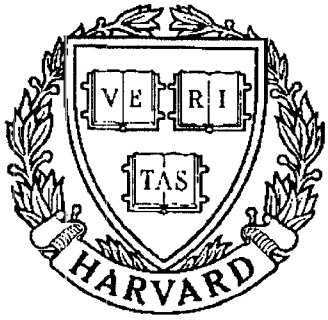
THESIS REPORT
Master's Degree

S Y S T E M S
R E S E A R C H
C E N T E R



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

VLSI Design of Discrete Fourier Transform Processors

by T.A. Goodrich
Advisor: J. Já Já

M.S. 91-3

ABSTRACT

Title of Thesis: VLSI DESIGN OF DISCRETE FOURIER TRANSFORM PROCESSORS

Name of degree candidate: Todd Anthony Goodrich

Degree and Year: Master of Science, 1991

Thesis directed by: Professor Joseph F. Ja'Ja', Electrical Engineering

A bit-serial cell library is presented which can been used to rapidly implement discrete Fourier transform algorithms in VLSI circuit technology. The design methodology employs systolic summation arrays and multiplier arrays which are bit-serial and fully pipelined. To demonstrate the utility and performance of this cell library, an 8-point discrete Fourier transform (DFT) processor has been designed and implemented as a VLSI chip which operates at 50 MHz and has a throughput of 2.9 million complex 8-point 16-bit DFT's per second.

VLSI DESIGN OF DISCRETE FOURIER TRANSFORM PROCESSORS

by

Todd Anthony Goodrich

Thesis submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1991

Advisory Committee:

Professor Joseph Ja'Ja', Chairman/Advisor
Professor Bernard Menezes
Professor Robert Newcomb

DEDICATION

To my loving wife Lisa

ACKNOWLEDGEMENTS

I would like to sincerely thank my graduate advisor, Dr. Joseph F. Ja'Ja', for his guidance and support, and for providing me with a thesis project which was exactly matched to my interests.

I would also like to thank Ravi Kolagotla and Shu-Sun Yu of the University of Maryland VLSI Design Lab for their excellent technical contributions to this thesis. Ravi's previous work in VLSI design provided a foundation upon which I based most of my VLSI layouts. In addition, his support as the CAD software manager was truly outstanding.

Finally, I would like to thank the Department of Defense and my parents, Richard D. and Mary E. Goodrich, for their generous financial support of my college career.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 - The Discrete Fourier Transform

Introduction

This paper is concerned with the design of discrete Fourier transform (DFT) processors in VLSI circuit technology. The DFT operation is very frequently used in digital signal processing applications to compute the frequency content of sampled signals. Many of these applications are real-time systems which require high speed DFT computations on continuous streams of input data.

In this chapter the DFT is defined and developed into a form which can be easily implemented in VLSI circuit technology, as shown in previous papers on DFT design[1,2,3]. Subsequent chapters describe exactly how to construct high performance DFT processors using a new library of bit-serial standard cells and new software which automatically creates VLSI layouts constructed from these cells.

Definition

The discrete Fourier transform (DFT) of a sequence of N complex numbers, x_n, is defined as another sequence of N complex numbers, X_k, where

$$X_k = \frac{1}{N}\sum_{n=0}^{N-1} x_n \cdot e^{-j\frac{2\pi}{N}kn}, \text{ for } k = 0, 1, 2, \ldots, N-1 \qquad \text{(Eq 1)}$$

Matrix Representation

The DFT can be represented in matrix form. Arrange the input and output sequences as column vectors,

$$\vec{x} = \left[x_0\, x_1\, x_2 \ldots x_{N-1}\right]^T \qquad \text{(Eq 2)}$$

$$\vec{X} = \left[X_0\, X_1\, X_2 \ldots X_{N-1}\right]^T \qquad \text{(Eq 3)}$$

Note that the superscript T denotes the matrix transpose operation.

Now (Eq 1) can be written as

$$\vec{X} = \overline{D} \cdot \vec{x} \qquad \text{(Eq 4)}$$

where $\overline{D}$ is a $N \times N$ matrix and

$$D_{k,n} = \frac{1}{N} \cdot e^{-j\frac{2\pi}{N}kn} \qquad \text{(Eq 5)}$$

is the element in the k-th row and n-th column of $\overline{D}$.

Small-N Algorithms

For N=2, 3, 4, 5, 7, 8, 9, and 16, it can be shown[4] that the $\overline{D}$ matrix can be factored into three sparse matrices

$$\overline{D} = \overline{S} \cdot \overline{C} \cdot \overline{T} \qquad \text{(Eq 6)}$$

where $\overline{D}$ is a $N \times N$ matrix, $\overline{S}$ is a $N \times \delta$ matrix, $\overline{C}$ is a $\delta \times \delta$ matrix, and $\overline{T}$ is a $\delta \times N$ matrix. These factored representations are known as small-N DFT algorithms.

Small-N algorithms have the following properties[4]:

- Every element of the $\overline{S}$ and $\overline{T}$ matrices is a member of the set $\{0, 1, -1\}$
- The $\overline{C}$ matrix is always diagonal and has elements which are either real or imaginary

Therefore, small-N algorithms can be computed by applying $\overline{T}$ to accomplish input summations, then $\overline{C}$ to accomplish all multiplications, and then $\overline{S}$ to accomplish output summations, i.e.

$$\vec{X} = \overline{S} \cdot (\overline{C} \cdot (\overline{T} \cdot \vec{x})) \qquad \text{(Eq 7)}$$

Example:

For N = 3, the DFT can be written as

$$\bar{D} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{\cos(\phi) - 1}{3} & 0 \\ 0 & 0 & \frac{j\sin(\phi)}{3} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & -1 & 1 \end{bmatrix} \quad \text{(Eq 8)}$$

where $\phi \equiv \frac{2\pi}{3}$.

Large-*N* Algorithms from Small-*N* Algorithms

Small-*N* algorithms can be combined into large-*N* algorithms using their Kronecker product[4]. The Kronecker product of two matrices, is defined as

$$\bar{A} \otimes \bar{B} = \begin{bmatrix} A_{1,1} \cdot \bar{B} & A_{1,2} \cdot \bar{B} & \dots & A_{1,n} \cdot \bar{B} \\ A_{2,1} \cdot \bar{B} & A_{2,2} \cdot \bar{B} & \dots & A_{2,n} \cdot \bar{B} \\ \dots & \dots & \dots & \dots \\ A_{m,1} \cdot \bar{B} & A_{m,2} \cdot \bar{B} & \dots & A_{m,n} \cdot \bar{B} \end{bmatrix} \quad \text{(Eq 9)}$$

where $\bar{A}$ is a $m \times n$ matrix. If $\bar{B}$ is a $k \times l$ matrix, then $\bar{A} \otimes \bar{B}$ will be a $mk \times nl$ matrix.

Let $\bar{D}_L, \dots, \bar{D}_2, \bar{D}_1$ be small-*N* DFT algorithms with naturally ordered indices and with dimensions $N_L \times N_L, \dots, N_2 \times N_2$, and $N_1 \times N_1$, respectively. Then,

$$\bar{D}_N = \bar{D}_L \otimes \dots \otimes \bar{D}_2 \otimes \bar{D}_1 \quad \text{(Eq 10)}$$

is a large-*N* DFT algorithm, which is a $N \times N$ matrix, where $N = N_L \cdot \dots \cdot N_2 \cdot N_1$.

Each small-*N* DFT algorithm of dimension N_i can be put in the form $\bar{D}_i = \bar{S}_i \cdot \bar{C}_i \cdot \bar{T}_i$ so that

$$\bar{D}_N = (\bar{S}_L \bar{C}_L \bar{T}_L) \otimes \dots \otimes (\bar{S}_2 \bar{C}_2 \bar{T}_2) \otimes (\bar{S}_1 \bar{C}_1 \bar{T}_1) \quad \text{(Eq 11)}$$

After repeatedly using the relationship that $(\bar{A}\bar{B}) \otimes (\bar{C}\bar{D}) = (\bar{A} \otimes \bar{C})\ (\bar{B} \otimes \bar{D})$, we find

$$\bar{D}_N = (\bar{S}_L \otimes \ldots \otimes \bar{S}_2 \otimes \bar{S}_1)\ (\bar{C}_L \otimes \ldots \otimes \bar{C}_2 \otimes \bar{C}_1)\ (\bar{T}_L \otimes \ldots \otimes \bar{T}_2 \otimes \bar{T}_1) \quad \text{(Eq 12)}$$

Thus, large-N algorithms can be factored in the same way as small-N algorithms, so that they can be evaluated by using $(\bar{T}_L \otimes \ldots \otimes \bar{T}_2 \otimes \bar{T}_1)$ to compute input summations, followed by $(\bar{C}_L \otimes \ldots \otimes \bar{C}_2 \otimes \bar{C}_1)$ to compute all multiplications, followed by $(\bar{S}_L \otimes \ldots \otimes \bar{S}_2 \otimes \bar{S}_1)$ to compute output summations.

Summary

DFT algorithms of arbitrarily large size can be represented in the factored form $\bar{D} = \bar{S} \cdot \bar{C} \cdot \bar{T}$. In the following chapters it will be shown that this factored form can be very easily implemented in VLSI circuit technology.

Chapter 2 - VLSI Design Strategy

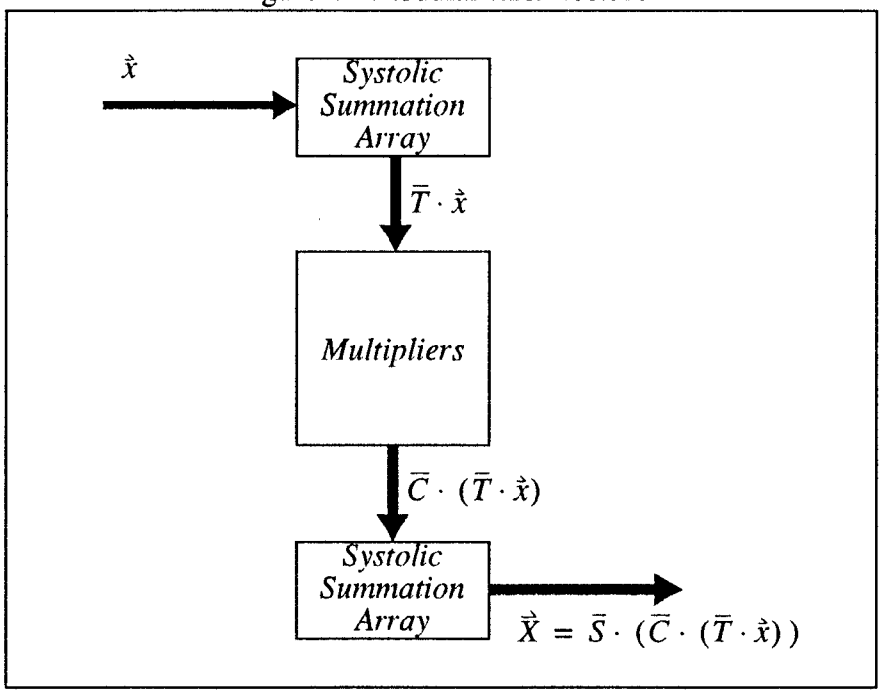
Basic Architecture

By taking advantage of the factored form of DFT algorithms, i.e

$$\vec{X} = \bar{S} \cdot \bar{C} \cdot \bar{T} \cdot \vec{x} \qquad \text{(Eq 13)}$$

DFT processors can be designed in a modular architecture[1,2,3] as shown in Figure 1.

Figure 1 - Modular Architecture



The factored form allows the computational architecture to be decomposed into three modules. The first module is a systolic summation array which computes the input summations corresponding to $\bar{T} \cdot \vec{x}$. The second module is an array of multipliers which perform the scaling corresponding to $\bar{C} \cdot (\bar{T} \cdot \vec{x})$. The third module is yet another systolic summation array which computes the output summations corresponding to $\bar{S} \cdot (\bar{C} \cdot (\bar{T} \cdot \vec{x}))$.

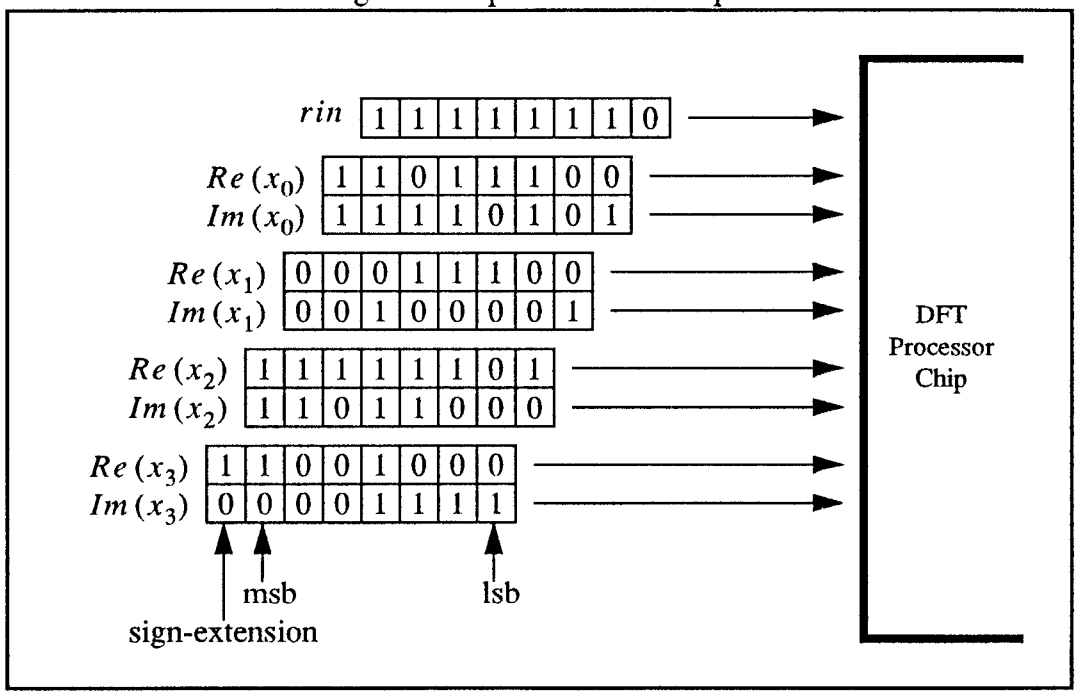
It will be shown that by using bit-serial arithmetic techniques, this DFT architecture can be implemented as a VLSI design which is very dense, simple to layout, fully-

pipelined for high performance, and requires only a single control signal.

Input Data and Control Specification

The input data, $\vec{x}$, consists of N complex numbers. Each complex number will be represented by two signed integers corresponding to the real and imaginary parts of the complex number. Let each integer be a p-bit two's-complement binary number plus a one-bit sign-extension. Figure 2 shows an example (for $N = 4, p = 7$) of data and control flow entering a DFT processor.

Figure 2 - Input Format Example



The data signals enter the DFT processor in a bit-serial fashion, with the least-significant-bit (lsb) entering first and the most-significant-bit (msb) entering last, followed immediately by the sign-extension bit.

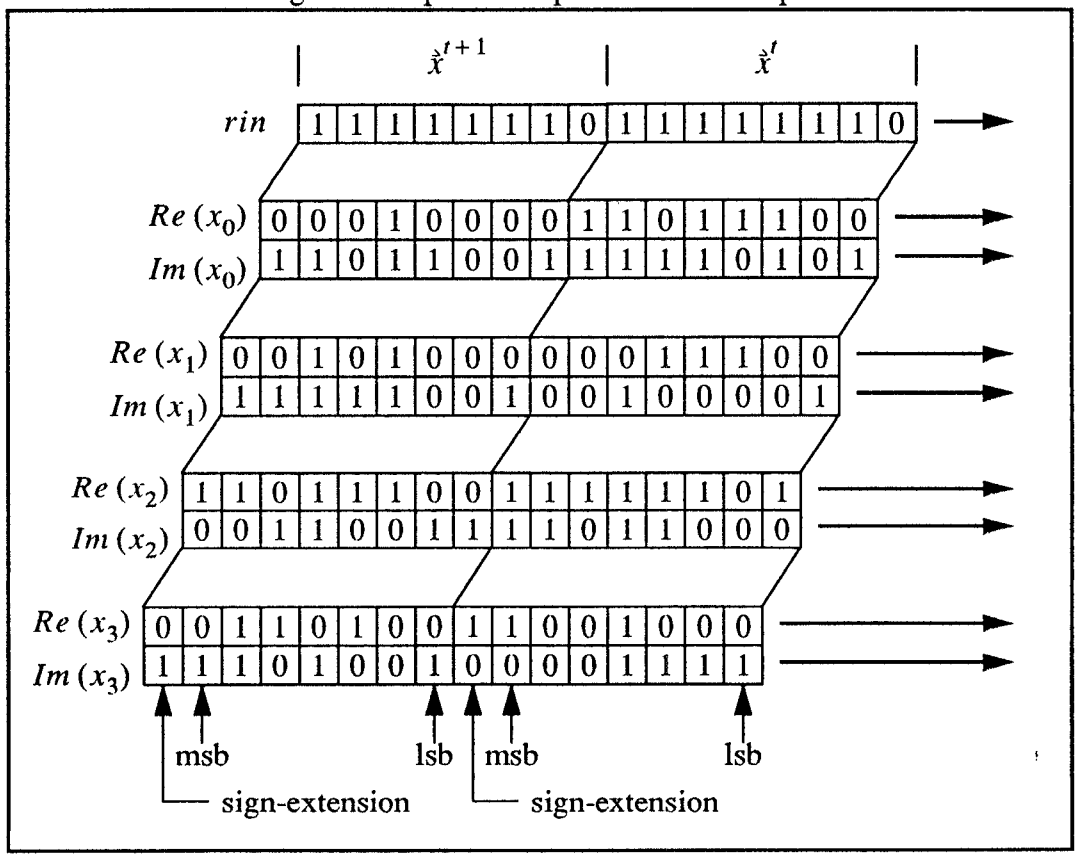
A single control signal, *rin*, enters the processor first with a leading zero. This leading zero alerts the processor that a new DFT computation will begin on the next clock cycle.

