Lawrence Berkeley National Laboratory
Lawrence Berkeley National Laboratory

Title
AN ULTRAFAST FOURIER TRANSFORM PARALLEL PROCESSOR

Permalink
https://escholarship.org/uc/item/7k6746xv

Author
Greenberg, W.L.

Publication Date
2013-06-27

eScholarship.org Powered by the California Digital Library
University of California

UC-32
LBL-10897
c.1

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

RECEIVED
LAWRENCE
BERKELEY LABORATORY

OCT 21 1981

LIBRARY AND
DOCUMENTS SECTION

AN ULTRAFAST FOURIER TRANSFORM PARALLEL PROCESSOR

William L. Greenberg
(Ph.D. thesis)

April 1980

For Reference

Not to be taken from this room

Donner Laboratory

Biology & Medicine Division

LBL-10897 c.1

Prepared for the U.S. Department of Energy under Contract W-7405-ENG-48

LEGAL NOTICE

This book was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Lawrence Berkeley Laboratory is an equal opportunity employer.

LBL-10897

AN ULTRAFAST FOURIER TRANSFORM PARALLEL PROCESSOR

William L. Greenberg

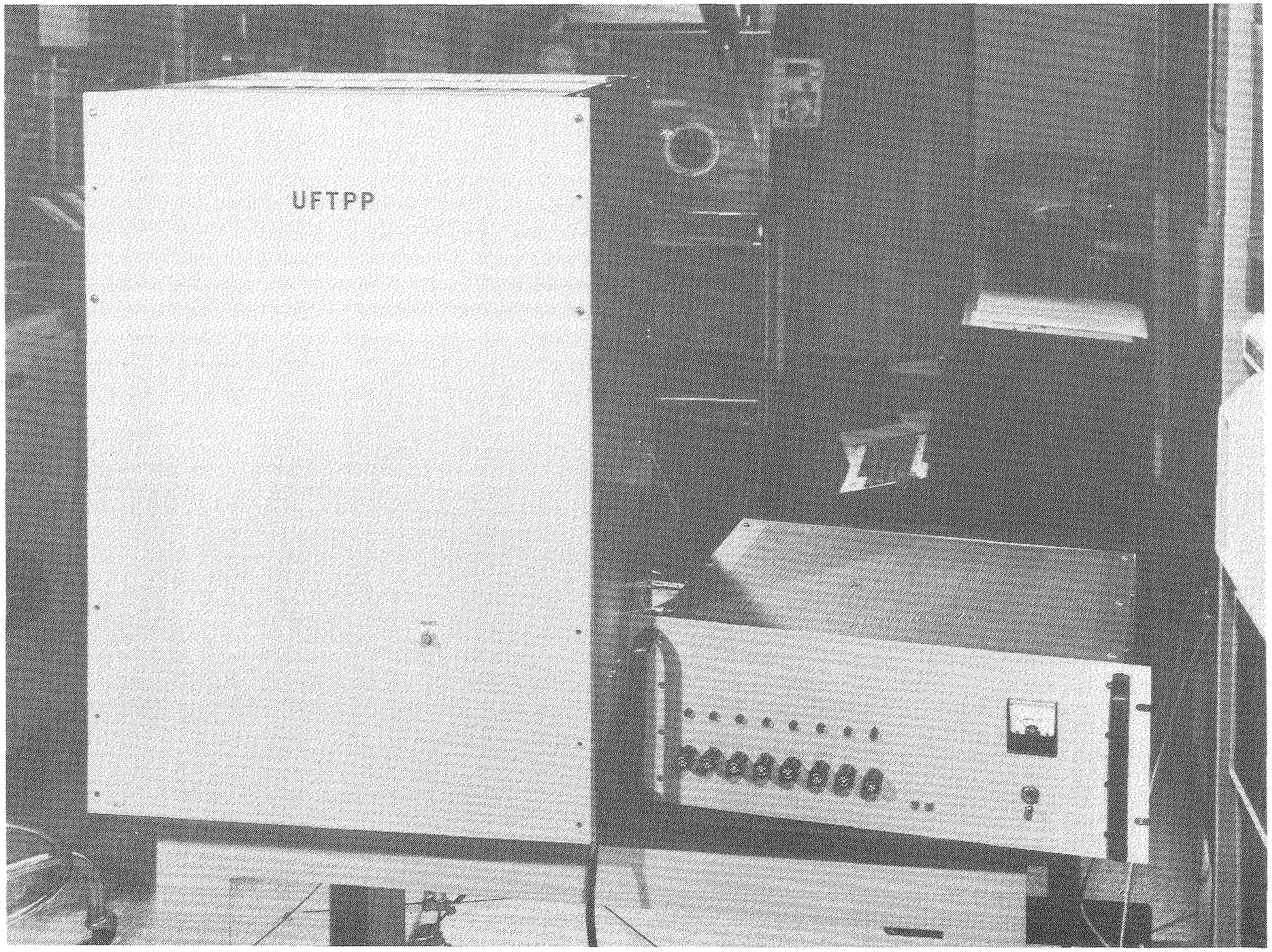
Ph.D. Thesis

April 1980

Department of Electrical Engineering & Computer Science
and
Biology and Medicine Division
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

This work was supported by the U.S. Department of Energy under Contract W-7405-ENG-48 and the National Institutes of Health.

1.



CBB804-4796

ABSTRACT

AN ULTRAFAST FOURIER TRANSFORM PARALLEL PROCESSOR

William L. Greenberg
Ph.D.

Department of Electrical Engineering & Computer Science
and
Donner Laboratory (Biomedical Division of Lawerence Berkeley Laboratory)
University of California, Berkeley, CA 94720
Sponsors: Department of Energy and the National Institutes of Health

A new, flexible, parallel-processing architecture is developed for a high-speed, high-precision Fourier transform processor. The processor is intended for use in 2-D signal processing including spatial filtering, matched filtering and image reconstruction from projections.

The flexible architecture is developed from the fully parallel architecture proposed by Pease. It is shown that the "perfect shuffle" data routing used by Pease may be used in a serial processor by use of a perfect shuffle generating function which generates the output address of each datum from its input address. Further development of this idea shows that the simplified architecture allowed by use of the perfect shuffle lends itself well to pipelining to effect horizontal parallel processing and increase execution speed. Finally, it is demonstrated that several (up to N/r, where N is the transform size and r is the radix of the transform) pipelined arithmetic units may be put in parallel operation to further increase the speed of transform computation. Important design parameters such as the radix of the transform, number of arithmetic units, number representation and word size are examined with respect to their effect on accuracy, dynamic range, transform computation speed and memory fragmentation.

The Ultrafast Fourier Transform Parallel Processor (UFTPP) architecture is compared and contrasted with the classical architectures (serial, cascade, parallel and array) and is shown to have considerable advantages in most cases.

To demonstrate the feasibility of implementation of a processor designed using this architecture, a one-arithmetic unit version has been constructed which can compute up to a 4096 complex point transform with maximum throughput of 500,000 samples per second. A software system was also implemented that uses the UFTPP to perform 2-D spatial filtering, execute matched filtering between a scene and a template and reconstruct images from their projections using the Backprojection of Filtered Projections (BFP) reconstruction algorithm. Two new techniques for determining filters to be used with the BFP algorithm are cited. The first new filter, proposed by Gullberg, incorporates attenuation correction of emission data from Computer Assisted Tomography (CAT) studies into the reconstruction operation. The second, proposed by Tsui, uses a stochastic filter to allow considerable dose reduction in CAT. Both require computation of a large number of Fourier transforms and are excellent applications for a high speed processor such as the UFTPP.

In summary, the outstanding characteristics of the UFTPP are:

(1) A pipelined "butterfly" computation module

(2) Flexible parallel processing architecture easily expandable to achieve extremely high speeds.

(3) Use of recently available LSI multiplier chips

(4) Use of the "perfect shuffle" data routing algorithm

(5) Extremely simple control logic

(6) Low cost

Approved:

Thomas F. Budinger

Thomas F. Budinger
Chairman, Dissertation Committee

TABLE OF CONTENTS

1.0 Introduction 1

1.1 Previous Work 1

1.2 New Work 3

2.0 FFT Derivation and Modification 8

2.1 Cooley-Tukey FFT Derivation 8

2.2 FFT Modification for Parallel Processing 10

2.3 The Perfect Shuffle Generating Function 15

3.0 Architecture 20

3.1 Arithmetic Unit Implementation for Different Radices 20

3.2 Perfect Shuffle Data Handling 23

3.3 Pipeline Architecture for UFTPP (Horizontal Parallel Processing) 24

3.4 Multiple Arithmetic Units (Vertical Parallel Processing) 26

3.5 Comparison of the UFTPP Architecture with Other Architectures 29

4.0 Number Representation & Arithmetic 32

4.1 Fixed Point vs. Floating Point Representation 32

5.0 Processor Implementation 41

5.1 Overall System Description 41

5.2 The Arithmetic Unit 43

5.3 The Memory System 49

5.4 The Multiplexer System 52

5.5 The Pipeline 55

5.6 Overall Timing and Control 56

5.6.1 The Perfect Shuffle Address Generator 57

5.6.2 Control/Status Register and the N - Register 57

5.6.3 The Iteration Counter 58

5.6.4 The Finite State Generator 58

5.6.5 Memory Counter 59

5.6.6 Clock Generation and Distribution 59

5.7 Computer I/O Interfaces 61

5.8 Computer Operation and I/O 63

5.9 Implementation Costs 64

6.0 Applications 65

6.1 Implementation 65

6.2 Image Processing Applications 65

6.2.1 High & Lowpass Filtering 65

6.2.2 Matched Filtering 67

6.3 Image Reconstruction from Projections 70

7.0 Conclusion 78

REFERENCES 81

ACKNOWLEDGEMENT

I would like to thank Professor Thomas F. Budinger for his invaluable advice, encouragement and support in what sometimes seemed like a never ending endeavor. His confidence in me provided the momentum to carry me over the "inertia" barriors and pushed me to strive for excellence. For this I am truly grateful.

I would also like to thank Professor Martin Graham for lending his hardware expertise to the solution of some of my worst problems and Profesor Walter Freeman for his valuable comments at the thesis stage.

Certainly the least formal, but possibly some of the most valuable comments on the project were received in conversations with Drs. Ron Huesman and Steve Derenzo and Mr. John Cahoon.

Probably the most time consuming portion of this project was the actual wiring of the circuit boards. I would like to express my deep appreciation to my "technical support" group": Mimi Winer, Peggy Eisenbach, Valerie Budinger, Jamie Halpern and Kelly Finnerty (all of whom donated their time). I would also like to thank Mimi and Peggy for their help in preparation of the thesis.

In addtion to her technical assistance, I would like to thank Mimi for her incredible patience and support in our relationship which sufferred the brunt of my frustration on many occasions.

Finally I would like to express my deep gratitude to the entire Research Medicine group at Donner Laboratory for providing me with the excellent experience in scientific research that I have gained during the past 4 3/4 years. Further, in all sincerety, the comaraderie among the members of the group has made the job truly enjoyable. The friendships I have developed will be sorely missed.

This work was supported by the U.S. Department of Energy under Contract W-7405-ENG-48 and the National Institutes of Health.

GLOSSARY OF TERMS

Array architecture -- a Fourier transform processor architecture with $N/r \log_r N$ arithmetic units. After a latency of $\log_r N$ cycles, a new transform is completed every cycle. Data routing may be by the Cooley-Tukey algorithm or by the perfect shuffle algorithm. The transforms are fixed length.

"Butterfly" operation -- the basic computation performed in Fourier transformation consisting of a complex multiplication followed by a complex addition and a complex subtraction. The name is derived from the patterns present in the signal flow diagrams of the Fourier transform.

Cascade architecture -- a Fourier transform processor architecture containing $m=\log_r N$ arithmetic units arranged sequentially. The output from an arithmetic unit becomes the input to the next. A given unit processes points separated by r^{-i} where i is the position of the arithmetic in the cascade. After a latency of N cycles, frequency coefficients begin to emerge from the output one per cycle. From this point on computation may be continuous. A new time or space sample may be input each cycle and a new frequency coefficient read from the output each cycle. The size of transforms computed is fixed.

Parallel architecture -- a Fourier transform architecture with N/r arithmetic units. Transforms are computed every $\log_r N$ cycles. Data routing is via the perfect shuffle algorithm. Size of the transform is fixed.

Perfect shuffle -- an algorithm used for pairing the correct data values at each iteration in Fourier transform computation which maintains a uniform manipulation of data throughout all iterations as opposed to the Cooley-Tukey data routing which changes at each iteration. It is called the perfect shuffle since it is similar to shuffling a deck of playing cards.

Serial architecture -- an FT processor architecture containing one arithmetic unit. It processes the data exactly as the Cooley-Tukey algorithm. A transform is computed in $N/r \log_r N$ cycles. It can be set up to process variable length transforms.

"Twiddle factor" -- in the butterfly operation one of the complex inputs is a value composed of a cosine term in the real part and a sine term in the imaginary part. This complex value, known as the "twiddle factor" multiplied times one of the input data values.

UFTPP architecture -- a flexible Fourier transform processor architecture with a variable number of arithmetic units which computes a transform in

$$\frac{N}{r} \cdot \frac{\log_r N}{P} \text{ cycles.}$$

P is the number of arithmetic units. May be programmed to compute different length transforms. The maximum number of arithmetic elements is N/r.

CHAPTER 1

INTRODUCTION

Fourier analysis has long been recognized as a major technique of signal processing for accomplishing such tasks as filtering, cross-correlation, auto-correlation, power spectrum deternmination, and other computations involving frequency domain manipulations. Prior to the advent of digital computing machines, most Fourier analysis was done by analog methods. When calculating machines and computers began to become available, discrete methods for calculation of spectra became important. The discrete Fourier transform has long been known to mathematicians and engineers as an algorithm which computes the discrete spectrum of a sampled time or space varying signal at a speed proportional to N^2 (where N is the number of complex points in a transform). In 1965, Cooley and Tukey published their now famous paper [18] which gave an algorithm for computation of discrete spectra in a time proportional to $N \log_2 N$. This algorithm, now known as the Fast Fourier Transform (FFT), had been in use in various forms as early as 1904, but was not widely known until the Cooley-Tukey rediscovery in 1965.

The $N \log_2 N$ speed of the Cooley-Tukey algorithm showed a huge increase in efficiency over the N^2 algorithms; but even digital computers could not perform this number of operations fast enough to allow real-time computation (i.e. to compute a transform in the same amount of time as it takes to acquire the N sample points). This thesis gives a method for accomplishing the FFT on large data arrays using a hard-wired processor.

1.1. Previous Work

Almost immediately after the Cooley-Tukey algorithm was published, engineers began to explore designs for its implementation in a dedicated processor which had a cycle time short enough to allow real-time computation to be realized. Over the past ten years a great body of literature has accumulated in the area of hardware FFT processors. Initially, brute force approaches produced serial architectures [3,5,6,62] that iteratively processed the input data exactly as the software that preceded them. Subsequently, the serial architecture was improved to the cascade architecture, sometimes called a pipeline [5,6,37] (the term "pipeline" here is different from the pipelining technique for increasing execution speed that will be discussed). The cascade architecture contains $m=\log_r N$ stages each of which processes data in blocks of $r^{(m-i)}$ $(1 \leq i \leq m)$. The advantage of this architecture is that samples can be fed continuously into the input end and at fixed time later the transformed values emerge.

Parallel and array processor architectures which are faster than those just mentioned have been proposed by Pease [48] and Bergland [7]. However, these are much more complex and much more expensive due to the large amount of hardware necessary for their implementation.

One major obstacle impeding all of these architectures was the time and hardware complexity in performing multiplication of complex numbers. Algorithms for determining the product of two binary numbers were well known, but relatively slow compared to the desired cycle times of the processors. Several ingenious techniques were developed to avoid direct multiplications, notably the CORDIC technique [71] used by Despain [26] and a somewhat similar technique developed by Liu and Peled [45]. These schemes rely on bitwise generation of the products through various algorithms. Recently, LSI multiplier chips have become available commercially. These chips function at high speed and are relatively inexpensive in comparison to the cost of parallel implementations of the bitwise algorithms just mentioned. These chips remove the obstacle that the alternate techniques were trying to circumvent.

Other algorithms due to Rader and Brenner [57] and Winograd [74] have been developed which convert the complex multiplications into pure real or pure imaginary operations. While these are excellent techniques in architectures where multiplications are relatively expensive (in terms of time or hardware), the LSI multipliers make them less attractive than algorithms such as will be proposed in this thesis. The Winograd algorithm does not generate a scheme which allows reduction of hardware when the complex multiplications are done as a parallel operation since some of the operations are between two real numbers, some between two imaginaries and some between a real and an imaginary.

The design put forth in this proposal is based in good part on algorithmic concepts by Pease. Others [34,36,70,73] have discussed hardware implementations using Pease's ideas, but none has recognized its true flexibility and possibilities for modularity. Also, Corinthios [20,21,22] has used the formalism of Pease to implement an algorithm (previously noted by Cochran, Cooley, et. al. [16]) which does not produce the output in bit reversed order. However, the shuffle operation which affects data routing is different for each iteration of the transform, and the associated control circuitry is more complicated.

1.2. New Work

This thesis presents a new hardware implementation of a modification of the FFT algorithm. The processor exhibits the following characteristics:

1) A pipelined "butterfly" operation computation module

2) Parallel processing architecture easily expandable to achieve very high speeds (e.g. computation of a 60 million point transform in 10 seconds).

3) Use of recently available LSI multiplier chips.

4) Use of a "perfect shuffle" operation to accomplishing data routing.

5) Extremely simple control logic

6) Low cost

The intended application areas for the specific processor which was constructed as a demonstration of this design strategy are image processing and image reconstruction from projections. Image processing is the process of manipulating an image to obtain increased visual impact or to emphasize interesting structures in order that more information may be extracted from the image. Any linear modification of an image may be expressed as an operation on its frequency spectrum. Thus, Fourier filtering is a convenient tool to use in effecting the modifications since quantitative description of the changes (in terms of resolution, SNR, contrast texture, etc.) is easily obtained and interpreted through the transfer function of the filter. Up to the present, image processing investigators have been hindered by the inablility to perform complex image processing operations in a timely manner due to software Fourier transform implementations. However, a high speed hardware Fourier transform processor such as will be described should provide investigators with the means to carry out such work.

Some of the image processing operations that are easily carried out in the frequency domain are lowpass filtering to improve visual impact, highpass filtering for edge detection or deblurring and matched filtering for object identification and location.

Image reconstruction from projections has become a very important area since the advent of Computer Assisted Tomography (CAT) as a diagnostic medical tool. Rapid reconstruction (< 10 secs.) of large images (e.g. 256 x 256 pixels) from a large number of projections (e.g. 150) necessitates high-speed special purpose hardware. There are several algorithms for reconstruction which utilize the Fourier transform and, thus, could be implemented using a hardware processor. Further, it is of great interest in this field to perform dynamic imaging. That is, acquiring "snapshots" of an area of the body at several closely spaced (e.g. 100 msec) points in time to observe a system in operation. This increases the need for high data throughput and accentuates the need for high speed hardware.

In addition to two dimensional signal processing, a high performance Fourier transform processor such as will be described is useful for one-dimensional signal processing where the amount of computation necessary makes implementation impractical. Some of these areas are image reconstruction from projections chemical analysis using NMR techniques, NMR flow imaging and X-ray crystallography.

While Fourier analysis is an extremely powerful tool, the image processing investigator must remain aware of its limitations. Mathematically, there are functions which do not have Fourier transforms. There are 3 sufficient conditions for the existence of the Fourier transform of a function, f(x):

1) The integral of f(x) from $-\infty$ to $+\infty$ exists.

2) The number of discontinuities in f are finite.

3) f(x) is of bounded variation.

The first two conditions are self-explanatory. The third is a bit more complicated and its treatment is beyond the scope of this thesis. Suffice it to say that a function that is of bounded variation has a finite number of maxima and minima in any finite interval. Sin x^{-1} is a good example of a function <u>without</u> bounded variation. There are also functions whose Fourier transforms do not strictly exist, but are said to have transforms in the limit. Impulses and purely periodic functions are in this class.

Any physical waveform that can be measured has a Fourier transform by virtue of its physical existence. However, whenever mathematical modeling or analysis is used to represent a physical entity, one must be careful not to violate these conditions of existence.

Further, one must be careful to realize the assumptions that are made in applying Fourier analysis to a physical system. The most important is that the system is linear. That is, the output from the sum of a set of inputs is the same as the sum of their individual outputs. Secondly, the system must be time stationary. The action referred to above does not change with time.

Finally, the investigator must be aware of the rules which must be followed in moving from continuous Fourier analysis to the use of any of the discrete Fourier transform techniques. An example of this is the Shannon Sampling Theorem.

While real-time image processing is still a thing of the future, a system utilizing an Ultrafast Fourier Transform Parallel Processor (UFTPP) will increase processing speed by several orders of magnitude over software and at least an order of magnitude over other hardware transform devices. This increase in processing speed brings image processing to the interactive level. That is, an image processing investigator may be able to perform an operation on an image and see the results within a few seconds (e.g. ~10 sec.). This allows the investigator more continuity of thought and the ability to exercise subjective judgment much more easily than previously when several minutes of computation were necessary to perform a single operation on an image. Further, when the investigator arrives at an image which satisfies him/her, one can quantitatively describe what processing has been done by description of the various filters which have been applied.

A system such as the one described in this thesis might be utilized as shown in Figure 1. The UFTPP looks like any other peripheral to the central processor of the host computer; the programmer may view it as an extremely fast FFT subroutine. A typical image processing session may proceed as follows:

1) User sits down at the terminal and calls up an image stored on the host computer's mass storage.

2) The user inputs a frequency domain filter to the program (through use of a joystick, trackball, TTY, lightpen, etc.).

3) The computer Fourier transforms the image, multiplies it by the input filter and displays the results in about 10 seconds.

4) The user studies the result and instructs the computer whether to save this image or discard it or return to step 2 for further processing.

A system such as this may be realized for < $15K not including the host computer. The host computer need be nothing more than a small minicomputer (e.g. PDP-11/10) or even a microprocessor. Of course, the faster the host computer the faster the turnaround from image to image. Almost all of the processing time will be the I/O time to and from the UFTPP, disk and display. By todays technology standards, only such superminis as the VAX-11/780 will be able to keep pace with the UFTPP.

