $n-j$ stages. Since the structures we will derive will be proven to be functionally identical to the structure above, they will share this feature.

It should be noted that the above structure is inefficient in the sense that it does not utilize the butterfly module while passing data into or out of the shift register. The structure presented by Gold and Bially [10] avoids this and achieves twice the transform rate, but requires twice the bandwidth. Although

the Gold and Bially structure will not be discussed specifically in the following, the results will be applicable to it.

### 3.2.2. Notation

Parker [11] has recently introduced a set of algebraic tools which can be used to describe processor networks in terms of their patterns of connection. Although his notation is too limited to be used directly to handle parallel-pipeline structures, we can easily extend it to meet our needs. The operation of one stage of Despain's structure can be viewed as the application of a series of operators to the incoming data stream, which transform the data from one dimension to two dimensions and which perform the butterfly operations. If the index of a datum is defined as its coordinates $[x,y]$ in the data stream, where the original one-dimensional data stream (one row and N columns) is led by a datum with index $[0,0]$ and ending with a datum with index $[N-1,0]$, we can define the operators as follows. The first operator breaks up each data stream into several streams, and defines the operation of the shift register in each stage:

$$\mu_{(j,k)}[x,y] = [[x_u \cdots x_{k+1}x_{k-j} \cdots x_1],[y_v \cdots y_1 x_k \cdots x_{k-j+1}]] \qquad (5)$$
$$\mu^{-1}_{(j,k)}[x,y] = [[x_u \cdots x_{k-j+1}y_j \cdots y_1 x_{k-j} \cdots x_1],[y_v \cdots y_{j+1}]]$$

where $[x,y] = [[x_u \cdots x_1],[y_v \cdots y_1]]$ when x and y are described in binary notation. This is well defined as long as $j \le k \le u$. Note that $2^j$ is the number of rows each original row is broken into, and that the operator processes the input data in chunks of $2^k$ columns. As an example of the operation of $\mu$, consider the case $n=3$, $N=8$. The input data can be viewed as a one-dimensional array coming in from the left as

$$[\, d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0\,].$$

Applying the operator $\mu_{(1,3)}$ converts this stream to a two dimensional array of size $2^1 \times 2^{n-1}$

$$[\, d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0\,] \xrightarrow{\mu_{(1,3)}} \begin{bmatrix} b_3 & b_2 & b_1 & b_0 \\ b_7 & b_6 & b_5 & b_4 \end{bmatrix}.$$

It is easily seen that the two rows of this new array are the two streams which are fed simultaneously into the first butterfly unit of the Despain cascade. Applying the operator $\mu_{(1,2)}$ has a different effect

$$[\, d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0\,] \xrightarrow{\mu_{(1,2)}} \begin{bmatrix} b_5 & b_4 & b_1 & b_0 \\ b_7 & b_6 & b_3 & b_2 \end{bmatrix}.$$

Since $2^k = 2^2 = 4$, the input data is broken into 2 sets of 4 columns before it is transformed.

The butterfly operator, $B$, which cannot be defined in terms of the indexes of the data, but which takes the rows of the two-dimensional data and combines them in pairs, performing a 2-point DFT on each column in the pair, and places the sum output in the row with the smaller $y$ index and the difference output in the row with the larger $y$ index. For simplicity, we will include the twiddle multiply in this operation. As an example, a structure to perform a $2^n$-point transform can now be notated as

$$\mu_{(1,n)} B \mu^{-1}_{(1,n)} \mu_{(2,n)} B \mu^{-1}_{(2,n)} \cdots \mu_{(n,n)} B \mu^{-1}_{(n,n)} \qquad (6)$$

where one should remember that the output is in bit-reversed order. We have kept to Parker's convention and written the order of operation from left to right, i.e. $\pi_1\pi_2[x,y] = (\pi_2 \circ \pi_1)([x,y]) = \pi_2(\pi_1[x,y])$. This is done so that the strings

of operators will match the structures exactly when they are diagrammed.

Using this notation, we can define a decimation operator $\delta$ which converts one row to many by breaking each row into columns.

$$\delta_{(k)}[x,y] = [[x_u \cdots x_k],[y_v \cdots y_1 x_{k-1} \cdots x_1]] \tag{7}$$

In this case, each row is broken into $2^k$ rows and the operator is well defined if $k \le u$. Using the same example as before, the operation of $\delta_{(1)}$ is

$$[\, d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0\,] \overset{\delta_{(1)}}{\rightarrow} \begin{bmatrix} b_6 & b_4 & b_2 & b_0 \\ b_7 & b_5 & b_3 & b_1 \end{bmatrix}.$$

The operation of $\delta_{(2)}$ is shown by

$$[\, d_7\ d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0\,] \overset{\delta_{(2)}}{\rightarrow} \begin{bmatrix} b_4 & b_0 \\ b_5 & b_1 \\ b_6 & b_2 \\ b_7 & b_3 \end{bmatrix}.$$

We will soon see a need for an operator which transposes the two dimensional data in subblocks of $2^k$ rows by $2^j$ columns.

$$T_{(k,j)}[x,y] = [[x_u \cdots x_{j+1} y_k \cdots y_1],[y_v \cdots y_{k+1} x_j \cdots x_1]] \tag{8}$$

As an example, take $n=4$ where the input stream has already been operated on by $\delta_{(2)}$.

$$\begin{bmatrix} b_{12} & b_8 & b_4 & b_0 \\ b_{13} & b_9 & b_5 & b_1 \\ b_{14} & b_{10} & b_6 & b_2 \\ b_{15} & b_{11} & b_7 & b_3 \end{bmatrix} \overset{T_{(1,1)}}{\rightarrow} \begin{bmatrix} b_9 & b_8 & b_1 & b_0 \\ b_{13} & b_{12} & b_5 & b_4 \\ b_{11} & b_{10} & b_3 & b_2 \\ b_{15} & b_{14} & b_7 & b_6 \end{bmatrix}.$$

With these definitions, we may note some simple identities:

$$\mu_{(k,l)}\mu_{(j,l-k)} = \mu_{(k+j,l)} \qquad j+k \le l \tag{9}$$

$$\mu^{-1}_{(k,l-j)}\mu^{-1}_{(j,l)} = \mu^{-1}_{(j+k,l)} \quad j+k \le l \tag{10}$$

$$\mu^{-1}_{(k,l)}\mu_{(j,l)} = \begin{cases} \mu_{(j-k,l-k)} & j>k \\ \mu^{-1}_{(k-j,l-j)} & j \le k \end{cases} \quad j,k \le l \tag{11}$$

$$\delta_{(k)}T_{(k,l-k)} = \mu_{(l-k,l)} \quad k \le l \tag{12}$$

*Proof of (9):*

$$\begin{aligned} \mu_{(k,l)}\mu_{(j,l-k)}[x,y] &= \mu_{(j,l-k)}[[x_u \cdots x_{l+1}x_{l-k} \cdots x_1],[y_v \cdots y_1 x_l \cdots x_{l-k+1}]] \\ &= [[x_u \cdots x_{l+1}x_{l-(k+j)} \cdots x_1],[y_v \cdots y_1 x_l \cdots x_{l-k+1}x_{l-k} \\ &\qquad \cdots x_{l-(k+j)+1}]] \\ &= \mu_{(k+j,l)}[x,y]. \end{aligned}$$

Proofs for the other identities are as straightforward.

The usefulness of these operators and their realization in hardware is the next topic.

### 3.2.3. Derivation of Parallel-Pipeline Structures

We make the first attempt at parallelizing Despain's structure by applying the Gold and Bially procedure [10].

*Theorem 1:*

$$\mu_{(1,n)}B\mu^{-1}_{(1,n)}\cdots\mu_{(n,n)}B\mu^{-1}_{(n,n)} = \delta_{(k)}\mu_{(1,n-k)}B\mu^{-1}_{(1,n-k)}\cdots\mu_{(n-k,n-k)}B\mu^{-1}_{(n-k,n-k)} \quad (13)$$
$$T_{(k,n-k)}\mu_{(1,k)}B\mu^{-1}_{(1,k)}\cdots\mu^{(k,k)}B\mu^{-1}_{(k,k)}\mu^{-1}_{(n-k,n)}.$$

The $\delta_{(k)}$ merely breaks the data into $2^k$ rows of length $2^{n-k}$, each of which is transformed. The rows and columns are transposed by the $T_{(k,n-k)}$ operator, and the rows of the transposed matrix are then transformed. The final $\mu^{-1}_{(n-k,n)}$ rearranges the data back to one dimension.

*Proof of Theorem 1:* The identity in (12) implies that $\delta_{(k)}T_{(k,n-k)}\mu^{-1}_{(n-k,n)}$ is an identity transformation. Therefore, we can rewrite the left hand side of (13) as

$$\mu_{(1,n)}B\mu^{-1}_{(1,n)}\cdots\mu_{(n-k,n)}B\mu^{-1}_{(n-k,n)}\delta_{(k)}T_{(k,n-k)}\mu^{-1}_{(n-k,n)} \quad (14)$$
$$\mu_{(n-k+1,n)}B\mu^{-1}_{(n-k+1,n)}\cdots\mu_{(n,n)}B\mu^{-1}_{(n,n)}.$$

Note the identity

$$\mu_{(j,n)}B\mu^{-1}_{(j,n)}\delta_{(k)} = \delta_{(k)}\mu_{(j,n-k)}B\mu^{-1}_{(j,n-k)} \quad j\le n-k, k\le n. \quad (15)$$

Using this to propagate the $\delta_{(k)}$ to the left, we can rewrite (14) as

$$\delta_{(k)}\mu_{(1,n-k)}B\mu^{-1}_{(1,n-k)}\cdots\mu_{(n-k,n-k)}B\mu^{-1}_{(n-k,n-k)}T_{(k,n-k)}\mu^{-1}_{(n-k,n)} \quad (16)$$
$$\mu_{(n-k+1,n)}B\mu^{-1}_{(n-k+1,n)}\cdots\mu_{(n,n)}B\mu^{-1}_{(n,n)}.$$

Using the identity in (11), we have

$$\delta_{(k)}\mu_{(1,n-k)}B\mu^{-1}_{(1,n-k)}\cdots\mu_{(n-k,n-k)}B\mu^{-1}_{(n-k,n-k)}T_{(k,n-k)} \quad (17)$$
$$\mu_{(1,k)}B\mu_{(1,k-1)}B\mu_{(1,k-2)}\cdots B\mu_{(1,1)}B\mu^{-1}_{(n,n)}.$$

Again using (11), we see that

$$\mu_{(1,k-j)} = \mu^{-1}_{(j,k)}\mu_{(j+1,k)} \quad j+1\le k \quad (18)$$

which allows us to rewrite (17) as

$$\delta_{(k)}\mu_{(1,n-k)}B\mu^{-1}_{(1,n-k)}\cdots\mu_{(n-k,n-k)}B\mu^{-1}_{(n-k,n-k)}T_{(k,n-k)} \quad (19)$$
$$\mu_{(1,k)}B\mu^{-1}_{(1,k)}\mu_{(2,k)}B\mu^{-1}_{(2,k)}\cdots\mu_{(k,k)}B\mu^{-1}_{(n,n)}.$$

One final use of (11) gives us the expression

$$\delta_{(k)}\mu_{(1,n-k)}B\mu^{-1}_{(1,n-k)}\cdots\mu_{(n-k,n-k)}B\mu^{-1}_{(n-k,n-k)}T_{(k,n-k)} \quad (20)$$
$$\mu_{(1,k)}B\mu^{-1}_{(1,k)}\cdots\mu_{(k,k)}B\mu^{-1}_{(k,k)}\mu^{-1}_{(n-k,n)}$$

which completes the proof. Note that this theorem, although a good first step toward the parallelization of (6), does not give us a very good structure, since the transposition operation in the middle would require a buffer of size N to implement directly.

The next step is to break down $T_{(j,k)}$ to some operators which are easier to realize in hardware. Let us define the following *row interchange* operator:

$$R_{(k)}[x,y] = [x,[y_v\cdots y_2\,(y_1\oplus y_{k+1})\,]] \quad (21)$$

where "$\oplus$" means boolean "exclusive or". This operator leaves the first $2^k$ rows of the data unchanged, and interchanges the next $2^k$ rows in pairs. As an example,

$$\begin{bmatrix} b_4 & b_0 \\ b_5 & b_1 \\ b_6 & b_2 \\ b_7 & b_3 \end{bmatrix} \overset{R_{(1)}}{\rightarrow} \begin{bmatrix} b_5 & b_1 \\ b_4 & b_0 \\ b_7 & b_3 \\ b_6 & b_2 \end{bmatrix}$$

We also define a *commutator* operator:

$$\varphi_{(k,j)}[x,y] = [x,[y_v \cdots y_{k+j+1}\ (y_{k+j} \oplus x_k)\ y_{k+j-1} \cdots y_1]] \tag{22}$$
$$\varphi_{(k)}[x,y] = \varphi_{(k,0)}[x,y] = [x,[y_v \cdots y_{k+1}\ (y_k \oplus x_k)\ y_{k-1} \cdots y_1]].$$

As examples,

$$\begin{bmatrix} b_{12} & b_8 & b_4 & b_0 \\ b_{13} & b_9 & b_5 & b_1 \\ b_{14} & b_{10} & b_6 & b_2 \\ b_{15} & b_{11} & b_7 & b_3 \end{bmatrix} \overset{\varphi_{(1)}}{\rightarrow} \begin{bmatrix} b_{13} & b_8 & b_5 & b_0 \\ b_{12} & b_9 & b_4 & b_1 \\ b_{15} & b_{10} & b_7 & b_2 \\ b_{14} & b_{11} & b_6 & b_3 \end{bmatrix}$$

and

$$\begin{bmatrix} b_{12} & b_8 & b_4 & b_0 \\ b_{13} & b_9 & b_5 & b_1 \\ b_{14} & b_{10} & b_6 & b_2 \\ b_{15} & b_{11} & b_7 & b_3 \end{bmatrix} \overset{\varphi_{(2)}}{\rightarrow} \begin{bmatrix} b_{14} & b_{10} & b_4 & b_0 \\ b_{15} & b_{11} & b_5 & b_1 \\ b_{12} & b_8 & b_6 & b_2 \\ b_{13} & b_9 & b_7 & b_3 \end{bmatrix}$$

Since this operator is just a permutation within each column of the data, it can be implemented with a commutator (switch or multiplexer). In fact, it should be noted that $\varphi_{(k)} \cdots \varphi_{(k-l)}$ requires the use of $2^{l+1}$-pole switches.

Now we are in a position to prove the following lemma.

*Lemma 1:*

$$T_{(k,k)} = \mu_{(1,k)} R_{(k)} \mu_{(\bar{1},k)}^{-1} \cdots \mu_{(k,k)} R_{(k)} \mu_{(\bar{k},k)}^{-1} \varphi_{(k)} \cdots \varphi_{(1)} \tag{23}$$
$$\mu_{(1,k)} R_{(k)} \mu_{(\bar{1},k)}^{-1} \cdots \mu_{(k,k)} R_{(k)} \mu_{(\bar{k},k)}^{-1}.$$

*Proof:* Proof is by induction on k. For $k = 1$, it is easily verified that

$$T_{(1,1)} = \mu_{(1,1)} R_{(j)} \mu_{(\bar{1},1)}^{-1} \varphi_{(1)} \mu_{(1,1)} R_{(j)} \mu_{(\bar{1},1)}^{-1}. \tag{24}$$

Assume that the lemma holds for k. Note that

$$\begin{aligned} &\mu_{(1,k+1)} R_{(k+1)} \mu_{(\bar{1},k+1)}^{-1} \varphi_{(k+1)} \mu_{(1,k+1)} R_{(k+1)} \mu_{(1,k+1)} T_{(k,k)}[x,y] \qquad (25) \\ &= \varphi_{(k+1)} \mu_{(1,k+1)} R_{(k+1)} \mu_{(1,k+1)} T_{(k,k)}[[x_u \cdots x_{k+2}\ (x_{k+1} \oplus y_{k+1})\ x_k \cdots x_1], y] \\ &= \mu_{(1,k+1)} R_{(k+1)} \mu_{(1,k+1)} T_{(k,k)}[[x_u \cdots x_{k+2}\ (x_{k+1} \oplus y_{k+1})\ x_k \cdots x_1], \\ &\qquad [y_v \cdots y_{k+2} x_{k+1} y_k \cdots y_1]] \\ &= T_{(k,k)}[[x_u \cdots x_{k+2} y_{k+1} x_k \cdots x_1], [y_v \cdots y_{k+2} x_{k+1} y_k \cdots y_1]] \\ &= [[x_u \cdots x_{k+2} y_{k+1} \cdots y_1], [y_v \cdots y_{k+2} x_{k+1} \cdots x_1]] \\ &= T_{(k+1,k+1)}[x,y]. \end{aligned}$$

Using the identities

$$\mu_{(j,k)} R_{(k)} \mu_{(\bar{j},k)}^{-1} \varphi_{(l)} = \varphi_{(l)} \mu_{(j,k)} R_{(k)} \mu_{(\bar{j},k)}^{-1} \qquad j \le k, l \ne k-j+1 \tag{26}$$

$$\mu_{(j,k)} R_{(k)} \mu_{(\bar{j},k)}^{-1} \mu_{(l,k)} R_{(k)} \mu_{(\bar{l},k)}^{-1} = \mu_{(l,k)} R_{(k)} \mu_{(\bar{l},k)}^{-1} \mu_{(j,k)} R_{(k)} \mu_{(\bar{j},k)}^{-1} \qquad l \ne j, \tag{27}$$

we see that the left hand side of (25) is the same as the right hand side of (23) which completes the proof.

We now look at the structure of Theorem 1 for three cases.

*Case 1:* $n-k = k$. For this case, the right hand side of (13) reduces to

$$\delta_{(k)}\mu_{(1,k)}B\mu_{(\bar{1},k)}^{-1}\cdots\mu_{(k,k)}B\mu_{(\bar{k},k)}^{-1}T_{(k,k)} \qquad (28)$$
$$\mu_{(1,k)}B\mu_{(\bar{1},k)}^{-1}\cdots\mu_{(k,k)}B\mu_{(\bar{k},k)}^{-1}\mu_{(\bar{k},n)}^{-1}.$$

Substituting in for $T_{(k,k)}$ from Lemma 1 and using the identities

$$\mu_{(j,k)}B\mu_{(\bar{j},k)}^{-1}\mu_{(l,k)}R_{(k)}\mu_{(\bar{l},k)}^{-1} = \begin{cases} \mu_{(l,k)}R_{(k)}\mu_{(\bar{l},k)}^{-1}\mu_{(j,k)}B\mu_{(\bar{j},k)}^{-1} & l\neq j \\ \mu_{(j,k)}BR_{(k)}\mu_{(\bar{j},k)}^{-1} & l=j \end{cases} \qquad (29)$$

and

$$\mu_{(j,k)}R_{(k)}\mu_{(\bar{j},k)}^{-1}\mu_{(j,k)}B\mu_{(\bar{j},k)}^{-1} = \mu_{(j,k)}R_{(k)}B\mu_{(\bar{j},k)}^{-1} \qquad (30)$$

leads to the expression

$$\delta_{(k)}\mu_{(1,k)}BR_{(k)}\mu_{(\bar{1},k)}^{-1}\cdots\mu_{(k,k)}BR_{(k)}\mu_{(k,k)}\varphi_{(k)}\cdots\varphi_{(1)} \qquad (31)$$
$$\mu_{(1,k)}R_{(k)}B\mu_{(\bar{1},k)}^{-1}\cdots\mu_{(k,k)}R_{(k)}B\mu_{(\bar{k},k)}^{-1}\mu_{(\bar{k},n)}^{-1}$$

for the case $n = 2k$.

We now have a structure which is easily realizable, since $\mu_{(j,k)}BR_{(k)}\mu_{(\bar{j},k)}^{-1}$ can be performed by a set of butterfly modules, half of which reverse the order in which they output the butterfly results. The $\mu_{(j,k)}R_{(k)}B\mu_{(\bar{j},k)}^{-1}$ are also implemented by a column of butterfly modules, although half of these reverse the order in which they accept their inputs. As we have noted previously, the $\varphi_{(k)}\cdots\varphi_{(1)}$ can be implemented with a $2^k$-pole commutator. An example of this structure will be demonstrated below.

*Case 2:* $n-k > k$. For this case, we note that the transposition operator $T_{(k,n-k)}$ in (13) is difficult to implement since it changes the number of data streams (rows) from $2^k$ up to $2^{n-k}$. However, it is easy to show that

$$T_{(k,n-k)} = T_{(k,k)}\mu_{(\bar{k},2k)}^{-1}\mu_{(n-k,n)} \qquad k<n-k. \qquad (32)$$

Substituting this into (13), expanding $T_{(k,k)}$ by Lemma 1 and propagating the $\mu$'s gives us the structure

$$\delta_{(k)}\mu_{(1,n-k)}B\mu_{(\bar{1},n-k)}^{-1}\cdots\mu_{(n-2k,n-k)}B\mu_{(\bar{n}-2k,n-k)}^{-1}\mu_{(n-2k+1,n-k)}BR_{(k)}\mu_{(\bar{n}-2k+1,n-k)}^{-1} \qquad (33)$$
$$\cdots\mu_{(n-k,n-k)}BR_{(k)}\mu_{(\bar{n}-k,n-k)}^{-1}\varphi_{(k)}\cdots\varphi_{(1)}\mu_{(1,k)}R_{(k)}B\mu_{(\bar{1},k)}^{-1}$$
$$\cdots\mu_{(k,k)}R_{(k)}B\mu_{(\bar{k},k)}^{-1}\mu_{(\bar{k},2k)}^{-1}$$

which is as easy to implement as (31).

*Case 3:* $n-k < k$. In this case, the application of the transposition operator $T_{(k,n-k)}$ in (13) results in fewer data streams (rows) . Consider the portion of (13) which comes after this operator. Since it is of the same form as the left hand side of (13), we can apply Theorem 1 again and rewrite the right hand side of (13) as

$$\delta_{(k)}\mu_{(1,n-k)}B\mu_{(\bar{1},n-k)}^{-1}\cdots\mu_{(n-k,n-k)}B\mu_{(\bar{n}-k,n-k)}^{-1}T_{(k,n-k)} \qquad (34)$$
$$[\delta_{(2k-n)}\mu_{(1,n-k)}B\mu_{(\bar{1},n-k)}^{-1}\cdots\mu_{(n-k,n-k)}B\mu_{(\bar{n}-k,n-k)}^{-1}$$
$$T_{(2k-n,n-k)}\mu_{(1,2k-n)}B\mu_{(\bar{1},2k-n)}^{-1}\cdots\mu_{(2k-n,2k-n)}B\mu_{(\bar{2}k-n,2k-n)}^{-1}$$
$$\mu_{(\bar{n}-k,k)}^{-1}]\,\mu_{(\bar{n}-k,n)}^{-1}$$

where the effect of Theorem 1 has been bracketed for clarity. The combination of operators $T_{(k,n-k)}\delta_{(2k-n)}$ leaves the number of rows constant and can be implemented with the help of the following lemma.

*Lemma 2:*

$$T_{(k,j)}\delta_{(k-j)} = \mu_{(1,j)}R_{(k)}\mu_{(1,j)}^{-1}\cdots\mu_{(j,j)}R_{(k)}\mu_{(j,j)}^{-1}\varphi_{(j,k-j)}\cdots\varphi_{(1,k-j)} \qquad (35)$$
$$\mu_{(1,j)}R_{(k)}\mu_{(1,j)}^{-1}\cdots\mu_{(j,j)}R_{(k)}\mu_{(j,j)}^{-1} \qquad k>j.$$

The proof of this is similar to that of Lemma 1 and is omitted. If we use this lemma and the identities (29) and (30) to expand (34), we arrive at

$$\delta_{(k)}\mu_{(1,n-k)}BR_{(n-k)}\mu_{(1,n-k)}^{-1}\cdots\mu_{(n-k,n-k)}BR_{(n-k)}\mu_{(n-k,n-k)}^{-1}\varphi_{(n-k,2k-n)} \qquad (36)$$
$$\cdots\varphi_{(1,2k-n)}\mu_{(1,n-k)}R_{(n-k)}B\mu_{(1,n-k)}^{-1}\cdots\mu_{(n-k,n-k)}R_{(n-k)}B\mu_{(n-k,n-k)}^{-1}$$
$$T_{(2k-n,n-k)}\mu_{(1,2k-n)}B\mu_{(1,2k-n)}^{-1}\cdots\mu_{(2k-n,2k-n)}B\mu_{(2k-n,2k-n)}^{-1}$$
$$\mu_{(n-k,n)}^{-1}\mu_{(n-k,n)}.$$

The first half of this structure is now easily realizable, and the $T_{(2k-n,n-k)}$ will be expanded in one of the three ways we have just looked at. If necessary, case 3 should be applied recursively to the structure until the last $T$ operator can be expanded as in case 1 or 2.

### 3.2.4. Partitions of Derived Structures

Clearly, one should again try to partition the structure into chips by including on each chip as many stages of each pipeline as possible. We will again denote the number of butterfly units per chip by $l$, and assume that it is constant, although the following is easily extended to those cases where it is not. From the results of the previous section, one can see that communication between the pipelines is required no later than after the first $n-k$ stages, which implies that $l \le n-k$. If $l \mid n-k$ and $l \mid \langle n \rangle_{n-k}$ (where $\langle\cdot\rangle_m$ denotes the residue mod $m$ and $\mid$ is read "divides"), the structures of the previous section can be used directly. However, if these requirements are not met, one can use the identities

$$\mu_{(j,k)}R_{(k)}\mu_{(j,k)}^{-1}\varphi_{(l)} \qquad (37)$$
$$= \varphi_{(l)}\mu_{(j,k)}R_{(k)}\mu_{(j,k)}^{-1} \qquad l \neq k-j+1$$

and

$$\mu_{(j,k)}R_{(l)}\mu_{(j,k)}^{-1}\varphi_{(\tau,l-k)} \qquad (38)$$
$$= \varphi_{(\tau,l-k)}\mu_{(j,k)}R_{(l)}\mu_{(j,k)}^{-1} \qquad \tau \neq j-l+1$$

to move the $\varphi$'s through the structure so that there is always a multiple of $l$ stages between them. An example of the this will be shown below.

These identities may also be used to derive structures when $l$ is not a constant, as long as no $l$ violates the inequality $l \le n-k$.

### 3.2.5. Examples of Parallel-Pipeline Structures

For the following examples, we choose $n=6$ ($N=64$), and derive structures for various numbers of data streams ($2^k$) and number of butterfly modules per chip ($l$).

*Example 1:* $k=3$, $l=3$. This corresponds directly to case 1 and is notated by

$$\delta_{(3)}\mu_{(1,3)}BR_{(3)}\mu_{(1,3)}^{-1}\cdots\mu_{(3,3)}BR_{(3)}\mu_{(3,3)}\varphi_{(3)}\cdots\varphi_{(1)} \qquad (39)$$
$$\mu_{(1,3)}R_{(3)}B\mu_{(1,3)}^{-1}\cdots\mu_{(3,3)}R_{(3)}B\mu_{(3,3)}^{-1}\mu_{(3,6)}^{-1}$$

where the commutator consists of one 8-pole switch. This example is diagrammed in figure 2, where the reversal of the butterfly inputs or outputs is shown by an "R" placed at the input or output of the butterfly module,

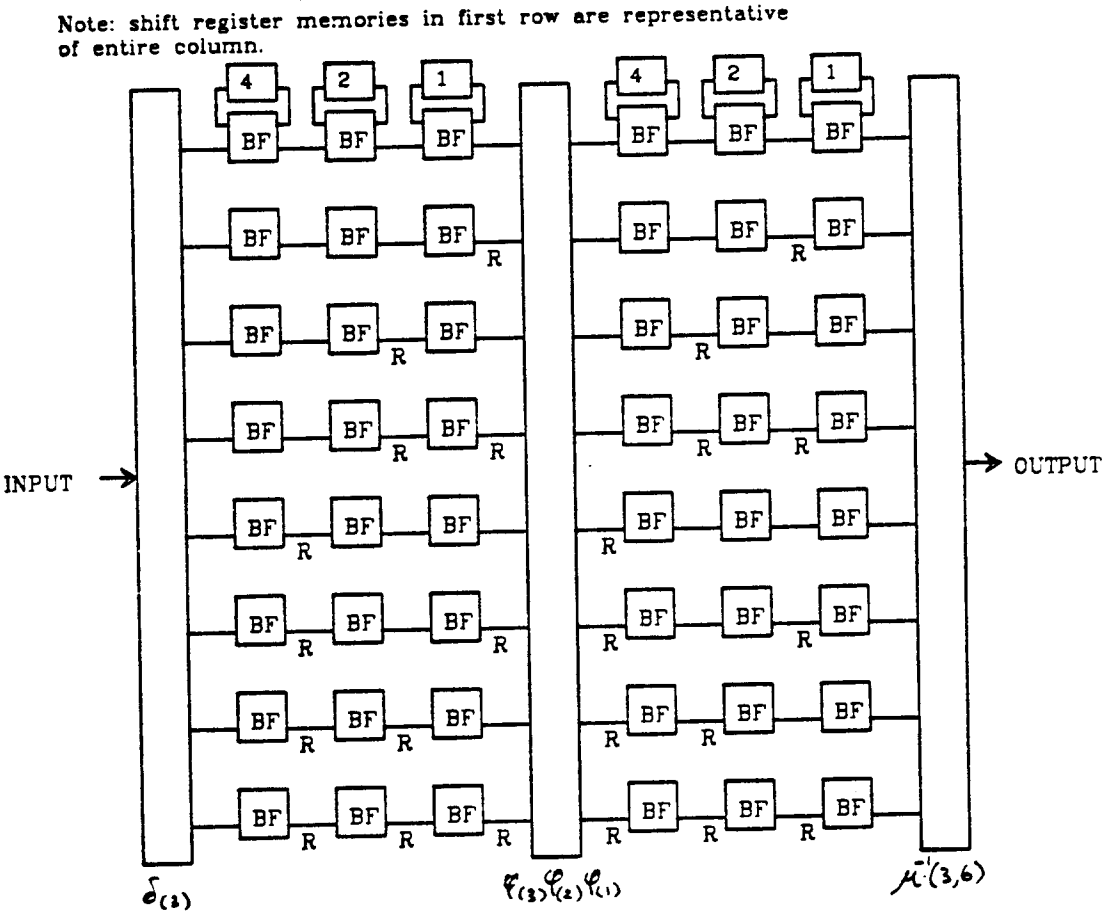



Figure 2: Example 1 ($k=3$, $l=3$)

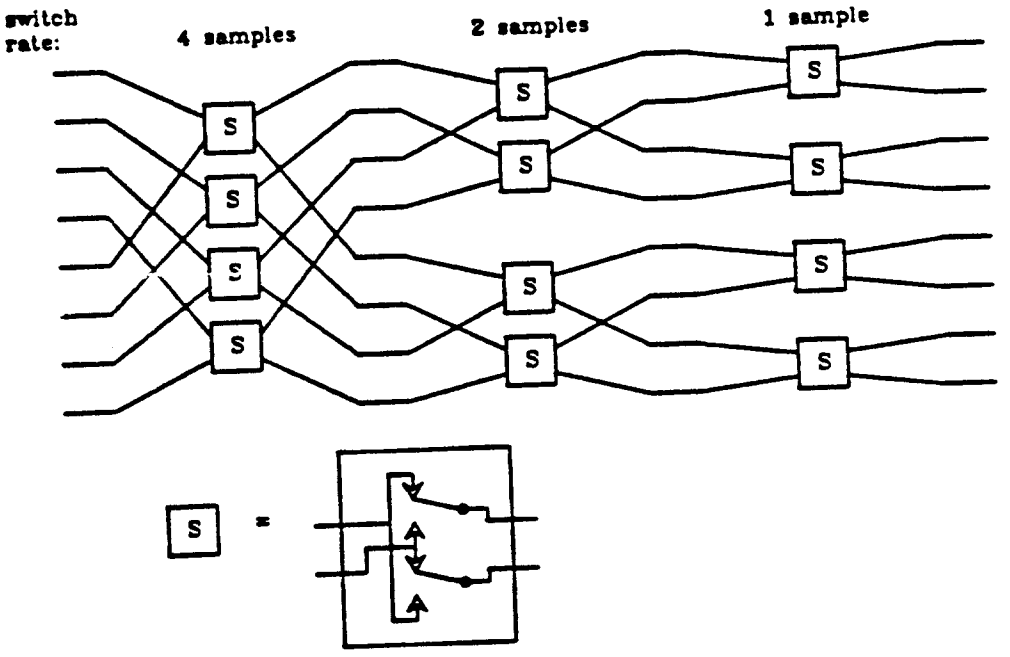



Figure 3: Commutator for Example 1

respectively. The commutator required is shown in figure 3, where all the internal switches start in the position shown, then switch at the rates shown in the figure.

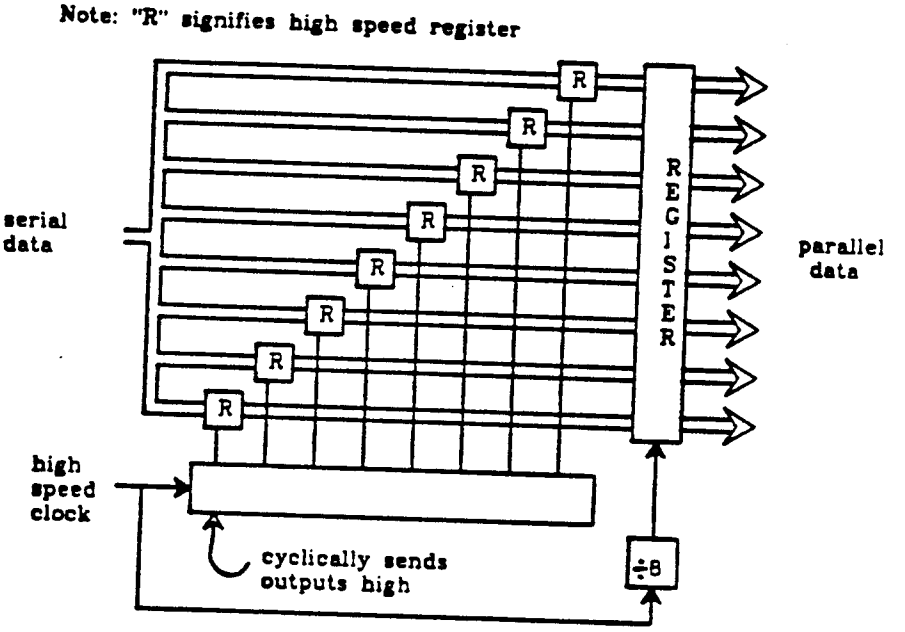



Figure 4: Decimator for Example 1

The design for the decimator shown in figure 4 assumes that the data arrives in a single stream (from an A/D, for example), and requires the use of a few fast registers. The $\mu_{(3,6)}^{-1}$ at the output would in general not be implemented, since its implementation would require logic which was $2^k$ times as fast as the logic in the FFT processor.

*Example 2:* $k=3$, $l=2$. Here, we modify (39) using the identity in (37) to

$$\delta_{(3)}\mu_{(1,3)}BR_{(3)}\mu_{(1,3)}^{-1}\mu_{(2,3)}BR_{(3)}\mu_{(2,3)}^{-1}\varphi_{(3)}\varphi_{(2)} \qquad (40)$$
$$\mu_{(3,3)}BR_{(3)}\mu_{(3,3)}^{-1}\mu_{(1,3)}R_{(3)}B\mu_{(1,3)}^{-1}\varphi_{(1)}$$
$$\mu_{(2,3)}R_{(3)}B\mu_{(2,3)}^{-1}\mu_{(3,3)}R_{(3)}B\mu_{(3,3)}^{-1}\mu_{(3,6)}^{-1}$$

where we now have two commutators, the first consisting of 2 4-pole switches and the second consisting of 1 2-pole switch. This configuration is diagrammed in figure 5. In this case, the first commutator consists of the first two sections and the second commutator consists of the third section of the commutator shown in the last example. Note that we could just as easily have moved $\varphi_{(2)}$ to the right and combined it with $\varphi_{(1)}$ with no change in the operation of the other parts of the circuit.