On the next cycle, the input data begins to arrive, but, due to the systolic nature of the DFT processor, each successive input data point, x_i, must be delayed by one clock cycle relative to the data point before it, x_{i-1}, as shown in Figure 2.

Pipelined Input Format

The DFT processor design will be fully pipelined so that multiple DFT's can be computed back-to-back by appending the next set of input data, $\vec{x}^{t+1}$, directly behind the current set of input data, $\vec{x}^{t}$, as shown in Figure 3.

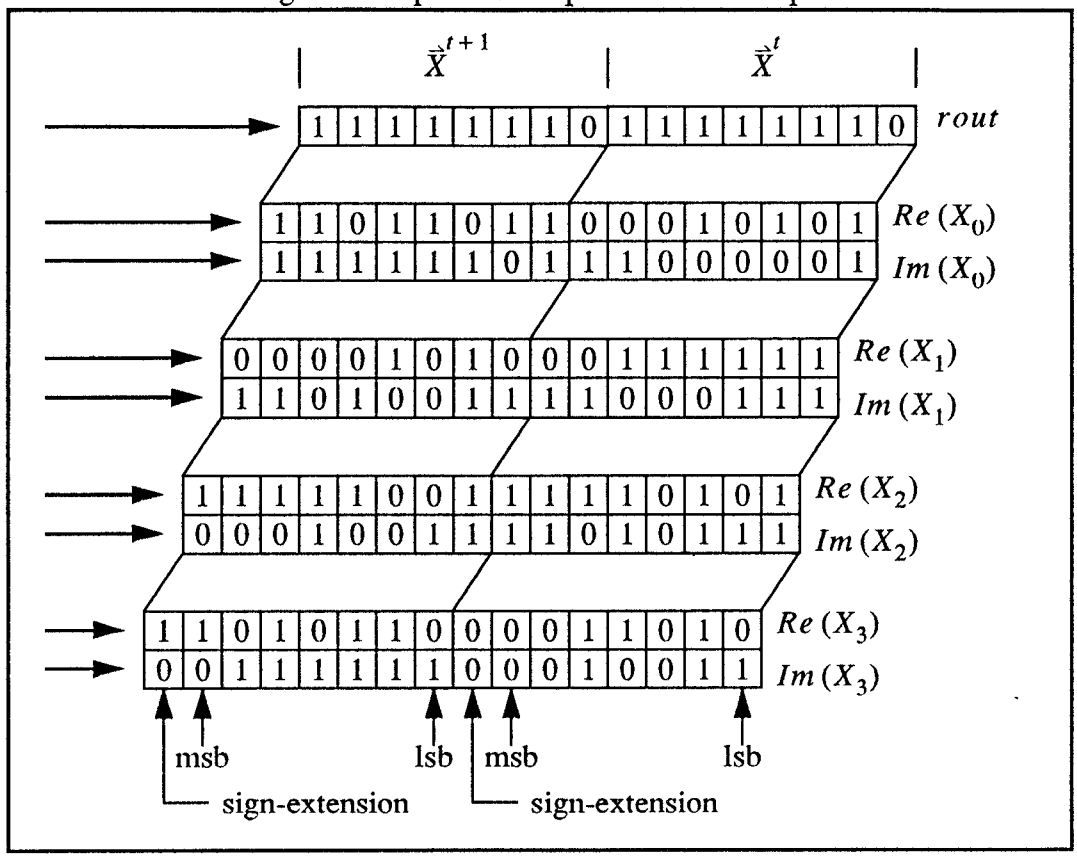
Figure 3 - Pipelined Input Format Example



Output Data and Control Specification

The computed DFT results, $\vec{X}^t = DFT(\vec{x}^t)$, will emerge from the output of the DFT processor in the exact same format as the input, as shown in Figure 4.

Figure 4 - Pipelined Output Format Example

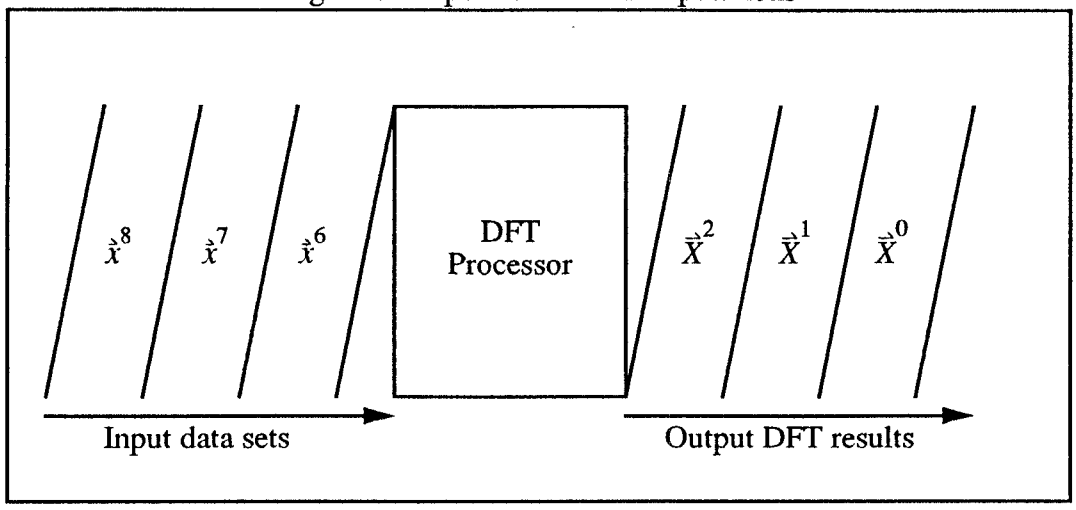


The results will exit the DFT processor in a bit-serial fashion, with the lsb exiting first and the msb exiting last, followed immediately by the sign-extension bit.

A single output control signal, *rout*, exits the processor first with a leading zero. This indicates that a new DFT result will begin to emerge from the processor on the next clock cycle.

The overall view of data flow is shown in Figure 5. Notice that the figure correctly indicates that the computation of DFT's numbered $\vec{X}^3$, $\vec{X}^4$, and $\vec{X}^5$ are concurrently in progress. This is possible since the DFT processor design is fully pipelined.

Figure 5 - Pipelined DFT Computations

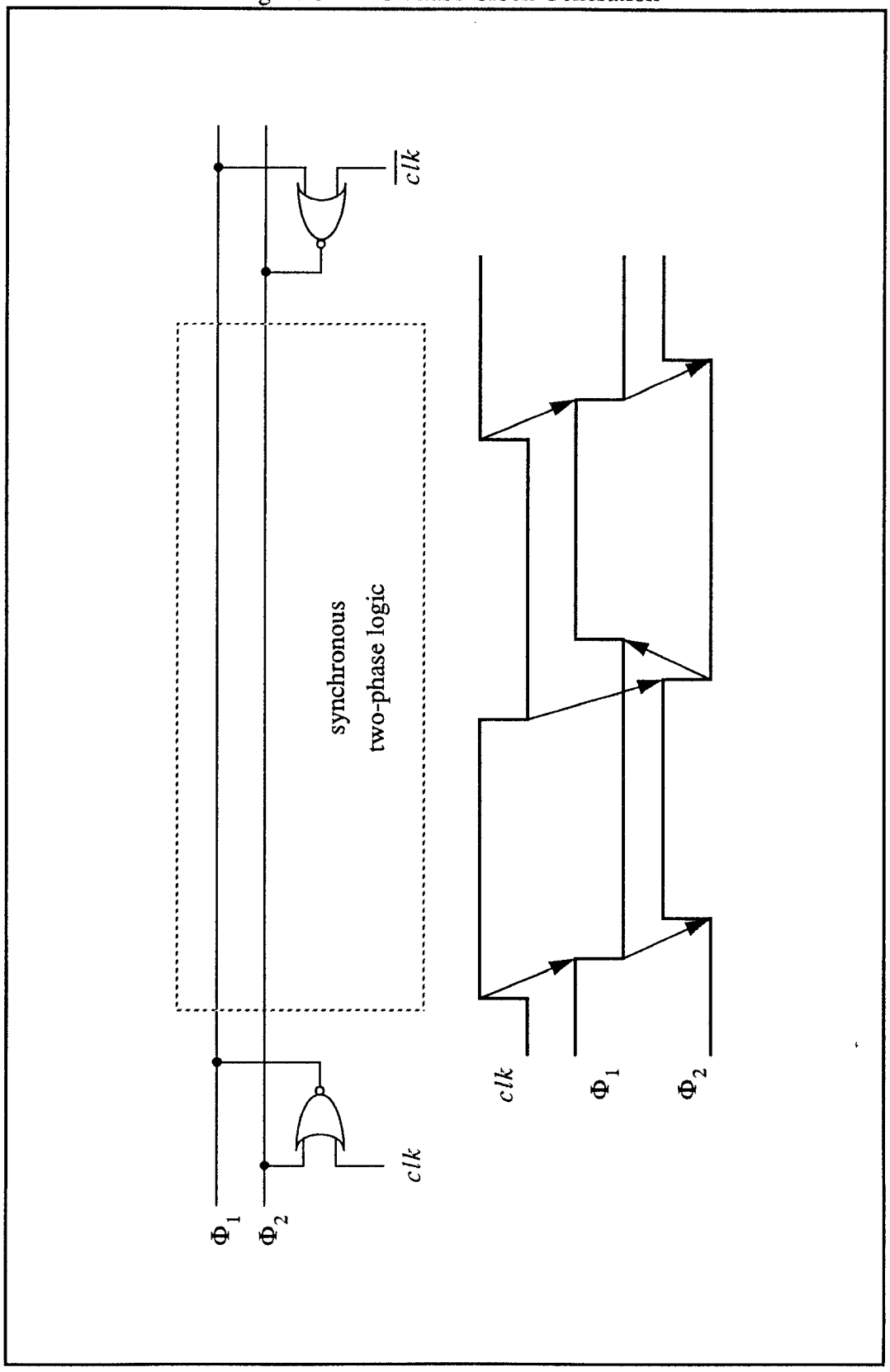


Clocking Methodology

The DFT processor will be designed to accept a single-phase clock. The DFT processor will be considered a positive-edge triggered device. Internally to the processor, the single-phase input clock will be decomposed[5] into a two-phase non-overlapping clock with components named Φ_1 and Φ_2 using the circuitry shown in Figure 6.

The generation of the two-phase clock is accomplished using self-timed circuitry which actually uses clock-skew to its advantage to guarantee that the two phases of the clock will be non-overlapping.

Figure 6 - Two-Phase Clock Generation



Chapter 3 - Summation Array Design

Problem Definition

The DFT architecture shown in Figure 1 requires the design of two summation arrays:

- The first summation array must compute $\bar{T} \cdot \vec{x}$ where $\bar{T}$ is a $\delta \times N$ matrix whose elements are all either -1, +1, or 0, and where $\vec{x}$ is a column vector of N complex numbers.
- The second summation array must compute $\bar{S} \cdot (\bar{C} \cdot \bar{T} \cdot \vec{x})$ where $\bar{S}$ is a $N \times \delta$ matrix whose elements are all either -1, +1, or 0, and where $\bar{C} \cdot \bar{T} \cdot \vec{x}$ is a column vector of δ complex numbers.

Instead of solving these two specific design problems, consider the more general problem of designing a summation array which can compute $\vec{z} = \bar{A} \cdot \vec{x}$ where $\bar{A}$ is a $M \times N$ matrix whose elements are all either -1, +1, or 0, and where $\vec{x}$ is a column vector of N complex numbers.

$\vec{z}$ will be a column vector of M complex numbers where

$$z_i = \sum_{j=0}^{N-1} A_{i,j} \cdot x_j, \text{ for } i = 0, 1, \ldots, M-1 \qquad \text{(Eq 14)}$$

but, since each $A_{i,j} \in \{-1, 1, 0\}$, it is clear that $\vec{z}$ can be computed from $\vec{x}$ using only addition and subtraction operations (no multiplications are required).

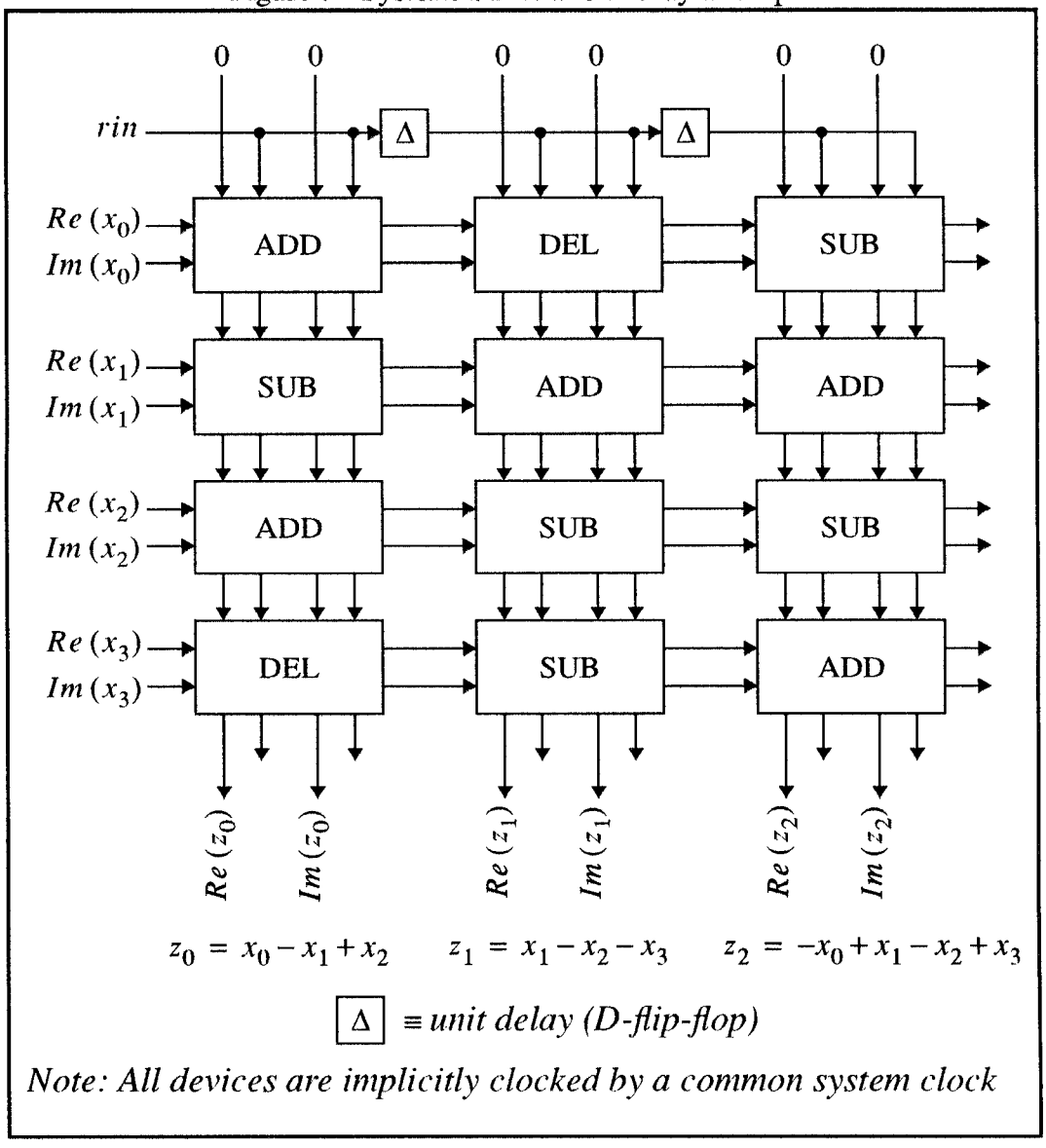
Design Solution Using Systolic Arrays

The stated design problem can be solved by constructing a systolic array[3] of fully pipelined bit-serial adders, subtracters, and delay cells, which operate on complex numbers.

An example summation array design is shown in Figure 7 for the case M=3, N=4, and

$$\bar{A} = \begin{bmatrix} 1 & 0 & -1 \\ -1 & 1 & 1 \\ 1 & -1 & -1 \\ 0 & -1 & 1 \end{bmatrix}^T \qquad \text{(Eq 15)}$$

Figure 7 - Systolic Summation Array Example



The standard cells named ADD, SUB, and DEL have been designed as VLSI circuits which all have the same footprints and all have I/O ports in the same locations so that they are drop-replaceable. These cells have been designed to connect by abutment.

Standard Cell ADD

The ADD cell has been designed to add two complex numbers, *xin* and *zin* which arrive bit-serially with the lsb's coming first. The result is $zout = xin + zin$. Since each complex addition requires two real additions, the ADD cell contains two independent bit-serial adders. The symbol and schematic of the ADD cell is shown in Figure 8 and Figure 9 respectively.