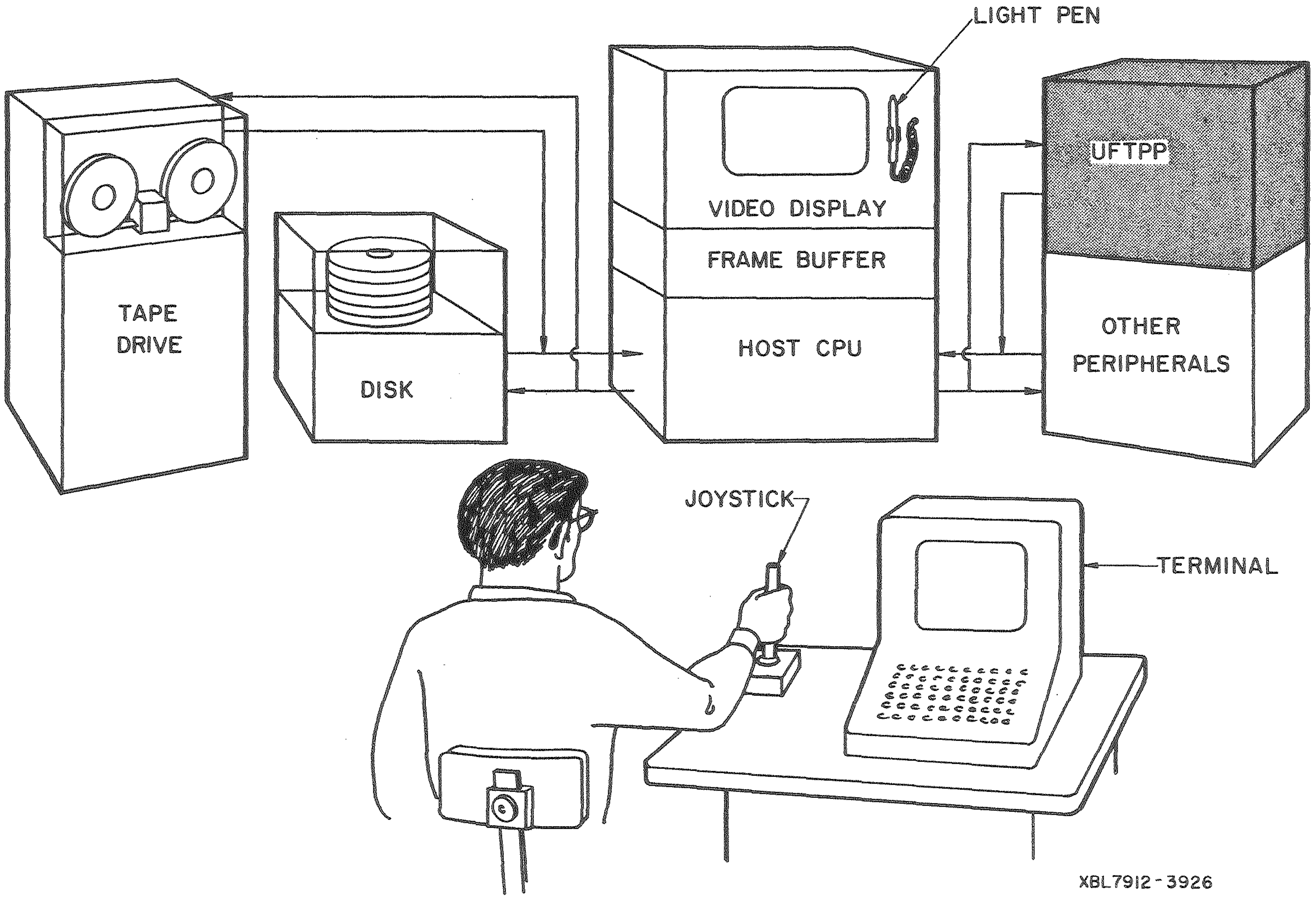

LIGHT PEN
TAPE DRIVE
DISK
VIDEO DISPLAY
FRAME BUFFER
HOST CPU
UFTPP
OTHER PERIPHERALS
JOYSTICK
TERMINAL
XBL7912-3926

Figure 1

CHAPTER 2

FFT DERIVATION AND MODIFICATION

2.1. Cooley-Tukey FFT Derivation

The continuous Fourier Transform, H(f), of a function h(t) is given by

$$H(f) = \int_{-\infty}^{\infty} h(t)e^{-j2\pi ft}dt \tag{1}$$

The discrete Fourier Transform is derived from the above by introduction of a real space sampling function, a real space window and a frequency space sampling function. The resulting expression for the discrete transform is

$$\begin{aligned} H(f/NT) &= \sum_{g=0}^{N-1} h(gT)\ e^{-j2\pi fg/N} \\ &= \sum_{g=0}^{N-1} h(gT)W^{fg} \end{aligned} \tag{2}$$

where

N = # of points sampled
T = interval between sample points
$f = 0, 1, 2, 3, \ldots, N-1$
$W = e^{-j2\pi/N}$

The FFT algorithm for computing (2) when $N=r_1 \cdot r_2 \cdot \ldots \cdot r_m$ is

$$H(n_0, n_1, \ldots, n_{m-1}) = \tag{3}$$

$$\sum_{k_0=0}^{r_m-1} \sum_{k_1=0}^{r_{m-1}-1} \ldots \sum_{k_{m-1}=0}^{r_1-1} h(k_{m-1}, k_{m-2}, \ldots, k_0)W^{nk}$$

where

$$n = n_{m-1}(r_1 r_2 \cdots r_{m-1}) + n_{m-2}(r_1 r_2 \cdots r_{m-2}) + \cdots + n_1 r_1 + n_0$$

$$k = k_{m-1}(r_2 r_3 \cdots r_{m-1}) + k_{m-2}(r_3 r_4 \cdots r_m) + \cdots + k_1 r_m + k_0$$

When $N=r^m$, then (3) simplifies to

$$H(n_0,n_1, \ldots ,n_{m-1}) = \tag{4}$$
$$\sum_{k_0=0}^{r-1} \sum_{k_1=0}^{r-1} \cdots \sum_{k_{n-1}=0}^{r-1} h(k_{m-1},k_{m-2}, \ldots ,k_0)W^{nk}$$

where

$$n=n_{m-1}r^{m-1}+n_{m-2}r^{m-2}+\ldots+n_0$$
$$k=k_{m-1}r^{m-1}+k_{m-2}r^{m-2}+\ldots+k_0$$

and

$$n_i= 0,1,2,\ldots,r-1.$$

Implementation on a binary digital computer is most simply done when $N = 2^m$ and (4) becomes

$$H(n_0,n_1, \ldots ,n_{m-1}) = \tag{5}$$
$$\sum_{k_0=0}^{1} \sum_{k_1=0}^{1} \cdots \sum_{k_{m-1}=0}^{1} h(k_{m-1},k_{m-2}, \ldots ,k_0)W^{nk}$$

where

$$n=n_{m-1}2^{m-1} + n_{m-2}2^{m-2} + \ldots + n_0$$
$$k=k_{m-1}2^{m-1} + k_{m-2}2^{m-2} + \ldots + k_0$$

and

$$n_i = 0,1.$$

It is easy to see that computation of (5) requires m iterations where the i^{th} iteration involves the intermediate results from the previous iteration. The intermediate results, $H_i(t)$, for the i^{th} iteration may be expressed as

$$H_i(n_0,n_1,\ldots,n_{i-1},k_{m-2},k_{m-3},\ldots,k_0) \qquad (6)$$

$$= [\sum_{k_{m-i}=0}^{r-1} h_{i-1}(n_0,n_1,\ldots,n_{i-2},k_{m-i},k_{m-i-1},\ldots,k_0)\cdot W^{n_{i-1}k_{m-i}N/r}]$$

$$\cdot W^{(n_{i-1}r^{i-1} + n_{i-2}r^{i-2} + \ldots + n_0)\,k_{m-i-1}r^{(m-i-1)}}$$

$$i = 1,2,\ \ldots\ ,m$$

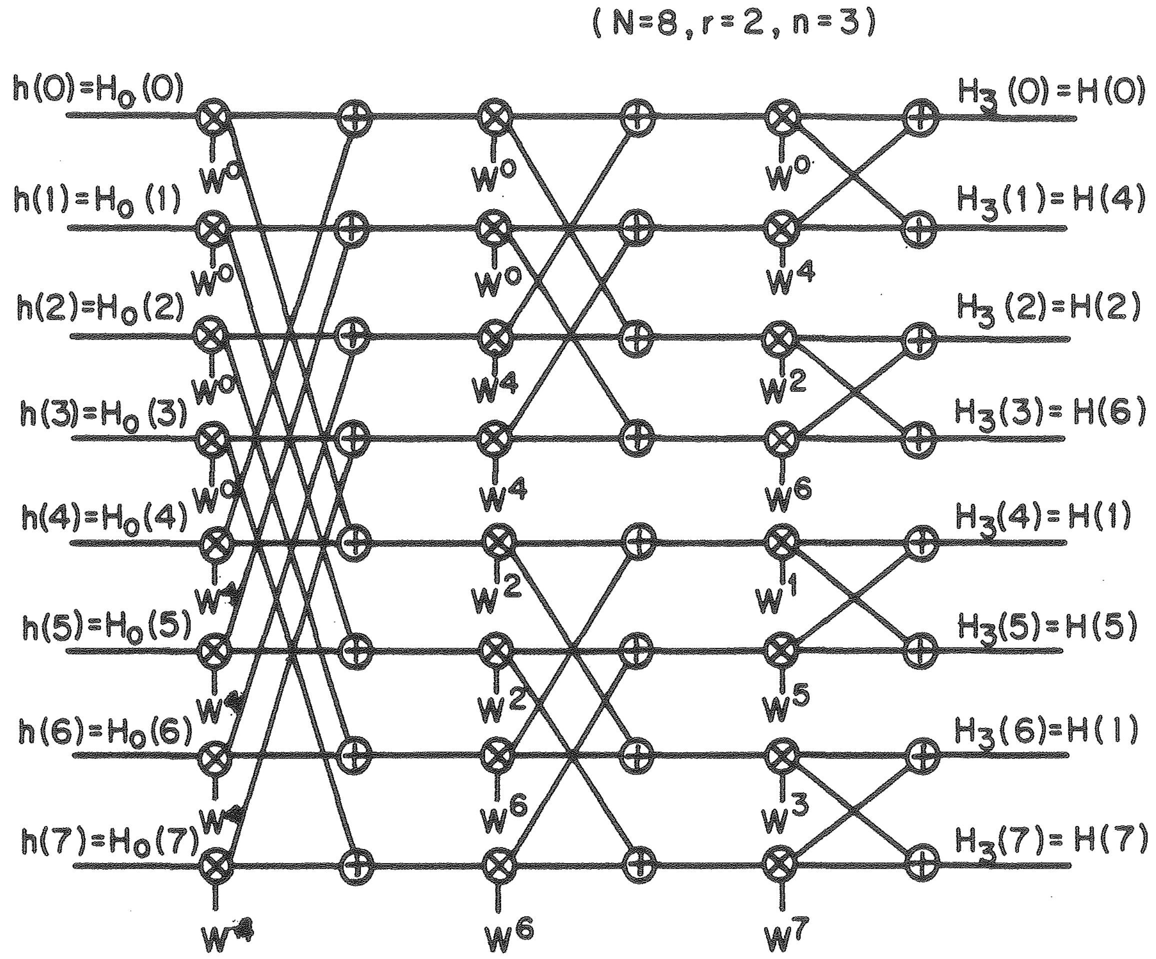
The expression inside the [] is the expression for a radix-r "butterfly" operation and the outside portion is known as the "twiddle factor" (after Gentleman & Sande [32]). This expression effectively says that each iteration in the calculation is an r-point "butterfly" applied to N/r sets of points appropriately chosen from the intermediate results of the previous iteration. The results of this calculation are then individually "twiddled" according to the outer expression. The radix, r, of the transform is chosen according to the type of computing power available. (This will be discussed in detail in chapter 3.) In the case of radix-2, the "butterfly" results in a simple addition and subtraction of the two complex numbers followed by multiplication of the difference by a twiddle factor. The twiddle factor for the sum is always one.

It is important to notice that the mth iteration produces the set $F_m(n_0,n_1,n_2,\ldots,n_{m-1})$ as a result, but the frequency coefficients in their correct order are $F_m(n_{m-1},n_{m-2},\ldots,n_0)$. In other words, the FFT algorithm produces its results in digit reversed order which must be compensated for. (This will be taken up in chapter 5.)

A graphic representation of the FFT algorithm for radix-2 is shown in Figure 2. This is known as the signal flow graph. Figure 3 shows the signal flow graph for the "twiddle factor" modification.

2.2. FFT Modification for Parallel Processing

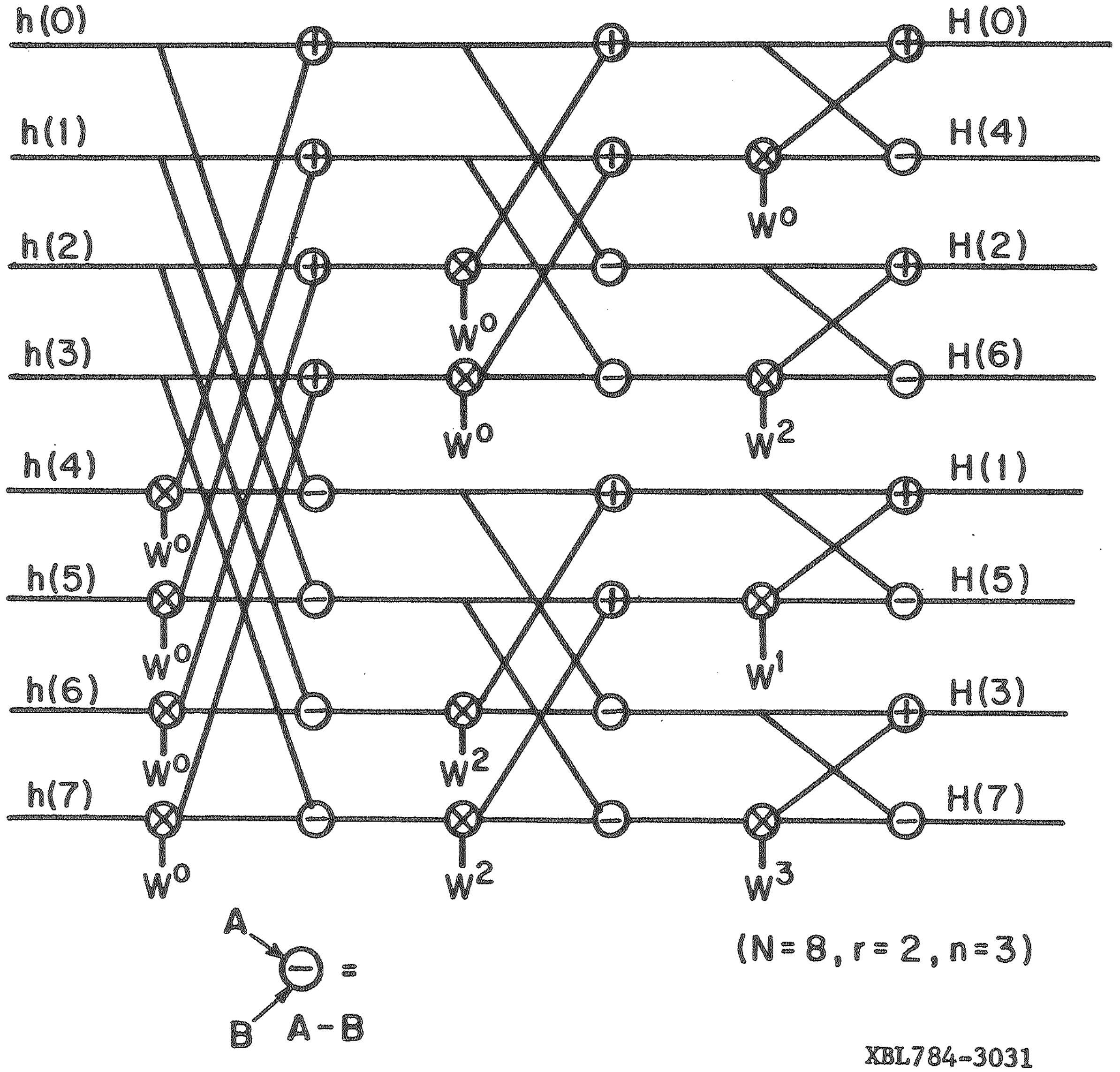
Pease [48] describes the DFT as a matrix operation and proceeds to derive the Cooley-Tukey FFT algorithm as follows. Let the DFT be expressed as:



XBL784-3030

Figure 2

Figure 3



XBL784-3031

$$g_r = \sum_{s=0}^{N-1} e^{-j2\pi rs/N} \cdot f_s \qquad r = 0,1,\ldots,N-1 \tag{7}$$

If we allow the matrix $T_{rs} = e^{-j2\pi rs/N}$ and the sets $\{g_r\}$ and $\{f_s\}$ to be $\overline{g}$ and $\overline{f}$, respectively, then (7) can be expressed as

$$\overline{g} = T_N \cdot \overline{f} \tag{8}$$

Since the FFT produces its output in scrambled order (digit-reversed) we must introduce an operator to reflect this:

$$T'_N = Q_N \cdot T_N \tag{9}$$

where Q_N is the digit reversal matrix (operator) for size N.

Pease shows that the key to the FFT lies in the factorization of T' as follows:

$$\begin{aligned} T'_N &= \begin{bmatrix} T'_{N/2} & T_{N/2}' \\ T'_{N/2}K & -T'_{N/2}K \end{bmatrix} \\ &= \begin{bmatrix} T'_{N/2} & 0 \\ 0 & T'_{N/2} \end{bmatrix} \cdot \begin{bmatrix} I & I \\ K_{N/2} & -K_{N/2} \end{bmatrix} \\ &= \begin{bmatrix} T'_{N/2} & 0 \\ 0 & T'_{N/2} \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ 0 & K_{N/2} \end{bmatrix} \cdot \begin{bmatrix} I & I \\ I & -I \end{bmatrix} \end{aligned} \tag{10}$$

where I is the identity matrix and K_L the diagonal matrix

$$\text{diag}(W^0, W^1, W^2, \ldots, W^{L-1}) \qquad (\text{again } W = e^{-j2\pi/N})$$

It is obvious that this process may be continued to any degree as long as N/2 is an integer. When the factorization is halted the subscript on the T' matrices of the left side of (10) is the radix of the transform referred to above.

At this point the Kronecker or direct product of matrices is used. The Kronecker product for two matrices $A = a_{ij}$ and $B = b_{kh}$ is defined as

$$A \times B = \begin{bmatrix} a_{00}B & a_{10}B & \cdots & a_{i0}B \\ a_{01}B & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ a_{0j}B & \cdot & \cdots & a_{ij}B \end{bmatrix}$$

The result has dimensions i*k by j*h. Thus, (10) may be expressed as

$$T'_N = (T'_{N/2} \times I_2)D_N(I_{N/2} \times T'_2) \qquad (11)$$

where D_N is the matrix quasi-diag (IK) and

$$T'_2 = \begin{bmatrix} W^0 & W^0 \\ W^0 & W^{N/2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

When the factorization is carried all the way down to a radix-2 transform (11) becomes, for $N = 2^n$,

$$T'_N = (T'_2 \times I_2 \times \cdots \times I_2)\cdot Q_1 \cdot (I_2 \times T'_2 \times \cdots \times I_2)\cdot Q_2 \cdot \qquad (12)$$
$$\cdots \cdot (I_2 \times \cdots \times I_2 \times T'_2)$$

where $Q_i = D_{r(i+1)} \times I_{r(n-i-1)}$.

As with regular matrix multiplication, the Kronecker product is not commutative. So, another operator is introduced which allows commutation of matrices under the Kronecker product. This operator, called the perfect shuffle operator, is defined as P_N such that

$$P_N \cdot \mathrm{col}(x_0, x_1, \ldots, x_{N-1}) = \mathrm{col}(x_0, x_{N/2}, x_1, x_{N/2+1}, \ldots, x_{N-1})$$

Using P_N, it can be shown that

$$I_2 \times T'_2 \times I_{N/4} = P_N(T'_2 \times I_{N/2})P_N^{-1} \qquad (13)$$

or

$$I_4 \times T'_2 \times I_{N/8} = P_N^2(T'_2 \times I_{N/2})P_N^{-2}$$

And, (12) can be rewritten as

$$T'_N = (T'_2 \times I_{N/2})\cdot Q_1 \cdot P(T'_2 \times I_{N/2})P^{-1} \cdot Q_2 \cdot \cdots \qquad (14)$$
$$\cdot P^{(N-1)}(T'_2 \times I_{N/2})P^{-(N-1)}$$

If we define $C = (T'_2 \times I_{N/2})$ and $Q_i = P^{(i-1)} E_i P^{-(i-1)}$ and notice that $P_N^n = P_N$ and $P_N^{-(n-1)} = P_N$ for $N = 2^n$, then we may write (14) as

$$T'_N = C\, P^0\, E_1\, P^{-0}\, P^1\, C\, P^{-1}\, P^1\, E_2\, P^{-1}\, P^2 \cdots CP^{(n-1)}\, E_n\, P^{-(n-1)}$$

$$= C\, E_1\, P\, C\, E_2\, P \cdots C\, E_n\, P \qquad (15)$$

And, finally

$$T'_{n=2^n} = \prod_{i=1}^{n} C\, E_i\, P \qquad (16)$$

Thus, the FFT may be computed employing only the C operator (the butterfly operator), the E_i operator (the twiddle factor operator) and the shuffle operator P. The importance of this representation is that the data routing is accomplished using the identical operator at each iteration. The new signal flow graph for this operation is shown in Figure 4. One will notice that this arrangement allows for the completely parallel computation of the FFT with data routing accomplished by hardwired paths as shown in Figure 5. This effectively reduces the computation time to the order of $\log_2 N$ since all N/2 2-point butterflies are accomplished in parallel. However, if a hardware implementation were designed, it would require N/2 complex multipliers and N complex adders. At the present state of the art, this is clearly too expensive.

2.3. The Perfect Shuffle Generating Function

Examination of Figure 5 shows that the parallel arithmetic units are identical and that a single one working successively on each pair of inputs could produce the results for that iteration. If it is possible to implement the perfect shuffle in a serial fashion, then an FFT processor with flexible parallel processing architecture and simplified control logic may be realized.

Close inspection of the perfect shuffle operator reveals that there is a perfect shuffle generating function, p(i), (i.e. a function which generates the index in the output vector of a given index from the input vector) given by

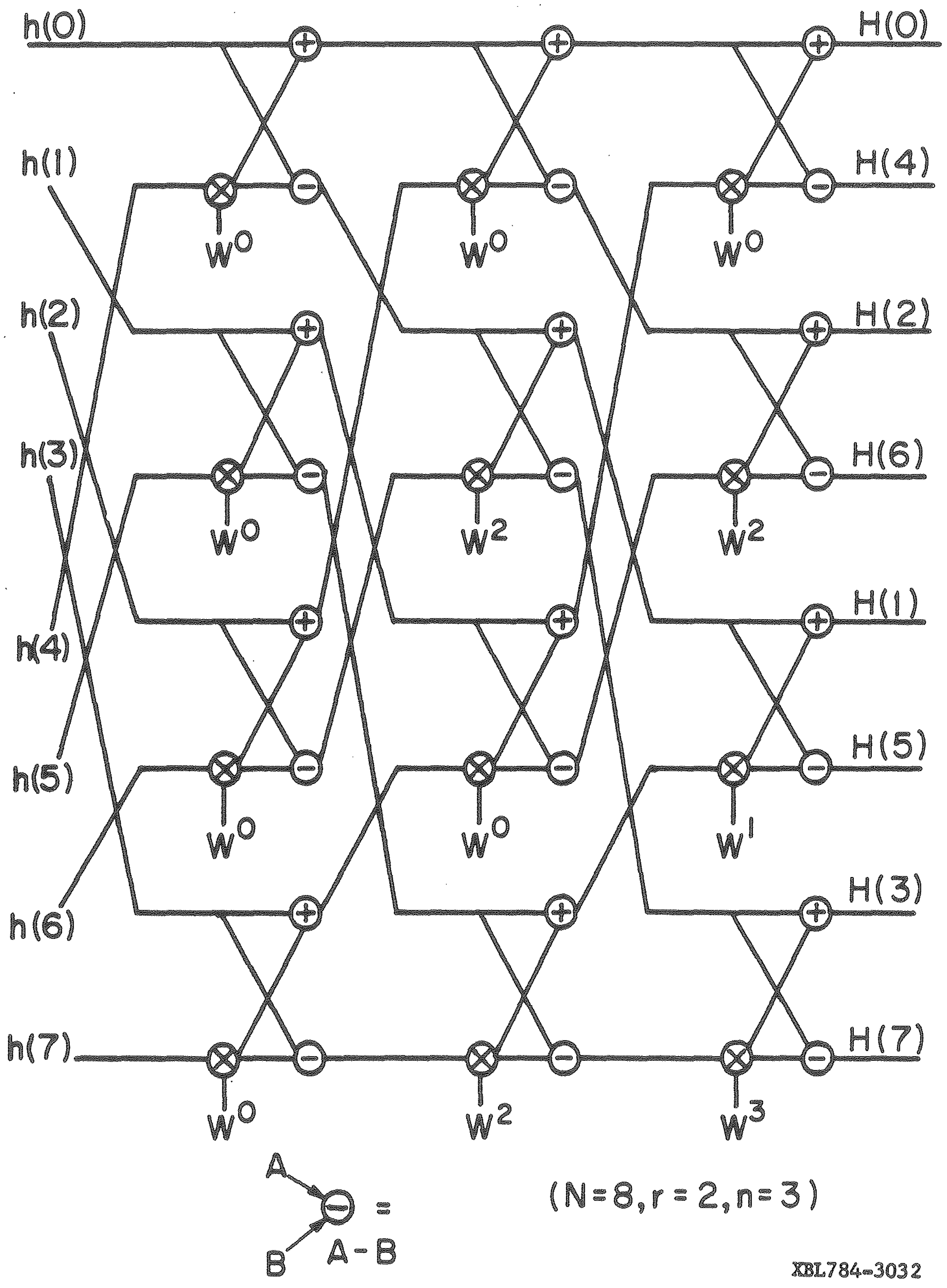
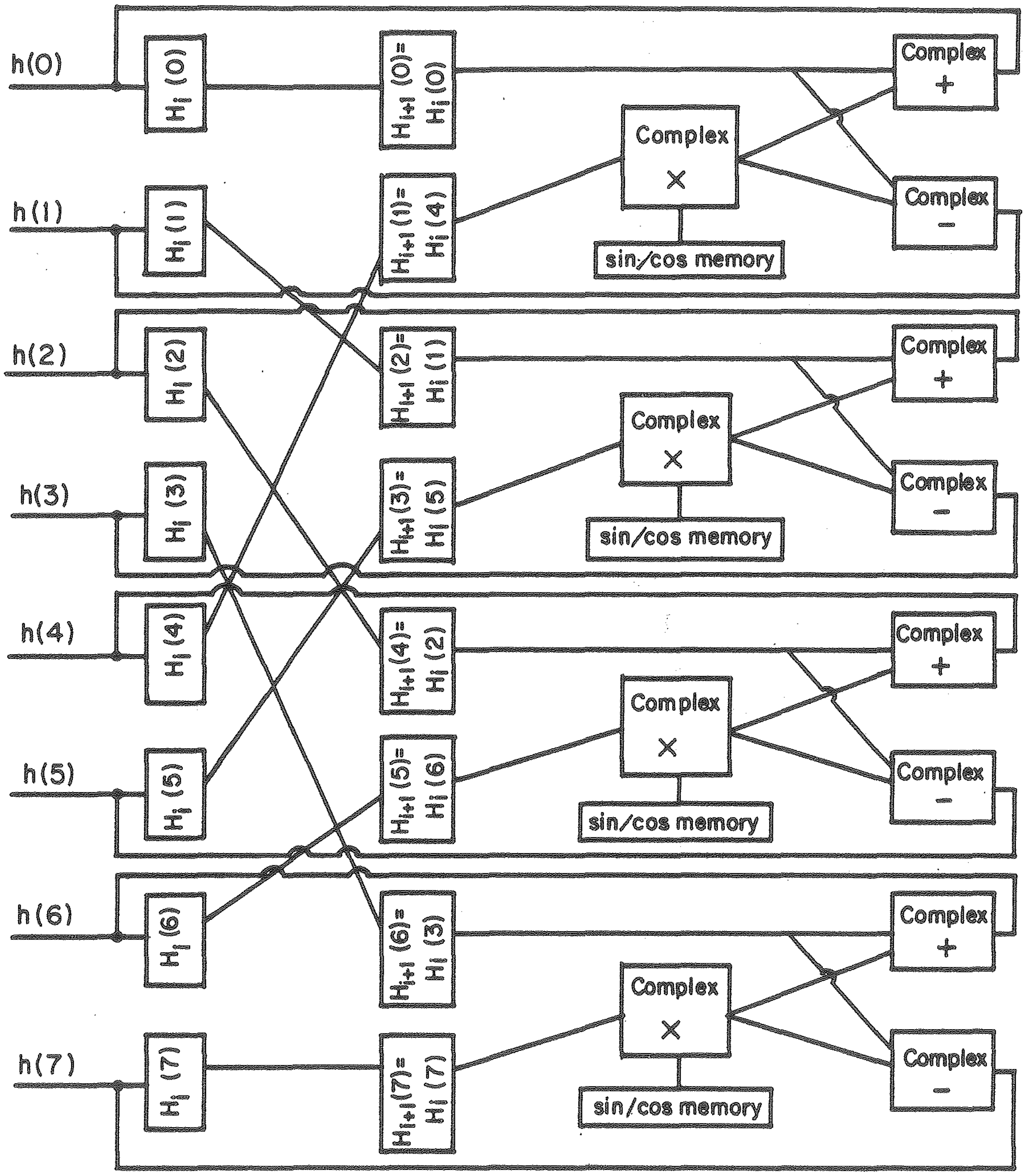

h(0)
h(1)
h(2)
h(3)
h(4)
h(5)
h(6)
h(7)
H(0)
H(4)
H(2)
H(6)
H(1)
H(5)
H(3)
H(7)
W0
W0
W0
W0
W2
W2
W0
W0
W1
W0
W2
W3
A
B
= A-B
(N=8, r=2, n=3)
XBL784-3032