*Example 3:* $k=4$, $l=2$. The first transposition operator can be expanded as in case 3, and the resulting transposition operator can be expanded as n case 1. The resulting structure is

$$\delta_{(4)}\mu_{(1,2)}BR_{(2)}\mu_{(1,2)}^{-1}\mu_{(2,2)}BR_{(2)}\mu_{(2,2)}^{-1}\varphi_{(2,2)}\varphi_{(1,2)} \qquad (41)$$
$$\mu_{(1,2)}R_{(2)}BR_{(2)}\mu_{(1,2)}^{-1}\mu_{(2,2)}R_{(2)}BR_{(2)}\mu_{(2,2)}^{-1}\varphi_{(2)}\varphi_{(1)}$$
$$\mu_{(1,2)}R_{(2)}B\mu_{(1,2)}^{-1}\mu_{(2,2)}R_{(2)}B\mu_{(2,2)}^{-1}\mu_{(2,4)}^{-1}\mu_{(2,6)}^{-1}.$$

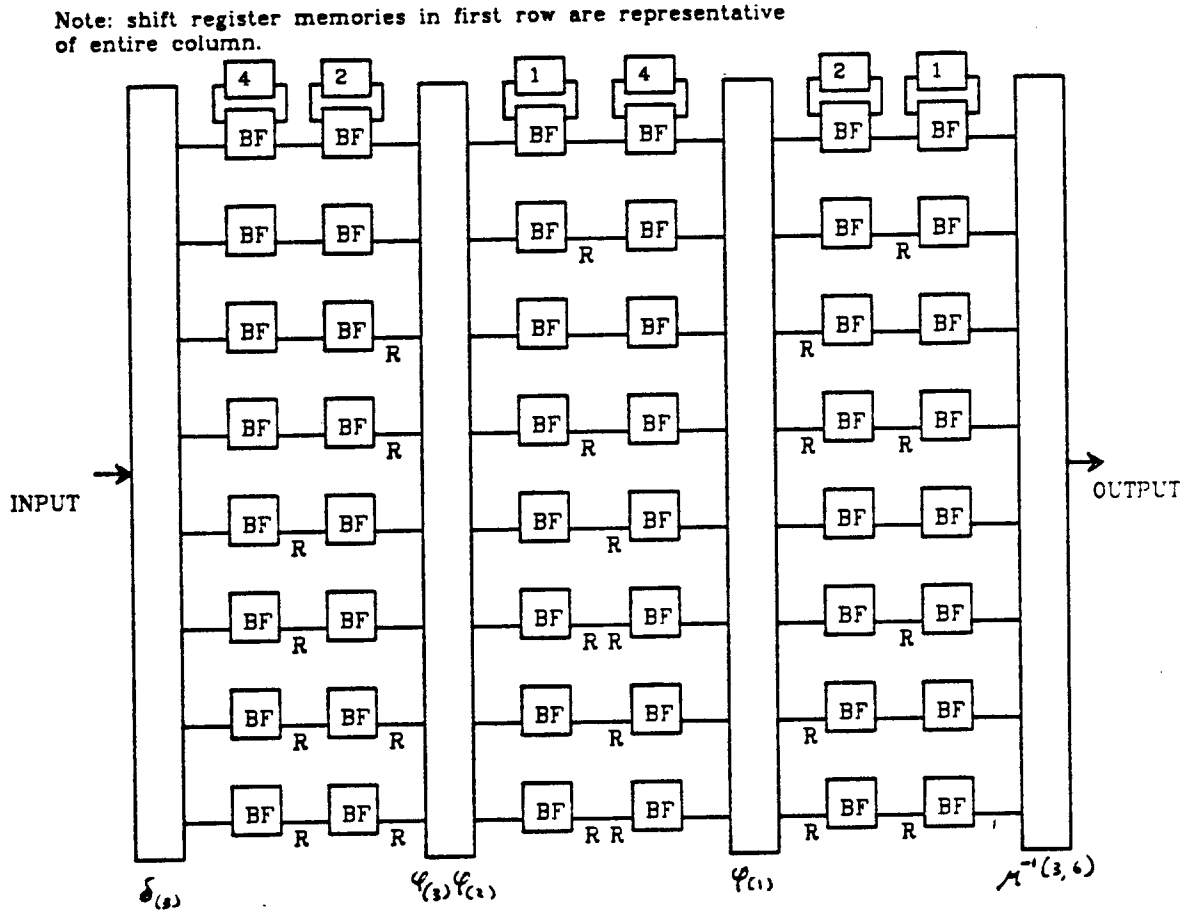



Figure 5: Example 2 ($k=3$, $l=2$)

### 3.2.6. Benchmarks of the Derived Structures

If $N=2^n$ is the transform size, $2^k$ the number of data streams, and $l$ the number of butterfly modules per chip, the number of chips required to compute the radix-2 FFT is

$$C = \left\lceil \frac{n}{l} \right\rceil 2^k \tag{42}$$

and we can process

$$F_t = \frac{2^k}{T_d N} \tag{43}$$

transforms per second. It should be noted again that this structure, like the pipeline it was derived from, can process $2^j$ intermixed channels of length $N2^{-j}$.

For a desired transform rate, the number of chips required can be computed by eliminating $k$ from the above equations and noting that k must be an integer. Thus,

$$C = \left\lceil \frac{n}{l} \right\rceil 2^{\lceil \log_2 F_t T_d N \rceil}. \tag{44}$$

Since $l \le n-k$, we can derive a lower bound for $C$ given by

$$\begin{aligned} C &\ge \frac{n}{n-k} 2^k \\ &\ge \frac{n}{\log_2 \frac{1}{F_t T_d}} F_t T_d N. \end{aligned} \tag{45}$$

This is of interest, since it states that the dependence of the lower bound of $C$ on $F_t$ is worse than linear. Thus, although the speedup is optimal in the sense that, to raise $F_t$ by a factor of $2^k$, one multiplies the number of butterfly modules by the same factor, the speedup is not optimal if one counts chips. Figure 6 has a graph of $C$ versus $F_t T_d N$ for the case $N=1024$, where it is also assumed that $l$ is limited to $\leq 4$ due to area limitations on each chip.

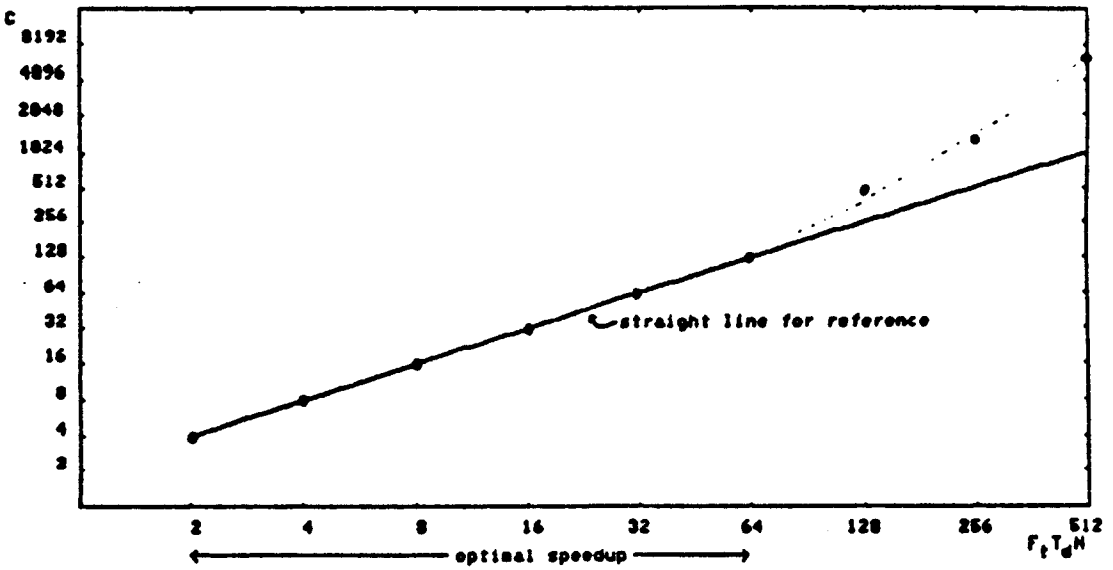



Figure 6: Cost in number of chips

Since $k \leq n-1$, the highest transform rate possible is given by

$$F_t(max) = \frac{1}{2T_d} \tag{46}$$

at a cost of

$$C = \frac{Nn}{2} \tag{47}$$

chips. The latency is given by

$$T_l = (n-1)T_{\#} + nT_B + \frac{N}{2^k} \tag{48}$$

For large transforms, the latency is reduced by $2^k$ over the original pipeline structure.

### 3.3. The Radix-r Cooley-Tukey Algorithm

#### 3.3.1. Pipeline Structure

The pipeline structure of Despain extends easily to arbitrary radix by including in each stage a computational unit capable of performing an $r$-point DFT and a CORDIC rotator module. An example of a radix-4 FFT processor of this form can be found above and in [1]. If each chip contains $l$ of these stages, and if the transform size is now given by $N=r^n$, the number of chips needed to perform the FFT is given by

$$C = \left\lceil \frac{\log_r N}{l} \right\rceil = \left\lceil \frac{n}{l} \right\rceil \tag{49}$$

and the transform rate is again

$$F_t = \frac{1}{T_d N} \tag{50}$$

### 3.3.2. Notation

The notation for the radix-2 case can be extended in a straightforward manner to cover the radix-r case. The index of the data is now expressed in terms of its base-r representation as $[x,y] = [[x_u \cdots x_1],[y_v \cdots y_1]]$. The definitions of $\mu_{(j,k)}$ and $\delta_{(k)}$ remain unchanged, but the definitions of $R_{(k)}$ and $\varphi_{(k,j)}$ must be generalized to

$$R^{+}_{(k)}[x,y] = [x,[y_v \cdots y_2 \langle y_1+y_{k+1}\rangle_r]] \tag{51}$$
$$R^{-}_{(k)}[x,y] = [x,[y_v \cdots y_2 \langle y_1-y_{k+1}\rangle_r]]$$

and

$$\varphi_{(k,j)}[x,y] = [x,[y_v \cdots y_{k+j+1} \langle x_k-y_{k+j}\rangle_r\, y_{k+j-1} \cdots y_1]] \tag{52}$$
$$\varphi_{(k,0)}=\varphi_{(k)}.$$

Note that the $\varphi$ operator can still be implemented as a commutator. In this more general case, Despain's structure is notated in the same way as for radix-2, with the $B$ operator now interpreted as a radix-$r$ DFT. Theorem 1 still holds, and Lemmas 1 are easily generalized as follows.

*Lemma 1:*

$$T_{(k,k)} = \mu_{(1,k)}R^{+}_{(k)}\mu^{-1}_{(1,k)} \cdots \mu_{(k,k)}R^{+}_{(k)}\mu^{-1}_{(k,k)}\varphi_{(k)} \cdots \varphi_{(1)} \tag{53}$$
$$\mu_{(1,k)}R^{-}_{(k)}\mu^{-1}_{(1,k)} \cdots \mu_{(k,k)}R^{-}_{(k)}\mu^{-1}_{(k,k)}.$$

*Lemma 2:*

$$T_{(k,j)}\delta_{(k-j)} = \mu_{(1,j)}R^{+}_{(k)}\mu^{-1}_{(1,j)} \cdots \mu_{(j,j)}R^{+}_{(k)}\mu^{-1}_{(j,j)}\varphi_{(j,k-j)} \cdots \varphi_{(1,k-j)} \tag{54}$$
$$\mu_{(1,j)}R^{-}_{(k)}\mu^{-1}_{(1,j)} \cdots \mu_{(j,j)}R^{-}_{(k)}\mu^{-1}_{(j,j)} \quad k>j.$$

The structures for the different cases are identical to the structures derived before, except that the $R^{+}_{(k)}$ operators will be propagated to the left and the $R^{-}_{(k)}$ operators will be propagated to the right. For example, the $n-k = k$ case would generalize to

$$\delta_{(k)}\mu_{(1,k)}BR^{+}_{(k)}\mu^{-1}_{(1,k)} \cdots \mu_{(k,k)}BR^{+}_{(k)}\mu^{-1}_{(k,k)}\varphi_{(k)} \cdots \varphi_{(1)} \tag{55}$$
$$\mu_{(1,k)}R^{-}_{(k)}B\mu^{-1}_{(1,k)} \cdots \mu_{(k,k)}R^{-}_{(k)}B\mu^{-1}_{(k,k)}\mu^{-1}_{(k,n)}.$$

This is again easily realizable, although the data at the input and output to the radix-$r$ DFT modules will need to be rearranged in a more complicated pattern (although there will still be no buffering required.) Structures for the other cases are generalized similarly.

### 3.3.3. Benchmark of the Radix-r Structures

If the number of data streams is given by $r^k$, and if $l$ is the number of computational units per chip, the number of chips is given by

$$C = \left\lceil \frac{n}{l} \right\rceil r^k \tag{56}$$

and the transform rate by

$$F_t = \frac{r^k}{T_d N} \tag{57}$$

Following the same argument as before, we can arrive at a lower bound for $C$ for a given $F_t$.

$$C \geq \frac{n}{\log_r \frac{1}{F_t T_d}} F_t T_d N. \tag{58}$$

The highest possible transform rate is given by

$$F_t(max) = \frac{1}{r T_d} \tag{59}$$

at a cost of

$$C = \frac{Nn}{r} \tag{60}$$

chips. The latency becomes

$$T_l = (n-1)T_W + nT_B + \frac{N}{r^k} \tag{61}$$

where $T_B$ is now the time it takes to do a radix-r DFT and $T_W$ is the time required for a CORDIC rotation.

## 3.4. The Use of the Winograd Algorithm in Pipeline Processors

### 3.4.1. Background

Certain DFT sizes are easier to implement than others. Although in the past, powers of 2 have been a common choice due to the Cooley-Tukey algorithm [4], there is often a large advantage to employing other sizes as will be seen below. In fact, it is possible to reduce the number of multiplies to $O(N)$ for certain values of $N$. The algorithms which achieve this are based on the reduction of DFTs to convolutions by Rader [12], the Good prime factor algorithm [5], and the combining of multipliers due to Winograd [6].

We have already seen that the important idea in all the FFT algorithms is to factor the DFT and thereby reduce the number of operations over its direct calculation. This corresponds to a factorization of $N$, the transform size. Since we are interested mainly in pipeline organizations of FFT processors, each of the factors of $N$ will correspond to a pipeline module.

### 3.4.2. Module Implementations

#### 3.4.2.1. Base $2^n$ Modules

We have already discussed these modules in detail and there is quite a bit in the open literature about them. The pipeline processors of Despain as described in [1] and [2] would be of the most interest here.

#### 3.4.2.2. Base 3 Modules

The derivation of the base 3 module begins with the DFT for $N=3$:

$$A_r = \sum_{k=0}^{2} B_k W^{rk} \qquad r=0,1,2 \tag{1}$$

where

$$W = \frac{1}{2} + \frac{\sqrt{-3}}{2}.$$

Define

$$a_1 = B_1 + B_2$$

$$a_0 = a_1 + B_0$$

$$a_2 = a_1 - 2B_2.$$

Equation (1) can now be factored into

$$A_0 = a_0$$

$$A_1 = A_0 - \frac{3a_1}{2} + \frac{a_2\sqrt{-3}}{2}$$

$$A_3 = A_1 - a_2\sqrt{-3}.$$

The signal flow graph of this transform is shown in figure 7.

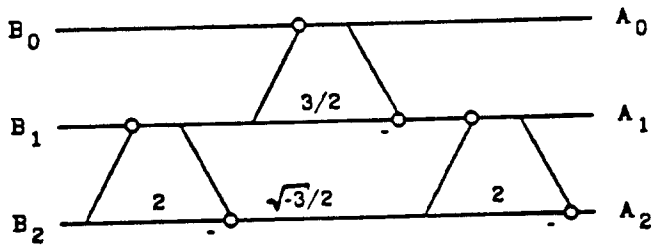



Figure 7: Signal Flow Graph of Base 3 DFT

It can now be observed that this calculation requires 7 real additions, one multiplication by $\sqrt{3}$, and several shift operations.

The term $\sqrt{3}$ can be approximated by a ratio of simple integers as in [2]. The result is that only about 4 complex additions per data point are required for the base 3 transform.

If the algorithm of figure 7 is to be realized in pipeline form, considerable data reordering is required. Thus, we will use a slight modification of the signal flow graph. This is illustrated in the circuit diagrams below. Figure 8 shows the overall base 3 circuit. Figures 9 and 10 show the add/subtract portions of the base 3 circuit, while the multiplication module could be implemented as either a full multiplier circuit or as a rational approximation as shown in figures 11 , 12 , and 13. If the base 3 module is not the last module in a cascade, shift registers would be necessary to multiplex the data as was done for the base $2^n$ modules previously derived.

From these figures it can be seen that, while the number of arithmetic operations is small, the complexity of rearranging the data is large. In particular it would be costly to employ a base 3 module at the front of a large FFT pipeline due to the large shift register memory which would be required relative to the base $2^n$ modules.

A similar analysis of other prime factors such as 5, 7, and 11 indicate that the complexity of rearranging the data grows very quickly with the value of the prime. Because it is severe for the case $N=3$, by the time the case $N=5$ is considered, the complexity negates many of the advantages of the prime factor technique, especially for the pipeline processors considered here. The problem is not so great for specialized, single random access memory processors.

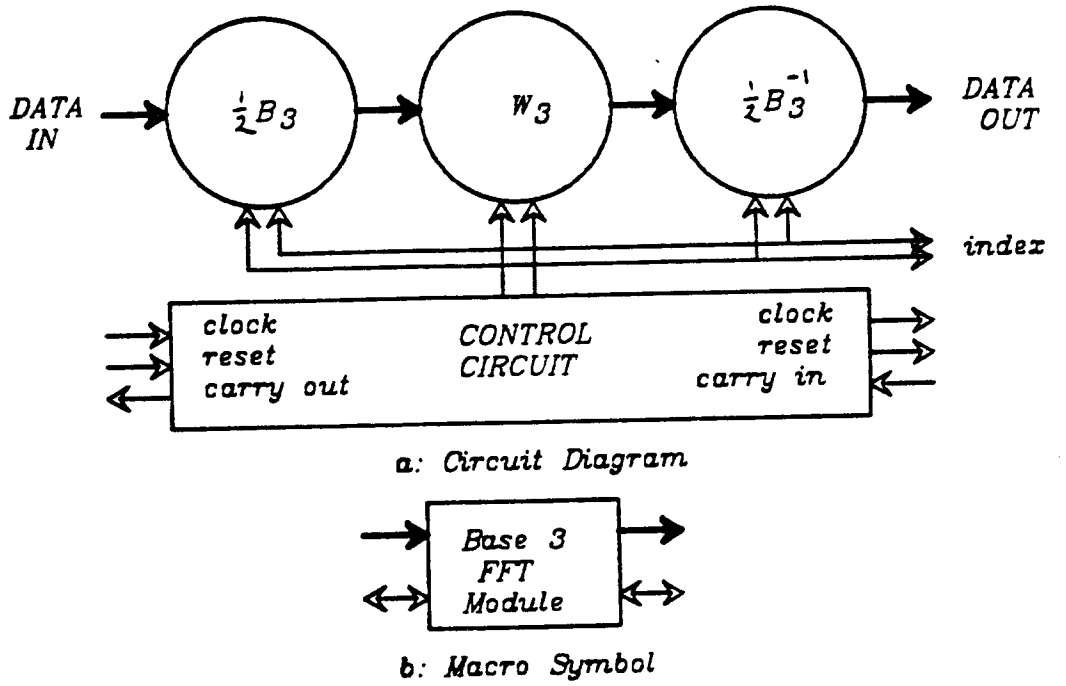



Figure 8: Overall Base 3 Circuit

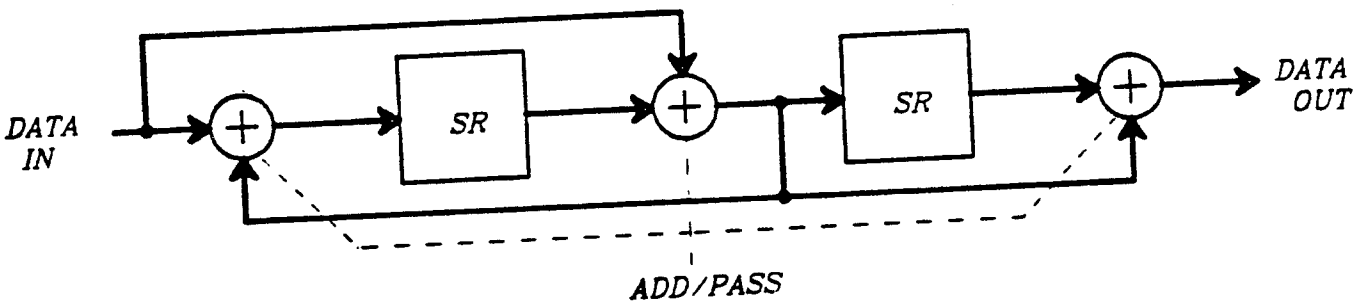



Figure 9: First Half of Base 3 Butterfly

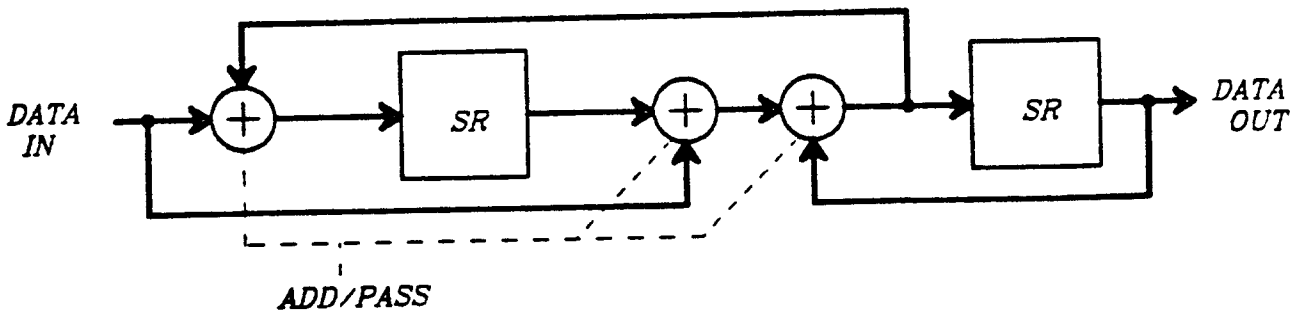
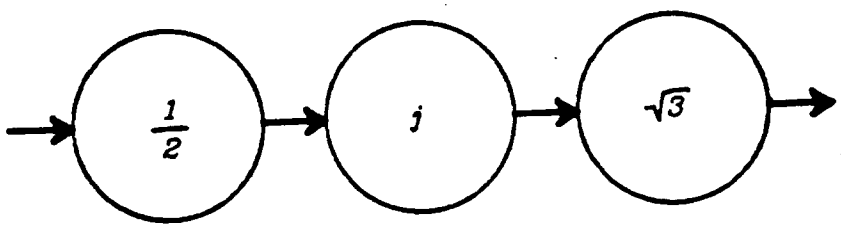


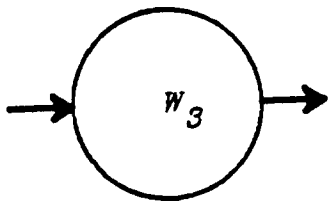

Figure 10: Last Half of Base 3 Butterfly

### 3.4.2.3. Base 5 Module

There are several approaches to deriving a base 5 algorithm suitable for pipeline cascade processors. If the algorithms outlined by Winograd [6] and developed in detail by Kolba and Parks [13] are to be employed, then the flow

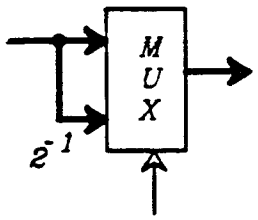



*a: Block Diagram*

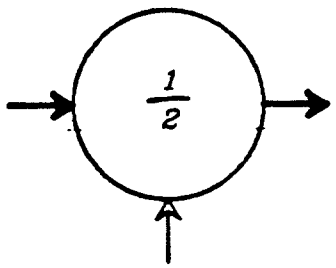



*b: Macro Diagram*

Figure 11: Base 3 Specialized Multiplier Circuit




*a: Circuit Diagram*

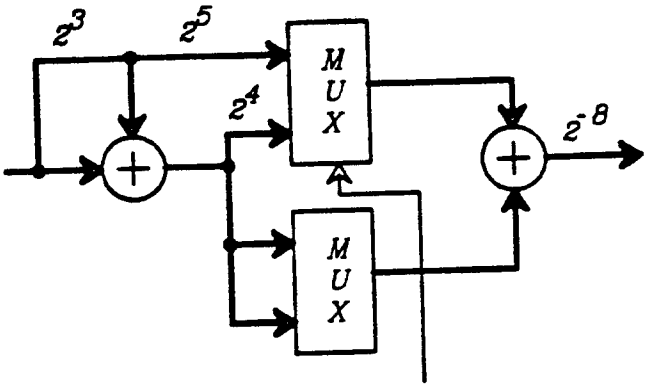



*b: Macro Symbol*

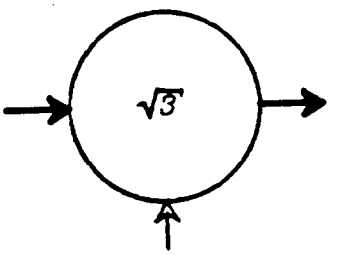
Figure 12: Divide by 2 Circuit

graph of figure 14 results. Although this form of the algorithm could be employed with the use of input and output buffers and buffers for temporary storage, it is better to derive an algorithm that is inherently in the pipeline form.

This algorithm will be derived to meet the following constraints:




*a: Circuit Diagram*




*b: Macro Symbol*

Figure 13: Root 3 Circuit

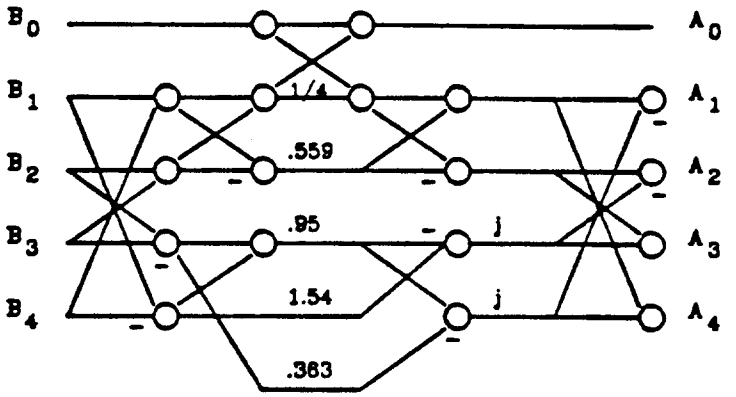



Figure 14: Base 5 Signal Flow Graph

1. pipeline organization
2. Winograd form (central multipliers)
3. minimized multiplies
4. minimized memory requirements.

The first step is due to Rader [12]. As an example, we will look at N=5. We can write the DFT in matrix form as

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & W^1 & W^2 & W^3 & W^4 \\ 1 & W^2 & W^4 & W^1 & W^3 \\ 1 & W^3 & W^1 & W^4 & W^2 \\ 1 & W^4 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \\ B_4 \end{bmatrix}$$

where $W = e^{-j\frac{2\pi}{5}}$. The difficult part of this calculation is

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} W^1 & W^2 & W^3 & W^4 \\ W^2 & W^4 & W^1 & W^3 \\ W^3 & W^1 & W^4 & W^2 \\ W^4 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{bmatrix} \tag{1}$$

since

$$A_0 = B_0+B_1+B_2+B_3+B_4$$

and

$$A_1 = B_0+a_1$$

$$A_2 = B_0+a_2$$

$$A_3 = B_0+a_3$$

$$A_4 = B_0+a_4.$$

Since 5 is prime, we are guaranteed [14] that we can find a primitive root, g, such that $g^k \bmod 5$ for $k = 0,1,2,\ldots$ forms the set {1,2,3,4} which is the set of all the positive integers less than 5. This primitive root defines a permutation of the set {1,2,3,4} by applying the function $g^k \bmod 5$ to the numbers {0,1,2,3} which gives us {1,2,4,3}. If we apply this permutation to the computation in (1), switching the last two rows and columns of the matrix, we end up with

$$\begin{bmatrix} a_1 \\ a_2 \\ a_4 \\ a_3 \end{bmatrix} = \begin{bmatrix} W^1 & W^2 & W^4 & W^3 \\ W^2 & W^4 & W^3 & W^1 \\ W^4 & W^3 & W^1 & W^2 \\ W^3 & W^1 & W^2 & W^4 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_4 \\ B_3 \end{bmatrix}. \tag{9}$$

This can be recognized as the cyclic correlation of $\vec{W}$ and $\vec{B}$ where

$$\vec{W} = [W^1 \;\; W^2 \;\; W^4 \;\; W^3]^T$$

$$\vec{B} = [B^1 \;\; B^2 \;\; B^4 \;\; B^3]^T.$$

It is well known that convolutions and correlations can be performed with the DFT in the following manner [15]

$$\vec{a} = DFT^{-1}(DFT(\vec{W}) \times DFT(\vec{B})) \tag{10}$$

where × is a component by component multiply. It can be shown [6, 16] that the DFT of $\vec{W}$, which can be precalculated, is always pure real or pure imaginary, so that the multiplications which need to be performed involve at worst one real and one complex value. The calculation of the DFT of $\vec{B}$ and the inverse DFT in (10) can be performed by the 4 point DFT algorithm which has already been defined. A signal flow graph of the entire $N=5$ algorithm is shown in figure 15. This algorithms shows a great deal more regularity than that of Kolba and Parks which we saw previously, and is thus much more suitable for a pipeline organization. A circuit which performs this algorithm is shown in figures 16, 17, 18, and 19. The multiplier $W_5$ is most easily realized as a full multiplier circuit.

### 3.4.2.4. Base 7 Module

The derivation of the base 5 algorithm above can be used as a prototype for the base 7 algorithm. The primitive root 3 defines the permutation {1,3,2,6,4,5} which is used to reorder the inputs and outputs. Figures 20, 21, 22, 23, and 24 show the form of the circuit. First the input data is reordered according to the sequence given above. Then a base 6 DFT is applied. This is just a base 3 followed by a base 2 transform, since 3 and 2 are relatively prime. Next a

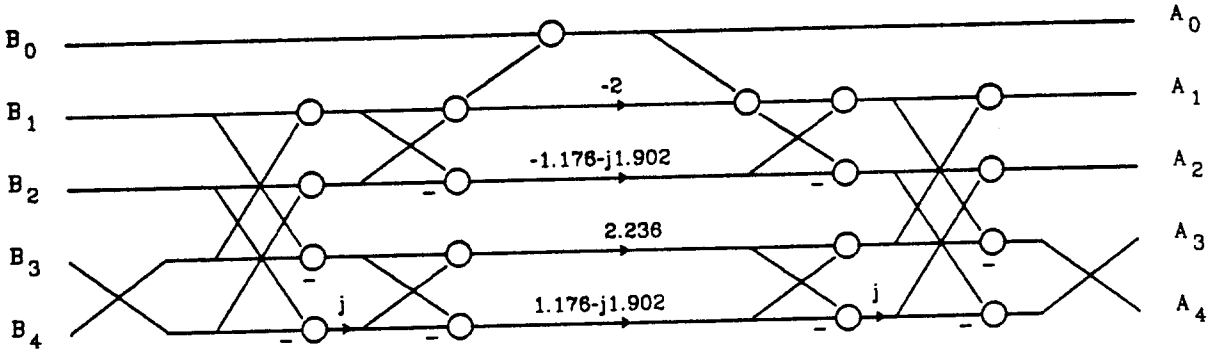



Figure 15: New Base 5 Signal Flow Graph

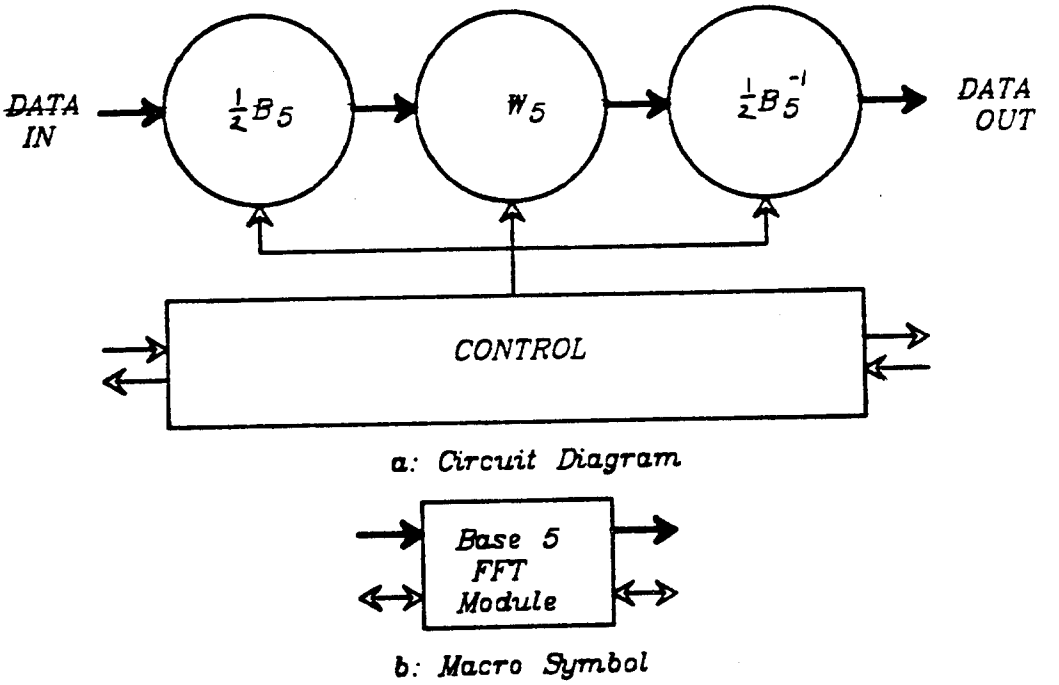



Figure 16: Base 5 DFT Module

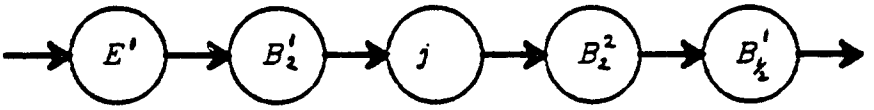
multiplication by the DFT of the $\vec{W}$ is performed and then each of the above operations is undone in reverse order as in the base 3 and base 5 algorithms.

### 3.4.2.5. Base 11 Module

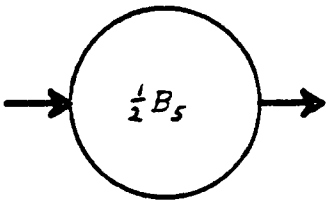
Since 11=2×5+1, 11 is not an attractive base, as each of the base 5 DFTs would require a multiplier for a total of three multipliers.

### 3.4.2.6. Base 13 Module

Because 13=3×4+1, and since we have good algorithms for 3 and 4, this is an attractive base. The procedure to derive this algorithm is the same as for the base 5 and base 7 cases. The primitive root in this case is 2 which defines the permutation {1,2,4,8,3,6,12,11,9,5,10,7}. The only new circuit which is needed is the reorder network shown in figure 25.