Figure 8 - ADD Symbol

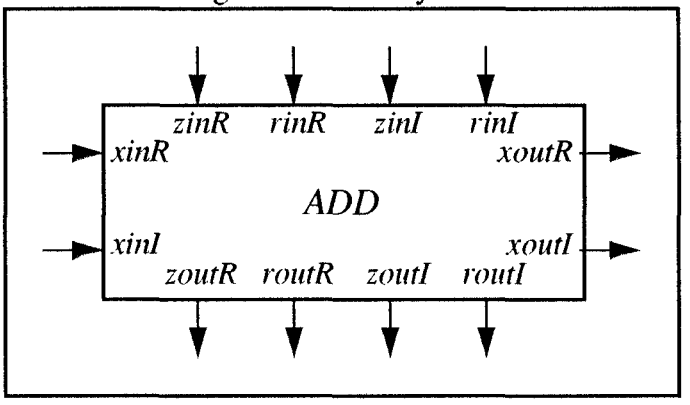
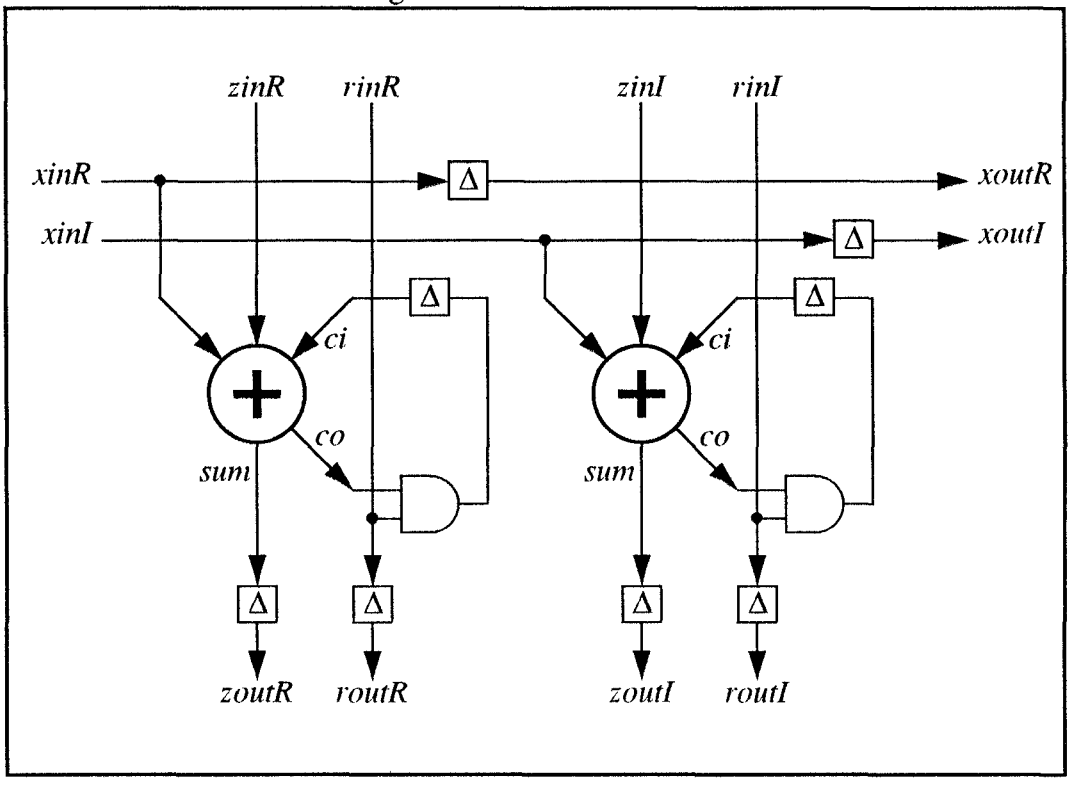


Figure 9 - ADD Schematic



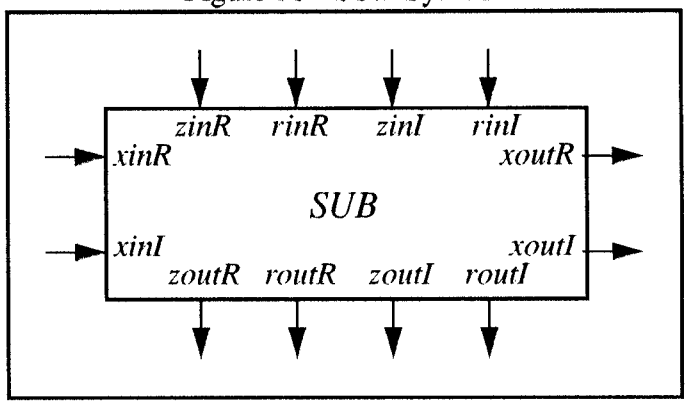
Pin names which end in a *'R'*, such as *xinR*, correspond to real components and pin names which end in an *'I'*, such as *xinI*, correspond to imaginary components.

The ADD cell requires two input control signals, *rinR* and *rinI*, which must arrive with a leading zero exactly one cycle before the lsb's of *xin* and *zin* arrive. These *rin* signals are used to reset the carry flip-flops inside the bit-serial adders.

Standard Cell SUB

The SUB cell has been designed to subtract two complex numbers, *xin* and *zin* which arrive bit-serially with the lsb's coming first. The result is $zout = zin - xin$. The symbol of the SUB cell is shown in Figure 10.

Figure 10 - SUB Symbol

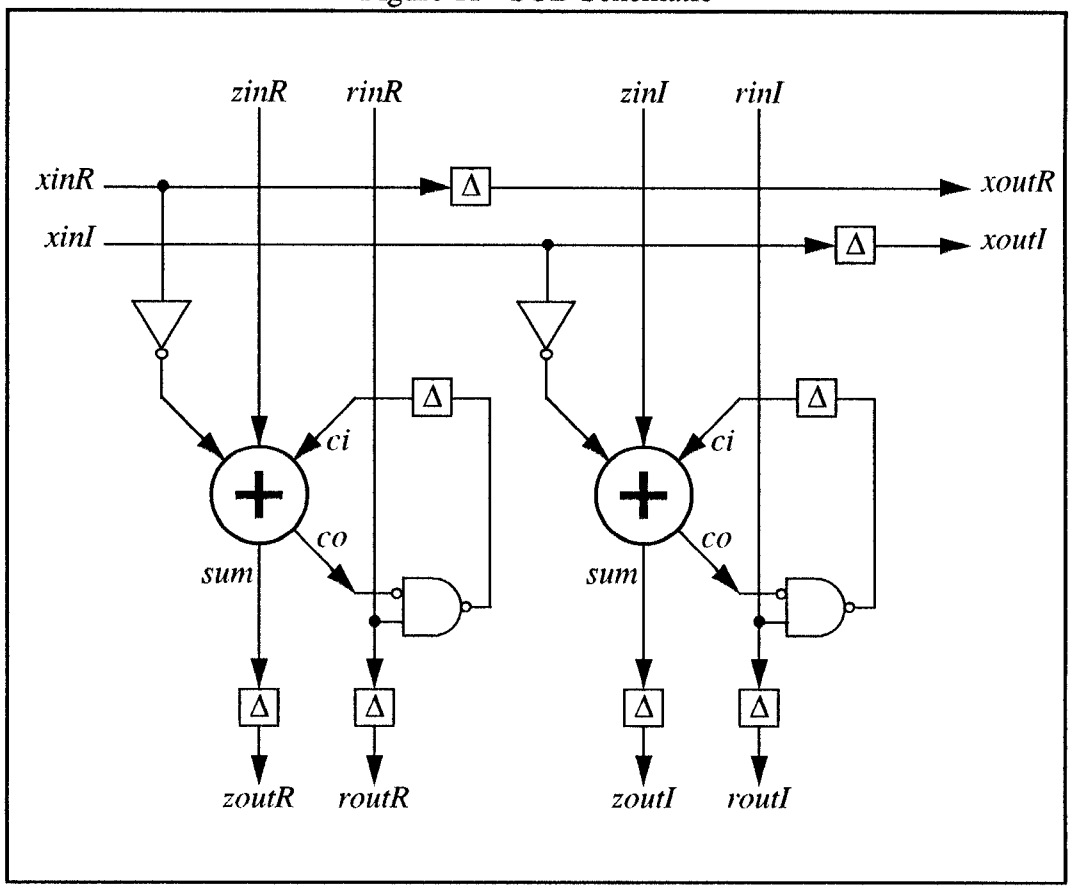


The difference, *zin* - *xin*, can be computed by taking the two's complement of *xin* and adding it to *zin*. Taking the two's complement of *xin* can be accomplished by inverting *xin* and adding one.

Therefore, the SUB schematic, shown in Figure 11, differs from the ADD schematic in the following ways:

- The *xin* input to each full-adder has been inverted
- To accomplish the addition of a one to *xin*, the function driving the carry flip-flop has been changed so that the carry is SET instead of RESET when the *rin* is zero (this adds one to the lsb's)

Figure 11 - SUB Schematic



Standard Cell DEL

The DEL cell (or DELAY cell) is designed to simply pass *zin* to *zout* without adding anything to it or subtracting anything from it. The symbol and schematic of the DEL cell is shown in Figure 12 and Figure 13 respectively.

Figure 12 - DEL Symbol

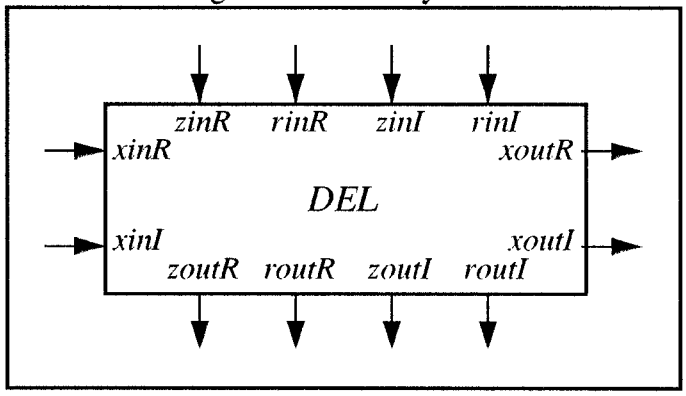
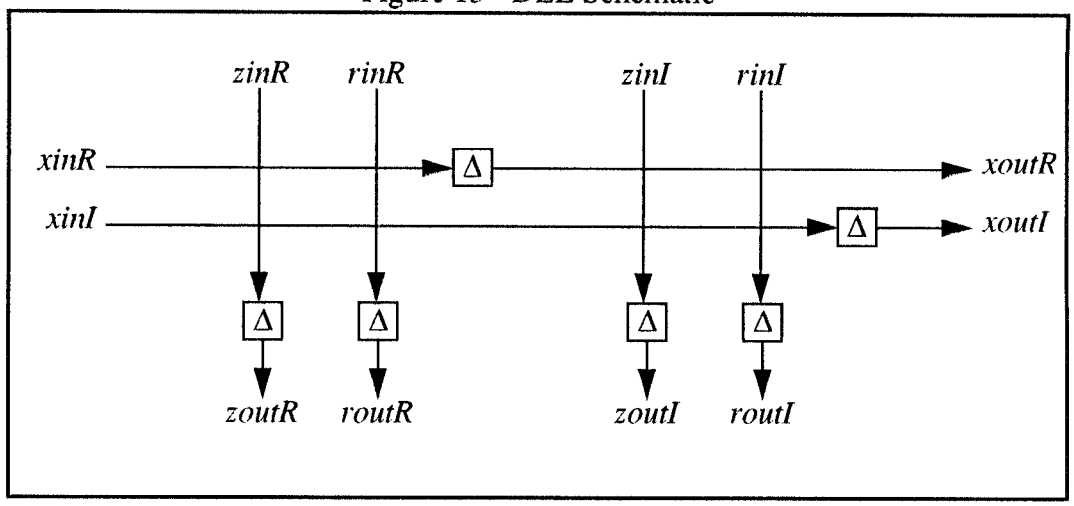


Figure 13 - DEL Schematic



Automated Summation Array Generation

Summation arrays can be easily constructed using the ADD, SUB, and DEL standard cells. The arrangement of these cells in the VLSI layout of a summation array directly corresponds to the arrangement of +1's, -1's, and 0's respectively in the transpose of the $\bar{A}$ matrix.

A program named *sumgen*[6] is now available which accepts as input the transpose of any $\bar{A}$ matrix and automatically outputs a VLSI layout of the corresponding summation array in *magic*[7] format. In addition, the generated layout will include the required *rin* delay registers shown across the top of Figure 7, will properly ground-out all the *zin*'s along the top of the summation array, and will surround all of this with single-phase to two-phase clock generation and distribution circuitry.

The source code for *sumgen* is written in the ANSI C programming language and is available upon request.

Chapter 4 - Multiplier Array Design

Problem Definition

The DFT architecture shown in Figure 1 requires the design of a multiplier array which must compute $\overline{C} \cdot (\overline{T} \cdot \vec{x})$ where $\overline{C}$ is a diagonal matrix with elements which are either real or imaginary, and where $\overline{T} \cdot \vec{x}$ is a column vector of δ complex numbers.

To simplify notation, consider the problem of designing a multiplier array which must compute $\vec{z} = \overline{C} \cdot \vec{x}$ where $\overline{C}$ is a diagonal matrix with elements which are either real or imaginary, and where both $\vec{z}$ and $\vec{x}$ are column vectors of N complex numbers.

Design Solution

According to the problem definition, the elements of $\vec{z}$ can be computed from the elements of $\vec{x}$ using the relationship

$$z_i = C_{i,i} \cdot x_i, \text{ for } i = 0, 1, 2, \ldots, N-1 \qquad \text{(Eq 16)}$$

Decomposing (Eq 16) into real and imaginary components we get

$$Re(z_i) = Re(C_{i,i})\,Re(x_i) - Im(C_{i,i})\,Im(x_i) \qquad \text{(Eq 17)}$$

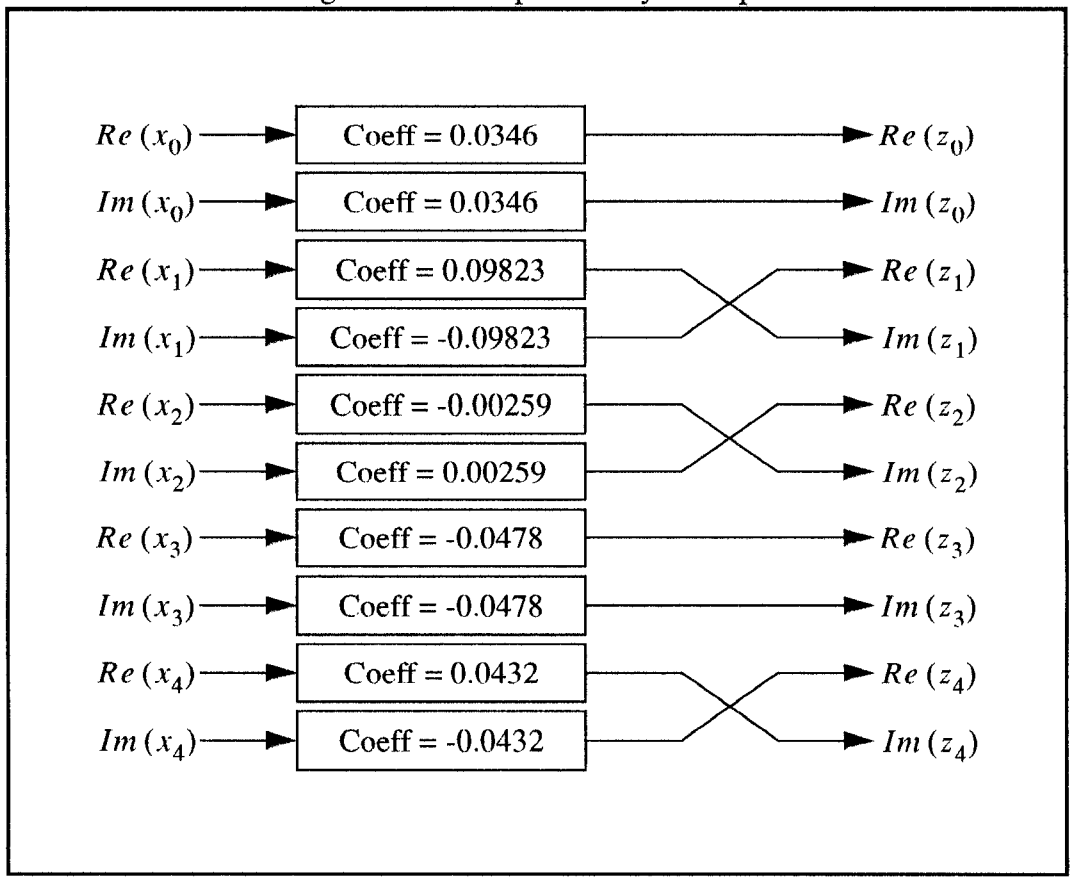
$$Im(z_i) = Re(C_{i,i})\,Im(x_i) + Im(C_{i,i})\,Re(x_i) \qquad \text{(Eq 18)}$$

but, since either $Re(C_{i,i})$ or $Im(C_{i,i})$ is zero for each i, it is clear that each z_i can be computed using just two multiplications instead of four. Therefore, $\vec{z}$ can be computed using just $2N$ multiplications.