Figure 4



XBL784-3033

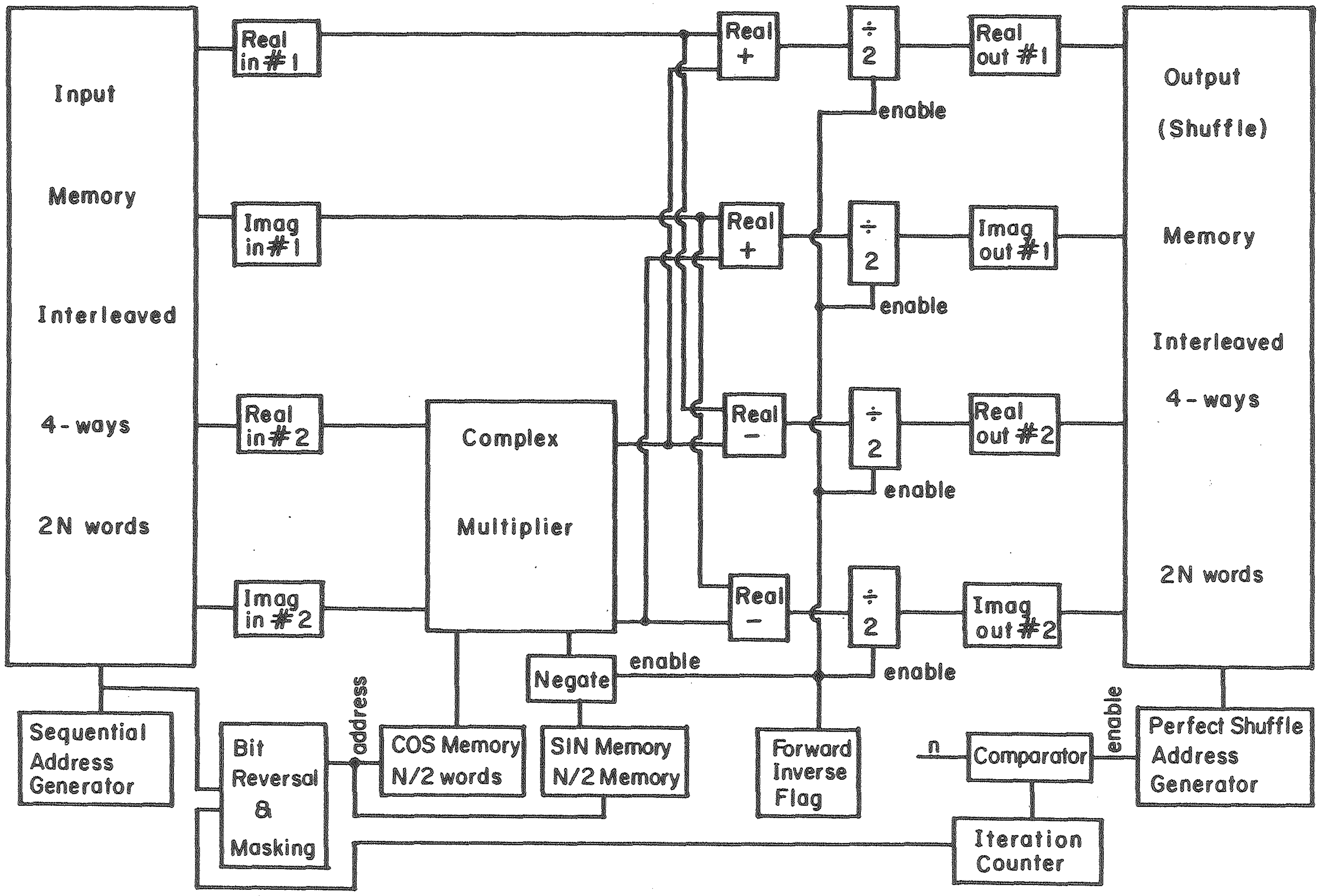
Figure 5

$$j = p(i) = i \cdot r - ((i \cdot r)/N \cdot N) + (i \cdot r)/N \qquad (17)$$

when r is the radix of the shuffle (and the transform) and N is the length of the vector being shuffled (i.e. the length of the transform). All operations are truncated integer arithmetic. In words, the input address i is multiplied by r modulo N to yield the output address j, which amounts to a left circular shift of i by r-1 bits.

By use of the perfect shuffle generating function, the results of the i^{th} iteration, H_i, may be determined serially from the results of the previous iteration, H_{i-1} (see Eq. 6). Figure 6 shows a diagram of a serial FFT processor which uses the perfect shuffle for data routing. Since the data routing does not change from iteration to iteration as in the Cooley-Tukey algorithm, this machine has very little control logic and a low parts count. And, as will be developed in the next chapter, lends itself very well to techniques for increasing execution speed such as pipelining and parallel processing.

Figure 6



XBL784-3034

CHAPTER 3

ARCHITECTURE

3.1. Arithmetic Unit Implementation For Different Radices

A major factor which impacts the architecture of a Fourier transform processor is the radix of the transform. Eq. 4 in chapter 2 gives the general expression for a radix-r transform (figure 3 shows the butterfly operations for an 8-point, radix-2 transform). In this section we will examine what the implications of that expression are in terms of hardware. The discussion will be limited to radices which are a power of two since the Cooley-Tukey algorithm is most easily implemented under that condition.

As noted in Chapter 2, there are r complex points involved in the computation of one "butterfly" at each node of the signal flow diagram for a radix-r transform. For maximum speed, this means that there must be r parallel data paths in the butterfly computation unit. Each datum must be correctly weighted and combined with the other weighted inputs to correctly compute the r-point transform dictated by T_r' (Eq. 11).

In order to compare hardware costs of various radix transforms, it is valuable to notice that any radix transform, T_r, may be further factored until it is expressed in terms of a radix-2 transform, assuming that $r=2^z$. In the same manner as Eq. 12, we can see that recursive application of Eq. 11 eventually will express the radix-r transform in terms of radix-2. Further, consider Eq. 13 and its application to the resulting expression. The final result is an expression which may be realized exactly as in the signal flow diagram shown in Figure 3. In fact, Figure 3 may be interpreted as either the signal flow diagram for a radix-2, 8-point transform of as the inside of a "butterfly" computation element for a radix-8 transform. The only difference is that the eight inputs to the radix-8 computation element are weighted according to the "twiddle factor" formula given in Eq. 6.

There are three important implications of this analysis. First, we note that a radix-r transformer may be implemented with r $\log_2 r$ complex adder/subtracter combinations (one complex adder and one complex subtracter is necessary to implement one radix-2 "butterfly" operation.) An equally fast radix r/2 processor would require four parallel arithemetic units due to the N/r $\log_r N$ execution speed of the Cooley-Tukey algorithm. In general, moving from radix-r to radix s = q x r, the speed increase is

$$\frac{N}{r}\log_s N \cdot X = \frac{N}{s}\cdot\log_s N$$

$$=> \qquad X = q\cdot\log_s q$$

It can be seen that the number of adders/subtracters stays constant for equally fast machines of different radices.

Secondly, we see a difference in the number of complex multipliers necessary to implement different radix proocessors. Again, referring to Eq. 6, we first consider the number of multipliers necessary within the radix-r transform computer. The recursive nature of the FFT allows us to see that implementation of a radix-r ($r=2^q$) transform using radix-2 "butterflys" requires q steps. At the i^{th} step, $2^{(i-1)}$ transforms of radix-$2^{(q-i+1)}$ are performed on the outputs from the $i-1^{st}$ step. At each step i, the weights used in the transform are given by

$$W^x = e^{(-j2\pi x/i)} \qquad \text{where } x = 0,1,2, \ldots ,i-1.$$

This would indicate i different weights for an i-point transform. However, recalling that $W^{(i-y)} = -W^y$, this is reduced to i/2. Further, we notice that $W^{(i/4)} = j$ and $W^0 = 1$. Multiplication by j simply requires interchanging the real and imaginary parts of the complex number. Thus for radices ≥ 4, there is another savings of two complex multipliers. For radix-2, there is only one.

The values input to the radix-r transform must be weighted appropriately before the computation. Eq. 6 shows that the weights are dependent on the position of the input value in the input vector. However, there is always one weight which is unity. Therefore, another (r-1) multipliers are necessary to complete the entire transform step calculation.

In summary, the number of multipliers necessary to implement a radix-r ($r=2^q$) transform arithmetic unit is given by

$$M = [\sum_{i=1}^{q-1} 2^{i-1}(\frac{r}{2^{i-1}\cdot 2} - 2)] + (r-1) + 1 \tag{18}$$

$$= [\sum_{i=1}^{q-1} r/2 - 2^{i}] + (r-1)$$

$$= (q-1)\cdot r/2 - (2^q - 2) + (r-1)$$

$$= (q-1)\cdot r/2 + 1$$

$$= \frac{r}{2}\log_2 r - 1$$

The expression in [] is the number of multipliers within the arithmetic unit. The (r-1) is the number of multipliers used for initial weighting. Table I shows some examples of the number of complex multipliers necessary for radix-r arithmetic units (M) compared with an equally fast version of a radix-2 machine (N).

TABLE I

r	q	M	N
2	1	1	1
4	2	3	4
8	3	9	12
16	4	25	32
32	5	65	90

A third consideration for the radix-r processor is the amount of coefficient storage necessary. Eq. 6 shows that the "twiddle factor" at step i uses the (i-1) most significant digits (base r) of the data address, digit reversed and masked as the base value to which W must be raised. This value is then multiplied by $k_{(m-i-1)}$ which can take on the values 0,1,2, ... ,(r-1). When i=m, m-1 digits are used to compute the base value. During this iteration N/r different values will be used indicating that the minimum number of coefficients that must be stored for a radix-r transform is N/r. (It is assumed that all coefficients will be stored at the address equal to the value to which W must be raised to obtain that coefficient. For example, W^8 would be stored at location 8.)

The $k_{(m-i-1)}$ modifier of the base value has considerable hardware implication. Arithmetically, the final "twiddle factor" value may be determined from the base "twiddle factor" by squaring, cubing, etc. (The highest power being r-1.) This requires $\log_2 r$ stages of complex multipliers for a total of r-2. Alternatively, all of the weighting factors may be stored directly raising the quantity of storage necessary to (r-1)/r x N coefficients. In general, the lower price of ROM storage would argue in favor of use of the extra storage. However, parallel access of all r-1 coefficients is not possible since the $k_{(m-i-1)}$ parameter takes on all integral values less than r and the base values vary with iteration. Thus, multiple read cycles would be necessary to find all coefficients; or, multiple copies of the coefficients may be stored to allow parallel access. Even storage of multiple copies of the coefficients would be cheaper than the multiplier implementation (by today's standards).

3.2. Perfect Shuffle Data Handling

Any radix-r transformer must process r complex data points per "butterfly". When a pipelined architecture is employed (as will be discussed in the next section), it is desirable to access all r in parallel as this provides the shortest pipe cycle time. It is in the memory organization that the full impact of the perfect shuffle data routing algorithm is found. While the Cooley-Tukey algorithm changes the separation of the data points to be accessed at each iteration of the transform, the perfect shuffle maintains a constant separation between points. The Cooley-Tukey (decimation in time) algorithm begins with a separation of data points of $2^{(m-1)}$ (Eq. 6) for input during the first iteration and ends with a separation of 1 at the m^{th} iteration. It is exactly the opposite for decimation in frequency. Output at each iteration is sequential for both algorithms. The perfect shuffle algorithm always accesses the data sequentially and at a separation of r on output.

No memory architecture is possible that allows direct parallel access of all input data values during all iterations of the Cooley-Tukey algorithm. Some form of variable length buffer must be interposed between the memory and arithmetic unit to match the correct values. However, using RAM and an interleaved architecture guarantees that all data values may be accessed in parallel when the perfect shuffle is employed.

Close examination of the expression for the perfect shuffle reveals that a physical memory cannot function as both input and output simultaneously, as some input values will be destroyed by output before they are read out. However, a memory may function in both capacities at different times. When used as an input memory, access is sequential, r values per cycle. This would require r-way interleaving with addresses $0,r,2r,\ldots$ in the first memory leaf; $1,r+1,2r+1,\ldots$ in the second and so on. When functioning as an output memory, r complex values must also be accessed in parallel. This time however, at intervals of r. This demands that addresses separated by r be stored in different memory leaves. Notice this is exactly contrary to the sequential interleaving needed for input. The result is a requirement for r^2-way interleaving in general in order to accomplish both sequential access and a radix-r shuffle. Thus, addresses $0,r^2,2r^2,\ldots$ occupy a leaf; $1,r^2+1,2r^2+1,\ldots$ occupy the next and so on. Which leaves are written into during a given memory cycle is a function of whether the memory is reading or writing (input or output mode) as well as what the address is.

If multiple memory accesses can be accomplished during a machine cycle, then the interleaving may be reduced to r^2/C where C is the number of memory cycles that may be executed during one processor execution cycle.

Shift register type memories may also be employed in a similar flexible architecture [22]; however, I/O considerations argue in favor of RAM (see Section 5.6.7).

3.3. Pipeline Architecture For The UFTPP (Horizontal Parallel Processing)

Pipelining is a familiar architectural technique for achieving a form of parallel processing in computers. The basic idea is to break a complex operation into a series of simpler steps. Between steps the intermediate results are stored in registers. The operation is begun on a new set of input data at the beginning of each clock period as the results from the first step proceed to the second and so on. After a fixed delay, the entire pipeline will be filled and results will begin to emerge from the end of the pipe, one per clock period. Since the simple operations require less time to complete than the entire complex operation, the clock frequency may be increased. Parallel processing is occurring since there are several data sets at various stages of completion during each execution cycle.

If the original operation would have taken time T to complete and it has now been divided into a pipeline of S steps each of duration t, then the following expressions hold (assume M data sets are to be processed)

$$\begin{aligned} X &= \text{total time to process without pipelining} = M \cdot T \\ Y &= \text{latency of the pipeline} = S \cdot t \\ Z &= \text{processing time with pipelining} = M \cdot t \\ V &= \text{total computation time with pipelining} = (S+M) \cdot t \end{aligned} \tag{19}$$

If $S \times t \approx T$ (i.e. each step in the pipeline requires almost the whole pipe cycle to complete it, in other words, the choice of operations to be performed in each pipe cycle is time efficient), then the speed-up factor is

$$\begin{aligned} \frac{X}{W} &= \frac{M \cdot T}{(S+M) \cdot t} \\ &\approx \frac{M \cdot T}{(M \cdot t)+T} \\ &= \frac{M \cdot n}{M + n} \\ &\approx n \qquad \text{for } M >> n. \end{aligned}$$

and $n = T/t$.

In the case of the FFT, the number of operations to be performed is

$$M = \frac{N}{r} \cdot \log_r N$$

where N is the size of the transform and r is the radix. This substituted into Eq. 19 gives

$$Z = \frac{N}{r} \cdot \log_r N \cdot t \tag{20}$$

for the total processing time if t is the processor cycle time.

The question then becomes "Can a radix-r Fourier Transform be implemented in a pipelined machine?" It is clear that the $\log_r N$

iterations of transform can be pipelined from the several processors that implement it in a cascade architecture [6]. In each iteration the following steps must be performed N/r times:

1) Read the next set of r data from the input memory; read the next set of r coefficients (sine-cosine values).

2) Perform complex multiplication of r input data pairs with the sine-cosine sets.

3) Perform the radix-r butterfly operation.

4) Write the set of r results into the output memory.

It has already been shown that determination of the addresses for the data in steps 1 & 4 is possible and straightforward. In section 3.1 it was shown that a radix-r butterfly may be recursively broken down into several radix-2 "butterfly" operations. A radix-2 "butterfly" is simply a complex multiplication followed by a complex addition and subtraction. So, steps 2 and 3 rely on performing a complex multiplication.

Complex multiplication may be expressed thus:

$$e + jf = (a + jb) \times (c + jd) \qquad (21)$$

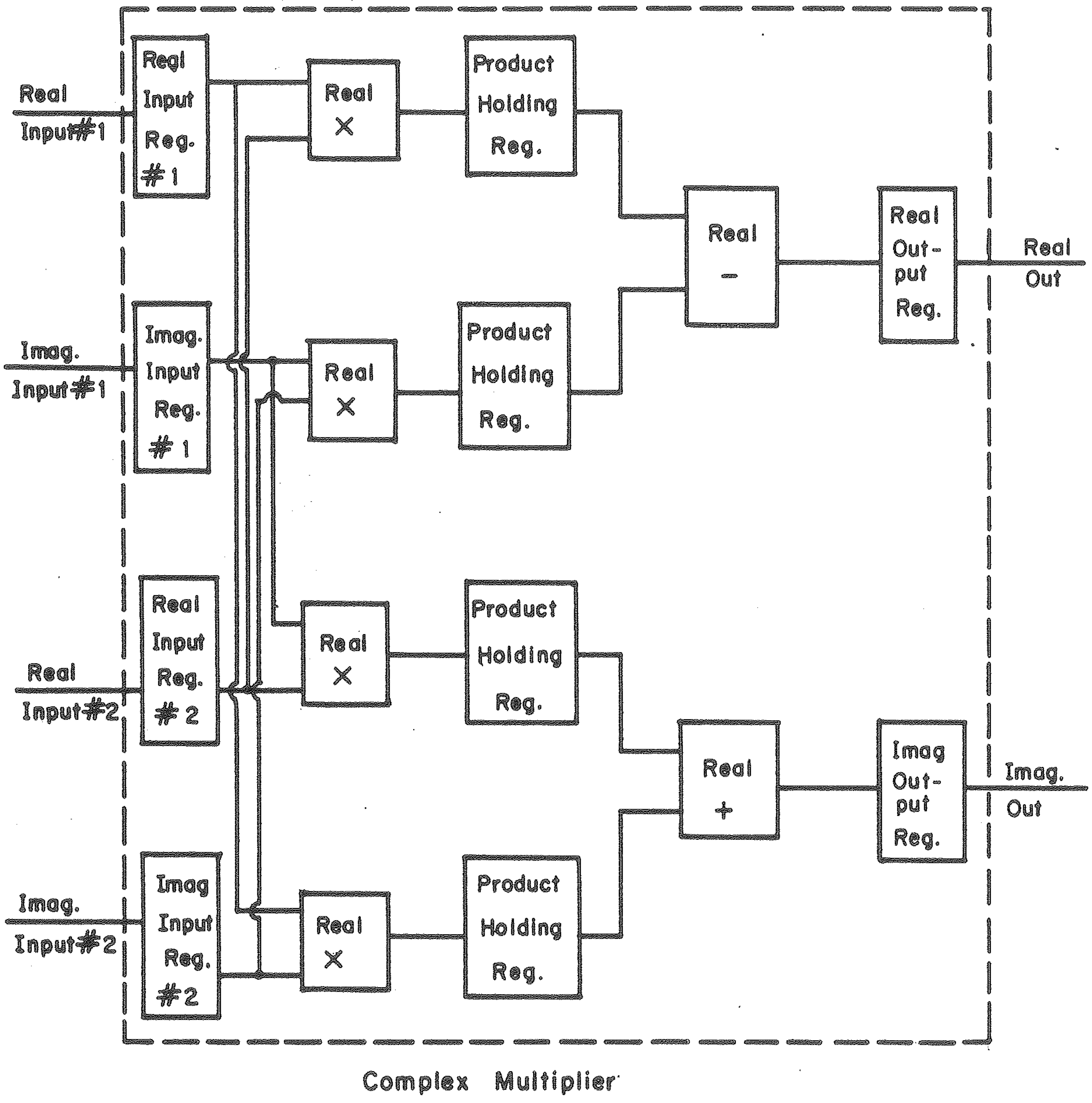
$$= (ac - bd) + j(bc + ad)$$

$$\Rightarrow e = ac - bd \quad \text{and} \quad f = bc + ad$$

It is easily seen that it requires 4 independent real multiplications and an addition and subtraction. A pipelined complex multiplier is shown in Figure 7.

3.4. Multiple Arithmetic Units (Vertical Parallel Processing)

Section 2.3 showed that the perfect shuffle data routing algorithm used by Pease could also be applied to a sequential processor to simplifiy the hardware and section 3.3 showed that the technique of pipelining could be applied to the simplified processor to achieve horizontal parallel processing. Vertical parallel processing (i.e. use of multiple arithemetic units working simultaneously) is also possible with this architecture.



XBL784-3035

Figure 7

Equation 6 in section 2.1 shows that the results of the i th iteration depends only upon the results from the i-1 st iteration. Further, it shows that pairs of data from that iteration are inputs to only one butterfly operation in the subsequent iteration. Thus, since the output address of any input pair may be determined from its input address (section 2.3), it is clear that there are no conflicts that would prevent several processors from working simultaneously on the same transform. (For example, two radix-2 arithmetic units working together would process 4 complex pairs at each pipeline step. Unit 1 would process pairs 0,1,4,5,8,9,... from the input and unit 2 would process 2,3,6,7,... Their corresponding output addresses would be 0,2,8,10,16,18,... and 4,6,12,14,..., respectively.)

From equation 19, it can be seen that processing time with P processors is

$$\frac{N \cdot \log_r N}{r \cdot P} \tag{22}$$

cycles. As noted in section 3.1 a factor of 4 is gained in processing speed for each factor of 2 increase in radix. Also, Table I showed that fewer multipliers are necessary for the higher radix processors as opposed to equally fast radix-2 units. However, there may be cases where multiple copies of a less complex, smaller radix unit may be more desirable from an implementation standpoint. One further consideration concerns the application of the processor. If flexibility in transform size is necessary, then the smaller the radix, the more sizes of transforms may be computed. (Recall that given radix r, only transforms of $N = r^m$ where m is an integer may be computed.)