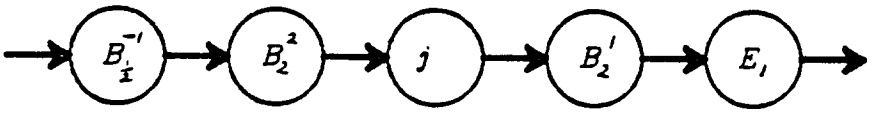



a: Block Diagram

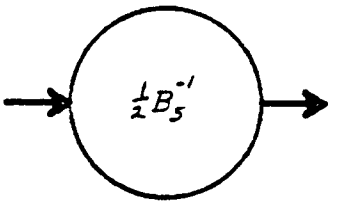



b: Macro Symbol

Figure 17: First Half of Base 5 Butterfly




a: Block Diagram




b: Macro Symbol

Figure 18: Last Half of Base 5 Butterfly

### 3.4.2.7. Base 17 Module

Since we have a good base 16 algorithm, base 17 is attractive as well. The primitive root 3 defines the permutation {1,3,9,10,13,5,15,11,16,14,8,7,4,12,2,6}. Again only a new reordering circuit is needed as in figures 26 and 27.

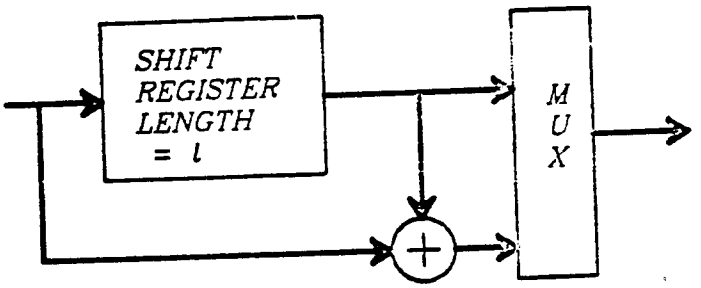
### 3.4.2.8. Higher Bases

Above 17, the Rader/Winograd form of DFT algorithms becomes more difficult.

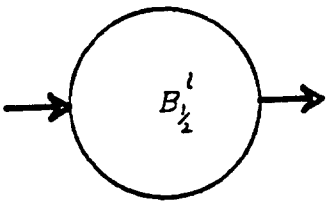
### 3.4.2.9. Proposed FFT Cascade

We have now developed a number of modules which can be employed to form a full FFT Cascade. The central multiplications of the base 5, 7, 13, and 17 modules can be combined into a single central multiplication. The various transform sizes which can be obtained with the restriction that only one multiplier be used is quite large.

Choose any combination of the $a_i$ such that each $a_i$ is used only once and

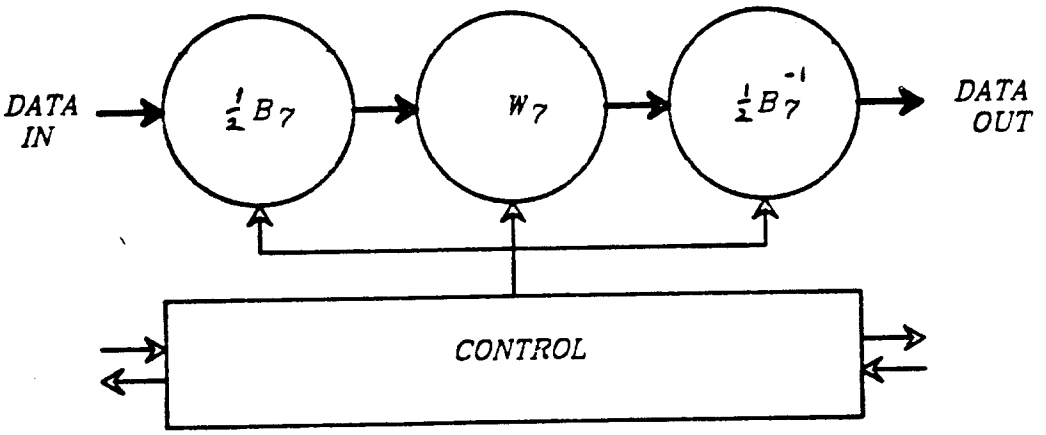



a: Circuit Diagram

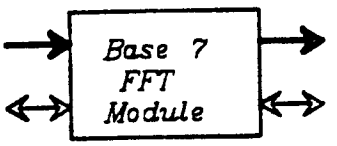



b: Macro Symbol

Figure 19: Half butterfly




a: Circuit Diagram




b: Macro Symbol

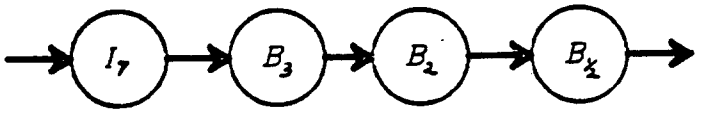
Figure 20: Base 7 DFT Module

$$N = a_i a_j a_k \cdots$$

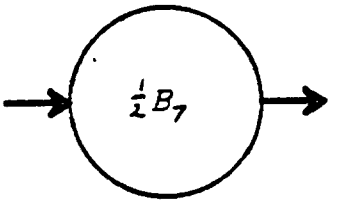
and where

$a_i$ = (2, 2, 3, 4, 5, 7, 13, 16, 17).

The maximum size will be limited to

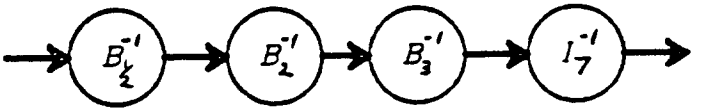



*a: Block Diagram*

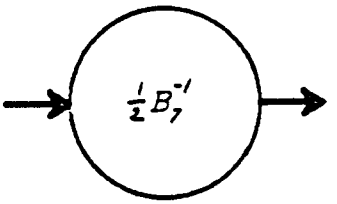



*b: Macro Symbol*

Figure 21: First Half Base 7 Butterfly




*a: Block Diagram*




*b: Macro Symbol*

Figure 22: Last Half Base 7 Butterfly

$$N = \prod_i a_i = 17{,}821{,}440 .$$

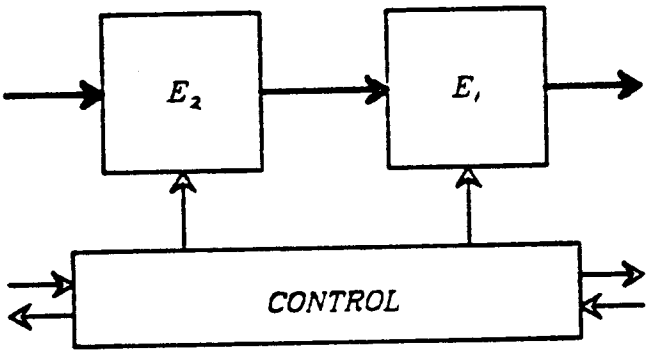
This should be large enough for most purposes. Figure 28 shows an FFT cascade of this size and smaller cascades can easily be derived from this figure. Some attractive values are

$N$ = 48, 256, 768, 2304, 4352, 13056, 39168.

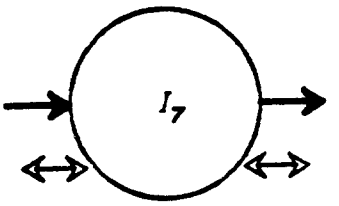
Note that multiple, multiplexed channels of shorter transform length can be obtained by tapping the cascade structure as shown.

### 3.4.2.10. Module Memory Costs

Each module has two different components to its cost. The first is the memory cost which is a function of the position of the module in the pipeline. Define a factor $l$ that represents the product of all the bases of the modules that follow. The factor $l$ then represents the number of samples to be stored in the shortest memory (shift register). For the base $2^n$ modules, $2l$ memory words will be required since a single sample has both a real and imaginary part. By adding up the memory segments from the previous figures, the relative costs of

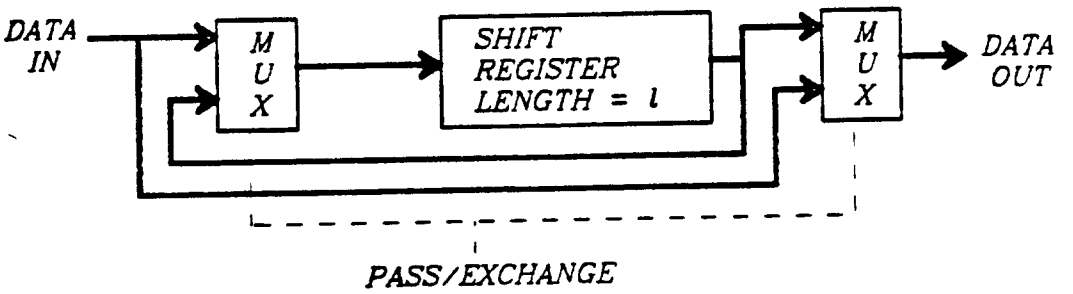



*a: Circuit Diagram*

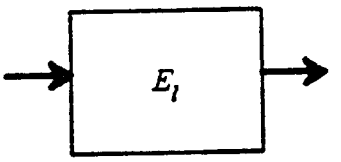



*b: Macro Symbol*

Figure 23: Reorder Network for Base 7 Module



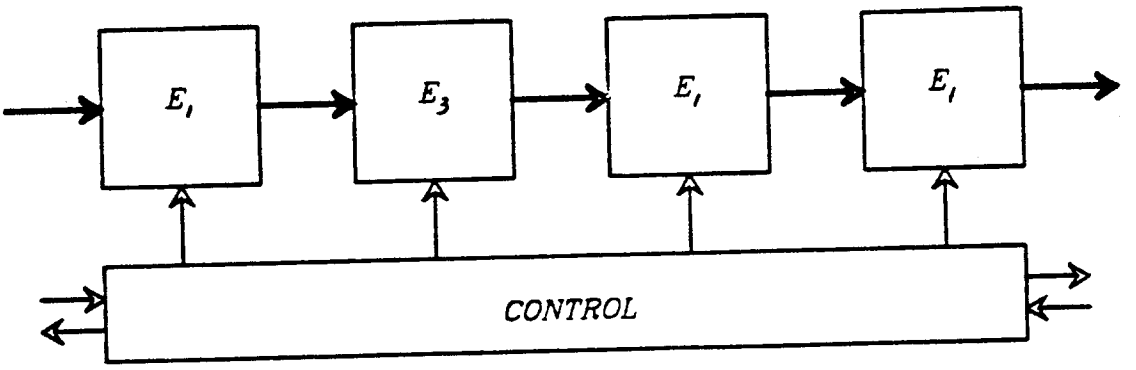

*a: Circuit Diagram*



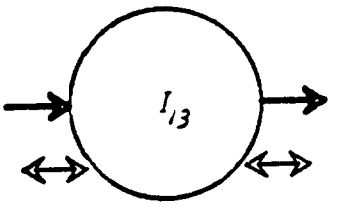

*b: Macro Symbol*

Figure 24: Exchange Circuit Module "E"

memory for each type module can be determined. The results are given in table 1.

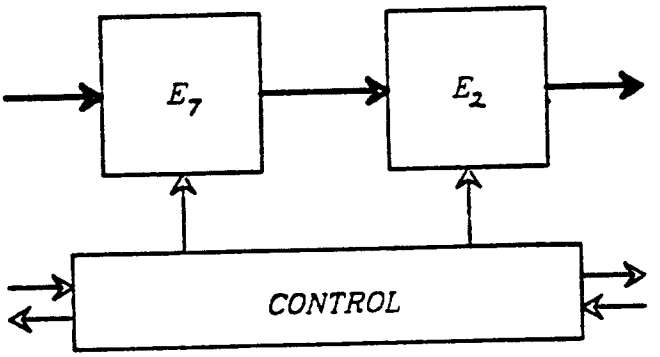



*a: Circuit Diagram*

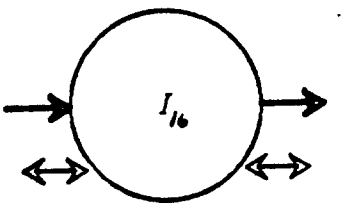



*b: Macro Symbol*

Figure 25: Reorder Network for Base 13 Module

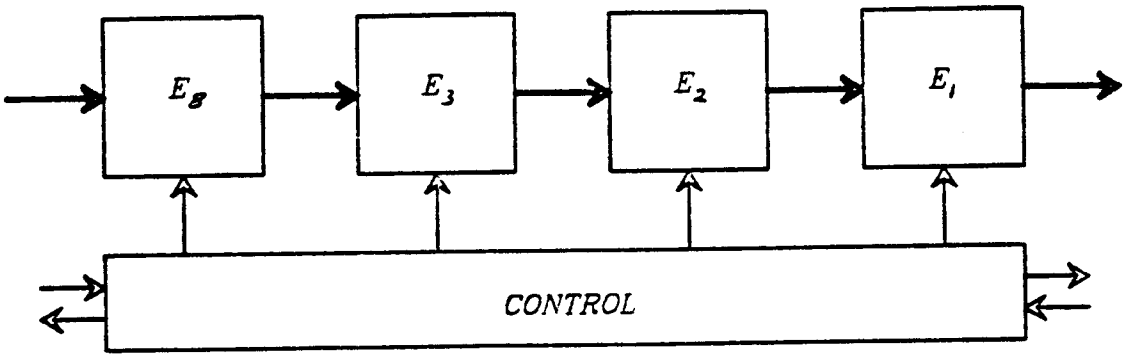



*a: Circuit Diagram*

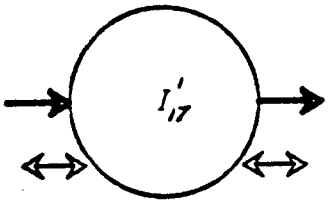



*b: Macro Symbol*

Figure 26: Base 16 Reorder Network




*a: Circuit Diagram*




*b: Macro Symbol*

Figure 27: Base 17 Reorder Network

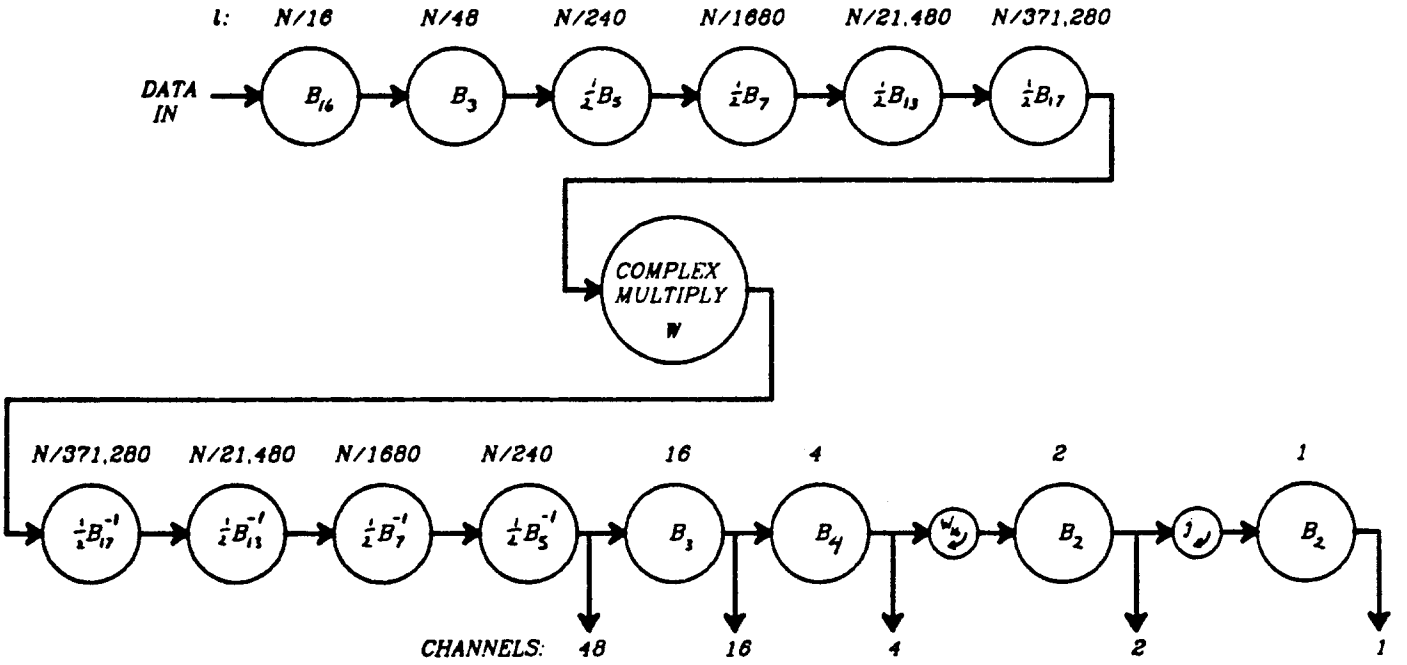



Figure 28: FFT Cascade for $N$ = 17,821,440

| Table 1: Module Memory Costs | | |
|---|---|---|
| Size | Weighting Factor ($\omega$) | Cost ($\omega l$) |
| 2 | 1.0 | 2.0 |
| 3 | 2.0 | 6.0 |
| 5 | 2.5 | 12.5 |
| 7 | 2.33 | 16.33 |
| 13 | 2.77 | 36.0 |
| 17 | 3.75 | 63.75 |

The second major cost factor is fixed for each type of module. This cost is determined by the number of adders and 2-input multiplexers, grouped under the term "Adds". Table 2 summarizes this cost for each module type.

| Table 2: Module "Adds" (Central Multiply not included) | |
|---|---|
| Size | Number of "Adds" |
| 2 | 4 |
| 3 | 12 |
| 4 | 10 |
| 5 | 20 |
| 7 | 40 |
| 13 | 64 |
| 17 | 68 |

# 4. IMPLEMENTATION TECHNOLOGY

Due to the pipeline organization of the FFT processor, it was originally thought that charge transfer technologies such as "Bucket Brigade" or "Charge Coupled Devices" would be the proper approach to take. We will first give a brief review of the principle of operation of charge transfer devices and of the basic implementation of such circuitry. Subsequently, we will discuss the difficulties in technology and layout that arise in the implementation of practical systems. Finally, we present the reasons why the charge-transfer approach was abandoned and standard silicon-gate n-channel MOS technology was favored for the implementation of the prototype building blocks.

## 4.1. Review of the Charge-Transfer Principle

"Charge Transfer Device" (CTD) [3] is a generic term which has come to be applied to a family of functional solid-state electronic devices which includes Bucket Brigade Devices (BBD) and Charge-Coupled Devices (CCD). Under the application of a proper sequence of clock pulses, these devices move quantities of electrical charge in a controlled manner across a semiconductor substrate. Using this basic mechanism, they can perform an amazingly wide range of electronic functions including image sensing, data storage, logic operations, and signal processing. Because of the shift-register nature of these devices, they are a natural match to serial memory or to pipelined signal processing systems.

There are two basic approaches to forming charge-transfer devices. In bucket brigade structures information is represented by majority carriers, e.g. the holes in the $p^+$-type diffused regions constituting the source or drain areas of a p-channel MOS transistor (Fig 29). Electrically a bucket brigade device can be understood as a dynamically operated chain of pass transistors. Under the influence of two clocks half the pass transistors are strongly turned off at any one time, while the others provide potential barriers that permit to skim of the signal charge from the background of majority carriers contained in the source electrodes. Capacitive coupling of the clocks to the diffused electrodes between the pass gates will properly bias these areas to make them act as sources or drains respectively.

In the charge coupled devices, a more sophisticated electrode structure is employed to create moving potential wells, that travel along the surface of the silicon crystal. Information is contained as a packet of minority carriers in these moving potential wells. Practically all the charge contained in one potential well location gets transfered to the subsequent position. Because the signal charge does not have to be skimmed off a majority carrier background, it is easier to obtain good "transfer efficiency".

The crucial performance parameter in both kind of devices is 'transfer inefficiency', a fractional number that indicates what part of the signal charge fails to get transferred properly and gets mixed into the subsequent signal packet. Bucket brigade devices have typical transfer inefficiencies of $10^{-3}$ to $10^{-4}$ per stage while CCDs achieve on the order of $10^{-5}$ per transfer. Overall transfer inefficiency of a charge transfer structure between input and output or between subsequent signal regenerators should not exceed 50% for digital applications. This determines the maximum number of stages that can be safely put into a single charge transfer section. Analog bucket brigade shift registers with several hundred stages can be built with acceptable signal degradation. On the other hand, CCD delay lines with up to 10,000 electrodes can be built with good performance.

While BBD's are normally implemented with only two clock phases, CCDs have been built with from 1 to 4 sets of clocked electrodes. Devices with 3 or 4

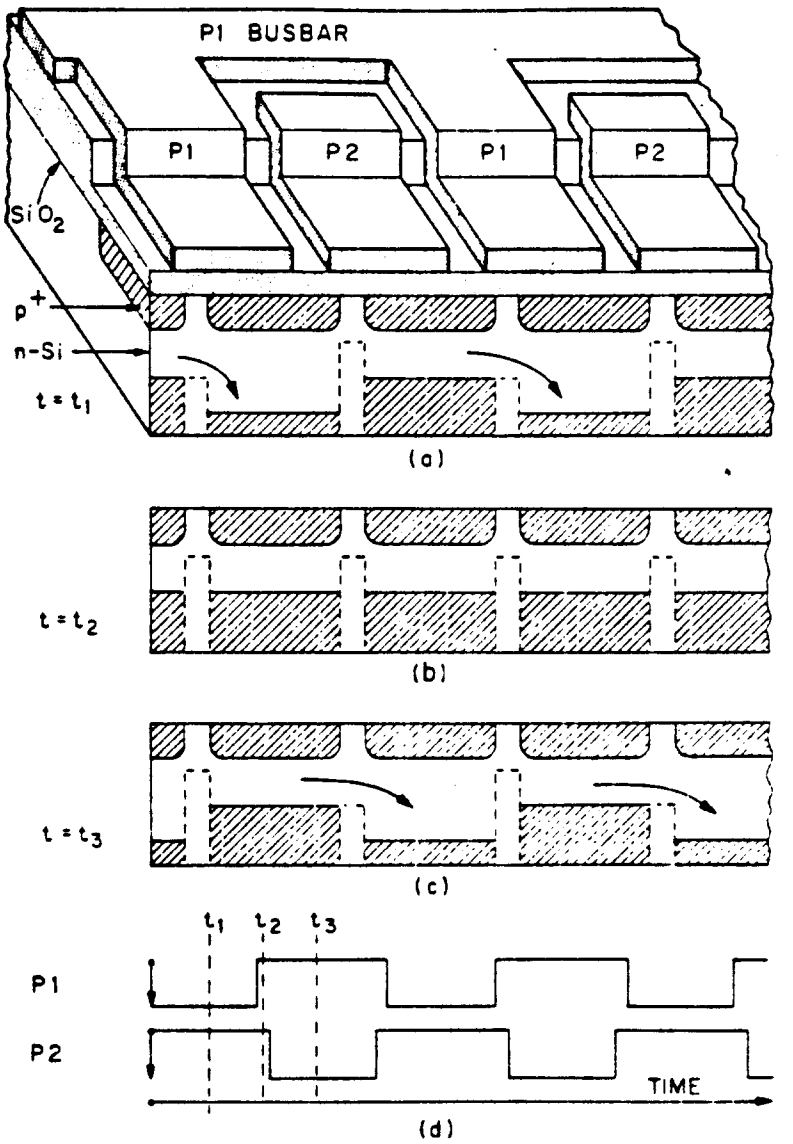



Figure 29 (a): Schematic rendering of a $p$-channel BBD with the associated potential diagram shown in the cross section of the silicon substrate. (b,c): Potential diagrams shown for various biasing conditions illustrating the transfer of charge. (d): The corresponding time slots marked in the diagram of the clock waveforms.

electrodes per stage can use simple unstructured electrodes, while the 1 or 2-phase devices need to have some structure built into each electrode in order to uniquely define the direction of charge transfer. The typical means to define this directionality is to use a step in the thickness of the insulating oxide layer under the gate electrode or a shallow implant at the surface of the substrate to produce a suitable asymmetry in the interface potential underneath each electrode. In both cases the signal charge will then accumulate in the part of the electrode where it has the lower potential energy and will be prevented from moving backwards by the barrier part of the potential profile. For this to work, the amount of signal charge must be restricted to be completely contained behind the barrier. For the same clock voltages and identical areas of the

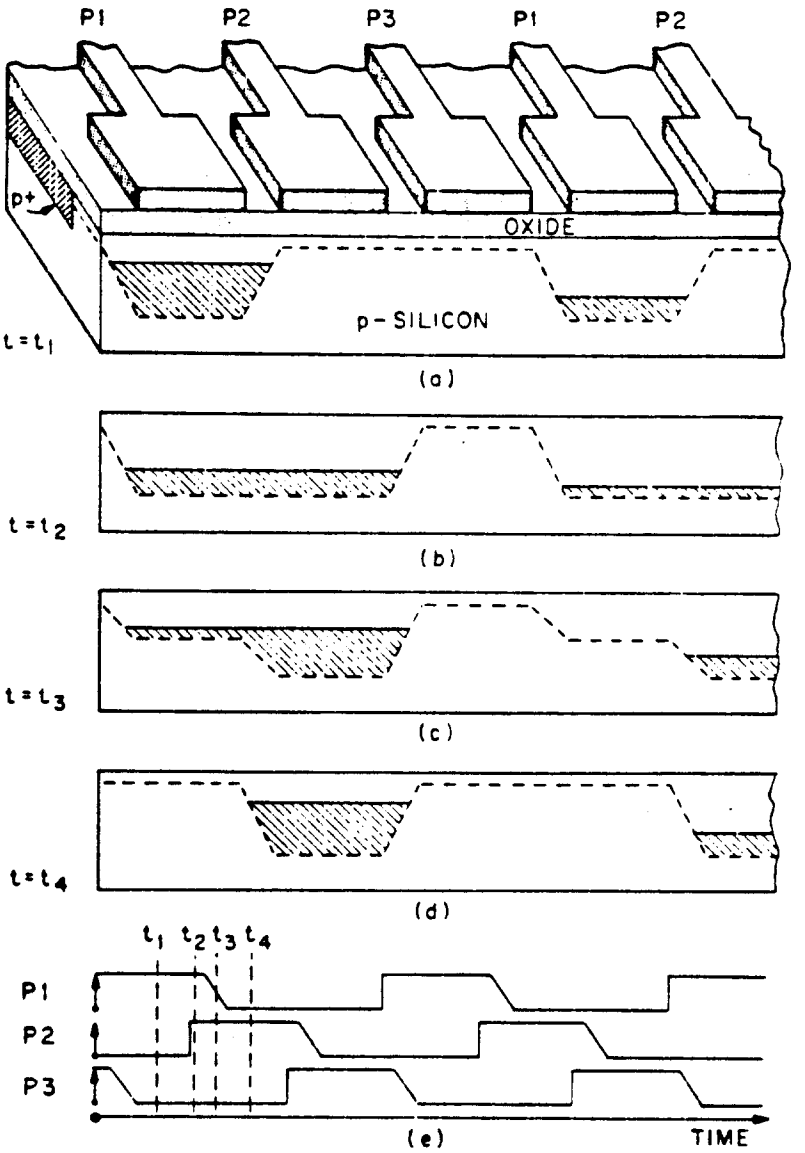



Figure 30 (a): Schematic rendering of a 3-phase *n*-channel CCD with the charge carrying potential wells shown in the cross section of the silicon substrate. (b,c,d): Potential wells shown at subsequent time intervals illustrating the transfer of charge. (e): The corresponding time slots marked in the diagram of the waveforms.

storage electrodes, the charge handling of devices with directional electrodes is thus smaller than that of devices with simple, uniform electrodes.

Bucket brigade devices have implanted or diffused source drain electrodes and asymmetrically arranged metal or silicon electrodes that serve simultaneously as transistor gates and as capacitors that properly bias the source and drain electrodes. These devices can be built with a single technological gate level. The area underneath the gap between the gate electrodes is bridged by the strongly doped source/drain areas (figure 29). Charge coupled devices on the other hand move minority carriers through lightly doped substrate regions close to the surface. The potential of all these areas must be carefully

controlled. Inter-electrode gaps lead to unpredictable signal handling and poor reliability. Thus the whole active channel area must be covered with clock electrodes. This normally implies the use of at least two conductive levels capable of providing good MOS gates or the use of special technological tricks such as selectively doped sheets of high-resistivity polysilicon. The normal CCD structure thus typically consists of two or more levels of partially overlapping gate electrodes (figure 30).

Because of the difficulty of routing different sets of clocks to all paths of a large charge transfer system, efforts have been made to build CCDs with only a single clocked electrode which covers the whole channel. It may at first seem surprising, but such structures are indeed possible, and experimental devices have been built in several laboratories. However, these structure typically require a more complicated, very tightly controlled fabrication process, and provide only very small signal handling capability measured as a fraction of the applied clock amplitudes. We are not aware of any practical systems that have been built with such uni-phase CCDs.

### 4.2. Implementation Trade-offs

There are a few fundamental trade-offs in the construction of charge transfer devices. As mentioned above, signal handling can be traded off versus the number of clock phases. Uni-phase devices can carry very little charge per volt of the applied clock signals. Two-phase devices have reasonable signal handling capabilities. From three phases on up the signal handling is very good, but the problem of routing all these clock phases to the proper points gets worse with increasing number of clocks. The figure of merit: (maximum signal charge) / (number of clock phases) reaches an optimum at four phases.

Similarly there is a trade-off in the number of clock phases that need to be routed to the charge transfer channel versus the sophistication of the implementation technology. All practical charge coupled systems need at least two levels of gate electrodes. This is true even for the uni-phase device because of the input/output structures. In addition, the two-phase devices need at least one to two implants in the area of the transfer channel to provide the necessary directionality for the electrodes. Uni-phase devices need at least three to four implants (or corresponding oxide-patterning steps to provide stepped electrodes). The dosage of these implants have to be very carefully controlled to guarantee proper operation of the device.

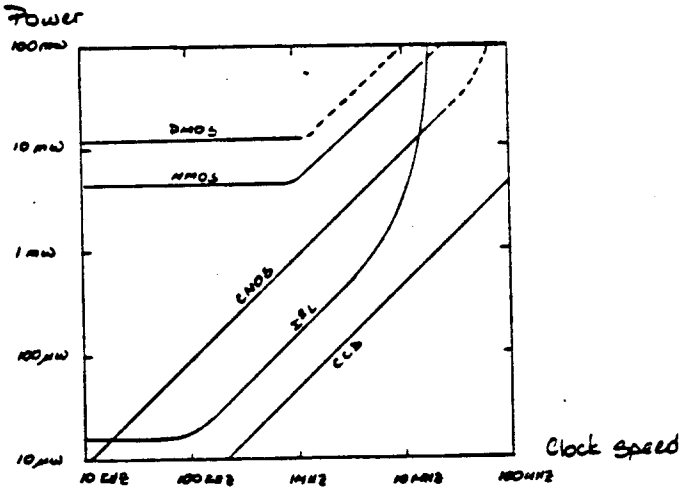
Bucket brigade devices can be constructed with both sets of electrodes belonging for the two clock phases in a single level of metal or poly-crystalline silicon. They can thus be constructed with standard n-channel or p-channel MOS technology. However, serial registers with good transfer efficiency are normally much larger than a corresponding CCD implementation.

In all these devices there is only a single plane in which the signal charge can move around. The transfer of these charge packets can occur only close to the silicon crystal surface. Crossing of two signal paths is thus only possible at the expense of considerable extra circuitry. Either the charge packets belonging to the two separate channels are time-multiplexed through the crossing point, which requires extra clocks and control gates; or at least one of the signal streams must be taken out of the charge domain and converted with a sense amplifier to a corresponding voltage. This voltage or current signal can then be transmitted in a wire across the charge transfer channel containing the other signal path. The voltage or current signal can then drive an injector circuit that recreates a new charge packet of corresponding size and injects this into another charge transfer channel. In both cases the extra amount of silicon area

and power required make such signal path crossings quite unattractive.

It has often been pointed out that VLSI chips will become ever more "wire limited". The active devices themselves get smaller and faster because of the scaling laws that apply to practically all MOS technologies. However, as the circuits scale down, all wiring will increase in resistance and will contribute in an ever increasing proportion to the overall delay in the system. In addition, unless the structure of the overall system layout is planned very carefully, the wiring of the system will use an ever larger fraction of the chip area. Thus one must give preference to those algorithms that use as few long distance interconnections and global signals as possible. This makes the one- and two-phase clock systems much more attractive.

Another serious limitation to the overall system complexity allowable on a single chip is power dissipation. At lower pulse frequencies the NMOS and PMOS circuits are dominated by static power dissipation. Equivalent circuits could be built with $I^2L$, CMOS or CTD technologies that consume 2 to 4 orders of magnitude less power. At frequencies above 10 MHz these differences are reduced to one or two orders of magnitude as shown in figure 31 [17].




IMPLEMENTATION OF A 16-BIT ADDER

| Technology | Estimated area ($mm^2$) | Power at 1MHz in watts | Power at 10MHz in watts |
|---|---|---|---|
| CCD | 6.19 | .0162 | .162 |
| PMOS | 11.3 | 2.9 | 4.6 |
| NMOS | 7.78 | .531 | 1.02 |
| CMOS | 16.5 | .582 | 1.8 |
| $I^2L$ | 14.9 | .029 | .596 |

Figure 31: Power Dissipation among Various Technologies

## 4.3. A Technology for VLSI FFT Processors

Early on in the program we studied the trade-offs between the various implementation technologies that could be used for the construction of the basic building blocks of a fast pipelined FFT VLSI processor. Dobrowolski [18]

compared different implementations of the important butterfly module in various MOS technologies. The technologies considered, NMOS, PMOS, CMOS, $I^2L$, and CCD, were compared in terms of active area and layout complexity as well as power dissipation and speed (through simulation). The key result was that the charge transfer approach did not look attractive at all for the implementation of the core logic modules in an FFT processor in which the data is represented in a parallel digital format. The signal flow graph of the FFT butterfly module or the CORDIC rotator module contains far too many topologically unavoidable signal path crossings. This would require that the signal representation constantly must switch from the charge domain to a voltage/current representation. It is then much more appropriate to implement the logic blocks using restoring logic with small charge steering networks of pass transistors interspersed, both of which can be fabricated using standard MOS technology.

Even for the implementation of serial memory charge transfer devices do not look very attractive anymore. For small blocks of memory the possible savings in area and power dissipation compared to almost any dynamic or static memory block are negligible since the overhead of the relatively complicated peripheral control circuitry dominates. In this case then one would prefer to use a type of memory that can be readily fabricated with the same technology as is used for the logic modules for easy integration of the whole system. If the memory block has to be fairly large, then power dissipation becomes a crucial issue. A purely serial memory would be unacceptable since the power would be proportional to the number of bits moved, rather than the number of bits stored. Since in a purely serial memory all bits move in every clock cycle, the power consumption can become prohibitive.

Most tricks that have permitted the charge coupled memories to reach rather high bit densities have now been adopted by the designers of the large dynamic RAMs as well, so that the density advantage of CCDs no longer outweighs the more difficult fabrication process.

Based on this comparative analysis we have decided to concentrate on the readily available NMOS technology for the implementation of the prototypes of the logic modules needed in a VLSI FFT processor.

## 5. DESCRIPTION OF INTEGRATED CIRCUITS

### 5.1. 16-Point DFT Processor

In [2], Despain describes a pipeline processor for computing a 16-point DFT. This processor, shown in figure 32, consists of four basic modules, one which computes the butterfly operation of the FFT, and three which perform various vector rotations.