An example multiplier array is shown in Figure 14 for the case N=5 and

$$\overline{C} = \begin{bmatrix} 0.0346 & 0 & 0 & 0 & 0 \\ 0 & 0.09823j & 0 & 0 & 0 \\ 0 & 0 & -0.00259j & 0 & 0 \\ 0 & 0 & 0 & -0.0478 & 0 \\ 0 & 0 & 0 & 0 & 0.0432j \end{bmatrix} \quad \text{(Eq 19)}$$

Figure 14 - Multiplier Array Example



Notice that for each $C_{i,i}$ which is imaginary, the outputs of the multipliers are switched and the sign of the coefficient of the lower multiplier is switched.

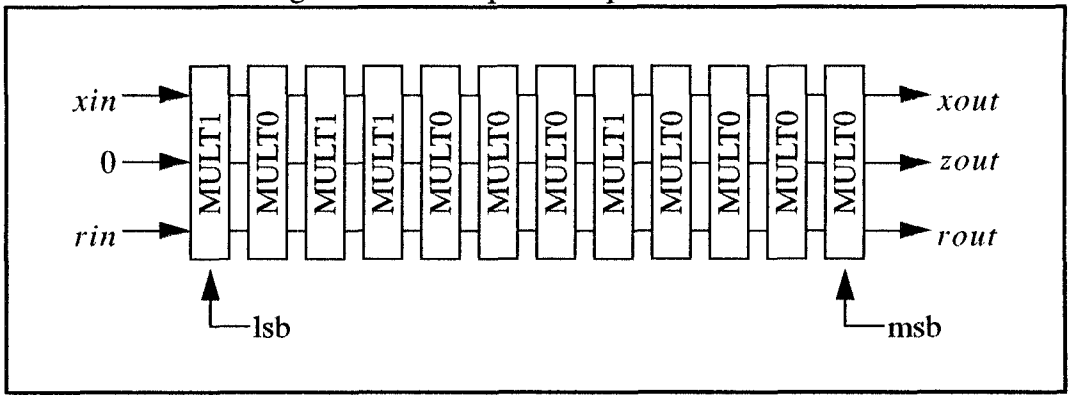
Multiplier Design

Each multiplier will be implemented as a fixed-coefficient, bit-serial multiplier which is fully pipelined[8,9]. The input data will be a p-bit number and the coefficient will be a p-bit number.

Normally the product would be a $2p$-bit number, but this multiplier design will assume that the coefficient is a fraction (with magnitude ≤ 0.5), so that the product will also be a p-bit number. Accordingly, the multiplier design will automatically truncate its result internally to a p-bit number.

Each multiplier will be constructed from standard cells named MULT1 and MULT0 which are placed side-by-side (connecting by abutment) in a pattern which corresponds directly to the pattern of 1's and 0's in the coefficient. For example, consider the coefficient 0.0346 (decimal) which is 0.000010001101 (binary) using 12-bit precision. The corresponding multiplier schematic is shown in Figure 15.

Figure 15 - Example Multiplier Schematic



Notice that the pattern of MULT1's and MULT0's corresponds directly to the mirror image of the pattern of 1's and 0's in the binary coefficient.

One cycle before the lsb of *xin* arrives, the *rin* signal must arrive with a leading zero, which alerts the multiplier that a new multiplication is beginning.

The product, *zout*, will emerge from the *ppout* output of the final stage of the multi-

plier with the lsb emerging first and the msb emerging last, followed immediately by a sign-extension bit.

The *rout* output control signal will emerge from the final stage with a leading zero, one cycle before the lsb's of *zout* and *xout* emerge.

The *xout* and *rout* signals are simply delayed versions of the *xin* and *rin* signals which both emerge time-aligned with the multiplication result, *zout*.

The *ppin* input of the first multiplier stage must be tied to a logic-zero as shown in Figure 15. The names *ppin* and *ppout* stand for 'partial product in' and 'partial product out' respectively.

There is a special rule concerning the final stage of the multiplier which was not previously mentioned. If the multiplier coefficient is a negative number then the final stage of the multiplier must be a MULT-1 instead of a MULT1 in order to account for the negative sign of the coefficient.

The MULT1, MULT-1, and MULT0 standard cells are described in the following sections.

Standard Cell MULT1

The MULT1 cell has been designed to add *xin* to the partial product *ppin*. The symbol and schematic for MULT1 are shown in Figure 16 and Figure 17 respectively.

Figure 16 - MULT1 Symbol

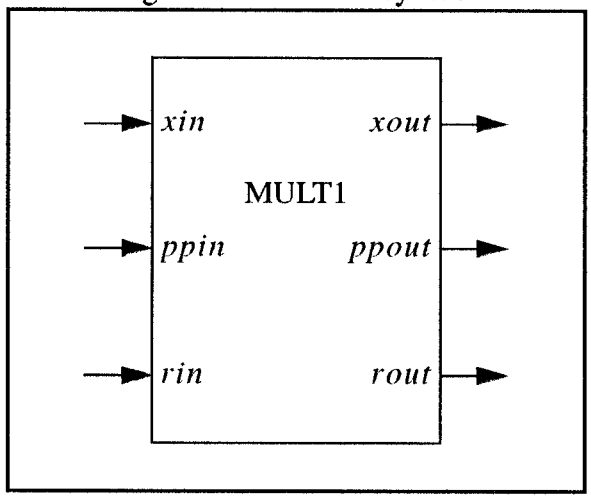
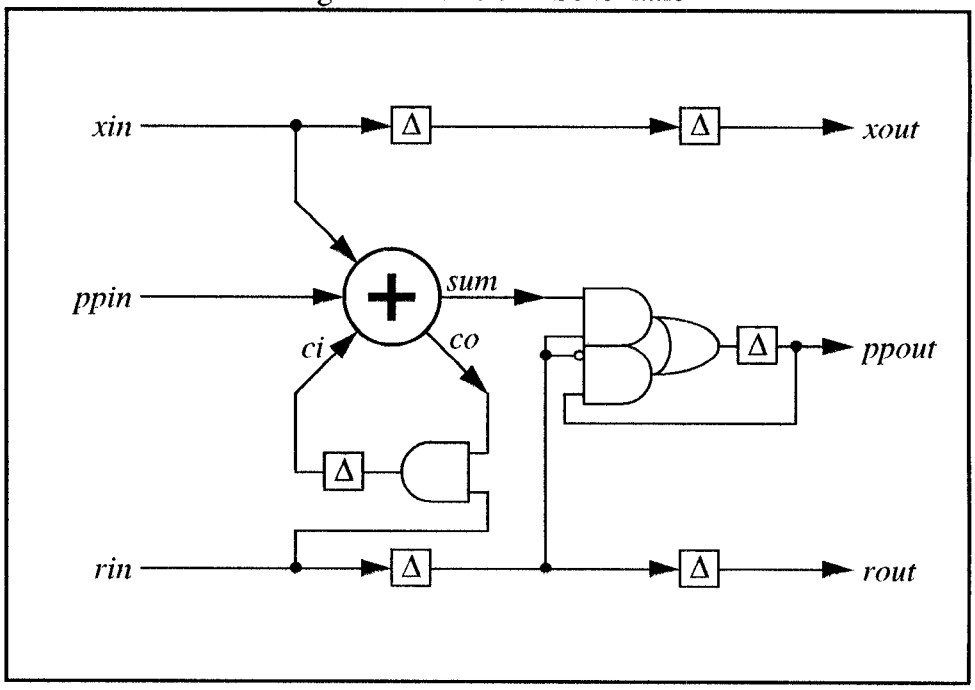


Figure 17 - MULT1 Schematic



The data selector driving the *ppout* flip-flop is used to sign-extend the previous partial product, instead of storing the *sum* corresponding to the lsb of the next partial product, thereby simultaneously truncating the next product to *p*-bits.

Standard Cell MULT-1

The MULT-1 cell has been designed to be placed only as the final stage of a multiplier which has a negative coefficient. Accordingly, MULT-1 will subtract *xin* from the partial product. This is accomplished by adding the two's-complement of *xin* to *ppin*. The symbol and schematic for MULT-1 are shown in Figure 18 and Figure 19 respectively.

Figure 18 - MULT-1 Symbol

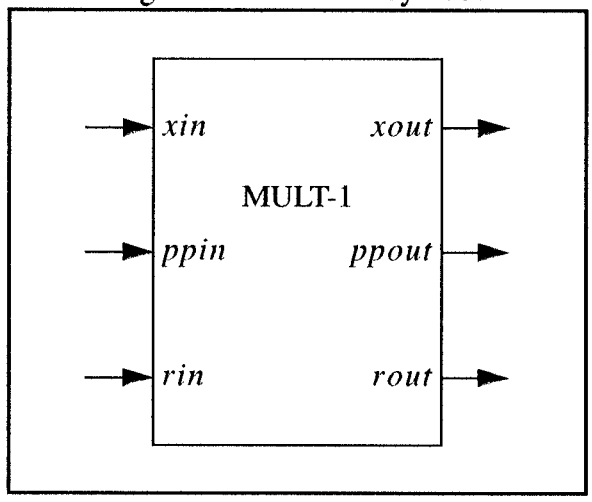
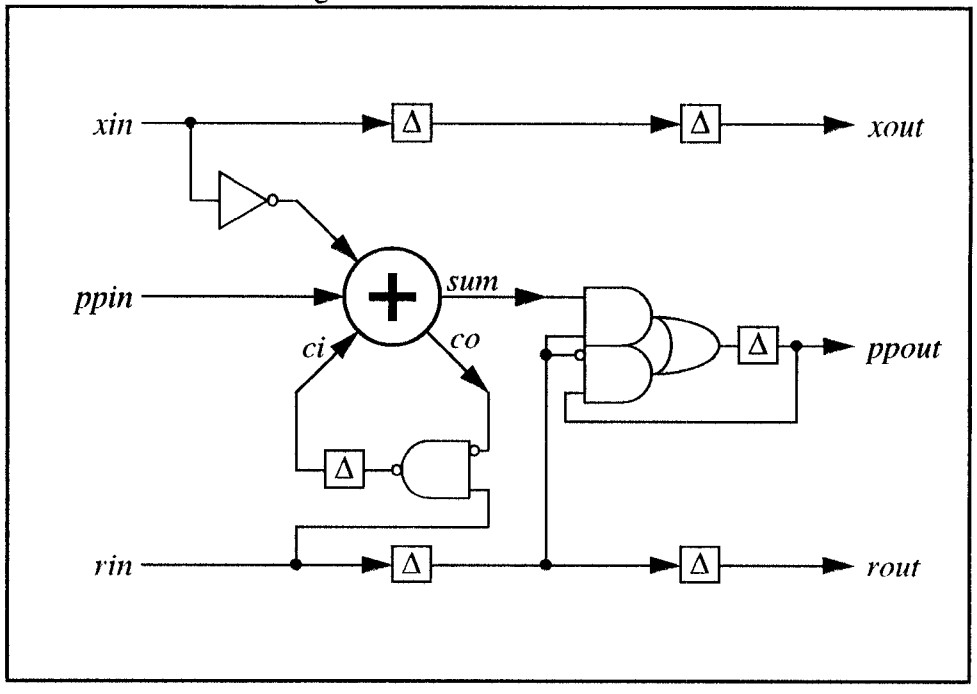


Figure 19 - MULT-1 Schematic



Standard Cell MULT0

The MULT0 cell has been designed to delay the *xin*, *ppin*, and *rin* signals. The symbol and schematic for MULT0 are shown in Figure 20 and Figure 21 respectively.

Figure 20 - MULT0 Symbol

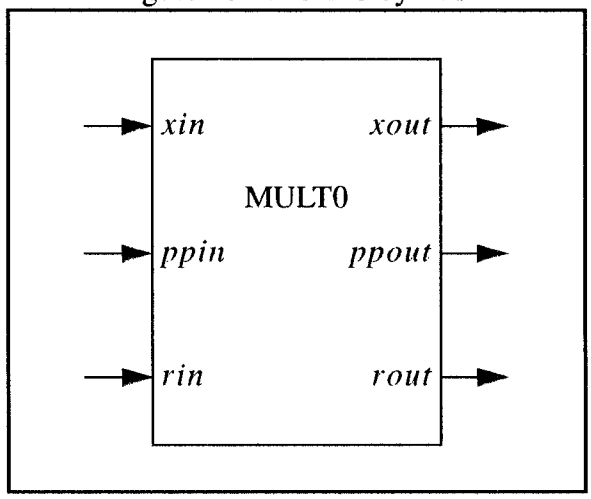
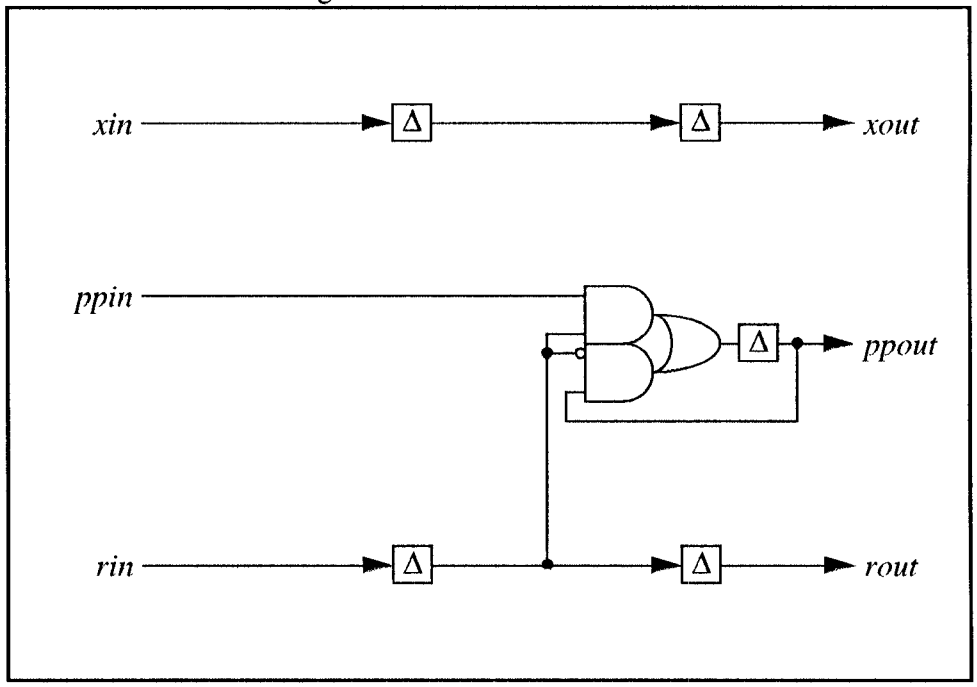


Figure 21 - MULT0 Schematic



Automated Multiplier Array Generation

A program named *multgen*[10] is now available which accepts as input a list of multiplier coefficients (in binary form) and automatically outputs a VLSI layout of the corresponding array of multipliers in *magic* format. In addition, the generated layout will properly ground-out the *ppin*'s at the first stage of each multiplier, and will surround the entire multiplier array with single-phase to two-phase clock generation and distribution circuitry.

The source code for *multgen* is written in the ANSI C programming language and is available upon request.

Chapter 5 - Analysis

Introduction

In this chapter several numerical issues will be considered. First, internal numerical overflow will be considered at each stage of the DFT pipeline. This will lead to a specification which requires the input dynamic range of the DFT processor to be limited. Second, the dynamic range of the DFT outputs will be computed. Third, the effects of truncation error will be considered. Finally, expressions will be derived for the pipeline depth, latency, and performance of the DFT processor design.

Avoiding Internal Overflow

The summation array at the top of Figure 1 will be referred to as T since it computes the summations corresponding to the $\overline{T}$ matrix. Similarly, the multiplier array will be referred to as C, and the summation array at the bottom of the figure will be referred to as S.

The T array computes summations over the input data $\hat{z} = \overline{T} \cdot \hat{x}$. There are N points in $\hat{x}$, and each point is represented by a real and imaginary component, where each component is a p-bit signed integer.

If $\overline{T}$ is a $\delta \times N$ matrix, then T will compute δ complex summations over N numbers.

$$z_i = \sum_{j=0}^{N-1} T_{i,j} \cdot x_j, \text{ for } i = 0, 1, \ldots, \delta - 1 \text{ where } T_{i,j} \in \{-1, 1, 0\} \qquad \text{(Eq 20)}$$

The real and imaginary components of z_i are computed independently as

$$Re(z_i) = \sum_{j=0}^{N-1} T_{i,j} \cdot Re(x_j), \text{ for } i = 0, 1, \ldots, \delta - 1 \qquad \text{(Eq 21)}$$

$$Im(z_i) = \sum_{j=0}^{N-1} T_{i,j} \cdot Im(x_j) \text{ , for } i = 0, 1, \ldots, \delta - 1 \qquad \text{(Eq 22)}$$

The real and imaginary components of z_i will also be p-bit signed integers.

When considering overflow in the T array, the worst case situation can be attained by assuming that $T_{i,j} = 1$ for every (i, j) and that $Re(x_j) = Im(x_j) = 2^{p-1}$ for all j. Note that 2^{p-1} is the largest possible p-bit signed integer. In this situation, (Eq 21) and (Eq 22) can be used to show that $Re(z_i) = Im(z_i) = N \cdot 2^{p-1}$ for all i. But, since N is usually much greater that one, this situation would result in internal overflow.

To guarantee that internal overflow will be avoided in the T array, the input data, $\hat{x}$, must have a limited dynamic range according to the following specification:

$$|x_j| \leq \frac{2^{p-1}}{N}, \text{ for } j = 0, 1, \ldots, N - 1 \qquad \text{(Eq 23)}$$

If (Eq 23) is satisfied, then it follows that

$$|Re(x_j)| \leq \frac{2^{p-1}}{N} \text{ and } |Im(x_j)| \leq \frac{2^{p-1}}{N} \text{ for } j = 0, 1, \ldots, N - 1 \qquad \text{(Eq 24)}$$

and by plugging (Eq 24) into (Eq 21) and (Eq 22) we find that

$$|Re(z_i)| \leq 2^{p-1} \text{ and } |Im(z_i)| \leq 2^{p-1} \text{ for } i = 0, 1, \ldots, \delta - 1 \qquad \text{(Eq 25)}$$

which proves that internal overflow will be avoided in the T array.