It should be noted that certain values of P (Eq. 22) are more convenient than others. When P is an integral power of 2, then the only modification to the architecture is interleaving the memories further. A parallel machine will process $r \cdot P$ data sets per cycle and thus the memory must be interleaved $r^2 \cdot P$ ways (see section 3.2). It is possible to implement a processor that does not contain an integral power of two arithmetic units. However, in order to guarantee that all $r \cdot P$ data sets can be accessed in parallel at all addresses, the memories must be interleaved $r^2 \cdot P'$ ways where P' is the smallest integral power of two larger than P. In addition, the addressing hardware must be alterred to translate the logical addresses to physical ones.

Both increasing the number of parallel arithmetic units and increasing the radix cause memory fragmentation at the same quadratic rate due to the $r^2 \cdot P$ memory interleaving requirement.

The modularity of this design allows expansion of the machine architecture to meet a wide range of speed and budget requirements subject only to the constraints discussed above.

3.5. Comparison of the UFTPP architecture with other architectures

The basic classes of Fourier transform processor architectures are the sequential, cascade, parallel and array types [5]. The parallel and array architectures are, as yet, paper designs since their implementation is not economically feasible. The parallel architecture is depicted in Figure 5. The speed of the parallel architecture is achieved when N/r arithmetic modules are used in the UFTPP architecture. The array architecture is simply an extension of the parallel and may be implemented in several ways (e.g. Figure 4, Figure 3 or see [4]). The UFTPP cannot attain the speed of an array architecture. The sequential architecture contains only 1 arithmetic unit and has the speed of a 1 arithmetic unit UFTPP.

The most interesting architectures for comparison with the UFTPP architecture is the cascade type (described in detail by Groginsky & Works [37] and Bergland [5]). The cascade architecture computes a fixed length transform using $m=\log_r N$ arithmetic units arranged in series. Each of the m radix-r units computes the results for a given iteration (i.e. the results of butterfly operations between complex points seperated by $r^{-i}N$ where i is the position of the arithmetic unit in the series starting with 1) from the output of the previous unit. Each unit has associated with it a digital delay line (i.e. shift register) of length $(r-1)r^{-i}N$ which pairs the correct data points for the butterfly computation for that iteration. The latency of this architecture is N cycles (i.e. the time from entrance of the first data sample to the emergence of the first frequency coefficient) and the amount of intermediate storage necessary is N complex words. The results are output in digit reversed order and requires another N cycles.

The UFTPP architecture with $m=\log_r N$ arithmetic units has a computation time of $N/r + \log_r N\ S$ cycles (from Eq. 19). This is slightly different than the latency as the entire transform is complete at that time. However without modification to the architecture this is when the output is ready to be unloaded. Unloading also requires N cycles. The UFTPP architecture requires 2N complex words of storage.

If we set the expressions for the latency of the two architectures equal to one another we see that

$$\frac{N}{r} + \log_r N\ S = N(r-1) \sum_{i=1}^{m} r^{-i}$$

or,

$$S = \frac{N}{\log_r N}$$

When S=15 (i.e. there are 15 pipeline segments in the UFTPP), compute times (assuming identical clock speeds) for both architectures will be equal when N=256, r=2; N=64, r=4; N=8, r=8. For N larger at the given radix , the UFTPP will be faster.

The cascade architecture has the advantage that loading of new input, unloading results and computation all take place simultaneously. This can only be accomplished with the UFTPP when another 2N complex words of storage are used as input and output buffers. These buffers are filled while transform computation is taking place on data from two other memories.

The cascade architecture relies on the assumption that computation, loading and unloading speeds are approximately equal in order to obtain its hardware efficiency. However, when higher clock speeds are possible within the processor than are possible for I/O, then the cascade loses its advantage. If T is the clock period for I/O data transfers and t the processor clock period, then for T/t values in the range of 2-5 the UFTPP architecture shows considerable savings in hardware. Table II shows the number of arithmetic modules necessary to perform a transform (P) in the same time as the m modules needed for a cascade implementation for various radices (R), transform sizes (N) and speed-up ratios T/t. It is clear that for values of T/t > 5 and radices > 4 the UFTPP advantage will increase.

TABLE II

			T/t= 2	3	4	5
N	r	m	P	P	P	P
64	2	6	6	2	2	1
128	2	7	3	2	2	1
256	2	8	3	2	2	1
512	2	9	3	2	2	1
1024	2	10	3	2	2	2
2048	2	11	3	2	2	2
4096	2	12	4	3	2	2
8192	2	13	4	3	2	2
16384	2	14	4	3	2	2
64	4	3	1	1	1	1
256	4	4	1	1	1	1
1024	4	5	1	1	1	1
4096	4	6	1	1	1	1
16384	4	7	1	1	1	1

It has been shown that the cascade architecture is more hardware efficient when no buffer memories are used and the I/O clock speed and processor clock speeds are the same. However, if the I/O clock speed is slower than possible for the processor (which is generally the case when the Fouier transform processor is a peripheral device to a central processor), then considerable hardware savings may be realized in the arithmetic units when arranged in the UFTPP architecture rather than the cascade. The UFTPP architecture requires 4N complex memory words as opposed to N for the cascade; however, this is offset by the savings in the arithmetic units as shown by Table II.

CHAPTER 4

NUMBER REPRESENTATION & ARITHMETIC

4.1. Fixed Point vs. Floating Point Representation

The establishment of machine word size, number representation and arithmetic required an analysis of the errors to be expected with each of the various choices. This analysis was carried out via software simulation of the entire Fourier transform computation on a CDC 7600 computer.

The advantages and disadvantages of fixed point arithmetic vs. floating point arithmetic have been extensively discussed in the computer science literature [e.g. 31,65]. The facts which bear directly on the implementation of a Fourier Transform processor are:

1) Fixed point arithmetic offers a constant absolute error dependent on the number of bits used. Floating point exhibits a constant relative error; that is, the error associated with a number is a fixed fraction of its value with the value of the fraction dependent on the number of bits used.

2) Floating point offers a larger dynamic range (i.e. the range of numbers that can be represented with a given number of bits) than does fixed point.

3) Floating point arithmetic requires more hardware than does fixed point.

A fixed point number is simply a group of n bits whose values are agreed upon by convention. Regardless of these conventions there are only 2^n different combinations possible. If the least significant bit represents the value X, then the precision of that number is $\pm X/2$. A floating point number consists of two parts, the mantissa and the exponent. The mantissa is a fixed point number and the exponent is an exponential modifier whose base is agreed upon by convention. For example, scientific notation is a form of floating point notation. The number 3.1415×10^8 has a mantissa equal to 3.1415 and an exponent of 8. The base of the exponent is 10. The precision of the floating point number is a function of both parts. In the above example, the precision of the mantissa is $\pm$ .00005, but the error is multiplied by 10^8 to give a final error of $\pm$ 5000. It can be seen that floating point representations exhibit a constant relative error.

It is also easy to see from the above example that floating point offers a much larger dynamic range than fixed point. The range of floating point is determined by the base used and the number of bits in the exponent. The number of finite values that can be represented (using binary) is $2^m \cdot 2^n = 2^{m+n}$ where m is the number of bits in the mantissa and n the number of bits in the exponent. Thus, it is seen that for a constant number of bits the same number of numbers may be represented in either a floating or fixed-point system but their precision and spacing are different.

When performing floating point arithmetic there are two operations which must be accomplished that are not necessary in fixed point. The first, binary point alignment, concerns addition and subtraction only. Before two numbers of differing exponents can be added or subtracted their exponents must be made equal and mantissas adjusted accordingly. The second operation is normalization of the result following addition, subtraction or multiplication. In floating point representation, the minimum error is achieved when the most significant bit has value (i.e. is not a place holder). After the above operations, this may not be the case. The operation of shifting the result until its most significant bit occupies the most significant position and adjusting the exponent is called normalization. Both of these operations are easily implemented; however, they do represent an increase in necessary hardware.

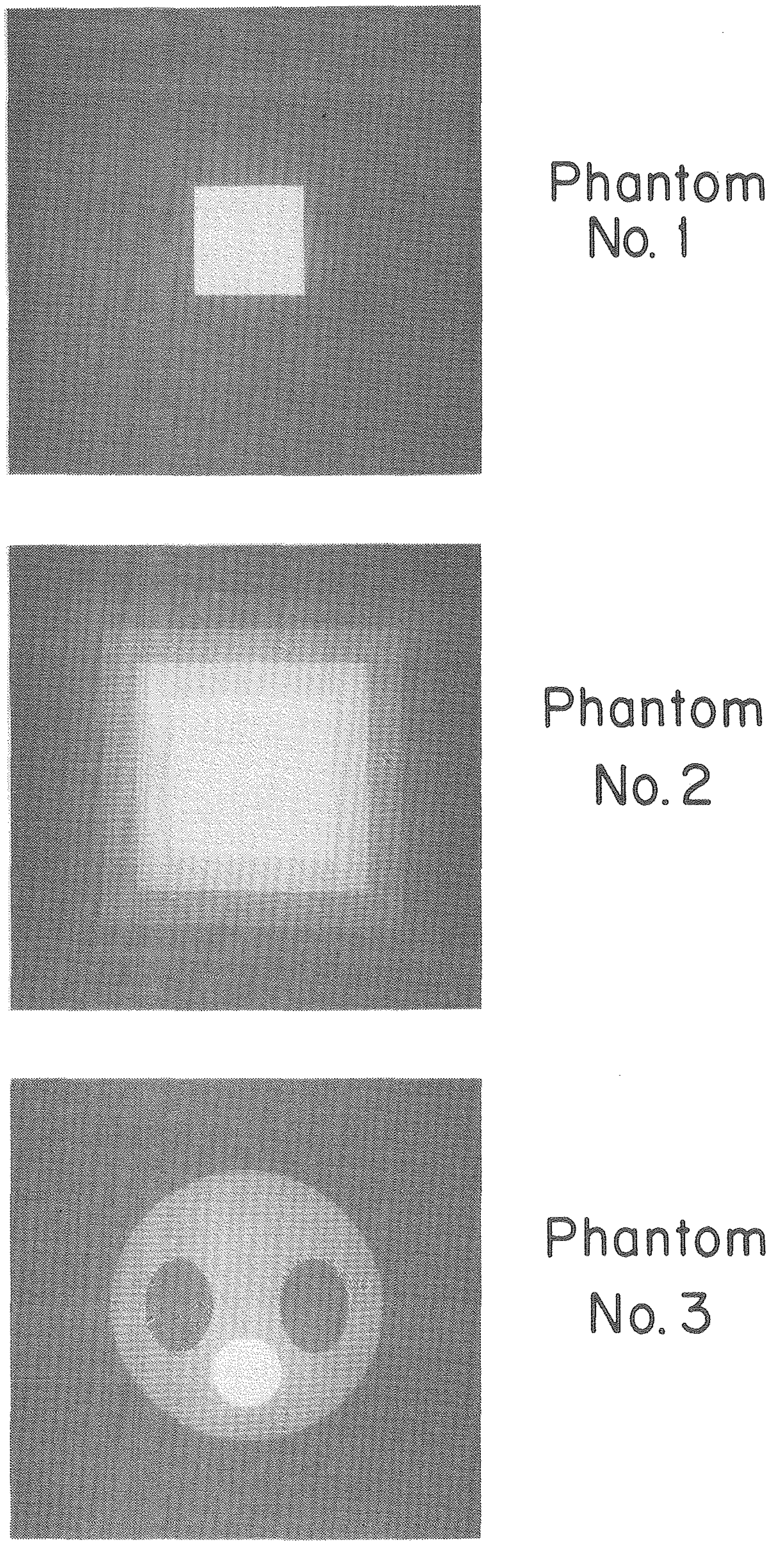
Error propogation in computing Fourier Transforms has been given considerable attention in the literature for both the fixed-point [15,54,67,69,72] and floating-point cases [15,41,44]. Because of the complexity of the Fourier transform, only upper and lower bound analyses have been performed. The most useful result for the purpose of this thesis is that of Welch [72] which shows for an array of N values

$$\frac{\text{rms (error)}}{\text{rms(computed result)}} \propto \frac{\sqrt{N}}{\text{rms(orig. array)}}$$

This result for the one-dimensional case will be shown to extend to two-dimensional transforms performed as a series of one-dimensional transforms which are of interest here.

In order to make decisions as to word size, fixed or floating point representation and, in the case of fixed-point, the number of precision bits to be used, computer simulations were performed using several different phantoms (Figure 8). Computer subroutines were written which performed both fixed and floating point arithmetic using different word sizes and different numbers of precision bits. (Precision bits are the bits to the right of the binary point.) For each simulation the input array (phantom) was forward transformed and inverse transformed several times. Each forward-inverse computation is referred to as an iteration. At several points, absolute errors were computed by subtracting the original image from the current result. The absolute values of the errors were histogrammed. The metric used was the 97th percentile of the absolute value of the errors. Figure 9 shows the behavior of fixed-point arithmetic when the number of precision bits is varied. Data were taken after 1, 10 and 20 iterations. It can be seen that the error varies as the log of the number of bits used and propogates linearly with the number of iterations. Several different word lengths were used. The bits not used for precision allowed different ranges of values in the input arrays. (For example, if a 24 bit word was being used with 16 precision bits, 8 bits are available for range.) Assuming twos complement representation, input values could vary from -2^7 to $+2^7$. Regardless of word length, input values or range, all results were the same within statistical variation supporting the premise that fixed-point representation exhibits constant absolute error behavior.

For the case of floating-point representation, it was observed that for a given input array the relative behavior was the same as fixed-point although the values of the errors were different. Figure 10 shows the behavior of phantom #1 whose range was 10. Table III shows the error for various phantoms of various ranges after 10 iterations using 24-bit floating point. This is an excellent illus-



XBB804-4842

Figure 8

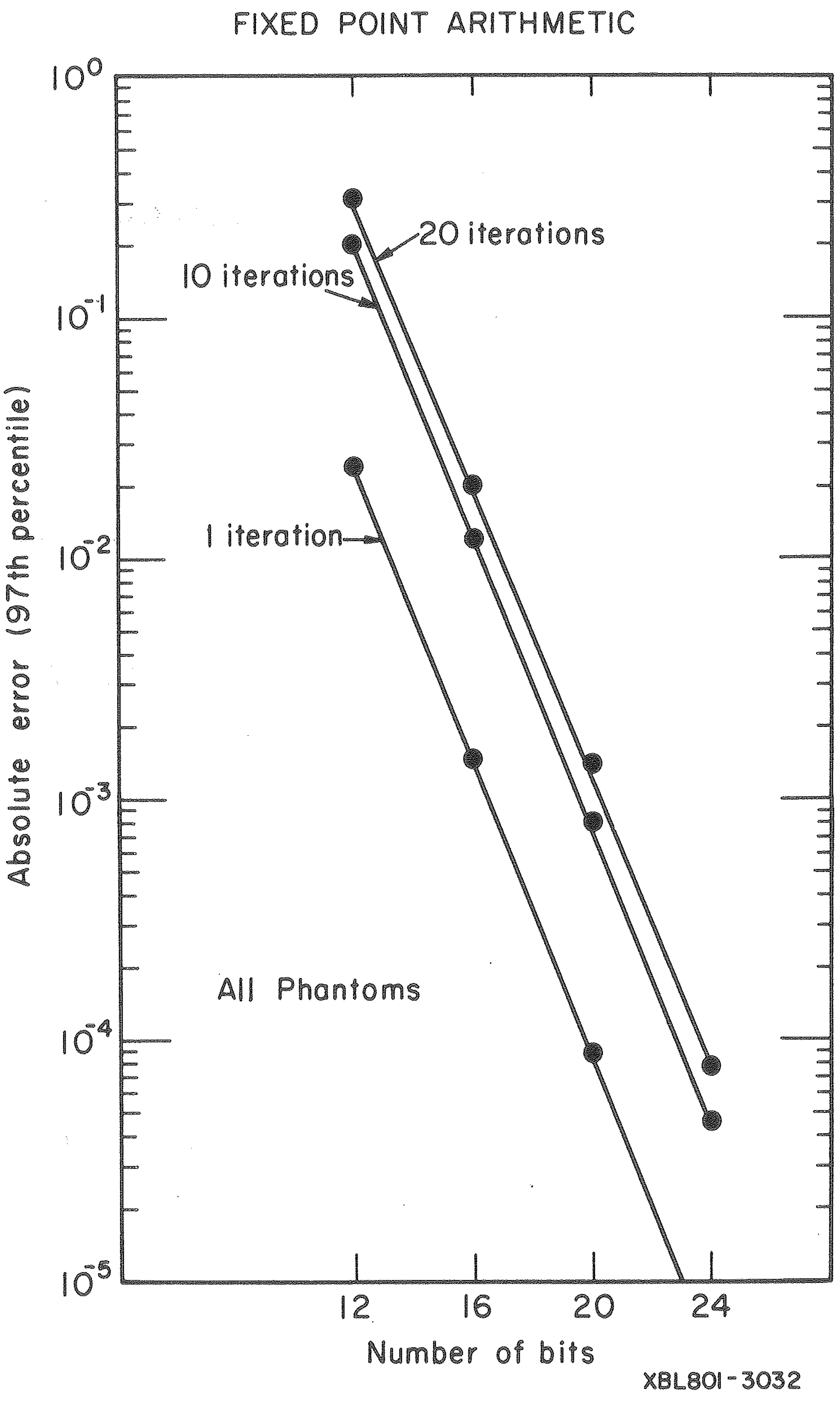

FIXED POINT ARITHMETIC
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
Absolute error (97th percentile)
20 iterations
10 iterations
1 iteration
All Phantoms
12
16
20
24
Number of bits
XBL801-3032

Figure 9

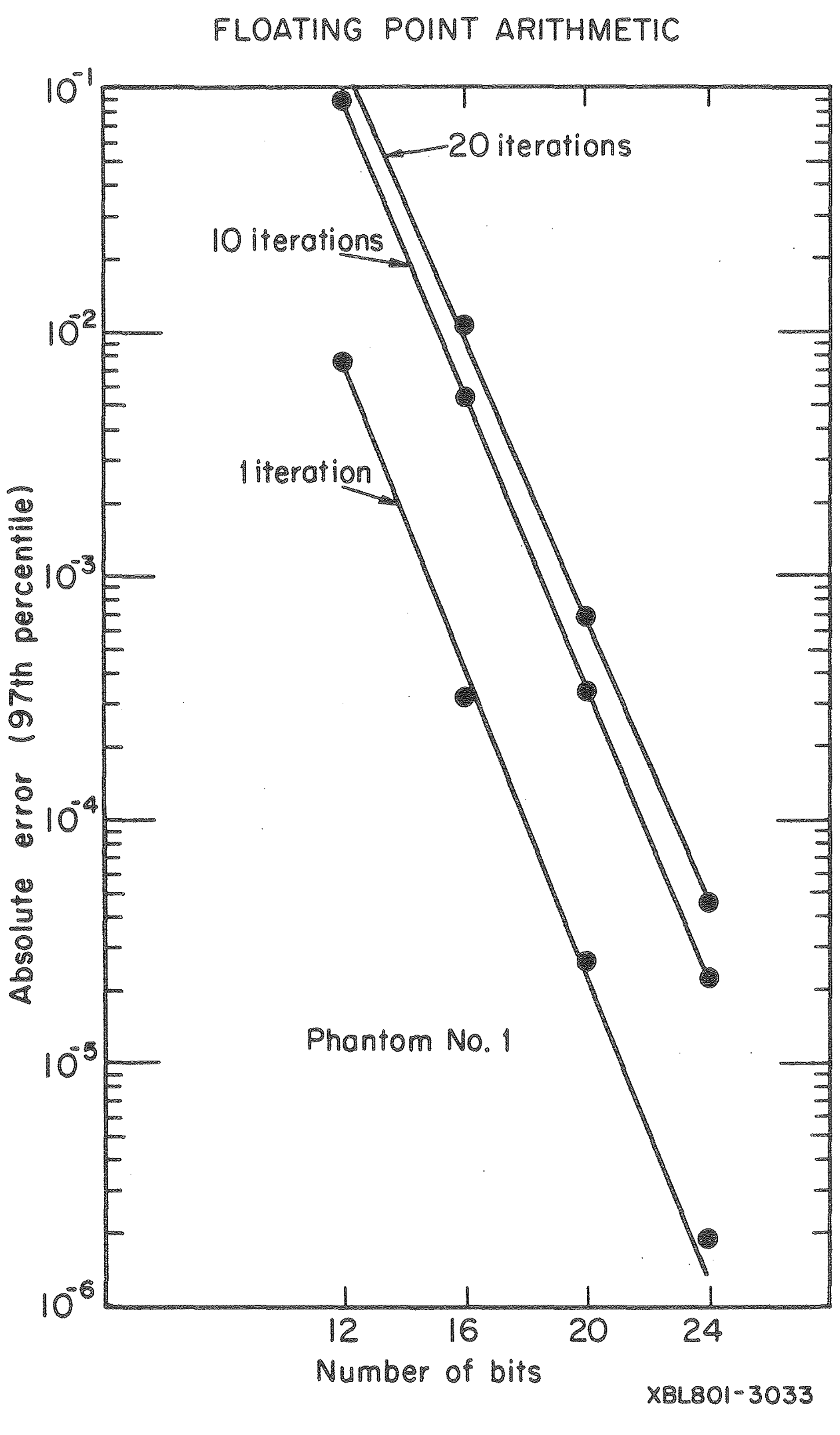

FLOATING POINT ARITHMETIC
20 iterations
10 iterations
1 iteration
Phantom No. 1
Absolute error (97th percentile)
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
12
16
20
24
Number of bits
XBL801-3033

Figure 10

tration of the data dependent behavior of errors under floating-point representation.

TABLE III

Phantom	Error
Phantom #2 (range 0 to 50)	1.7×10^{-4}
Phantom #2 (range 0 to 2000)	8.5×10^{-3}
Phantom #3 (range 0 to 250)	1.1×10^{-4}
Phantom #3 (range 0 to 1000)	1.2×10^{-3}

Considering the image processing applications of the UFTPP, it was decided that it was desirable to keep errors in the 1% range. The simulations indicated that this would require approximately 16 precision bits in the fixed point case and 20-24 in floating point. The fixed point would require additional bits to provide adequate range as well. Even though floating-point representation might allow a slightly shorter word length, the simplicity of fixed-point arithmetic hardware argued in favor of its use in the first version of this machine. In subsequent versions, however, it might be advantageous to use floating point.

The last consideration with respect to error propogation concerns handling of normalization in the fixed point case. The choice is to shift right one bit after each iteration or to divide by N after the entire transform. If one shifts after each iteration one prevents the possibility of overflow but sacrifices some precision. Waiting to divide by N after the entire transform requires $m=\log_2 N$ more bits to be added to all words which represents a very significant addition in hardware. Simulations were performed to observe just how much additional error would be introduced by dividing by 2 after each iteration. Table IV shows the differences in error under the same conditions as in Figure 9. It can be seen from this table that the difference in error is only about 10%. The amount of

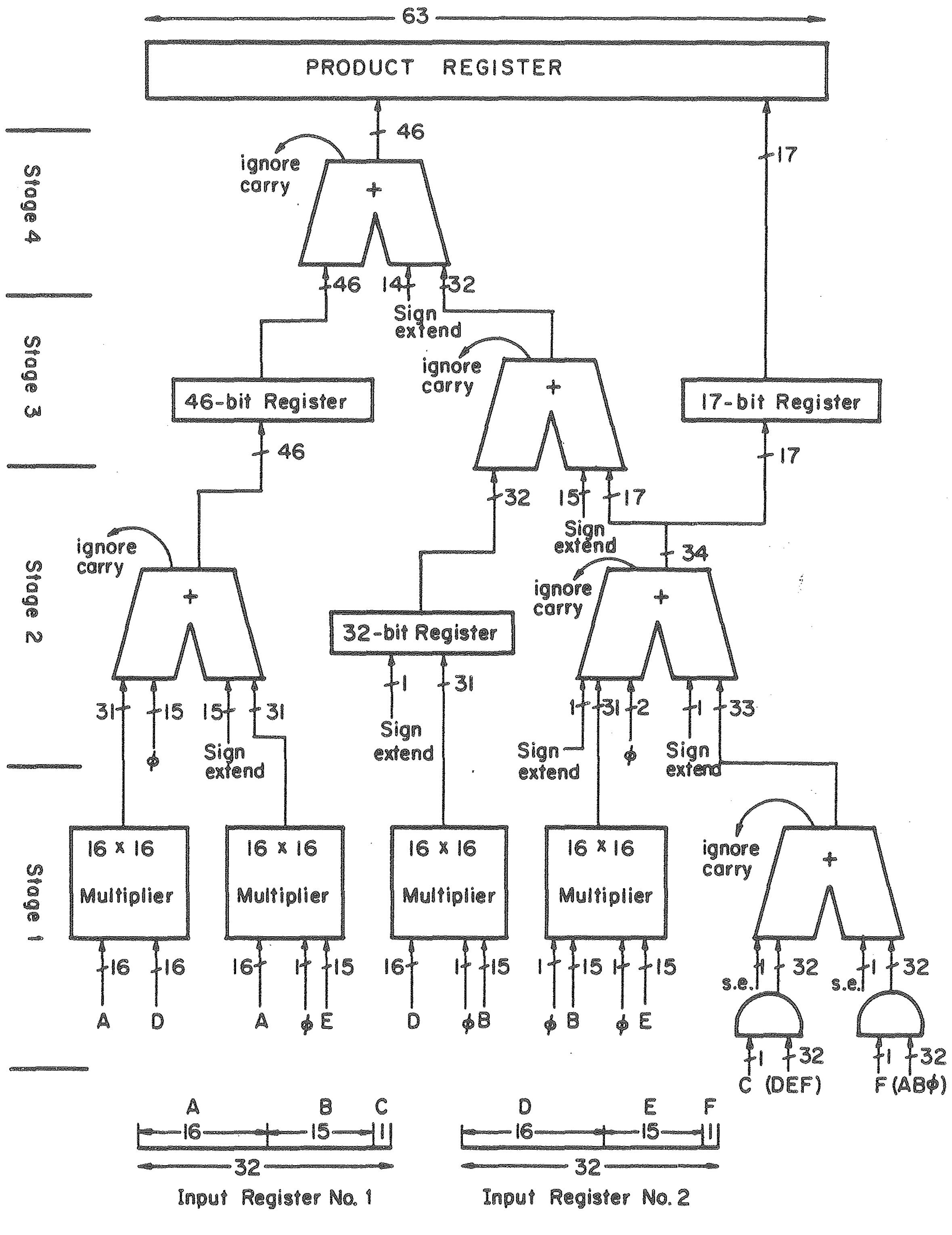
TABLE IV

	Iterations		
Bits	1	10	20
12	3.0×10^{-3}	6.0×10^{-2}	1.3×10^{-1}
16	3.0×10^{-4}	4.0×10^{-3}	4.0×10^{-1}
20	1.5×10^{-5}	3.5×10^{-4}	3.0×10^{-4}

hardware saved by performing the normalization at each iteration is worth the tradeoff of a 10% larger error.