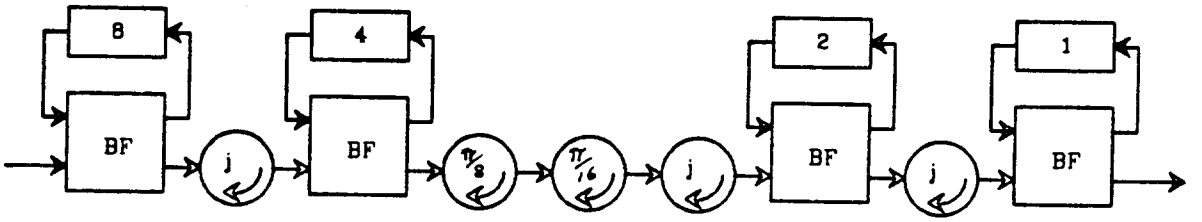



Figure 32: 16 point DFT processor

The $\pi/16$ and $\pi/8$ rotators work by rational approximation as described in [2]. Of these two, only one, the $\pi/16$, was actually implemented, although a more general vector rotator which is capable of rotating a vector by any angle was also designed, and could take the place of the collection of $\pi/n$ modules in a processor design. The more general vector rotator could also be used to build processors for computing transforms of much larger size as described earlier in this paper. The $\pi/2$ rotation, since trivial, was included on the butterfly module chip.

### 5.2. System-wide Considerations

#### 5.2.1. Bit Skewing

All data words (16 bit integers) in the processor are skewed bitwise so that the least significant bit of the word arrives at the chip one clock cycle before the next least significant bit and so on. This allows the carry from the addition of one pair of bits to propagate while the next most significant pair of bits is arriving. Thus, the irregularity of full carry-lookahead adders is avoided, and a chain of simple, one-bit carry-save adders can be used. The effect of this is to increase the throughput of the pipeline processor, since one can make the clock cycle equal to the time necessary to perform a one bit addition instead of a 16 bit addition. However, latency is increased for several reasons. First, there is the obvious first-order effect due to the fact that 16 clock cycles are required to input or output one datum. Clearly, this would be negligible in any signal-processing application. The more serious effect is due to the fact that shift and add operations, which comprise the whole of the CORDIC algorithm, introduce a latency equal in size to the magnitude of the shift. For example, if bit 3 of a data word is to be added to bit 7 of the same data word, bit 3 must be stored in a register until bit 7 arrives four clock cycles later. In an n-bit CORDIC vector rotator, the latency due to this effect would be

$$\textit{Shift latency} = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

where $n$ is the number of bits of accuracy of the CORDIC operation. The registers which are necessary for this intermediate storage also increase the area of the chip by a significant amount. In fact, in the 16-bit CORDIC rotator, these registers accounted for 60% of the active area of the datapath.

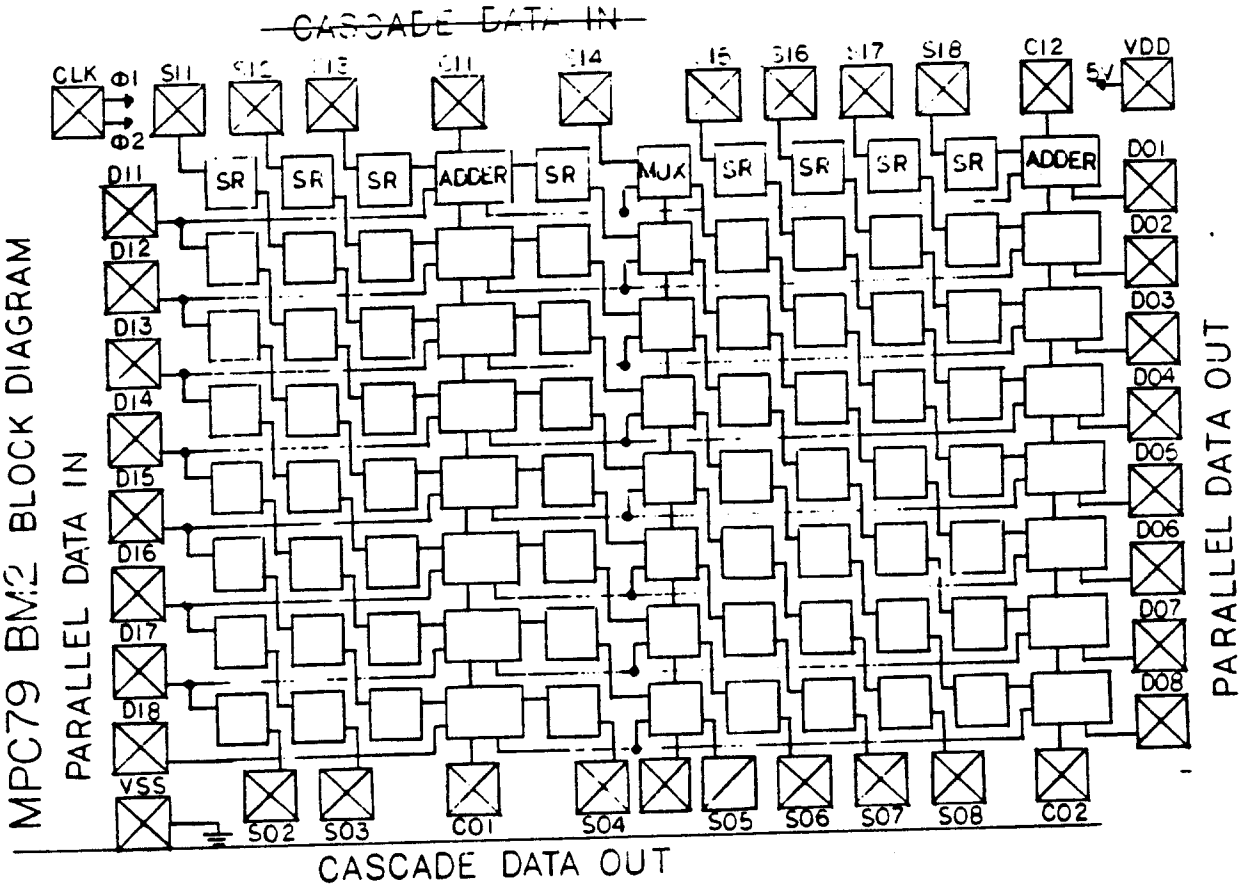



Figure 33: Block Diagram of Root 3 Circuit

If the high latency or the increased area of the bit-skewing technique where a problem, a tradeoff can be made by skewing the data words in blocks of $m$ bits, where $m<n$. This relieves both problems at the expense of reducing throughput, since now the basic clock cycle must be on the order of the time necessary to perform an $m$ bit addition, unless the adder itself is pipelined.

### 5.2.2. Multiplexing of the Real and Imaginary Parts

It was also decided at an early stage to multiplex the real and imaginary parts of the complex data vectors through the same pins. Although this reduces the throughput, it would have been impossible to have built the butterfly chip any other way, since the number of pins this chip uses is right at our current limit of 84. Also, it reduced the complexity of the crossover problem a great deal, especially in the CORDIC chip and the $\frac{\pi}{16}$ rotator, which otherwise would have had to have global chip communications at every stage of the algorithm. Rotation by $\frac{\pi}{2}$, used at the front end of the butterfly chip, became trivial, since it merely entailed the use of a buffer to reorder the real and imaginary parts of the data, whereas global communications would have been required if the data paths for the two parts had been separate.

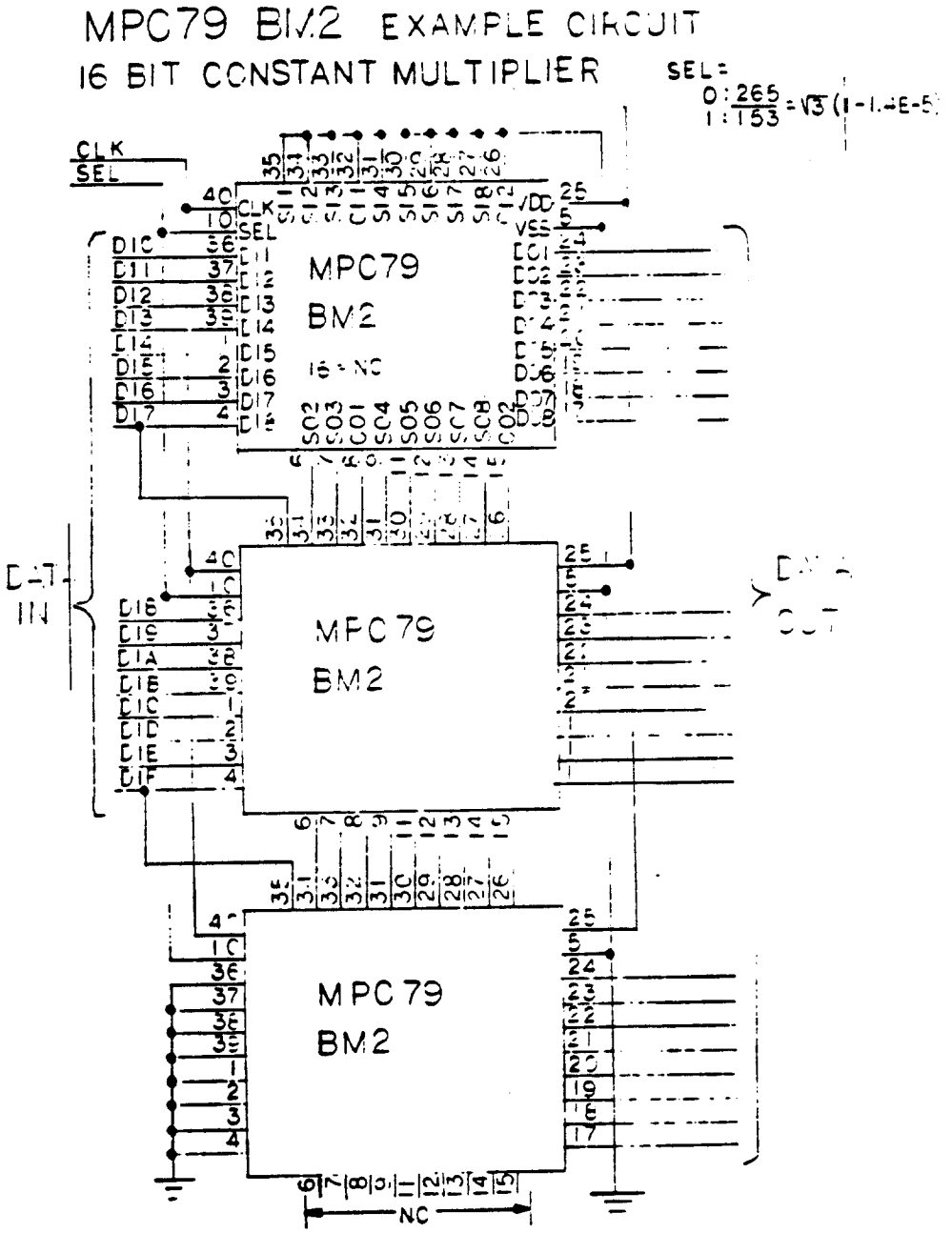



Figure 34: 16 bit Root 3 Circuit

### 5.3. Root 3 Circuit

In the prime factor algorithm of Good [5] large transforms are built up from smaller, relatively prime factors. The advantage of the technique is that no twiddle factors are necessary as in the Cooley-Tukey algorithm, although the complexity of data rearrangement is much higher. In [2] Despain suggested an algorithm for performing the base-3 DFT which could be used in conjunction with a radix-2 FFT processor handle transform sizes of the form $3\times2^n$ without the need for twiddle factors. One of the basic computations of the base-3 DFT is a multiplication by $\sqrt{3}$, which can be performed by the use of a rational approximation. If one is willing to accept an arbitrary gain factor in the result of the DFT, one can then multiply the entire DFT equation by the denominator of the

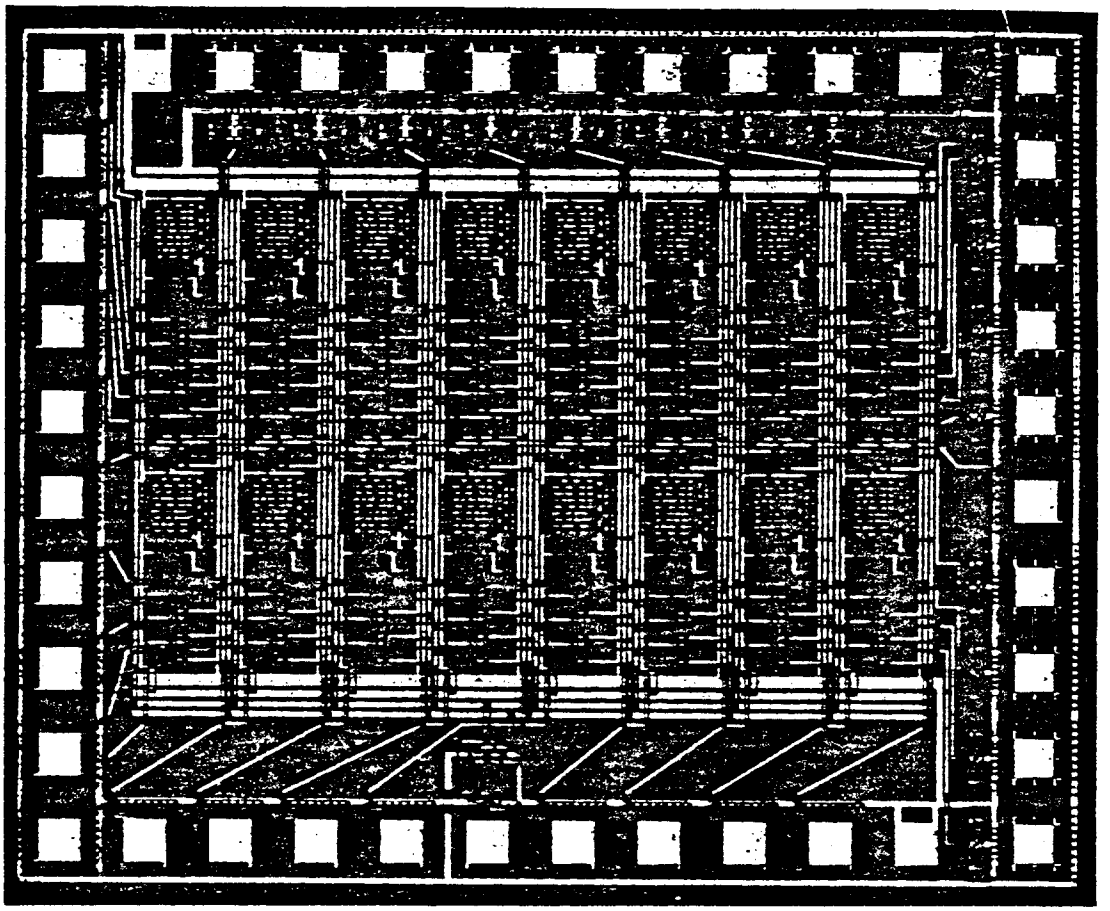


Figure 35: Fabricated Root 3 Chip

rational approximation, thus limiting the necessary computations to constant real multiplies. For sixteen-bit accuracy, a good approximation is given by $\frac{265}{153}$. The use of this approximation also minimizes the number of shifts and adds necessary to perform the operation, since a multiplication by

$$265 = (2^5+1)2^3+1$$

requires 2 shifts and 2 adds and a multiplication by

$$153 = (2^4+1)2^3+(2^4+1)$$

also requires 2 shifts and 2 adds. In addition, it is easy to build hardware capable of performing both multiplications.

Figure 33 is a detailed block diagram of the chip as it was actually implemented. The blocks marked SR shift a datum right one bit, while the blocks marked ADDER are one bit adders. Due to area limitations, the chips was realized as a bit-slice, requiring three chips for the full 16-bits of precision as shown in figure 34. The input data are applied on pins DI1-DI8 and the output data are received on pins DO1-DO8. The outputs SO1-SO8 and CO1-CO2 would be connected directly from the first (second) chip in the cascade to the inputs SI1-SI8 and CI1-CI2 on the second (third) chip. When the input SEL is high, the chip multiplies by 265, and when SEL is low, the chip multiplies the input data by 153. Unfortunately, since the chip was designed before the bit-skewed data format

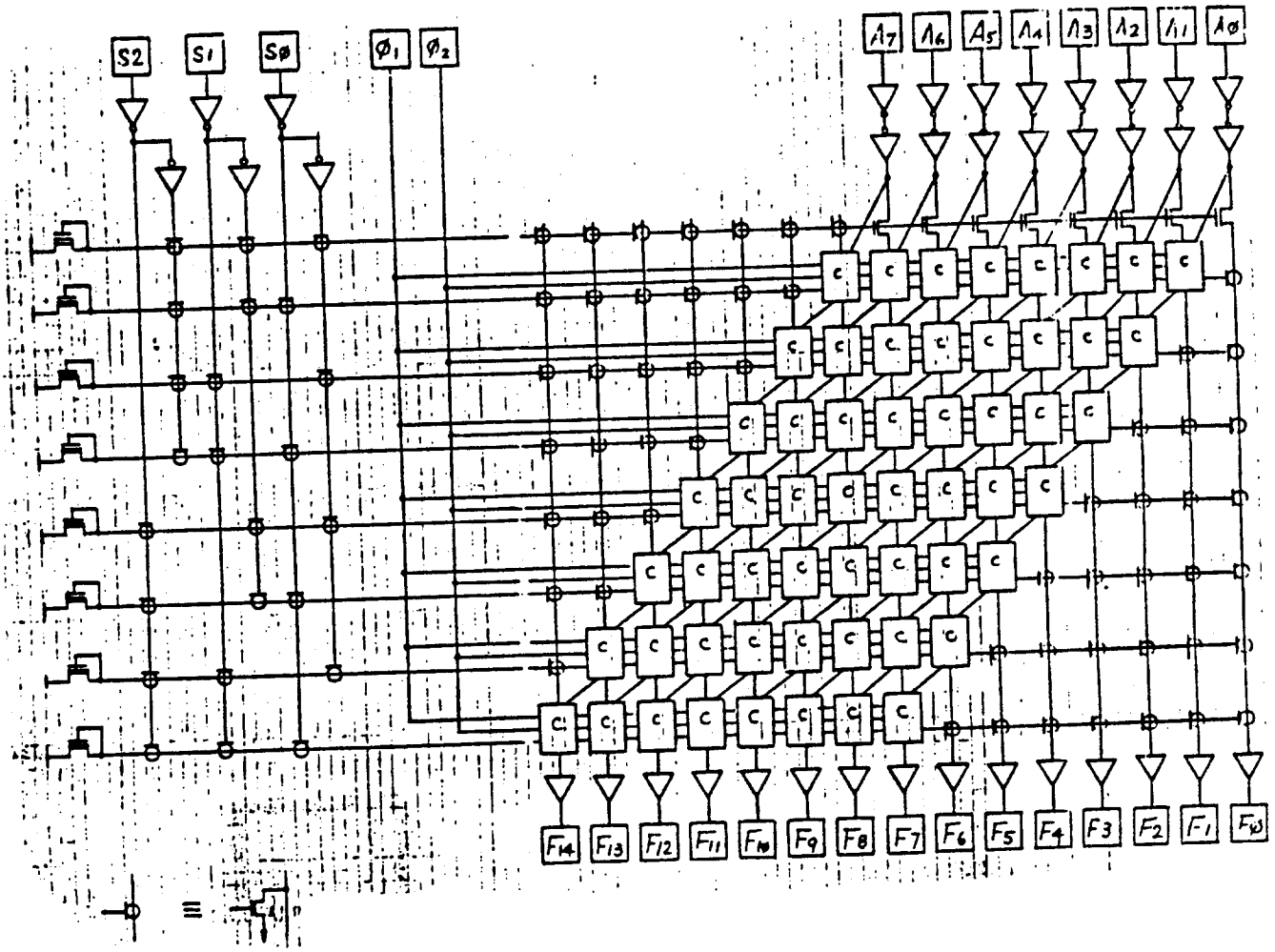



Figure 36: Schematic of Barrel Shifter

was decided upon, it is incompatible with the later chips which used that format.

The fabricated chip, shown in figure 35, was tested and found to be operational up to 6 MHz, which was the limit of our test equipment at the time. The power consumption was measured to be 32 ma quiescent and 60 ma at 6 MHz.

### 5.4. Barrel Shifter

The use of the CORDIC algorithm for vector rotation in the computation of the DFT has already been discussed. Hardware capable of performing this algorithm has as one of its basic building blocks a suitable shift network. A preliminary study of a programmable barrel shifter capable of left shifts of arbitrary size was done and an 8-bit version was designed and fabricated. A schematic of this circuit is shown in figure 36. The chip has 8 data inputs, 3 control inputs which specify the number of bits by which the data word is to be shifted, and 15 data outputs. The input data enter the chip on the lines marked A0-A7 in figure 1a and pass into the array of "C" cells seen on the right side. Each of these cells is capable of sending a datum straight through or shifting it to the left depending on the state of the S0-S2 control signals.

The finished chip (shown in figure 37 ) was tested and found to be operational at clock rates up to 10.4 MHz.

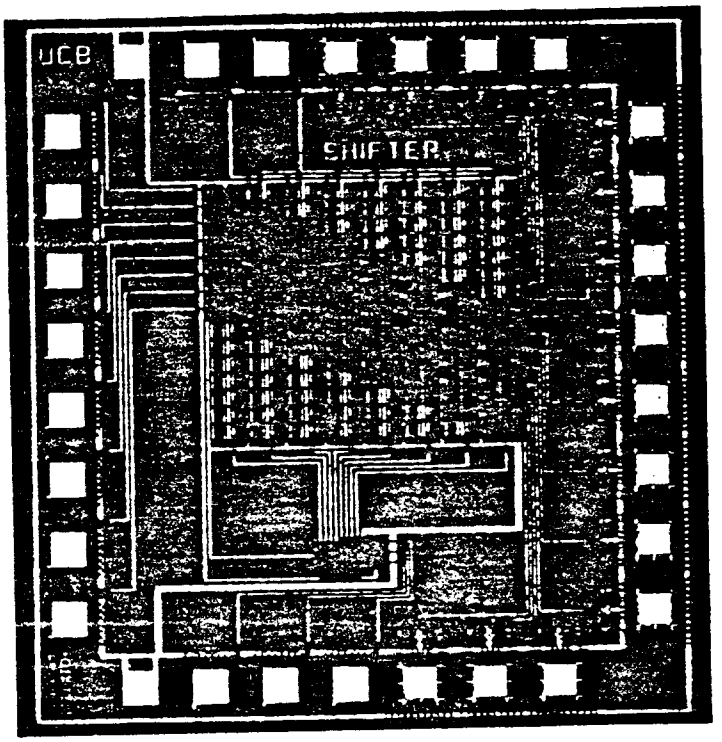



Figure 37: Fabricated Barrel Shifter Chip

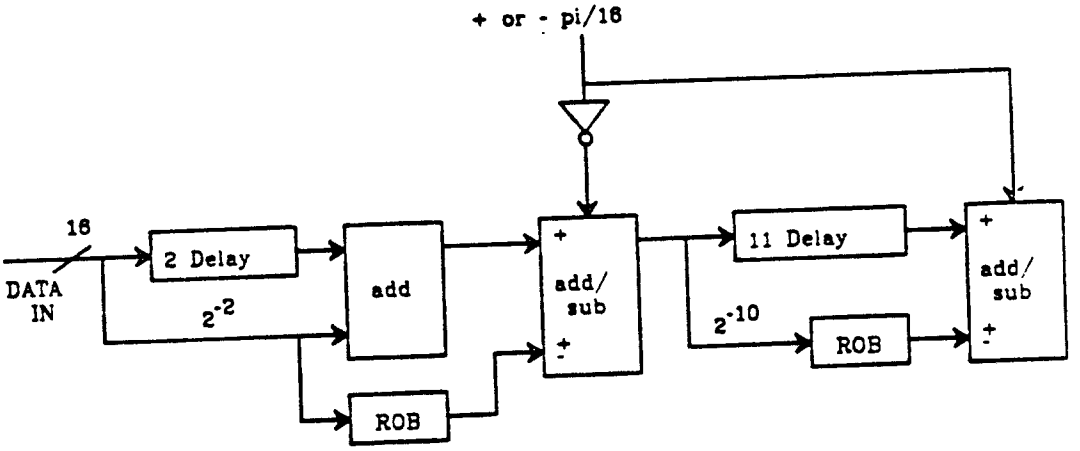



Figure 38: Block Diagram of $\frac{\pi}{16}$ Circuit

In the final version of the CORDIC rotator which is discussed below, the shifts were hardwired rather than handled by programmable shifters at each stage. However, a need for a programmable shifter would arise in a lower performance iterative CORDIC unit which used the same hardware to process all the stages of the CORDIC algorithm.

## 5.5. $\frac{\pi}{16}$ rotator

### 5.5.1. Theory of operation

In [2] Despain discusses the use of rational approximations for rotations. In particular, algorithms for $\frac{\pi}{16}$ and $\frac{\pi}{8}$ rotations are developed which are optimum in the sense that they reduce the number of additions necessary to achieve the accuracy desired. The algorithm which was implemented was for a

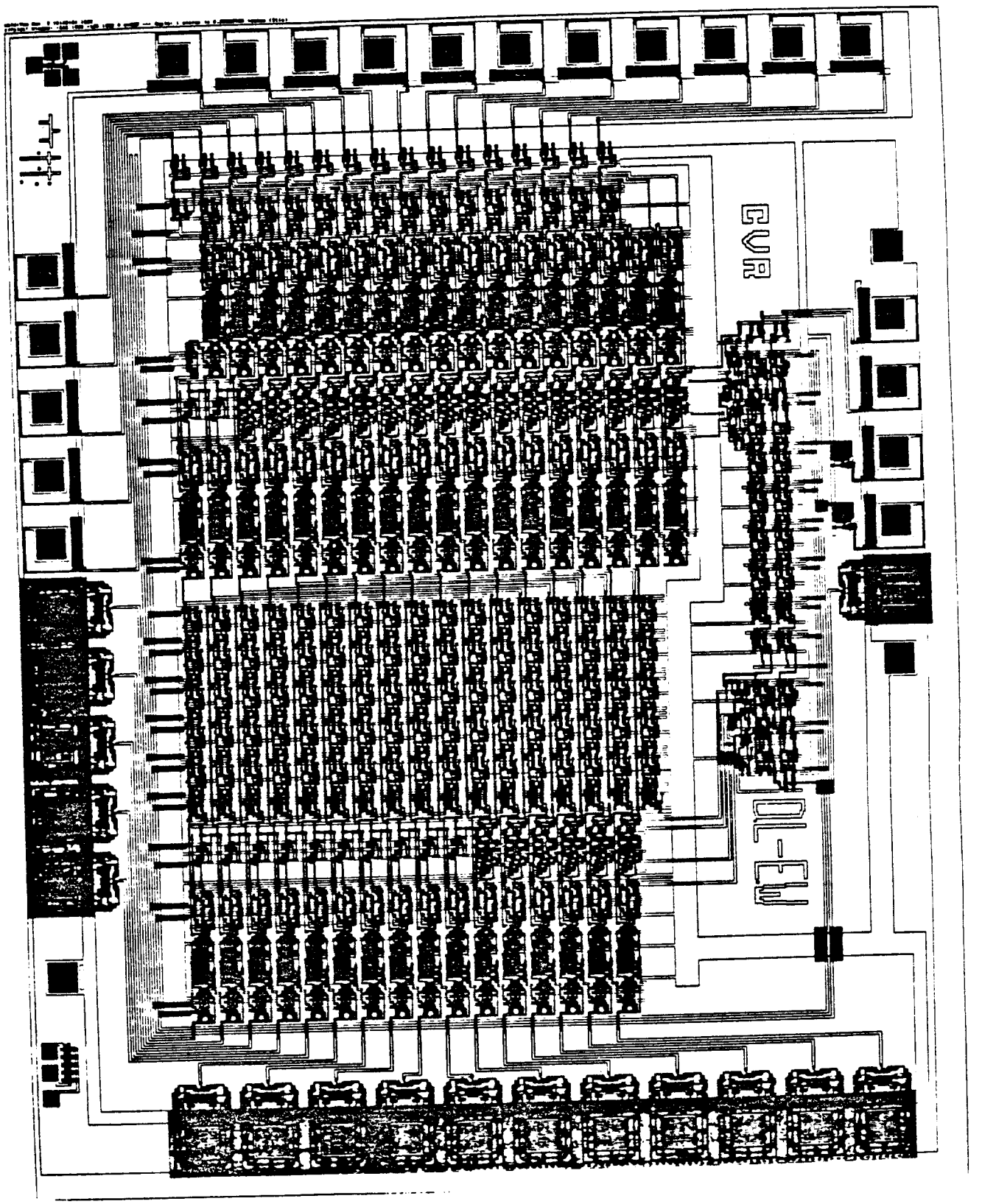


Figure 39: CIFPLOT of $\frac{\pi}{16}$ Chip

$\frac{\pi}{16}$ rotation and can be written as

$$r_1 = (2^{-2}+1)r_0$$
$$i_1 = (2^{-2}+1)i_0$$
$$r_2 = r_1 \mp 2^{-2} i_1$$
$$i_2 = i_1 \pm 2^{-2} r_1$$

followed by

$$r_3 = r_2 \pm 2^{-10} i_2$$
$$i_3 = i_2 \mp 2^{-10} r_2$$

where $r_0$ and $i_0$ are the real and imaginary parts, respectively, of the vector to be rotated. This algorithm is similar to the full CORDIC algorithm in that it consists only of shifts and adds.

Figure 38 and figure 39 show a block diagram and CIFPLOT of the completed circuit. The layout and function of the chip is very similar to that of the CORDIC rotator, and thus will not be explained in great detail. In fact, the major difference is that this algorithm consists of only three stages, whereas the full CORDIC algorithm requires sixteen for the same accuracy. This forced a different aspect ratio on the basic cells to avoid a tall and narrow chip.

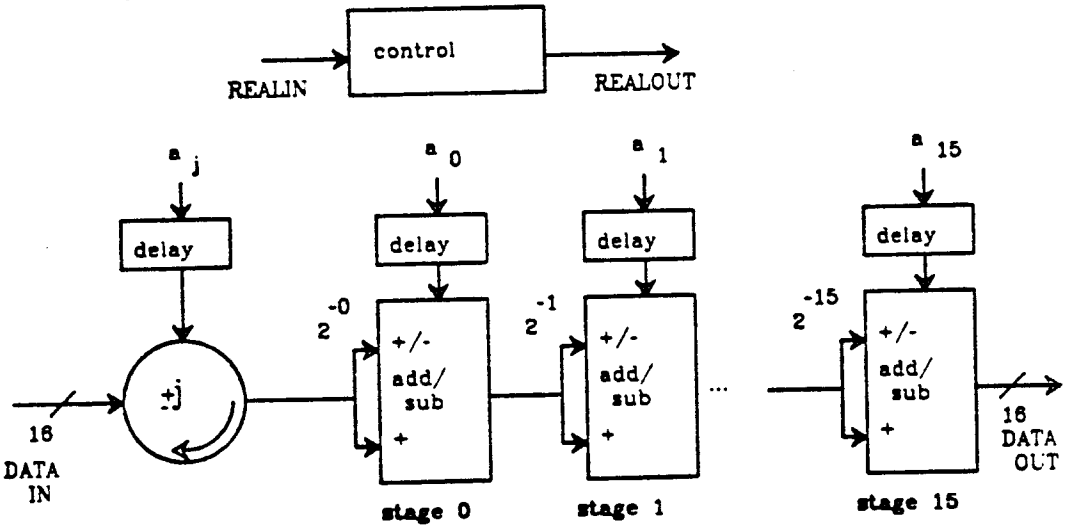



Figure 40: Block Diagram of CORDIC rotator

## 5.6. CORDIC Rotator Chip

### 5.6.1. Theory of Operation

The chip was specified to work with 16 bit two's complement data words which set the number of iterations to 17. Since, in a Fourier Transform processor, the rotation angles are known, we have assumed that the $\alpha_k$ have been previously computed and are delivered to the CORDIC rotator chip by the control circuitry (probably a ROM). Given a complex input vector $r_0+i_0 j$, the algorithm can now be expressed as

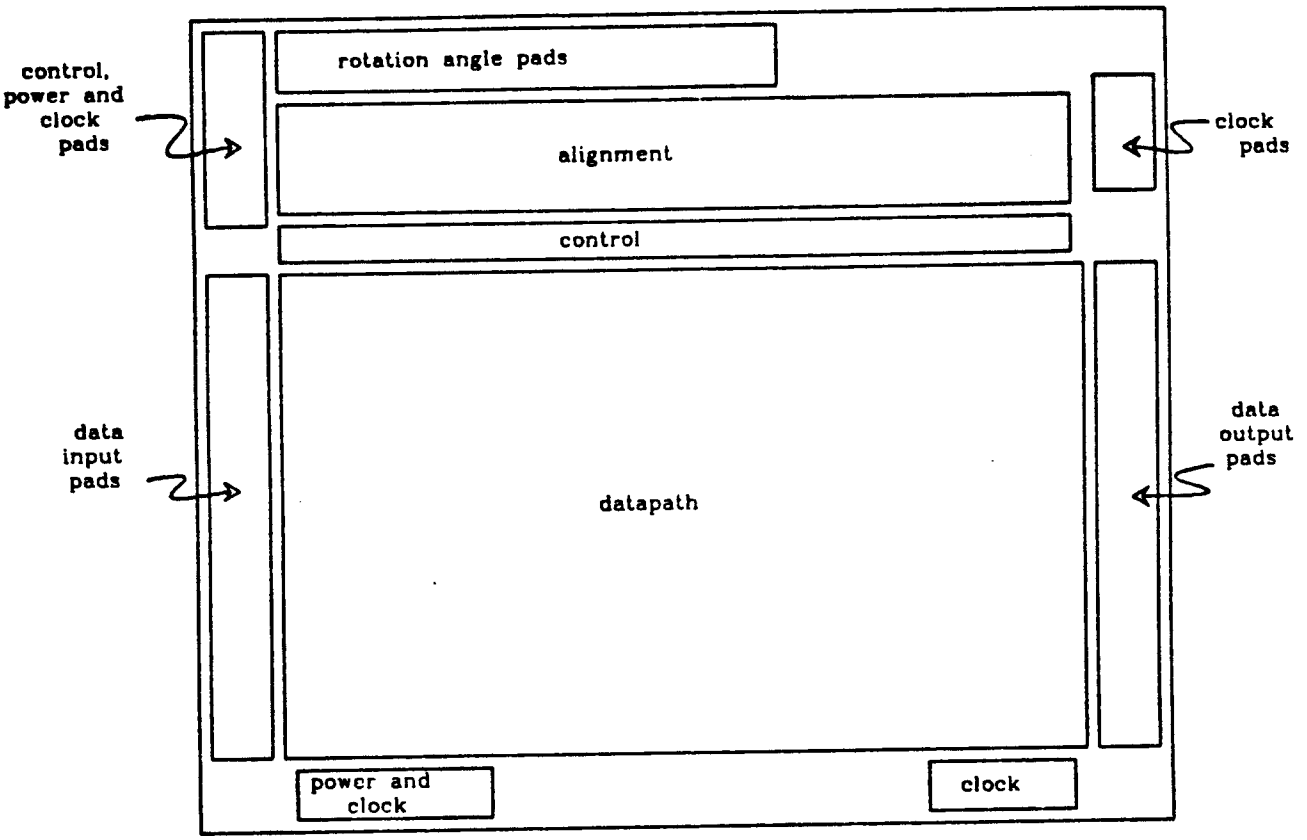



Figure 41: Floor Plan of CORDIC rotator

IF $a_0$=0 THEN DO

$$r_1 \leftarrow -i_0$$
$$i_1 \leftarrow r_0$$

ELSE IF $a_0$=1 DO

$$r_1 \leftarrow i_0$$
$$i_1 \leftarrow -r_0$$

FOR $k \leftarrow 1,16$ DO
IF $a_k$=0 THEN DO

$$r_{k+1} \leftarrow r_k + i_k 2^{-k+1} \qquad (3)$$
$$i_{k+1} \leftarrow i_k - r_k 2^{-k+1}$$

ELSE IF $a_k$=1 THEN DO

$$r_{k+1} \leftarrow r_k - i_k 2^{-k+1}$$
$$i_{k+1} \leftarrow i_k + r_k 2^{-k+1}$$

where the $a_k$ are externally supplied control signals which determine the order of addition and subtraction at each stage. The $k$=0 stage is a $\pm\frac{\pi}{2}$ rotation which is necessary if one wishes to rotate by angles from $+\pi$ to $-\pi$.