Next, consider the C array which simply scales the results produced by the T array. If (Eq 23) is satisfied, then the outputs of the T array will have magnitudes less than or equal to 2^{p-1} (i.e. they will be representable as p-bit signed integers).

Since every bit-serial multiplier in C is assumed to have a coefficient with magnitude less than or equal to 0.5, it is clear that the outputs of the C array will have magnitudes less than or equal to 2^{p-2}. Therefore internal overflow will not occur in the C array.

Unity Gain Property

If one takes the next logical step and simply considers the possibility of overflow in the S array, which computes N summations over the δ complex numbers produced by the C array, one would find that the outputs of the S array would have magnitudes bounded by $2^{p-2} \cdot \delta$. But since δ is typically comparable to N in most DFT algorithms, this result indicates that internal overflow is possible in the S array, though the following theorem proves the contrary.

Theorem: The Unity Gain Property of the DFT

If $|x_n| \leq M$ for $n = 0, 1, 2, \ldots, N-1$, then $|X_k| \leq M$ for $k = 0, 1, 2, \ldots, N-1$.

Proof:

$$|X_k| = \left| \frac{1}{N} \sum_{n=0}^{N-1} x_n \cdot e^{-j\frac{2\pi}{N}nk} \right|, \text{ for } k = 0, 1, 2, \ldots, N-1 \qquad \text{(Eq 26)}$$

The magnitude of a sum is always less than or equal to the sum of the magnitudes

$$|X_k| \leq \frac{1}{N} \sum_{n=0}^{N-1} \left| x_n \cdot e^{-j\frac{2\pi}{N}nk} \right|, \text{ for } k = 0, 1, 2, \ldots, N-1 \qquad \text{(Eq 27)}$$

The magnitude of the complex exponential is always equal to one

$$|X_k| \leq \frac{1}{N} \sum_{n=0}^{N-1} |x_n| \cdot 1, \text{ for } k = 0, 1, 2, \ldots, N-1 \qquad \text{(Eq 28)}$$

$$|X_k| \leq \frac{1}{N} \sum_{n=0}^{N-1} M = \frac{NM}{N} = M, \text{ for } k = 0, 1, 2, \ldots, N-1, \text{ Q.E.D.} \qquad \text{(Eq 29)}$$

The unity gain property of the DFT can now be applied to the DFT processor design at hand to show that the S array will never overflow. If (Eq 23) is satisfied, then we have that $|x_n| \leq \frac{2^{p-1}}{N} \equiv M$ and therefore, $|X_k| \leq \frac{2^{p-1}}{N}$.

Truncation Error

The bit-serial multipliers in the C array are self-truncating. Each multiplier computes the product of two p-bit numbers and the result is internally truncated to a p-bit number. Therefore, each product is within one lsb of the correct answer.

The truncation errors induced by the multipliers can be amplified by the S array which computes summations over at most δ multiplier outputs. In the worst case, assuming all multiplier truncation errors add in phase, any DFT output is guaranteed to be correct within δ lsb's.

Pipeline Depth

The number of clock cycles required to fill a pipeline will be referred to as the pipeline depth. The depth of the DFT processor shown in Figure 1 can be computed by adding the depths of the T, C, and S arrays.

The pipeline depth of a summation array is equal to the number of rows in the summation array. The pipeline depth of a multiplier array is equal to $2p$ where p is the number of bits in the input data (i.e. the precision).

Therefore, the depth of the T array is N cycles, the depth of the C array is $2p$ cycles, and the depth of the S array is δ cycles.

Therefore, the overall depth of the DFT processor design is $N + 2p + \delta$ cycles.

Latency

Latency is defined as the number of cycles occurring between the start of two successive pipeline operations[11]. In the DFT processor, the start of two successive DFT operations must be separated by p+1 clock cycles. This is the number of cycles required to shift-in the p-bit data word x_0 plus one cycle for the sign-extension of x_0. Thus, the latency of the DFT processor design is p+1 cycles/operation.

Performance

The performance (or throughput) of a DFT processor can be measured by the number of p-bit N-point DFT operations which can be computed per second.

If the system clock rate of a DFT processor is f, then the performance is given by f (cycles/s) divided by the latency (cycles/operation), resulting in a performance of $\frac{f}{p+1}$ DFT operations per second.

The DFT architecture presented has been designed to provide a performance which is not a function of N. That is, every p-bit N-point DFT processor will compute about the same number of DFT's per second, independent of how large N becomes.

This result is quite different from most other DFT processor architectures which have performances which degrade rapidly as a function of N. Of course, the trade-off is that the hardware requirements of the presented DFT architecture are relatively strong functions of N. Therefore, this DFT architecture is best suited for real-time digital signal processing systems, where hardware is cheap but time is very expensive.

Chapter 6 - VLSI Implementation of an 8-Point DFT Processor

Introduction

To demonstrate the high performance of this DFT architecture, an 8-point DFT processor has been designed and implemented as a VLSI chip which operates at 50 MHz and has a throughput of 2.9 million complex 8-point 16-bit DFT's per second.

Design Data

For this design, $N = 8$ and $p = 16$ and the three matrices $\bar{S}$, $\bar{C}$, and $\bar{T}$ are given below[4]

$$\bar{S} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & -1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & -1 & -1 \end{bmatrix} \qquad \text{(Eq 30)}$$

$$\bar{C} = \frac{1}{8} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cos\left(\frac{2\pi}{8}\right) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -j\sin\left(\frac{2\pi}{8}\right) \end{bmatrix} \qquad \text{(Eq 31)}$$

$$\bar{T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 \end{bmatrix} \quad \text{(Eq 32)}$$

Since $\bar{C}$ is an 8×8 matrix, it is clear that $\delta = 8$.

VLSI Implementation

The transpose of the $\bar{S}$ matrix was entered into a text file named 'S' which is shown in Figure 22.

Figure 22 - Text File Input for *sumgen* for the *S* Array

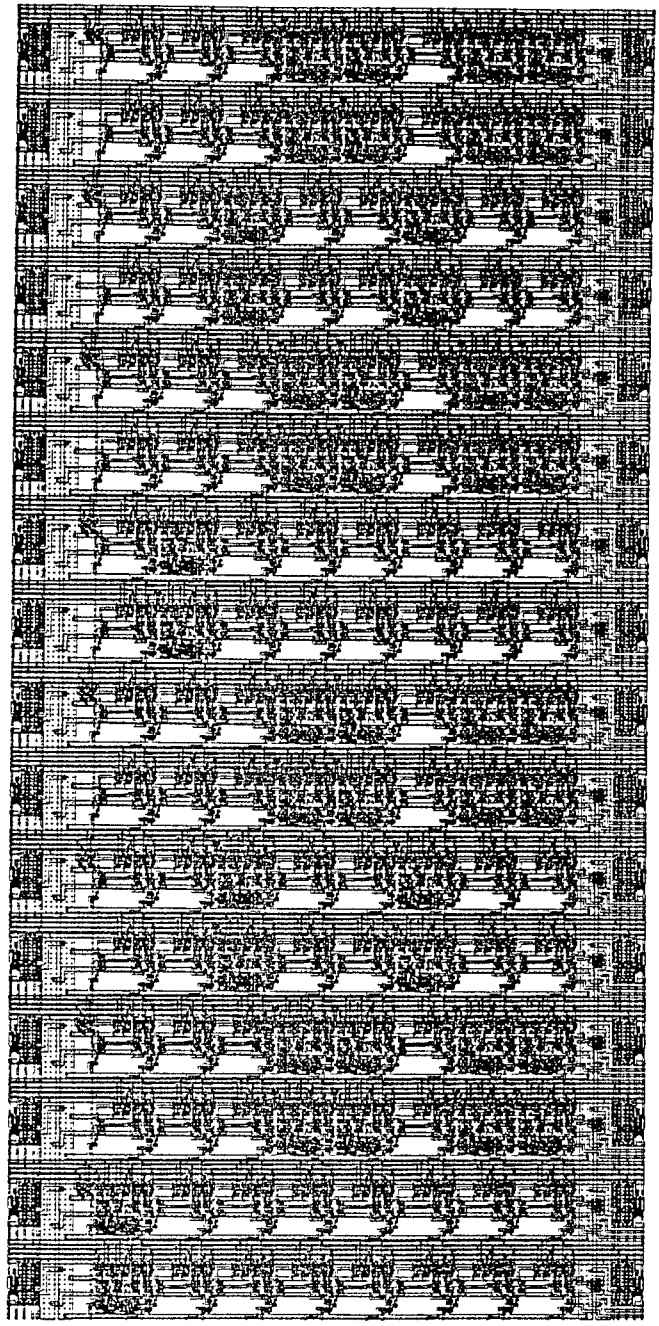
1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	1	0	0	0	1	0
0	1	0	1	0	1	0	1
0	1	0	-1	0	-1	0	1
0	0	1	0	0	0	-1	0
0	1	0	-1	0	1	0	-1
0	1	0	1	0	-1	0	-1

Similarly, the transpose of the $\bar{T}$ matrix was entered into a text file named 'T'.

The layouts for the *S* and *T* arrays were then generated automatically by entering the following commands at the UNIX prompt: 'sumgen -c < S > S.mag' and 'sumgen -c < T > T.mag'. The '-c' option indicates to *sumgen* that a complex summation array is to be generated. By default, *sumgen* creates summation arrays which sum real numbers only.

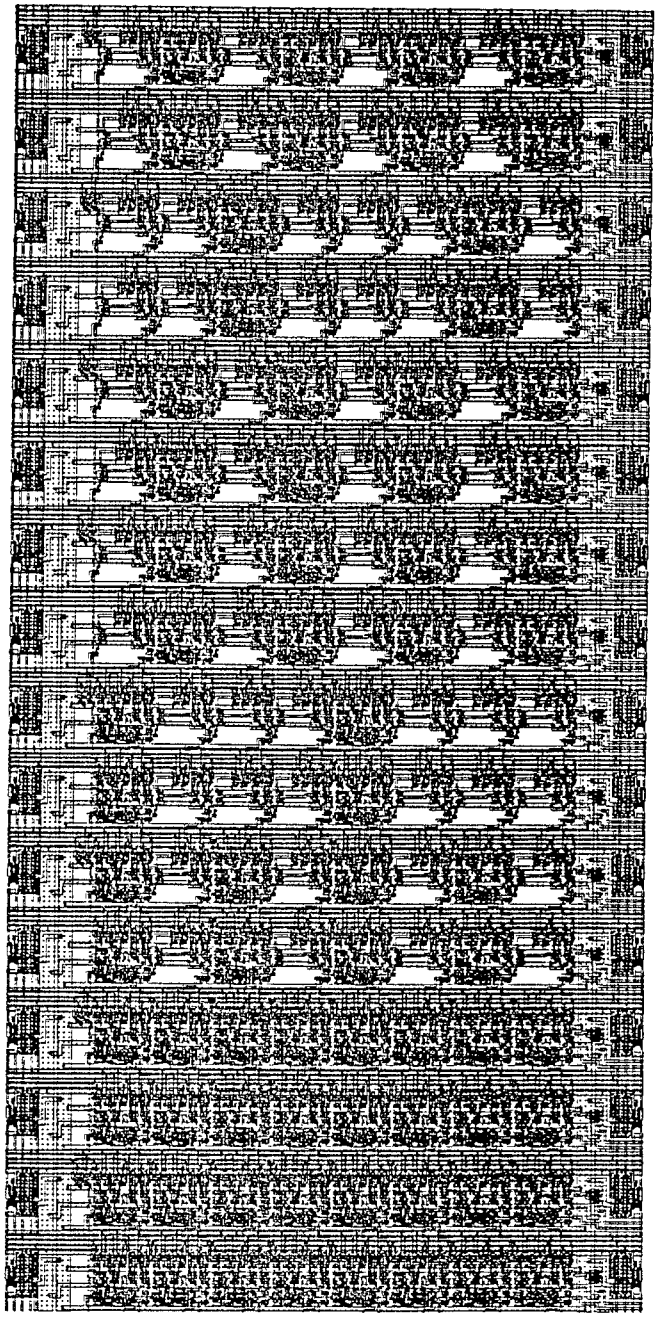
The layouts of the *S* and *T* arrays are shown in Figure 23 and Figure 24 respectively.

Figure 23 - Layout of the *S* Array



Height = 2720λ, Width = 1320λ

Figure 24 - Layout of the *T* Array



Height = 2720λ, Width = 1320λ

The diagonal elements of the $\overline{C}$ matrix were entered into a text file named 'C' which is shown in Figure 25.

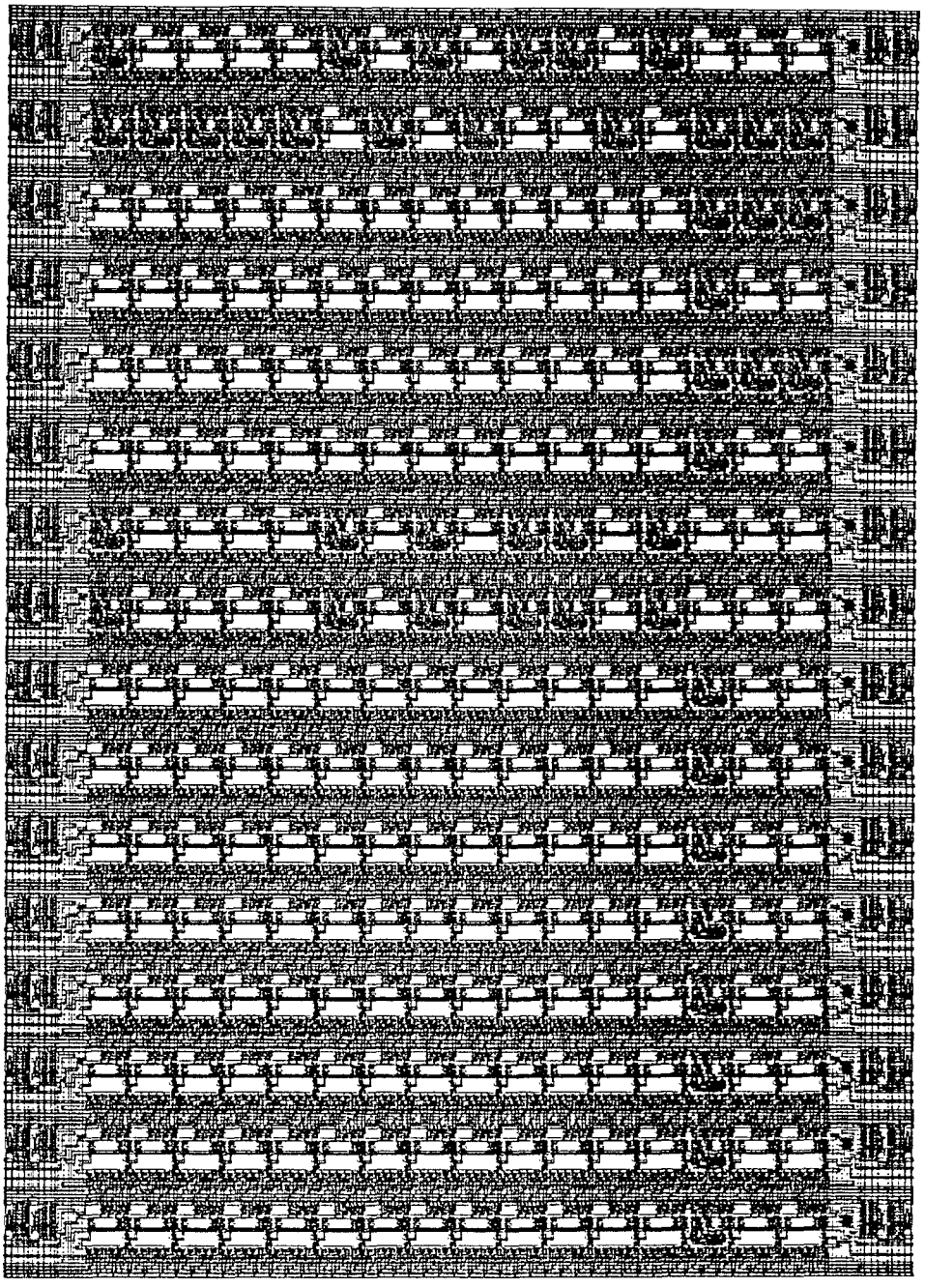
Figure 25 - Input for *multgen* for the *C* Array

```
.0010000000000000
.0010000000000000

.0010000000000000
.0010000000000000

.0010000000000000
.0010000000000000

.0010000000000000
.0010000000000000

.0001011010100001
.0001011010100001

.0010000000000000
.1110000000000000

.0010000000000000
.1110000000000000

.1110100101011111
.0001011010100001
```

Notice that the multiplier coefficients are entered as p-bit binary fractions and that the error in discretizing the coefficients will be bounded by 2^{-p}.

The layout for the *C* array was then generated by entering the following command at the UNIX prompt: 'multgen < C > C.mag'. The layout of the *C* array is shown in Figure 26.