In summary, it has been shown that for the purposes of the UFTPP and its applications fixed-point arithmetic using 16 precision bits should result in about 1% accuracy even when several iterations of transforms are performed. Since the two-dimensional transforms are to be accomplished with a series of one-dimensional transforms, it seems reasonable that the error growth should continue to follow the square root of N dependence demonstrated by Welch.

The overall word length to be used is largely determined by the cost that can be tolerated for the implementation of the real multipliers. As mentioned in Chapter 1, LSI multiplier chips have become available that perform 8-, 12-, or 16-bit multiplications. It is relatively straightforward to implement a double precision multiplication using four of these chips. The choice is then whether 24- or 32-bit words are more desirable. When 16 bits are used for precision, the available input ranges would be $\pm$ 128 or $\pm$ 32,768 for the 24- or 32-bit cases, respectively. It is clear that 24 bits simply does not allow a reasonable range and, thus, 32-bit words were chosen. Figure 11 shows the hardware configuration for performing a 32 x 32-bit multiplication yielding a 63-bit product.



XBL792-3212

Figure 11

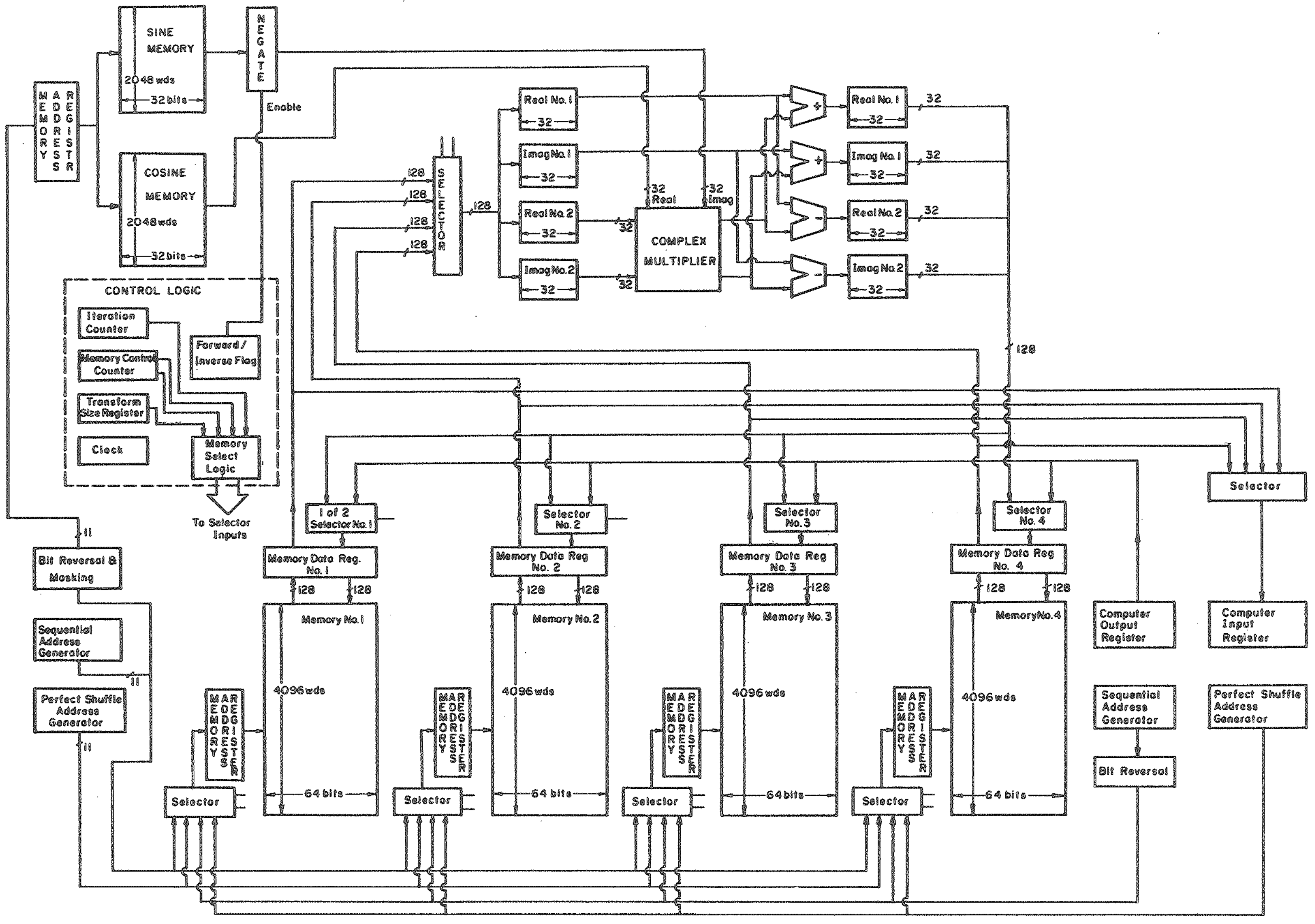
CHAPTER 5

PROCESSOR IMPLEMENTATION

5.1. Overall System Description

Demonstration and implementation of the architectural design put forth in this thesis was achieved by construction of a radix-2, one arithmetic unit prototype system for transformation of arrays up to 4096 complex values. As outlined in previous chapters larger and faster systems are easily designed. The system consists of an arithmetic unit, a memory system, a multiplexer system, an interface (for the host computer) and overall control logic. Figure 12 shows a schematic diagram of these constituents and their interconnections. The arithmetic unit houses the complex multiplier, cosine and sine read-only-memories, the butterfly computation module, and normalization circuitry. The memory system consists of four independent physical memories that may serve as processor input memory, processor output memory, computer input buffer or computer output buffer. The multiplexer system consists of all the selectors shown in Figure 12. It performs a routing function that brings the data from/to the appropriate memories to/from their destinations/sources. The interface allows the loading of data from the computer into the computer output buffer (in the UFTPP), unloading results from the computer input buffer to the computer and control of the UFTPP by the host computer. The control logic section initializes the processor, monitors progress of transform computation, buffer loading/unloading and communicates information concerning the status of the processor to the computer via the interface.

The processor is implemented on a total of 8 boards each measuring 24" x 18.5". Each board has 400 edge-connectors along one of the 24" edges. Of these, 80 are reserved for power and ground connections leaving 320 available for general use. A power and ground plane was laid down (one on each side of each board) using printed circuit technology to allow for low resistance power and ground distribution and to minimize ground loops. All other connections were



XBL7911-3916

Figure 12

wirewrapped. The eight circuit boards communicate with each other via a wirewrapped backplane.

5.2. The Arithmetic Unit

The arithmetic unit is implemented on two boards. Together they house a complex multiplier, the butterfly module, the sine/cosine memories, and the normalization circuitry. Each arithmetic board houses 8 LSI multiplier chips, 16 PROM chips and approximately 325 other MSI integrated circuits (e.g. 4-bit adders, 6-bit registers, etc.). Each board draws about 20 amps. Figure 13 shows the component side of one of the arithmetic boards and Figure 14 shows the wiring side of the same board.

In order to minimize the number of connectors necessary on each board, a technique similar to bit-slicing but applied to words has been employed. The output of the entire arithmetic unit is two complex values. Rather than having one board compute the first complex value and the other board the second one, a more efficient usuage of the connectors is made if one board compures the real parts of both complex values and the second board computes the imaginary parts. Thus, each board houses two real multipliers, a sine or cosine memory and half of a butterfly module. If we further divide the two complex inputs into their real and imaginary halves ($C_1 = (R_1,I_1)$ and $C_2 = (R_2,I_2)$), then the two real multipliers on the first board, A1, compute the values: R_1 cos and I_2 sin. The difference between these two values is the real part of the complex product, R_3, as indicated by Eq. 21 (Chapter 3). The multipliers on the second board, A2, compute the values R_2 sin and I_2 cos. The sum of these two is the imaginary part of the complex product, I_3.

The basic algorithm for implementation of the real multiplier is as shown in Figure 11 (Chapter 4). However, since the 63-bit product is to be combined with a 32-bit value in the butterfly module, it is not necessary to compute all 63 bits. Only 32-bit accuracy is necessary at the output of the complex multiplier and it can be shown that only 37-bits need to be computed at the outputs of the real multipliers. Figure 15 shows the hardware configuration necessary to

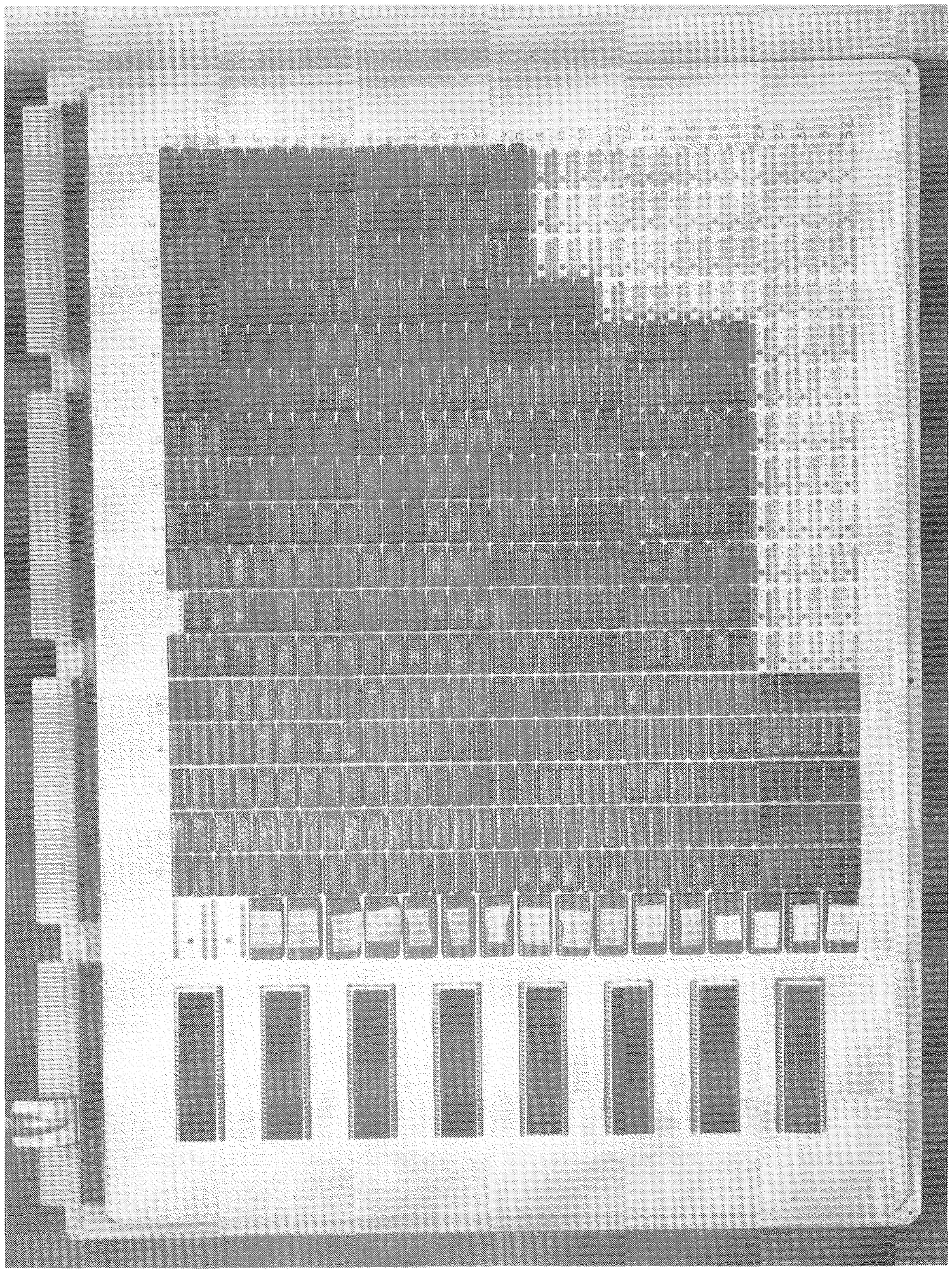


Figure 13 CBB790-16710

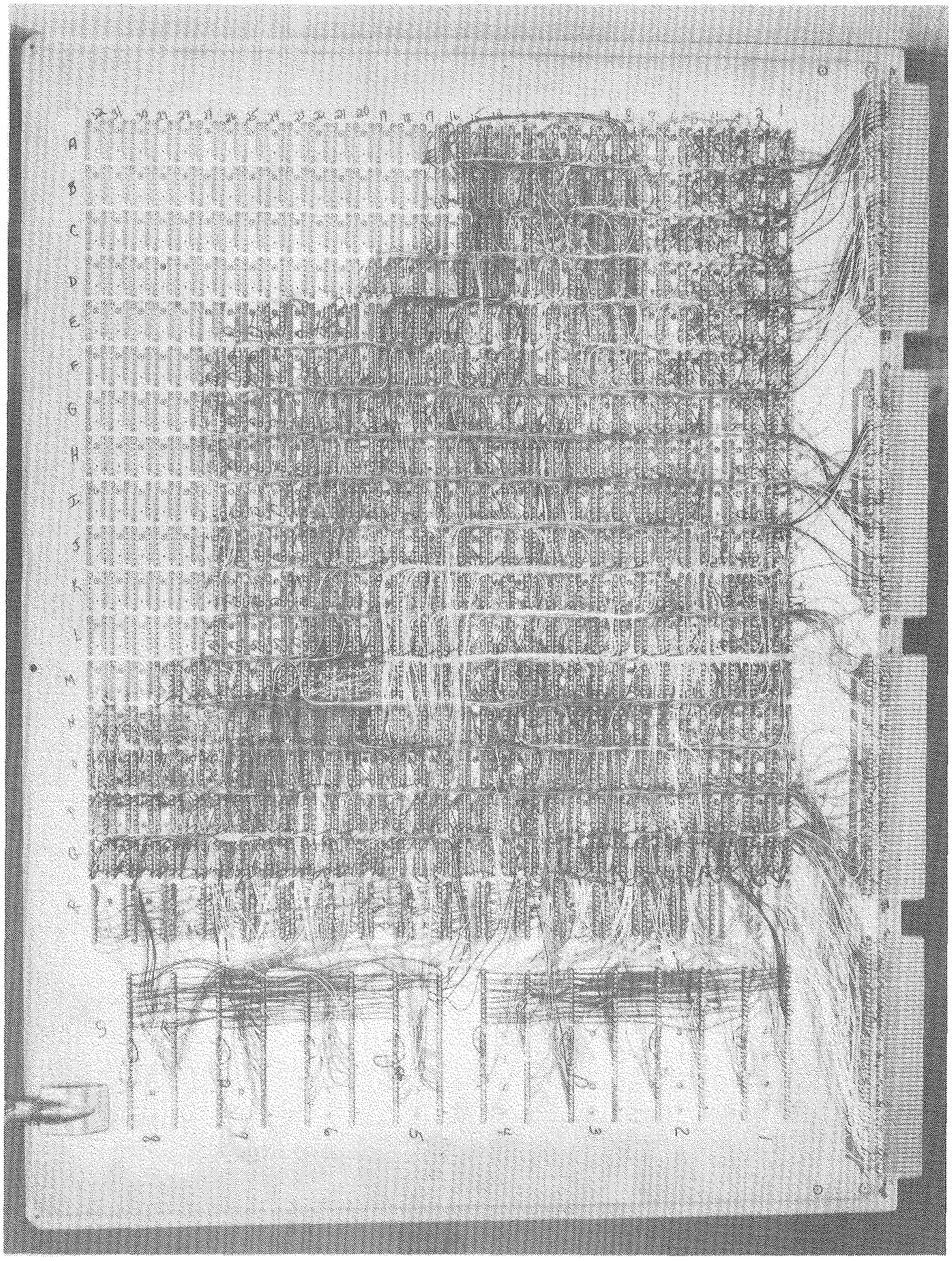


Figure 14 CBB790-16712

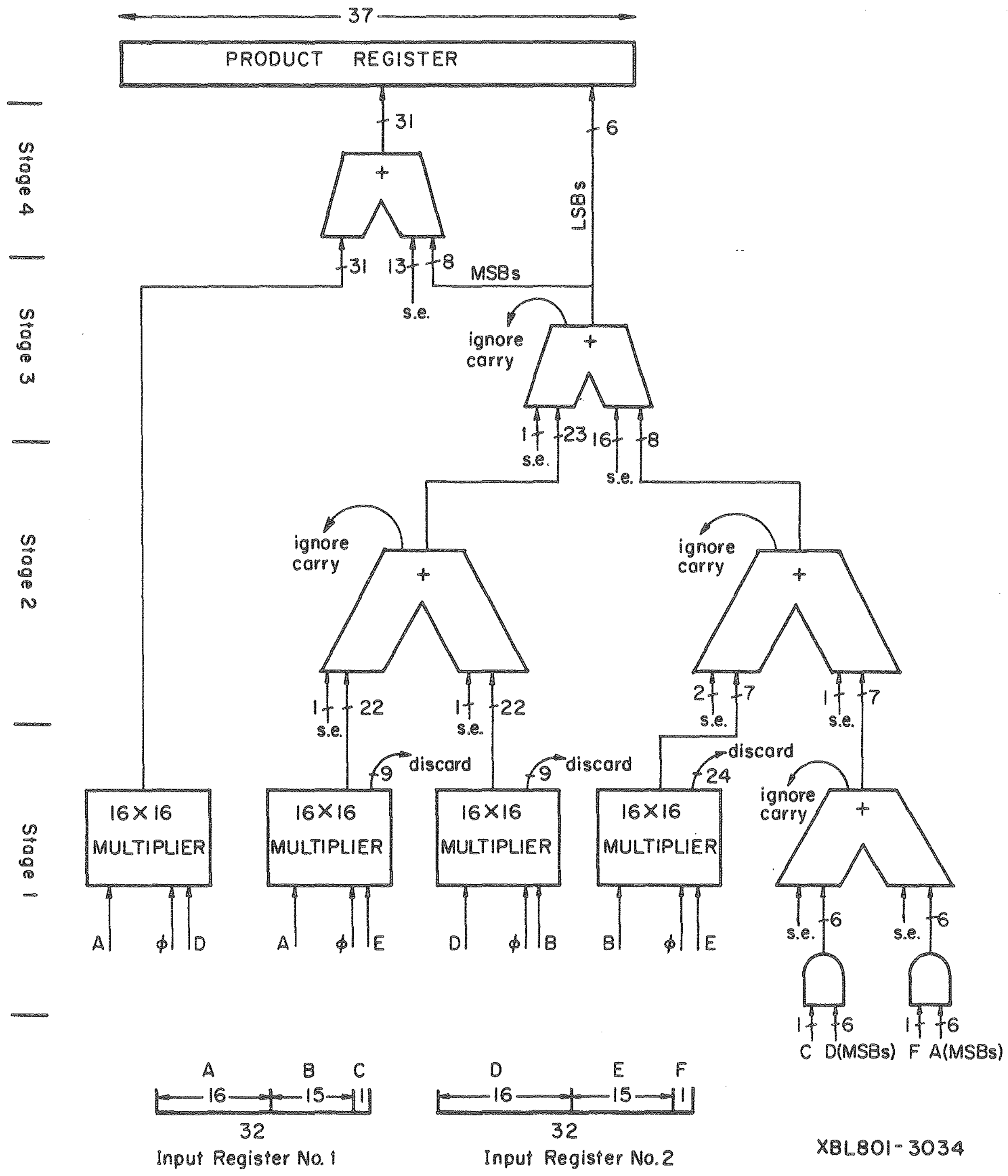

37
PRODUCT REGISTER
31
6
LSBs
Stage 4
31
13
8
MSBs
s.e.
Stage 3
ignore carry
1
23
16
8
s.e.
s.e.
Stage 2
ignore carry
ignore carry
1
22
1
22
2
7
1
7
s.e.
s.e.
s.e.
s.e.
9 discard
9 discard
discard
24
ignore carry
Stage 1
16×16 MULTIPLIER
16×16 MULTIPLIER
16×16 MULTIPLIER
16×16 MULTIPLIER
A
ϕ
D
A
ϕ
E
D
ϕ
B
B
ϕ
E
s.e.
6
s.e.
6
1
6
1
6
C D(MSBs)
F A(MSBs)
A B C
16 15 1
32
Input Register No. 1
D E F
16 15 1
32
Input Register No. 2
XBL801-3034

Figure 15

implement a real multiplier of this nature. Simulations have shown that results obtained from this implementation do not differ from the 63-bit result by more than 3×10^{-6}. Since the numbers used in the UFTPP have 16 precision bits, their accuracy is $\pm 2^{-17} = 7.6 \times 10^{-6}$. Thus, the results in the 63-bit product are indistinguishable from the results in the 37-bit version. This represents a 50% reduction in hardware for each of the four real multipliers.

The real multipliers are pipelined as described in Chapter 3. Figure 15 indicates the 4 stages each contains. Between each of the stages are registers which capture the results from the previous stage at each clock interval.

The inputs to the multipliers are either data values from a memory or a sine/cosine value. The sine and cosines are stored in ROM. The sine ROM is on one board and the cosine on the other. Each board makes its value available to the other one. The addresses for the ROMs are generated through a bit reversal and masking function described by Pease [48].

In all of the simulations cited in Chapter 4, it was assumed that the sine and cosine values are represented with N-2 bits of precision. The two remaining bits allow for a range of ± 2 (actually $-2 + 2^{-(N-2)}$ to $+2 - 2^{-(N-2)}$). The sine/cosine memories are 32 bits wide using 30 bits of precision.

The data inputs to the arithmetic unit are pairs of complex values as indicated by the signal flow graph (Figure 4) in Chapter 2. If we label them C_1 and C_2, then C_2 is the value to be "twiddled" (i.e. multiplied by the sine/cosine values) and C_1 is saved until the butterfly operation is ready to be performed.

The butterfly module consists of two adders and two subtractors. "Word-slicing" each results in half containing one adder and one subtractor (and produces either the real parts or the imaginary parts of the two complex pairs. The result from the half complex multiplier is added to and subtracted from the corresponding half of the other input value to yield the result. The first board, A1, computes $R_1 + R_3 = R_4$ and $R_1 - R_3 = R_5$ and the second board computes $I_1 + I_3 = I_4$

and $I_1 - I_3 = I_5$. Depending upon the direction of the transform, this result may be divided by two (i.e. shifted right one bit with sign extension) to incorporate the 1/N normalization necessary for the forward transform. This result is then the output of the arithmetic module.

The forward transform computation is

$$H(f) = \frac{1}{N} \sum_{i=1}^{n} h(t)\ e^{-j2\pi ft/N}$$

and the inverse is

$$h(t) = \sum_{i=0}^{n} H(f)\ e^{j2\pi ft/N}$$

The latency of the pipeline through the arithmetic unit is 10 clock periods including generation of sine/cosine table look-up address, reading the sine/cosine values, delay through the real multtipliers, performing the butterfly and normalization. Enumeration of all pipe cycles is given in detail in section 5.5.