Since the chip was specified to work in a pipeline DFT processor, it was implemented as a pipeline, with each iteration being handled by a separate

piece of hardware. This allows the multiplications by $2^k$ to be performed by hardwired shifts between each iteration stage. Also, due to the bit skewing throughout the DFT processor, we have not used full carry-lookahead adders, but have utilized one bit carry-save adders which allow the carries to propagate before they are needed. The operation of a set of these carry-save adders will be explained in detail below. This turned out to have an advantage in that the regularity of the entire chip was greatly increased, reducing the design time considerably.

Looking at the block diagram in figure 40 and the floor plan in figure 41, one can see that the data comes in on the left and flows through the 17 iterations of the CORDIC algorithm. The two-phase clocks used by the chip are assumed to be generated and driven by circuitry off chip (since they run through the entire DFT processor). On the chip, the clocks run vertically across the entire circuit in metal along with power and ground. At the top of the block diagram are the $a_k$ control signals. As one can see from the algorithm, the $a_k$ determine whether the shifted half of the vector is added to or subtracted from the other half at each stage. Since, for any given input vector, all 17 of the $a_k$ are input at the same time, each $a_k$ must be delayed so that it will reach the stage it is to control at the same time as the data. These delays are accomplished by entering each $a_k$ into simple shift registers (pass transistors and inverters) of the proper length. The reordering buffers and the reordering buffer control are discussed below.

### 5.6.2. Detailed Description of Data Path and Control

We will now look closely at the computation of one iteration of the algorithm (stage 5) as shown in figure 42. The notation $A_k(b)$ signifies an adder/subtractor in stage $k$ which handles bit $b$ of the data word. This module is shown in detail in figure 43. Note that $b$ takes values from -4 to 15, since we keep 4 guard bits in the partial results as recommended by Walther [19]. The input to the adder marked ± will be added to or subtracted from the input marked + depending on the value of the control input $p_k(b)$, where $k$ and $b$ again denote the stage and bit, respectively. Since we are working in two's complement, subtraction is performed by complementing the ± input and holding the carry-in, $c_k(b)$, high. This is realized by connecting $c_5(-4)$ to $p_5(-4)$, since a high value on $p_k(b)$ signifies subtraction. Note that the signal $p_5(-4)$ is merely the $a_5$ of the algorithm which has been delayed as mentioned in the previous section. In addition to the sum output which appears on the right, each adder also produces two control signals which are passed on to the next adder in the chain. One of these is the $p_k(b)$ input delayed by one clock cycle, and the other is the carry-out resulting from the addition. We will also use the notation $r_k(b)$ and $i_k(b)$ to denote bit $b$ of the real and imaginary parts of the data, respectively, at stage $k$.

The operation of this stage is as follows. During phi2, $A_4(-4)$ produces the output $r_5(-4)$, which is entered into the shift register $5D_5(-4)$ during phi1. At the next phi2, the same adder will produce $i_5(-4)$, which will go into the same shift register. By the algorithm in (3), we can derive the following:

$$r_6(-4) = r_5(-4) \pm i_5(0) \qquad (4)$$
$$i_6(-4) = i_5(-4) \mp r_5(0)$$

which correspond to a shift of $2^{-4}$ and an add/subtract operation. Since the bits of the data flowing through the chip are skewed so that the higher order bits trail the lower order bits, the outputs $r_5(0)$ and $i_5(0)$ from $A_4(0)$ will be computed 4 clock cycles after the outputs of $A_4(-4)$ in (4). Thus, the outputs of

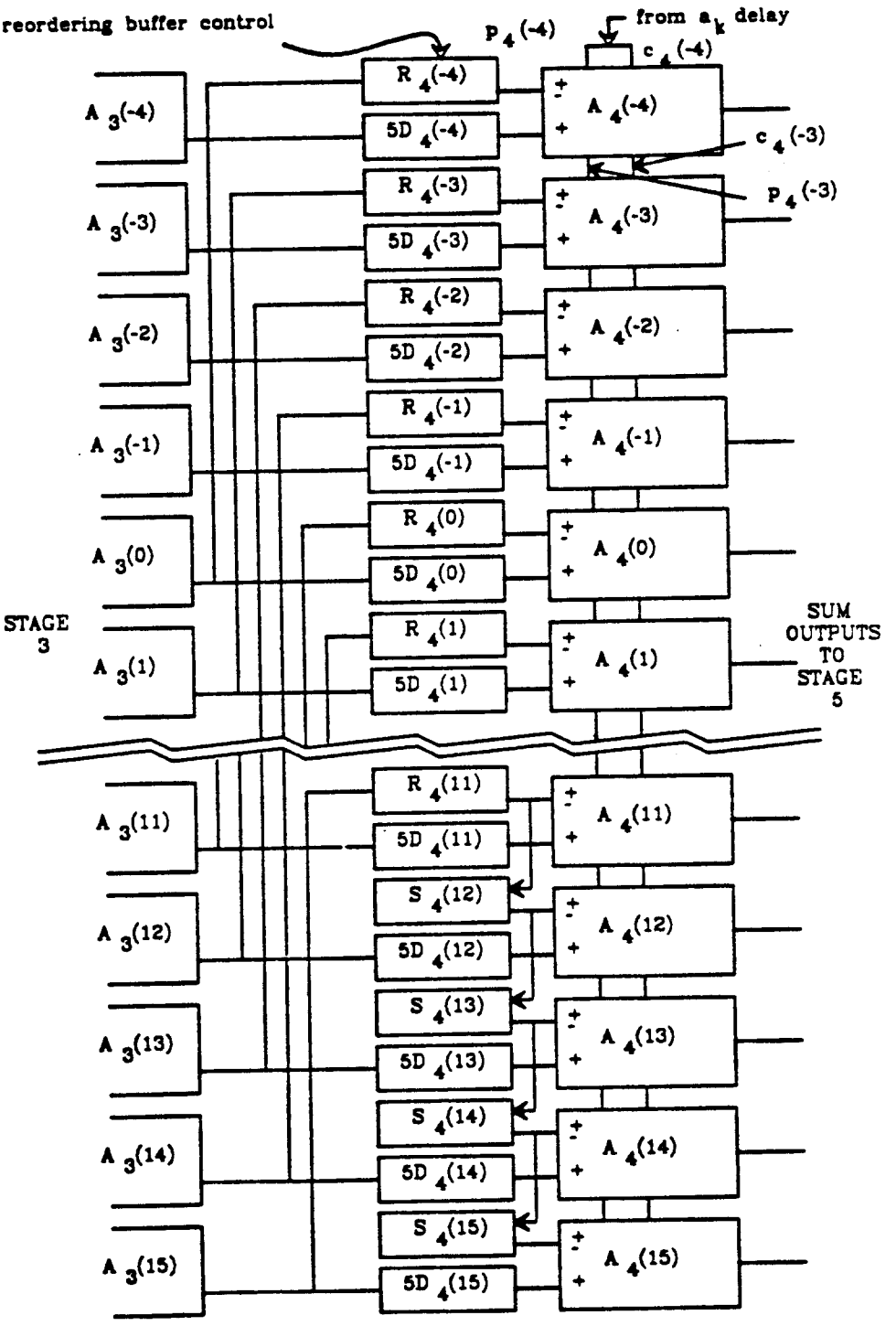



Figure 42: Stage 5 of the CORDIC Rotator

$A_4(-4)$ need to be delayed for at least 4 clock cycles to allow the outputs of $A_4(0)$ to catch up. However, note that the outputs of $A_4(0)$ are not in the correct order for (4) to be computed. This necessitates the use of a reordering buffer (denoted by $R_5(-4)$), which inputs $\tau_4(0)$ on phi1, stores it in a shift register, allows $i_4(0)$ to pass undelayed on the next phi1, then allows $\tau_4(0)$ to be input to $A_5(-\tau)$ during the following phi1. The control of the reordering buffer will be discussed later. Note that the added delay in this module requires that the outputs

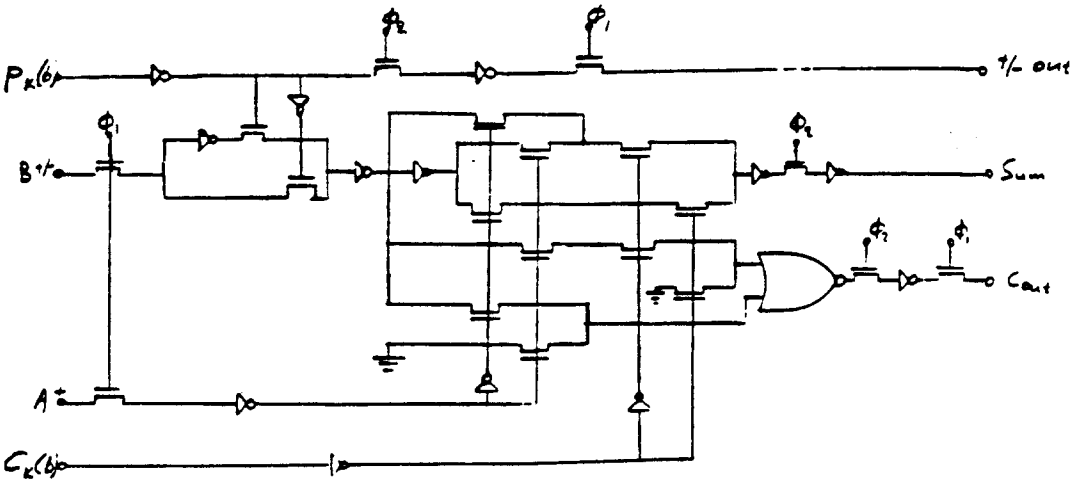


Figure 43: Adder Module

of $A_4(-4)$ be delayed by 5 clock cycles altogether before being entered at the next phi1 into $A_5(-4)$. An identical sequence of events occurs for the two data inputs to $A_5(-3)$, except that the entire sequence occurs one clock cycle later. In addition, $A_5(-4)$ holds onto its carry-out until phi1, when it is passed to $A_5(-3)$ along with the delayed $p_5(-4)$ control signal. At the bottom of the figure, the modules labeled $S_k(b)$ perform the sign-extension necessary for two's complement arithmetic by delaying (by shift register) the output of $R_5(11)$ by one clock cycle each. Since the output of this reordering buffer is the sign bit of the previous stage's output, this is the correct operation.

In general, stage $k$ in the iteration consists of a shift by $2^{-k+1}$, $20-k+1$ reordering buffers, a set of delays of length $k$, $k-1$ sign-extension delays, and a set of 20 adder/subtractor modules.

The control circuitry for the chip consists of the delays for the $a_k$ (described earlier) and the control circuit for the reordering buffers. The chip requires that the user supply a high level to the input REALIN whenever the least significant bit of the input data is from the real part of an input vector, and a low level whenever it is from the imaginary part of an input vector. This signal has two functions. First, it inverts the $a_k$ input whenever an imaginary number is being input, thus deriving the two $p_k(b)$ signals necessary for the addition and subtraction in each iteration of the algorithm. The circuit which performs this inversion is shown in figure 44.

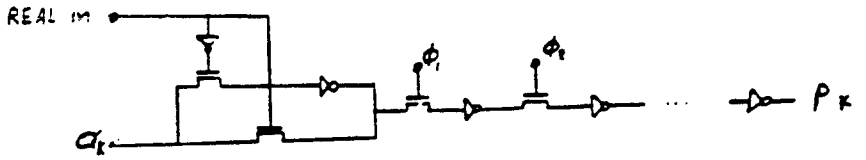


Figure 44: Inversion of $a_k$

Second, it is entered into a shift register from which the control signals to the reordering buffers are tapped as shown in figure 45. When the appropriate stage of the shift register contains a high level, the signal $r_1$ is produced which causes the reordering buffer to store its present input as the real part of the data vector. When this level is passed to the next stage of the shift register, producing the signal $im$, the data at the input of the reordering buffer must be the imaginary part of the data vector, and is thus allowed through without delay.

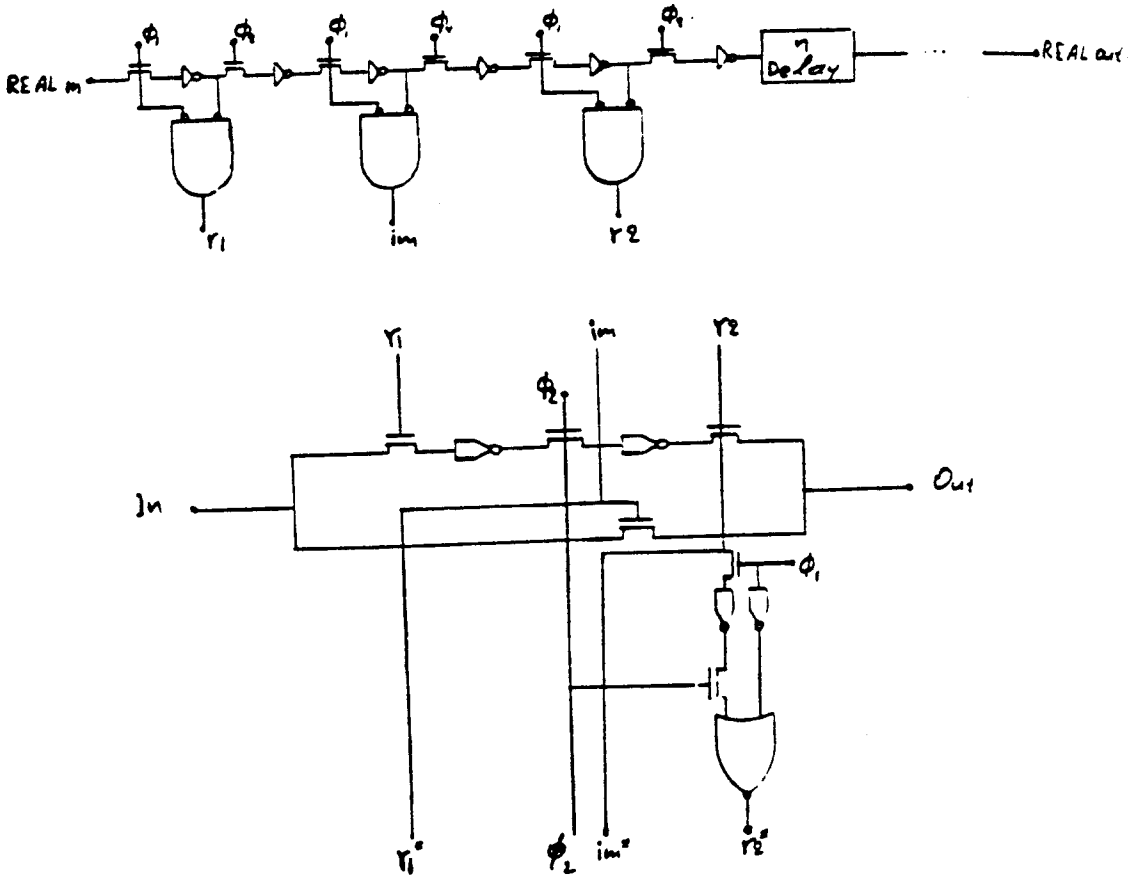



Figure 45: ROB and ROB Control Signals

Similarly, the signal $r_2$ causes the previously stored real part to be sent to the adder as explained above. When the REALIN signal has propagated the length of the shift register, it is brought off the chip as REALOUT to signal the fact that the least significant bit of the output data is real. This allows the chips in the DFT processor, which all use the same data format, to be cascaded with no extra circuitry.

A CIFPLOT of the finished chip is shown in figure 46.

### 5.6.3. Performance Estimation

Since the chip is designed as a pipeline, the performance is limited by the longest delay between any two dynamic registers. The longest delay in the chip occurs in the adder module, which contains the only substantial combinatorial logic on the chip. A SPICE run on this circuit reported a worst case delay from input to carry-out of approximately 40ns. The only significant wiring which might affect the circuit's performance exists between the output of the $a_k$ delays and the adders, but this was hand estimated to be less than 40ns, and so would not reduce the clock rate further. Thus, the overall maximum clock rate should be expected to be 10 to 12 MHZ.

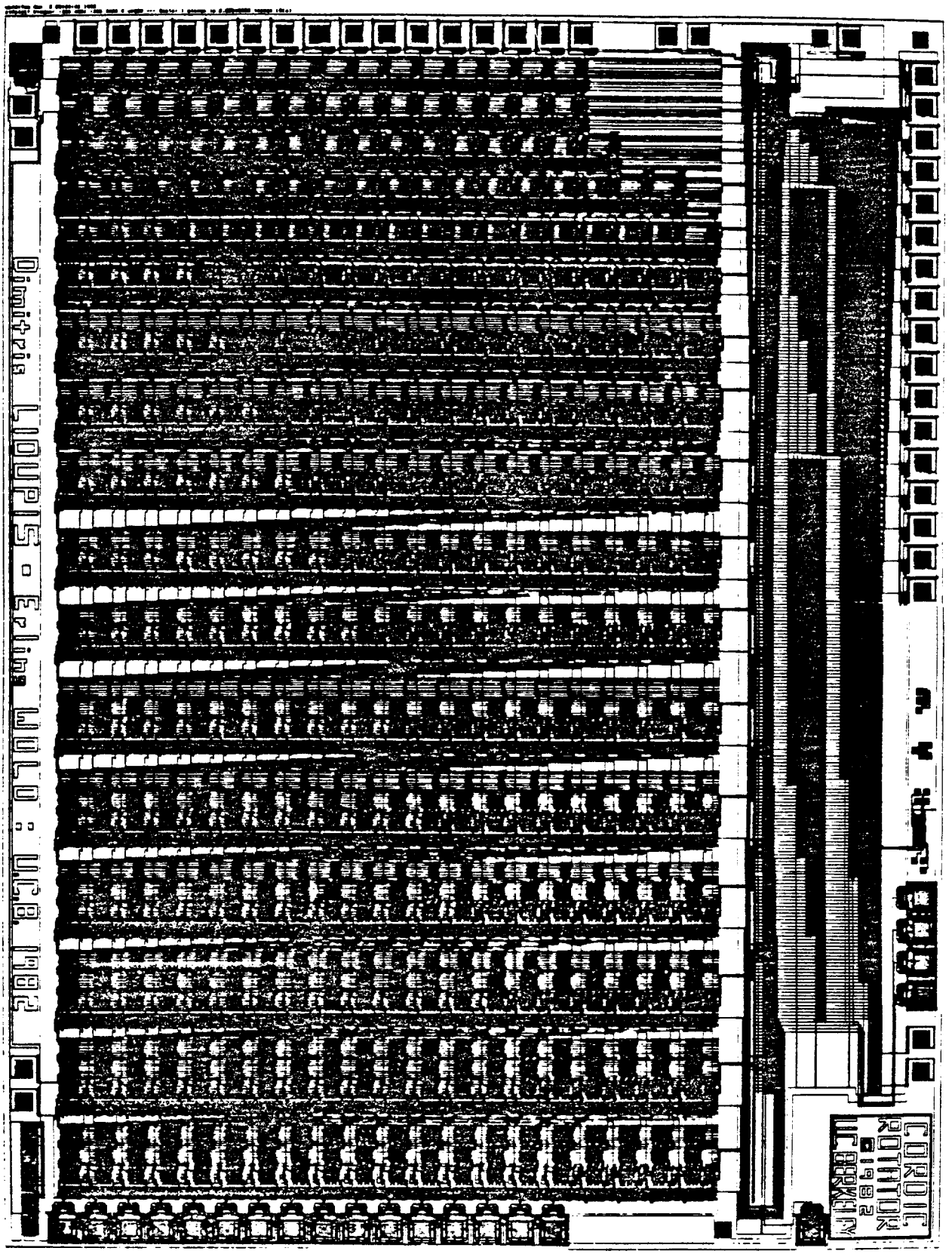



Figure 46: CIFPLOT of CORDIC Rotator Chip

### 5.6.4. Testing

Due to area limitations, there are no registers between the stages of the pipe and thus it is impossible to look at intermediate results of the CORDIC calculation on a fabricated chip. However, it would be possible to add a register at the output of each adder module and string these together if one is willing to give up area to do it. One could turn off phi2 at the output of the adder and enable a pass transistor which would allow the sum output to be chained to the next adder using a shift register stage. Clearly, this would require at least two pass transistors and two inverters for each signal one wished to chain together in this way, along with at least one more control line running down the adder chain. Since this would easily take the chip past 10mm in length, it was not implemented.

## 5.7. Butterfly Circuits

### 5.7.1. Introduction

The butterfly computation takes two complex inputs $A_k$ and $B_k$ and computes the outputs $C_k$ and $D_k$ according to the rule

$$C_k = A_k + B_k \qquad (1)$$
$$D_k = A_k - B_k$$

where $A_k$, $B_k$, $C_k$, and $D_k$ are complex.

A preliminary study of various technologies for implementing the butterfly chip was undertaken early on in the research. A 4-bit-slice NMOS chip was built at that time to test the speed of the adder that had been designed while verifying simulation results. This chip did not use the multiplexed real and imaginary format and is thus incompatible with the CORDIC rotator without the use of extensive support circuitry. However, another NMOS chip was later designed which did use this format and could be used in conjunction with the CORDIC rotator to build FFT processors of arbitrary size.

Following the structure of the DFT processor in [2], both chips were designed to operate in the following way. First, the input data ($A_k$ in (1)) is passed into a shift register until it is full. At this point, the chip is switched into add mode and the data stored in the shift register is combined with the incoming data ($B_k$) to form the sum and differences according to (1). The sum output ($C_k$) is sent immediately through the output pins to the next stage of the processor, while the difference outputs ($D_k$) are stored in the vacant stages of the shift register. When the $C_k$ have all been passed out, the shift register is then emptied through the output pins by switching the chip back to pass mode.

### 5.7.2. Preliminary Butterfly Processor

The preliminary butterfly chip consisted of a set of four 1-bit adder/subtractor units which could be used to build up arbitrarily large butterfly processors. A block diagram of a 16-bit butterfly processor built from these modules is seen in figure 47 along with a single module whose inputs and outputs are shown. A and B are the data inputs, C and D are the data outputs, Ci, Co, Bi, Bo the carry and borrow inputs and outputs from the other chips in the cascade, and Mi and Mo the mode input and output which determine whether the circuit is in pass or add mode as described above. The circuit itself was implemented with a NOR PLA, since it is a regular structure which is can be made fairly compact and fast.

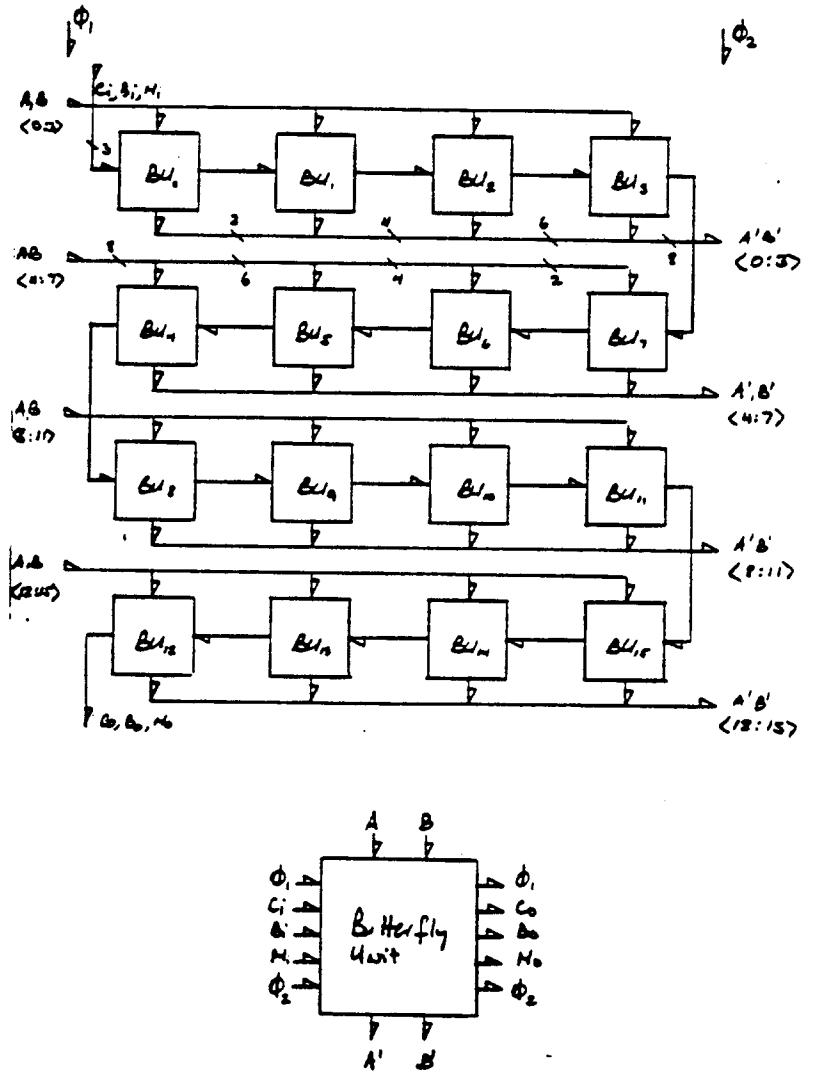



Figure 47: 16 Bit Butterfly Processor

The resulting 1-bit module, which can be seen replicated four times in the finished chip (figure 48) measured 420 by 657 $\mu$ with $\lambda = 2.5\mu$. A ring oscillator circuit was set up to measure the PLA delay, yielding a result of 120ns. Note that this is quite a bit slower than the circuit which was later devised for the CORDIC rotator. Power consumption was measured to be 15 mw per PLA with Vdd = 5v.

### 5.7.3. Compatible Butterfly Processor

For convenience, it was decided to include a $\pi/2$ rotator at the front end of this chip which is realized by a circuit which performs

$$Re(B_k) = \pm Im(B_k) \qquad (2)$$
$$Im(B_k) = \mp Re(B_k) .$$

Also, enough shift register memory was included on chip so that a 16-point DFT processor could be built without the need for any external memory. This memory can be bypassed for larger transforms.

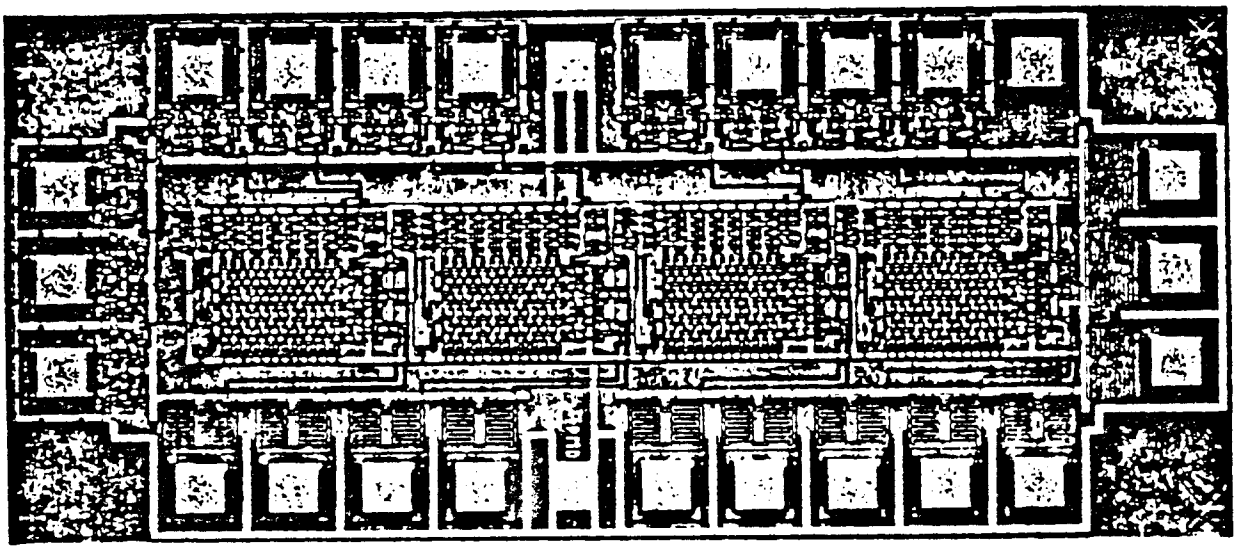


Figure 48: Fabricated 4 Bit Butterfly Chip

Referring to the floorplan in figure 49

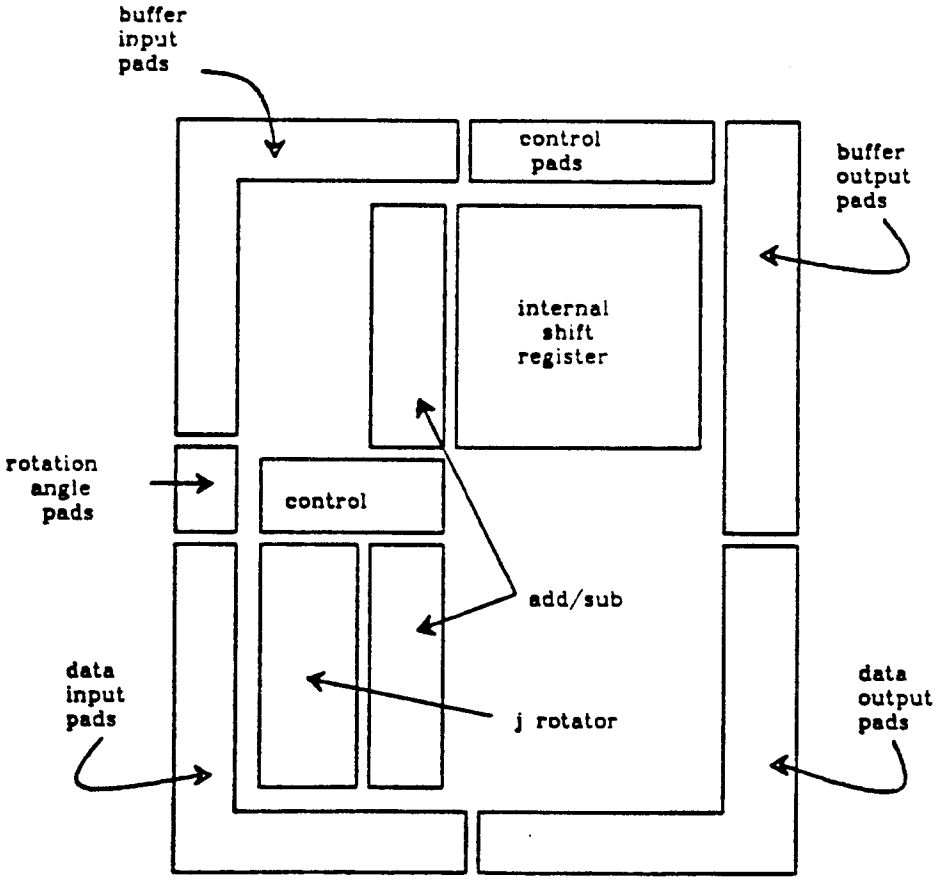



Figure 49: Floorplan of Butterfly Module

and the schematic in figure 50, the chip is utilized as follows.

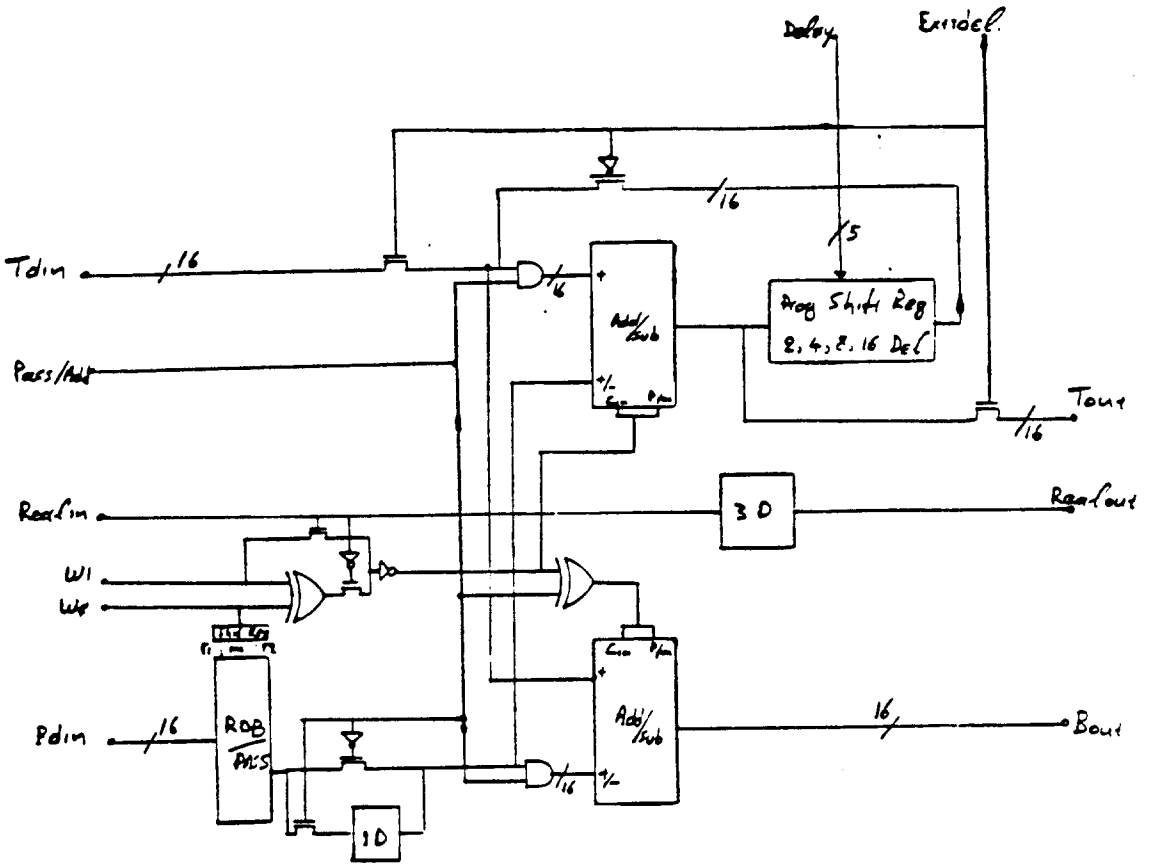



Figure 50: Schematic of Butterfly Module

1. An internal or external shift register is chosen through the use of the input "extrdel". When "extrdel" is high then the internal shift register is disabled and an external register can be connected to the pins Tout and Tin.
2. If the internal shift register is chosen, the length must be set through the use of inputs del0 through del4. Table 3 shows the settings required for the length of shift register desired.