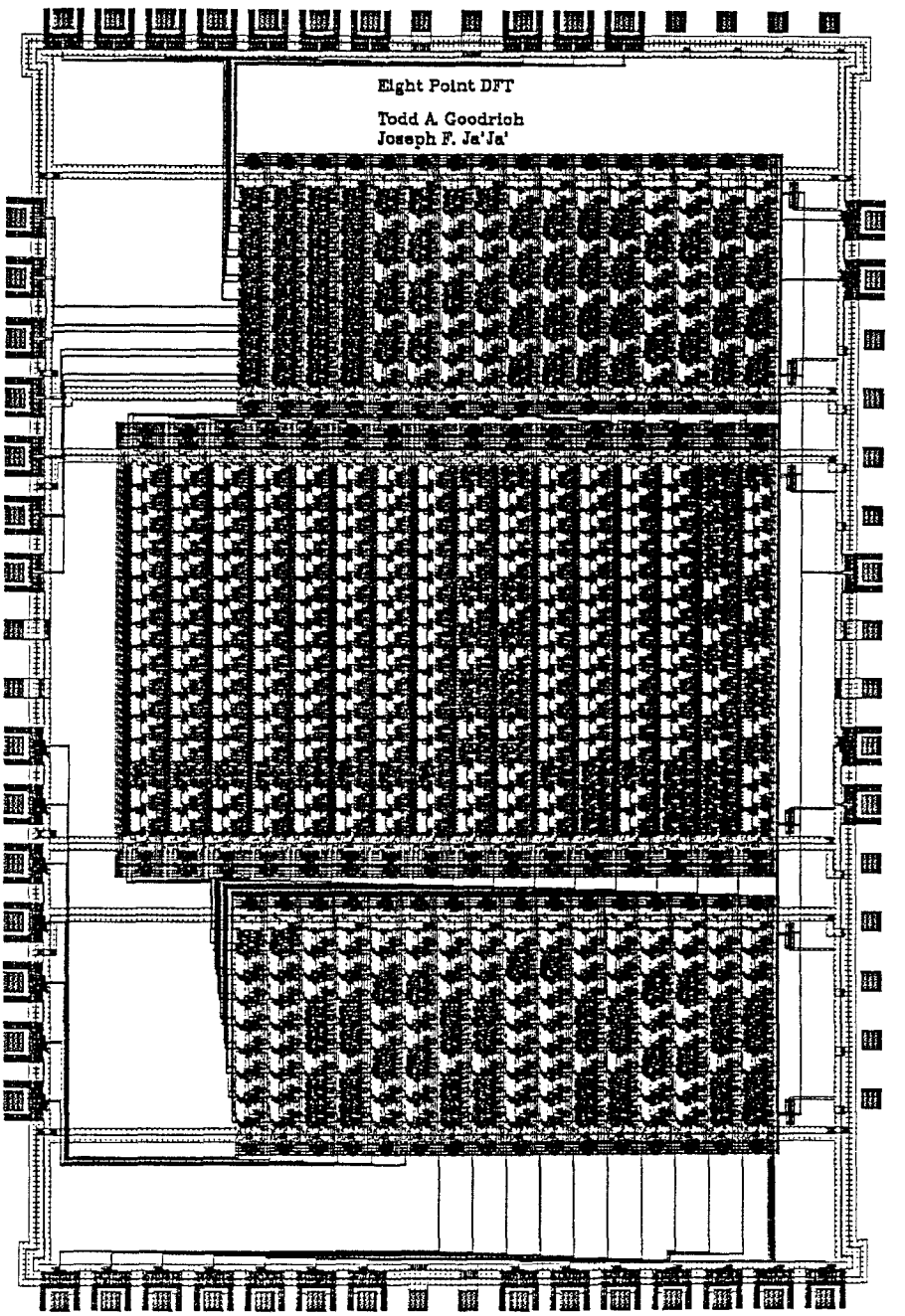
Figure 26 - Layout of the *C* Array



Height = 3316λ, Width = 2344λ

The layouts for the S, C, and T arrays were then placed as macro-cells using the *magic* layout editor and routing was added manually to connect the outputs of the T array to the inputs of the C array and to connect the outputs of the C array to the inputs of the S array. The entire design was then placed into a standard frame of 2μm NWELL pad cells (which are available from MOSIS) and the DFT inputs and outputs were routed to the pads. Also, the power, ground, and the input clock signals were routed to each macro-cell. The final layout of the 8-point DFT chip processor is shown in Figure 27.

Figure 27 - Layout of the 8-Point DFT Processor Chip



Height = 6600λ, Width = 4400λ

Simulation

First, the individual standard cells were fully simulated using SPICE2G6. Next, switch-level simulations were performed at various levels of the processor design using *irsim*[7]. Finally, the DFT processor was simulated at the top level using *irsim.*

An example of the input and simulated output of the 8-point DFT processor chip are shown in Figure 28 along with the theoretical correct output (computed using (Eq 1) and rounded to the nearest integer) and the corresponding percent error.

Figure 28 - Simulation Example

Inputs	Simulated Output	Theoretical Output	% Error
$Re(x_0) = 1543$	$Re(X_0) = 73$	$Re(X_0) = 73$	0.0
$Im(x_0) = -3964$	$Im(X_0) = 464$	$Im(X_0) = 464$	0.0
$Re(x_1) = -974$	$Re(X_1) = -439$	$Re(X_1) = -437$	0.46
$Im(x_1) = 968$	$Im(X_1) = -2745$	$Im(X_1) = -2743$	0.07
$Re(x_2) = 3620$	$Re(X_2) = 1393$	$Re(X_2) = 1393$	0.0
$Im(x_2) = 2710$	$Im(X_2) = -225$	$Im(X_2) = -223$	0.90
$Re(x_3) = 3703$	$Re(X_3) = 389$	$Re(X_3) = 389$	0.0
$Im(x_3) = -1600$	$Im(X_3) = -313$	$Im(X_3) = -314$	0.32
$Re(x_4) = 1073$	$Re(X_4) = 515$	$Re(X_4) = 515$	0.0
$Im(x_4) = 3154$	$Im(X_4) = 546$	$Im(X_4) = 546$	0.0
$Re(x_5) = -1850$	$Re(X_5) = 695$	$Re(X_5) = 696$	0.14
$Im(x_5) = 3005$	$Im(X_5) = -911$	$Im(X_5) = -910$	0.11
$Re(x_6) = -3878$	$Re(X_6) = -675$	$Re(X_6) = -674$	0.15
$Im(x_6) = 2145$	$Im(X_6) = -1193$	$Im(X_6) = -1193$	0.0
$Re(x_7) = -2648$	$Re(X_7) = -413$	$Re(X_7) = -412$	0.24
$Im(x_7) = -2700$	$Im(X_7) = 409$	$Im(X_7) = 409$	0.0

Testing

The 8-point DFT processor was fabricated through MOSIS in a 2μm NWELL CMOS process and packaged as 64-pin DIP's.

The packaged parts were functionally tested and were found to operate correctly and consistently at clock frequencies up to 50 MHz. The actual parts consistently produced the same results as the *irsim* simulations predicted.

Performance

The 16-bit 8-point DFT processor has a pipeline depth of $8 + 2 \times 16 + 8 = 48$ cycles and a latency of $16 + 1 = 17$ cycles/operation.

At $f = 50$ MHz, the performance is $\frac{50 \times 10^6}{17} = 2.9 \times 10^6$ DFT operations per second.

Compared to commercially available DFT processors, which have performance measured in only thousands of DFT's per second, the presented DFT architecture provides a performance improvement of several orders of magnitude. In addition, as the number of points, N, is increased, the performance of commercially available DFT processors degrades rapidly, while the performance of the presented DFT architecture remains nearly constant.

Compared to the previous generation of DFT processors designed by Kapoor[2], which were designed with the same basic VLSI architecture as shown in Figure 1, but which were constructed using older technology and more complicated arithmetic schemes, this new generation of DFT processors are more general (in that they handle complex input data), simpler to understand (since simple two's complement arithmetic is used throughout), and offer higher performance (2.9 million 16-bit DFT's per second are now possible compared to the 30,000 16-bit DFT's per second achieved by Kapoor).

Chapter 7 - Conclusion

A DFT design architecture has been presented which takes advantage of the factored form of DFT algorithms to rapidly implement DFT processors in VLSI circuit technology. The VLSI design methodology employs systolic summation arrays and bit-serial multiplier arrays which are fully pipelined for high performance and which can be automatically constructed using the layout generation programs *sumgen* and *multgen*.

Analysis of the DFT architecture showed that if the magnitude of the complex data inputs are bounded by $\frac{2^{p-1}}{N}$, then internal numerical overflow will be avoided and the magnitude of the complex DFT outputs will also be bounded by $\frac{2^{p-1}}{N}$.

It was also shown that the latency of the DFT processor will be $p + 1$ cycles/operation and that the performance (measured in p-bit N-point DFT operations per second) will be $\frac{f}{p+1}$, which is independent of N.

A 16-bit 8-point DFT processor was implemented as a VLSI chip using 2μm NWELL CMOS technology, which demonstrated that a 16-bit DFT processor of arbitrary size can be rapidly constructed which will perform about 2.9 million DFT operations per second.

APPENDIX

Summation Array Components

The *sumgen* program places standard cells side-by-side to construct summation array layouts. The pattern of the placements of the ADD, SUB, and DEL standard cells in the generated layout is directly related to the pattern of 1's, -1's, and 0's in the user-supplied input matrix. The layouts of the ADD, SUB, and DEL standard cells are shown in Figure 29, Figure 30, and Figure 31, respectively. These cells are designed to connect by abutment.

Since the full-adders in the ADD and SUB cells are in the critical delay path, they are implemented using pass-transistor logic in order to achieve high performance. It is well known that pass-transistor adders are difficult for most switch-level simulators to handle correctly[12], so special attention was paid to verify that the *irsim*[7] simulator handles these adders properly. The *irsim* simulator was used to perform the chip-level logic simulation of the 8-point DFT processor chip.

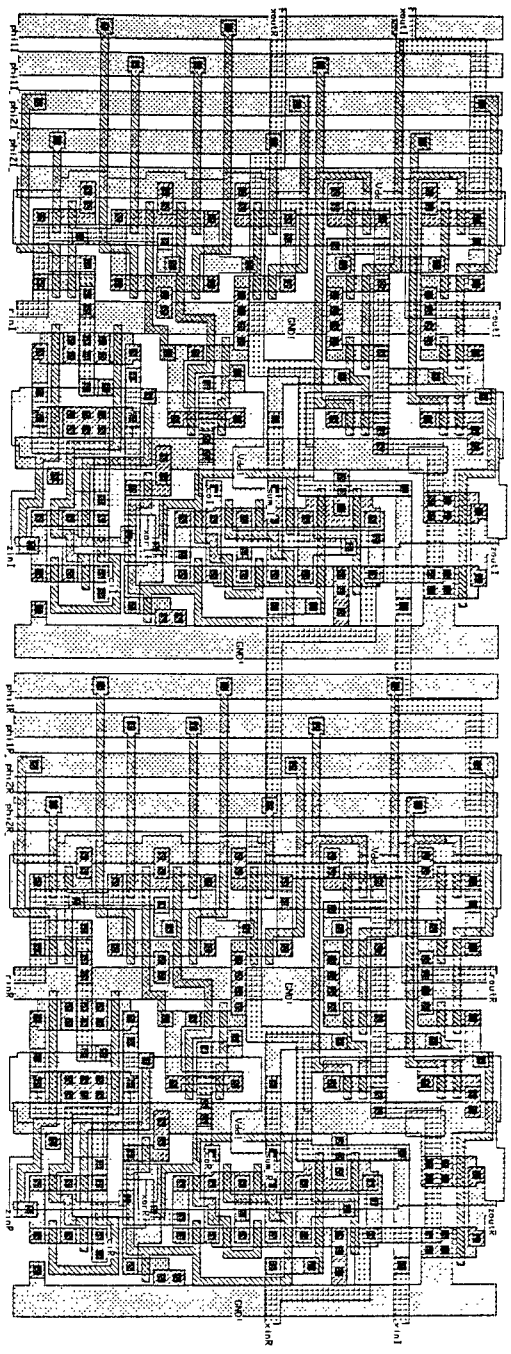
Note that all D-flip-flops are implemented as dynamic two-phase flip-flops so care must be taken to guarantee that the clock does not remain inactive for long periods of time or else the data stored in the dynamic flip-flops may become invalid.

The *sumgen* program also places a standard cell named RIN at the top of each column of the summation array to implement the required *rin* delay registers as shown across the top of Figure 7. In addition, the RIN cell automatically ties-off the *zin* input of the cell placed below it to ground as required. The layout of the RIN cell is shown in Figure 32.

Finally, the *sumgen* program also places the PHI1 standard cell at the top of each column of the summation array (on top of each RIN cell) and the PHI2 standard cell at the bottom of each column of the summation array. These cells implement the generation of two-phase clocking from single phase clocking[5].

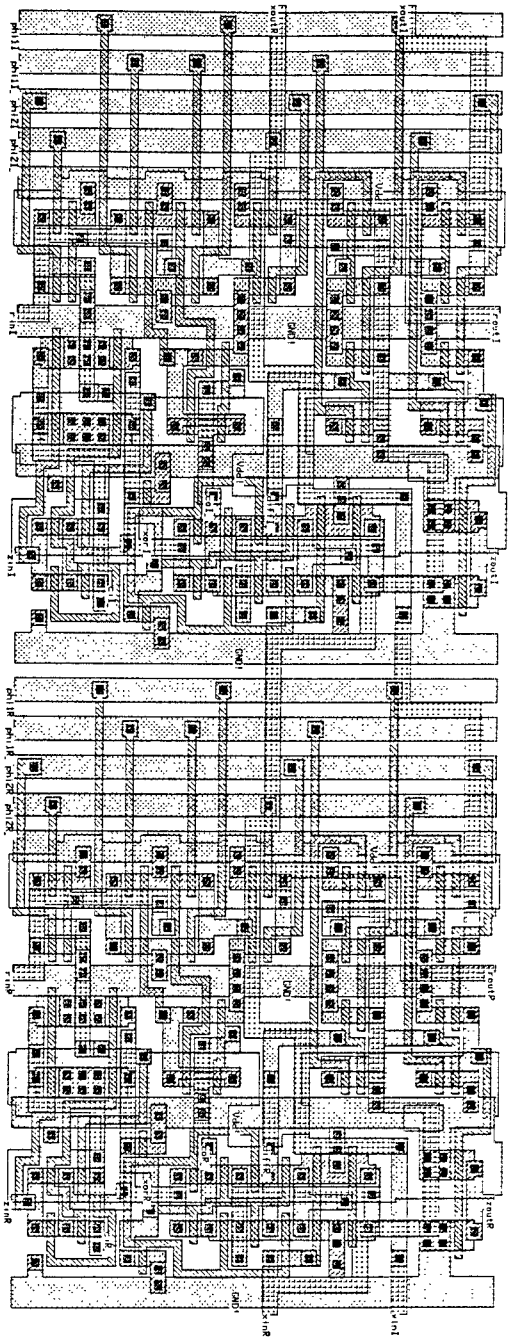
An example of the output of *sumgen* is shown in Figure 35. This simple summation array consists of a single adder and is intended to show with fair detail how *sumgen* constructs summation arrays. The layout consists of a PHI1 cell placed above a RIN cell placed above an ADD cell placed above a PHI2 cell.