Overflow detection is relatively simple when twos complement arithmetic is used. Overflow may occur in the final stage of the complex multiplier where the real products are combined or in any of the adders/subtractors in the butterfly unit. Special precautions have been taken in the multiplier to insure that no overflow will occur there.

Under twos complement arithmetic, subtraction of two arguments, a - b, is simply a + (-b) where -b is b complemented bitwise and then incremented. It is clear that overflow may occur in twos complement addition only where the signs of both inputs are alike. Similarly, for subtraction overflow may occur only when a and -b have the same sign. Addition overflow of two binary numbers results in a sum that is one bit longer than the inputs. Therefore, overflow detection may be accomplished by comparing the sign bit of the N-bit result with the sign bit of inputs. If overflow has occurred the sign bit will be complemented (the true result would be N + 1 bits long) and will no longer be the same as the sign bits of inputs. The boolean expression for this function is:

$$O = (a + b) \circ (a + s)$$

where

a is the sign bit of the first input
b is the sign bit of the second input
s is the sign bit of the N-bit sum

This may be realized in only 2 stages of logic if the XOR function is available.

As noted by Welch [72], in order to insure that overflow will not occur two conditions must be met. First, all of the input complex values must have a modulus $\leq M/2$ (where M is the maximum value which can be represented with the available bits) and a normalization of 1/2 must be performed after each iteration of the transform. In the case of the processor under consideration here, 16-bits are used for the range of input values. This allows values from $(-2^{15} + 1)$ to $(+2^{15} - 1)$. Thus, the maximum value of the modulus of any input should be $2^{14} - 1 = 16,383$.

5.3. The Memory System

The memory system consists of four physical memories any one of which may serve as processor input, processor output, computer input or computer output. The motivation for including the computer input/output buffers was to allow processing of the current transform while loading of the next and unloading of the last was taking place. Thus, the pipelining technique is extended one more level and further time savings are realized.

Each of the four memories are identical, with the exception of control logic for each which determines which of the four logical functions each memory will perform at any given time. Each memory contains a memory address register, MAR, (11-bits wide), a bidirectional memory data register, MDR, (128-bits wide) and a 4K word x 64-bit memory array. The data path is 128-bits wide because on each memory cycle two complex values must be read/written or two 64-bit words. (Each 64-bit word represents one complex value, the real part occupies 32-bits and the imaginary part occupies 32-bits).

Each memory consists of 64 memory chips, 64 4-bit registers for the MDR, 3 chips for the MAR, 44 hex 3-state buffer ICs and 10 chips for control. Total chip count for each board is 370 or 740 for the entire system. Each board draws about 15-17 amps for a total of 30-34 for the system.

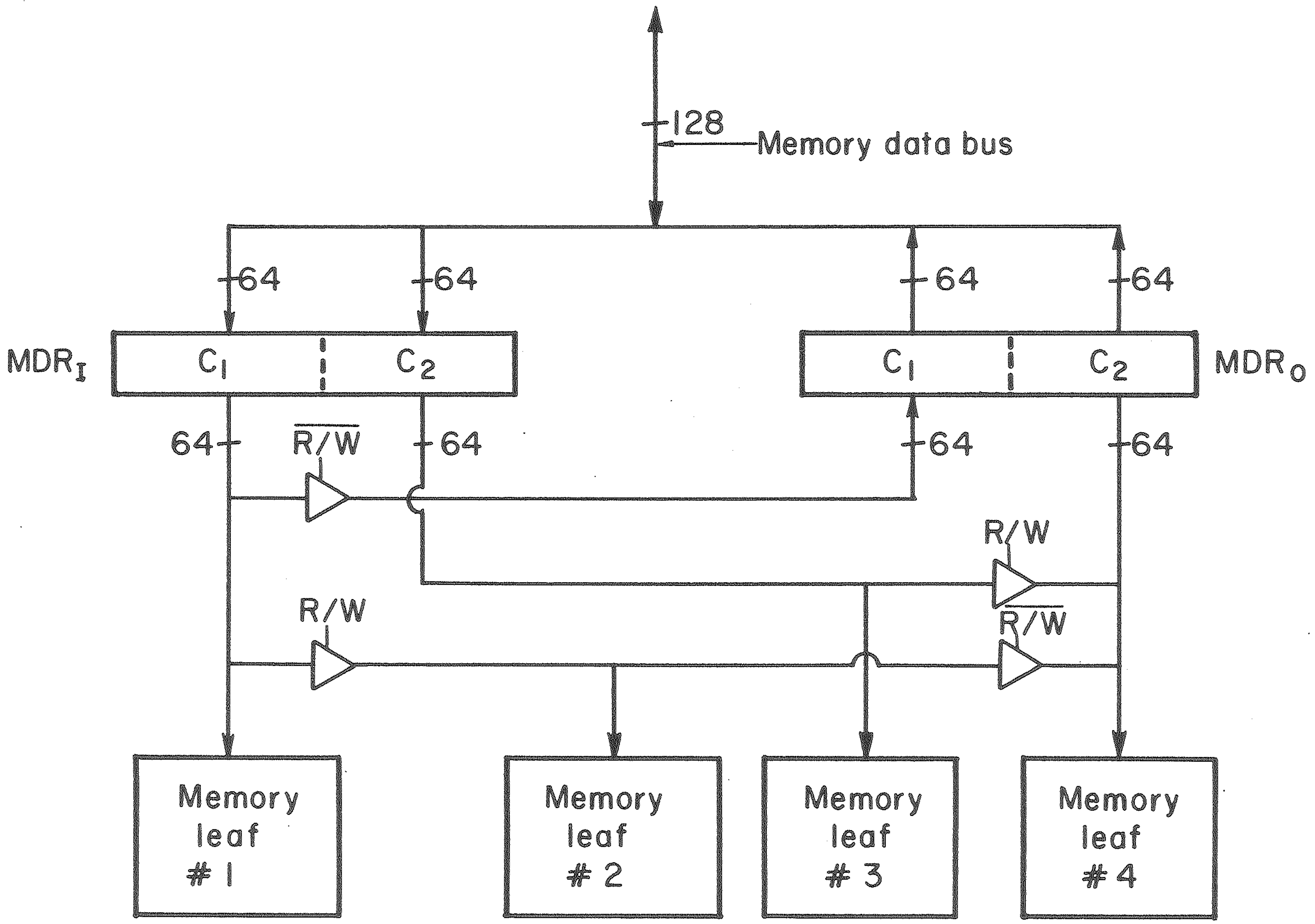
As described in section 3.2, this memory must be interleaved $2^2=4$ ways for radix-2. At any one time, the logical interleaving is only 2-ways to allow both complex values to be accessed in parallel; however, when the memory is an input memory (i.e. in read mode) access must be sequential and when it is an output memory (i.e. write mode) access must be at intervals of 2. This flexible interleaving architecture requires an array of 3-state buffers as shown in Figure 16 even though the memory chips and registers have 3-state outputs. Careful examination of data paths during read and write cycles shows that a single data bus cannot be constructed that will function properly in both read and write modes. The 3-state buffers serve to configure to separate busses, one for reading and one for writing.

The longest transform which may be computed is dictated by memory size which in this case is 4K. Since there must be four memory leaves, each will be 1 K long. The memory chips which were chosen for this processor are the AMI 2114A-1 1K x 4-bit VMOS semiconductor memories with a cycle time of 150 nanoseconds. Each memory leaf contains 16 of these chips.

Timing and control of the memory is relatively straight forward. At the beginning of each memory cycle, the address is clocked into the MAR. Which memory leaves will be accessed during this cycle is a function of the LSB (least significant bit) of the address and whether the memory is in read or write mode. This information is in the form of a chip select signal. The read/write signal and the remaining address bits are presented to the memory chips after a fixed delay of 50 nsec. which allows the signals to settle.

After 150 nsec. (the cycle time of the memory chips) the data has either been written into the memory or read from the memory. When in read mode, the data are clocked into the memory data register at the beginning of the subsequent clock period.

Figure 16



XBL802-3096

When a given memory is serving as processor input or output, a new cycle is initiated at each clock period. However, when it serves as either computer input/output a new cycle is initiated only when the computer has filled/emptied the MDR from the previous cycle. The input/output interface informs the memory when to cycle via two control lines, one for input and one for output. Further details on this are given below in section 5.7.

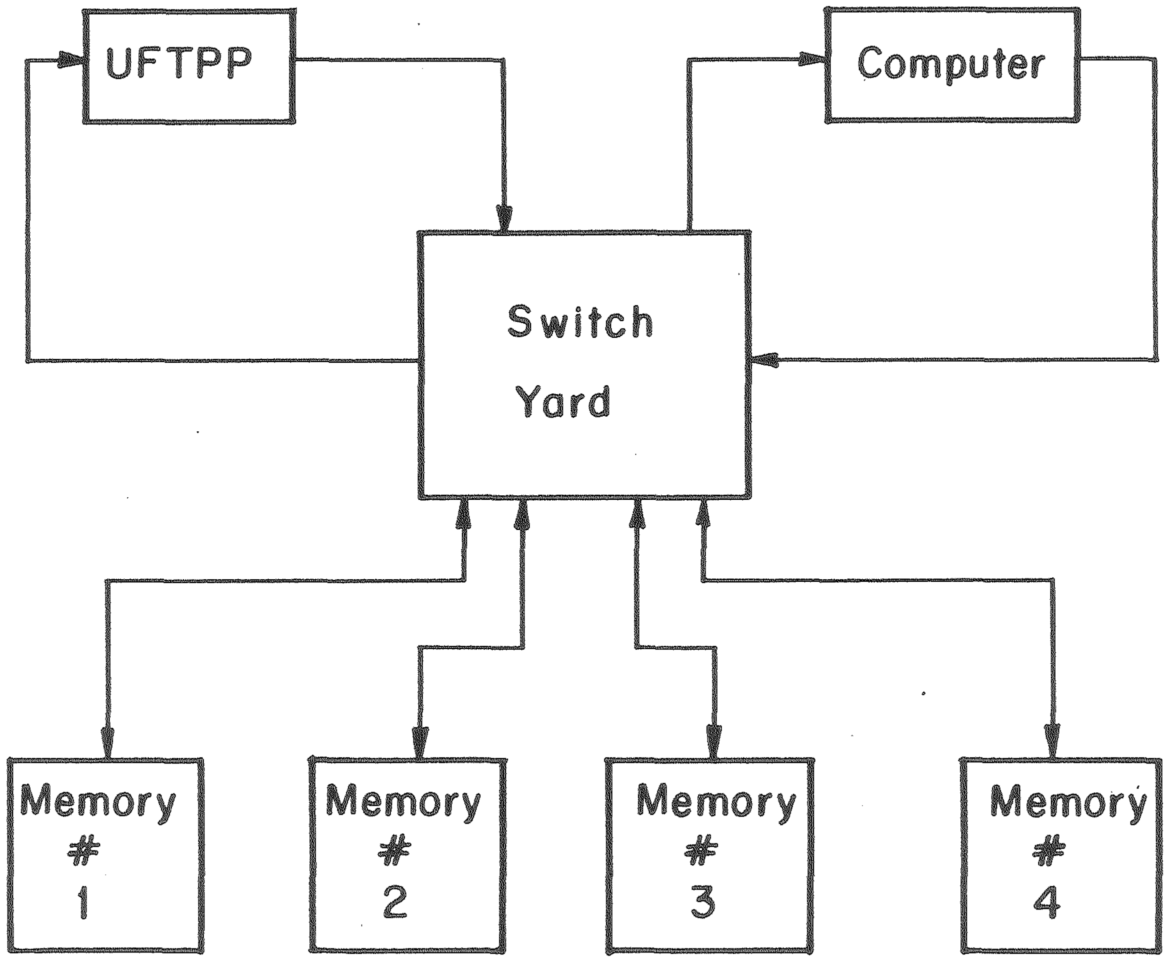
The overall control of what logical function a physical memory will serve is a function of four variables. The first is the iteration number within the transform since at the end of each iteration, the processor input and processor output memories must exchange roles. Secondly, the size of transform affects which memory will contain the final results of the transform. When $m = \log_2 N$ is odd, then the memory that was the processor output during the first iteration will contain the final results. When m is even, the original processor input memory will contain the final results. The memory that contains the results must become the computer input buffer on initiation of the next transform. The other will become the output buffer. The third and fourth variables are specified by a 2-bit counter which cycles through 4 distinct states to allow each memory to occupy one of the four functional positions.

Table V shows the function each of the four memories (numbered 1-4) will perform for each of the 16 combinations of the four control variables. M is the LSB of the value $\log_2 N$ (N is the size of the transform), I is the LSB of the iteration number (beginning at 0) and MC1 and MC0 are the MSB and LSB of the 2-bit memory control counter, respectively. UI indicates UFTPP input, UO indicates UFTPP output, CI computer input and CO computer output.

This state table is easily converted to truth tables for the read/write and other control signals needed for memory operation.

5.4. The Multiplexer System

The multiplexer system is a switchyard which connects four fixed direction busses with four bidirectional busses as depicted in Figure 17. The four fixed direction busses are the processor input bus, PHYSICAL SET-UP



XBL792-3207

Figure 17

TABLE V

				Memory Function			
M	MC1	MC0	I	UI	UO	CI	CO
0	0	0	0	1	2	3	4
0	0	0	1	2	1	3	4
0	0	1	0	4	3	1	2
0	0	1	1	3	4	1	2
0	1	0	0	2	1	4	3
0	1	0	1	1	2	4	3
0	1	1	0	3	4	2	1
0	1	1	1	4	3	2	1
1	0	0	0	1	2	3	4
1	0	0	1	2	1	3	4
1	0	1	0	4	3	2	1
1	0	1	1	3	4	2	1
1	1	0	0	1	2	3	4
1	1	0	1	2	1	3	4
1	1	1	0	4	3	2	1
1	1	1	1	3	4	2	1

processor output bus, the computer input bus and the computer output bus. The bidirectional busses are the four memory busses. Each bus has a memory address associated with it.

Each of the four multiplexer boards house approximately 100 chips and draws about 6-8 amps. This yields 400 chips and 24-32 amps for the entire multiplexer system.

The source and destination of the four data paths which flow through the switchyard are determined from Table V. The same four control lines which go to the memory are inputs to the multiplexer. The multiplexer itself consists of all of the selectors shown in Figure 12. There are two 1-of-4 selectors which determine the processor input and computer input memories and there are four 1-of-2 selectors which determine which memories receive data from the processor and computer. Since the memory busses are bidirectional, these selectors are 3-state and each is enabled only when the memory to whose bus it is connected is in read mode.

As can be seen from Figure 17 the entire multiplexer has a total of 128 x 8 = 1024 I/O lines. This is clearly too many for implementation on one board. Implementation of even one of the 1-of-4 selectors would require 128 x 5 =640 lines which is too many given the board layout described in section 5.1. However, the "word-slicing" approach may be applied here again. If the eight switchyard inputs were reduced from 128 to 32, then the total number of connections is reduced to 256 which is within the limit of 320 available. This approach requires 4 boards to implement the entire multiplexer. Each board contains the two 1-of 4 selectors and four 1-of-2 selectors. They are each 32-bits wide.

The result is that each data bus is split into 4 parts as it enters the multiplexer and reassembled as it leaves the multiplexer. The four address busses, each 12 bits wide and associated with one of the data busses, is treated in the same manner. However, due to hardware implementation considerations each was split into three slices of 4-bits each. The address selectors add 32 to the total number of connectors necessary for a total of 288.

The four control lines are inputs to combinatorial logic which derives the enable and select signals for each individual selector. Since the word-slice approach is used, the control logic is identical for each card.

Due to connector restraints on the arithmetic boards, where the input and output interfaces are housed, the computer input and output busses are not implemented exactly as indicated in Figure 12. Rather than 128-bit busses, they are 32-bit busses that are time division multiplexed. There are four time divisions. The data that is present on a bus during each time division is determined by two control lines from the corresponding interface. Each multiplexer board responds to only one of the four possible conbinations of the two lines and thereby the 128-bit pair of complex values is slowly built up and then written into the memory in only one cycle.

5.5. The Pipeline

The memory, arithmetic unit and multiplexer are connected together in a pipeline architecture as described in Chapter 3. The general flow of data is from the memory, through the multiplexer, into the arithmetic unit, out of the arithmetic unit, through the multiplexer and back into the memory. The clock synchronizes movement of data from one stage to another. There are a total of 15 stages in the UFTPP pipeline. The functions performed at each stage are:

0) Generate Input Data Memory Address
   Generate SIN/COS address

1) Route address through the multiplexer.
   Bit reverse and mask SIN/COS address

2) Read data from memory
   Read SIN and COS

3) Route data through multiplexer
   Negate SIN if this is a forward transform

4) Complex multiplication - Stage #1

5) Complex multiplication - wait for LSI multiplier chips

6) Complex multiplication - Stage #2

7) Complex multiplication - Stage #3

8) Complex multiplication - Stage #4

9) Complex multiplication - Stage #5 - check for overflow

10) Perform Butterfly - check for overflow

11) Normalize result if this is a forward transform
    Generate output address and shuffle

12) Route butterfly results and address through the multiplexer

13) Write results into memory

After 13 clock periods (the latency of the pipe), the pipeline is full and results for that iteration begin to emerge. At any given time there are 13 data items being processed and are at various stages of completion.

5.6. Overall Timing and Control

When the perfect shuffle routing algorithm is used, very little control logic is necessary. Most control functions are concerned with global operations such as detection of iteration completion, transform completion and host computer I/O. The only functions directly tied to the perfect shuffle are the address generators. The control section of the UFTPP is composed of the following functional blocks:

1) Perfect Shuffle Address Generator
2) Control/Status Register
3) N - register
4) Iteration Counter
5) Finite State Generator
6) Memory Counter
7) Clock Generation and Distribution
8) Computer Input Interface
9) Computer Output Interface

5.6.1. The Perfect Shuffle Address Generator

As noted in section 2.3, perfect shuffle address generation involves a left circular shift of the input address by r-1 bits. Hardware implementaion of this is trivial for fixed n where n is the number of bits in the address to be shuffled. For variable n, classical digital design techniques may be employed to implement a circuit to select the correct bit(s) and insert them in the low order position(s) while the other bits are shifted to the left.

Two perfect shuffle address generators are necessary for the UFTPP. One to generate the processor output memory addresses and one for the computer output buffer address.

5.6.2. Control/Status Register and the N - Register

The control/status register (CSR) is the main vehicle for communication with the host computer. It is a seven-bit register consisting of a GO bit, a computer input buffer empty flag (B_i), a computer output buffer full (B_o) flag, a transform complete flag (T), an overflow flag (O), a transform direction flag (F) and a reset bit (R). All bits are readable and writable from the computer. The N - register is a 12-bit register that is loaded with the size of transform to be computed by the host computer.

The B_i, B_o, and T flags are self-explanatory. They are cleared at transform initiation and are set when the appropriate condition exists. (For example, B_i is set after N complex data values have been sent to the computer.) The Overflow flag (O) is cleared at transform initiation and set if at any time during the transform computation an overflow is detected. This flag is purely an error indicator and does not affect the operation of the processor in any way. The transform direction flag tells the processor whether to perform a forward (F = 1) or inverse (F = 0) transform. The GO and RESET bits are the only two "commands" that the processor understands. When the RESET bit is set, the processor performs a master reset and comes to a state of readiness to begin processing. This bit is also accessible through a button on the front panel. The GO bit initiates a transform as well as readiing the buffers to be filled/emptied.

5.6.3. The Iteration Counter

The iteration counter is a four-bit counter that keeps track of the present iteration of the transform being computed. It is cleared at transform initiation and incremented each time that the output address counter reaches N, the size of the transform. When it reaches $m = \log_2 N$, the transform is complete and the T bit is set.

5.6.4. The Finite State Generator

The UFTPP is a finite state machine and the phase of execution is under the control of the finite state generator. The 6 states which the machine may be in are:

0) Initialization

1) Initialization pause

2) Compute state

3) Iteration transition

4) Iteration pause

5) Transform pause

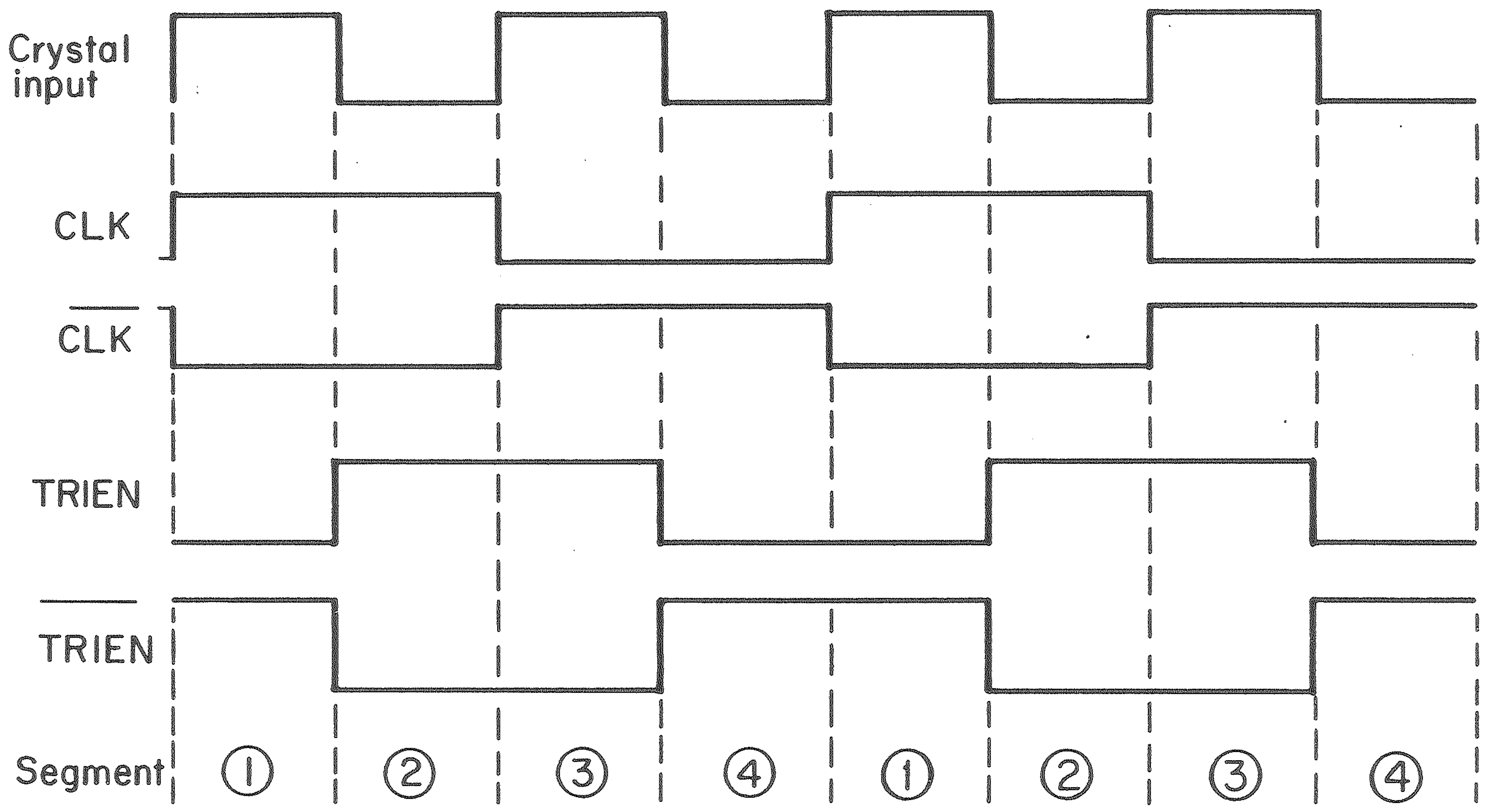
State 5 is the state which is entered when RESET is asserted. The processor remains in this state until GO is asserted. The transition is then made to State 0 where all address counters, control/status flags and the iteration counter are initialized and the memory control counter updated. Upon the initiation of the next clock period, State 1, a wait state, is entered which is to allow control signals derived from the just initialized signals to settle. The following clock transition initiates State 2, during which one iteration of the transform is computed. When the iteration is complete, State 3 is entered. Here the processor input and output address counters are reinitialized and the iteration counter incremented. State 4 is entered on the next clock transition. During this state, the transform complete signal is generated from the new iteration value. If the transform is complete, State 5 is entered, otherwise State 1 is entered and another iteration is begun.

5.6.5. Memory Counter

The memory counter is a two-bit counter that controls which physical memory will perform which logical function as outlined in section 5.4. It is initialized only when RESET is asserted. Otherwise, it is incremented each time a new transform is initiated. Basically, it assures that the memory which contains the results from the last transform becomes the next computer input buffer and that the last computer output buffer becomes the processor input for the first iteration. The transition table which defines the functions of each of the four memories was given in Table V, section 5.5.