Table 3: Choosing Shift Register Length

| Extrdel | Del0 | Del1 | Del2 | Del3 | Del4 | DELAY |
|---|---|---|---|---|---|---|
| 1 | x | x | x | x | x | external delay |
| 0 | 1 | 1 | 1 | 1 | 1 | 2 D |
| 0 | 0 | 0 | 1 | 1 | 1 | 4 D |
| 0 | 0 | 0 | 0 | 0 | 1 | 8 D |
| 0 | 0 | 0 | 0 | 0 | 0 | 16 D |

3. The input "pass/add" is set high (pass).
4. The $A_k$ are entered sequentially, real part first, then imaginary part. The input "realin" must be high when entering the least significant bit of the real part and low when entering the least significant bit of the imaginary part.

5. When the shift register is full, the input "pass/add" is set low and the $B_k$ are entered in the same way as the $A_k$ except that one must also set the inputs "w0" and "w1" to determine the rotation angle desired. These inputs must be stable during the input of least significant bits of both the real and imaginary parts of $B_k$. The relationship of these inputs to the rotation angle is shown in table 4.

| Table 4: Choosing rotation angle | | |
|---|---|---|
| w1 | w0 | angle |
| 0 | 0 | 0 |
| 0 | 1 | $\pi/2$ |
| 1 | 0 | $\pi$ |
| 1 | 1 | $-\pi/2$ |

The input "realin" must still be sequenced correctly.

6. After all the $B_k$ have been entered, the "pass/add" input is set high and the $D_k$ are output. The output "realout", when high, signals the real part of an output datum. The next set of $A_k$ can be entered simultaneously with this, and steps 5-6 can be repeated indefinitely.

The $\pi/2$ rotator is implemented as recommended by Despain in [2]. If the $B_k$ are to be rotated by $\pm\pi/2$ (determined by "w0"), the real and imaginary parts are interchanged in the reordering buffer (ROB), as is required by (2). The change of sign in (2) is accomplished by merely inverting the plus or minus control signal to the adder/subtractors. If the $B_k$ are to rotated by $\pi$, the plus or minus control signals are inverted without interchanging the real and imaginary parts. If the $B_k$ are to rotated by an angle of 0, nothing is done.

### 5.7.4. Chip Description

The size of the chip is 3.03x2.70 mm square which was determined by the large number of pads necessary for I/O.

The adder/subtractor circuit uses a MUX to generate the sum and carry outputs, which makes its operation relatively slow. The circuit configuration of the programmable delay is shown in figure 51.

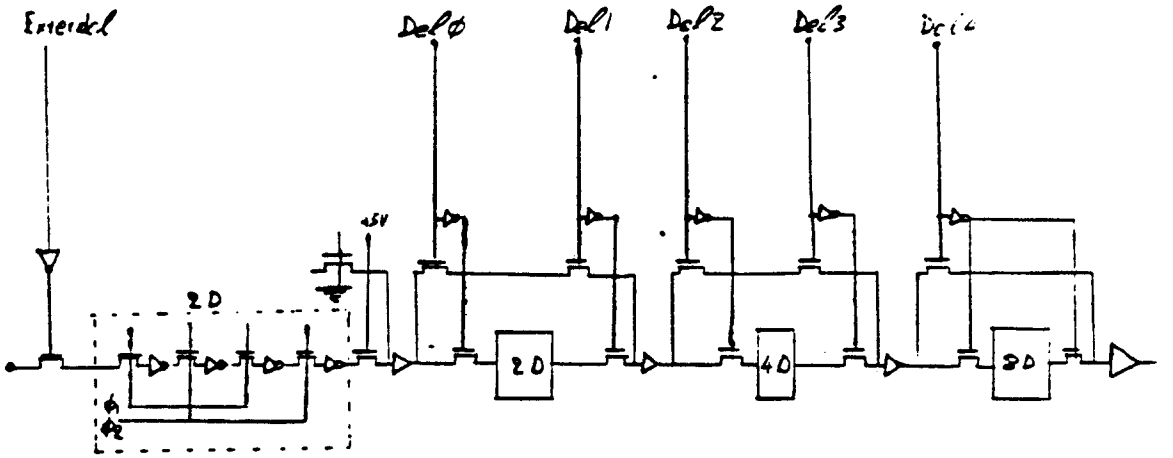



Figure 51: Programmable Delay Circuit

Superbuffers are used in between the delay stages so that with any number of delays programmed the feedback loop is driven quickly.

The modules have been topologically arranged to minimize wiring but there are long metal wires that connect the outputs to the pads. However, this should not impose a severe penalty on the speed of operation since we have used superbuffer drivers.

### 5.7.5. Performance Estimation

The speed of the butterfly module depends on the speed of the adder since it is in the critical path. The adder, which is a conventional Mead and Conway [9] design, was SPICE simulated and the carry out and sum propagation delay was 100ns. Since there is only one add taking place in phase 1 of the cycle this phase can be the same as this delay. So, assuming a 50ns nonoverlap, the cycle of the clock can be 300ns. Therefore the data rate through the butterfly module is 3.3MHz. In the programmable delay module and the feedback loop to the input of the adder superbuffers are used to avoid long delays.

### 5.7.6. Testing

There is no means on chip to test intermediate internal state, although some sections of the circuit can be operated fairly independently from the rest. For example, one could select the external shift register and thus have access to the inputs and outputs of both of the add/subtract chains, although the data would pass through some other gates and the $\pi/2$ rotator. The internal shift register can only be loaded and flushed through the adder/subtractors.

# 6. THEORETICAL WORK

## 6.1. Minimum Latency Transforms

### 6.1.1. Justification

In traditional signal processing, throughput has been the only important measure of performance. However, in many applications, latency is a very important issue. It is important to see that the pipeline processors which we have been discussing are geared only toward high throughput, and that, in fact, pipelining any computation will increase the throughput at the expense of increased latency. One example of possible importance to the Army is the real time side-lobe cancellation of jamming signals fed into arrays in which beam-forming is accomplished via the FFT. Low latency is achievable both through the use of structures which have this quality innately, and through the development of fast circuits which allow all DFT structures to run more quickly.

Since both the butterfly and CORDIC modules make extensive use of adders, the latency of these adders is an important part of the entire system latency. The speed of the adders is governed by the propagation of the carry bit during the addition. Either many pipeline stages must be inserted into the adder circuit or fast carry circuits must be devised. The insertion of pipeline stages will provide high bandwidth computation be at the cost of increased latency in providing the result. Thus a compromise is generally made. Some pipelining and some carry propagation within a pipeline stage are employed. As a result, it is important to use fast carry-lookahead circuits to keep the bandwidth as high as possible. The basic design problem of fast carry look-ahead circuits is to realize a fast circuit with a minimum of gates. Further, the speed and the cost of the circuits depend on the fan-in and fan-out capabilities of the gates used to implement the circuit. The basic speed of a gate depends on these same factors in a negative and non-linear way as well. Thus an optimum design must consider not only the number of gate delays but the interaction of the design with the gate delay time itself.

### 6.1.2. What is the Absolute Minimum?

Clearly, it would seem that minimum latency would be achieved through the use of the original DFT equation, where all $N^2$ products are computed simultaneously, and these products are summed in groups of $N$ to form the outputs. The difficulty is that this would require an adder fan-in of $N$ to compute the sum in one add time which is generally not practical nor available in communication bound VLSI designs. The common method which overcomes this difficulty is the use of fan-in trees which add $f$ numbers at a time and come to the solution of the larger problem after $\log_f N$ add times. Similarly, one does not have multipliers which can fan-out to N adders simultaneously, so that one is forced to utilize a tree of multiplexers or redundant multipliers to communicate the products to their destinations. The result of both of these observations is that one may as well use FFT-like structures of the Cooley-Tukey or Good type merely due to the limited fan-in and fan-out one has available.

### 6.1.3. VLSI Fan-in and Fan-out Considerations

Consider the circuits in figure 52. If one finds that the circuit one desires can be simplified by the use of higher fan-in gates, one would like the gate on the right half of the figure to have a delay no greater than the delay of the circuit on the left. In fact, for most logic families, this is true, even for fan-in as large as 12-15. Of course, the use of these gates in a circuit may cause second order

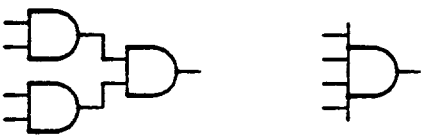


Figure 52: Fan-in Comparison

effects such as increased wire lengths to come into play, reducing the speedup for very large fan-in. Also, some circuits cannot make effective use of large fan-in gates. However, we will see that adders can reap great rewards through the use of gates of fan-in greater than 2.

### 6.1.4. Fast Carry Lookahead

Ladner and Fischer [20] developed a method of reducing the computation time of linear and many nonlinear recurrences from $O(N)$ time to $O(log_2 N)$ time by transforming the recurrences to binary trees. We will apply these techniques to fast carry lookahead circuits and extend them to higher fan-in and fan-out [21, 22].

On the left side of figure 53 is a recurrence which has been expressed in terms of a binary operation denoted by "O".

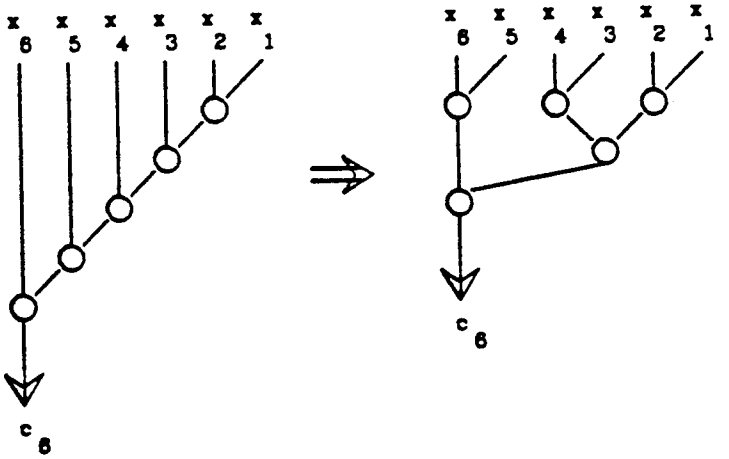



Figure 53: Desired Transformation

The idea is to transform this to the tree on the right side of the figure which clearly does the computation more quickly. This transformation requires the operation to satisfy an associativity property, but does not require linearity, for example. Figure 54 shows circuits for various numbers of inputs where the notation $P_k^j(n)$ is a circuit of $n$ inputs with fan in j. The subscript $k$ allows us to index circuits of different cost/performance, so that $k=0$ implies the highest performance structure, $k=1$ the next lower performance structure, and so on. Note that the figure also shows how these circuits can be built up recursively from circuits of smaller size.

For the general fan-in case, the circuits of figure 55 result. Again, it is seen in figure 56 that the circuits can be built up in a recursive fashion.

Let us now proceed to develop circuits for carry-lookahead. The operation of a full adder circuit for the $i$th bit can be expressed

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

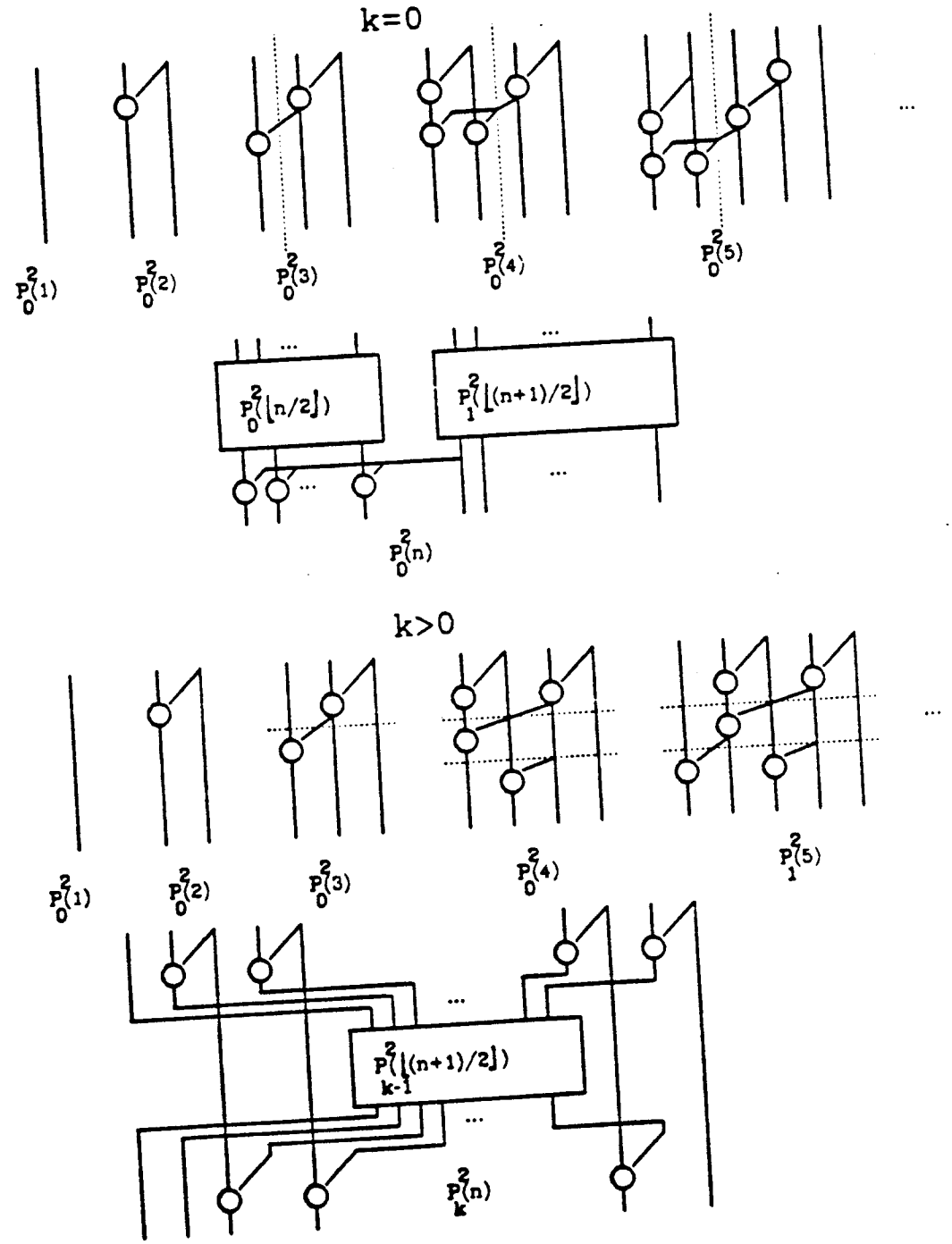



Figure 54: Definition of $P_k^2(n)$ Circuits

$$C_i = (A_i B_i) + (A_i + B_i) C_{i-1}$$

where $\oplus$ means "exclusive or". The recurrence is clearly seen in the expression for $C_i$ and is generally accepted to be the hard part of the calculation. Since it is relatively easy to form the $S_i$ once we have calculated the $C_i$, we will concentrate on speeding up the carry generation part of the circuit. Figure 57 shows the well known "ripple carry" circuit. Here, the circled part of the circuit is not in the proper form to immediately apply the preceding ideas, since it does not

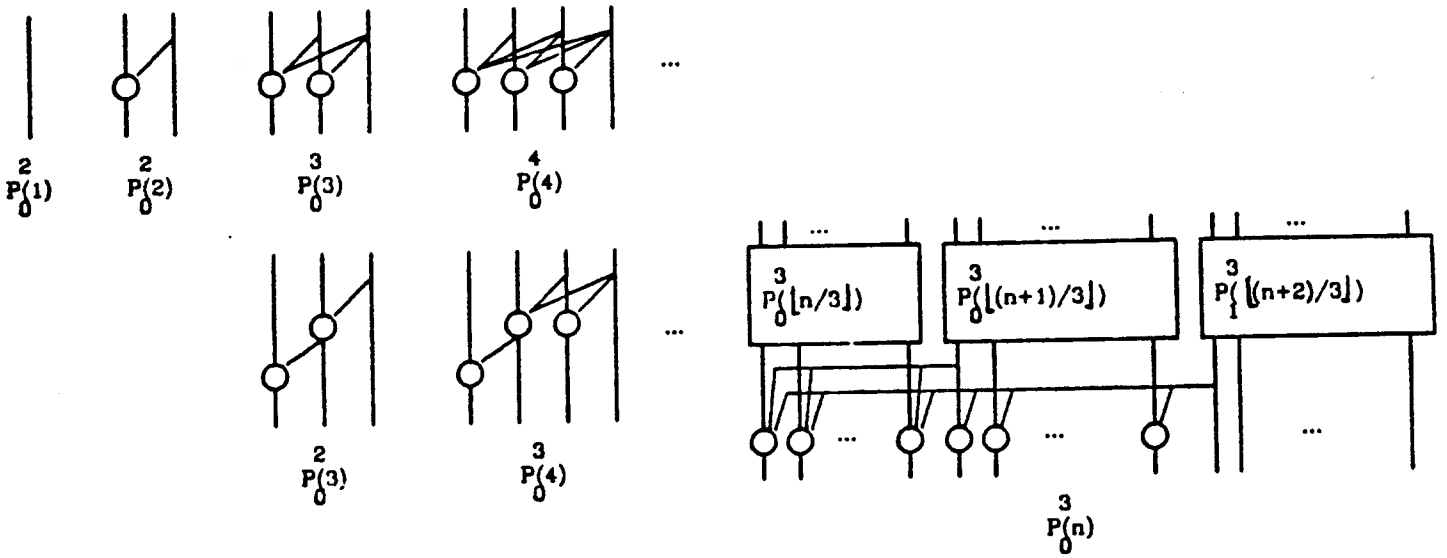



Figure 55: High Fan-in Circuits

satisfy an associativity property. However, it is easy to formulate the circuit so that it does show an associativity which we can exploit. The new operator is shown in figure 58, along with its use in a circuit of size 3.

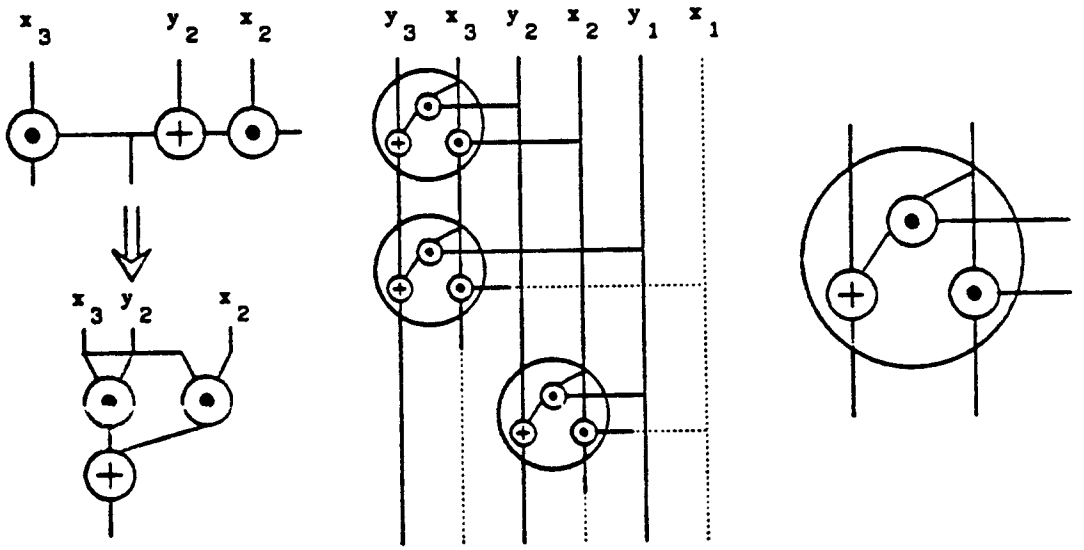



Figure 58: Development of Carry Operator

Figure 59 shows the construction of this node for higher fan-in. Although the operator is now in the proper form to use the $P_k^j(n)$ circuits directly, an additional speedup can be realized by noting that this operator is "asymmetrical" in the sense that the delay from $g_2$ to $g_0$ is two units while the delay from $g_1$ to $g_0$ is one unit. Therefore, our constructions should be modified to take this into account. A diagram of several of these modified circuits, denoted by $Q_k^j(n)$ is shown in figure 60. It is easily seen that, for example, the longest path through the circuit $Q_0^2(3)$ is 3 units using the carry operator above, while the

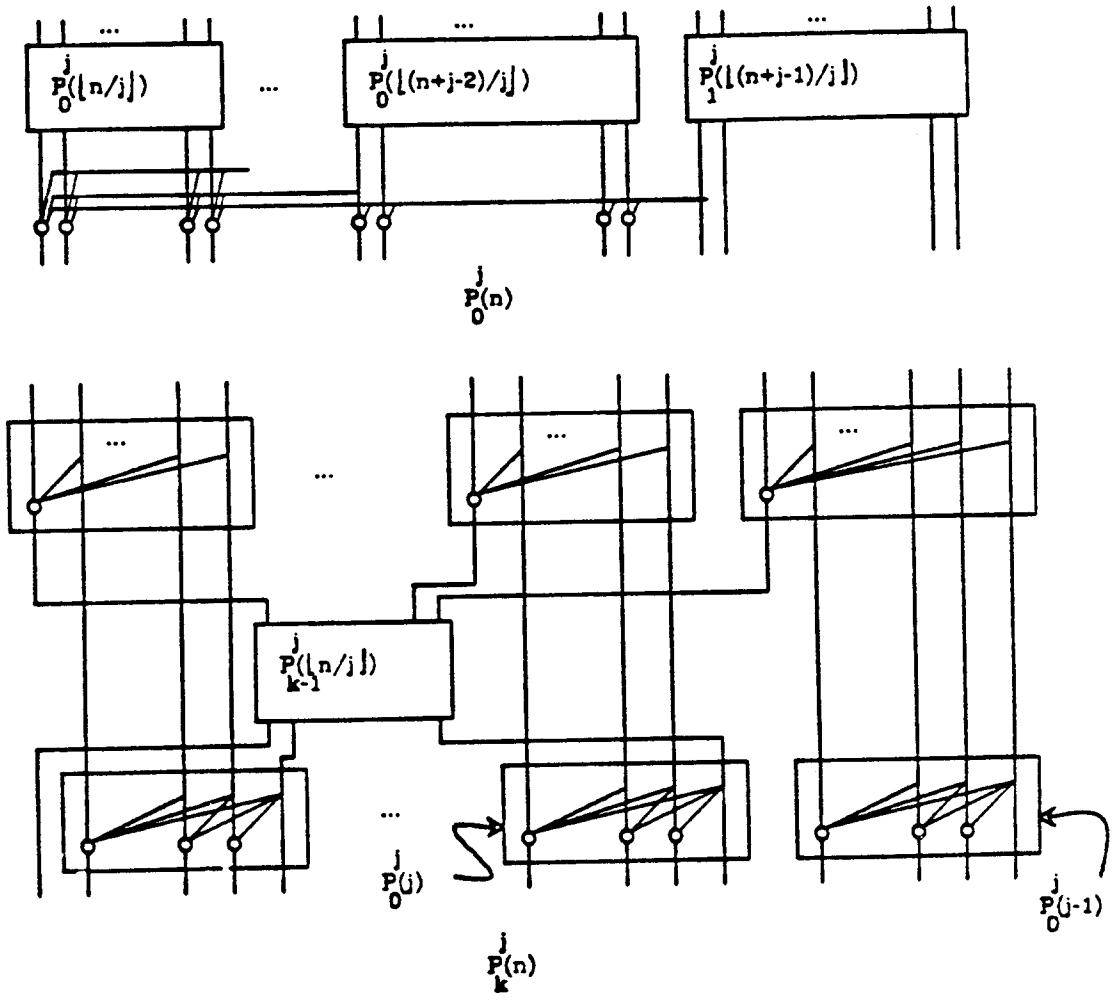



Figure 56: Recursive Construction of High Fan-in Circuits

straightforward use of the $P_0^2(3)$ circuit would have resulted in a path with a 4 unit delay.

For the general fan-in case, figure 61 shows the construction of the highest performance $Q_0^j(N_m)$ circuits. Lower performance circuits can be realized as in figures 62, 63, and 64. An example for fan-in 3 is shown in figure 65.

The total delay of our carry lookahead circuit is thus

$$delay = T(m,j,k) = \begin{cases} 0 & if\ m=0,1 \\ m+k & otherwise \end{cases}$$

and the size of the highest performance ($k=0$) circuit (number of bits wide) is

$$size = N(m,j) = N(m-1,j)+(j-1)N(m-2,j); \quad N(0,j) = N(1,j) = 1.$$

Table 5 shows the various sizes which are available for a given fan-in and delay.

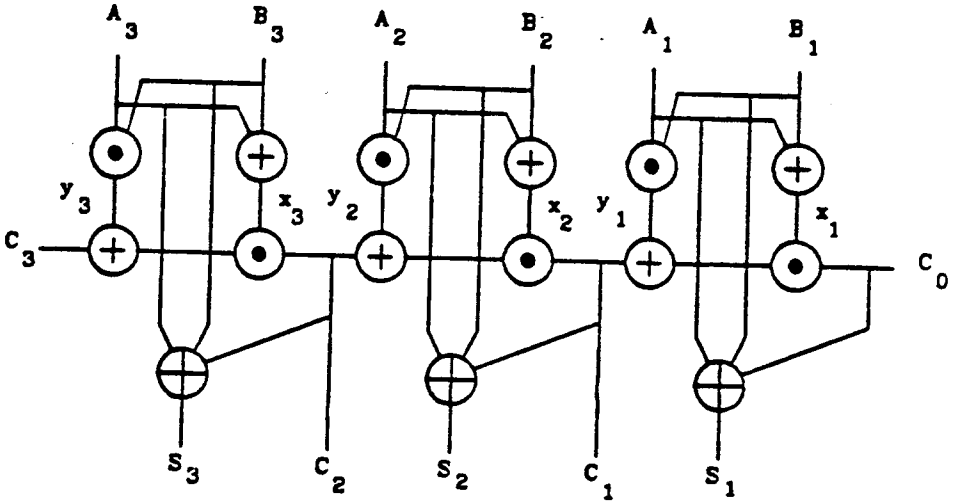



Figure 57: Ripple Carry Circuit

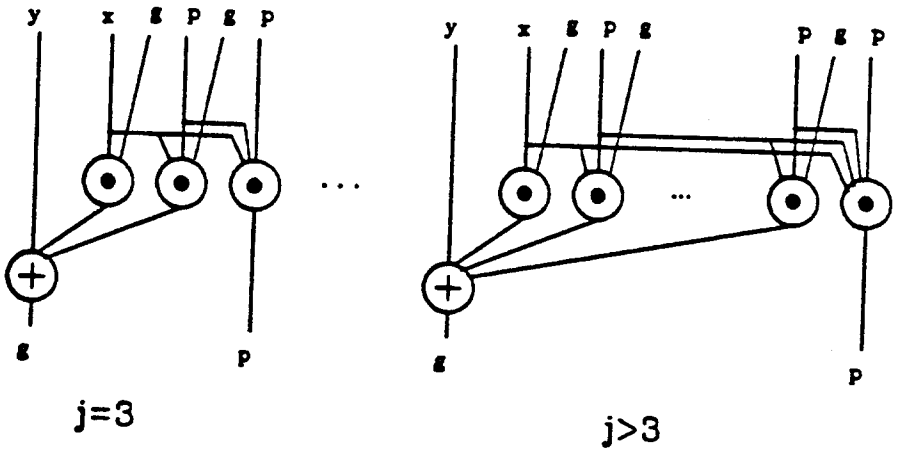



Figure 59: High Fan-in Nodes

Table 5: Circuit Size as a Function of Fan-in and Circuit Delay

| Delay | Fan-in (j) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 3 | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 | 21 | 23 | 25 | 27 |
| 4 | 5 | 11 | 19 | 29 | 41 | 55 | 71 | 89 | 109 | 131 | 155 | 181 | 209 |
| 5 | 8 | 21 | 40 | 65 | 96 | 133 | 176 | 225 | 280 | | | | |
| 6 | 13 | 43 | 97 | 181 | 301 | | | | | | | | |
| 7 | 21 | 85 | 217 | 441 | | | | | | | | | |
| 8 | 34 | 171 | 508 | | | | | | | | | | |
| 9 | 55 | 341 | | | | | | | | | | | |
| 10 | 89 | | | | | | | | | | | | |
| 11 | 144 | | | | | | | | | | | | |
| 12 | 233 | | | | | | | | | | | | |
| 13 | 377 | | | | | | | | | | | | |

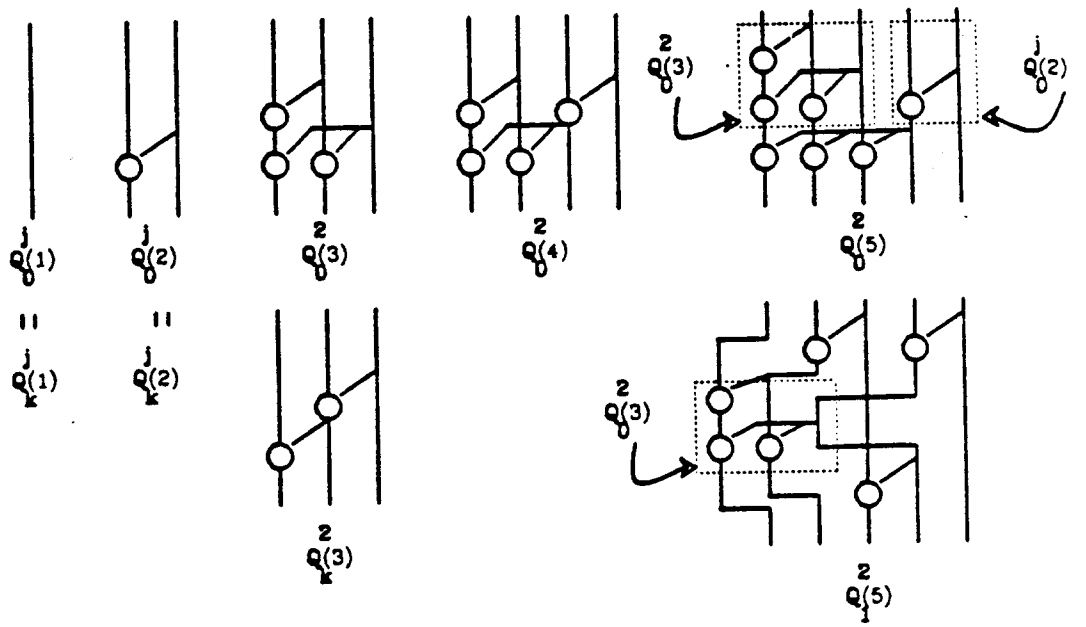


Figure 60: Definition of $Q_k^j(n)$ Circuits

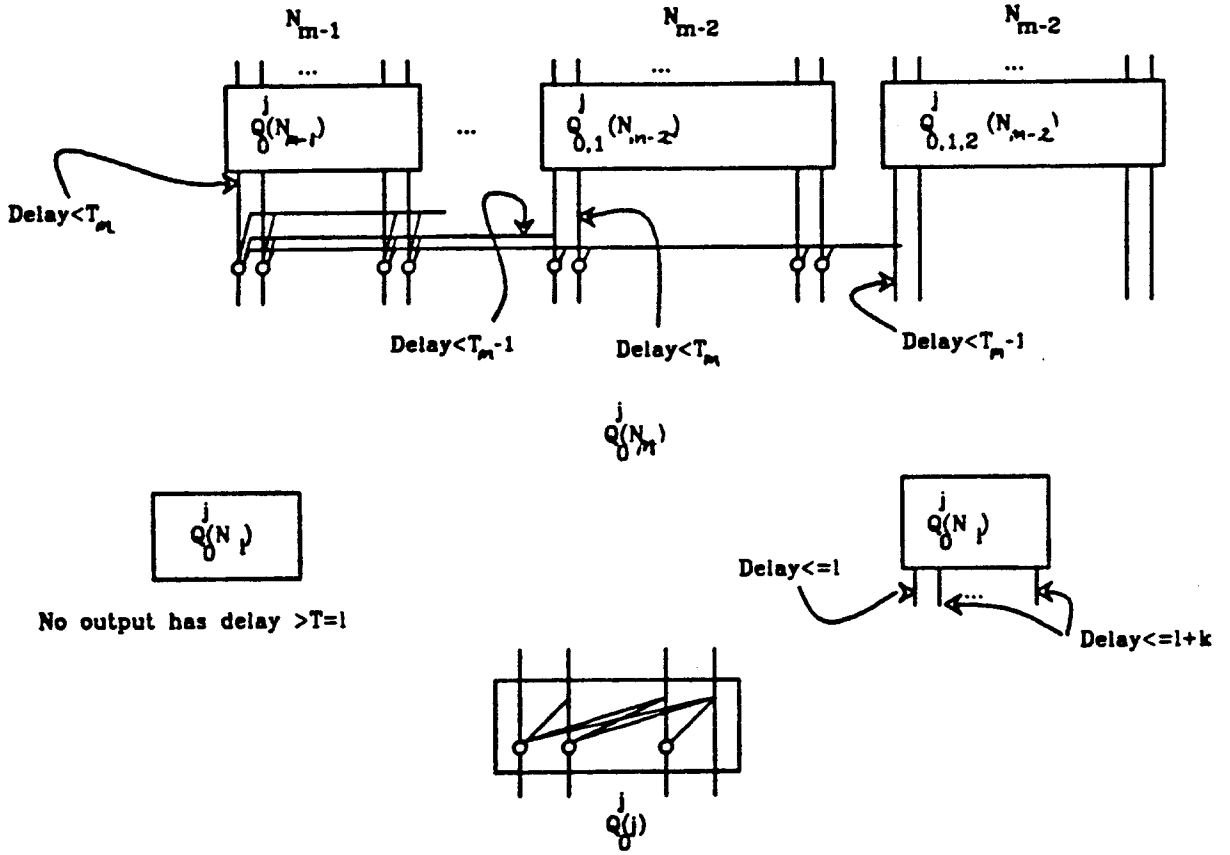



Figure 61: General Construction of $Q_0^j(N_m)$

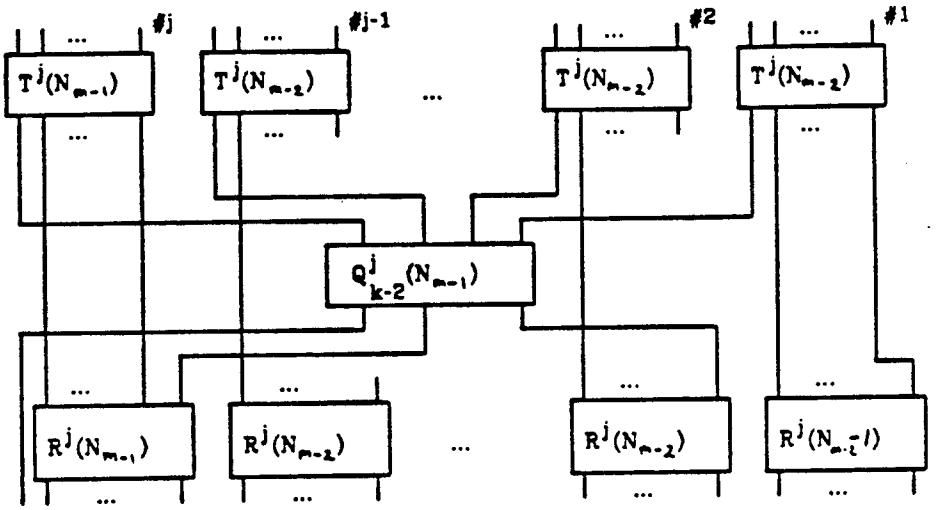



Figure 62: General Construction of $Q^j_k(N_m)$

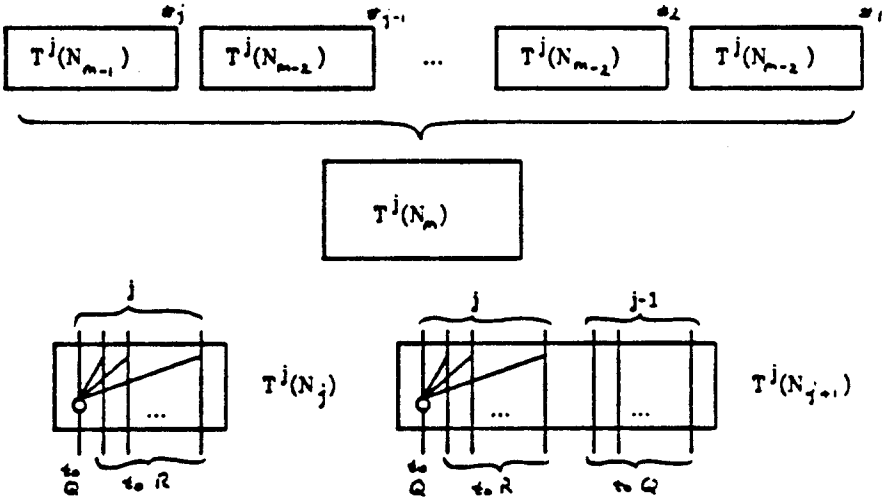



Figure 63: Construction of $T^j(N_m)$

The cost of the high performance circuits is given by the construction of $Q^j_0$ which yields

$$cost = S^j_0(m) = S^j_0(m-1)+(j-2)S^j_1(m-2)+S^j_2(m-2)+(j+1)(N_{m-1}+(j-2)N_{m-2})$$

$$S^j_k(m) = U^j_m + V^j_m + S^j_{k-2}(m) - \begin{cases} j+1 & \text{if } m \text{ is even} \\ 0 & \text{else} \end{cases}$$

$$S^j_0(j) = \frac{(j-1)(j+4)}{2}$$

$$U^j_j = U^j_{j+1} = j+1 \text{ gates}$$

$$V^j_j = V^j_{j+1} = S^j_0(j) \text{ gates}$$

$$U^j_m = U^j_{m-1} + (j-1)U^j_{m-2}$$

$$V^j_m = V^j_{m-1} + (j-1)V^j_{m-2}$$

where $U^j_m$ is the cost of the $T^j_m$ and $V^j_m$ is the cost of the $R^j_m$.