Figure 29 - ADD Layout



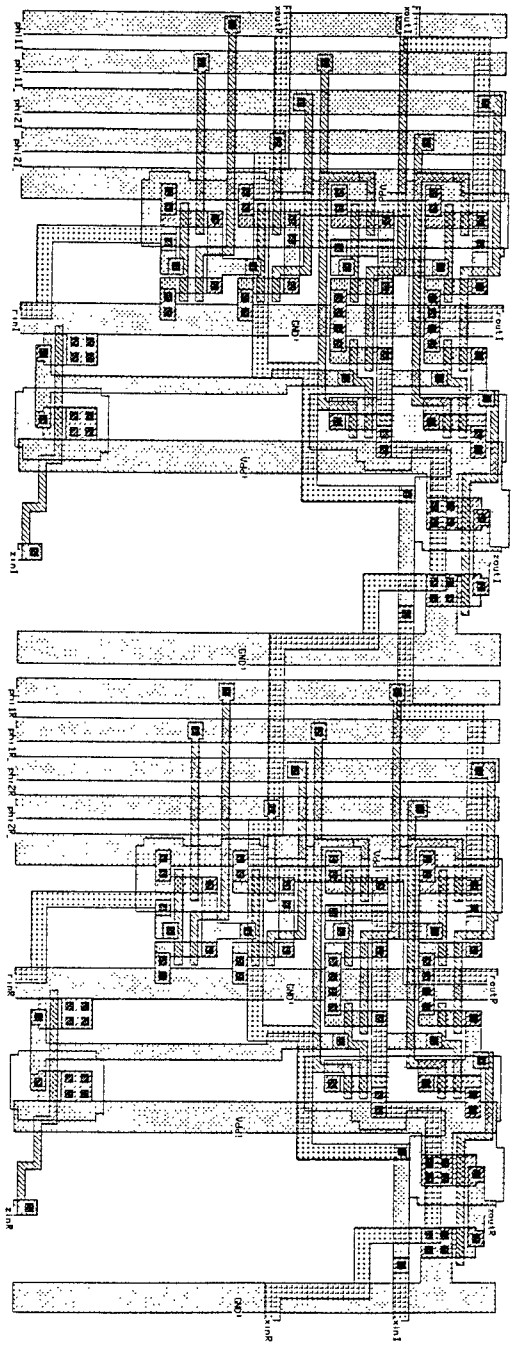
Height = 170λ, Width = 124λ

Figure 30 - SUB Layout



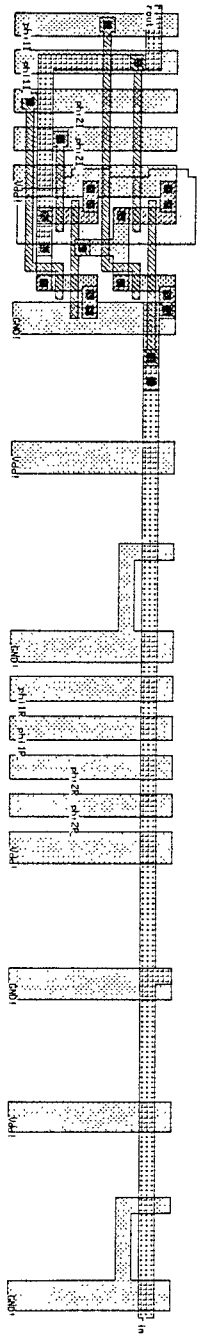
Height = 170λ, Width = 124λ

Figure 31 - DEL Layout



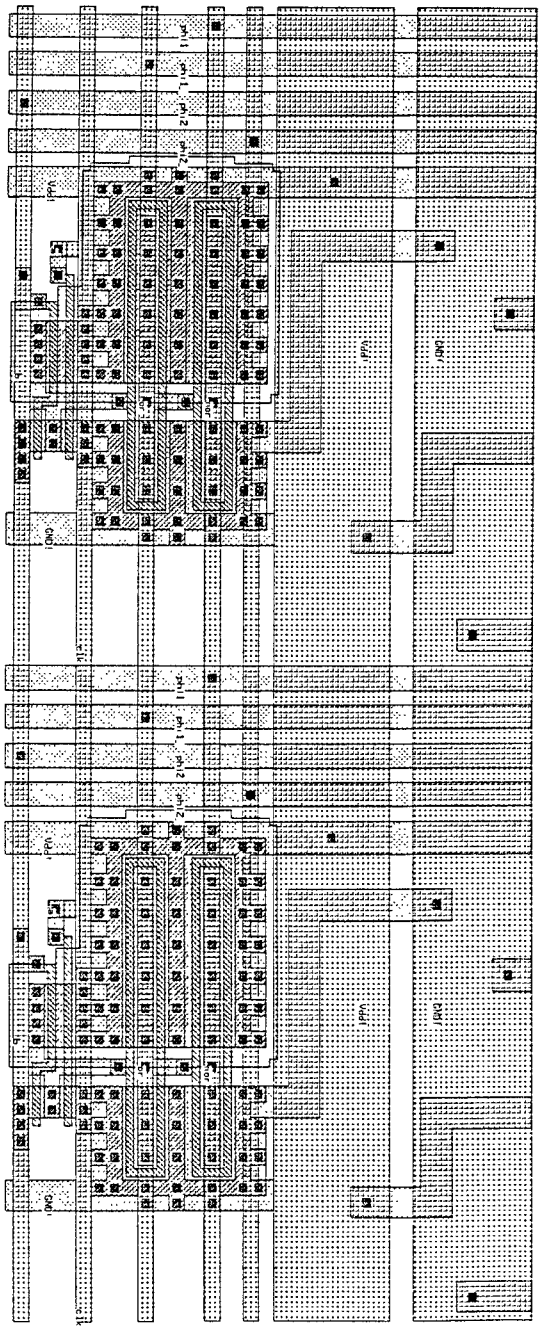
Height = 170λ, Width = 124λ

Figure 32 - RIN Layout



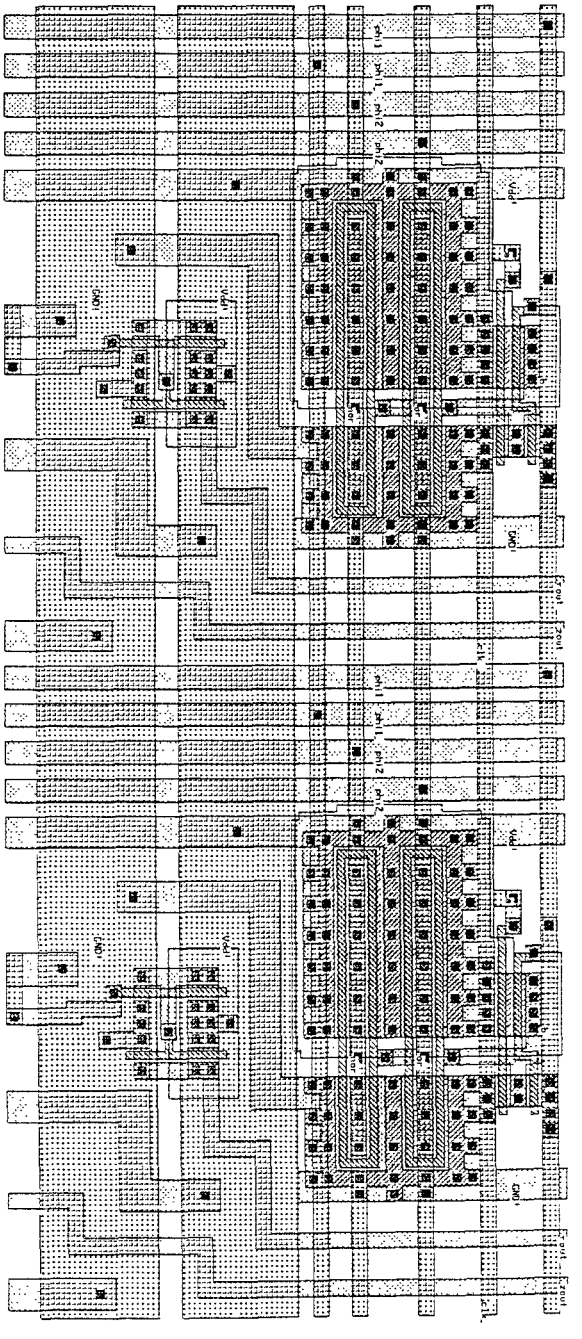
Height = 170λ, Width = 42λ

Figure 33 - Layout of PHI1 for Summation Arrays



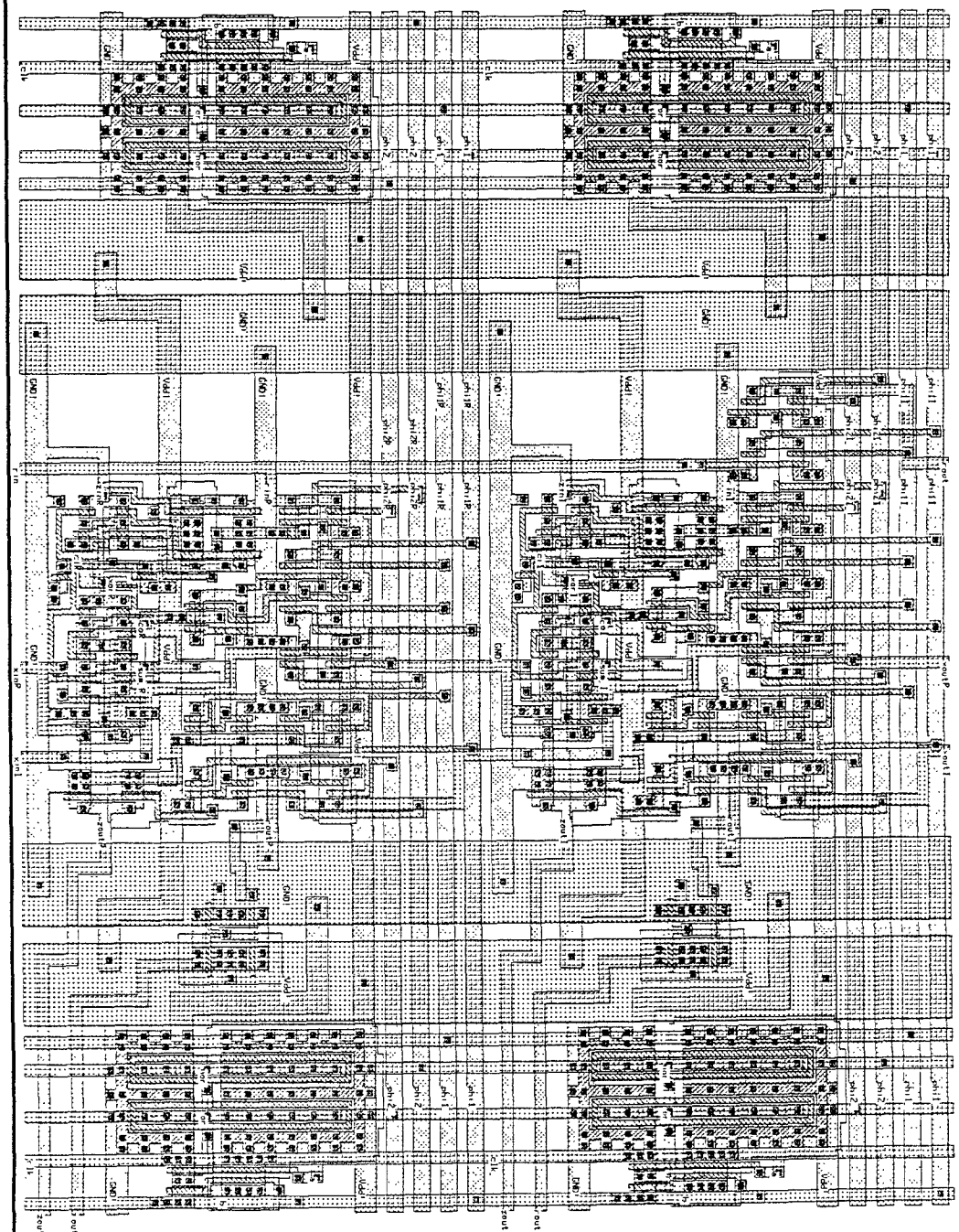
Height = 340λ, Width = 135λ

Figure 34 - Layout of PHI2 for Summation Arrays



Height = 340λ, Width = 143λ

Figure 35 - Example *sumgen* Output



Height = 444λ, Width = 340λ

Multiplier Array Components

The *multgen* program places standard cells side-by-side to construct multiplier array layouts. The pattern of the placements of the MULT1 and MULT0 standard cells in the generated layout is directly related to the pattern of 1's and 0's in the user-supplied multiplier coefficients. The MULT-1 standard cell is used as the final stage of any multiplier with a negative coefficient. The layouts of the MULT1, MULT-1, and MULT0 standard cells are shown in Figure 36, Figure 37, and Figure 38, respectively. These cells are designed to connect by abutment.

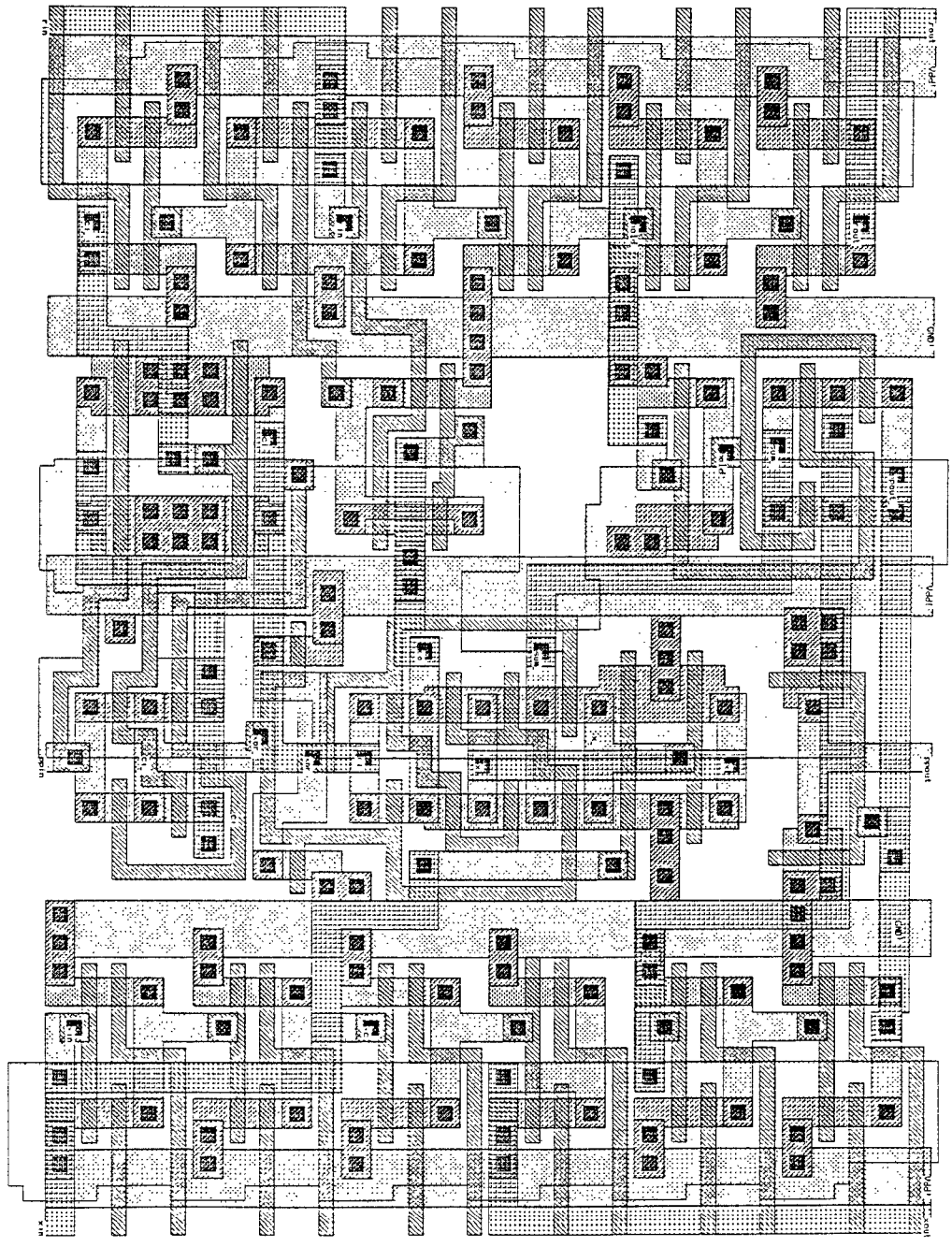
Since the full-adders in the MULT1 and MULT-1 cells are in the critical delay path, they are implemented using pass-transistor logic in order to achieve high performance. Note that all D-flip-flops are implemented as dynamic two-phase flip-flops.

The *multgen* program also places the PHI1 standard cell at the top of each column of the multiplier array and the PHI2 standard cell at the bottom of each column of the multiplier array. These cells implement the generation of two-phase clocking from single phase clocking[5] and, in addition, the PHI1 cell ties-off the *ppin* input of the first stage of each multiplier to ground as required.

Finally, the *multgen* program also places bus structures throughout the multiplier array to distribute the power, ground, and two-phase clock signals.

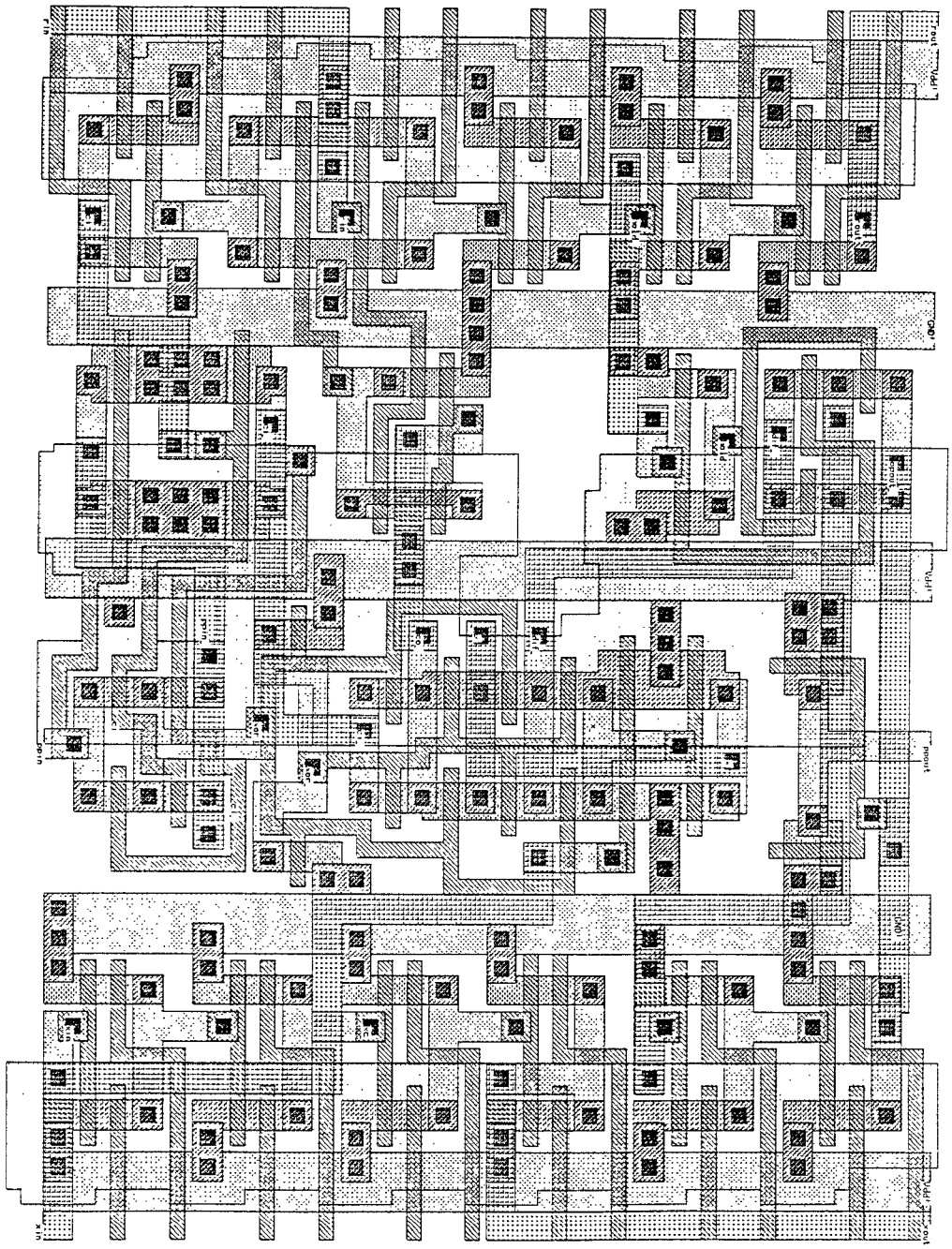
An example of the output of *multgen* is shown in Figure 41. This simple multiplier array consists of two 3-bit multipliers and is intended to show with fair detail how *multgen* constructs multiplier arrays. The coefficient of the left multiplier is .011, and the coefficient of the right multiplier is .110. Since the right multiplier has a negative coefficient (equal to -0.25), its final stage (which is at the bottom) is a MULT-1 cell instead of a MULT1 cell.