5.6.6. Clock Generation and Distribution

Most operations in the pipeline are combinatorial and do not require any sub-pipecycle timing (a pipecycle is the period between positive transitions of the pipeline clock). There are two operations that do require it, however. These are memory reads/writes and the 16-bit multiplications. For this purpose, the pipecycle has been divided into 4 segments which are delimited by transitions of two signals, the pipeline clock (CLK) and a tri-state enable signal (TRIEN) which is 90 degrees out of phase with the clock. These signals are supplied in both true and inverted forms to ease decoding as shown in Figure 18.



XBL802-3097

Figure 18

In the case of the memory, the first quarter of the pipecycle is used to allow the address bus to settle and derive the chip select and write enable signals. The remaining three-quarters is the memory access time.

The LSI multipliers have internal input and output registers which must be loaded and unloaded. Due to this architecture, the multipliers form their own segment in the pipeline (see pipeline stage 5, section 5.5 above). During segment 1 of the pipecycle, the new inputs to the LSI multipliers are clocked into external registers and the product from the previous cycle is clocked into the input register of the next pipe segment. During segment 2 of the pipecycle (beginning with the positive transition of TRIEN), the input registers of the multipliers are enabled and the output registers disabled. During segment 3, the data for the new inputs are clocked into the internal registers and the product from the previous set of inputs is clocked into the internal output registers. Segment 4 is used to enable the output registers so that they may be unloaded at the beginning of the next pipecycle.

A 20 MHz crystal is used as the timebase source from which both the CLK and TRIEN signals are derived. The UFTPP is designed to run at a minimum pipecycle duration of 200 nsec. This makes each segment of the cycle 50 nsec long. The CLK and TRIEN signals are transmitted to the other boards differentially to allow for minimal signal degradation. Once on the destination board they are converted back to single signals and distributed to the chips using standard TTL clock drivers.

5.7. Computer I/O Interfaces

The UFTPP operates as a completely asynchronous system from its host processor. The interfaces serve to synchronize data flow to/from the host from/to the processor input and output buffer memories. The processor under discussion here was interfaced to a PDP11/34 minicomputer which operates using 16-bit word I/O data transfers. Thus, 4 I/O operations are required to transfer one complex pair of data values. The interface sends/receives the 16-bit parcels to/from the computer and generates requests for memory cycles after every eight computer I/O transfers (since the memory writes two complex data values per cycle).

The input and output interfaces each has a memory address generator associated with it. The output interface receives data from the computer to be transformed. Since the Pease algorithm used here requires data shuffling previous to each iteration (see Eq. 15, ch. 2), there is a perfect shuffle address generator associated with the output interface. There is also a comparator attached to the address generator whose output sets the B_0 flag in the CSR when N complex values have been received.

The Pease algorithm produces its output in bit-reversed order as noted in Chapter 2. However, the algorithm is a pre-shuffle (the data are shuffled before each iteration) algorithm as noted above, but the output from the arithmetic unit is always shuffled when it is written into the output memory. Thus, an extra shuffle is performed. This has an interesting and useful effect. The results from Eq. 15 are stored at addresses which are bit-reversed values of the frequency bins they represent (see also Sec. 2.1). The extra shuffle performs another left circular shift of this address by r-1 bits (in the case of radix 2, one bit). This might be symbolicly represented as

$$(n_0,n_1,n_2,\ldots,n_{m-1}) \longrightarrow (n_1,n_2,n_3,\ldots,n_{m-1},n_0)$$

The result is that the frequencies are bit-reversed pairwise by their m-2 most significant bits. This is fortuitous in the case of the UFTPP architecture since the memories access data in sequential pairs during read cycles. Because the LSB of the frequency has been shuffled back into the least significant position, the frequency coefficients are stored in sequential pairs in the output buffer. The address generator for the computer input interface is a variable length bit reverser which operates on m-2 bits. This address is then routed to the appropriate memory and the frequency values are accessed in the normal pairwise fashion. The interface then sends the results to the computer in unscrambled order. Because this access is not in any sequential fashion, it is an argument against using a memory based on a sequential technology (such as CCD or magnetic bubbles) and for truly random access memories.

5.8. Computer Operation and I/O

The UFTPP can be veiwed as an extremely fast subroutine by the programmer. As currently implemented, the program must explicitly load data and unload results, in a programmed I/O fashion. However, if the computer side of the of the interface were given DMA (direct memory access) capabilities, the results would be loaded and unloaded transparently to the program. This would not require any change in the UFTPP side of the interface or to any other part of the UFTPP.

The UFTPP is designed to perform transforms continuosly in parallel with I/O to/from the computer. Each time the processor is instructed to GO, the previous results are made ready for trasmission to the host and the data just loaded from the computer are transformed. A single transform would have the following operational sequence:

(1) Initialize processor with RESET

(2) Issue GO command to ready computer output buffer

(3) Load UFTPP with data

(4) Issue GO command to perform transform

(5) Wait for completion

(6) Issue GO command to ready output buffer

(7) Unload results

When performing several transforms in succession the command stream is considerably streamlined:

(1) Initialize processor with Reset

(2) Issue GO commend to ready output buffer

(3) Load UFTPP with data; unload results unless this is first transform

(4) Issue GO command to perform transform

(5) Wait for completion

(6) Go to step 3, unless all data have been sent already

(7) Issue last GO command to ready last results

(8) Unload last results

5.9. Implementation Costs

The implementation costs for the prototype UFTPP were moderate to low by computer standards. The actual costs are enumerated in Table VI. The sixteen 16x16-bit multipliers were donated by TRW, Inc. Even including the cost of the multipliers, the $14.5K price tag is very low in comparison to the cost of array processors with similar computational capabilities.

TABLE VI

UFTPP Cost Summary

Memory Chips	$2750
Other Integrated Circuits	$4500
Printed Circuit Boards	$1000
Mechanical Hardware	$1300
Power Supply	$ 695
Chassis	$ 300
Multipliers (Donated)	$4000
Total	$14550

CHAPTER 6

APPLICATIONS

6.1. Implementation

The UFTPP prototype was connected to a host DEC PDP-11/34 computer at the Donner Laboratory (part of the Lawerence Berkeley Laboratory), and integrated into the data acquisition and reconstruction system [40] for the Donner 280-crystal Position Emission Tomograph [25]. The processor clock speed was set at 2.5 MHz. When compared to software running on the host or on a CDC 7600 computer, the UFTPP was faster by factors of 3400 and 17, respectively. Exact times are given in Table VII.

Two-dimensional (256 x 256) complex image transforms were performed using a series of one-dimensional transforms as described by Gonzalez and Wintz [35]. The intermediate result from the first pass were stored on disk due to memory limitations. Transposition of the rows and columns of the intermediate result was performed in place using the algorithm by Eklundh [29]. Following the second pass the results were again stored on disk as well as displayed on a 256 x 256 gray-level display. Inverse transforms on the computed frequency spectrum were performed in a similar manner.

6.2. Image Processing Applications

6.2.1. High & Lowpass Filtering

System software was developed to allow generalized 2-D filtering of images. Image filtering in general is the process of altering the

TABLE VII

(time in milliseconds)

N	PDP11/34	CDC7600	UFTPP
64	244	1.5	.077
128	581	3.1	.179
256	1316	6.8	.410
512	3137	15.2	.819
1024	7088	33.2	2.048
2048	16198	70.3	4.506
4096	33667	152.0	9.830

frequency content of an image. The operation may be performed in either real space using convolution or in frequency space using multiplication. When the size of the convolution kernel is small, there are advantages to performing the convolution in real space. However, for generalized 2-D filtering where one desires the capability of manipulating all frequency components up to the Nyquist limit, frequency space manipulations are usually superior. This is due to the fact that 2-D convolution requires the order of N^4 operations while 2-D Fourier filtering requires only $4N^2 \log_2 N + N^2$ operations ($2N^2 \log_2 N$ for the forward transform, N^2 for the filtering, and $2N^2 \log_2 N$ for the inverse transform). Thus when $4\log_2 N + 1 < N^2$ (i.e. $N \geq 4$) Fourier filtering requires less time.

If we have an image i(x,y) with Fourier transform I(u,v) and a filter kernel f(x,y) with Fourier transform F(u,v), then the filtering operation may be expressed as either

$$g(x,y) = i(x,y) * f(x,y)$$

(where * denotes convolution), or

$$G(u,v) = I(u,v) \times F(u,v)$$

F(u,v) is usually referred to as the transfer function of the filter and determines which frequencies are passed unaltered and which frequencies are attenuated or stopped completely. Often a filter is defined directly in frequency space via its transfer function.

The software written to implement generalized 2-D filtering includes 7 different classes of frequency space windows: ideal, exponential, trapezoidal, Hamming, Hann, Butterworth, and Parzen. For each filter window, the user may specifiy the cutoff frequency for the selected window. The definition of ideal, exponential and trapezoidal windows may be found in Gonzalez & Wintz[35], the definition of the Hamming, Hann and Butterworth windows may be found in Hamming [39]. For a given cutoff frequency, it is well known that the shape of the window effects both resolution of the image and artifacts introduced by the filtering process. The steeper the rolloff the higher the resolution, but the more artifacts introduced.

Different imaging situations have different requirements and require different window shapes. The cutoff frequency of a filter is the frequency at which the transfer function goes to zero. In the lowpass case, this defines the maximum resolution of the filtered image. However, if the window of the filter is not rectangular, the resolution of the image is further degraded by the rolloff of the window.

Lowpass filtering (i.e. passing low frequencies and attenuating high frequencies) is usually performed to suppress statistical noise which is dominant in the higher frequencies of an image. Noise suppression comes, of course, at a cost of lower resolution since the high frequency power associated with resolution is attenuated by the filter. In an image without statistical noise, the high frequencies contribute to the definition of the edges in the image. Figure 19 shows an example of an image without noise and the result after lowpass filtering with a Hamming filter with cutoff frequency of 85 cycles per domain or the delineation of the edges in the result is less clear since the image has been blurred by the low pass filter.

High pass filtering is a sharpening operation used for removing isotropic blurring or edge-detection. In high pass filtering, the high frequency components associated with abrupt changes in gray-level are passed and the low frequencies are attenuated. An example of edge detection using a highpass Butterworth filter is shown in Figure 19.

6.2.2. Matched Filtering

Up to this point the discussion of filtering has been concerned with radially symmetric high or low pass filters. There are cases where nonsymmetric filters are of use. One example of this is matched filtering. The operation of matched filtering involves the computation of the correlation function between an image and a template to determine if the pattern present in the template is also present anywhere in the image. Brigham [12] shows that correlation functions are easily computed in Fourier Space by multiplying the Fourier transform of the initial function by the complex conjugate of

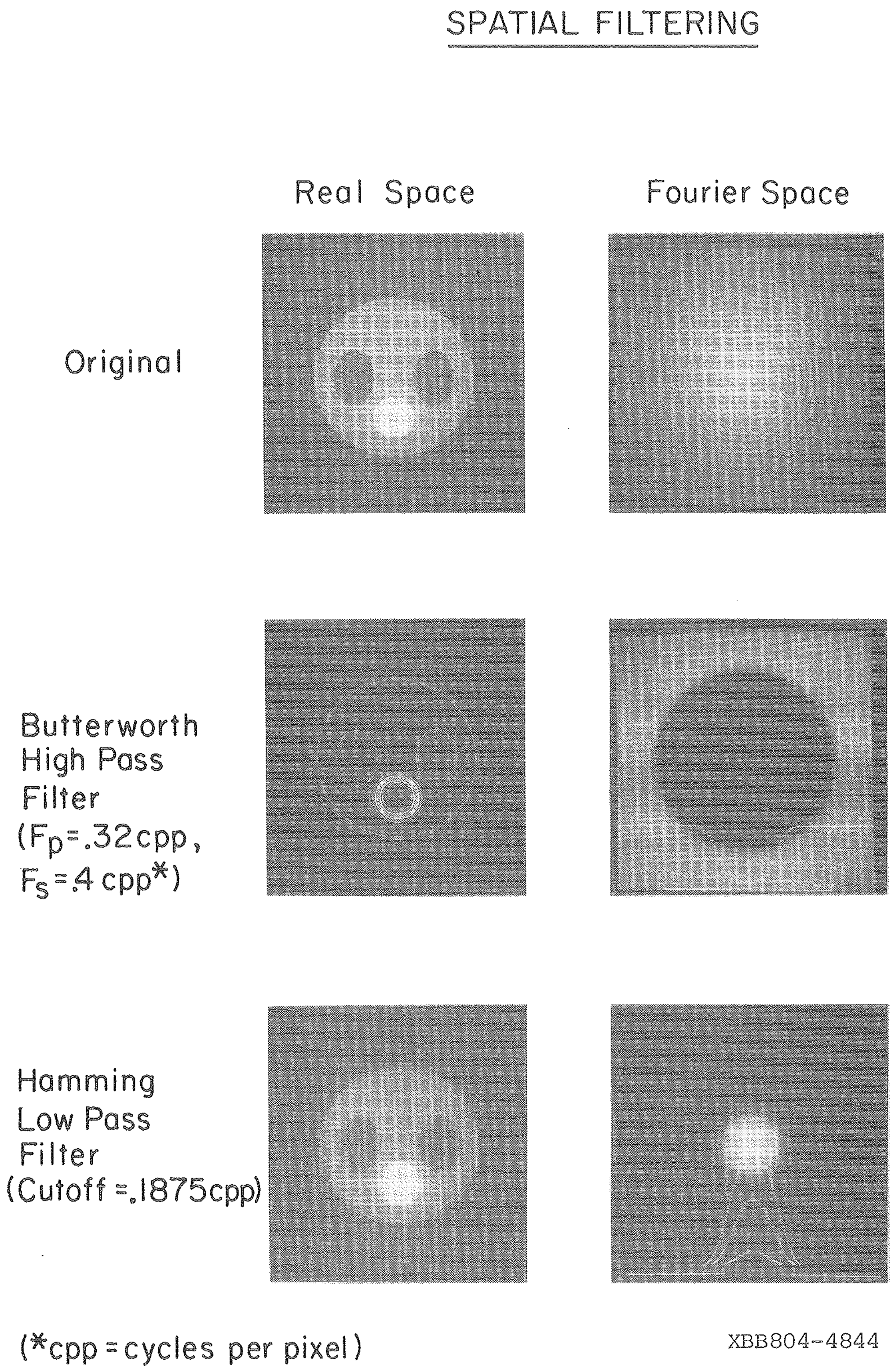

SPATIAL FILTERING
Real Space
Fourier Space
Original
Butterworth High Pass Filter (F_p=.32 cpp, F_s=.4 cpp*)
Hamming Low Pass Filter (Cutoff =.1875cpp)
(*cpp = cycles per pixel)
XBB804-4844

Figure 19

the Fourier transform of the correlation function followed by inverse transformation of the result.

Figures 20 & 21 show two examples of matched filtering. The initial image is shown in the upper left and the template to be matched in the upper right. The full correlation function is shown in the lower left. Notice that all objects in the image yield some degree of correlation. However, after threshholding only the objects with good matches to the template stand out. The object which matches the template best in size, shape and orientation yields the largest signal. Note the bright point in the upper right quadrant of Figure 20 which corresponds with the object which exactly matches the template. Figure 21 shows the same initial image matched with another template. The end result in this case (lower right) shows maximum correlation in the two positions corresponding to the two objects most similar to the template.

6.3. Image Reconstruction From Projections

The problem in image reconstruction is to recover an image, i(x,y), from a set of its projections, p(r,θ). The projections are sets of line integrals through the object and are assumed to be related to it by the expression

$$p(r,\theta) = \int_{-\infty}^{\infty} i(x,y)\,ds$$

where r and s are the coordinates of a system rotated by an angle θ from the fixed (x,y) system and are related by

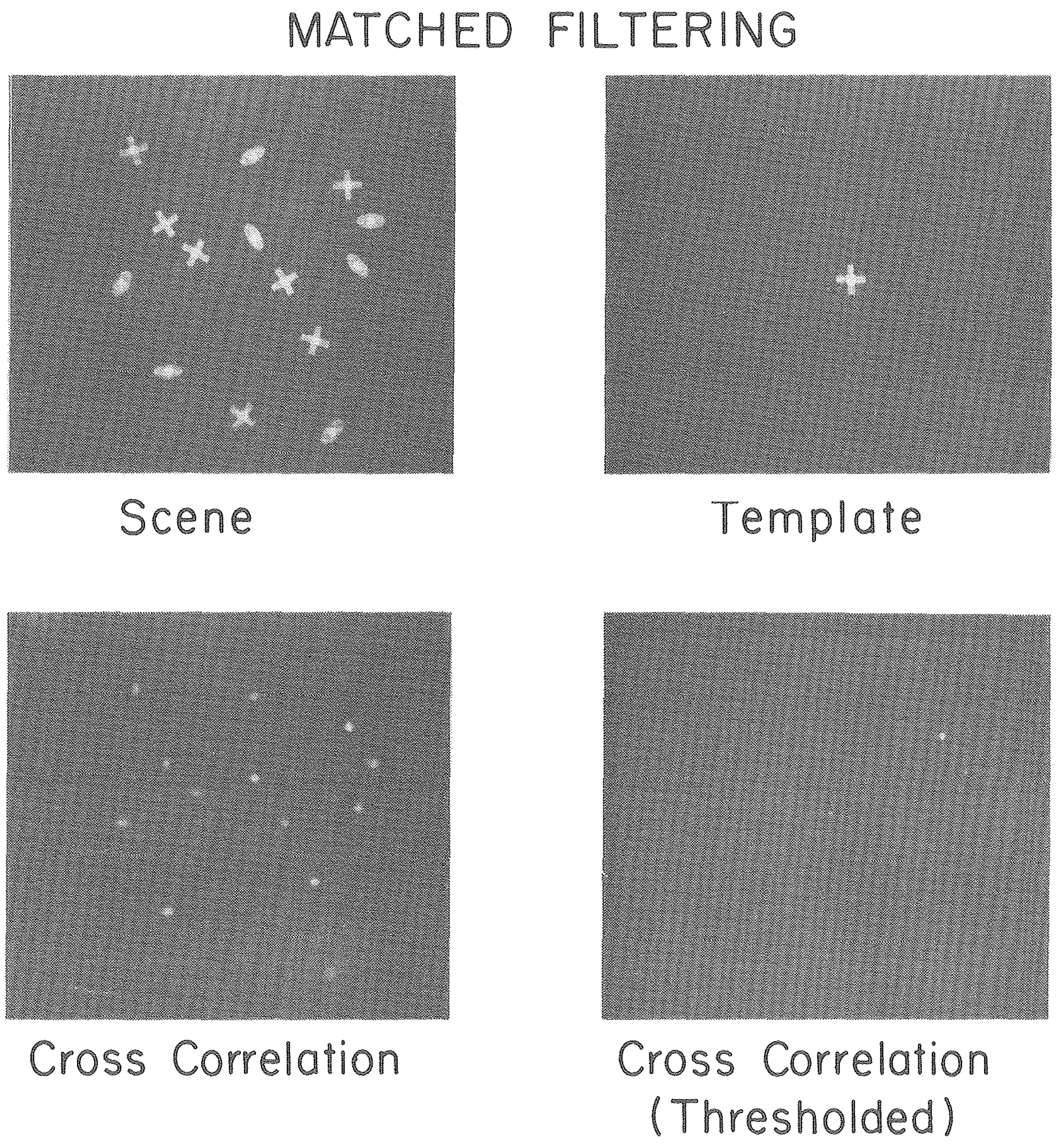
$$x = r\cdot\cos\theta - s\cdot\sin\theta$$

$$y = r\cdot\cos\theta + s\cdot\cos\theta$$

The basic operation in reconstruction is called backprojection and is given by the expression

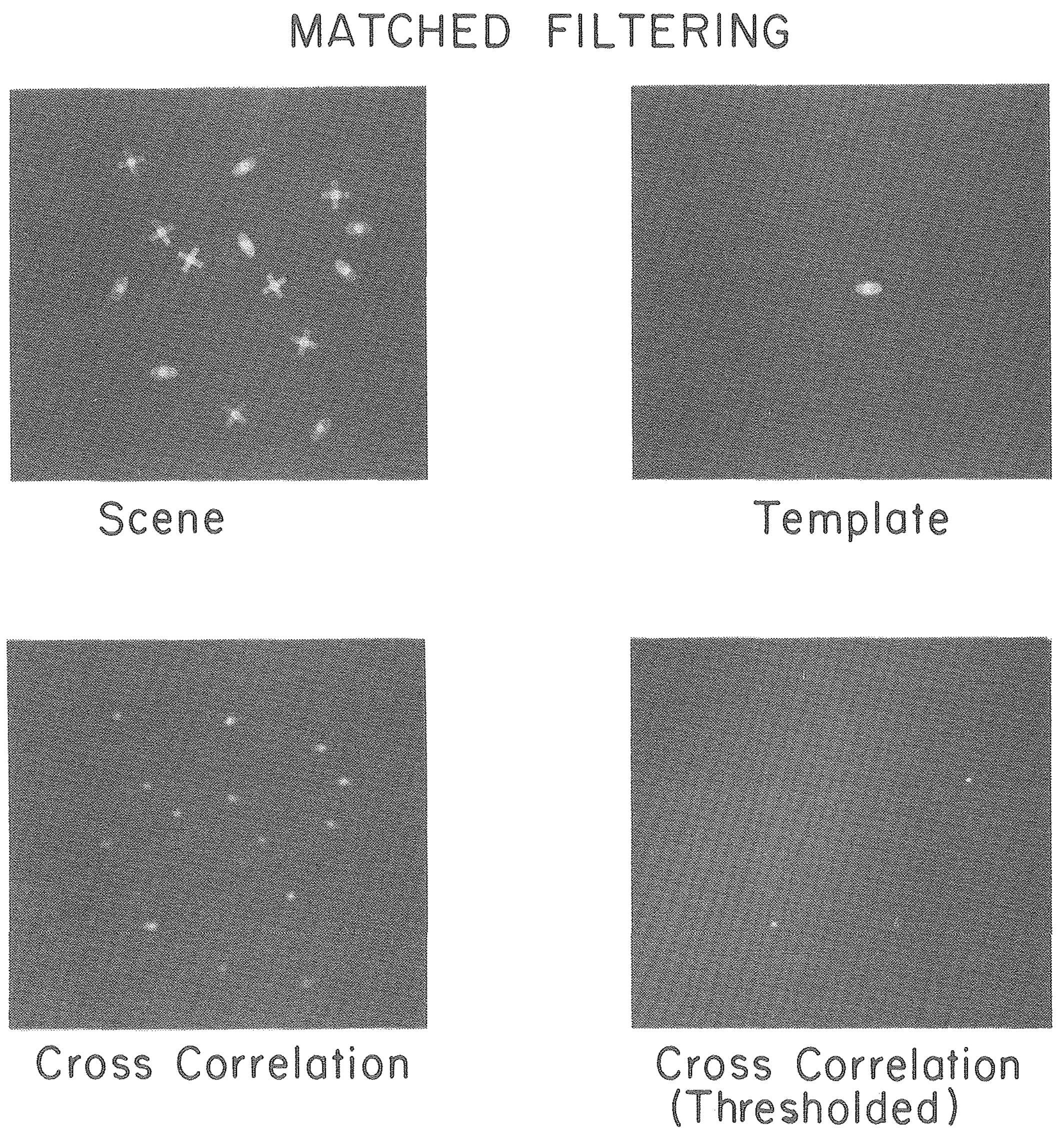
$$b(x,y) = \int_{0}^{\pi} p(x\cdot\cos\theta + y\cdot\cos\theta,\theta)\,d\theta$$

However, it has been shown that b is not the original image i, but rather i convolved with a smearing function whose frequency spectrum



XBB804-4841

Figure 20



XBB804-4840

Figure 21

is

$$Z(u,v) = |R|^{-1} = (u^2+v^2)^{-1/2}$$

Since b=i*z, it is immediately apparent that i may be recovered by the deconvolution or filtering of b in 2-space with a filter whose transfer function is

$$A(u,v) = |R| = (u^2+v^2)^{1/2}$$

This algorithm is known as the Filtered Backprojection.

It has also been shown [2] that this filtering operation is commutative with the backprojection operator and may be carried out on the projections before backprojection with equivalent results. This algorithm is known as the Backprojection of Filtered Projections.

When the projection data used for reconstruction contains statistical fluctuations, the ramp filter, A(u,v) (given above), amplifies the high frequency noise in the reconstructed image. In order to suppress this noise, the ramp may be rolled off by a window function such as one of those given above. The noise suppression comes, of course, at the expense of resolution.