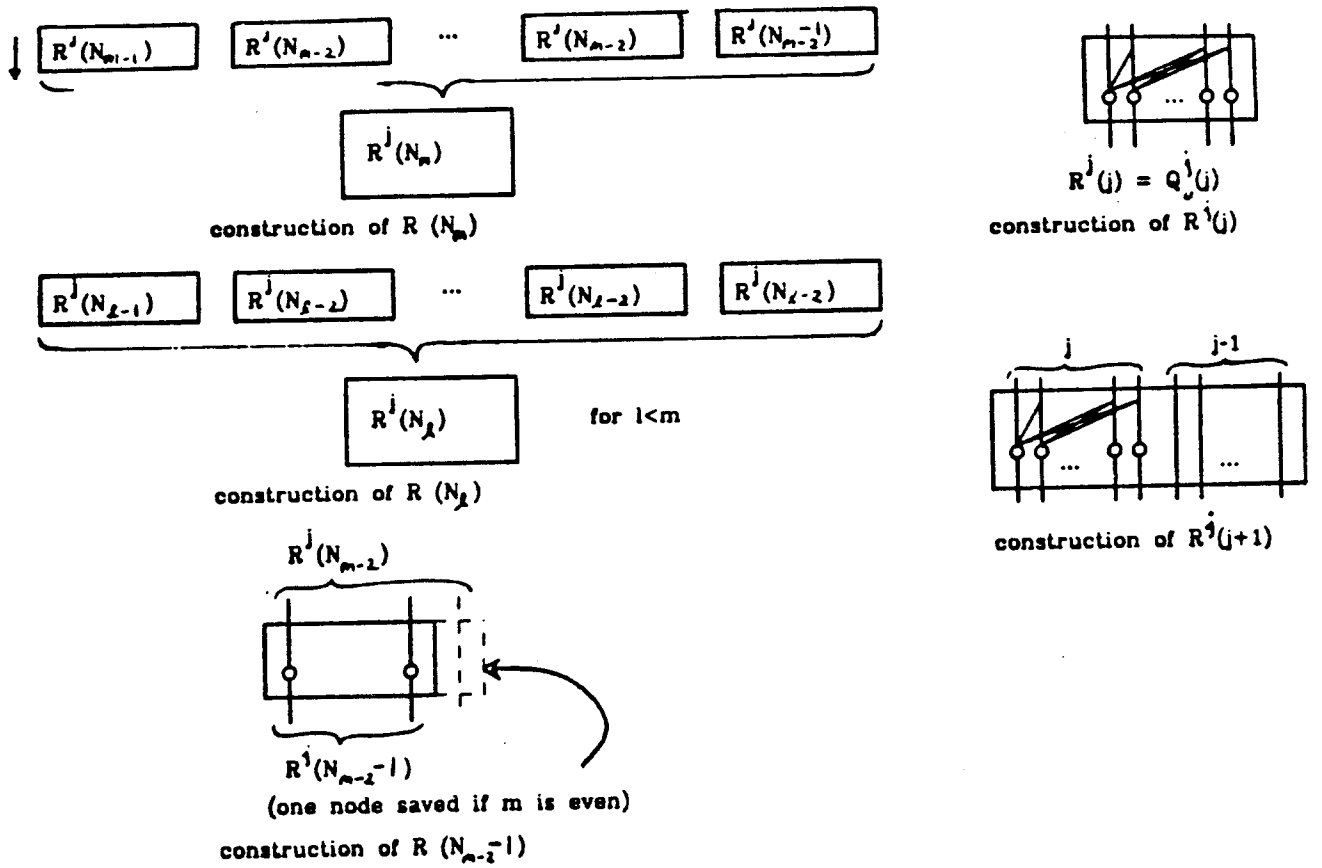



Figure 64: Construction of $R$ circuits

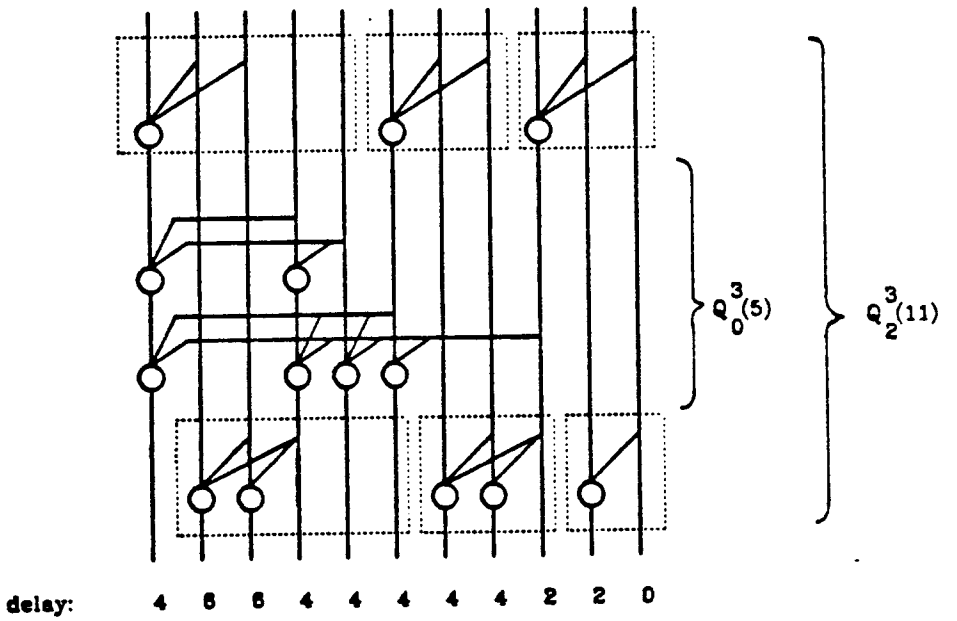



Figure 65: Example for Fan-in 3

If the circuit is the top level circuit, rather than one down inside the recursion, one saves $N_m - 1$ gates due to the unused "p" outputs. Also, $S_k^j(m)$ can be reduced due to the fact that the most significant bits in our present circuit have delay equal to $m$, which is faster than the other bits. Thus, it is advisable to use special circuitry here which is slower and cheaper. Table 6 shows a comparison between these adders for a given circuit size and published results by Ladner and Fischer [20] and Kuck [23], with a ripple carry circuit's cost included for comparison. Observe that these new circuits are not only faster, but in some cases cheaper than the others.

Table 6: Cost Comparison of Adder Circuits

| Delay | Size | Kuck | L.F. | Despain | ripple |
|---|---|---|---|---|---|
| 5 | 8 | | | 38 | |
| 6 | 8 | 29 | 29 | 29 | |
| 16 | 8 | | | | 16 |
| 7 | 16 | | | 75 | |
| 8 | 16 | | 78 | | |
| 9 | 16 | 82 | | | |
| 32 | 16 | | | | 32 |
| 8 | 32 | | | 230 | |
| 10 | 32 | | 191 | 170 | |
| 12 | 32 | 210 | 155 | 155 | |
| 64 | 32 | | | | 64 |

Figure 66 shows a fast adder circuit which would utilize the carry lookahead circuits just developed.

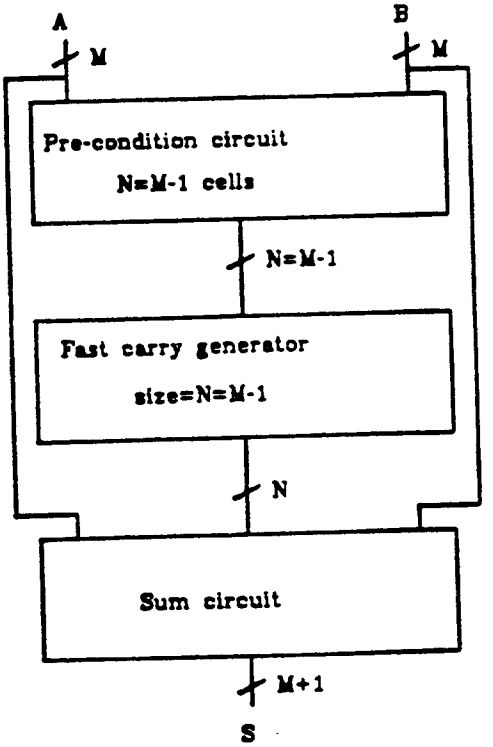



Figure 66: Fast Adder Circuit

### 6.1.5. Other Special Adder Circuits

Although several of the above mentioned circuits lay out regularly in VLSI, a complaint one might have with all of them is that wire lengths needed to interconnect the gates tend to increase without bound. One way this can be handled is by skewing in time bits of the data in blocks and performing lookahead along the blocks. Alternatively, one can use a redundant representation of the data as a sum word and a carry word. When an addition is performed, four words instead of two are presented to the adder circuit, but now only a four to two reduction, instead of a full addition is performed. A circuit to perform this reduction is shown in figure 67.

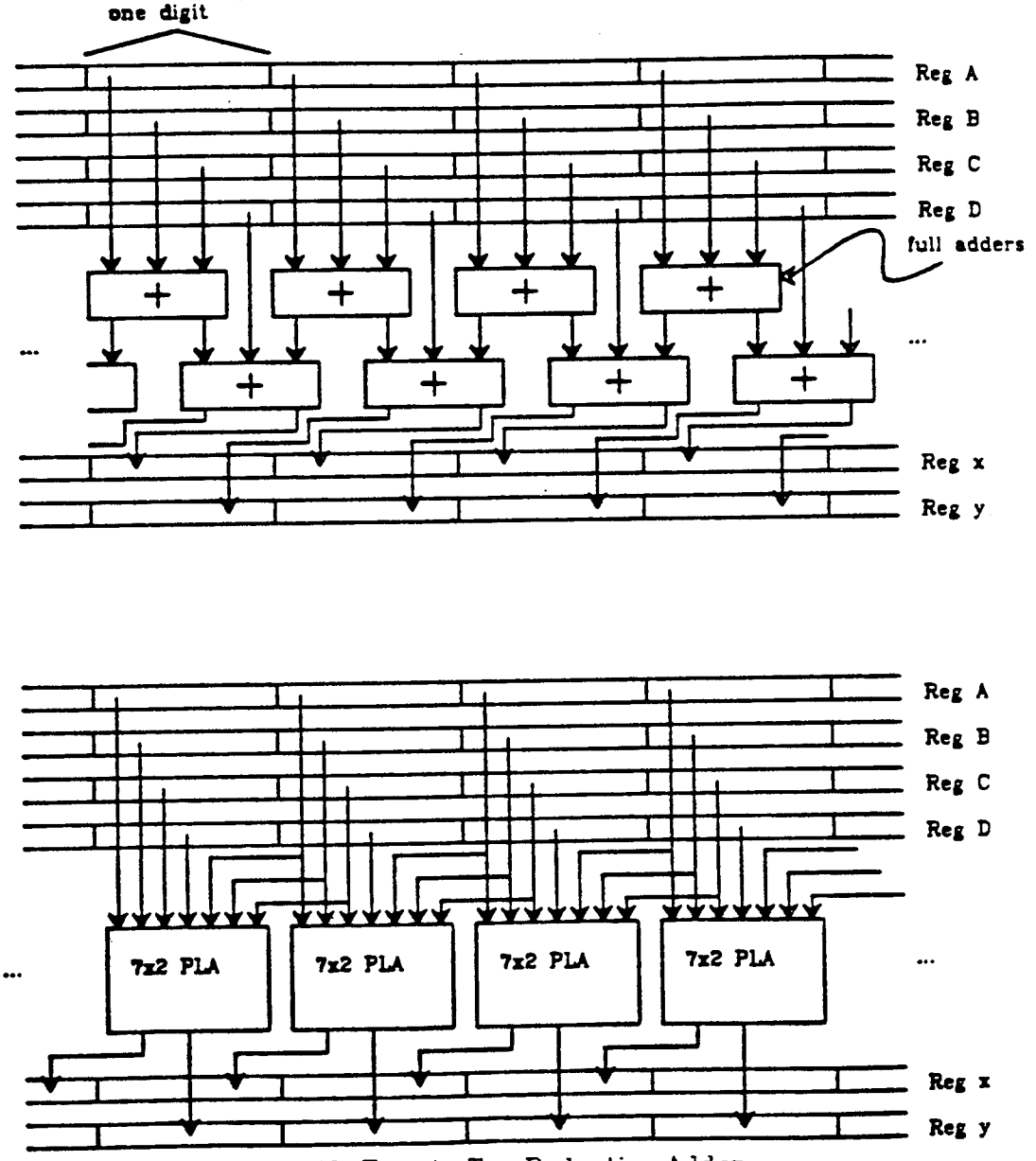



Figure 67: Four to Two Reduction Adder

It is easy to see that the wire lengths as well as the total add delay are constants and do not depend on the word size. The circuit is of course more expensive than a carry lookahead circuit due to the fact that 2 adders per bit are required and much area will be taken up by wiring due to the redundant representation.

However, the circuit clearly has the lowest latency of any we have seen so far and lends itself quite well to pipelined processors where many adds on the same data are performed as in a CORDIC rotator. It is also shown that the two adders which are required could be collapsed into a single 7×2 PLA.

### 6.1.6. Parallel Versus Cascade Structures

The latency of an FFT processor can be considered to have two parts. First, there will be a certain latency due to the delays of the gates and the adder circuits which we have already addressed. Secondly, there will be an additional latency which is a function of the degree of parallelism in the processor. In the Despain Cascade, for example, there is a fundamental lower bound on the latency proportional to the size of the transform which is due to the fact that one must wait for all the outputs to stream out of the processor serially. Clearly, this difficultly can be side-stepped through the use of hybrid parallel-pipeline structures such as those which were presented previously. If cost were no object, one could achieve latencies which were dominated by the delay through the $\log_r$ stages of radix-$r$ DFT modules and CORDIC rotators. However, in most cases, the cost is multiplied by the same factor as the speedup, and, in the VLSI case, the cost can go up even faster than this as we have seen before.

## 6.2. A Broad Survey of Fourier Transform Circuits

In an attempt to broaden our study of Fourier Transform circuitry, we briefly characterized as many different designs as possible. Our results are summarized in Table 7. Each line in the table corresponds to one of the circuits described later in this section. All the performance figures are reported in asymptotic form, in the sense described below.

A design is said to occupy area $N$ if there exists some constant $c$ for which the circuit (built according to that design) solving an $N$-element Fourier transform occupies no more than $cN$ square wire-widths of VLSI area. If the constant $c$ happened to be 10000, this means that a 100-element Fourier transform could be solved on a chip that was $\sqrt{10000*100} = 1000$ wire-widths on a side.

Similarly, a design with area performance $N^2$ defines a family of circuits with the property that a circuit solving an $N$-element transform occupies no more than $cN^2$ square wire-widths. In this case, doubling the size of the transform results in a circuit with approximately four times the area. The word "approximately" is crucial here, since equality may only be observed in the limit of infinitely large $N$. For small values of $N$, asymptotic performance figures can be misleading: an area performance of $N^2$ is assigned to a circuit occupying $c_1N^2 + c_2N$ area, even if $c_1$ is much smaller than $c_2$. Nonetheless, the asymptotic area performance of a design is usually a good indication of relative circuit size.

The "Time" figure for a design is also defined in asymptotic terms. A time performance of $N$ is assigned to a design if its $N$-element circuit can solve one Fourier transform every $cN$ clock cycles. The length of a clock cycle is independent of the value of $N$. This rules out the use of gates whose fanout grows with transform length, unless the output of the gate is fed into an amplifier whose design (and delay) varies with $N$. Similar considerations lead to a rather restrictive set of rules for designing designs. A complete list and explanation of these rules is contained in [24].

A "Delay" column is included for each design, for it may define pipelined circuits. If so, the "Delay" figure is larger than the "Time" figure. The former

refers to the latency of the circuit, the latter refers to the number of clock cycles that separate successive transformations.

The column marked Area*Time$^2$ indicates whether or not the design is an optimal one. The last four have the best possible Area*Time$^2$ figures Any modification leading to a smaller area figure must increase the time figure, for no circuit can have an Area*Time$^2$ performance better than $N^2\log^2 N$ [24].

| Design | Area | Time | Area*Time$^2$ | Delay |
|---|---|---|---|---|
| 1-cell DFT | $N \log N$ | $N^2 \log N$ | $N^5 \log^3 N$ | $N^2 \log N$ |
| $N$-cell DFT | $N \log N$ | $N \log N$ | $N^3 \log^3 N$ | $N^2 \log N$ |
| $N^2$-cell DFT | $N^2 \log N$ | $\log N$ | $N^2 \log^3 N$ | $N^2 \log N$ |
| 1-proc FFT | $N \log N$ | $N \log^2 N$ | $N^3 \log^5 N$ | $N \log^2 N$ |
| Cascade | $N \log N$ | $N \log N$ | $N^3 \log^3 N$ | $N \log^2 N$ |
| FFT Network | $N^2$ | $\log N$ | $N^2 \log^2 N$ | $\log^2 N$ |
| Perfect Shuffle | $N^2/\log^2 N$ | $\log^2 N$ | $N^2 \log^2 N$ | $\log^2 N$ |
| CCC | $N^2/\log^2 N$ | $\log^2 N$ | $N^2 \log^2 N$ | $\log^2 N$ |
| Mesh | $N \log^2 N$ | $\sqrt{N}$ | $N^2 \log^2 N$ | $\sqrt{N}$ |

**Table 7**: Area-time performance of the Fourier transform-solving circuits.

When delay figures are taken into consideration, only the last three designs are seen to be optimal. The Perfect Shuffle, the CCC and the Mesh are the only designs that achieve the limiting Area*Delay$^2$ product of $\Omega(N^2\log^2 N)$. These designs keep all their multiply-add cells and wires busy solving Fourier transforms using the efficient FFT algorithm. All the others, save one, use too few processors † or an inefficient algorithm. The FFT network is an interesting exception to this observation. Its delay inefficiency seems to be a result of its slow bit-serial multipliers. If fast parallel multipliers were employed, the delay in each stage of the FFT network might be as low as $O(\log\log N)$. This would not increase its total area significantly, since its area is still dominated by its "butterfly" wiring. The improved FFT network could thus have a area*time$^2$ product of as little as $O(N^2\log^2 N \log\log^2 N)$.

As indicated above, asymptotic figures can hide significant differences among supposedly optimal designs due to "constant factors". The area and time estimates employed in this study are not sensitive to the relative complexity of the various control circuits required in different designs. For example, the $N^2$-cell DFT, the Cascade, the FFT Network and the Perfect Shuffle are especially attractive designs because they have no complicated routing steps. They are thus given a more detailed examination below.

As indicated in Table 7, the $N^2$-cell DFT is nearly optimal in its area*time$^2$ performance. However, it is by far the largest design since it uses more than $N^2$ multiply-add cells. (The others use $O(N \log N)$ or fewer cells.) Using current technology, one might place 10 multiply-add cells on a chip [7]. This means that one hundred thousand chips would be needed for a thousand-element FFT! Thus the $N^2$-cell DFT design cannot be considered feasible until technology improves to the point that 100 or 1000 cells can be formed on a single wafer. Even then, the interconnections between chips will pose some difficulties, for

† The word "processor" refers to a stored-program computer. There may of course be many such processors in a single Fourier transform circuit. This usage of "processor" should not be confused with the "FFT processor" in the title of this report. An FFT processor is a complete Fourier transform circuit. In an attempt to avoid further confusion between "processors" and "FFT processors", this section always refers to the latter as "Fourier transform circuits".

there are 40 cells on the "edge" of a 100-cell chip.

The $N$-cell DFT is an attractive design at present, despite its non-optimal area*time$^2$ performance. It uses only $2N$ cells in a linear array, so that a thousand-element Fourier transform can be implemented with only $10^2$ chips of 10 multiply-add cells each. This design is of course much slower than the $N^2$-cell DFT, since it produces only one element of a transform at a time rather than an entire transform.

The FFT Network is also fairly attractive at present, for its $(N/2)*(\log N)$ cells can be formed on about the same number of chips as the $N$-cell DFT, yet its performance is equal to the $N^2$-cell DFT. The drawback of the FFT Network is that the wiring on and between the chips is very area-consuming. It also has very long intercell wires, whereas the DFT designs use only nearest-neighbor connections.

The constant factors involved in the Perfect Shuffle design are very similar to those in the FFT Network discussed above. The Perfect Shuffle uses a factor of $\log N$ fewer cells than the FFT Network, so it is a bit smaller and slower. However, it suffers from the same problem of long inter-chip wires and poor partitionability.

The Cascade is another non-optimal design, like the $N$-cell DFT, that deserves consideration because of its good "constant factors." It uses only $\log N$ multiply-add cells and $N$ words of shift-register memory. These are arranged in a simple linear fashion. The Cascade achieves the same performance as the $N$-cell DFT, producing one element of a Fourier transform during each multiply-add time. It is superior to the $N$-cell DFT in that it uses many fewer multiply-add cells.

### 6.2.1. Building blocks

All of the Fourier transform circuits described in the next nine subsections are built from a few basic building blocks: shift registers, multiply-add cells, random-access memories, and processors. These are described below.

A $k$-bit shift register can be built from a string of $k$ logic nodes in $O(k)$ area. Each of the logic nodes stores one bit. Shift registers are used to store the values of variables and constants; these values may be accessed in bit-serial fashion, one bit per time unit.

Multiply-add cells are used to perform the arithmetic operations in a Fourier transform. Each cell has three bit-serial inputs $\omega^k$, $x_0$ and $x_1$. It produces two bit-serial outputs

$$y_0 = x_0 + \omega^k x_1 \quad \text{and} \quad y_1 = x_0 - \omega^k x_1. \tag{1}$$

The inputs and the outputs are all $\lceil \log M \rceil = \Theta(\log N)$ bit integers.

It is fairly easy to see that a simple (if slow) multiply-add cell can be built from $O(\log N)$ logic gates [7]. The multiplication is performed by $O(\log N)$ steps of addition in a carry-save adder. The subsequent addition and subtraction can also be done in $O(\log N)$ time. Thus a complete multiply-add computation can be done in $O(\log N)$ time and $O(\log N)$ area.

The aspect ratios of the multiply-add cell and shift register may be adjusted at will. They should be designed as a rectangle of $O(1)$ width that can be folded into any rectangular shape.

An $S$-bit random-access memory with a cycle time of $O(\log S)$ can be built in $O(S)$ area, using the techniques of Mead and Rem [25]. (Their area and time analyses are essentially consistent with the model used here; see [7] for a comparative study of the two models.) The cycle time claimed above is the best

possible, given the logarithmic delay Assumption 3c, since most of the storage locations are at least $\sqrt{S}$ wire-widths from the output port of the memory. To achieve this optimal cycle time, the number of levels in Mead and Rem's hierarchical memory must grow proportionally with $\log S$.

Processors are used to generate control signals, whenever these become complex. Each processor is a simple von Neumann computer equipped with an $O(\log N)$-bit wide ALU, $O(\log N)$ registers, and a control store with $O(\log N)$ instructions. The cycle time of a PE is $O(\log N)$ time units. This is enough time to fetch and execute a register-to-register move, a conditional branch, an "add", or even a "multiply" instruction. It is also enough time to allow the processor's operands to come from an $N$-bit random-access memory.

At least $O(\log^2 N)$ units of area are required to implement a processor, since it has $O(\log N)$ words = $O(\log^2 N)$ bits of storage. A straightforward, if tedious, argument can be made to show that $O(\log^2 N)$ area is actually sufficient to build a processor [7]. Neither the ALU, the data paths, nor the instruction decoding circuitry will occupy more room (asymptotically) than the control store.

### 6.2.2. The Direct Fourier Transform on One Multiply-Add Cell

The naive or "direct" algorithm for computing the Fourier transform is to compute all terms in the matrix-vector product of Assumption 5d. Following this scheme, a total of $N^2$ multiplications are required when an $N$-element input vector $\vec{x}$ is multiplied by an $N$-by-$N$ matrix of constants $\vec{A}$, to yield an $N$-element output vector $\vec{y}$. Three degrees of parallelism immediately suggest themselves: the product may be calculated on one multiply-add cell, on $N$ multiply-add cells, or on $N^2$ multiply-add cells. Each possibility is discussed separately in the discussion that follows.

A single multiply-add cell will take $O(N^2 \log N)$ time to perform all the calculations required in the direct Fourier transform algorithm. (Recall that a multiply-add calculation takes $O(\log N)$ time.) To this must be added the overhead of calculating the constants in the matrix $\vec{A}$, since a prohibitively large amount of area would be required to store these explicitly. Fortunately, this calculation is quite simple. The constant required during the $ij$-th multiply-add step (see statement 4 of Figure 68) can generally be obtained by multiplying $\omega^i$ by the constant used in the previous multiply-add step, $\omega^{i(j-1)}$. A single processor is capable of performing this calculation, supplying the necessary constants to the multiply-add cell as rapidly as they are needed. The time performance of the uniprocessor DFT design is thus $O(N^2 \log N)$.

```
1. FOR i ← 0 TO N−1 DO
2.     y_i ← 0;
3.     FOR j ← 0 TO N−1 DO
4.         y_i ← y_i + ω^ij x_j;
5.     OD;
6. OD.
```

**Figure 68:** The naive or "direct" Fourier transform algorithm.

The area required by the single multiply-add cell design is $O(\log N)$ for the multiply-add cell, $O(\log^2 N)$ for the processor supplying the constants, and $O(N \log N)$ for the random-access memory containing the input and output registers. This last contribution clearly dominates the others, giving the uniprocessor DFT design a total area of $O(N \log N)$. Its combined area*time$^2$ performance is thus a dismal $O(N^5 \log^3 N)$. It has far too little parallelism for its area.

The designs in the next two subsections employ progressively more parallelism to achieve better performance figures.

### 6.2.3. The Direct Fourier Transform on $N$ Cells

Kung and Leiserson [25] were apparently the first to suggest that the Fourier transform could be computed by the "direct" algorithm on $2N-1$ multiply-add cells connected in a linear array. These cells operate with a 50% duty cycle: the even-numbered cells and the odd-numbered cells alternately perform the computational step described below. An obvious optimization [25] results in a circuit using only $N$ multiply-add cells to accumulate the terms in the DFT.

The entire DFT calculation is complete in $4N-3$ computational steps. During each step in which it is active, each even- (or odd-) numbered cell computes $y' \leftarrow y + ax$ using the value $y$ provided by its right-hand neighbor (the leftmost cell always uses $y=0$). The $y'$ values eventually emerging from the leftmost cell are the outputs $\vec{y}$ in natural order. The inputs $\vec{x}$ to the circuit enter through the leftmost cell and are passed, unchanged, down the line of cells. Due to the 50% duty cycle of the cells, one $y'$ value is produced (and one $x$ value is consumed) every other computational step.

The only complicated part of the circuit has to do with computing the constant values $a$. A complete description of this computation is rather lengthy [25]; only a sketch is attempted here. Suffice it to say that each $a$ value is obtained by a single multiplication from the $a$ value previously used by the cell next closest to the center of the line. The only exception to this rule is that the constant-generating circuitry for the centermost cell must perform four multiplications to obtain the next $a$ value. (Perhaps a fast multiplier might be provided for the centermost cell, to keep it from slowing down the whole array.) In any event, the constant-generating circuitry for each cell performs a fixed sequence of register-register operations, all off which can be completed in $O(\log N)$ time and $O(\log N)$ area.

The time performance of the $N$-cell DFT design is $O(N \log N)$, since each of the $4N-3$ computational steps can be completed in $O(\log N)$ time. The total area of the $N$ cells and their constant-generating circuitry is $O(N \log N)$.

Note that the total area of the $N$-cell DFT design is asymptotically identical to that of the 1-cell design. This is a reflection of the fact that a register takes the same amount of room (to within a constant factor) as a multiply-add cell. However, one can confidently expect that an actual implementation of the 1-cell design will be significantly smaller than an $N$-cell design due to this "constant factor difference."

The area*time$^2$ performance of the $N$-cell DFT design is $O(N^3 \log^3 N)$. This is far from optimal, but it is a great improvement on the 1-cell design. The next subsection describes an $N^2$-cell design that has a nearly optimal area*time$^2$ performance figure.

### 6.2.4. The Direct Fourier Transform on $N^2$ Cells

One way of boosting the efficiency of the $N$-cell DFT design is to pipeline its computation. Instead of circulating intermediate values among one row of $2N-1$ cells for $4N-3$ steps, one can "unroll" the computation onto $4N-3$ rows of $2N-1$ cells. Now each problem instance spends just one computational step on each row of cells before moving on to the next row. (Note that there are actually about $8N^2$ cells in the "$N^2$-cell" design.)

All I/O occurs through the leftmost cell in the odd-numbered rows, in the staggered order shown in Figure 69. This figure shows only the I/O for a single problem instance; inputs for successive problem instances may follow immediately behind the analogous inputs for the previous problem, after a delay of one computational step.

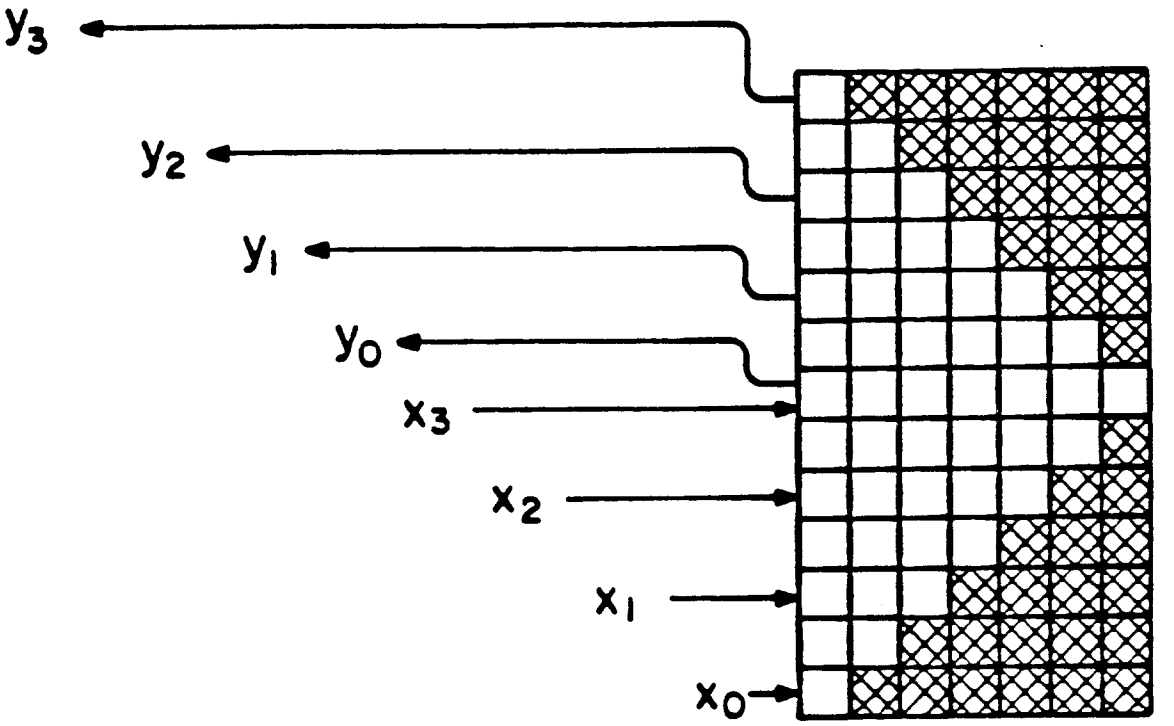



**Figure 69**: Staggered I/O pattern for the $N^2$-cell DFT design.

More precisely, the first input for each problem instance enters the leftmost cell of the first row. The second input enters the leftmost cell of the third row, two computational steps later (remember that each computational step, as defined in the previous subsection, involves only "even" or "odd" cells). The $N$-th input enters the leftmost cell of the $(2N-1)$-th row, $2N-2$ computational steps after the first input entered the circuit. At the end of this step, the first output is available from this same cell. The second output comes from the leftmost cell of the $(2N+1)$-th row, after two more steps...and finally the $N$-th output emerges from the leftmost cell of the $(4N-3)$-th row, $(4N-3)$ computational steps after the first input was injected into the circuit.

As noted above, the $k$-th input for another problem instance can follow immediately behind the $k$-th input for the previous problem, delayed by only one computational step. The circuit thus operates in pipelined time $T = O(\log N)$. The total area of the $N^2$-cell design is $A = O(N^2 \log N)$, since each cell occupies $O(\log N)$ area. The combined area*time$^2$ performance of the design is only a factor of $O(\log N)$ from the optimal figure of $\Omega(N^2 \log^2 N)$. Thus it is pointless to look for a smaller circuit with a similar pipelined time performance. However, it is possible to make great improvements on this circuit's solution delay, as shown by the $(N \log N)$-cell FFT design presented in a later subsection.

It is fairly easy to describe a few "constant factor" improvements to the $N^2$-cell DFT design. First of all, at least half of the cells on each row are idle, due

to the 50% duty cycle inherent in the Kung-Leiserson approach. Secondly, the computations done in the shaded portion of Figure 2 are irrelevant (the resulting $y'$ values do not affect the circuit's outputs). Each of these considerations halves the number of required multiply-add cells, leaving fewer than $2N^2$ cells in an optimized design. Finally, the constant-generating circuitry described for the $N$-cell design need not be carried over to the $N^2$-cell design, for each cell uses the same $a$ value every time it does a computational step. In other words, the constant matrix $\vec{A}$ can be "hard-wired" into the registers of the multiply-add cells. to circulate the constant matrix $\vec{A}$ among the multiply-add cells.