Figure 36 - MULT1 Layout



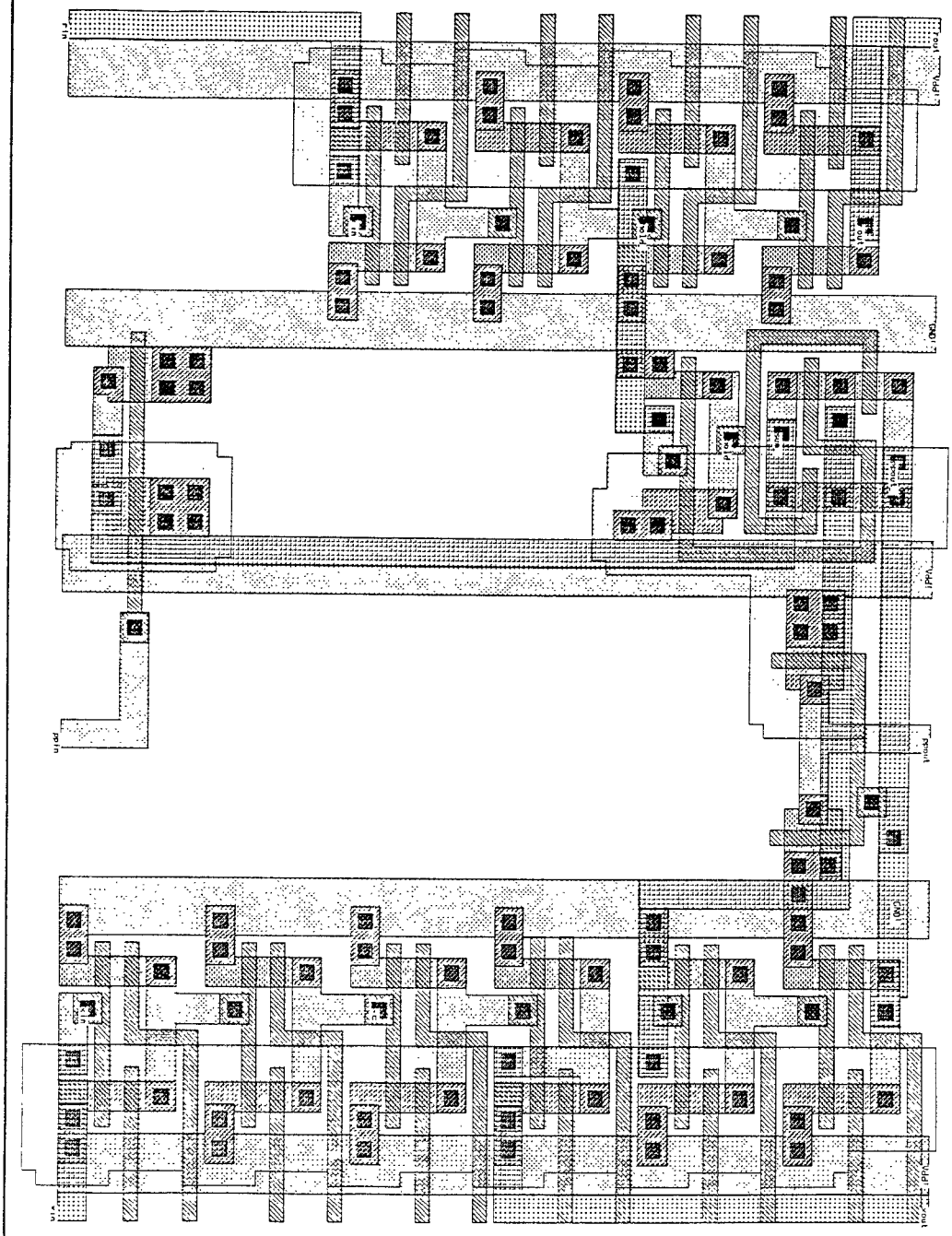
Height = 169λ, Width = 120λ

Figure 37 - MULT-1 Layout



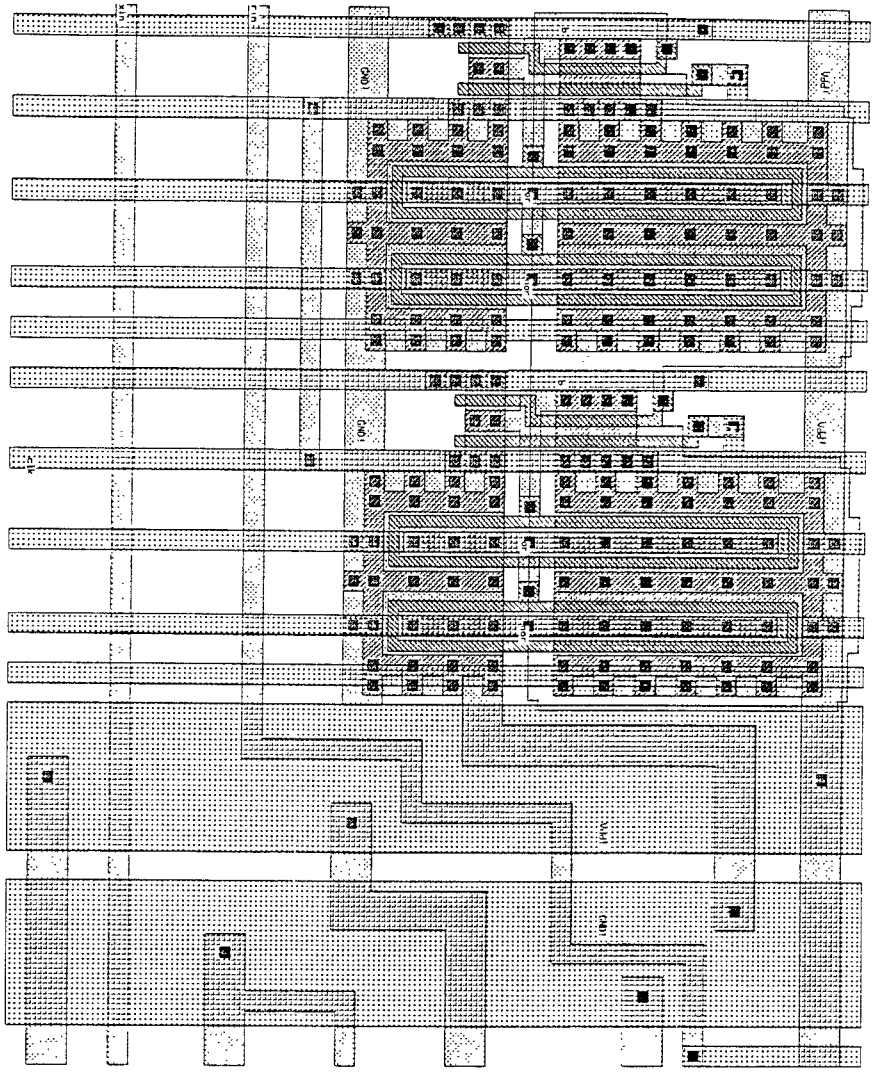
Height = 169λ, Width = 120λ

Figure 38 - MULT0 Layout



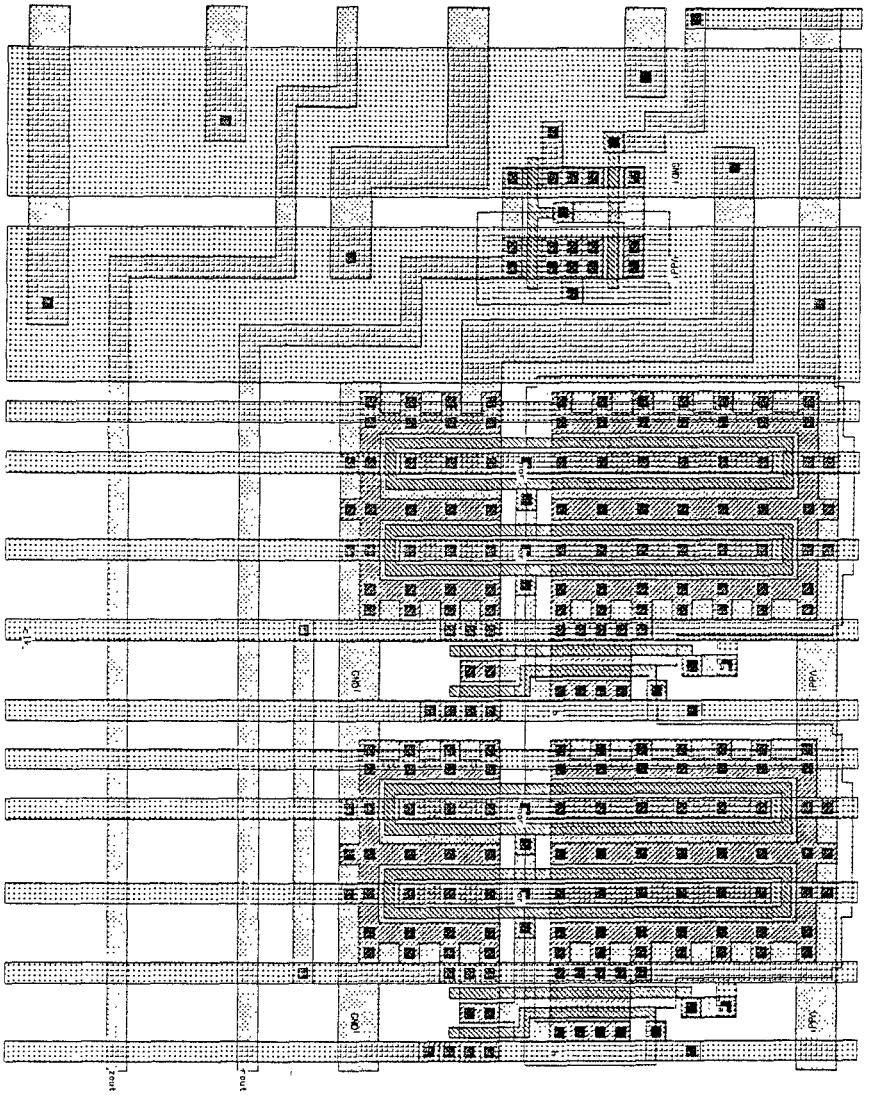
Height = 169λ, Width = 120λ

Figure 39 - Layout of PHI1 for Multiplier Arrays



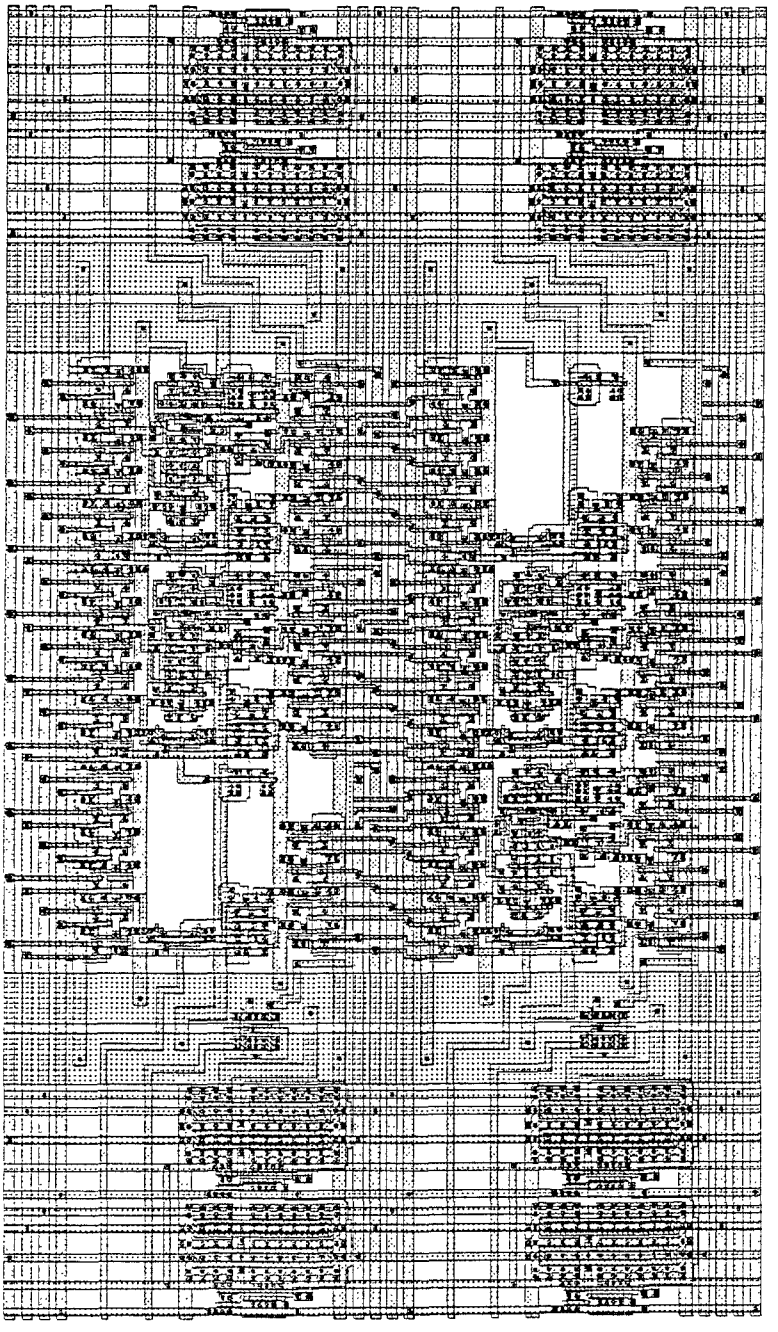
Height = 212λ, Width = 169λ

Figure 40 - Layout of PHI2 for Multiplier Arrays



Height = 212λ, Width = 169λ

Figure 41 - Example *multgen* Output



Height = 784λ, Width = 446λ

REFERENCES

1. Robert Michael Owens and Joseph F. Ja'Ja', "A VLSI Chip for the Winograd/ Prime Factor Algorithm to Compute the Discrete Fourier Transform", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34, No. 4, August 1986.

2. Anil K. Kapoor, "VLSI Implementation of Winograd/Prime Factor Algorithms to Compute the Discrete Fourier Transform", Masters Thesis in Electrical Engineering, University of Maryland, 1986.

3. C. Chakrabarti, VLSI Architectures for Real-Time Signal Processing, Doctoral Thesis in Electrical Engineering, University of Maryland, 1990.

4. Douglas F. Elliot and K. Ramamohan Rao, Fast Transforms: Algorithms, Analysis, Applications, Academic Press, New York, 1982, pp. 127-136.

5. C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley Publishing Company, Reading, Massachusetts, 1980, pp. 229-233.

6. Todd A. Goodrich, "Sumgen - An Automatic Summation Array Generator", VLSI Design Lab, University of Maryland at College Park, 1990.

7. Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, and Gordon D. Hamachi, 1990 DECWRL/Livermore Magic Release, Digital Equipment Corporation, Lawrence Livermore National Labs, Stanford University, University of California at Berkeley, Sept. 1990.

8. R. F. Lyon, "Two's Complement Pipeline Multipliers", IEEE Transactions on Communications, April 1976.

9. Gary L. Baldwin, Bernard L. Morris, David B. Fraser, and Angelo R. Tretola, "A Modular, High-Speed Serial Pipeline Multiplier for Digital Signal Processing", *IEEE Journal of Solid State Circuits*, Vol. SC-13, No. 3, June 1978.

10. Todd A. Goodrich, "Multgen - An Automatic Multiplier Array Generator", VLSI Design Lab, University of Maryland at College Park, 1990.

11. John P. Hayes, Computer Architecture and Organization, McGraw-Hill, New York, 1988, pp. 605.

12. C. Svensson and R. Tjärnström, "Switch-Level Simulation of the Pass Transistor Exor Gate", *IEEE Transactions on Computer-Aided Design*, Vol. 7, No. 9, September, 1988.