Image reconstruction from projections is fundamental to Computer Assisted Tomography (CAT), the recently developed diagnostic medical tool which produces cross-sectional images of the human body. The images may be collected in two modes, transmission or emission. In transmission mode, an external X-ray source is passed through the body and an attenuated beam emerges on the other side. The projection data are a function of the ratio of the emergent beam strength to the incident beam. The reconstructed image represents the distribution of attenuation coefficients in the plane of the body being imaged. In emission mode, the object is to map an unknown distribution of radioactive source. The data at each point of a projection collected in this mode are a sum of all the source strength along the line perpendicular to the projection and intersecting it at that point. Emission CAT (ECAT) suffers from the further problem that as the radioactivity leaves the body it is attenuated by the tissue that it passes through in the same manner as in transmission mode. However, the initial source strength is not known nor is the point of emission. Thus, one does not know how much tissue a given photon has travelled through so that correction for attenuation is difficult.

Gullberg [38] has recently developed a method for determining window functions which can be applied to the ramp filter to yield a transfer function which when used for reconstruction will correct for the effects of constant attenuation yet maintain a known point spread function (resolution). Attenuation correction in single-gamma ECAT has previously been very time consuming and computer memory expensive. However, this technique incorporates the attenuation correction into the filtering step of the Backprojection of Filtered Projections algorithm. Whereas other techniques of compensation for attenuation in ECAT do not require Fourier transform operations, this apparently superior method needs to use Fourier transformation of for its implementation. Thus, the need for a rapid Fourier transform method.

Since the patient is exposed to radiation in either the transmission or emission mode of CAT, it is always desirable to minimize the amount of exposure. Limiting the number of photons collected as projection data lowers the signal-to-noise ratio (SNR) in the projections. This lowered SNR is transferred to the image (and degraded further) by the reconstruction operation. Tsui [68] has recognized that the global SNR of a projection is not distributed uniformly among the frequency components of it. Using the Weiner formulation for a stochastic filter, he has developed an algorithm for determining a new transfer function at each projection angle which not only removes the 1/R blurring, but suppresses noise by weighting the frequency components of the projection according to the SNR at each frequency. That is, the frequencies which exhibit high SNR are passed relatively unattenuated and those with low SNR are highly attenuated. The determination of the transfer function requires computation of the power spectral density at each frequency. Once the filter is determined it is applied to the projection data before backprojection. Tsui has shown that this technique shows excellent results for large dose reductions.

The form of the minimum mean-squared error filter developed by Tsui is:

$$H(f) = \frac{|f|}{1 + \gamma\frac{|f|\bar{m}}{S(f)}}$$

where

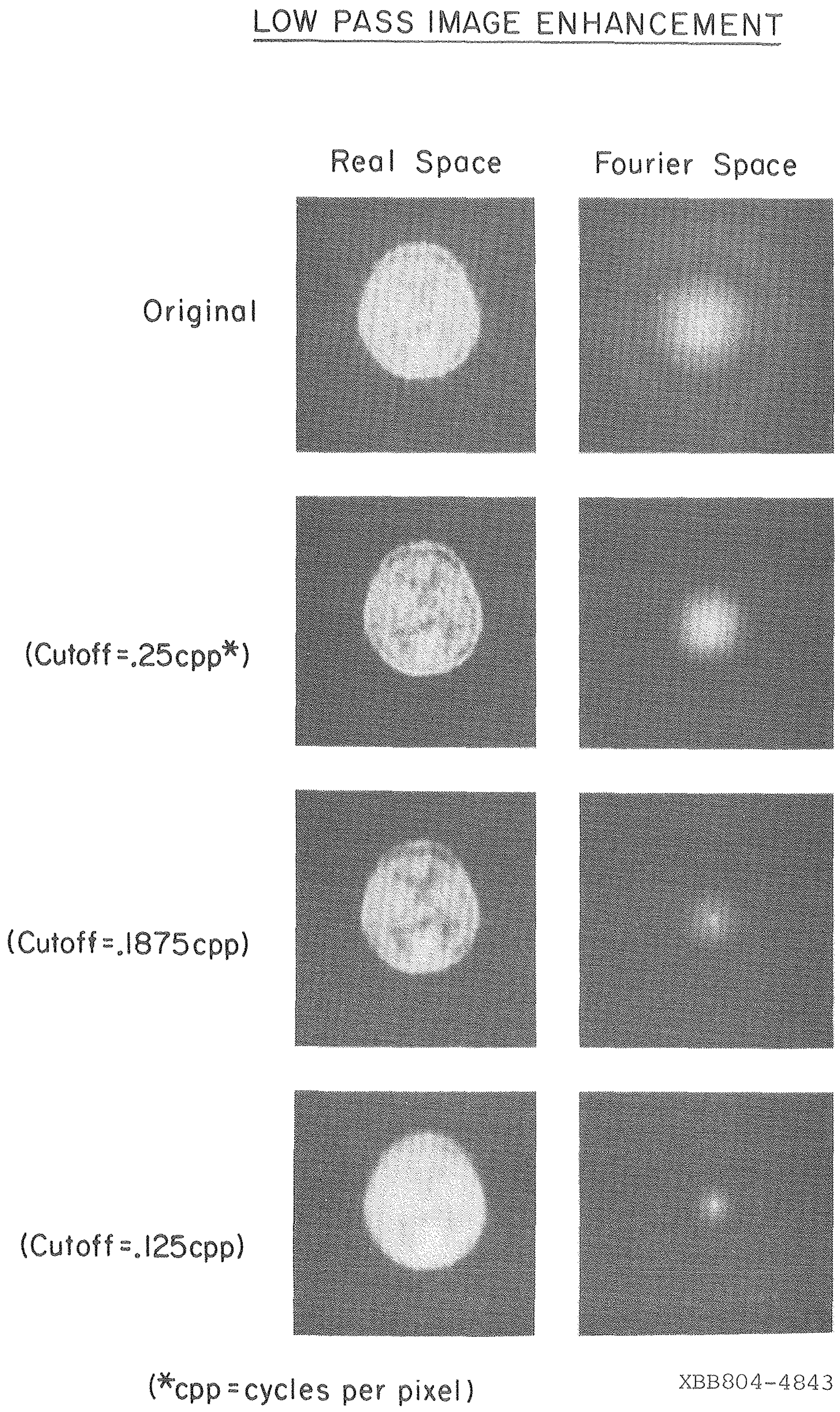
$\underline{f}$ is the frequency
$\bar{m}$ is the average value of the projection data for the angle under consideration
$S(f)$ is the value of the power spectrum of the projection data at f
γ is a gain factor which effects the smoothing characteristics (noise suppression)

Implementation of this reconstruction algorithm requires computation of the power spectrum (the square of the Fourier transform) of the projection data at each angle. A new filter transfer function is then constructed using the formula given above. This filter is applied to the projection data before backprojection.

Both of these important advances in reconstruction tomography rely on the capability to compute the Fourier transform. Due to the number of forward and inverse transforms that are necessary to perform a single reconstruction (approx. 150) software implementations of FFT computation are not practical in the clinical setting. However, even with the filtering operation being performed in software, a minicomputer (such as a PDP 11/34) equipped with a UFTPP and a comercially available hardware backprojection device can perform the reconstruction in 30-45 seconds.

Once obtained, the reconstructed images may be postprocessed with to increase visual impact or emphasize interesting structures. An example of a reconstructed emission image of the brain showing C-11 methionine distribution is shown in Figure 22. Three smoothed versions of the original data were obtained by lowpass filtering after reconstruction are also shown in Figure 22. Notice that the ventricles in the center of the brain are more apparent in the image that was smoothed with the Hamming filter whose cutoff was .1875 cpp.

Extraction of temporal relations in multidimensional signal processing requires time domain Fourier transformation of 2-D and 3-D spatial arrays of sequentially acquired data. For example, two-



XBB804-4843

Figure 22

dimensional projection images of isotope distribution in the heart can be obtained every 10 seconds. As many as 100 128x128 images might be collected. Analysis by the methods of Budinger [13] requires 16,384 128-point transforms to be computed.

These are only a few of the applications that high performance Fourier transform processor may be used for in the field of image processing. Investigators in the field of electron microscopy routinely deals with images of dimension 1024 x 1024, 2048 x 2048 and sometimes even 4096 x 4096 and could make excellent use of a UFTPP. The area of digital signal processing in general offers an even wider spectrum of possible applications. The fields of NMR chemical analysis, x-ray crystallography, geophysics and many others have large computation problems to which a processor such as the UFTPP might be applied.

CHAPTER 7

CONCLUSION

A new, flexible architecture for a hardware Fourier transform processor has been developed which can achieve extremely high throughput rates. It can handle variable length transforms and can be configured with anywhere from 1 to N/2 arithmetic units as required by speed or dictated by economic constraints. The outstanding characteristics of the architecture include

(1) A pipelined "butterfly" computation module

(2) Multiple arithmetic units (vertical parallel processing)

(3) Use of LSI multiplier chips

(4) Use of the "perfect shuffle" data routing algorithm

(5) Extremely simple control logic

(6) Low cost

The architecture followed from the fully parallel machine concept proposed by Pease in 1969. This machine uses a perfect shuffle data routing operation to maintain uniformity in data routing for all iterations. However, even today implementation of such a machine is impractical. By means of deriving a perfect shuffle generating function, it was denmonstrated that the output address of an input data value could be derived from the input address. This coupled with the fact that pairs of data from the previous iteration are inputs to only one butterfly operation in the succeeding iteration allows many arithmetic units to be in parallel operation. The perfect shuffle data routing allows addition of arithmetic units in increments of powers of two with the only cost being further interleaving of the memories. A processor with P arithmetic units processes a transform in

$$\frac{N \cdot \log_r N}{r \cdot P}$$

cycles.

The radix of the machine is a very improtant design parameter since each increase in radix by a factor of two yields an increase in computational speed by a factor of four as well as a savings in hardware over four radix-r/2 modules. However, this requires increased memory interleaving, a wider data path throughout the processor, more complex arithmetic units and limits the flexibility in the sizes of the transforms which can be computed.

The value of the flexibility of the UFTPP architecture demonstrates itself best when compared against the cascade architecture. The cascade must have $m=\log_r N$ stages and computes only an N-point transform in 2N execution cycles. The UFTPP may have 1 to N/2 modules with speeds varying from $\frac{N}{r}\log_r N$ to $\log_r N$ cycles. Further, a UFTPP may compute any size (power of 2) transform up to N. The cascade processor cycle time must be the same as the input/output cycle time as it has no buffer storage. However, a UFTPP must have 2N complex words of storage which may serve as an input/output buffer. This allows the processor cycle time to be much faster than the input/output cycle time and allows less hardware to do the job in the same time.

As demonstration of the feasibility of implementing such a processor, a 1 arithmetic unit version was fabricated. The machine has the capability of computing any size transform up to 4096 that is a power of two and uses 32-bit arithmetic. The UFTPP is implemented on 8 circuit boards consisting of 2 boards for the arithmetic unit, 4 boards for the multiplexer and 2 memory boards. Each board is 61 cm. x 47 cm. A total of approximately 1700 MSI integrated circuits and 16 LSI multipliers were used in the implementation. There are 4 physical memories that may serve the logical functions of processor input, processor output, computer input buffer, and computer output buffer. The multiplexer implements the variable connections.

Two-dimensional Fourier transforms were computed using a series of 2N one-dimensional transforms. Several types of spatial filters were implemented in a software system which utilized the UFTPP to filter phantom images and reconstructed Emission Computer Assisted Tomography images. The backprojection of filtered projections algorithm was also implemented using the standard ramp filter which can be rolled off by several different user-selected windows. In addition, the UFTPP can be utilized to constuct special types of filters for use in reconstruction such as the one proposed by Gullberg to correct for attenuation effects or the one proposed by Tsui which yields the minimum mean squared error and allows considerable dose reduction.

Digital signal processing is applicable to almost all areas of physical science. In most areas there are problems which are, at present, computationally intractable or are very expensive in terms of large computer time. The evolution of digital signal processing hardware since the formulation of the FFT algorithm has helped move the boundary of computationally intractable problems steadily back. The UFTPP architecture offers new possibilities for moving that boundary still further back. Its flexibility allows it to conform to most speed or economic requirements. Further developments in integrated circuit technology should make later versions even faster and simpler (in terms of number of ICs). However, even as the boundary is moved back, new even more difficult problems come into view from over the horizon. So goes the battle.

REFERENCES

1. Agarwal, R.C. and Cooley, J.W., "New Algorithms for Digital Convolution", IEEE Trans. Acous., Speech and Sig. Proc., ASSP-25:5, Oct. 1977, 392-410.

2. Bates, R.H. and Peters, T.M., "Toward Improvements in Tomography", New Zealand J. Sci. 14, 1971, 883-886.

3. Bergland, G.D. and Hale, H.W., "Digital Real-Time Spectral Analysis", IEEE Tran. Comput. EC-16:2, April 1967, 180-185.

4. Bergland, G.D., "A Fast Fourier Transform Algorithm Using Base 8 Iterations", J. Math. Comput. 22, April 1968, 275-279.

5. Bergland, G.D., "Fast Fourier Transform Hardware--An Overview", IEEE Trans. Aud. and Electroac. AU-17:2, June 1969, 104-108.

6. Bergland, G.D., "Fast Fourier Transform Hardware--A Survey", ibid. 109-119.

7. Bergland, G.D., Wilson, D.E., "An FFT Algorithm for a Global Highly Parallel Processor", IEEE Trans. Aud. and Electroac. AU-17:2, June 1969, 125-127.

8. Bergland, G.D., "A Parallel Implementation of the Fast Fourier Algorithm", IEEE Trans. Comput. C-21:4, April 1972, 366-370.

9. Bloomfield, P., Fourier Analysis of Time Series: An Introduction, John Wiley & Sons, New York, 1976.

10. Bongiovanni, G., Corsini, P. and Frosini, G., "One-dimensional and Two-dimensional Generalized Discrete Fourier Transforms", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-24:1, Feb. 1976, 97-99.

11. Bracewell, Ronald N., The Fourier Transform and Its Applications, McGraw-Hill, New York, 1978.

12. Brigham, E.O., The Fast Fourier Transform, Prentice-Hall, New Jersey, 1974.

13. Budinger, T.F., "Multidimensional Space and Time Signal Processing in Biology and Medicine", Proc. IEEE Int. Symposium on Circuits and Systems, Houston, April 28-30, 1980, (in press).

14. Buijs, H.L., Pomerleau, A., et al., "Implementation of a Fast Fourier Transform for Image Processing Applications", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-22:6, Dec. 1974, 420-424.

15. Chan, O.W.C. and Jury, E.F., "Roundoff Error in Multidimensional Generalized Discrete Transforms", IEEE Trans. Circ. and Sys. CAS-21:1, Jan. 1974, 100-108.

16. Cochran, W.T., Cooley, J.W., et al., "What Is the Fast Fourier Transform?", Proc. IEEE 55:10, Oct. 1967, 1664-1674.

17. Cohen, D., "Simplified Control of FFT Hardware", IEEE Trans. Acous., Speech and Sig. Proc. AASP-24:6, Dec. 1976, 577-579.

18. Cooley, J.W. and Tukey, J.W., "An Algorithm for the Machine Calculation of Complex Fourier Series", J. Math Comput. 19, April 1965, 297-301.

19. Cooley, J.W., Lewis, P.A.W. and Welch, P.D., "Historical Notes on the Fast Fourier Transform", Proc. IEEE 55:10, Oct. 1967, 1675-1677.

20. Corinthios, M.J., "The Design of a Class of Fast Fourier Transform Computers, IEEE Trans. Comput. C-20:6, June 1971, 617-623.

21. Corinthios, M.J., "A Fast Fourier Transform for High Speed Signal Processing", IEEE Trans. Comput. C-20:8, Aug. 1971, 843-844.

22. Corinthios, M.J., Smith, K.C. and Yin, J.L., "A Parallel Radix-4 Fast Fourier Transform Computer", IEEE Trans. Comput. C-24:1, Jan. 1975, 80-92.

23. Cyre, W.R. and Lipovski, G.J., "On Generating Multipliers for a Cellular Fast Fourier TRansform Processor", IEEE Trans. Comput. C-21, Jan. 1972, 83-87.

24. Dere, W.Y. and Sakrison, D.J., "Berkeley Array Processor", IEEE Trans. Comput. C-19:5, May 1979, 444-447.

25. Derenzo, S.E., Budinger, T.F., Cahoon, J.L., Greenberg, W.L., Huesman, R.H. and Vuletich, T., "The Donner 280-Crystal High Resolution Positron Tomograph", IEEE Trans. Nucl. Sci. NS-26:2, 1979, 2790-2793.

26. Despain, A.M., "Fourier Transform Computers Using CORDIC Iterations", IEEE Trans. Comput. C-23:10, Oct. 1974, 993-1001.

27. Despain, A.M., " Very Fast Fourier Transform Algorithms for Hardware Implementation", IEEE Trans. Comput. C-28:5, May 1979, 333-341.

28. Dunnigan, G.J. and Llewellyn, R.E., "FFT Implementation for Efficient Calculation of Overlapped Spectra", Proc. IEEE Electronis and Aerospace Systems Convention (EASCON '74), Washington, D.C., Oct. 7-9, 1974, 363-368.

29. Eklundh, J.O., "A Fast Computer Method for Matrix Transposing", IEEE Trans. Comput. C-21:7, July 1972, 801-803.

30. Fino, B.J. and Algaxi, V.R., "Parallel and Pipeline Computation of Fast Unitary Transforms", Electron Lett. 11:5, March 1975, 93-94.

31. Flores, S., The Logic of Computer Arithmetic, Prentice-Hall, New Jersey, 1963.

32. Gentlemen, W.M. and Sande, G., "Fast Fourier Transform for Fun and Profit", AFIPS Proceedings, 1966. Fall Joint Computer Conference, Vol. 29, 563-678.

33. Gibbs, J.E., "Instant Fourier Transform", Electron. Lett. 13:5, March 3, 1977, 122-123.

34. Gold, B. and Bially, T., "Parallelism in FFT Hardware", IEEE Trans. Aud. and Electroac. AU-21:1, Feb. 1973, 5-16.

35. Gonzalez, R.C. and Wintz, P., Digital Image Processing, Addison Wesley, Reading, Mass., 1977.

36. Gottlieb, P. and DeLorenzo, L.J., "Parallel Data Streams and Serial Arithmetic for FFT Processors", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-22:2, April 1974, 111-117.

37. Gorginsky, H.L. and Works, G., "A Pipeline FFT", IEEE Trans. Comput. C-19:11, Nov. 1970, 1015-1019.

38. Gullberg, G.T., "The Attenuated Radon Transform: Theory and Application in Medicine and Biology", Ph.D. Thesis, University of California, Berkeley, Lawrence Berkeley Laboratory LBL-7486, June 1979.

39. Hamming, R.W., Digital Filters, Prentice Hall, Englewood Cliffs, New Jersey, 1977.

40. Huesman, R.H. and Cahoon, J.L., "Data Acquisition, Reconstruction and Display for the Donner 280-Crystal Positron Tomograph", presented at the IEEE Nuclear Science Symposium, San Francisco, California, October 17-18, 1979 (in press).

41. James, D., "Quantization Errors in the Fast Fourier Transform", IEEE Trans. Acous., Speech and Sig. Proc. AASP-23:3, June 1975, 277-283.

42. Kahaner, D.K., "Matrix Description of the Fast Fourier Transform", IEEE Trans. Aud. and Electroac. AU-18:4, Dec. 1970, 442-450.

43. Klahn, R., Shwety, R.R., et al., "The Time-saver: FFT Hardware", Electronics, June 24, 1968, 92-97.

44. Liu, B. and Kaneko, T., "Roundoff Error in Fast Fourier Transforms", Proc. IEEE 63:6, June 1975, 991-992.

45. Liu, B. and Peled, A., "A New Hardware Realiztion of High-Speed Fast Fourier Transformers", IEEE Trans. Acous., Speech and Sig. Proc., ASSP-23:6, Dec. 1975, 543-547.

46. Martinson, L.W. and Smith, R.J., "Digital Matched Filtering with Pipelined Floating Point Fast Fourier Transforms", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-23:2, April 1975, 222-234.

47. Pease, M.C., "The Direct Product and Kronecker Sum", Ch. XIV. Methods of Matrix Algebra, Academic Press, New York, 1965.

48. Pease, M.C., "An Adaption of the Fast Fourier Transform for Parallel Processing", JACM 15:2, April 1968, 252-264.

49. Pease, M.C., "Organization of Large Scale Fourier Processors", JACM 16:3, July 1969, 474-482.

50. Peled, A. and Liu, B., "A New Hardware Realization of Digital Filters, IEEE Trans. Acous., Speech and Sig. Proc. ASSP-22:6, Dec. 1974, 456-462.

51. Peled, A. and Liu, B., "Some New Realizations of Dedicated Hardware Digital Signal Processors", Proc. IEEE Electronics and Aerospace Convention (EASCON '74), Washington D.C., Oct. 7-9, 1974, 464-468.

52. Peled, A., "On the Hardware Implementation of Digital Signal Processors", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-24:1, Feb. 1976, 76-86.

53. Peled, A. and Liu, B., Digital Signal Processing: Theory, Design and Implementation, Wiley & Sons, New York, 1976.

54. Pomerleau, A., Buijs, H.L., and Fournier, M., "A Two-Pass Fixed Point Fast Fourier Transform Error Analysis", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-25:6, 582.

55. Pratt, W., Digital Image Processing, John Wiley & Sons, New York, 1978.

56. Prescott, J., "An Improved Fast Fourier Transform", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-23:3, June 1976, 226-227.

57. Rader, C.M. and Brenner, N.M., "A New Principle for Fast Fourier Transformation", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-24:3, June 1976, 264-266.

58. Rader, C.M. and Brenner, N.M., "Application of the Rader-Brenner FFT Algorithm to Number Theoretic Transforms", IEEE Trans. Acous., Speech and Sig. Proc., ASSP-25:2, April 1977, 196-198.

59. Rosenfeld, A. and Kak, A.C., Digital Picture Processing, Academic Press, New York, 1976.

60. Rivard, G.E., "Direct Fast Fourier Transform of Bivariate Functions", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-25:3, June 1977, 250-252.

61. Shanks, J.L. and Cairns, T.W., "Use of a Digital Convolution Device to Perform Recursive Filtering and the Cooley-Tukey Algorithm", IEEE Trans. Comput. C-17:10, Oct. 1968, 943-949.

62. Shively, R.R., "A Digital P-ocessor to Generate Spectra in Real Time", IEEE Trans. Comput. C-17, May 1968, 485-491.

63. Silverman, H.F., "An Introduction to Programming the Winograd Fourier Transform Algorithm (WFTA)", IEEE Trans. Acous., Speech and Sig. Proc., ASSP-25:2, April 1977, 152-165.

64. Sloate, H., "Matrix Representations for Sorting and the Fast Fourier Transform", IEEE Trans. Circ. and Sys. CAS-21:1, Jan. 1974, 109-116.

65. Stein, M.L. and Munr, W.B., Introduction to Machine Arithmetic, Addison-Wesley, New York, 1971.

66. Stone, H.S., "Parallel Processing with the Perfect Shuffle", IEEE Trans. Comput. C-20:2, Feb. 1971, 153-161.

67. Sundaramurthy, M., and Umpathireddy, V., "Some Results in Fixed-Point Fast Fourier Transform Error Analysis", IEEE Trans. Comput. C-26:2, March 1977, 305-308.

68. Tsui, E.T. and Budinger, T.F., "A Stochastic Filter for Transverse Section Reconstruction", IEEE Tran. Nucl. Sci. NS-26:2, 1979, 2687-2690.

69. Thong, T. and Liu, B., "Fixed Point Fast Fourier Transform Error Analysis", IEEE Trans. Acous., Speech and Sig. Proc. ASSP-24:6, Dec. 1976, 563-573.

70. Veenkant, R.L., "A Serial Minded FFT", IEEE Trans. Aud. and Electroac., AU-20:5, August 1972, 180-185.

71. Volder, J.E., "The CORDIC Trigonometric Computing Technique", IRE Tran. Elect. Comput. ED-8, Sept. 1959, 330-334.

72. Welch, P.D., "A Fixed Point Fast Fourier Transform Error Analysis", IEEE Trans. Aud. and Electroac. AU-17:2, June 1969, 151-157.

73. Welchel, J.E. and Gunn, D.F., "FFT Organizations for High-Speed Digital Filtering", IEEE Trans. Aud. and Electroac. AU-18:2, June 1970, 159-168.

74. Winograd, S., "On Computing the Discrete Fourier Transform", Proc. Nat. Acad. Sci. USA 73:4, April 1976, 1005-1006.

75. Yuen, C.K., "On the Twiddling Factors", IEEE Trans. Comput. C-22:5, May 1973, 544-545.

76. Zohar, S., "Fast Hardware Fourier Transformation Through Counting", IEEE Trans. Comput. C-22:5, May 1973, 433-441.

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720