### 6.2.5. The Fast Fourier Transform on One Processor

Up to now, all the circuits have computed the Fourier transform by the naive or direct algorithm. Great increases in efficiency are observed in conventional uniprocessors using the fast Fourier transform algorithm; it would be remarkable indeed if we could not take advantage of our knowledge of the FFT in the design of Fourier transform circuits.

There are a number of versions of the FFT in the literature, differing chiefly in the order in which they use inputs, outputs, and constants. Figure 70 shows a "decimation in time" algorithm, taken from Figure 5 of [26]. Figure 71 shows a "decimation in frequency" algorithm, adapted from Figure 10 of [26]. In both cases, the $N$ problem inputs are stored in $x_i$, the $N$ problem outputs are $y_i$, and $\omega$ is a principal $N$-th root of unity.

```
1.  FOR b ← (log N) − 1 TO 0 BY −1 DO
2.      p ← 2^b;   q ← N/p;  /* note that N = pq  */;
3.      z ← ω^p;  /* z is a principal q-th root of unity  */;
4.      FOR i ← 0 TO N−1 DO
5.          j ← i mod q;   k ← reverse(i);
6.          IF (k mod p) = (k mod 2p) THEN
7.              <x_k, x_{k+p}> ← <x_k + z^j x_{k+p},  x_k − z^j x_{k+p}>;
8.          FI;
9.      OD;
10. OD;
11. FOR i ← 0 TO N−1 DO    /* unscramble outputs  */;
12.     y_{reverse(i)} ← x_i;
13. OD.
```

**Figure 70:** The FFT by "decimation in time." Note: $reverse(i)$ interprets $i$ as an unsigned (log $N$)-bit binary integer then outputs that integer with its bits reversed, i.e., with its most-significant bit in the least-significant position.

Either Figure 70 or 71 may be used as an algorithm for a uniprocessor that runs in $O(N \log N)$ computational steps. The total area of such a design is $O(N \log N)$, due mostly to input and output storage. (Recall that a single processor fits in $O(\log^2 N)$ area.) Total time for an $N$-element FFT is $O(N \log^2 N)$, since each computational step takes $O(\log N)$ time units. This is, as expected, a vast improvement over the uniprocessor DFT circuit. However, it is far from being area*time$^2$ optimal, for its processor/memory ratio is too high. Adding more processors, as in the following design, increases the performance of an FFT circuit.

```
1.  FOR b ← (log N) − 1 TO 0 BY −1 DO
2.     p ← 2^b;   q ← N/p;
3.     z ← ω^(q/2);  /* z is a principal 2p-th root of unity */;
4.     FOR i ← 0 TO N−1 DO
5.        j ← i mod p;
6.        IF (i mod 2p) = j THEN
7.           <x_i, x_{i+p}> ← <x_i + x_{i+p},  z^j x_i − z^j x_{i+p}>;
8.        FI;
9.     OD;
10. OD;
11. FOR i ← 0 TO N−1 DO  /* unscramble outputs */;
12.    y_{reverse(i)} ← x_i;
13. OD.
```

**Figure 71**: The FFT by "decimation in frequency."

### 6.2.6. The Cascade Implementation of the Fast Fourier Transform

The Cascade arrangement of log $N$ multiply-add cells [2] was discussed at great length earlier in this report in Section 2.2. At the risk of seeming repetitious, we will describe it again using the theoretical notation of this section.

In a Cascade, one of the outputs of each multiply-add cell is connected to the input of a shift register of an appropriate length. See Figure 72. The shift register's output is connected to one of the multiply-add cell's inputs, forming a feedback loop. The remaining inputs and outputs of the multiply-add cells are used to connect them into a linear array. Problem inputs (values of $\vec{x}$) are fed into the leftmost cell; problem outputs (values of $\vec{y}$) emerge from the rightmost cell. The decimation in frequency algorithm of Figure 71 is employed, to keep the cells' computations as simple as possible.

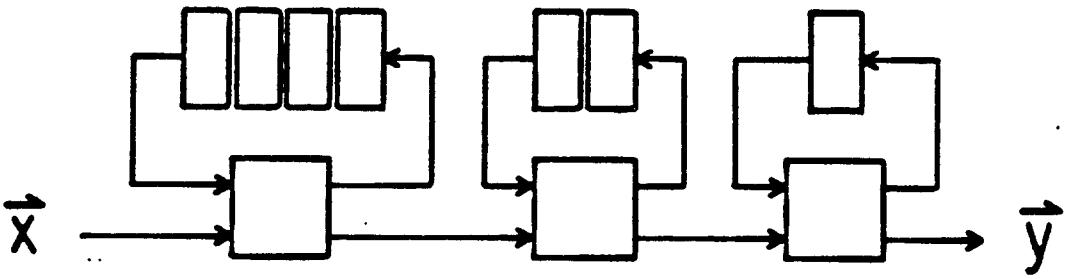



**Figure 72**: The Cascade arrangement of 3 multiply-add cells, for computing 8-element FFTs. The multiply-add cells are square; the rectangular boxes each represent one word of shift register storage.

Each cell handles the computations associated with a single value of the loop index $b$ in Figure 71. The leftmost cell performs the loop for $b = \log N - 1$; the rightmost cell performs the loop computations for $b = 0$. The pairing of $x$ values indicated in statement 7 of Figure 71 is accomplished by the $2^b$-word shift register associated with cell $b$.

The attentive reader will note that statement 7 is not exactly the same as the multiply-add step defined in Equation (1). Statement 7 involves one constant value $z^j$, two variable values $x_i$ and $x_{i+n}$, two additions, but two (instead of one) multiplications. Thus its computation will take about twice as much time or area as a "standard" multiply-add step.

The conditional test of statement 6 is implemented by having each cell monitor the $b$-th bit of the count $i$ of input elements that it has already processed. The condition of statement 6 is satisfied whenever that bit is 0. In this case, a cell performs the computation indicated in statement 7. It sends the new value for $x_i$ to the right, and retains the new value for $x_{i+p}$ in its shift register. Whenever the $b$-th bit of $i$ is 1, no multiply-add computations are performed. However, some data movement is necessary: the data appearing on the cell's lower input line should be copied into its shift register. Also, the values emerging from its shift register should be sent on to the next cell on its right.

One of the advantages of using the decimation in frequency algorithm on the Cascade is the ease of computing the constants for its multiply-add steps. Only a few registers and a single multiplier are required to generate the constants required by each cell. Referring again to the program of Figure 71, the constant $z^j$ required in statement 7 may be obtained by multiplying the previously generated constant $z^{j-1}$ by $z$. If this multiplication is performed whether or not statement 7 is executed, no conditional transfers are necessary in the constant-generating circuitry.†

As noted above, the constant-generating circuitry for each cell consists of a multiplier and a few registers. It is thus comparable in area and time complexity to the multiply-add cell itself. Thus the total area of the Cascade design is obtained by multiplying the number of cells, $\log N$, by the area per cell $O(\log N)$. To this must be added the area of the shift registers. Unfortunately, there is a total of $N-1$ words of storage in these registers, so the entire design occupies $O(N \log N)$ area. Thus the Cascade, like the one-processor design, is almost all memory. An entire problem instance must be stored in the circuit while the Fourier transform is in progress.

The time performance of the Cascade is somewhat improved over the one-processor design. Input values enter the leftmost processor at the rate of one per multiply-add step. An entire problem instance is thus loaded in $O(N \log N)$ time units. It is easy to see that the Cascade can start processing a new problem instance as soon as the previous one has been completely loaded, so its "pipelined time" performance is $T = O(N \log N)$.

One awkward feature of the Cascade is that it produces its output values in bit-reversed order. Formally, their order is derived from the natural left-to-right indexing (0 to $N-1$) by reversing the bits in each index value, so that the least significant bit is interpreted as the most significant bit. The last few lines of Figure 71 perform this bit-reversal, but they cannot be performed on the circuit described thus far. If natural ordering is desired, a processor should be attached to the output end of the Cascade. If this processor has $N$ words of RAM storage, a simple algorithm will allow it to reorder the outputs of the Cascade as rapidly as they are produced.

### 6.2.7. The FFT Network

One of the most obvious ways of implementing the FFT in hardware is to provide one multiply-add cell for each execution of statement 7 in the algorithm of Figure 70. (The algorithm of Figure 71 might also be used, but, as noted in the previous subsection, its multiply-add computation is a little more complex.) Each cell is provided with a register holding its particular value of $z^j$. Since

---

† Note that $z^i = z^j$ whenever the $b$-th bit of $i$ is 0, since $z$ is a $2p$-th root of unity. Of course, exact equality obtains only when exact arithmetic is employed. This is easy to arrange in a number-theoretic transform. When round-off errors cannot be avoided, for example in a complex-valued transform, it is probably best to use a conditional transfer to reset $z^j$ to 1 whenever $j = 0$.

statement 7 is executed $N/2 \log N$ times, a total of $N/2 \log N$ multiply-add cells are required for this "full parallelization" of the FFT.

One possible layout for the cells in an FFT network is to have $\log N$ rows of $N/2$ cells each, as shown in Figure 73. Each row of cells in the FFT network corresponds to an entire iteration of the "FOR $b$" loop of the algorithm of Figure 70. The interconnections between the rows are defined by the way that the array $x$ is accessed. The reader is invited to check that each multiply-add cell in Figure 73 corresponds to an execution of statement 7 in Figure 70 for the case $N=8$.

Note that the inputs to the FFT network are in "bit-shuffled" order and its outputs are in "bit-reversed" order. This seems to minimize the amount of area required for interconnecting the rows. Additional wiring may of course be added to place inputs and outputs in their natural, left-to-right order.

The interconnections of Figure 73 may be obtained from the following general scheme. Number the cells naturally: from 0 to $N/2-1$, from left to right. Then cell $i$ in the first row is connected to two cells in the second row: cell $i$ and cell $(i + N/4) \bmod N/2$. Cell $i$ in the second row is connected to cells $i$ and $\lfloor i/(N/4) \rfloor + ((i + N/8) \bmod N/4)$ in the third row. Cell $i$ in the $k$th row (where $k=1,2,\ldots \log N - 1$) is connected to two cells in the $(k+1)$-th row: cell $i$ and cell $\lfloor i/(N/2^k) \rfloor + ((i + N/2^{k+1}) \bmod N/2^k)$. Another way of describing this "butterfly" interconnection pattern is to say that a cell on the $k$th row connects to the two cells on the next row whose indices differ at most in their $k$th most significant bit. (The interconnections between rows in an FFT network can also be laid out in the "perfect shuffle" pattern described in the next subsection. However, this seems to lead to a larger layout, if only by a constant factor.)

A careful study of Figure 73 and the preceding paragraph should convince the reader that $N/2$ horizontal tracks are necessary and sufficient for laying out the interconnections between the first two rows. Essentially, each cell in the first row has one "long" output wire that must cross the vertical midline of the diagram. This connection must be assigned a unique horizontal track to cross the midline. Once this is done, the rest of the wiring for that row is trivial, especially if the cells are "staggered" slightly as in Figure 73.

The connections between the second and third rows occupy only $N/4$ horizontal tracks. No wires cross the vertical midline of the diagram, but each of the $N/4$ cells on either side of the midline have a fairly long connection that takes up to half of a horizontal track.

In general, the connections emerging from the $k$th row ($k=0,1,\ldots \log N - 1$) occupy $N/2^{k+1}$ tracks. Straight vertical wires are used to connect cell $i$ in the $k$th row with cell $i$ in the $(k+1)$th row. The horizontal tracks are divided into $2^k$ equally-sized pieces, then individually assigned to the "long" connection from each cell.

Following the scheme outlined above, a total of $N-1$ horizontal tracks are required to lay out the inter-row connections. An additional $N$ horizontal tracks could be added above and below the FFT network to bring its inputs and outputs into natural order.

The number of vertical tracks in an FFT network depends strongly upon the width of the multiply-add cells. If these are set on end, so that each is $O(1)$ units tall and $O(\log N)$ units wide, then the entire network will fit into a rectangular region that is $O(N)$ units wide and $O(N)$ units tall. The height of the $\log N$ rows of multiply-add cells is asymptotically negligible.

The pipelined time performance of the FFT network is clearly $O(\log N)$ since a new problem instance can enter the network as soon as the previous one

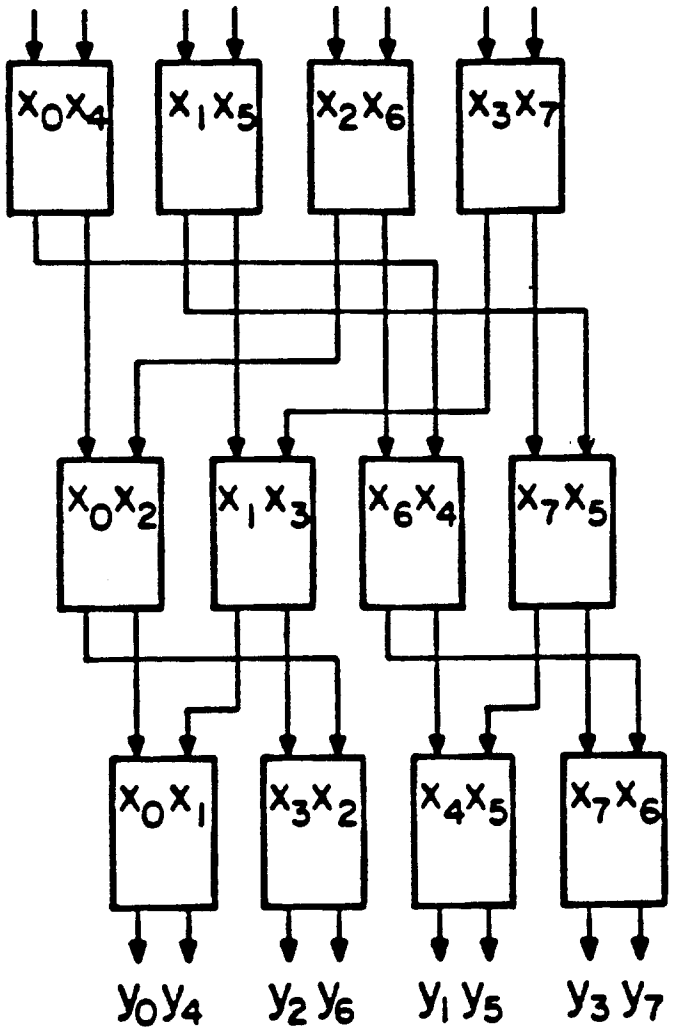



**Figure 73:** The FFT network for $N=8$.

has left the first row of multiply-add cells. The delay imposed by each row's multiply-add computation and long-wire drivers is $O(\log N)$, and there are $O(\log N)$ rows, so the total delay of the network is $O(\log^2 N)$.

Note that this layout of the FFT network must be optimal, for the circuit has an optimal area*time$^2$ performance of $O(N^2 \log^2 N)$. Any asymptotic improvement in the layout area would amount to a disproof of Vuillemin's optimality result [27].

### 6.2.8. The Perfect-Shuffle Implementation of the FFT

Over a decade ago, Stone [28] noted that the "perfect shuffle" interconnection pattern of $N/2$ multiply-add cells is perfectly suited for an FFT computation by decimation in time. Figure 74 shows the perfect shuffle network for the 8-element FFT, and figure 70 shows the appropriate version of the FFT algorithm.

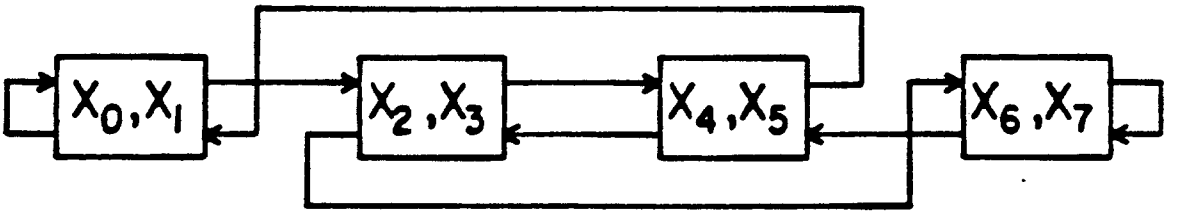



**Figure 74**: The perfect shuffle interconnections for $N=8$.

Each multiply-add cell in a perfect shuffle network is associated with two input values, $x_k$ and $x_{k+1}$. Here, $k$ is an even number in the range $0 \leq k < N-1$. A connection is provided from one of the outputs of the cell containing $x_k$ to one of the inputs of the cell containing $x_j$ if and only if $j = 2k \bmod N-1$. Note that this mapping of output indices onto input indices is one-to-one, and that it corresponds to an "end-around left shift" of the ($\log N$)-bit binary representation of $k$.

The computation of the FFT on the perfect shuffle network can now be described. First, the input values $x_k$ are loaded into their respective multiply-add cells. Then a multiply-add step is performed: each cell ships its original $x_k$ values out over its output lines, and computes new $x_k$ values according to Equation (1). It is not very obvious, but nonetheless it is true, that this corresponds to an entire iteration of the "FOR $b$" loop of Figure 70. For example, the left-most cell of Figure 73 computes new values for $x_0$ and $x_4$, having received the original value of the former from its own output line and the original value of the latter from the third cell. This is the computation required by step 7 of Figure 70, when $N=8$, $b=2$, $p=4$, $q=2$, $i=0$, $j=0$, and $k=0$.

The FFT computation proceeds in this fashion for $\log N$ parallel multiply-add steps. In each step, the cell containing the (updated) version of $x_k$ ships this value to the cell formerly containing the (updated) version of $x_{2k \bmod N-1}$. Each cell then performs a multiply-add computation, updating the two data values currently in its possession.

At the end of $\log N$ parallel multiply-add steps, each cell contains the final versions of its original data values. Unfortunately, the FFT computation of Figure 70 is not complete. The outputs $y$ are all available among the final $x$ values, but they appear in "bit-reversed" order. Additional circuitry is required to bring them into natural order, following steps 11-13 of Figure 70. The techniques of [29] could be employed in the design of reordering circuitry that would operate in $O(\log^2 N)$ time, without affecting the area performance of the perfect shuffle network.

The $\log N$ parallel multiply-add steps require a total of $O(\log^2 N)$ time. The data movement involved in each multiply-add step does not require any additional time, at least in an asymptotic sense. As will be seen below, the "shuffle" connections between cells are implemented as single wires carrying bit-serial data. Each wire is less than $O(N)$ units long, and each word has $O(\log N)$ bits, so that the data transmission time per step is the same as the multiplication time, $O(\log N)$ time units.

The total area of the perfect shuffle implementation is a bit harder to estimate. There are $N/2$ multiply-add cells, each occupying $O(\log N)$ are. However, the best embedding known for the shuffle interconnections takes up $O(N^2/\log^2 N)$ area [30]. It is easy to see that no better embedding is possible,

since otherwise the perfect shuffle circuit would have an impossibly good area*time$^2$ performance.

### 6.2.9. The CCC Network

The cube-connected-cycles (CCC) interconnection for $N$ cells is capable of performing an $N$-element FFT in $O(\log N)$ multiply-add steps [31]. Using the multiply-add cell of the previous constructions, the complete FFT takes $O(\log^2 N)$ time.

The CCC network is very closely related to the FFT network. In fact, a CCC network is just an FFT network with "end-around" connections between the first and last rows. For this reason, CCC networks do not exist for all $N$, only for those $N$ of the form $(K/2)*(\log K)$ for some integer $K$. Figure 75 illustrates the CCC network for $N=8$. It is derived from the 4-element FFT network with "split cells": each cell handles one element of the input vector $\vec{x}$, instead of two as in the FFT network of Figure 73. (The reader is invited to redraw Figure 75, combining the cells linked by horizontal data paths. The resulting graph should be isomorphic to a "butterfly" whose outputs have been fed back into its inputs.)

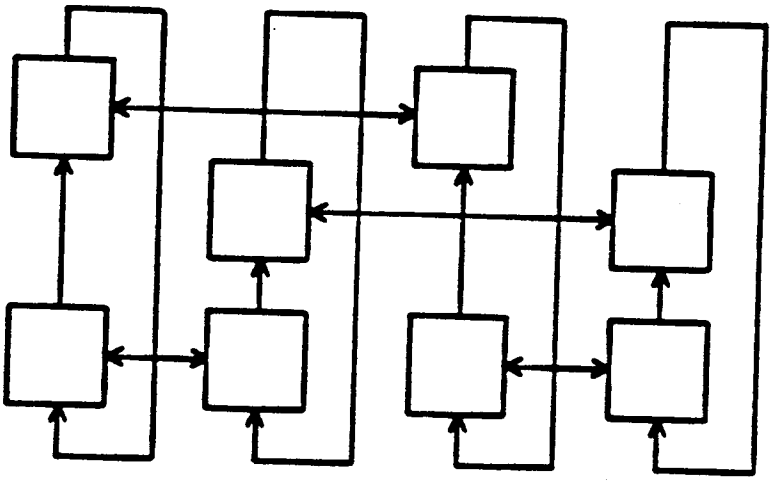


**Figure 75**: The CCC network for $N=8$.

The CCC network is somewhat smaller than the FFT network, since it uses only $N$ cells to solve an $N$-element problem instead of the FFT network's $(N/2)*(\log N)$ cells. Furthermore, the CCC's interconnections can be embedded in only $O(N^2/\log^2 N)$ area [31]. This is an optimal embedding, for the combined area*time$^2$ performance is within a constant factor of the limit, $\Omega(N^2 \log^2 N)$.

It is rather difficult to describe the data routing pattern during the computation of a Fourier transform on a CCC, although the basic approach is similar to that taken on the perfect shuffle network. Each of the log $N$ multiply-add steps is preceded and followed by a routing step. These routing steps take $O(\log N)$ time each, for they move $O(1)$ words over each intercellular connection. Thus the total time spent in routing data does not dominate the time spent on multiply-add computations.

### 6.2.10. The Mesh Implementation

A square Mesh of $N$ processors is shown in Figure 76. It consists of approximately $\sqrt{N}$ rows of $\sqrt{N}$ processors each, fitted with word-parallel interconnections. It is thus essentially the ILLIAC IV architecture, with the difference that each processor in the Mesh is capable of running its own program. (A closer approximation to the ILLIAC IV would have $N$ multiply-add cells, each deriving control signals from a central processor.)

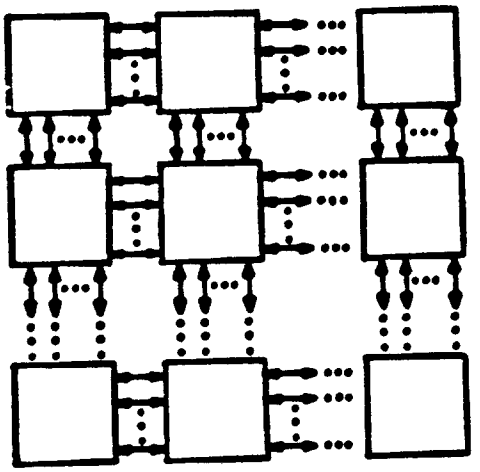


**Figure 76:** The Mesh of $N$ processors, formed of $2^{\lceil (\log N)/2 \rceil}$ rows and $2^{\lfloor (\log N)/2 \rfloor}$ columns.

The total area of the Mesh is $O(N \log^2 N)$, since there are $N$ processors each of $O(\log^2 N)$ area. The processors should each be laid out with a square aspect ratio, so that the $O(\log N)$ wires in each word-parallel data path do not add to the asymptotic area of the layout. Note that it takes $O(\text{loglog } N)$ time to send a word of data from one processor to its neighbor, since the interprocessor wires are $O(\log N)$ in length.

Stevens [32] appears to have been the first to point out that the Mesh can perform an $N$-element FFT in $\log N$ steps of computation. Each "step" consists of an entire iteration of the *FOR* $b$ loop of Figure 70. Each processor in the Mesh performs the loop computation for one value of the index variable $k$. The total amount of data movement during the FFT can be minimized by making an appropriate assignment of index values $k$ to individual Mesh processors. It turns out that a fairly good choice is obtained from the natural row-major ordering (0 to $N-1$) of the Mesh. Processor $k$ is then the "home" of the variable $x_k$.

(Another, more intuitive way of visualizing the computation of the FFT on the Mesh is to view the latter as a time-multiplexed version of the FFT network. During each step, $N/2$ of the Mesh's processors take on the role of the $N/2$ cells in one row of the FFT network. The wires connecting the rows of the FFT network are simulated by data movement among the processors of the Mesh.)

An iteration of the *FOR* $b$ loop of Figure 70 can now be described. Each mesh processor examines the $b$-th bit of *reverse*$(k)$ to decide if it will perform the computation of statement 7. (For example when $b=0$, $n2=1$ so that only the even-numbered processors will perform statement 7.) Next, each processor that will *not* perform statement 7 sends its current value of $x_k$ to processor $k+2^b$. (For example, when $b=0$, each odd-numbered processor sends its $x$ value to the processor on its left.) Statement 7 is then executed, and finally the updated $x_k$ values are returned to their "home" processors.

When $b = \log N - 1$, the data movement required before statement 7 can be visualized by "sliding" all the $x_k$ values in the bottom half of the Mesh up to the top half of the Mesh. In this way, processor 0 receives the current value of $x_{N/2}$, processor 1 receives the value of $x_{N/2+1}$, etc. This particular data movement will be called a "distance-N/2 route." In general, a distance-$2^b$ route must be performed both before and after each execution of statement 7.

The time required by a distance-$2^b$ route depends, of course, on the value of $b$. When $b = 0$ or $2^{\lceil (\log N)/2 \rceil}$, all data movement is between nearest neighbors (horizontal or vertical) in the Mesh. As mentioned above, this takes only $O(\text{loglog } N)$ time.

When $b = 2^{\lceil (\log N)/2 \rceil}$ or $\log N - 1$, it would seem that $O(\sqrt{N}\ \text{loglog } N)$ time is required for a distance-$2^b$ route. Each data element must ripple through about $\sqrt{N}/2$ processors. However, this result may be improved by using the "high-power" inputs on the long-wire drivers on the interprocessor data paths (see Assumption 1d). Once the bits in a data element have been amplified enough to be sent to a neighboring processor, only one more stage of amplification is necessary to send these bits on to the next processor. Since the amplifier stages in a long-wire driver are individually clocked, all data elements in a routing operation may "slide" toward their destination simultaneously, moving by one processor-processor distance every time unit. The total time taken by a distance-$2^b$ routing is thus easily seen to be $(2^b \bmod \lceil (\log N)/2 \rceil) + O(\text{loglog } N)$.

The total time taken by all routings in a complete FFT computation is bounded by $O(\sqrt{N})$. Essentially, this is the sum of a geometric series whose largest term is the time taken by the longest routing operation, $O(\sqrt{N})$. The time performance of the Mesh design is thus $O(\sqrt{N})$. At least asymptotically, the $O(\log^2 N)$ time required for the multiply-add computations is insignificant compared to the time required for the routing operations.

Three aspects of the Mesh implementation deserve further attention. First of all, the individual processors are expected to come up with their own $z^j$ values, as they execute statement 7 of Figure 70. This is not difficult to arrange: each processor has $O(\log^2 N)$ bits of program storage, so it can easily perform a table look-up to obtain the required constants. One constant is needed for each processor, for each value of $b$.

Secondly, the algorithm described computes the $y$ values in bit-reversed order (relative to the natural row-major ordering of the Mesh). If the outputs are desired in natural order, another $O(\sqrt{N})$ routing operations are required [29], and the individual processors' programs become a bit more complicated.

One final note: the Mesh implementation, as described, is area*time$^2$ optimal. A slightly less efficient, but possibly more practical design has been suggested. Instead of using word-parallel buses between $N$ processors in a mesh, one might provide bit-serial buses between $N$ cells in a mesh. Now the best possible time performance is constrained by the bit-serial buses to be no better than $O(\sqrt{N}\ \log N)$. Similarly, the area could be reduced to as little as $O(N \log N)$. However, it will be a bit tricky to attain these performance figures. There is not enough area to store each cell's $z^j$ values locally, so these values must be computed "on the fly" in (hopefully) only few extra multiplications. This seems to be impossible to accomplish directly. One solution to this difficulty is to have the cells exchange $z^j$ values as well as $x_k$ values. The bit-serial approach is thus inherently slower both in routing time and in the number of necessary multiplications. On the other hand, the word-parallel approach has wider buses and perhaps larger look-up tables, so that it takes up somewhat more area.

## 7. CONCLUSIONS

We have investigated the problem of implementing Discrete Fourier Transform processors in VLSI on many different levels. The original idea of using Charge Transfer Devices to implement the building blocks of these processors was discarded when it became clear that other technologies such as CMOS and NMOS had advantages in terms of power dissipation, speed, compactness, and complexity of handling the necessary crossovers.

On an organizational level, the pipeline structure of Despain was shown to map very well into a VLSI structure where it is desired to build as few different chip types as possible while allowing the construction of processors which are capable of computing transforms of arbitrary size. An extension of this structure to allow higher throughput, lower latency processors to be built also has the same desirable features due to the incredible flexibility which is inherent in the DFT computation. This extension allows one to trade off speed and cost in any way desired, especially for large transform sizes where the communication requirements do not force one to use less than the maximum number of transistors available on a chip.

The higher radix structures which were derived are to be preferred in general since they require less full CORDIC rotators. This reduces latency and hardware costs, especially where the operations needed to realize the higher radix butterfly are simple and can utilize partial multiplier circuits.

Several chips which could be used to build processors in this style were designed. The butterfly and CORDIC chips could be used to put together arbitrarily large radix 2 or radix 4 (due to the built-in $\frac{\pi}{2}$ rotator of the butterfly) transforms. The bit-skewed format turned out to have unfortunate properties for the CORDIC-style circuits, but worked well for the butterfly modules. The real and imaginary multiplexing was necessary at the time due to pin limitations, but also avoided crossovers internal to the chips. The $\frac{\pi}{16}$ rotator was to be used along with a $\frac{\pi}{8}$ rotator in the construction of a 16-point DFT processor, but improvements in circuit density allowed us to build the more general CORDIC circuit which could replace them both. Also, various experimental chips such as the root-3 partial multiplier and the 4 bit-slice butterfly which were fabricated early on verified the ease of mapping the algorithms into silicon. The barrel shifter module was successful and could be used in a less expensive, lower performance, iterative CORDIC design.

Regular, pipeline organizations which drew on the work of Winograd to reduce multiplies in DFT computations were developed. These structures are quite compact, but would require the development of many different chip types if very large VLSI processors were to be built.

The problem of latency, already addressed from an organizational viewpoint, has also been attacked by the development of new carry lookahead circuits which are faster and cheaper than any others which have been seen in the published literature. The new circuits also allow one to realize the inherent speed advantages that exist in high fan-in over low fan-in gates.

Finally, new work in complexity theory using a model which takes into account the special characteristics of VLSI has allowed a comparison of different styles of DFT processor design free from the details of a specific implementation.

**References**


1. A. M. Despain , "Fourier Transform Computers Using CORDIC Iterations," *IEEE Trans. Comput.* **C-23** pp. 993-1001 (Oct. 1974).
2. A. M. Despain , "Very Fast Fourier Transform Algorithms for Hardware Implementation," *IEEE Trans. Comput.* **C-28** pp. 333-341 (May 1979).
3. C. H. Séquin and M. F. Tompsett, *Charge Transfer Devices*, Academic Press Inc. (1975).
4. J. W. Cooley and J. W. Tukey , "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.* **19** pp. 297-301 (Apr. 1965 ).
5. J. J. Good , "The Interaction Algorithm and Practical Fourier Analysis," *J. Royal Statist. Soc.* **B-20** pp. 361-372 (1958).
6. S. Winograd, "On Computing the Discrete Fourier Transform," *Proc. Nat. Acad. Sci. U.S.* **73** pp. 1005-1006 (Apr. 1976).
7. C. Thompson, *A Complexity Theory for VLSI*, Ph.D. Thesis August 1980..
8. J. E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electron. Comput.* **EC-8** pp. 330-334 (Sept. 1959).
9. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison Wesley (1980).
10. B. Gold and T. Bially, "Parallelism in Fast Fourier Transform Hardware," *IEEE Trans. Audio Electroacoust.* **AU-21** pp. 5-16 (Feb. 1973).
11. D. S. Parker, "Notes on Shuffle/Exchange-Type Switching Networks," *IEEE Trans. Comput.* **C-29** pp. 213-222 (Mar. 1980).
12. C. M. Rader, "Discrete Fourier Transforms When the Number of Data Samples is Prime," *Proc. IEEE* **56**(6) pp. pp. 1107-1108 (June 1968).
13. D. P. Kolba and T. W. Parks, "A Prime Factor FFT Algorithm Using High-Speed Convolution," *IEEE Trans. Acoust. Speech. Sig. Proc.* **ASSP-25** pp. 281-294 (Aug. 1977).
14. J. H. McClellan and C. M. Rader, *Number Theory in Digital Signal Processing*, Prentice-Hall, Englewood Cliffs (1979).
15. T. G. Stockham, "High Speed Convolution and Correlation," pp. 229-233 in *Spring Joint Comput Conf., AFIPS Conf. Proc.*, Spartan, Washington, D.C. (1966).
16. S. Winograd, "On Computing the Discrete Fourier Transform," *Math. Comput.* **32**(141) pp. 175-199 (Jan. 1978).
17. R. A. Allen, R. J. Handy, and J. E. Sandor, "Charge Coupled Devices in Digital LSI," *IEDM 1976, Tech. Digest*, (1976).
18. P. M. Dobrowolski, "Evaluation of Approaches to LSI Implementation of the Butterfly Unit for a Modular, Pipelined Fast Fourier Transform Processor," *Masters Report*, U.C. Berkeley, (June 1979).
19. J.S. Walther, "A Unified Algorithm for Elementary Functions," *AFIPS Proc. 1971 Spring Joint Comput. Conf.* **38** pp. 379-385. AFIPS Press , (1971).
20. Ladner and Fischer, "Parallel Prefix Computation," *JACM* **27**(4) pp. 831-838 (Oct. 1980).
21. A. M. Despain, "General Fan-in Prefix Calculations," *UCB-ERL Report*, (Oct. 1981).

22. F. Fich, "The Parallel Prefix Problem and Applications to Fast Adders," *UCB-ERL Report*, (Dec. 1981).

23. D. J. Kuck, *The Structure of Computers and Computations*, John Wiley and Sons (1978).

24. C. Thompson, *Fourier Transforms in VLSI*.

25. J. Savage, "Area-Time Tradeoffs for Matrix Multiplication and Related Problems in VLSI Models," in *TR-CS-50*, , Dept. of Computer Science, Brown University (August 1979).

26. W. Cochran , J. Cooley, and *et al.*, "What is the Fast Fourier Transform?," *IEEE Trans. on Audio and Electro.* **AU-15**(2) pp. 45-55 (June 1967).

27. J. Vuillemin, "A Combinatorial Limit to the Computing Power of VLSI Circuits," pp. 294-300 in *Proc. 21st Symp. on the Foundations of Computer Science*, IEEE Computer Society (October 1980).

28. H. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Comput.* **C-20**(2) pp. 153-161 (February 1971).

29. C. Thompson, "Generalized Connection Networks for Parallel Processor Intercommunication," *IEEE Trans. Comput.* **C-27**(12) pp. 1119-1125 (December 1978.).

30. F.T. Leighton, *Layouts for the Shuffle-Exchange Graphs and Lower Bound Techniques for VLSI*, Ph.D. Dissertation August 1981.

31. F. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," pp. 140-147 in *Proc. 20th Annual Symp. on Foundations of Computer Science*, IEEE Computer Society (October 1979).

32. J. Stevens, "A Fast Fourier Transform Subroutine for Illiac IV ," in *Technical Report*, , Center for Advanced Computation, Illinois (1971).